A

Tratamento de Exceções no Desenvolvimento de Sistemas Tolerantes a Falhas Baseados em Componentes

Este exemplar corresponde à redação final da Tese devidamente corrigida e defendida por Fernando J. Castor de Lima Filho e aprovada pela Banca Examinadora.

Campinas, 30 de novembro de 2006.

Cecília Mary Fischer Rubira

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: Miriam Cristina Alves – CRB8a / 5094

Lima Filho, Fernando José Castor de.
L628t Tratamento de exceções no desenvolvimento de sistemas tolerantes a falhas baseados em componentes/Fernando José Castor de Lima Filho -- Campinas, [S.P. :s.n.], 2006.

Orientadora : Cecília Mary Fischer Rubira

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

1. Tolerância a falhas (Computação). 2. Componentes de software. 3. Software - Arquitetura. I. Rubira, Cecília Mary Fischer. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Título em inglês: Exception handling in the development of fault-tolerant component-based systems

Palavras-chave em inglês (Keywords): 1. Fault tolerance (Computing). 2. Software components. 3. Software -- Architecture.

Área de concentração: Engenharia de software

Titulação: Doutor em Ciência da Computação

Banca examinadora: Profa. Dra. Cecília Mary Fischer Rubira (IC-Unicamp)
Prof. Dr. Paulo Henrique Monteiro Borba (CIn-UFPE)
Prof. Dr. Avelino Francisco Zorzo (FACIN-PUCRS)
Profa. Dra. Eliane Martins (IC-Unicamp)
Prof. Dr. Luiz Eduardo Buzato (IC-Unicamp)

Data da defesa: 30/11/2006

Programa de Pós-Graduação: Doutorado em Ciência da Computação

TERMO DE APROVAÇÃO

Tese defendida e aprovada em 30 de novembro de 2006, pela Banca examinadora composta pelos Professores Doutores:

[illegible]

Prof. Dr. Paulo Henrique Monteiro Borba
Cin - UFPE

[illegible]

Prof. Dr. Avelino Francisco Zorzo
FACIN - PUCRS

Eliane Martins

Profa. Dra. Eliane Martins
IC - UNICAMP

[illegible]

Prof. Dr. Luiz Eduardo Buzato
IC - UNICAMP

Cecília Mary Fischer Rubira

Profa. Dra. Cecília Mary Fischer Rubira
IC - UNICAMP

200734399

Instituto de Computação
Universidade Estadual de Campinas

Tratamento de Exceções no Desenvolvimento de Sistemas Tolerantes a Falhas Baseados em Componentes

Fernando J. Castor de Lima Filho [1]

Novembro de 2006

Banca Examinadora:

- Cecília Mary Fischer Rubira (Orientadora)
- Paulo Borba
  Centro de Informática - UFPE
- Avelino Zorzo
  Faculdade de Informática - PUC-RS
- Eliane Martins
  Instituto de Computação - UNICAMP
- Luiz Eduardo Buzato
  Instituto de Computação - UNICAMP
- Ana Cristina Melo (Suplente)
  Instituo de Matemática e Estatística - USP
- Ariadne Rizzoni Carvalho (Suplente)
  Instituto de Computação - UNICAMP

[1] **Bolsista da FAPESP, processo 02/13996-2.**

Resumo

Mecanismos de tratamento de exceções foram concebidos com o intuito de facilitar o gerenciamento da complexidade de sistemas de software tolerantes a falhas. Eles promovem uma separação textual explícita entre o código normal e o código que lida com situações anormais, a fim de dar suporte à construção de programas que são mais concisos, fáceis de evoluir e confiáveis. Diversas linguagens de programação modernas e a maioria dos modelos de componentes implementam mecanismos de tratamento de exceções.

Apesar de seus muitos benefícios, tratamento de exceções pode ser a fonte de diversas falhas de projeto se usado de maneira indisciplinada. Estudos recentes mostram que desenvolvedores de sistemas de grande escala baseados em infra-estruturas de componentes têm hábitos, no tocante ao uso de tratamento de exceções, que tornam suas aplicações vulneráveis a falhas e difíceis de se manter. Componentes de software criam novos desafios com os quais mecanismos de tratamento de exceções tradicionais não lidam, o que aumenta a probabilidade de que problemas ocorram. Alguns exemplos são indisponibilidade de código fonte e incompatibilidades arquiteturais.

Neste trabalho propomos duas técnicas complementares centradas em tratamento de exceções para a construção de sistemas tolerantes a falhas baseados em componentes. Ambas têm ênfase na estrutura do sistema como um meio para se reduzir o impacto de mecanismos de tolerância a falhas em sua complexidade total e o número de falhas de projeto decorrentes dessa complexidade. A primeira é uma abordagem para o projeto arquitetural dos mecanismos de recuperação de erros de um sistema. Ela trata do problema de verificar se uma arquitetura de software satisfaz certas propriedades relativas ao fluxo de exceções entre componentes arquiteturais, por exemplo, se todas as exceções lançadas no nível arquitetural são tratadas. A abordagem proposta lança mão de diversas ferramentas existentes para automatizar ao máximo esse processo. A segunda consiste em aplicar programação orientada a aspectos (AOP) a fim de melhorar a modularização de código de tratamento de exceções. Conduzimos um estudo aprofundado com o objetivo de melhorar o entendimento geral sobre o efeitos de AOP no código de tratamento de exceções e identificar as situações onde seu uso é vantajoso e onde não é.

Abstract

Exception handling mechanisms were conceived as a means to help managing the complexity of fault-tolerant software. They promote an explicit textual separation between normal code and the code that deals with abnormal situations, in order to support the construction of programs that are more concise, evolvable, and reliable. Several mainstream programming languages and most of the existing component models implement exception handling mechanisms.

In spite of its many benefits, exception handling can be a source of many design faults if used in an ad hoc fashion. Recent studies show that developers of large-scale software systems based on component infrastructures have habits concerning the use of exception handling that make applications vulnerable to faults and hard to maintain. Software components introduce new challenges which are not addressed by traditional exception handling mechanisms and increase the chances of problems occurring. Examples include unavailability of source code and architectural mismatches.

In this work, we propose two complementary techniques centered on exception handling for the construction of fault-tolerant component-based systems. Both of them emphasize system structure as a means to reduce the impact of fault tolerance mechanisms on the overall complexity of a software system and the number of design faults that stem from complexity. The first one is an approach for the architectural design of a system's error handling capabilities. It addresses the problem of verifying whether a software architecture satisfies certain properties of interest pertaining the flow of exceptions between architectural components, e.g., if all the exceptions signaled at the architectural level are eventually handled. The proposed approach is based on a set of existing tools that automate this process as much as possible. The second one consists in applying aspect-oriented programming (AOP) to better modularize exception handling code. We have conducted a through study aimed at improving our understanding of the effects of AOP on exception handling code and identifying the situations where its use is advantageous and the ones where it is not.

Agradecimentos

Normalmente, uma tese é o resultado do sangue, do suor, das lágrimas e dos neurônios de uma quantidade de gente muito maior do que caberia na maior das listas de agradecimentos. Esta não é exceção à regra e eu gostaria de expressar meus mui sinceros agradecimentos a todas as pessoas que contribuíram de alguma forma, mínima que seja, para a sua realização. Em especial, eu gostaria de agradecer...

Em primeiro lugar à minha família, sem a qual eu não existiria e graças a quem eu cheguei até aqui: meus pais, Fernando e Maria, minha irmã, Donandréa, minha "mãe preta", Tia Zefinha, meu cunhado, Filipe, meu Avô, Nelson, e o dogo Bara. Amo vocês.

À minha orientadora Cecília, pelos muitos ensinamentos, pelas várias discussões e idéias interessantes, sempre regadas a muito café, pelas diversas oportunidades que me proporcionou de evoluir como pesquisador e indivíduo, por estar lá nos momentos particularmente difíceis e por ser muito legal.

À minha querida esposa, Amália, a quem dedico esta tese e as minhas principais conquistas. Muito obrigado por me fazer tão feliz e por ser a companheira mais maravilhosa com quem alguém poderia sonhar.

Aos meus colegas do Grupo de Engenharia de Software e Confiabilidade do IC-UNICAMP, com quem aprendi bastante e que foram instrumentais para a conclusão deste trabalho: Ana Elisa, Hélder, Leonardo, Leonel, Milton, Moacir, Patrick, Paulo Asterio, Peterson, Raquel, Ricardo, Rodrigo, Tiago, e Vinícius.

À minha querida família aqui em Campinas, Tio Chico, Tia Virgília, Cacá, Má e Gabi, por terem tornado tranquila a mudança de cidade e por sempre terem feito eu me sentir em casa, me acolhendo como um filho/irmão.

Aos inúmeros amigos que fiz ao longo desses anos de UNICAMP: Adilson, Alexandre (Capitão), Borin, Daniel Manzatto, Dentinho, Diogo, Evandro (O Predador), Fernanda, Giovani, Greg, Jackson, Juliana Freitag, Juliana Santi, Lásaro, Leo Rangel, Renato, Rodrigo Minetto, Rodrigo Oliveira, Thaís e Tiago. Se eu esqueci o seu nome é porque sou meio desatento. Por favor, tolere a minha falha.

Aos meus muitos amigos de Recife que, apesar de longe geograficamente, sempre se fizeram muito presentes. Nesse caso é tanta gente que eu prefiro não sair listando porque sei que vou terminar esquecendo alguém.

Aos Profs. Alessandro Garcia, Rogério de Lemos e Alexander Romanovsky, pela oportunidade de imergir um pouco nas culturas de outras universidades, em outro país, e pelas contribuições fundamentais que fizeram para esta tese. Agradeço também a Manuel Fähndrich, pela chance de fazer um estágio na Microsoft Research e pesquisar sobre assuntos que não tinham praticamente nada a ver com a minha tese mas eram muito divertidos. Agradeço também aos alunos de outras universidades com os quais tive o prazer de trabalhar diretamente: Eduardo Magno Figueiredo e Nélio Cacho.

Aos Profs. Paulo Borba, Ana Cavalcanti e Augusto Sampaio, por me apresentar ao trabalho que eu decidi fazer pro resto da vida e por me mostrar que engenharia de software é uma área interessantíssima, apesar do que aquelas matérias de graduação em que a quantidade de documentação é medida em arrobas dizem.

A todos os Profs. do IC-UNICAMP, em especial Resende, Eliane, Buzato, Edmundo e Cortes, e a todos os funcionários do IC-UNICAMP, em especial, Dna. Neuza, Nelson (estuda estuda que a vida não tá fácil não!), Ademilson, Vera, Carlos e Camila.

À FAPESP, processo 02/13996-2, pelo apoio financeiro, sem o qual este trabalho não teria sido possível. Pelo apoio financeiro agradeço também aos projetos CompGov (FINEP 1843/04) e RODIN (IST 511599).

E por último, mas nem um tico menos importante, ao *Rock'n'Roll*, porque todos os dias ele está salvando a minha vida.

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Este capítulo contextualiza a tese, introduz o problema que ela visa tratar, descreve sucintamente as soluções adotadas e apresenta um resumo das contribuições. O capítulo também explica alguns dos conceitos que serão usados ao longo de toda a tese e descreve sua organização.

1.1 Contexto

Nos últimos anos, o desenvolvimento baseado em componentes [170] (DBC) vem sendo usado na construção de grandes sistemas de software. Essa tendência é embasada na idéia de que componentes reusáveis que provêm funcionalidades prontas para usar podem encurtar tempos de desenvolvimento e tempo para comercializar[1] [181]. O princípio fundamental do DBC é que sistemas de software devem ser desenvolvidos a partir da integração de componentes pré-existentes, normalmente construídos por várias organizações. Algumas das principais vantagens do uso componentes de software são (i) facilidade de substituição e extensão [41, 9], (ii) reuso [170] e (iii) aumento na previsibilidade do sistema [9].

Uma implicação direta do DBC no processo de desenvolvimento de software é a divisão de responsabilidades entre a construção de componentes e sua integração para formar sistemas [170]. Por um lado, **construtores de componentes** de software reusáveis não têm conhecimento sobre os diferentes contextos nos quais componentes serão implantados[2] (*design for reuse* [104]). Por outro lado, esses componentes são providos normalmente como 'caixas pretas'. Isto é, os projetos internos desses componentes e seus códigos-fonte muitas vezes não estão disponíveis para **integradores de sistemas** (*design with*

[1]Do inglês: *time-to-market*.
[2]Do inglês: *deployed*.

reuse [104]). Estes últimos têm conhecimento apenas sobre o que é especificado nas interfaces dos componentes, normalmente os serviços que o componente oferece (interfaces providas) e aqueles dos quais depende (interfaces requeridas). Para tornar viável a integração de componentes desenvolvidos independentemente, o processo de desenvolvimento empregado deve ter forte ênfase na arquitetura de software do sistema. De acordo com Clements e Northrop [48], arquitetura de software é a estrutura dos componentes de um programa/sistema, suas relações e princípios e diretrizes que controlam seu projeto e evolução ao longo do tempo.

Nos últimos anos, o uso de componentes de software, que até então era restrito à construção de sistemas de informação, com requisitos moderados de confiabilidade[3], vem se expandindo na direção de áreas de aplicação nas quais o preço de uma falha pode ser altíssimo, por exemplo, sistemas embarcados executados em aviões [124]. De acordo com o Departamento de Transportes dos EUA, o reuso de componentes de software em aplicações com essa característica se deve a questões econômicas e ao avanço que o DBC vem experimentando nos últimos anos [176]. Na construção de sistemas de software com um alto grau de confiabilidade, uma consequência da dicotomia construção/integração é o requisito de que abordagens para o desenvolvimento de tais aplicações garantam essa confiabilidade em dois níveis: (i) para cada componente individual do sistema, de modo que seja capaz de prover serviços de acordo com sua especificação; e (ii) na integração de um conjunto de componentes, de modo que as interações entre esses componentes não produzam resultados inesperados.

Sistemas cujos defeitos podem ameaçar vidas humanas ou resultar em grandes perdas financeiras normalmente são feitos tolerantes a falhas [5], ou seja, capazes de prover o seu serviço, ainda que apenas parcialmente, na presença de falhas. Sistemas tolerantes a falhas são construídos partindo-se do pressuposto de que não é possível desenvolver um sistema que sempre se comporte de acordo com sua especificação [5, 146]. Portanto, esses sistemas apresentam mecanismos para detectar erros em seus estados e se recuperar desses erros. Em um sistema tolerante a falhas, quando um erro é detectado, é preciso levar o estado do sistema para um estado consistente e bem-definido para que o sistema possa continuar sua execução normal.

Técnicas de tolerância a falhas surgem como candidatas naturais para auxiliar no desenvolvimento de sistemas confiáveis baseados em componentes. Primeiro, porque a visibilidade de caixa preta de alguns componentes de software torna difícil garantir que esses componentes apresentam o grau esperado de confiabilidade, apesar de alguns esforços recentes visando reverter essa situação para certos tipos de componente [105]. Segundo, porque mesmo quando essas garantias são fornecidas, normalmente não é possível ter certeza que as interações entre componentes produzidos por diferentes fontes sempre pro-

[3]Do inglês: *dependability*.

duzirão o resultado esperado. Finalmente, ainda que seja possível verificar de maneira automática que interações entre componentes de um sistema sempre seguem um protocolo bem definido, é necessário garantir que o sistema não falha de maneira catastrófica quando um problema ocorre devido a um elemento externo, por exemplo, um canal de comunicação falho ou um dispositivo mecânico defeituoso com o qual o sistema interage. Tendo esses fatores em vista, pode-se afirmar que técnicas de tolerância a falhas complementam outras técnicas para tornar confiável um sistema baseado em componentes, por exemplo, verificação automática [6] e testes [123, 183].

Esta tese foca no desenvolvimento de sistemas tolerantes a falhas baseados em componentes. Nossa abordagem consiste em combinar avanços recentes nas áreas de arquitetura de software [159] e desenvolvimento de software orientado a aspectos [106] com técnicas conhecidas de estruturação de sistemas tolerantes a falhas, mais especificamente, tratamento de exceções [52]. As Seções 1.1.1-1.1.4 descrevem alguns dos conceitos necessários para o entendimento deste trabalho. Nas seções seguintes são apresentados o problema que a tese se propõe a tratar (Seção 1.2), as soluções propostas para esse problema (Seções 1.3 e 1.4), uma lista de contribuições (Seção 1.5) e a organização da tese (Seção 1.6).

1.1.1 Desenvolvimento Baseado em Componentes

A idéia de construir sistemas a partir de partes pré-fabricadas é antiga na indústria. Há décadas que carros, computadores e até mesmo casas são construídos dessa maneira. O conceito de produção de software a partir de componentes pré-fabricados é relativamente recente, porém. Foi mencionado pela primeira vez por Mcllroy[126] em 1969. Em 1996 foi realizado o *First International Workshop on Component-Oriented Programming* (WCOP'96), a partir do qual o interesse pelas técnicas baseadas em componentes vem crescendo e se consolidando, seja sob o título de desenvolvimento baseado em componentes (DBC), ou de engenharia de software baseada em componentes (ESBC). Alguns autores afirmam que o reuso de componentes de software tem potencial para reduzir drasticamente os custos e o tempo necessários para construir uma aplicação [127]. Há diversas definições para ESBC e DBC, como por exemplo:

> *"In CBD, the developer's task is to assemble a (large) software system from (a large number of) reusable components, and perhaps a comparatively small amount of code written for the particular application".* [149].

> *"Component-based software engineering is concerned with the rapid assembly of systems from components where: (i) components and frameworks have certified properties; and (ii) these certified properties provide the basis for predicting the properties of systems built from components."*[9].

Szyperski afirma que um componente, no contexto do desenvolvimento baseado em componentes, deve [169]: (i) presumir que faz parte de uma arquitetura; (ii) apresentar funcionalidades através de interfaces providas; (iii) explicitar suas dependências paramétricas através de interfaces requeridas; (iv) ter dependências estáticas; (v) ser baseado em alguma plataforma de componentes específica; (vi) requerer outros componentes; e (vii) requerer contexto por-instância. O autor afirma ainda que, para tornar possível a composição de componentes, todas as dependências e suposições importantes da implementação de um componente precisam ser capturadas. Essa é uma afirmação importante, já que vai contra a tendência dos modelos de componentes atuais de enfatizar apenas as funcionalidades que componentes provêm.

Apesar das diferenças entre as definições, há um consenso de que no DBC a ênfase está na integração entre componentes e não apenas em sua construção. Espera-se que essa mudança no foco do processo de desenvolvimento torne possível a construção de sistemas melhores, com maior velocidade, maior grau de reuso e gastando-se menos dinheiro.

Para tornar viável a integração entre componentes desenvolvidos independentemente, foram criados os modelos de componentes. Um modelo de componentes especifica padrões que devem ser impostos sobre desenvolvedores de componentes. A conformidade com um modelo de componentes é uma das características que diferencia componentes de outras formas de modularização [9]. Para auxiliar no desenvolvimento baseado em modelos de componentes, foram desenvolvidos diversos *frameworks* de componentes, implementações de serviços que dão suporte a um determinado modelo de componentes e garantem que suas restrições não são violadas. Atualmente, os modelos de componentes mais populares são Enterprise Javabeans [168], da Sun, DCOM [131], da Microsoft, e o CORBA Component Model [139], da OMG.

1.1.2 Tolerância a Falhas e Tratamento de Exceções

Sistemas de *hardware* normalmente incluem mecanismos para detecção e correção de falhas. Alguns exemplos desses mecanismos são o uso de códigos de redundância cíclica (CRC) e de replicação de dispositivos (como em discos rígidos RAID). Para aplicações que envolvem risco para vidas humanas (como aplicações médicas) ou risco de grandes perdas financeiras (como na pesquisa espacial), chamadas **críticas**, é essencial garantir que o serviço pelo qual a aplicação é responsável continuará sendo provido, ainda que apenas parcialmente, em situações nas quais falhas ocorram. Mesmo para sistemas que não são de missão crítica, diversas razões motivam o uso de mecanismos que os tornem capazes de prover serviço na presença de falhas[8]. Definimos um sistema como *tolerante a falhas* se este é capaz de prover serviço de acordo com sua especificação, mesmo na presença de falhas[5].

Embora os termos “falha”, “erro” e “defeito” sejam usados no dia-a-dia como sinônimos, na terminologia de tolerância a falhas essas palavras têm significados distintos e bem definidos. Um **defeito** ocorre quando um sistema deixa de prover um determinado serviço ou o faz em desacordo com a sua especificação. Defeitos são sempre observáveis externamente ao sistema. Um defeito é causado por um **erro**; uma inconsistência no estado interno do sistema. A ocorrência de erros pode ter como conseqüência a ocorrência de defeitos. Uma **falha** corresponde a um evento ou seqüência de eventos que propicia o surgimento de um erro num componente ou no sistema como um todo. Um erro é a manifestação de uma falha no sistema.

De acordo com Anderson e Lee [5], a tolerância a falhas se divide em quatro fases que devem formar a base do projeto e implementação de qualquer sistema tolerante a falhas. Essas fases são (i) detecção de erros, (ii) confinamento e avaliação de danos, (iii) recuperação de erros e (iv) tratamento de falhas e continuação de serviço.

Detecção de erros. Para que um sistema seja capaz de tolerar uma falha, seus efeitos precisam primeiro ser detectados. Uma falha se manifesta em um sistema através de um erro, uma inconsistência em seu estado interno. Embora falhas não possam ser detectadas diretamente [5], erros podem. Consequentemente todo sistema tolerante a falhas deve ser capaz de detectar erros que possam vir a se manifestar em seu estado. Um exemplo de técnica bastante popular para detectar erros é o uso de invariantes, predicados que devem ser verdade durante toda a execução do sistema. Se, em algum momento da execução do sistema, um desses invariantes não for satisfeito, um erro ocorreu.

Confinamento e avaliação de danos. Depois que um erro é detectado, é possível que já tenha afetado uma parte bem maior do estado do sistema do que se suspeita, devido ao intervalo de tempo entre o momento em que o erro ocorre e o momento em que é detectado. Consequentemente, quando um erro é detectado, é necessário avaliar a extensão dos danos causados. Sistemas tolerantes a falhas devem ser desenvolvidos de modo que erros possam ser confinados a regiões bem-definidas, quando ocorrem, facilitando assim a avaliação de danos.

Recuperação de erros. Quando um erro é detectado, é preciso levar o estado do sistema para um estado consistente e bem-definido para que o sistema possa continuar sua execução normal. Técnicas para recuperação de erros se dividem em duas grandes categorias que diferem na abrangência e na maneira como corrigem o estado do sistema: recuperação de erros por retrocesso[4] e recuperação de erros por avanço [5].

---

[4]Do inglês: *backward error recovery*.

[5]Do inglês: *forward error recovery*.

Técnicas de recuperação de erros por retrocesso restauram o estado do sistema para algum estado anterior na esperança de que este seja consistente. Essas técnicas são [5]: independentes de avaliação de danos; (ii) capazes de prover recuperação de falhas arbitrárias; (iii) um conceito geral, aplicável a todos os sistemas; e (iv) facilmente providas como um mecanismo. Mecanismos de pontos de recuperação[6] são um exemplo típico de técnica de recuperação por retrocesso.

Transações atômicas são um exemplo bastante popular de técnica de recuperação de erros por retrocesso.

Técnicas de recuperação de erros por avanço partem do princípio de que, quando um erro é detectado, o sistema deve ser levado para um novo estado garantidamente consistente. Recuperação de erros por avanço permite que políticas de recuperação eficientes e altamente especializadas sejam implementadas. Entretanto, essas técnicas requerem uma quantidade maior de informação contextual para que possam ser empregadas. A recuperação de erros por avanço é [5]: (i) dependente da avaliação de danos; (ii) inapropriada para recuperar o sistema de falhas não-antecipadas; (iii) projetada especificamente para um sistema particular; e (iv) impossível de implementar como um mecanismo genérico. Recuperação de erros por avanço normalmente é introduzida em sistemas de software através de tratamento de exceções. Esta técnica é discutida com mais detalhes na Seção 1.1.2.

Tratamento de falhas e continuação de serviço. Em algumas situações, levar o sistema para um estado consistente não é o suficiente para garantir que este funcionará de forma correta. Por exemplo, se um erro ocorre em decorrência de uma falha de projeto, corrigir o erro não resolve o problema. Como a falha que o produziu é permanente, esse erro voltará sistematicamente a acontecer. Neste caso, a única maneira de garantir que o sistema funcionará corretamente é remover a própria falha. Um problema no tratamento de falhas é que a detecção de um erro não necessariamente serve para identificar a falha. Conforme dito anteriormente, um erro pode ter um efeito não-local sobre o estado do sistema e relacioná-lo com a ocorrência de uma falha pode ser muito difícil. Partindo do pressuposto de que a falha foi identificada corretamente, técnicas para tratamento de falhas normalmente se baseiam na idéia de substituir o componente problemático por uma variante correta.

Tratamento de Exceções

Quando um programa apresenta desvios com relação à sua especificação devido a uma falha, é possível que erros sejam introduzidos em seu estado. Quando um erro é detec-

[6]Do inglês: *checkpointing.*

tado, é necessário sinalizar sua ocorrência para que o sistema, caso inclua mecanismos de tolerância a falhas, seja capaz de corrigi-lo. Como espera-se que essa situação ocorra raramente, o sinal que indica a ocorrência de um erro é tradicionalmente chamado de **exceção**. Para que o sistema possa continuar a se comportar de acordo com sua especificação, mediante o lançamento de uma exceção, é necessário tratá-la. A parte de um sistema que implementa seu comportamento na ausência de falhas, conforme definido por sua especificação, é chamada de **comportamento normal**. Similarmente, a parte do sistema que implementa seus mecanismos de recuperação de erros, responsáveis por torná-lo tolerante a falhas, é chamada de **comportamento excepcional**, ou **anormal**. Tratamento de exceções [80] é uma técnica para estruturar o comportamento excepcional de um programa, de modo que erros possam ser detectados, sinalizados e tratados.

Diversas linguagens de programação populares dão suporte a tratamento de exceções, através da definição e implementação de sistemas de tratamento exceções[7]. Essas linguagens permitem que desenvolvedores definam exceções e os tratadores correspondentes. Quando um erro é detectado, uma exceção é sinalizada e o tratador adequado é procurado e acionado de maneira automática. O conjunto de tratadores de exceções de um programa define a seu comportamento excepcional. Se uma mesma exceção pode ser sinalizada em diferentes partes de um programa, é possível que tratadores diferentes sejam executados. A escolha do tratador que será acionado depende do contexto de tratamento de exceções em que a exceção é lançada. Um contexto de tratamento é uma região de um programa dentro da qual as mesmas exceções são tratadas da mesma maneira. Cada contexto tem um conjunto de tratadores associados que são acionados quando as exceções correspondentes são sinalizadas. Exemplos típicos de contextos de tratamento de exceções em linguagens orientadas a objetos são comandos, blocos de código, métodos, classes e objetos.

O conceito de **componente tolerante a falhas ideal** (CTFI) [5] define um *framework* conceitual para estruturar tratamento de exceções em sistemas de software. Um CTFI é um componente[8] no qual as partes responsáveis pelo comportamento normal e pelo comportamento excepcional estão separadas e bem-definidas, dentro da sua estrutura interna, permitindo que erros sejam detectados e tratados com maior facilidade. O objetivo da abordagem de CTFI é proporcionar uma forma de estruturar sistemas que minimize o impacto dos mecanismos de tolerância a falhas em sua complexidade global.

A Figura 1.1 apresenta a estrutura interna de um componente tolerante a falhas ideal e os tipos de mensagem que ele troca com outros componentes na arquitetura. Ao receber uma requisição de serviço, um CTFI fornece uma *resposta normal* se a requisição é

[7]Do inglês: *exception handling systems*.

[8]No sentido mais amplo; um objeto, um subsistema, um componente de software ou até mesmo um sistema inteiro.

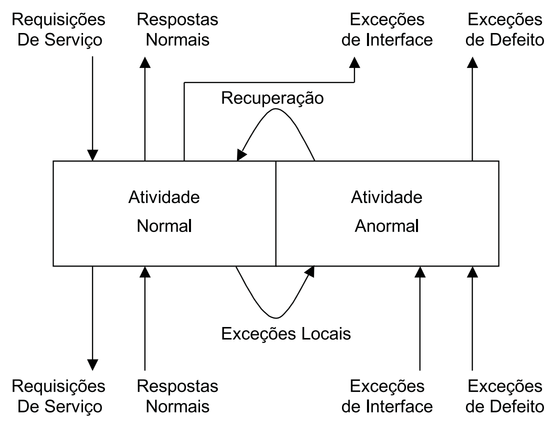


Figura 1.1: Componente tolerante a falhas ideal.

processada com sucesso. Se a requisição de serviço não é válida, é sinalizada uma *exceção de interface*. Se o serviço está disponível mas ocorre uma falha durante o processamento da requisição e o CTFI não é capaz de tratá-la internamente, é levantada uma *exceção de defeito*.

1.1.3 Arquitetura de Software

Assim como acontece com o desenvolvimento de software baseado em componentes, não existe uma definição universalmente aceita para arquitetura de software [11, 143]. Há um conjunto de características, porém, que é mencionado na grande maioria das definições existentes na literatura. Essas características são sumarizadas pela definição adotada por Clements e Northrop [48]:

> *"(Software Architecture is) the structure of the components of a program/system, their interrelationships and principles and guidelines governing their design and evolution over time."*

A arquitetura de um sistema de software reflete o conjunto de decisões que devem ser tomadas primeiro no projeto desse sistema; aquelas que são mais difíceis de mudar em etapas posteriores do desenvolvimento. A arquitetura de software envolve a estrutura e organização através das quais componentes e subsistemas interagem para formar sistemas e as propriedades de sistemas que podem ser melhor projetadas e analisadas sistemicamente [110]. É largamente aceito que a arquitetura de um sistema de software tem um forte impacto em sua capacidade de satisfazer seus requisitos de qualidade, como segurança, disponibilidade e desempenho, entre outros [11, 47, 165].

Uma arquitetura de software normalmente é representada e documentada através de várias perspectivas, ou **visões** [109]. Cada visão descreve um aspecto diferente da arquitetura do sistema, similarmente aos diversos tipos de plantas que são usados para representar os vários aspectos de uma construção. Alguns exemplos clássicos [109] de visões arquiteturais são: (i) a visão de processos, que lida com concorrência e distribuição e representa um conjunto de unidades de computação, potencialmente distribuídas, que se comunicam através de um barramento ou uma rede; e (ii) a visão lógica, que decompõe o sistema em um conjunto de abstrações e indica os serviços que essas abstrações provêm e requerem e como elas se relacionam. Uma visão arquitetural mostra como a arquitetura do sistema atinge um determinado atributo de qualidade [11]. Por exemplo, a visão de camadas indica o quão portável um sistema é, enquanto a visão de implantação pode ser usada para se analisar a performance e a confiabilidade do sistema [46].

As linguagens para descrição de arquiteturas, ou ADLs[9], surgiram devido à necessidade de representar arquiteturas de uma maneira mais formal. ACME[76] e Wright[4] são exemplos de ADLs. Embora exista uma diversidade considerável no que concerne às capacidades das diferentes ADLs existentes, todas compartilham de uma mesma base conceitual da qual os principais elementos são *componentes*, *conectores* e configurações [128]. *Componentes* são elementos arquiteturais responsáveis por realizar computações. Como é esperado que as interações entre componentes sejam complexas, essas interações são materializadas explicitamente através de *conectores*. Um conjunto de componentes interligados através de conectores caracteriza uma *configuração*.

A arquitetura de sistemas de software não-triviais normalmente envolvem diversos estilos arquiteturais. Um estilo arquitetural define um vocabulário de tipos de elementos de projeto que fazem parte de uma família de arquiteturas e as regras pelas quais esse elementos são compostos [76]. Alguns exemplos são o baseado em camadas, cliente-servidor e *pipes and filters* [158].

1.1.4 Programação Orientada a Aspectos

A programação orientada a aspectos (AOP) [106] foi proposta na segunda metade dos anos 90 como uma maneira de modularizar interesses transversais[10], interesses cuja implementação está espalhada em diversas partes do código do sistema, misturada a código responsável por outros requisitos. Exemplos de tais interesses são *logging* e autenticação. AOP é baseada na idéia de que sistemas computacionais são melhor programados se especificarmos separadamente os seus vários interesses e alguma descrição de seus relacionamentos e usarmos os mecanismos providos por AOP para combiná-los em um programa

[9]Do inglês: *architecture description languages.*

[10]Do inglês: *crosscutting concerns.*

coerente [63].

Um elemento importante da programação orientada a aspectos é a dicotomia entre linguagem base, ou de componentes, e linguagem de aspectos [40]. Essa dicotomia se baseia nos seguintes princípios: (i) sistemas são decompostos em aspectos e componentes; (ii) aspectos modularizam interesses *crosscutting*; (iii) componentes modularizam interesses não-*crosscutting*; e (iv) aspectos devem ser explicitamente representados a parte de componentes e outros aspectos. Alguns exemplos de linguagens orientadas a aspectos são AspectJ [114] e Eos [144]. A primeira usa Java como linguagem base, enquanto a segunda usa C#. Programas nas linguagens base e de aspectos são combinados através de um processo chamado *weaving*. A ferramenta responsável por essa tarefa é chamada de *weaver*.

As duas propriedades fundamentais que tornam AOP tão útil e diferenciam esse paradigma de todos os que vieram antes são quantificação[11] e transparência[12] [67]. Quantificação é a capacidade de escrever comandos unitários que são capazes de afetar diversas partes de um programa. Quantificação torna possível expressar comandos do tipo "nos programas P, sempre que a condição C for verdadeira, execute a ação A". Transparência, no contexto da programação orientada a aspectos, quer dizer que programas escritos na linguagem base não precisam ser preparados especificamente para serem combinados com aspectos; podem ser escritos como se aspectos simplesmente não existissem. Alguns autores afirmam que transparência é um termo muito forte e que há benefícios em organizar programas na linguage base de tal maneira que combiná-los com os aspectos sejam mais fácil [136]. Por isso, muitos autores preferem usar o termo não-invasão[13], ao invés de transparência. É importante frisar que, independentemente do termo empregado, é fundamental a idéia de que programas na linguagem base não devem fazer referência explícita a aspectos.

Diversos trabalhos existentes na literatura mostram que AOP é útil para modularizar vários requisitos comuns no desenvolvimento de sistemas de software. Alguns exemplos são distribuição [37, 164], persistência [114, 164], autenticação [114], tratamento de exceções [24, 117] e implementação de padrões de projeto [19, 74, 91]. O impacto da programação orientada a aspectos tanto na academia quanto na indústria nos últimos anos é tão grande que a revista Technology Review de janeiro de 2001 incluiu a programação orientada a aspectos em uma lista de "dez tecnologias emergentes que mudarão o mundo" [147].

[11]Do inglês: *quantification*.
[12]Do inglês: *obliviousness*.
[13]Do inglês: *non-invasiveness*.

1.2 Problema

Desde os primórdios da computação tolerante a falhas, em geral, e da tolerância a falhas em sistemas de software, em particular, é sabido que a principal causa de falhas de projeto residuais, erros de programação popularmente conhecidos como *bugs*, em sistemas de software é a complexidade desses sistemas [146]. Na área de sistemas de software críticos, há um consenso de que sistemas mais simples tendem a ser mais seguros[14] [116, 94]. O uso de técnicas adequadas de estruturação é crucial para a construção de sistemas de software tolerantes a falhas [146]. Caso contrário, a complexidade adicional que mecanismos de tolerância a falhas introduzem nos sistemas propiciará a aparição de um número ainda mais acentuado de falhas de projeto.

Mecanismos de tratamento de exceções [52, 80] foram concebidos com o intuito de se gerenciar a complexidade de software tolerante a falhas. Esses mecanismos visam separar o código "normal", responsável pelas funcionalidades da aplicação, e o código que torna o sistema tolerante a falhas, a fim de dar suporte à construção de programas mais confiáveis, concisos e fáceis de evoluir [142]. Tratamento de exceções é uma técnica para a implementação de recuperação de erros por avanço (Seção 1.1). Consequentemente, complementa outras abordagens para tornar sistemas mais confiáveis, como transações atômicas [83]. O uso de tratamento de exceções na construção de diversos sistemas de grande escala e o fato de diversas linguagens de programação orientadas a objetos modernas, como Java [82], Ada [171], C# [95] e C++ [108], e modelos de componentes, como DCOM [131], CCM [139] e EJB [168], implementarem tratamento de exceções atestam sua importância para a prática atual do desenvolvimento de software.

Frequentemente, uma parte considerável do código de um sistema é dedicada à detecção e ao tratamento de erros. Em 1989, Cristian [52] afirmava que frequentemente essa parcela do código correspondia a mais de dois terços do código total de uma aplicação. Um estudo mais recente [182] envolvendo um conjunto de aplicações de código aberto escritas em Java detectou que entre 1 e 5% dos textos dos programas consistia de tratadores de exceções (blocos `catch` em Java) e ações de limpeza (blocos `finally`). Em outro estudo [148] que levou em consideração 5 aplicações de grande escala baseadas (com centas de milhares ou mesmo milhões de linhas de código) na plataforma Java Enterprise Edition [13], a razão entre o número de tratadores de exceções e o número de operações em cada aplicação variou entre 0,058 e 1,79. Em um estudo realizado [24] por mim e alguns colegas (apresentado no Capítulo 6) envolvendo quatro aplicações, duas produzidas na indústria e duas na academia, foi detectado que entre 9,9% e 20,8% das operações incluem código de tratamento de exceções.

Esses números mostram que tratamento de exceções é uma técnica amplamente di-

[14]Do inglês: *safe*.

fundida no desenvolvimento de aplicações reais. Apesar disso, desenvolvedores costumam focar no comportamento normal das aplicações e só lidar com o código responsável por detectar e tratar erros na etapa de implementação [52, 148], de maneira *ad hoc*. Essa prática de projeto e implementação cria uma situação propícia para o aparecimento de falhas de projeto. O código responsável por detectar e tratar erros costuma estar espalhado em diversos módulos de um sistema, misturado ao código responsável pelo comportamento normal, o que dificulta sua estruturação, entendimento e manutenção [117]. Adicionalmente, mecanismos de tratamento de exceções, quando usados de maneira descuidada, podem ter um efeito no comportamento do programa similar ao do comando `goto` [161]. Como consequência desses fatores, um número significativo de falhas de projeto em um sistema de software costuma estar localizado no código responsável pelo seu comportamento excepcional. Exemplos de tais falhas incluem vazamento de recursos [22, 182], exceções ignoradas [125, 148] e fluxo de controle imprevisto (uma exceção que não é tratada no local esperado) [150].

Em sistemas baseados em componentes, um fenômeno similar pode ser observado. Há evidência de que desenvolvedores de sistemas baseados na plataforma J2EE [13], um dos padrões da indústria para DBC, apresentam hábitos de programação, no tocante ao uso de tratamento de exceções, que tornam aplicações vulneráveis a falhas de projeto e difíceis de manter [148]. Alguns desses hábitos são tratadores vazios e não liberação de recursos alocados. A isso soma-se o fato de que sistemas baseados em componentes introduzem complicações que não existem na construção de sistemas de software "tradicionais": (1) em um sistema baseado em componentes, o código fonte de um ou mais componentes pode não estar disponível no momento em que o sistema é integrado [181]; (2) mesmo quando o código de um componente está disponível, pode não ser recomendável modificá-lo sob pena de introduzir falhas de projeto [181]; (3) componentes desenvolvidos por organizações distintas e integrados em um mesmo sistema podem fazer suposições conflitantes sobre as maneiras como falham, quais defeitos decorrem dessas falhas e como a falha de um componente servidor é sinalizada para seus componentes clientes [58] e dessas suposições resultam incompatibilidades arquiteturais[15]; e (4) suposições imprecisas feitas pelos integradores do sistema sobre a confiabilidade dos componentes no contexto em que serão usados podem esconder modos de falhas [102], por exemplo, o foguete Ariane-5 explodiu porque um componente reusado do foguete Ariane-4 produziu uma exceção devido a uma conversão numérica e os integradores do sistema supuseram que esse cenário nem precisaria ser testado já que, no foguete Ariane-4, ele não podia ocorrer [78].

Este trabalho examina o problema da construção de sistemas tolerantes a falhas baseados em componentes. O enfoque do trabalho está em sistemas de software nos quais a separação entre comportamento normal e comportamento excepcional é feita através de

[15]Do inglês: *architectural mismatches* [75].

tratamento de exceções. Tendo em vista a ênfase em sistemas baseados em componentes, a investigação se divide em duas partes, uma relativa à integração de componentes de software (*design with reuse*) e outra relativa à sua construção (*design for reuse*), o que resulta em duas perguntas de pesquisa:

1. *Como reduzir o número de falhas, em sistemas baseados em componentes, decorrentes de suposições conflitantes ou incompletas sobre o comportamento excepcional dos componentes integrados no sistema?*

2. *Como melhorar a estruturação interna de componentes de software, de modo a minimizar o impacto do comportamento excepcional desses componentes sobre a sua complexidade total?*

Esta tese propõe soluções para diminuir a quantidade de falhas de projeto introduzidas em sistemas baseados em componentes pelo uso de tratamento de exceções. Para alcançar esse fim, são apresentadas duas abordagens complementares, cada uma centrada em um elemento da dicotomia integração/construção. Ambas têm foco na estrutura do sistema como uma maneira de gerenciar a sua complexidade e reduzir o número de falhas de projeto. As Seções 1.3 e 1.4 descrevem as abordagens propostas para integração e construção de componentes de software tolerantes a falhas, respectivamente.

1.3 Uma Abordagem Tolerante a Falhas para a Integração de Componentes

1.3.1 Visão Geral

A primeira parte desta tese foca na integração de componentes de software para a construção de sistemas confiáveis. Mais especificamente, propomos uma abordagem rigorosa para o projeto arquitetural do comportamento excepcional de sistemas baseados em componentes. Conforme mencionado na Seção 1.2, o código de tratamento de exceções costuma ser a fonte de diversas falhas de projeto. Frequentemente, isso se deve ao fato desse código ser implementado de maneira *ad hoc*, sem nenhum planejamento *a priori*. Nos últimos anos, alguns autores vêm defendendo a idéia de que, para atingir os níveis desejados de confiabilidade, mecanismos para detectar e tratar erros devem ser desenvolvidos de maneira sistemática, ao longo de todas as fases do desenvolvimento de um sistema [59, 154, 160]: requisitos, projeto arquitetural, projeto detalhado, implementação e testes. Idealmente, a construção dos mecanismos de tolerância a falhas de um sistema devem seguir uma metodologia de desenvolvimento rigorosa (que lança mão de ferramentas e técnicas formais para a construção de partes críticas do sistema) ou inteiramente formal [12], a fim de garantir que os mecanismos que tornam esse sistema confiável são, eles próprios confiáveis.

Tendo em vista a influência da arquitetura de um sistema de software em sua qualidade final, se um sistema tem requisitos estritos de confiabilidade e tratamento de exceções será usado para estruturar seus mecanismos de tolerância a falhas, acreditamos que pode ser benéfico dar atenção especial ao comportamento excepcional desse sistema durante a etapa de projeto arquitetural [30, 89, 58]. Partindo dessa premissa, quatro capítulos (Capítulos 2-5) desta tese refinam, explicam e validam a seguinte hipótese, que responde à pergunta de pesquisa 1:

> **Hipótese 1**: *A descrição rigorosa de como exceções fluem entre componentes no nível arquitetural e a verificação automática de propriedades relativas ao comportamento excepcional do sistema nesse nível de abstração aumentam a chance de que falhas de projeto relacionadas ao comportamento excepcional sejam detectadas antes que o sistema seja implementado.*

1.3.2 Um Mecanismo de Tratamento de Exceções no Nível Arquitetural

A primeira parte desta tese propõe uma abordagem para modelar a arquitetura de um sistema da perspectiva do seu comportamento excepcional. No nível arquitetural, é necessário levar em consideração a maneira como estilos arquiteturais [158] influenciam a propagação de exceções. Um estilo arquitetural define um vocabulário de tipos de elementos de projeto que fazem parte de uma família de arquiteturas e as regras pelas quais esse elementos são compostos [76]. Alguns exemplos são o baseado em camadas, cliente-servidor e *pipes and filters*. O artigo "An Architectural-Level Exception Handling System for Component-Based Applications", apresentado no Capítulo 2, mostra como um estilo arquitetural específico, chamado C2 [173], pode influenciar o projeto de um mecanismo de tratamento de exceções, em especial no que se refere à propagação de exceções. Esse artigo foca nas exceções que fluem entre os componentes integrados para compor um sistema.

Alguns autores propuseram arcabouços [162, 179] que estendem plataformas de componentes existentes com suporte à detecção e ao tratamento de exceções. Apesar de funcionarem no nível arquitetural, esses arcabouços usam os mecanismos de tratamento de exceções das linguagens de programação empregadas e não levam em consideração as particularidades de diferentes estilos arquiteturais.

1.3.3 Especificação do Fluxo de Exceções em Arquiteturas de Software

Tendo em vista a influência de estilos arquiteturais na propagação de exceções entre componentes, abordagens para a descrição e análise de uma arquitetura da perspectiva do seu comportamento excepcional devem incluir a noção de estilo arquitetural. Essa idéia é materializada pelo arcabouço Aereal (*Architectural Exceptions Reasoning and Analysis*) [28], um conjunto de atividades, ferramentas e modelos para a descrição e análise do comportamento excepcional de um sistema de software a partir de sua arquitetura. Como estilos arquiteturais diferentes têm políticas distintas para propagação de exceções, o arcabouço dá suporte à definição de regras sobre como exceções fluem entre elementos arquiteturais em diferentes estilos. Aereal é baseado em ACME [76], uma linguagem de intercâmbio para a descrição de arquiteturas, Alloy [98], uma linguagem de especificação baseada em lógica relacional de primeira ordem, e nos conjuntos de ferramentas associados [101, 157]. O artigo "Specification of Exception Flow in Software Architectures" descreve em detalhes a abordagem proposta. Esse artigo compõe o Capítulo 3 desta tese.

Desenvolvimento de software centrado na arquitetura com Aereal exige uma metodologia de desenvolvimento que inclua atividades específicas para o levantamento do modelo de falhas do sistema (as maneiras como o sistema falha e as exceções que são geradas como consequência) e a especificação do seu comportamento excepcional [54, 154]. O arcabouço usa como entradas uma descrição da arquitetura do sistema e especificações informais do seu modelo de falhas e de seu comportamento excepcional. A partir dessas especificações, o arquiteto de software define uma nova visão arquitetural, chamada Visão de Fluxo de Exceções, que indica as exceções geradas e tratadas pelos elementos arquiteturais. Considera-se que um tratador pode **propagar** uma exceção, ou seja, relançar a exceção recebida ou uma exceção diferente, ou mascarar a exceção, ou seja, terminar sua execução sem lançar nenhuma exceção. Em seguida, uma ferramenta provida por Aereal usa essas informações para identificar todas as exceções que são recebidas e lançadas pelos elementos arquiteturais do sistema e complementa a descrição da arquitetura com essa informação. Diversas propriedades relativas ao fluxo de exceções entre os componentes arquiteturais podem ser analisadas a partir dessa descrição arquitetural estendida. Adicionalmente, o arquiteto pode definir restrições sobre como exceções são propagadas nos estilos arquiteturais instanciados em uma arquitetura. Usando-se as ferramentas da linguagem ACME, é possível verificar de maneira automática se a arquitetura obedece essas restrições.

Nos últimos anos, surgiram vários trabalhos cuja idéia central é modelar tratamento de exceções em nível arquitetural. Alguns desses trabalhos propõem novos mecanismos para a estruturação do comportamento excepcional no nível arquitetural [89, 57, 58], enquanto outros descrevem extensões relacionadas ao comportamento excepcional para ADLs [97]. Nenhum deles, porém, foca especificamente na descrição do comportamento excepcional e na verificação de propriedades relacionadas.

1.3.4 Um Modelo de Fluxo de Exceções no Nível Arquitetural

O arcabouço Aereal é baseado em um modelo formal genérico que define quais são responsabilidades de cada componente arquitetural, no tocante ao lançamento, recebimento e tratamento de exceções, e como exceções fluem entre esses componentes. O modelo também inclui um conjunto de regras formalmente especificadas que descrevem o funcionamento do mecanismo de tratamento de exceções. Esse modelo pode ser mapeado de forma quase direta para linguagens de especificação bem conhecidas, como Alloy [98] e B [3], e suas instâncias descrevem arquiteturas de sistemas de software. Através das ferramentas associadas a essas linguagens, é possível verificar de maneira automática se a arquitetura de um sistema satisfaz diversas propriedades que dizem respeito ao seu comportamento excepcional. O artigo "Reasoning about Exception Flow at the Architectural Level", que aparece no Capítulo 4, descreve esse modelo formal em detalhes.

Diversos trabalhos [38, 64, 150, 156, 186] propõem análises estáticas de código fonte que geram informações sobre o fluxo de exceções em programas. Nossa abordagem se baseia nesses trabalhos mas foca em fases anteriores do desenvolvimento de um sistema, mais especificamente, no seu projeto arquitetural, e leva em consideração as particularidades desse nível de abstração. Adicionalmente, a decomposição das regras de um mecanismo de tratamento de exceções em propriedades que possam ser verificadas isoladamente também é uma contribuição original.

1.3.5 Verificação de Tratamento de Exceções em Sistemas Concorrentes Cooperativos

Inicialmente, a abordagem proposta é apresentada no contexto de sistemas nos quais vigora a suposição de que erros são independentes. Consequentemente, é possível tratar cada exceção como se fosse a única em determinado instante de tempo. A maioria dos sistemas de software existentes e a maioria das linguagens de programação que implementam tratamento de exceções (C++, Java, Ada, Eiffel, etc.) aderem a esse modelo sequencial de funcionamento. Essa abordagem foi estendida para também contemplar sistemas concorrentes cooperativos, nos quais não é possível supor que erros sejam independentes. Em tais sistemas, múltiplas unidades de computação (processos, *threads*, componentes, etc.) concorrentes se comunicam assincronamente e cooperam com o fim de atingir um objetivo comum [20]. Esses sistemas diferem de sistemas concorrentes competitivos, nos quais múltiplas unidades de computação competem para ter acesso a recursos compartilhados. Em sistemas concorrentes cooperativos, se múltiplas exceções são levantadas simultaneamente, é necessário tratá-las de maneira cooperativa, já que podem ser decorrentes da mesma falha. Devido às complicações inerentes à programação de sistemas concorrentes, mecanismos de tratamento de exceções para sistemas cooperativos são sensivelmente diferentes de mecanismos para sistemas sequenciais. O artigo "Verification of Coordinated Exception Handling", que constitui o Capítulo 5 desta tese, apresenta uma técnica para a descrição e verificação do comportamento excepcional de tais sistemas.

Há diversas formalizações de mecanismos de tratamento de exceções para sistemas concorrentes cooperativos [172, 177, 185]. Entretanto, esses trabalhos focam em aspectos diferentes de tais mecanismos como: (i) a ordenação temporal de eventos [185]; (ii) o dinamismo das estruturas desses sistemas; e (iii) uma especificação detalhada de sua semântica, sem levar em consideração a verificação de sistemas [177]. Até onde pudemos averiguar, nenhum deles lida especificamente com a verificação de propriedades relativas à propagação de exceções.

1.3.6 Validação

Esta parte da tese foi validada através de execução de vários estudos de caso. Dois deles são descritos no Capítulo 3 e um outro é descrito no Capítulo 5. Um dos estudos de caso do Capítulo 3 é baseado em um exemplo de livro texto [163]. Os outros dois são baseados nas especificações de sistemas reais [54, 21]. Esses estudos de caso mostraram que a abordagem proposta valida a hipótese de pesquisa. Falhas de projeto decorrentes de suposições implícitas e/ou incompletas relativas ao comportamento excepcional dos componentes dos sistemas foram encontrados nos três estudos de caso. Alguns dos problemas foram detectados durante a etapa de modelagem dos sistemas, enquanto outros foram detectados pelas ferramentas de verificação empregadas. Adicionalmente, os artefatos resultantes do uso da abordagem servem como documentação da arquitetura do sistema, com a vantagem de poderem ser analisados de forma automática com o auxílio de ferramentas.

1.4 Uma Abordagem Tolerante a Falhas para a Construção de Componentes

1.4.1 Visão Geral

A segunda parte desta tese trata da construção de componentes de software. Mais especificamente, investigamos a adequação do paradigma de programação orientada a aspectos

(AOP) [106] para melhorar a estruturação interna de componentes, através de uma separação explícita entre comportamentos normal e excepcional. A idéia de empregar AOP na modularização de tratamento de exceções parte da premissa de que tratamento de exceções é um interesse inerentemente transversal [106, 117, 114, 180].

O uso de AOP para modularizar tratamento de exceções também é motivado pelo fato de AOP generalizar propostas de linguagens como Guide [112] e Extended Ada [53] para flexibilizar a associação de tratadores de exceções a diferentes elementos do programa. Por exemplo, na linguagem Guide, tratadores podem ser associados a comandos, métodos e classes. Já na linguagem Extended Ada, além dos tipos de associação já implementados por Ada, comandos, blocos, métodos e classes, também é possível associar tratadores de exceções a objetos. Em uma linguagem orientada a aspectos, a capacidade de associar tratadores a elementos de um programa é limitada apenas pelos mecanismos disponíveis na linguagem para selecionar pontos específicos desse programa[16]. Por exemplo, na linguagem AspectJ [114], uma extensão orientada a aspectos de propósito geral para Java, é possível associar tratadores a classes, métodos, exceções, comandos e até fluxos de execução. Já Eos [144], uma extensão orientada a aspectos de C#, além desses tipos de associação, permite que tratadores sejam associados a objetos de forma simples.

O uso de AOP para modularizar tratamento de exceções também é uma evolução com relação a trabalhos anteriores [71, 133] baseados em reflexão computacional [121]. Linguagens como as supracitadas AspectJ e Eos permitem que diversos pontos de um programa que não podem ser selecionados por protocolos de meta-objetos conhecidos [42, 138] sejam usados como contextos de tratamento de exceções. Exemplos incluem fluxos de execução e comandos arbitrários dentro do corpo de um método. Com base nesses fatores, dois capítulos desta tese validam e refinam a seguinte hipótese, que responde à pergunta de pesquisa 2:

> **Hipótese 2**: *O uso de programação orientada a aspectos para modularizar o código de tratamento de exceções de um programa melhora a qualidade desse programa, tanto do ponto de vista do comportamento normal quanto do excepcional.*

1.4.2 Um Estudo Quantitativo sobre a Aspectização de Tratamento de Exceções

Foi conduzido um estudo para avaliar as vantagens que o uso de AOP traz para a modularização do tratamento de exceções. Esse estudo é descrito em detalhes no artigo "Exceptions and Aspects: The Devil is in the Details" [24], que aparece no Capítulo 6.

[16]Do inglês: *join points*.

O estudo consistiu em refatorar para aspectos o código de tratamento de exceções de quatro aplicações diferentes, três delas baseadas em componentes. Três dessas aplicações foram implementadas originalmente em Java, enquanto a quarta foi escrita em AspectJ. Empregamos a linguagem AspectJ como representante do paradigma orientado a aspectos para separar tratamento de exceções dos outros interesses de cada sistema. Foi usado um conjunto de métricas [73] para medir, nas versões originais e refatoradas, quatro atributos de qualidade: separação de interesses, concisão, coesão e acoplamento. A investigação incluiu também uma análise dos sistemas refatorados levando em conta (i) a reusabilidade do código de tratamento de exceções e (ii) a escalabilidade de AOP para modularizar tratamento de exceções na presença de outros interesses transversais.

De acordo com os resultados desse estudo, o uso de AOP para separar os comportamentos normal e excepcional de um sistema é benéfico em várias situações recorrentes no desenvolvimento de software, mas isso depende de uma combinação de diversos fatores. Em diversas situações corriqueiras no desenvolvimento de software, aspectização *ad hoc* pode não ser possível ou piorar a qualidade do sistema. Em especial, a coesão dos sistemas-alvo do estudo, conforme medida pela métrica que empregamos, piorou consistentemente depois do tratamento de exceções ter sido modularizado com aspectos. Isso se deveu em grande parte à necessidade de, em alguns casos, modificar o código original antes de extrair o tratamento de exceções para aspectos. Adicionalmente, o grau de reuso de código de tratamento de exceções foi baixo. Embora esse resultado tenha sido esperado, tendo em vista a natureza específica de aplicação do código de tratamento de exceções [5], ele contradiz os resultados de um estudo anterior bastante conhecido [117]. Nesse estudo anterior, uma das consequências do uso de AOP foi uma grande economia de linhas de código devido ao reuso de tratadores. Finalmente, é difícil aspectizar um trecho de código quando mais de um interesse transversal afeta esse trecho.

Embora textos introdutórios [106, 114, 180] normalmente mencionem tratamento de exceções como um exemplo da utilidade de AOP, poucos trabalhos avaliam a adequação de AOP para modularizar o interesse de tratamento de exceções. Neste trabalho foi elaborado e executado um estudo detalhado, focando em quatro aplicações reais, para avaliar os benefícios trazidos pelo uso de AOP para se modularizar o comportamento excepcional desses sistemas. Até onde foi possível averiguar, não há nenhum estudo na literatura com o mesmo grau de profundidade ou com a mesma abrangência. O único estudo visando avaliar os benefícios do emprego de AOP para este fim, realizado por Lippert e Lopes [117], se baseou em uma infra-estrutura reusável, na qual o código de tratamento de exceções era muito simples e não-dependente de aplicação. Adicionalmente, a avaliação qualitativa desse estudo foi realizada em termos de atributos não tradicionais na literatura de Engenharia de Software, como *plugability* e suporte ao desenvolvimento incremental, e a avaliação quantitativa usou número de linhas de código como única métrica.

1.4.3 Um Catálogo de Cenários para Guiar a Modularização de Tratamento de Exceções com Aspectos

Com base nos resultados obtidos, propomos uma classificação para código de tratamento de exceções. Essa classificação leva em consideração os fatores que têm maior influência sobre a extração do comportamento excepcional de um sistema para aspectos, de acordo com os resultados do estudo. Adicionalmente, propomos um catálogo de cenários que consiste de combinações desses fatores. O objetivo desse conjunto de cenários é servir de guia para desenvolvedores incubidos da tarefa de modularizar tratamento de exceções usando aspectos, tanto para sistemas já existentes quanto para sistemas novos. Para atingir esse fim, o catálogo indica em quais cenários o uso de AOP melhora a qualidade do sistema e em quais piora. Um cenário é considerado benéfico quando: (i) a aspectização de tratamento de exceções tem um efeito positivo nos atributos de qualidade do sistema, conforme medido pelo conjunto de métricas empregado; e (ii) o código resultante não exibe "maus cheiros" [68]. Tanto a classificação proposta quanto os cenários são apresentados em detalhes no artigo "Implementing Modular Error Handling with Aspects: Best and Worst Practices", que aparece no Capítulo 7.

Apesar da popularidade crescente de AOP e de sua adoção inclusive na indústria [39], ainda não existem catálogos reunindo conhecimento relativo ao uso prático de AOP para estruturar tratamento de exceções. Essa situação difere fortemente do que acontece com a programação orientada a objetos [22, 60, 125, 141, 148].

1.5 Contribuições

Esta tese apresenta as seguintes contribuições:

1. Um mecanismo de tratamento de exceções que funciona no nível da arquitetura de software e leva em consideração algumas das particularidades de sistemas baseados em componentes. Esse mecanismo é centrado em um estilo arquitetural específico, C2 [173], e evidencia a influência que estilos arquiteturais têm na propagação de exceções (Seção 1.3.2 e Capítulo 2).

2. Uma abordagem para dar suporte à descrição e à análise de arquiteturas de software, da perspectiva do seu comportamento excepcional. Essa abordagem leva em consideração a maneira como diferentes estilos arquiteturais influenciam a propagação de exceções e complementa metodologias de desenvolvimento cujo foco é o comportamento excepcional do sistema (Seção 1.3.3 e Capítulo 3).

3. Um modelo formal que indica as responsabilidades de cada elemento arquitetural, no que se refere ao lançamento, recebimento e tratamento de exceções, e que permite que certas propriedades úteis relativas ao fluxo de exceções entre esses elementos sejam verificadas de maneira automática. Essas propriedades foram organizadas de tal maneira que cada uma pode ser verificada independentemente e o conjunto de todas elas descreve um mecanismo arquitetural de tratamento de exceções (Seção 1.3.4 e Capítulo 4).

4. Uma abordagem para a especificação e verificação de sistemas concorrentes cooperativos da perspectiva do seu comportamento excepcional (Seção 1.3.5 e Capítulo 5).

5. Uma análise dos fatores que tiveram influência na modularização do código de tratamento de exceções usando aspectos, com base na experiência adquirida através da refatoração de quatro aplicações distintas (Seção 1.4.2 e Capítulo 6).

6. Uma avaliação inicial dos efeitos da estruturação de tratamento de exceções usando aspectos quando outros interesses transversais também são modularizados através de aspectos (Seção 1.4.2 e Capítulo 6).

7. Um catálogo de cenários de tratamento de exceções que indica, para cada cenário, se é vantajoso usar aspectos para separar comportamento normal e comportamento excepcional (Seção 1.4.3 e Capítulo 7).

1.6 Organização da Tese

A tese foi estruturada como uma coletânea de seis artigos escritos em inglês que foram publicados ou submetidos para publicação em conferências e periódicos internacionais. Cada artigo corresponde a um capítulo e, para cada capítulo, há uma introdução e uma conclusão curtas, escritas em português, responsáveis por ligar os capítulos de forma mais coerente. Todos os artigos são reproduzidos na íntegra, com pequenas modificações no texto original para corrigir erros de escrita, diagramação e notação.

Esta tese está organizada da seguinte maneira:

- O Capítulo 2 descreve um mecanismo de tratamento de exceções que funciona no nível da arquitetura de software e leva em consideração as particularidades de sistemas baseados em componentes. Através da descrição dos elementos desse mecanismo, o capítulo introduz alguns conceitos úteis para o entendimento do restante da tese.

- O Capítulo 3 descreve o arcabouço Aereal para a descrição e análise de exceções no nível arquitetural.

- O Capítulo 4 descreve formalmente o modelo de fluxo de exceções usado pelo arcabouço Aereal. Esse modelo foca em sistemas onde vigora a suposição de que erros são independentes e que cada exceção pode ser tratada como se fosse a única levantada no sistema, em determinado instante de tempo.
- O Capítulo 5 descreve uma abordagem para a modelagem e verificação do comportamento excepcional de sistemas concorrentes cooperativos, nos quais não é possível presumir que múltiplos erros manifestados concorrentemente são consequência de falhas independentes.
- O Capítulo 6 apresenta o estudo que realizamos para avaliar as vantagens e desvantagens do uso de AOP para modularizar código de tratamento de exceções.
- O Capítulo 7 descreve um conjunto de cenários que visa auxiliar desenvolvedores a decidir em que situações o uso de AOP é vantajoso e quando é prejudicial.
- O último capítulo apresenta considerações finais, resumindo as contribuições deste trabalho e estabelecendo direções para pesquisas futuras.

Capítulo 2

Um Mecanismo de Tratamento de Exceções em Nível Arquitetural para Aplicações Baseadas em Componentes

Este capítulo se refere à Seção 1.3.2 do Capítulo 1 e descreve um mecanismo de tratamento de exceções que funciona no nível arquitetural. Esse mecanismo leva em consideração características específicas de sistemas baseados em componentes e foca nas exceções que fluem entre os componentes de software no nível arquitetural. O mecanismo proposto foi implementado através de um arcabouço orientado a objetos chamado FaTC2 (*Fault Tolerant C2*), uma extensão do *arcabouço* C2.FW [127]. Este último provê uma infraestrutura para a construção de aplicações baseadas no estilo C2.

O principal objetivo desse artigo é investigar como funciona a propagação de exceções entre os componentes de software integrados em um sistema. Acreditamos que ela tem diferenças fundamentais com relação à maneira como exceções fluem entre métodos e procedimentos no nível da linguagem de programação. Para ilustrar esse ponto, o artigo mostra como o estilo arquitetural empregado, C2 [173], influencia a propagação de exceções entre os elementos arquiteturais.

O artigo que este capítulo contém foi apresentado no *First Latin-American Symposium on Dependable Computing*, em outubro de 2003 [30]. Uma versão preliminar [31] descrevendo o arcabouço FaTC2 foi apresentada no *ICSE'2003 Workshop on Software Architectures for Dependable Systems*, em maio do mesmo ano.

An Architectural-Level Exception Handling System for Component-Based Applications

Fernando Castor Filho Paulo Asterio de C. Guerra

Cecília Mary F. Rubira

Institute of Computing
State University of Campinas
Campinas - SP - Brasil
{fernando,asterio,cmrubira}@ic.unicamp.br

2.1 Introduction

Modern computing systems require evolving software that is built from existing software components, in general developed by independent sources [15]. Hence, the construction of component-based systems with high dependability requirements out of existing software components represents a major challenge, since few assumptions can generally be made about the level of confidence of off-the-shelf components. In this context, an approach for the provision of fault tolerance based on the system's software architecture[48] is necessary in order to build dependable software systems assembled from untrustworthy components [89].

Exception handling[80] is a well-known technique for incorporating fault tolerance into software systems. An exception handling system (EHS) offers control structures which allow developers to define actions that should be executed when an error is detected. The user of such a system should be able to signal exceptions when an error is detected. The EHS should be able to find and activate an exception handler to recover from errors and put the system back in a coherent state. Even though many well-known programming languages provide EHSs [82, 108, 171], exception handling for component-based systems addressed at the architectural level remains an open issue. Component-based systems introduce new challenges which are not addressed by traditional EHSs. Some of these challenges include:

- exception handlers should be attached to higher-level abstractions specific to component-based systems, such as components and connectors, not only to implementation language constructs, such as methods and classes;

- the source code for the system components may not be available, specially when off-the-shelf components are employed. Hence, it is not possible to modify these components in order to introduce exception handling;

In this paper, we present ALEx, an architectural-level EHS which addresses the concerns presented above. ALEx is based on the work of Guerra et al [56], which describes a structuring concept for building fault-tolerant component-based systems based on the concept of idealised fault-tolerant component [5]. An idealised fault-tolerant component promotes separation of concerns between the abnormal activity (fault tolerance measures) of a system and its normal activity. Upon the receipt of a service request, an idealised fault-tolerant component produces three types of responses: *normal responses* in case the request is successfully processed, *interface exceptions* in case the request is not valid, and failure exceptions, which are produced when a valid request is received but cannot be successfully processed. Idealised fault-tolerant components may be organized into layers, so that components may handle exceptions raised by components located in other layers.

We also describe an object-oriented framework, called FaTC2, which implements our proposed EHS. FaTC2 is an extension of C2.FW [127], an OO framework that provides an infrastructure for building applications using the C2 architectural style [173]. The C2 style is a component-based architectural style [76] that supports large grain reuse and flexible system composition, emphasizing weak bindings between components. This style was chosen due to its ability to compose heterogeneous off-the-shelf components [127].

The rest of this paper is organized as follows. Section 2.2 provides some background information. Section 2.3 gives a motivation for the construction of an EHS for component-based applications. Section 2.4 presents ALEx, our approach for architectural-level exception handling. Section 2.5 provides a brief description of FaTC2, a Java implementation of ALEx based on the C2 architectural style. Section 2.7 gives a brief comparison with related work. Finally, Section 2.8 summarizes our conclusions and suggests directions for future work.

2.2 Background

2.2.1 Exception Handling

Following the terminology adopted by Lee and Anderson [5], a system consists of a set of components that interact under the control of a design. A fault in a component may cause an error in the internal state of the system which eventually leads to the failure of the system. Two techniques are available for errors: (i) *forward error recovery* and (ii) *backward error recovery*. The first technique attempts to return the system to an error-free state by applying corrections to the damaged state. The second technique attempts to restore a previous state which is presumed to be free from errors. Exceptions and exception handling constitute a common mechanism applied to the provision of forward error recovery.

An exception handling system (EHS) allows software developers to define exceptional conditions and to structure the abnormal activity of software components. When an exception is raised by a component, the EHS is responsible for changing the normal control flow of the computation within a component to its exceptional control flow. Therefore, raising an exception results in the interruption of the normal activity of the component, followed by the search for an appropriate *exception handler* (or simply *handler*) to deal with the signaled exception. The set of handlers of a component constitutes its *abnormal activity* part. For any exception mechanism, *handling contexts* associate exceptions and handlers. Handling contexts are defined as regions in which the same exceptions are treated in the same way. Each context should have a set of associated handlers, which are executed when the corresponding exception is raised.

2.2.2 C2 Architectural Style

In the C2 architectural style [173], components communicate by exchanging asynchronous messages sent through connectors, which are responsible for the routing, filtering, and broadcast of messages. Figure 2.1 shows a software architecture using the C2 style where the elements A, B, and D are components, and C is a connector. The thin vertical lines correspond to connections, ports for connecting the architectural elements. A configuration is a set of components, connectors, and connections between these elements. Components and connectors have a *top interface* and a *bottom interface* (Figure 2.1). Systems are composed in a layered style, where the top interface of a component may be connected to the bottom interface of a connector and its bottom interface may be connected to the top interface of another connector. Each side of a connector may be connected to any number of components or connectors. Two types of messages are defined by the C2 style: requests, which are sent upwards through the architecture, and notifications, which are sent downwards. Requests ask components in upper layers of the architecture for some service to be provided, while notifications signal a change in the internal state of a component.

The C2.FW framework [127, 175] provides an infrastructure for building C2 applications. The C2.FW Java [82] framework comprises a set of classes and interfaces which implement the abstractions of the C2 style, such as components, connectors, messages, and connections. C2.FW has been implemented in C++, Java, Python and Ada.

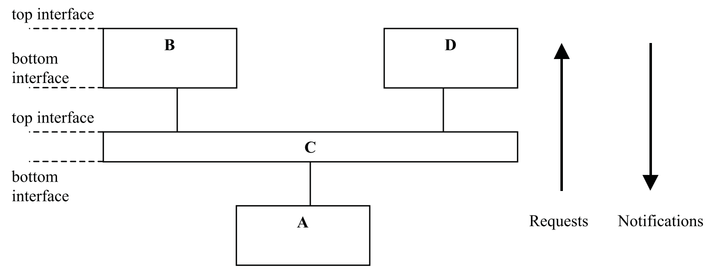


Figura 2.1: An example of software architecture based on the C2 style.

2.2.3 Idealised C2 Component

The work of Guerra et al.[56] uses the concept of Idealised Fault-Tolerant Component to structure the architecture of component-based software systems compliant with the C2 architectural style. It introduces the Idealized C2 Component (iC2C), which is equivalent, in structure and behavior, to the idealised fault-tolerant component. Service requests and normal responses of an idealised fault-tolerant component are mapped as requests and notifications in the C2 architectural style. Interface and failure exceptions of an idealised fault-tolerant component are considered subtypes of notifications.

The iC2C is composed of five elements: NormalActivity and AbnormalActivity components, and iC2C_top, iC2C_internal, and iC2C_bottom connectors. Its internal structure is presented in Figure 2.2. The NormalActivity component processes service requests and answers them through notifications. It also implements the error detection mechanisms of the iC2C. Internally, a NormalActivity component may be either a stateless or stateful component. However, since an iC2C should guarantee that each request received is processed in the initial state of the iC2C[89], stateless components are easier to deal with. In order to employ stateful components, some means for guaranteeing that requests are processed atomically, such as atomic transactions, should be provided.

The AbnormalActivity component encapsulates the exception handlers (error recovery) of the iC2C. While an iC2C is in its normal state, the AbnormalActivity component remains inactive. When an exceptional condition is detected, it is activated to handle the exception. In case the exception is successfully handled, the iC2C returns to its normal state and the NormalActivity component resumes processing. Otherwise, a failure exception is signalled to components in lower layers of the architecture, which become responsible for handling it.

The iC2C_bottom connector is responsible for filtering and serializing requests received by the iC2C. This conservative policy aims at guaranteeing that requests are always received by the NormalActivity component in its initial state, to avoid possible side-effects

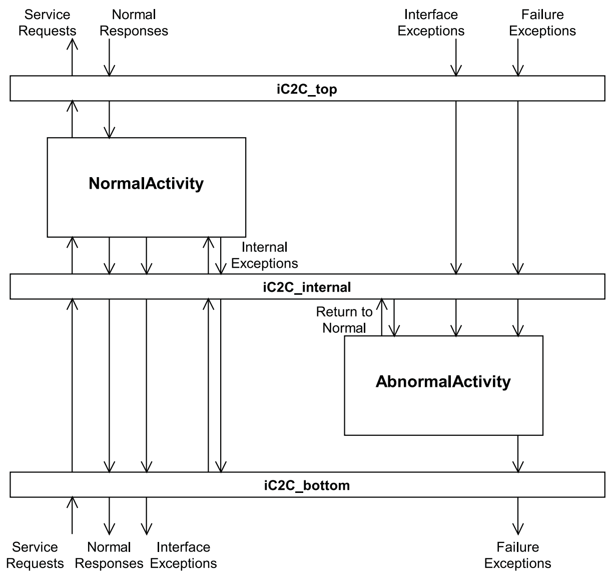


Figura 2.2: Internal structure of an iC2C.

of an exceptional condition caused by a concurrent service request. The iC2C_internal connector is responsible for message routing inside the iC2C. The destination of the messages sent by the internal elements of the iC2C depends on its type and whether the iC2C is in a normal or abnormal state.

The iC2C_top connector encapsulates the interaction between the iC2C and components located in upper levels of the architecture. It is responsible for guaranteeing that service requests sent by the NormalActivity and AbnormalActivity components to other components located in upper levels of the architecture are processed synchronously (request/response). The iC2C_top connector also performs domain translation, converting incoming notifications to a format which the iC2C understands and outgoing requests to a format which the application understands.

The structure of the iC2C makes it compatible with the constraints imposed by the C2 architectural style. Hence, an iC2C may be incorporated into an existing C2 configuration. Previous experiments [56, 89] with the iC2C model have shown its adequacy for the construction of component-based systems, including systems built from off-the-shelf components [88].

2.3 Exception Handling at the Architectural Level

Exception handling for component-based software systems introduces some difficulties which are not addressed by traditional EHSs. These particularities are a consequence of the differences between the abstractions supported by programming languages and component-based software development. In a traditional EHS defined by a programming language, exception handlers are attached to the constructs the language supports. For example, Guide [112] is an object-oriented language which supports the attachment of handlers to statements, methods and classes. Similarly, an EHS which supports the construction of component-based systems should allow exception handlers to be attached to their basic elements, namely, components, connectors, and configurations.

In traditional EHSs, when an exception is raised, a handler is searched locally, depending on the handling context. If none is found, the exception is signaled to the enclosing method invocation. However, this scheme for exception propagation may not be adequate for component-based systems. In software architectures, connectors are represented as first-class design entities, due to the potential complexity of the interactions among components. This feature allows the support of more sophisticated communication elements than simple method calls, and separates explicitly the execution flow of an application from its chain of method invocations. An EHS for component-based applications should consider these requirements during its design and support the propagation of exceptions according to the flow of information between components. In the C2 style, for example, components communicate by means of asynchronous message passing. Service requests are sent from components in lower layers of an architecture to components in upper layers, and response notifications flow in the opposite direction. Additionally, components may run in separate threads. Hence, in a C2 architecture, if a component issues a request which triggers the raising of an exception, the latter should be propagated to the component which issued the request. In other words, it should be propagated downwards the architecture, and not along the method invocation chain.

Finally, an architectural-level exception handling system should support the attachment of handlers to components without requiring modifications on them. This requirement is very important because, in component-based software development, the source code for the system components may not be available, specially if off-the-shelf components are employed. In order to leverage exception handling for component-based applications, we have devised an architectural-level EHS, called ALEx, which addresses the concerns stated above.

2.4 An Exception Handling System for Component-Based Applications

In this section, we present the main characteristics of our exception handling system and discuss our design decisions to address the requirements discussed in Section 2.3. We follow the guidelines established by Garcia et al [71], and describe ALEx in terms of the following features: exception representation, handler definition, handler attachment, exception propagation, handler search, and continuation of the control flow.

In our approach, an iC2C (Section 2.2.3) can represent a component, connector or configuration which has an architectural-level exception handler attached. We also employ the C2 architectural style for describing examples of architecture configurations, since it is leveraged by the concept of iC2C. Exceptions are wrapped by C2 notifications in order to be propagated along the layers of the system's architecture. Notifications representing exceptions are called *exception notifications*, and they are distinguished from normal notifications. Furthermore, we define regular C2 components as C2 components that: (i) do not send exception notifications, and (ii) will not recognize an exception notification received as being the signalling of an exception. In other words, a regular C2 component will recognize an exception notification as an ordinary notification sent in response to a request issued prior to the receipt of the notification. We will not consider special cases, such as regular C2 components capable of sending exception notifications.

Hereafter, we use the term "exception" to designate both *exceptions* and *exception notifications*, since the latter is simply an implementation concept which is not directly related to the EHS. For situations where implementation issues are relevant, we highlight the difference between the two terms.

2.4.1 Exception Representation

According to the taxonomy defined by Garcia et al [72], ALEx represents exceptions as *data objects*, that is, objects which are used for holding context information which is relevant for their handling. More specifically, we define exceptions as messages, which constitute a suitable abstraction for component-based systems. Exceptions are raised by calling a specific keyword (in our Java implementation, the `throw` keyword).

Different types of exceptions are organized hierarchically as classes. The class ArchitecturalException is the root of this hierarchy. ALEx defines two subclasses of ArchitecturalException, FailureArchitecturalException and InterfaceArchitecturalException, which correspond to the failure exceptions and interface exceptions of an idealised fault-tolerant component, respectively. Any raised exception which is not of type InterfaceArchitecturalException is treated as a failure exception, indicating that the architectural element may

```
public class AbnormalActivity ...  {
 public Message handleException(IOException e){
  // Body of a handler for IOException.
 }
 ...
 public Message handleException(ArchitecturalException e){
  // Body of the generic handler.  Must be implemented!
 }

 ...
}
```

Figura 2.3: Definition of handlers in ALEx.

be in an inconsistent state. Interface exceptions must be explicitly signaled.

The signatures of the operations in component interfaces should explicitly indicate the exceptions they raise. According to the work of Garcia et al.[72], this practice leads to better readability, modularity, maintainability, reusability and testability. Furthermore, explicitly declaring the exceptions raised by a component allows static checks to be performed by means of analysis tools, increasing dependability.

2.4.2 Handler Attachment and Definition

Our EHS supports multi-level attachment of handlers, that is, handlers may be associated with: (i) a component, (ii) a connector, (iii) a configuration. Figure 2.2 shows a set of exception handlers, represented by the AbnormalActivity component, attached to a component, represented by the NormalActivity component. The abnormal activity of a component is defined by a class which implements a set of methods, each one representing an exception handler. Furthermore, each of these classes should define at least a *default handler*, capable of handling an exception of the ArchitecturalException type. Figure 2.3 shows a partial definition of an exception handler which handles exceptions of types IOException and ArchitecturalException.

Exception handlers may also be attached to configurations, as well as to components and connectors. The possibility of attaching handlers to specific configurations allows that different exception handling contexts be defined within a single application. This is achieved by nesting exception handlers, as illustrated by Figure 2.4, where an iC2C is used as the NormalActivity component of another iC2C. This feature is specially useful for systems which have critical regions that require a higher level of fault tolerance.

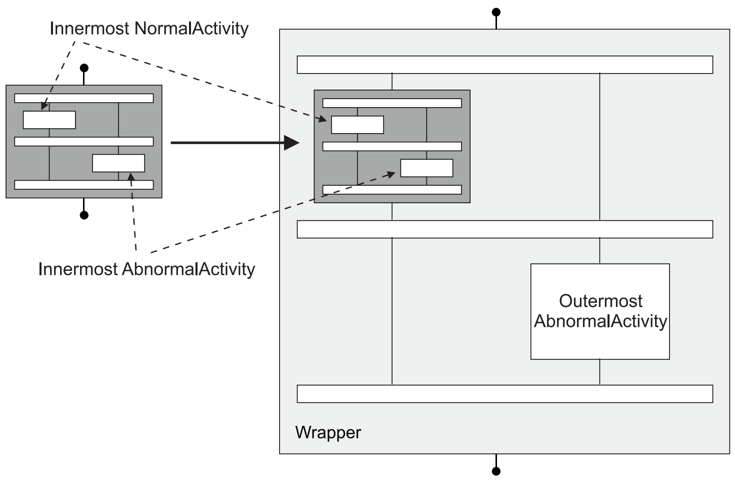


Figura 2.4: Nesting exception handling contexts.

2.4.3 Exception Propagation and Handler Search

Our exception handling model adopts explicit propagation of exceptions. The benefits of this approach are discussed in the work of Garcia et al.[72]. In ALEx, the handling and propagation of an exception depends on whether it is a *failure exception* or an *interface exception*. When an iC2C raises a failure exception, its handling is limited to the handlers within the same iC2C (that is, its AbnormalActivity component). If the exception can not be handled, then the predefined exception FailureArchitecturalException is further propagated. A handler may also explicitly resignal the exception to a component in a lower layer of the architecture. In Figure 2.4, exceptions raised by the NormalActivity component of the innermost iC2C are handled by the AbnormalActivity component within it. If the internal handling fails, the handler may resignal the exception to the AbnormalActivity component of the outermost iC2C. In case an exception reaches the lowest level of an architecture, an exception handler for the entire system should be executed.

An interface exception raised upon the receipt of a service request is not handled by the component, since it does not indicate that the component is faulty. In this case, the exception is propagated to the client component which issued the request. The client component handles this exception in the same manner as a failure exception, since it possibly indicates a fault within the client component.

The search for handlers for a failure exception raised by the NormalActivity component of an iC2C is defined as follows:

1. The EHS tries to find a specific handler for the exception in the AbnormalActivity component within the same iC2C. Handlers are searched according to the class

hierarchy defined for exceptions:

if an exception of class C is raised in a given system, a handler for exceptions of class C will be searched;

if none is found, the EHS proceeds searching for a handler for an immediate superclass S of C;

if none is found, the EHS searchers for a handler for an immediate superclass of S, and so on.

2. If no specific handler is found, the default handler is selected. This handler may actually handle the exception or simply resignal it.
3. In the latter case, if the iC2C which raised the exception is wrapped by another iC2C (as is the case in Figure 2.4), step 1 is repeated for the outermost AbnormalActivity component. Otherwise, the exception is propagated downwards the architecture, to the client component which issued the request that triggered its signaling. Step 1 is then repeated for the AbnormalActivity component of the client component.

The search for handlers for an interface exception raised by the NormalActivity component of an iC2C proceeds in a similar manner, except that steps 1 and 2 are not initially performed.

2.4.4 Continuation of the Control Flow

ALEx adopts the *termination* model[52], that is, if during the processing of a service request a component raises an exception and it is successfully handled by another component, execution is resumed by the latter.

Figure 2.5 presents an example of an architecture composed by two idealised C2 components (ClientComponent and ServerComponent), and a connector linking them. Figure 2.6 shows a UML seq. diagram illustrating a scenario where the control flow continues after an exception is successfully handled. The following steps describe the diagram:

1. ClientComponent requests a service from ServerComponent;
2. ServerNormalActivity tries to handle the service request;
3. ServerNormalActivity raises an exception which is received by ServerAbnormalActivity;
4. ServerAbnormalActivity tries to handle the exception;
5. ServerAbnormalActivity fails to handle the exception and raises a failure exception;

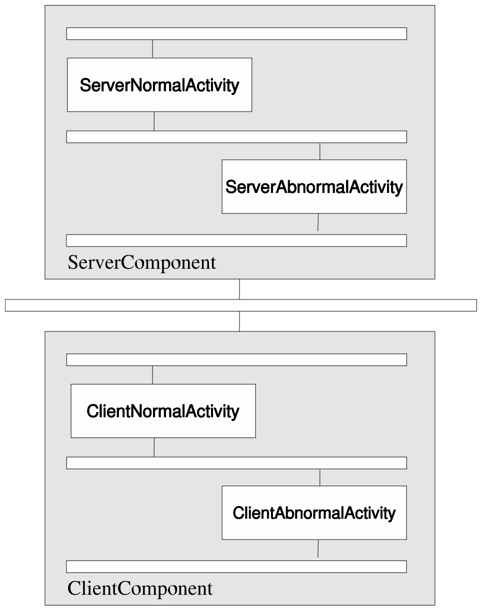


Figura 2.5: A C2 architecture composed by a client component and a server iC2C.

6. ClientComponent receives the exception and routes it to ClientAbnormalActivity;

7. ClientAbnormalActivity handles the exception and sends a *return to normal* request to ClientNormalActivity, indicating that processing should be resumed;

8. ClientNormalActivity resumes processing.

2.5 Exception Handling System in the Framework FaTC2

In this section, we describe FaTC2, an object-oriented framework which implements our architectural-level exception handling approach. FaTC2 is an extension of the Java [82] version of the C2.FW framework. The original C2.FW framework does not provide adequate support for the construction of fault-tolerant systems. FaTC2 extends C2.FW with the concept of iC2C, in order to provide the support for forward error recovery, by means of the EHS described in Section 2.4.

Figure 2.7 presents a partial class hierarchy for FaTC2, and its intersection with C2.FW. In the following sections, we describe the framework FaTC2, based on the notions described in Figure 2.2.

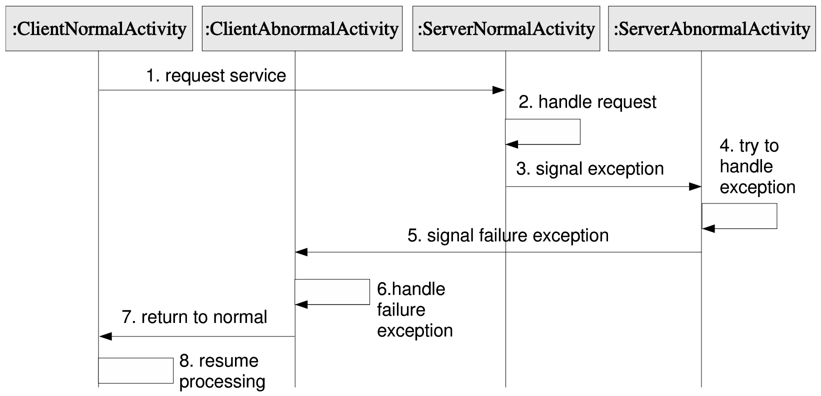


Figura 2.6: A scenario illustrating the termination model adopted by ALEx.

2.5.1 IC2C

The iC2C is the basic unit provided by FaTC2 for attaching handlers to components or configurations. The creation of an iC2C is encapsulated by the IC2C class. In order to create an instance of IC2C, objects representing the NormalActivity and AbnormalActivity components (Figure 2.2) must be supplied. Optionally, the developer may chose to also supply objects representing the iC2C_top and iC2C_bottom connectors, in case filtering or domain-translation are required. Otherwise, default implementations are employed.

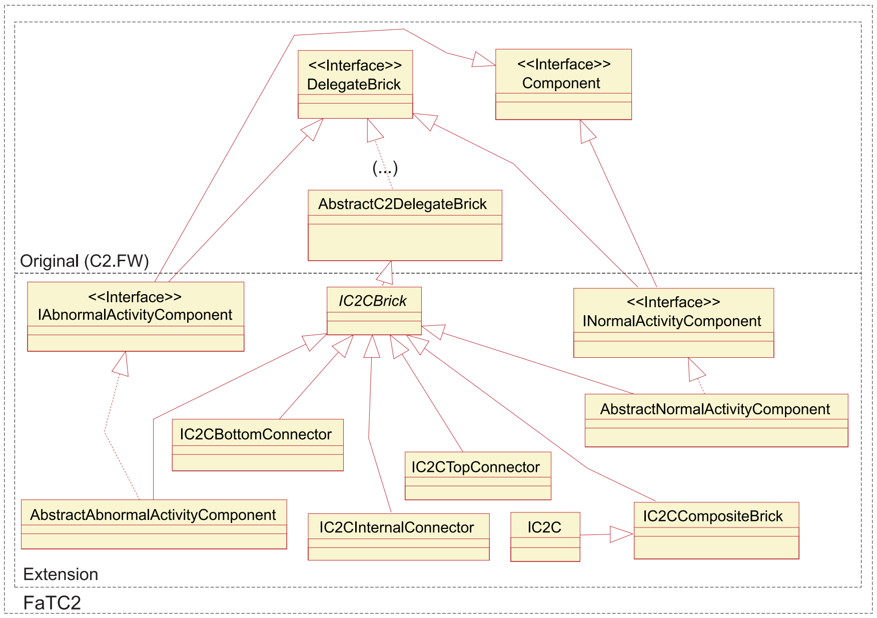
Although the IC2C class may be used directly in an application, it is recommended that developers create subclasses of it, specifying the NormalActivity and AbnormalActivity components, and iC2C_top and iC2C_bottom connectors which are to be used.

An analogous structuring may be used for representing fault-tolerant connectors, as long as the semantic differences between components and connectors are taken into account.

2.5.2 NormalActivity Component

The NormalActivity component encapsulates the functionality (normal activity) of an iC2C. It may represent both a single component and a configuration. In this work, we will only address the case where an iC2C represents a single component.

In order to define a NormalActivity component, a developer must provide a class that implements the INormalActivity interface. This interface declares three operations which define the application-dependent behavior of the component: `handleRequest()`, `returnToNormal()`, and `reset()`.



<<Interface>> DelegateBrick
<<Interface>> Component
(...)
AbstractC2DelegateBrick
Original (C2.FW)
<<Interface>> IAbnormalActivityComponent
IC2CBrick
<<Interface>> INormalActivityComponent
AbstractNormalActivityComponent
IC2CBottomConnector
IC2CTopConnector
AbstractAbnormalActivityComponent
IC2CInternalConnector
IC2C
IC2CCompositeBrick
Extension
FaTC2

Figura 2.7: A partial class hierarchy for C2.FW and FaTC2.

The `handleRequest()` method is responsible for (i) processing service requests and (ii) detecting errors. It takes as argument the request message to be processed, and returns a response notification to be delivered to the client component. If an error is detected during the processing of a service request, this method signals an exception, which may be a FailureArchitecturalException or an InterfaceArchitecturalException. Exceptions are caught by the framework and packaged as exception notifications, which are sent to the AbnormalActivity component.

The `returnToNormal()` and `reset()` methods are related to the abnormal activity of the iC2C. The former is called when the iC2C has successfully handled an exception, and should resume processing. The latter is called when the iC2C is unable to handle an exception, and should perform some cleanup actions before handling new requests.

FaTC2 provides developers with an abstract class which implements the application-independent behavior of the NormalActivity component, as defined by ALEx (Section 2.4). This class is called AbstractNormalActivityComponent, and should be extended by the class which implements the NormalActivity component for a given iC2C. In the situations described above, the tasks of delivering requests to the `handleRequest()` method, sending response notifications to client components, and packaging and sending exception notifications to the AbnormalActivity component are performed by AbstractNormalActivityComponent.

In case the handling of a request demands the NormalActivity component to request services from components located in upper layers of the architecture, the AbstractNormalActivityComponent class provides a utility method, `requestService()`, which may be used to send synchronous (request/response) requests transparently, upwards the architecture.

2.5.3 AbnormalActivity Component

The AbnormalActivity component encapsulates the exception handlers of an iC2C. In order to implement it, a developer must provide a class that implements the IAbnormalActivityComponent interface. This interface declares a single method, `handleException()`, which defines the default exception handler of the component. This scheme enforces the policy defined by ALEx, that at least the default exception handler must be implemented by every AbnormalActivity component.

Additional handlers are defined by *handler methods* that are declared in the same class which implements the IAbnormalActivityComponent interface. The order in which they are declared is not important. Handler methods present the following structure:

```
public Message handleException(<exception> e, Request m) raises Exception {
   // Body of a handler for <exception>.
}
```

In the code snippet above, *<exception>* stands for the exception type (a Java class) which the handler is capable of handling. The `Request r` parameter refers to the request which was being processed when the exception was signaled. If an exception is successfully handled, the handler method returns an object of type Message. This represents a C2 message which is delivered to the NormalActivity component in order for processing to be resumed. If an exception can not be handled, the handler method should resignal it or raise another exception. Either way, the exception is propagated to the enclosing exception handling context (Section 2.4.3).

FaTC2 provides developers with an abstract class which implements the application-independent behavior of the AbnormalActivity component, as defined by ALEx (Section 2.4). This class is called AbstractAbnormalActivityComponent, and should be extended by the class which implements the AbnormalActivity component for a given iC2C.

In case the handling of an exception requires the AbnormalActivityComponent to request services from other components, or from the NormalActivityComponent in the same iC2C, class AbstractAbnormalActivityComponent provides methods which allow synchronous requests to be carried transparently, similarly to the AbstractNormalActivityComponent class.

2.5.4 iC2C_top, iC2C_bottom and iC2C_internal Connectors

The IC2CTopConnector, IC2CBottomConnector, and IC2CInternalConnector classes are default implementations for the iC2C_top, iC2C_bottom, and iC2C_internal connectors, respectively.

IC2CTopConnector and IC2CBottomConnector may be extended in order to implement filtering of notifications in the top domain of an iC2C or requests in its bottom domain, respectively. A filtering scheme is defined by implementing the `accept()` method in a subclass of IC2CTopConnector or IC2CBottomConnector. A message m is processed only if `accept(`m`) == true`.

Subclasses of IC2CTopConnector may also implement domain translation in the top domain of the iC2C. The methods `translateIncomingMessage()` and `processOutgoingMessage()` are responsible for this task and are called by FaTC2, respectively, immediately after a message has been *accepted* by the iC2C_top connector, and immediately before a given message is sent by it. The iC2C_bottom connector is not expected to perform domain translation. In the C2 architectural style, an element placed in an upper layer of an architecture should make no assumptions about elements in the lower layers [173].

In case no filtering or domain translation is necessary, the default implementations for the iC2C_top and iC2C_bottom connectors may be used.

The IC2CInternalConnector class is reused without needing any specialization, since its only task is to route messages inside an iC2C.

2.6 An Application Example

In order to show the usability of FaTC2, we present a small example extracted from the Mine Pump Control System[163]. The problem is to control the amount of water that collects at the mine sump, switching on a pump when the water level rises above a certain limit and switching it off when the water has been sufficiently reduced. In this section, we describe an implementation for the example application which uses the infrastructure provided by FaTC2.

2.6.1 Description of the Architecture

The C2 architecture of our example is shown in Figure 2.8. The Pump component commands the physical pump to be turned on/off. Component LowWaterSensor signals a notification when the water level is low. WaterFlowSensor checks whether water is flowing out of the sump. The IdealPumpControlStation component controls the draining of the sump by turning on/off the pump, according to the level of the water in the sump. It includes an exception handler which is executed when the pump is turned on but no water

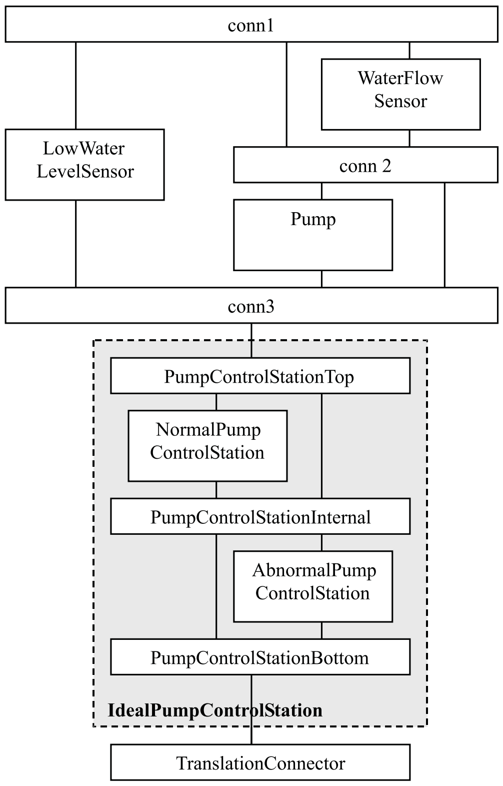


Figura 2.8: C2 configuration for the fault-tolerant Mine Pump Control System.

flow is detected. The error handler is implemented by the AbnormalPumpControlStation component. The Pump, LowWaterSensor and WaterFlowSensor components have been implemented as simple C2 components, while IdealPumpControlStation is an iC2C. In order to build the IdealPumpControlStation, five classes are implemented: NormlPumpControlStation, AbnormalPumpControlStation, PumpControlStationTop, IdealPumpControlStation and TranslationConnector.

Class NormalPumpControlStation implements the NormalActivity component of IdealPumpControlStation, that is, the methods defined by the INormalActivityComponent interface(Section 2.5.2). Due to the support provided by FaTC2, no messages need to be explicitly sent by any of the methods in NormalPumpControlStation; that is, the architect does not need to understand the internal protocol of the iC2C or the way it is implemented.

The AbnormalPumpControlStation class implements the exception handler of the IdealPumpControlStation. When an exception message is received by the `handleException()` method, the latter keeps sending new requests to Pump until either water flow is detected or the maximum number of retries permitted is reached. In the former case, normal activity is resumed(the method simply returns). In the latter, a FailureArchitecturalException message is sent downwards the architecture(the method throws an IC2CFailureException). The following code snippet partially illustrates this situation.

```
public Message handleException(Exception e, Request m)
 throws Exception {
(...)
 if(this.retries >= this.MAX_RETRIES) {
   throw new IC2CFailureException(e);
 }
(...)
```

In order to send an exception message downwards the architecture, the architect should throw a Java exception. In the example above, an exception of type IC2CFailureException, a subtype of Exception, is thrown.

The PumpControlStationTop class provides the IdealPumpControlStation component with an extension of the IC2CTopConnector class which performs filtering. When a request is issued by the IdealPumpControlStation, PumpControlStationTop records the type of the request sent, so that only a notification which is a response to that request is allowed to be processed. To build this filtering scheme, two methods had to be implemented: `accept()` and `processOutgoingMessage()`(Section 2.5.4).

IdealPumpControlStation is a subclass of IC2C. The IdealPumpControlStation class defines a public constructor which takes as argument the name of the IdealPumpControlStation instance to be created. TranslationConnector translates requests and notifications at the bottom interface of the IdealPumpControlStation (Figure 2.8).

The implementation of this architecture using FaTC2 was relatively easy, although a large amount of glue code had to be implemented in order for the NormalPumpControlStation component to work with FaTC2. This was due to the fact that the selected example application worked in an asynchronous manner while FaTC2 implements an iC2C which was modelled as a synchronous (request/response) entity. This highlighted some of the limitations of this synchronous approach and pointed out some directions we should follow in our future research, discussed in Section 2.8.

2.7 Related Work

Software fault-tolerance at the architectural level is a young research area that has recently gained considerable attention [88]. Most of the existing works in this area emphasize the creation of fault-tolerance mechanisms [50, 97, 145] and description of software architectures with respect to their dependability properties [155, 167]. Some approaches based on the idea of design diversity [5] have been developed in the context of the reliable evolution of component-based distributed systems. Both the Hercules framework [50] and the concept of Multi-Versioning Connectors [145] maintain old and new versions of components working concurrently, in order to guarantee that the expected service is provided, even if there are faults in the new versions. Both approaches are orthogonal to ours and could be used in conjunction.

Stavridou and Riemenscheneider [167], and Saridakis and Issarny [155] emphasize the formal description of architectures in order to prove they are reliable. By employing these specifications together with refinement laws which guarantee the preservation of the reliability property, both approaches intend on producing concrete architectural descriptions which are easily translated to code. None of the two works addresses the problem of incorporating error recovery into existing components.

The concept of iC2C[56] defines a structure for incorporating fault tolerance into component-based systems at the architectural level. It defines an internal protocol followed by its elements in order to enforce damage confinement [5]. Our work refines the concept of iC2C by introducing elements which are not addressed by its definition, such as the the representation of exception handlers and the enforcement of explicit exception propagation.

The work by Guerra et al.[88] deals with the problem of integrating COTS components in systems with high reliability requirements. It presents a case study where the concept of iC2C is used, together with protective wrappers [88]. The goal of this approach is to make non-reliable COTS components which represent a critical regions of systems reliable.

The work by Issarny and Banâtre [97] describes an extension to existing architecture description languages for specifying architectural-level exceptions (configuration exceptions). This work differs from ours because it emphasizes fault treatment [5] at the architectural level, by means of architecture reconfiguration. Our work, on the other hand, emphasizes architectural-level error recovery. Furthermore, it defines exceptions which should not be handled by any component in the architecture, that is, exception handlers are defined for the whole architecture and are activated under specific situations. In our approach, a component raises an architectural-level exception because it is unable to handle it, and other components in the architecture or exception handlers attached to enclosing configurations (as shown in Figure 2.4) may try to handle it as well.

2.8 Conclusions and Future Work

According to Sprott [166], there is a general consensus in the industry that software components will bring profound changes to the way software is built. Even now, software systems built out of reusable software components are used in a wide range of applications. Many of these systems have high dependability requirements and, in order to achieve the required levels of dependability, it is necessary to incorporate into them means for coping with software faults.

In this paper, we have presented ALEx, an architectural-level exception handling system which leverages the construction of fault-tolerant component-based applications. The use of traditional language-based exception handling systems for building fault-tolerant component-based systems presents some challenges which have been discussed in Section 2.3. ALEx addresses these issues by instituting exception handling at the architectural level. We have also briefly described FaTC2, an object-oriented framework for the construction of fault-tolerant component-based systems which implements ALEx.

It is important to note that architectural-level exception handling is not a replacement for language-level exception handling. In our view, exception handling in the language level should be the main technique for achieving fault tolerance internally to components (intra-components). Architectural-level exception handling (inter-components) should be employed when (i) an exception can not be handled by the component which raised it, and some other component in the architecture might be able to handle it, and (ii) mechanisms for error detection and recovery must be introduced in a component in order to make it trustworthy (or *more* trustworthy), but the component should not be modified or its source code is not available.

Until the present moment, the iC2C has been modeled as a synchronous (request/response) entity and the implementation of FaTC2 conforms to this model. That means that an iC2C is unable to handle asynchronous notifications and that requests are issued under the assumption that a response will be eventually received. This restriction might be too conservative for some applications, since a large amount of glue code may be necessary if a synchronous iC2C needs to interact with asynchronous components (Section 2.6). Hence, a future work for is the definition of an iC2C for which some of these restrictions are relaxed.

FaTC2 still does not implement all the features defined by ALEx. Some features, such as support for attaching handlers to arbitrary configurations and hierarchical handler search upon the receipt of an exception (Section 2.4.3) have not been implemented yet. Hence, another future work for FaTC2 is the implementation of the remaining features described by the specification of ALEx.

Finally, we believe that the design and implementation of ALEx are independent of the C2 architectural style and the iC2C. Hence, we plan to evaluate our approach on generic layered architectures by means of an implementation which does not rely on the concept of iC2C. In this case, new means for associating exception handlers to components would need to be established, in order to maintain the features described in

Section 2.4. Computational Reflection [121] and Aspect-Oriented Programming [106] are good candidates for this task.

2.9 Resumo do Capítulo 2

Este capítulo apresentou uma proposta de mecanismo de tratamento de exceções para aplicações baseadas em componentes. Diferentemente de mecanismos tradicionais, que focam nas construções existentes em linguagens de programação, esse mecanismo de tratamento de exceções funciona em um nível de abstração mais alto e se baseia nos elementos que compõem a arquitetura de um sistema de software. Foi discutida também a influência que o estilo arquitetural adotado, C2, tem no funcionamento do mecanismo, em particular na propagação de exceções e na busca por tratadores. O capítulo também descreveu sucintamente um arcabouço orientado a objetos chamado FaTC2 que implementa o mecanismo de tratamento de exceções proposto.

O próximo capítulo generaliza algumas das idéias apresentadas neste, no contexto do estilo C2, e propõe uma abordagem para a descrição da arquitetura de um sistema do ponto de vista do seu comportamento excepcional. Essa abordagem permite que desenvolvedores especifiquem a maneira como exceções são propagadas em diferentes estilos arquiteturais. Além disso, torna possível verificar diversas propriedades de interesse relativas ao fluxo de exceções entre os elementos da arquitetura. O foco do trabalho sai da implementação de arquiteturas tolerantes a falhas e passa para o projeto arquitetural do comportamento excepcional. Essa mudança de foco visa criar um ferramental que dê suporte à integração confiável de componentes de software no nível arquitetural, evitando incompatibilidades arquiteturais relativas ao comportamento excepcional dos componentes integrados.

Capítulo 3

Especificação de Fluxo de Exceções em Arquiteturas de Software

Este capítulo se refere à Seção 1.3.3 do Capítulo 1 e descreve o arcabouço Aereal (*Architectural Exceptions Reasoning and Analysis*), um conjunto de atividades, ferramentas e modelos para a descrição e análise do comportamento excepcional de um sistema software a partir de sua arquitetura. Esse arcabouço é centrado em uma nova visão arquitetural, chamada Visão de Fluxo de Exceções, que indica as exceções geradas e tratadas (mascaramento e propagação) pelos elementos arquiteturais. Como estilos arquiteturais têm políticas distintas para propagação de exceções, o arcabouço dá suporte à definição de regras sobre como exceções fluem entre elementos arquiteturais em diferentes estilos. Aereal é baseado em ACME [76], uma linguagem de intercâmbio para a descrição de arquiteturas, e Alloy [98], uma linguagem de especificação baseada em lógica relacional de primeira ordem. Através dos conjuntos de ferramentas associados a essas linguagens [101, 157] e das ferramentas providas por Aereal, diversas propriedades interessantes podem ser verificadas a partir da Visão de Fluxo de Exceções. Algumas dessas propriedades se referem à aderência da arquitetura às restrições impostas por diferentes estilos arquiteturais, no tocante à propagação de exceções. Outras dizem respeito às regras do mecanismo de tratamento de exceções que o arcabouço define e às características desejáveis do comportamento excepcional de uma aplicação. A abordagem proposta é validada através de dois estudos de caso.

O arcabouço proposto visa apoiar a integração confiável de componentes, exigindo que arquitetos sejam explícitos com relação às suposições que fazem no tocante ao comportamento excepcional dos componentes integrados. É fato bem conhecido que suposições incompletas ou incorretas são a principal causa de falhas introduzidas em um sistema durante a etapa de integração dos seus componentes [75]

O artigo que este capítulo contém foi publicado no Journal of Systems and Software [28]. Uma versão preliminar [25] foi apresentada no *ICSE'2005 Workshop on Architecting Dependable Systems*, em maio de 2005.

Specification of Exception Flow in Software Architectures

Fernando Castor Filho Patrick Henrique da S. Brito

Cecília Mary F. Rubira

Institute of Computing
State University of Campinas
Campinas - SP - Brasil
{fernando,pbrito,cmrubira}@ic.unicamp.br

3.1 Introduction

The concept of software architecture [159] has been recognized in the last decade as a means to cope with the growing complexity of software systems. According to Clements and Northrop [48], software architecture is the structure of the components of a program/system, their interrelationships and principles, and guidelines governing their design and evolution over time. It is widely accepted that the architecture of a software system has a large impact on its capacity to meet its intended quality requirements, such as reliability, security, availability, and performance, amongst others [11]. Software architectures are described formally using architecture description languages, or ADLs [128]. Most ADLs share the same conceptual basis whose main elements are architectural components (loci of computation or data stores), architectural connectors (loci of interaction between components), and architectural configurations (connected graphs of components and connectors that describe architectural structure) [128]. Software architectures described in an ADL can be analyzed by tools, in order to verify whether it satisfies some desired properties. In particular, automated and semi-automated analysis of software architectures are valuable tools in the construction of dependable systems.

Applications that can cause risks for human lives or risk of great financial losses are usually made fault-tolerant [5], so that they are capable of providing their intended service, even if only partially, when faults occur. Fault-tolerant systems include mechanisms for detecting errors in their states and recovering from these errors. Exception handling [52] is a well-known mechanism for structuring error recovery in software systems. Since exception handling is an application-specific technique, it complements other techniques for improving system reliability, such as atomic transactions [83], and promotes the implementation of very specialized and sophisticated error recovery measures. Furthermore, in applications where a rollback is not possible, such as those that interact with mechanical devices, exception handling may be the only choice available. An exception handling system (EHS) helps in the construction of dependable software by imposing constraints on the way exceptions and exception handlers may be used in a given language and detecting violations of these constraints. For instance, the EHS of Java detects unhandled exceptions at compile-time, unless the developer states explicitly that this checking should not be performed (by declaring the exception *unchecked*).

Usually, a system's error detection and handling mechanisms are developed in an ad-hoc manner during system implementation [52, 148, 180]. However, some researchers argue that, to achieve the desired levels of reliability, mechanisms for detecting and handling errors should be developed systematically from the early phases of development [154, 59], that is, from requirements, passing by analysis and design (architectural design and detailed design) phases. One approach that offers potential benefits to the development of dependable systems at the architectural level is to combine ADLs and exceptions during the design of the architecture. By extending formal architecture descriptions with information about exceptions, developers (i) better document their decisions about the flow of exceptions at the architectural level, (ii) make their assumptions about the EHS of the language(s) that will be used during system implementation explicit, and (iii) can check inconsistencies between the architecture description and the assumed EHS. Hence, in this paper, we attempt to answer the following research question:

> **Research question #1**: *How can software architectures be enriched with information about exceptions so that it is easy to verify some useful properties regarding exception flow?*

A generic solution to this question should deal with software architectures based on multiple architectural styles. An architectural style defines a vocabulary of types of design elements which are part of a family of architectures and the rules by which these elements are composed [76]. Well-known examples are Client/Server and Publisher/Subscriber [18]. Since architectural styles dictate how components in an architecture interact, they also impact the way exceptions flow amongst architectural elements [30]. Therefore, we refine research question #1 to include the notion of architectural style:

> **Research question #2**: *How can software architectures be enriched with information about exceptions so that it is easy to verify some useful properties regarding exception flow while taking into account the constraints imposed by different architectural styles?*

We present the Aereal (Architectural Exceptions Reasoning and Analysis) framework, which supports the extension of architectural descriptions with information about exceptions. These extended descriptions can be analyzed in order to check if they satisfy the constraints imposed by different architectural styles on how exceptions flow between architectural elements. Moreover, it is possible to specify properties of interest regarding exception flow and verify automatically if they are satisfied by an architecture. As enabling technologies, Aereal uses Alloy [98], a first-order relational language, ACME [76], an interchange language for architecture description, and their associated tool sets.

This work is organized as follows. Section 3.2 provides some background information on exception handling, ACME, and Alloy. Section 3.3 gives a motivation for specifying exceptions at the architectural level and lists some requirements that a solution to research questions #1 and #2 should satisfy. Section 3.4 provides an overview of Aereal approach to architecture-centric software development. Section 3.5 describes two case studies we have conducted to assess the usefulness of the framework. Section 3.6 reviews some related works. The last section presents concluding remarks and points out directions for future works.

3.2 Background

3.2.1 Exception Handling

When a system receives a service request and produces a response according to its specification, the produced response is said to be *normal*. Conversely, if the system produces a response that does not conform with its specification, this response is said to be *abnormal*, or *exceptional*. Abnormal responses usually indicate the occurrence of an error and since these responses are expected to occur only rarely, they are called *exceptions*. When exceptions occur, the system should handle them in order to return to a coherent state. The part of the behavior of a system that is responsible for handling exceptions is called abnormal, or exceptional, activity. Conversely, the part of the behavior of a system that is responsible for its functionality, as defined by its specification, is called normal activity.

Exception handling [52] is a mechanism for structuring the exceptional activity of a system so that errors can be more easily detected, signaled, and handled. It is implemented by many mainstream programming languages, such as Java, Ada, C++, and C#. These languages allow the definition of exceptions and the corresponding handlers. The set of exceptions and exception handlers in a system define its exceptional activity.

The concept of *idealized fault-tolerant component* (IFTC) [5] defines a conceptual framework for structuring exception handling in software systems. An IFTC is a component[1] in which the parts responsible for the normal and abnormal activities are separated and well-defined, within its internal structure. The goal of the IFTC approach is to provide means to structure systems so that the impact of fault tolerance mechanisms in

[1] In a broader sense; an object, a subsystem, a software component, or a whole systems.

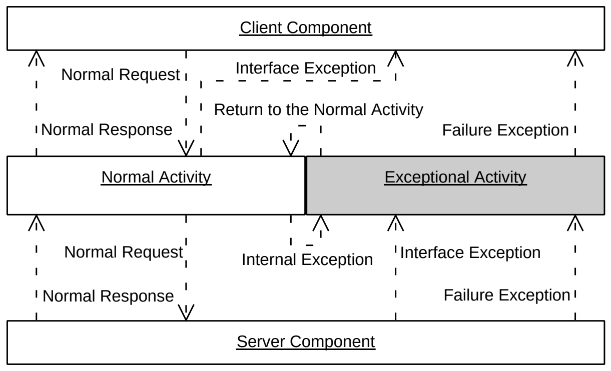


Figura 3.1: Idealised Fault-Tolerant Component.

the overall system complexity is minimized. This eases the detection and handling of errors. Figure 3.1 presents the internal structure of an IFTC and the types of messages it exchanges with other components in a system.

When an IFTC receives a service request, it produces a *normal response* if the request is successfully processed. If an IFTC receives an invalid service request, it *signals* an *interface exception*. If an error is detected during the processing of a valid request, the normal activity part of the IFTC *raises* an *internal exception*, which is received by the exceptional activity part of the IFTC. If the IFTC is capable of handling an internal exception properly, normal activity is resumed. If the IFTC has no handlers for an internal exception or is unable to handle an exception for which it has a handler, it *signals* a failure exception. Interface and failure exceptions are collectively called *external exceptions*. In this work, it is assumed that architectural elements behave like IFTCs. Hence, only external exceptions are taken into account, as strictly internal exceptions are not visible architecturally.

3.2.2 ACME

ACME [76] works as both an interchange language for architecture description and an ADL. It provides a simple structural framework for representing architectures, together with a flexible annotation mechanism. The language does not impose any semantic interpretation of an architectural description, but simply provides a syntactic structure to which semantic descriptions can be associated. This semantic information can then the interpreted by tools.

ACME supports the definition of four distinct aspects of architecture: (1) structure, (2) properties of interest, (3) constraints, and (4) types and styles. Structure aspect defines the organization of a system into its constituent parts, components, connectors, and attachments between these elements. The properties of interest aspect defines syntactic structures to which semantic information can be associated and analyzed by tools. Constraints are guidelines for how the architecture can change over time. The types and styles aspect defines classes and families of similar architectures.

In ACME, architectural styles are called *families*. An ACME family defines types of architectural elements that can be used in architectures that adhere to a specific architectural style and constraints on instances of these types. An ACME family[2] can extend another one by means of subtyping. Extension in ACME works as in object-oriented languages; all the elements defined by the parent ACME family are visible in and part of the child family. Architectural element types defined by a child ACME family may extend element types defined by the parent ACME family. Subtyping relations between element types work in the same way as subtyping relations between ACME families. Multiple inheritance is allowed and name conflicts result in invalid ACME families/element types.

Figure 3.2 presents part of the ACME definition of the Client/Server architectural style. The ClientAndServerFam family in the Figure 3.2 defines two types of components, ClientT and ServerT (Lines 6 and 9), and one type of connector, CSConnT (Line 13). Instances of each component type have exactly one access point, or "port" in ACME terminology (Lines 7 and 10). The ServerT component type has an integer property indicating the maximum number of concurrent requests an instance is capable of handling. In ACME, the access points of a connector (to which component ports are attached) are called "roles". Instances of CSConnT have two roles, one for the client and one for the server (Lines 14 and 15). Moreover, connector type CSConnT defines a simple constraint indicating that connectors of this type have exactly two roles (Line 16).

Figure 3.3 shows a partial definition of a very simple ACME system. An ACME system is an architecture description adhering to 0 or more architectural styles. Elements in an ACME system are instances of the element types defined by the styles to which it adheres. System NetBanking adheres only to the Client/Server style (Lines 2-3). It defines two components, InternetBankingServer (Lines 6-12) and Client1_WebBrowser (14-20), of types ServerT and ClientT, respectively, and one connector, conn (Line 13), of type CSConnT. Architectural structure is specified by defining attachments between component ports and connector roles. The receivedRequest port of InternetBankingServer is attached to the serverSide role of conn and the sendRequest port of Client1_WebBrowser is attached to the clientSide role of conn (Lines 4 and 5). Elements in a system can define instance-specific properties. In the example, both InternetBankingServer and Client1_WebBrowser define instance-specific properties regarding their positions in the screen (Lines 8-9, 16-17).

[2]In the rest of the paper, we use the terms "ACME family" and "architectural style" interchangeably.

```
Family ClientAndServerFam = {
   Port Type ClientPortT = { ... }
   Port Type ServerPortT = { ... }
   Role Type clientSideRoleT = { ... }
   Role Type serverSideRoleT = { ... }
   Component Type ClientT = {
       Port sendRequest : ClientPortT =  new ClientPortT;
   }
   Component Type ServerT = { ... }
       Port receiveRequest : ServerPortT =  new ServerPortT;
       Property max-concurrent-requests : int;
   }
   Connector Type CSConnT = {
       Role clientSide : clientSideRoleT =  new clientSideRoleT;
       Role serverSide : serverSideRoleT =  new serverSideRoleT;
       invariant size(self.roles) == 2;
   }
}
```

Figura 3.2: ACME definition of the Client/Server style.

Architectural modeling in ACME is supported by AcmeStudio [157]. AcmeStudio is an architecture development environment that allows the definition of new architectural styles and the modeling of systems which instantiate these styles using an intuitive graphical user interface. The environment includes a constraint solver called Armani [134] that checks whether an architecture satisfies the constraints defined by the styles to which it adheres.

Aereal assumes that architecture descriptions are written in ACME. Although other ADLs could be employed, some reasons have made us choose ACME: (i) it focuses on the structure of the system; (ii) it has powerful constructs for defining new architectural styles; (iii) it is extensible by the use of properties; and (iv) it has mature tool support. Since ACME is both an ADL and an interchange language, developers may employ other ADLs for modeling specific aspects of the system and then translate these specifications to ACME [77], so that they can be used with Aereal.

3.2.3 Alloy

Alloy [98] is a lightweight modeling language for software design. It is amenable to a fully automatic analysis, using the Alloy Analyzer (AA) [101], and provides a visualizer for making sense of solutions and counterexamples it finds. Similarly to other specification languages, such as Z and B [3], Alloy supports complex data structures and declarative models.

In Alloy, models are analyzed within a given scope, or size. The analysis performed by the AA is sound, since it never returns false positives, but incomplete, since the AA only checks things up to a certain scope. However, it is complete up to scope; the AA

```
import families\ClientAndServerFam.acme;
System Netbanking : ClientAndServerFam =
    new ClientAndServerFam extended with {
  Attachment Client1_WebBrowser.sendRequest to conn.clientSide;
  Attachment InternetBankingServer.receiveRequest to conn.serverSide;
  Component InternetBankingServer : ServerT =  new ServerT extended with {
    Port receiveRequest : ServerPortT =  new ServerPortT extended with {
      Property vis-x : float = 20.0;
      Property vis-y : float = -25.0;
    };
    ...
  };
  Connector conn : CSConnT =  new CSConnT;
  Component Client1_WebBrowser : ClientT =  new ClientT extended with {
    Port sendRequest : ClientPortT =  new ClientPortT extended with {
      Property vis-y : float = 65.0;
      Property vis-x : float = 176.0;
    };
    ...
  };
  ...
};
```

Figura 3.3: A trivial ACME system.

never misses a counterexample which is smaller than the specified scope. As pointed out by the Alloy tutorial [99], small scope checks are still very useful for finding errors.

Alloy is used by Aereal for analyzing exception flow mainly because (i) it focuses on how data is specified, an important feature for modeling exceptions as data objects [72]; (ii) it supports automated analysis; (iii) it has a mature constraint solver; and (iv) it is simpler and arguably easier to use than similar languages, such as B.

3.3 Specification of Exceptions at the Architectural Level

Usually, a large part of a system's code is devoted to error detection and handling [52, 148, 182]. However, since developers tend to focus on the normal activity of applications and only deal with the code responsible for error detection and handling at the implementation phase, this part of the code is usually the least understood, tested, and documented [52, 148]. In order to achieve the desired levels of reliability, mechanisms for detecting and handling errors should be developed systematically throughout all the phases of software development [59, 154]. As pointed out by many authors [11, 47, 165], the architecture of a software system has a strong impact on its ability to meet its intended quality requirements, for example, security, reliability, availability, and performance. Thus, if a system should be reliable and exception handling is one of the mechanisms that will be used to achieve this goal, it may be beneficial to consider exception handling-related issues during architectural design.

At the architectural level, an exception is a signal (message, event, language-level exception, etc.) employed by an architectural element to indicate to other elements that it has failed. An element that potentially receives an exception should be capable of doing something about it. During architectural design, it may still be too early to know exactly what. However, the design of an architecture involves assigning responsibility to architectural elements [48]. Therefore, it should be known at least which elements raise which exceptions, which elements are responsible for handling a certain exception, and which elements, upon receipt of an exception, just propagate it. This small amount of information is enough to reason about relevant properties regarding exception handling, for example, whether an exception raised by an element is actually handled by some other element in the architecture.

Architects often need to understand the architecture of a system from various perspectives, according to specific quality attributes. *Architectural views* [109] represent different aspects of the same architecture and each view shows how the architecture achieves a particular quality attribute [11]. These quality attributes are usually characterized in terms of properties of interest that the architecture should satisfy. A view for analyzing properties of interest about exception flow should include information such as: (i) the exceptions that each architectural element raises, handles, and propagates; and (ii) how exceptions flow between these elements. The following subsection presents an example of a view that describes the flow of exceptions in the architecture of an air traffic control (ATC) system. This example was extracted from a popular textbook on software architectures [11]. We highlight some of the limitations of the informal approach employed to specify this view. Section 3.3.2 lists some requirements we believe that any approach for modeling exceptions at the architectural level should satisfy.

3.3.1 A Motivating Example

An Air Traffic Control (ATC) system is a large-scale complex distributed system with very strict availability and performance requirements, meaning that the system should function 24/7 and timing deadlines must be met absolutely. The ATC system is an en route system which controls aircrafts from soon after takeoff until shortly before landing. Its end users are the air traffic controllers. It has a layered software architecture with one subsystem cooperating with others for services.

Due to its high availability requirements, the architects of the system designed a fault tolerance view of the software architecture (Figure 3.4) [10]. This structure describes how the faults are detected and isolated, and how the system recovers. So, besides presenting ways of detecting and isolating faults which belongs to specific components, the fault-

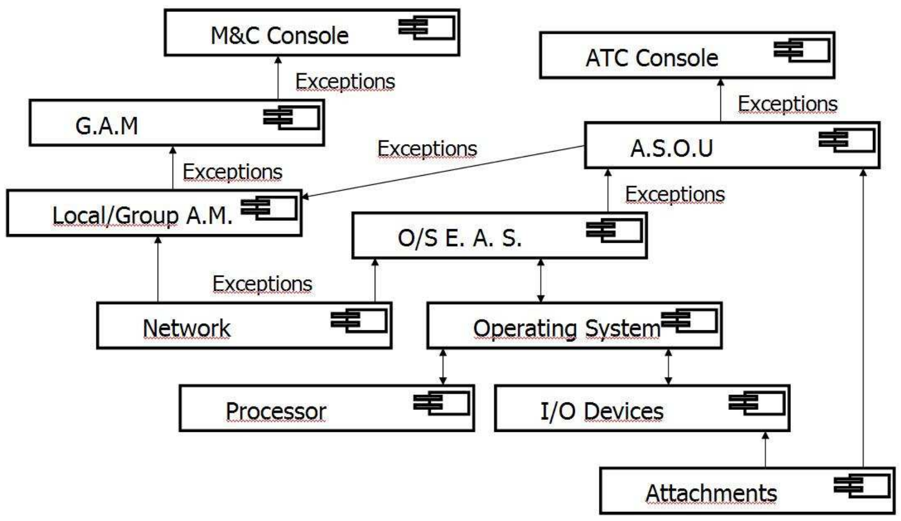


Figura 3.4: Fault tolerance architectural view of an ATC system.

tolerant hierarchy is designed do trap and recover from errors which are a consequence of cross-application interactions.

In Figure 3.4, the names of the components are abbreviated. Arrows indicate flow of exceptions between components (layers) of the architecture and a layer that receives exceptions from another layer is responsible for handling those exceptions. The idea of the authors is to express a hierarchical depiction of the system where layers that are closer to the top implement more general exception handling strategies. Each one of the eleven components of the architecture presented in Figure 3.4 has a specific role, as shown in Table 3.1.

The architecture of Figure 3.4 conveys useful information about exceptions in the system. For example, it is clear that exceptions flow from the Network component to the Local/Group A.M. and O/S E.A.S. components. It is also clear, due to the topology of the system, that the M&C Console and ATC Console layers have the most general exception handlers. This is expected, since, in an ATC system, users should always be notified of unhandled exceptions [11]. However, in spite of these useful pieces of information, the fault tolerance view of Figure 3.4 has some important shortcomings. First, this view does not specify what are the exceptions that each layer raises and handles. Second, it is not possible to infer whether there are relevant cause-effect relations between the exceptions that layers receive and signal. For example, there is no way of knowing whether the exceptions signaled by Local/Group A.M to G.A.M are a consequence of the exceptions received by the former from Network. Third, the view expresses a strictly hierarchical

Tabela 3.1: Components in the fault-tolerance view of the ATC system.

Component	Description
Monitor & Control Console (M&C Console.)	Gives an overview of the state of the system. It has special software to support monitoring and controlling functions and provides the top-level availability management functions.
ATC Console	A user console. It is also used by the controllers and is the access point to services which do not require a strict control of availability.
Global Availability Monitor (G.A.M.)	Manages the availability of functions within the suite. It determines which of the multiple redundant copies of an application program within a sector suite is the primary copy and should thus receive messages
Local/Group Availability Manager (A.M.)	A set of application-level system services. It is responsible for managing the initiation, termination, and availability of the application programs.
Application Software Operational Unit (A.S.O.U.)	Represents the application-level system services
Operating System Extensions Address Space (O/SE.A.S.)	Provides additional system services that are necessary to support a fault-tolerant distributed system. UNIX does not provide all these services.
Network	Provides communication to the redundant services in a transparent way. There are at least two networks: a local communication network (LCN), and a backup communication network (BCN). The latter is used in some LCN failure conditions.
Operating System.	A UNIX operating system.
Processor	The hardware that processes software instructions. Due to a design choice, redundant processors must exist.
I/O Devices	The device tasks are optional and are only present for applications that directly communicate with a device driver.
Attachments	The data, which either is necessary to process the application functions, or is stored as a result of a service.

structure for exception flow. It might not be appropriate to describe in the same manner the flow of exceptions in software architectures where components are peers. Fourth, it is not clear what a double-headed arrow means. These shortcomings could be alleviated by additional documentation complementing the diagram, but we did not find mention to such a documentation [10].

3.3.2 Requirements

To avoid the shortcomings highlighted in Section 3.3.1, we believe an effective approach for modeling exceptions at the architectural level should satisfy the following basic requirements:

Req.1: It should make it possible to unambiguously distinguish the exceptions that architectural components and connectors signal and catch, and how these exceptions flow between different architectural elements. Moreover, it should also be possible to specify the exceptions that an architectural element handles, the exceptions it raises, and the ones it propagates.

Req.2: To enhance maintainability, the specification of exceptions at the architectural level should be orthogonal and traceable to the "normal" architecture description.

Req.3: It should be supported by a pictorial (boxes-and-lines) representation, in order to be understandable by non-specialists and easier to use.

Req.4: It should take into account the notion of *architectural styles*. The approach should support the extension of traditional styles, such as layered (as in Section 3.3.1) and publisher/subscriber, with information about exceptions, in order to take into account the particularities of each style.

Req.5: It should support automated analysis. In this manner, it is possible to verify in a cost-effective way if the architecture presents some desired properties before the system is actually implemented.

Req.6: It should make it easy to verify if an architecture description adheres to rules defined by EHS of real-world programming languages, such as Java, Ada, C++, and C#.

3.4 The Aereal Framework

Architecture-based development with Aereal starts with the traditional activities of a software development process, namely, requirements analysis and architectural design of the system. It is also necessary to define scenarios in which the system may fail (fault model), what exceptions correspond to each type of error, and where and how the exceptions are handled (exceptional activity). The specification of the system's fault model and exceptional activity can be conducted as prescribed by some works in the literature [154, 54]. The most important results of these activities are a description of the system's architecture and informal specifications of the system's fault model and exceptional activity.

The use of Aereal in architecture-centric development involves the following activities:

1. Defining how exceptions flow between architectural elements for each architectural style used in the architecture description. This information is specified by defining new architectural styles, called *exceptional styles*, that complement the traditional (i.e. Client/Server, Layered) ones. Hereafter we call the latter *normal styles.*

2. Specifying the Exception Flow View of the software architecture. This view depicts the components that catch or signal exceptions (called *exceptional components*), as well as special connectors through which exceptions flow between these components (*exception ducts*). An exception flow view adheres to one or more exceptional styles.

3. Composing the architecture description with the exception flow view, producing

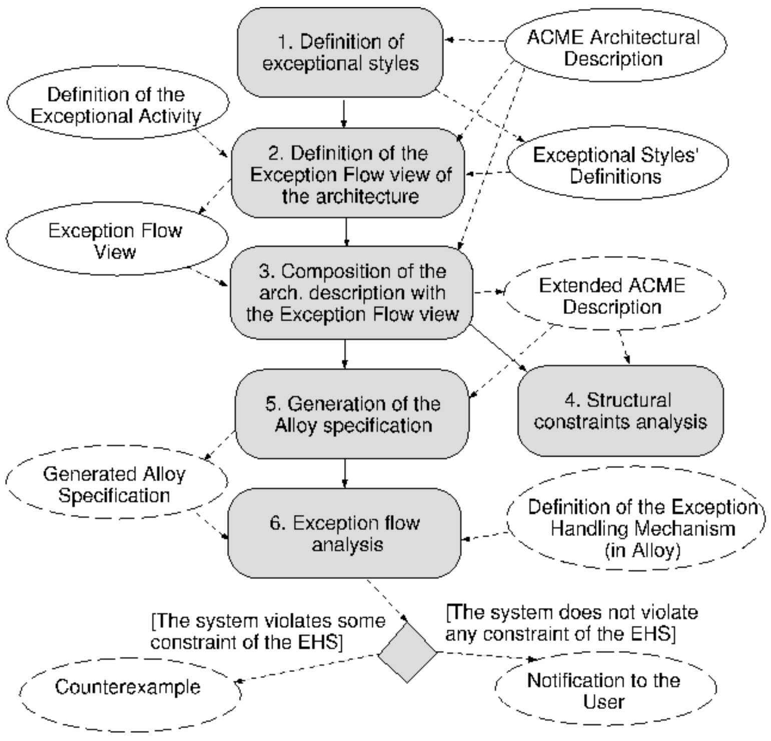

1. Definition of exceptional styles
ACME Architectural Description
Definition of the Exceptional Activity
2. Definition of the Exception Flow view of the architecture
Exceptional Styles' Definitions
Exception Flow View
3. Composition of the arch. description with the Exception Flow view
Extended ACME Description
5. Generation of the Alloy specification
4. Structural constraints analysis
Generated Alloy Specification
6. Exception flow analysis
Definition of the Exception Handling Mechanism (in Alloy)
[The system violates some constraint of the EHS]
[The system does not violate any constraint of the EHS]
Counterexample
Notification to the User

Figura 3.5: Overview of the Aereal framework.

an architecture description extended with information about exceptions (or simply *extended architecture description*).

4. Analyzing structural constraints, that is, checking whether the architecture description violates any of the constraints defined by the exceptional styles to which it adheres.

5. Generating an Alloy specification from the extended architecture description.

6. Analyzing exception flow based on the generated Alloy specification.

Figure 3.5 illustrates the main components of Aereal. In the figure, ovals represent artifacts and rectangles with rounded corners represent activities. Some of the ovals are dashed to indicate that they are either part of the infrastructure of the framework or generated automatically. An Aereal specification consists of a set of ACME system and family descriptions. One ACME system specifies the components and connectors view of the system [11]. The remaining ACME systems specify the exception flow views of the architecture. The ACME families specify architectural styles, both normal and exceptional, to which the ACME systems adhere. The definition of the exceptional activity and

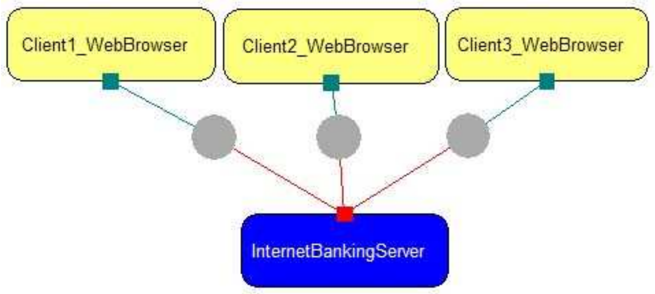


Figura 3.6: Architecture of a simple Internet Banking system.

the ACME architectural description in Figure 3.5 are assumed to exist and be produced as part of the software development process being used.

Figure 3.6 shows a high level components and connectors view of the architecture of an Internet Banking (IB) system. In the figure, components and connectors are represented by rectangles and circles, respectively. The architecture of the sytem adheres to the Client/Server architectural style. In the rest of this section, we use the IB system to make a more detailed presentation of Aereal.

The following subsections provide a detailed description of the workflow supported by Aereal.

3.4.1 Defining exceptional styles

Aereal uses special-purpose architectural connectors to model exception flow between components. These connectors, called exception ducts, are unidirectional point-to-point links through which only exceptions flow. Exception ducts may be refined when new exceptional styles are defined. In this manner, the specificity of each architectural style may be taken into account. The idea that simple point-to-point connections are suitable general-purpose abstractions for modeling style-independent communication between architectural components was proposed by Mehta and Medvidovic [129]. We have tailored this idea to our specific needs. This structural perspective on exception flow was adopted because: (i) it is intuitive to architects, who are used to thinking in terms of components and connectors; (ii) it is compatible with well-established views on what exception flow is [52]; and (iii) it is easy to integrate with the concept of architectural style. It abstracts away issues such as flow of control and flow of data.

Aereal includes a basic architectural style (an ACME family) called SingleExcFam on which all exceptional styles are based. SingleExceptionFam defines types that designate architectural elements that catch and/or signal exceptions. A new exceptional style must extend (be subtype of) two styles: SingleExcFam and the normal style to which it corresponds. The latter must be extended in order to make it possible to specify invariants for the new exceptional style that also involve elements of its corresponding normal style. It is possible to create many different exceptional styles for a single normal style and use some or all of them in an architecture description.

Style-specific features are introduced in a new exceptional style by creating subtypes of the element types that SingleExcFam defines and adding new internal elements (e.g. new ports in a component type), properties, and invariants. ExceptionalComponent is the supertype of all architectural components that deal with exceptions. By default, instances of ExceptionalComponent have no ports. New exceptional styles have to define subtypes of ExceptionalComponent that have ports, in order to specify whether instances of the type signal or catch exceptions, or both. SingleExcFam defines two port types, CatcherPortT and SignalerPortT, indicating that a component catches and signals exceptions, respectively. The supertype of all exception ducts is (for historical reasons) ExceptionalConnector. Since exception ducts are point-to-point channels, instances of this type have exactly two roles, of types CatcherRoleT and SignalerRoleT.

Example

Figure 3.7 presents a partial definition of the Exceptional Client/Server architectural style (ExceptionalClientAndServerFam ACME family). ExceptionalClientAndServerFam extends SingleExceptionFam and ClientAndServerFam (Lines 3 and 4). It defines two types of exceptional components, ExceptionalClientT and ExceptionalServerT (Lines 5 and 9), corresponding to clients and servers, respectively, and one type of exception duct, ExceptionalCSConnT (Line 13). Instances of ExceptionalClientT have a single port, of type CatcherPortT (Line 6). Instances of ExceptionalServerT also have a single port, of type SignalerPortT (Line 9). The invariants in lines 7 and 11 guarantee that ExceptionalClientT and ExceptionalServerT have no other ports. The invariant in lines 14-22 describes a general pattern of exception flow that can be specialized by each exceptional style. It states that if there is an exception duct linking two exceptional components in an exception flow view, then these components must exist as normal components in the normal architectural description and there must be a normal connector (a connector of type CSConnT, as shown in Line 20) linking them.

3.4.2 Specifying the exception flow view

An exception flow view comprises the components from the architecture description that signal or catch exceptions and exception ducts connecting these components. It adheres to one or more exceptional styles, depending on the normal styles the architecture description uses. Each element in an exception flow view is annotated with information about exceptions. All the elements in an exception flow view are instances of types that

```
import families\SingleExceptionFam.acme;
import families\ClientAndServerFam.acme;
Family ExceptionalClientAndServerFam extends SingleExcFam,
    ClientAndServerFam with {
  ...
  Component Type ExceptionalClientT extends ExceptionalComponent with {
    Port catchesPort : CatcherPortT =  new CatcherPortT;
    invariant size(self.ports) == 1;
  }
  ...
  Component Type ExceptionalServerT extends ExceptionalComponent with {
    Port signalsPort : SignalerPortT =  new SignalerPortT;
    invariant size(self.ports) == 1;
  }
  ...
  Connector Type ExceptionalCSConnT extends ExceptionalConnector with {}
  invariant Forall c1 : ExceptionalClientT in self.components |
    Forall c2 : ExceptionalServerT in self.components |
      Forall conn : ExceptionalCSConnT in self.connectors |
        ((attached(conn, c1) AND attached(conn, c2) AND
          connected(c1, c2)) -> (satisfiesType(c1, ClientT) AND
            satisfiesType(c2, ServerT) AND
              (Exists normalConn : CSConnT in self.connectors |
                attached(normalConn , c1) AND attached(normalConn , c2)) ))
                  <<label : string = "OK.";errMsg : string = "Problems.";>>;
  ...
}
```

Figura 3.7: Partial definition of the Exceptional Client/Server style.

the exceptional styles to which it adheres define. Aereal represents exception flow views as ACME systems.

The element types that SingleExcFam defines declare properties that are used to associate information about exceptions to architectural elements. In the exception flow view, instances of these types assign values to these properties. This information is taken into account during exception flow analysis (Section 3.4.4). Table 3.2 lists the properties declared by the element types of SingleExcFam and informally describes their semantics. All the properties in the table, except for the last one, are declared by ExceptionalComponent and ExceptionalConnector. Element types CatcherPortT and CatcherRoleT only declare properties handles, catches, and propagates. Element types SignalerPortT and SignalerRoleT only declare properties raises and signals. Only instances of SignalerPortT use property catcherPorts.

In the exception flow view, values assigned to properties declared by exceptional components and exception ducts add up to the values assigned to the homonym properties declared by associated ports and roles, respectively. For example, if an exceptional component C assigns value $\{RemoteException\}$ to property raises and signaler port P of C assigns value $\{IOException\}$ to its homonym property, it is assumed that the value of property $raises$ for P is $\{RemoteException, IOException\}$. This semantics makes it

Tabela 3.2: Properties employed by the exception flow view to associate information about exceptions to architectural elements.

Property	**Description**
signals	The set of exceptions signaled by an architectural element. Aereal automatically computes signals from the values assigned to raises, handles, and propagates for all the elements in an exception flow view. Optionally, it is possible to manually specify a set of exceptions. In this case, Aereal verifies if the element actually signals the exceptions during exception flow analysis.
catches	The set of exceptions caught by an architectural element. Similarly to signals, computed automatically by Aereal.
handles	The set of exceptions that an architectural element handles. In this case, "handles" means that the handler for the exception does not end its execution by raising an exception. After handling, normal execution is resumed.
propagates	This property is a variation of handles where the handler ends its execution by raising an exception. The propagates property specifies a cause-consequence relationship between an exception that an element catches and an exception that it signals.
raises	The set of exceptions that an element raises. Since only external exceptions are taken into account by Aereal, raises is a subset of signals
catcherPorts	The set of ports of type CatcherPortT associated to a port of type SignalerPortT. Exceptions caught by these catcher ports and not handled or propagated by the element are signaled through the associated signaler port. If catcherPorts is left unspecified for a given signaler port, Aereal assumes that this port is associated to all the catcher ports of the element.

possible to specify that different ports of the same component manipulate different sets of exceptions while retaining the ability to specify a common denominator. A similar rationale applies to exception ducts and roles.

Usually, unlike exceptional components, exception ducts are connectors that do not exist in the normal architecture description. They are orthogonal to "regular" connectors, in the sense that they do not necessarily follow the same rules for message/data passing and transfer of control. For instance, an exception duct between components that adhere to the Publisher/Subscriber style may or may not indicate transfer of control, depending on the semantics of the application. In our approach for exception flow analysis, this is not of utmost importance because we are not modeling system behavior. Moreover, even though a set of components in a Publisher/Subscriber architecture may communicate through a single normal connector, there may be several exception ducts between these components, depending on how the components may fail and which is responsible for handling which exceptions.

Example

The IB system's exception flow view only uses the Exceptional Client/Server style, defined by ExceptionalClientAndServerFam, since only the Client/Server style is used in the architecture description. It is important to notice, however, that various architectural styles could have been used.

Figure 3.8 shows part of the ACME definition of the IB system's exception flow view. This ACME system adheres to the Exceptional Client/Server architectural style (Lines 2-3). Line 9 specifies that the InternetBankingServer component signals an exception called RequestNotProcessedException. This exception is raised by the component itself (Line 8). Lines 14 and 15 state that exeception duct ExceptionalCSConnT0 catches exceptions of type RequestNotProcessedException and signals exceptions of type RemoteException, respectively. Furthermore, Lines 16-17 specify that ExceptionalCSConnT signals RemoteException as a consequence of catching RequestNotProcessedException.

```
 1 import families\ExceptionalClientAndServerFam.acme;
 2 System ExceptionalNetbanking:ExceptionalClientAndServerFam=
 3       new ExceptionalClientAndServerFam extended with {
 4
 5   Component InternetBankingServer : ExceptionalServerT =
 6           new ExceptionalServerT extended with {
 7     Port signalsPort : SignalerPortT =  new SignalerPortT extended with {
 8       Property raises : Set{} = {RequestNotProcessedException};
 9       Property signals : Set{} = {RequestNotProcessedException};
10     };
11   };
12   Connector ExceptionalCSConnT0 : ExceptionalCSConnT =
13       new ExceptionalCSConnT extended with {
14     Property catches : Set{} = {RequestNotProcessedException};
15     Property signals : Set{} = {RemoteException};
16     Property propagates : Sequence<> =
17       < RequestNotProcessedException, RemoteException >;
18   };
19   ...
20 };
```

Figura 3.8: Exception flow view of the IB system.

3.4.3 Composing architecture description and exception flow views

Structural constraints are evaluated after an extended architecture description is generated. This description is produced by the *Composer* tool, which is provided by Aereal. This tool reads the architecture description and the exception flow view and updates the former with exception-related information defined by the latter. This organization promotes separation of concerns at the architectural level, since the architecture description does not refer to the exception flow view, and the two exist as separate design artifacts and are composed automatically.

Figure 3.9 presents pseudo-code describing part of the functioning of the Composer. In the pseudo-code, variables representing architectural elements have fields through which it is possible to access their internal elements. For example, variable `excFlowView` represents the whole exception flow view and has the fields `families` and `components`, among others. The first is the set of all exceptional styles to which the exception flow view adheres and the second is the set of components in it. The Composer works by copying elements (components, connectors/exception ducts, ports in a component, attachments, etc.) from the exception flow view to the architecture description. Each element is identified by its name. The scope of an element is the architectural element of which it is part, for example, the scope of a component is a whole system, the scope of a port is a component, etc.

When the Composer attempts to copy an exceptional component for which there is a homonym component in the architecture description, the tool copies the properties and ports that store exception-related information (Table 3.2) to the component in the architecture description. As for exception ducts, to avoid conflicts between homonym ducts that appear in different exception flow views, the names of these elements are altered during composition and attachments involving them are updated to reflect these changes. It is not possible to combine existing exception ducts in the same way as exceptional components because this could potentially create exception ducts linking more than two components. Since the composition process has, by default, a “union” semantics, instead of “overwrite”, many different exception flow views can be composed with the same architecture description simultaneously. This supports a step-by-step development process where various exception flow views are specified for different parts of the architecture description. In later phases of system development, these views can be combined and analyzed in conjunction.

Structural constraints analysis employs the Armani constraint solver [134] or AcmeStudio (which includes Armani) to check if the extended architecture description satisfies the invariants defined by the exceptional styles to which it adheres. Violations of these invariants result in error messages.

Example

Figure 3.10 presents the components and connectors view of the extended architecture description of the IB system, as produced by the Composer. Notice that between the server and each client there are two connectors, a normal Client/Server connector and an exception duct.

```
Composer(ACMESystem excFlowView, ACMESystem archDescription) {
  // families stores the set of families to which a system adheres
  for each F in excFlowView.families
    if (F not in archDescription.families) add F to archDescription.families
  for each EC:ExceptionalComponent in excFlowView.components {
    if (archDescription.components contains an element NC:Component
        such that NC.name = EC.name) {
      // types stores the set of types of which an element is an instance
      for each T in EC.types
        if (T not in NC.types) add T to NC.types
      for each P:Set in EC.properties such that SingleExcFam defines P
        // NC also has P because it is an instance of all the
        // types of which EC is an instance
        NC.P = NC.P + EC.P // set union and assignment
      for each port PSE:SignalerPortT in EC.ports
        if (there is no PSN in NC.ports such that PSN.name = PSE.name)
          add PSE to NC.ports
        else if (SignalerPortT in PSN.types)
          for each P:Set in PSE.properties such that SingleExcFam defines P
            PSN.P = PSN.P + PSE.P // set union and assignment
        else report an error
      ... // The same for ports of type CatcherPortT
    }
    else add EC to archDescription.components
  }
  ... // copy exception ducts/roles
  for each A in excFlowView.attachments
    if not (A in archDescription.attachments)
      add A to archDescription.attachments
    else report an error
}
```

Figura 3.9: Algorithm executed by the Composer tool.

3.4.4 Analyzing exception flow

To analyze how exceptions flow in the architecture, it is necessary to generate an Alloy specification corresponding to the extended ACME description. Since generating the Alloy specification manually is a difficult and error-prone activity, Aereal provides a tool, called Converter, that automatically translates an extended ACME description to Alloy. The Converter works according to the following steps:

1. Read the ACME descriptions corresponding to the extended architecture description and the architectural styles (both normal and exceptional).
2. Build a language-independent representation of the architecture. This intermediate representation is based on a system model described elsewhere [26]. It includes architectural elements that are instances of types defined by exceptional styles and exception-related information associated to these elements through relations that correspond to the properties listed of Table 3.2. During the construction of the intermediate representation, the Converter discards information about architectural styles because this information is not used during exception flow analysis.

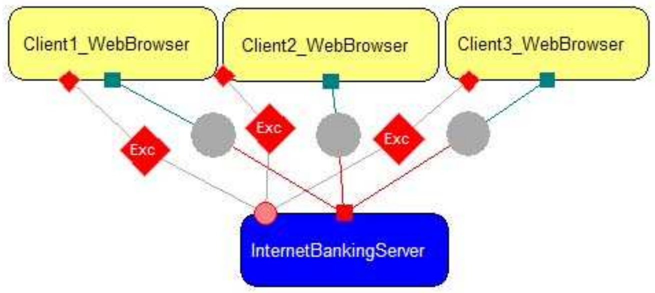


Figura 3.10: Extended architecture description of the Internet Banking system.

3. Compute signaled and caught exceptions. As pointed out in Section 3.4.2, the sets of exceptions that each architectural element signals and catches are automatically computed by Aereal. Manually-specified sets of signaled and/or caught exceptions are preserved by this process.

4. Write the text file comprising the Alloy specification of the system.

Exception flow analysis consists in verifying if the generated Alloy specification satisfies Alloy predicates corresponding to properties of interest. The properties of interest that a system must satisfy are split in three categories: basic, desired, and application-specific. Basic properties define the well-formedness rules of the model, the characteristics of valid systems. They specify the exception handling mechanism assumed by Aereal, which is based on C++', and how software architectures are structured. Examples of basic properties are presented below, stated informally.

BP1. Architectural elements signal all the exceptions they raise.

BP2. All exception ducts catch exceptions from and signal exceptions to exactly one exceptional component.

BP3. The graph formed by using exceptional components as vertexes and exception ducts as edges is connected.

BP4. Architectural elements signal all the exceptions they catch and do not handle.

Desired properties are general properties that are usually considered beneficial, although they are not part of the basic exception handling mechanism. In general, they assume that the basic properties hold. Some examples are the following.

DP1. Architectural elements do not have handlers for exceptions they do not catch.

DP2. All the exceptions caught by an architectural element are handled by it, even if some of its handlers end their execution by raising exceptions (explicit exception propagation [72]).

DP3. No unhandled exceptions.

Application-specific properties are rules regarding the flow of exceptions in a specific application. The Alloy definition of the exception handling mechanism used by Aereal (Figure 3.5) includes the specifications of several basic and desired properties that can be used "as-is". Developers only specify additional desired properties and application-specific properties, if any. The AA is employed to analyze the exception flow. If a property of interest is violated, the AA generates a counterexample with a configuration of the system for which the violated property of interest does not hold. Otherwise it notifies the user that the system may be valid.

By default, all exceptions are subtypes of RootException. It is possible to specify in Alloy a type hierarchy for the exceptions. For example, to mimic the EHS of Java, at least four exception types would be necessary: (i) Throwable, subtype of RootException; (ii) Exception, subtype of Throwable; (iii) Error, subtype of Throwable; and (iv) RuntimeException, subtype of Exception. Application-specific exceptions would inherit from these exception types. This type hierarchy is then taken into account when exception flow is analyzed. If no exception type hierarchy and no checks beyond the default ones are necessary, no knowledge of Alloy is required to use the framework.

Example

Figure 3.11 defines two Alloy predicates named `bp1` and `dp1`, formally specifying properties BP1 and DP1, respectively. Alloy predicates are logic sentences that must be checked by the AA. In the body of the predicates, `raises`, `signals`, `catches`, `propagates`, `handles`, and `catchesFrom` are names of relations that associate exception information to the elements of the system. For instance, the `signals` relation specifies what are the exceptions that a component signals and the exception ducts that catch them. The "." operator represents relational join. For example, given `raises` $\in$ `Component`$\leftrightarrow$`Duct`$\leftrightarrow$`RootException` and `C` $\in$ `Component`, where `Component`, `Duct`, and `RootException` are sets, `C.raises = ES` constrains the image of `C` under relation `raises` to be `ES` $\in$ `Duct`$\leftrightarrow$`RootException`. Predicate `bp1()` states that the set of exceptions that a component raises is a subset of the exceptions it signals. Predicate `dp1()` specifies that the set of exceptions that a component handles is a subset of the exceptions it catches. The operators `all`, `<:`, `&&`, and `in` represent, respectively, universal quantification, domain restriction, logical conjunction, and subset. A more detailed description of the system model defined by Aereal is available elsewhere [26].

```
/* Basic property BP1 */
pred bp1() {
  all C : Component | (C.raises in C.signals)
}
/* Desired property DP1 */
pred dp1() {
  all C : Component | all D : Duct | D in C.catchesFrom &&
  D.(C.handles) in D.(C.catches) &&
    (D.(C.catches))<:(D.(C.propagates))=D.(C.propagates)
}
```

Figura 3.11: Properties BP1 and DP1 specified in Alloy.

Figure 3.12 shows a counterexample generated by the AA when we analyzed exception flow in the Alloy specification of the IB system. This counterexample indicates that the generated Alloy specification violates basic property BP1. The error has been detected because we have modified the exception flow view of the IB system (Figure 3.8) so that InternetBankingServer signals RequestNotProcessedException but it does not raise it.

3.5 Evaluation

To assess whether Aereal meets the requirements of Section 3.3.2, we undertook two case studies to answer the following experimental questions:

EQ1. Does the abstraction for exception flow employed by Aereal, exception ducts, support the specification of exception flow views based on different architectural styles?

EQ2. Can Aereal help developers in gaining a deeper understanding of a system's exceptional activity?

EQ3. How useful is Aereal in the task of finding design errors and inconsistencies in the exceptional activity of a system?

EQ4. Does the approach employed by Aereal scale up well for systems with a large number of components, ducts, and/or exceptions?

The targets of the case studies were (i) a textbook example mining control system [163] (Section 3.5.1) and (ii) a financial system that registers and controls checkbooks, account contracts, and credit limits (Section 3.5.2). The latter was developed in an industrial setting and is part of a real application deployed in several companies.

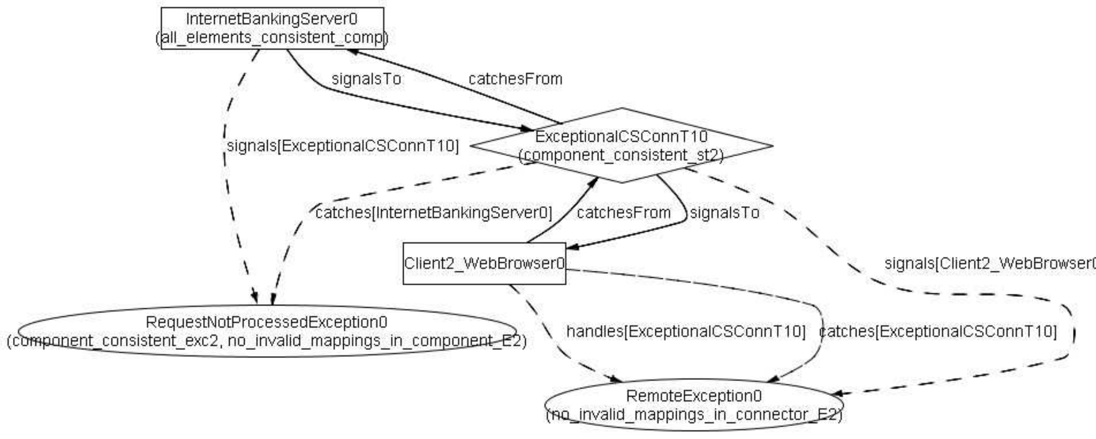

InternetBankingServer0
(all_elements_consistent_comp)
signalsTo
catchesFrom
signals[ExceptionalCSConnT10]
ExceptionalCSConnT10
(component_consistent_st2)
catches[InternetBankingServer0]
catchesFrom
signalsTo
Client2_WebBrowser0
signals[Client2_WebBrowser0]
RequestNotProcessedException0
(component_consistent_exc2, no_invalid_mappings_in_component_E2)
handles[ExceptionalCSConnT10]
catches[ExceptionalCSConnT10]
RemoteException0
(no_invalid_mappings_in_connector_E2)

Figura 3.12: A counterexample generated by AA.

3.5.1 Mining Control System Case Study

The case study is based on a simplified version of the control system for the mining environment [163]. 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 3.13. 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.

Figure 3.14 shows the components and connectors view of the architecture of the mining system [154]. This representation is related to the logical architecture of the system, that is, it describes the conceptual organization of the design elements into groups, independently of their physical packaging. In this architecture, collaborations between components are performed from higher to lower layers. Arrows between the software components indicate data flow between them. Components from upper layers make service requests

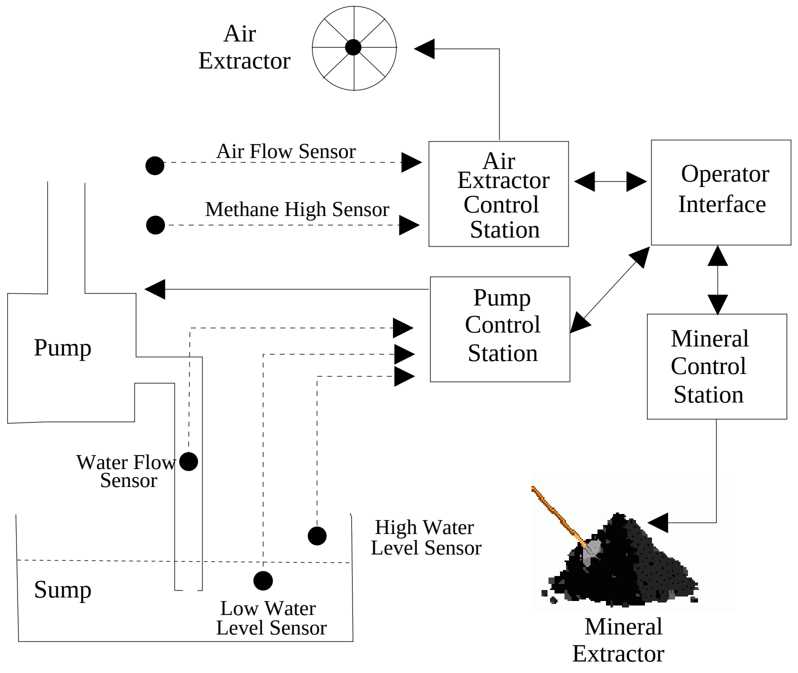


Figura 3.13: Schematic diagram of the mining system.

to components from lower layers and the latter produce responses that can be normal or exceptional. This architecture view uses only the layered architectural style [159].

The system can fail in several ways. All the sensors (MethaneHigh, AirFlow, WaterHigh, WaterLow, and WaterFlow) and actuators (AirExtractor, Pump, and MineralExtrator) may fail. Errors in these components are detected by the corresponding control stations. Sensors can fail by stopping to send information to their corresponding control stations. However, when they do send information, the latter is assumed to be correct. Actuators can fail by not working when switched on and not stopping to work when switched off.

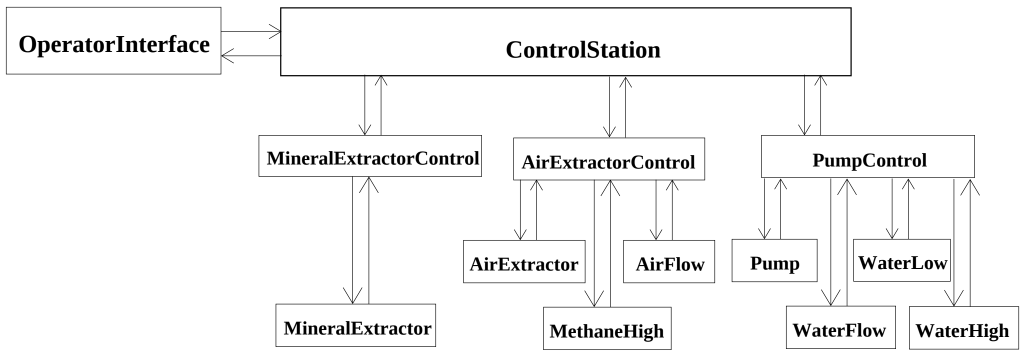


Figura 3.14: Architecture design of the mining control system.

Tabela 3.3: Exceptions related to the AirExtractorControl component.

Exception	Description
AirExtractorOffException	The level of methane is high, the air extractor is on, but no air flow is detected.
AirExtractorOnException	The level of methane is normal, the air extractor is off, but the sensor detects air flow.
SensorFailureException	A timeout occurs while the AirExtractorControl component is waiting for information from the sensors.

Whenever an error is detected, the detecting control station signals an exception to the ControlStation component, which attempts to interrupt execution and activate an alarm. We assume that communication between components is reliable. Table 3.3 shows the different types of exceptions that can be raised by AirExtractorControl. Exceptions raised by the other control stations follow the same pattern, except for MineralExtractorControl, which does not signal sensor-related exceptions. A detailed description of the exceptional activity of the mining system is available elsewhere [154].

This case study was conducted in two phases. Phase 1 (Section 3.5.1) addresses experimental questions EQ2 and EQ3. Phase 3 (Section 3.5.1) specifically targets experimental question EQ1. Both phases were conducted by one of the authors.

Applying Aereal - Phase 1

We started by applying the workflow presented in Figure 3.5 to the mining control system. The first activity consisted in describing the architecture of Figure 3.14 in ACME, using the definition provided by AcmeStudio for the layered architectural style. We then specified an exceptional layered style, represented by the ExceptionalLayeredFam ACME family. The latter is similar to the ExceptionalClientAndServerFam family (Section 3.4.1), but simpler. This simplicity is mainly due to two factors. First, exceptional components in layered architectures do not have ports by default, since any component is capable of (not)signaling/catching exceptions. This differs from Client/Server architectures, where responsibility for signaling and catching exceptions is unambiguously assigned to each type of component. Second, layered architectures have only one type of component, namely, the layer. A strictly layered style [18], where only adjacent layers communicate, would impose additional constraints, but we have not taken these constraints into account.

The following activity consisted in specifying the exception flow view of the architecture. Initially, this view comprised only the AirExtractorControl, PumpControl, MineralExtractorControl, and ControlStation components. The former three are responsible for detecting errors and signaling exceptions to the latter, which handles them. According to the specification of the system's exceptional activity [154], the expected behavior of ControlStation when it receives exception AirExtractorOnException is to "activate an external alarm, in order to notify the system operator about the existence of a safety threat, and shut down the system". Since the alarm is external to the system and its activation does not involve any exceptions being signaled by it or by ControlStation, we have not included it in the exception flow view. This approach faithfully reflects the specification of the system's exceptional activity. The problem with it, though, is that the operator of the system does not receive any information about the safety threat when the alarm rings. This seems unrealistic since the reason to ring the alarm is to notify the operator that it might be necessary to manually intervene, in order to avoid catastrophic damage. If the operator does not know anything about the safety threat that triggered the alarm, it is not possible to respond promptly.

In order to avoid the aforementioned problem, the ControlStation component should inform the operator about the nature of the threat. The natural way to do this is through the OperatorInterface component. However, the original specification of the mining system states that the two components only interact to start/stop mineral extraction. We interpret this is an incompleteness in the specification. To solve the problem, we modified the handling strategy of ControlStation. Besides attempting to shut down the system and activating the alarm, the handler now signals an exception, EmergencyException, to OperatorInterface. This new exception encapsulates the one that was caught, which includes information about the safety threat, and is handled by the OperatorInterface component. Two reasons motivated the choice of representing this interaction as an exception, instead of an additional service request: (i) the ControlStation component is notifying the OperatorInterface that an error occurred and exceptions are the natural means to do this; and (ii) ControlStation is located in a lower layer and service requests only flow from upper layers to lower layers, not the other way around. Figure 3.15 show part of the final specification of the ControlStation component.

After finishing the specification of the exception flow view, we used the Composer and Converter tools to produce the Alloy specification corresponding to the architecture of the mining system. We then used the AA to analyze exception flow and the latter reported that the specification satisfied all the basic and desired properties.

Applying Aereal - Phase 2

The second phase of the case study consisted in describing the mining control system using different modeling approaches. More specifically, our intent was to use different architectural styles, in order to provide at least an initial answer to experimental question EQ1. We produced two alternative designs for the mining system. The first one adheres to the C2 [173] architectural style and is an adaptation of another work [89]. The second alternative design for the mining system was an exercise aimed at assessing the difficulty

```
Component ControlStation : layerT, ExceptionalLayerT =
    new layerT, ExceptionalLayerT extended with {
  ... // other stuff
  Port CatcherPortT0 : CatcherPortT =  new CatcherPortT extended with {
    Property propagates : Sequence<> =
        < AirExtractorOnException, EmergencyException, ... >;
    Property catches : Set{} = {AirExtractorOnException,
        AirExtractorOffException,SensorFailureException};
   };
  Port CatcherPortT1 : CatcherPortT =  new CatcherPortT extended with {
    Property propagates : Sequence<> =
        < SwitchPumpOnException, EmergencyException, ... >; ...  };
  Port CatcherPortT2 : CatcherPortT =  new CatcherPortT extended with { ... };
  Port SignalerPortT0 : SignalerPortT =  new SignalerPortT extended with {
    Property catcherPorts : Set{} =
        {CatcherPortT0, CatcherPortT1, CatcherPortT2};
    Property signals : Set{} = {EmergencyException};
  };
};
```

Figura 3.15: Partial ACME specification of the ControlStation component.

of describing an exception flow view for an architecture that adheres to more than one architectural style.

In the C2 style, components communicate by exchanging asynchronous messages sent through connectors, which are responsible for the routing, filtering, and broadcasting of messages. Figure 3.16 shows a software architecture using the C2 style where the elements A, B, and D are components, and C is a connector. A configuration is a set of components, connectors, and links between these elements. Components and connectors have a *top interface* and a *bottom interface* (Figure 3.16). Systems are composed in a layered style, where the top interface of a component may be linked to the bottom interface of a connector and its bottom interface may be linked to the top interface of another connector. Each side of a connector may be connected to any number of components or connectors. Two types of messages are defined by the C2 style: requests, which are sent upwards through the architecture, and notifications, which are sent downwards.

As in Phase 1, we started by defining exceptional styles corresponding to the normal styles used in the architecture of each system. In the case of C2, we also had to specify the ACME family corresponding to the normal style. Specifying the exceptional C2 style in ACME was more difficult than doing the same for the layered and Client/Server styles. There were two reasons for this: (i) C2 has more complex topological rules than the styles we had seen so far and the corresponding exceptional style must respect these rules; and (ii) since connectors in C2 can be linked directly and ACME does not allow this, we had to model both C2 components and C2 connectors as ACME components. ACME connectors were used as links. Figure 3.17 presents an invariant from the ExceptionalC2Fam ACME family. This complex invariant states that there can only be an exception duct linking

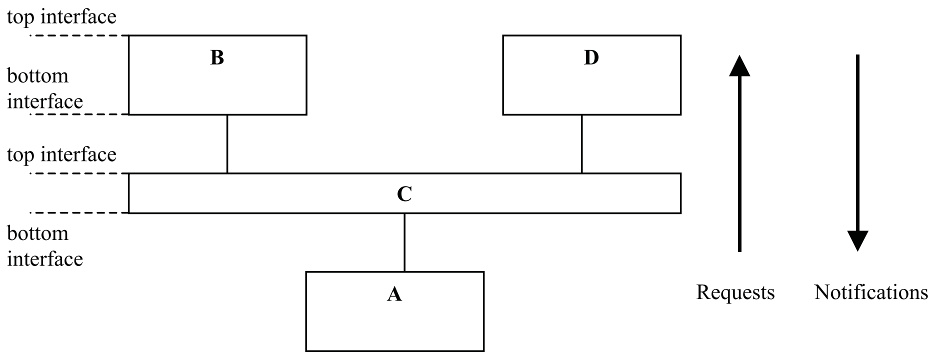


Figura 3.16: An example of software architecture based on the C2 style.

the signaler port of a component Cx and the catcher port of a component Cy if there is also a normal connector between these components such that: (i) the bottom interface of component Cx is linked to the top interface of the connector; and (ii) the top interface of component Cy is linked to the bottom interface of the connector.

```
invariant Forall conn : ExceptionalC2MessageBusT in self.connectors |
 Exists normalConn : C2MessageBusT in self.components |
  (Forall Cy : ExceptionalC2ComponentT in self.components |
   (reachable(normalConn, Cy) AND (Exists cp : CatcherPortT in Cy.ports |
    Exists cr : CatcherRoleT in conn.roles |
     (attached(cp, cr)) ) ) -> (Exists link2 : C2LinkT in self.connectors |
     (Exists tdp : TopPort in Cy.ports |
      Exists bdp : BottomPort in normalConn.ports |
       size(intersection(tdp.attachedRoles, link2.roles)) > 0 AND
       size(intersection(bdp.attachedRoles, link2.roles)) > 0 AND
       size(intersection(tdp.attachedRoles, bdp.attachedRoles)) == 0 ))) AND
       (Forall Cx : ExceptionalC2ComponentT in self.components |
        reachable(normalConn, Cx) -> (Exists sp : SignalerPortT in Cx.ports |
        Exists sr : SignalerRoleT in conn.roles |
         (attached(sp, sr)))  -> (Exists link1 : C2LinkT in self.connectors |
         (Exists bdp : BottomPort in Cx.ports |
          Exists tdp : TopPort in normalConn.ports |
           size(intersection(bdp.attachedRoles, link1.roles)) > 0 AND
           size(intersection(tdp.attachedRoles, link1.roles)) > 0 AND
           size(intersection(bdp.attachedRoles, tdp.attachedRoles)) == 0)));
```

Figura 3.17: An invariant defined by the ExceptionalC2Fam ACME family.

After finishing the ACME definition of the exceptional C2 style, we specified the C2-based exception flow view of the mining system. This task was straightforward because of the experience acquired in Phase 1. We then proceeded to generate the Alloy specification, and analyze exception flow. The AA reported that the specification satisfied all the basic and desired properties.

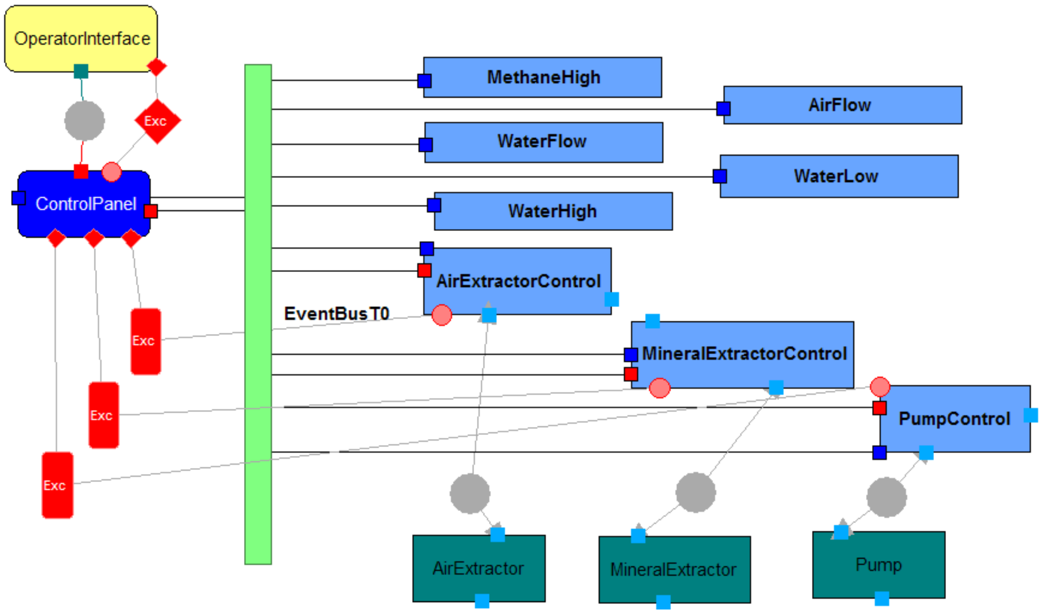


Figura 3.18: Extended architecture description of the multi-style mining control system.

The second alternative design for the mining control system uses three different architectural styles: Publisher/Subscriber, layered, and Client/Server. The subconfiguration that comprises components OperatorInterface and ControlStation adheres to the Client/-Server style. The subconfiguration comprising ControlStation, MineralExtractorControl, PumpControl, AirExtractorControl, and the sensor components adheres to the Publisher/-Subscriber architectural style. Finally, the subconfiguration that comprises the three actuators and their corresponding control stations adheres to the layered style. Some of the components play different roles in different subconfigurations. For example, ControlStation is a server in the first subconfiguration, and a publisher and subscriber in the second.

The exception flow view for the second alternative design only uses two exceptional styles: exceptional Publisher/Subscriber (ExceptionalPubSubFam ACME family) and exceptional Client/Server. It does not adhere to the exceptional layered style because no exceptions flow between the components of the layered subconfiguration. Figure 3.18 presents a diagram of the system's extended architecture description. The larger rectangles represent components, the circles and the vertical bar are normal connectors, and the small rectangles and the losangle with the Exc label are exception ducts.

Discussion

By modeling the flow of exceptions in the architecture of the mining system, we perceived that, even though proper handling of errors required manual intervention from the system's operator, the latter did not receive any information about the nature of these errors. This was stressed by the fact that the component responsible for handling exceptions did not directly interact with the operator, nor provided any information to components that did. Arguably, applying the Aereal approach to the mining control system gave us a deeper understanding of the system and, thus, helped in uncovering a design problem.

The experience of using three different approaches to model the mining system was positive. The ACME language allows the specification of different exceptional styles and adoption of more than one style by the same architecture description. Since Aereal is based on ACME and adopts a structural approach for the specification of exception flow, it leverages the features supported by the language. As shown in Section 3.5.1, it was possible to describe even complex exceptional style invariants. Furthermore, the task of describing a multi-style exception flow view was straightforward because all the exceptional styles have to extend the same ACME family and adhere to a predefined set of design rules (Section 3.4.1). This approach makes the specification of exception flow views uniform and, to a certain degree, independent of different exceptional styles. Furthermore, the Alloy specification abstracts away style-related information (Section 3.4.4) and retains only structural and exception-related information from the exception flow views. Therefore, in spite of the differences between the three designs of the mining system, the generated Alloy specifications for them are almost identical.

The first phase of the case study required approximately 13 developer hours. The second phase, including the specification of the C2Fam and ExceptionalC2Fam ACME families, required 16 developer hours.

3.5.2 Financial System Case Study

This case study consisted on developing part of a financial system with strict dependability requirements. The system belongs to the domain of banking applications and was being developed by a medium-sized Brazilian company specialized in banking automation. The part of the system that we used for the case study supports six basic operations:

1. Solicitation of checkbooks. The customer requests a check-book (in person, by phone, through the Internet, etc.).

2. Delivery of checkbooks. The system manages the delivery of previously requested check-books.

3. Cancellation of checks. In cases of loss, theft, or another specific reason, the customer can cancel checkbooks.

4. Retention of checks. The retention of a check occurs during a deposit in check. These checks are restrained and processed for future payment.

5. Cancellation of accounting contract. At any time, the customer can lose the credit of his account. In these cases, the contract of the customer must be canceled.

6. Inclusion of additional limit. Depending on necessity and credit conditions, the customer can receive an additional credit limit.

Operations #1-3 can be initiated by an operator or by a customer. Operations #4-6 can only be initiated by an operator of the system. Operations Cancellation of Accounting Contract and Cancellation of Checks have very strict availability requirements and should be online 24/7. The operations presented above were described as use cases. In this paper, for the sake of simplicity, we focus on the Cancellation of Accounting Contract use case. A very detailed specification of the financial system is available elsewhere [14]. The specification of each use case includes the input information expected by the system, normal, alternative, and exceptional scenarios, and assertions (pre and postconditions). Exceptional scenarios were derived from violations of assertions and alternative scenarios triggered by errors.

The financial system was developed using the MDCE+ method [54], which is an extension of the UML Components process [41] that includes activities for specifying a system's exceptional activity. The case study was planned by the authors and executed by two other developers, one of them a specialist in the domain of the application. The validation of the specified architecture was performed by one of the authors. The goal of this case study was twofold: (i) to gather experience from the use of Aereal in an industrial setting; and (ii) to assess the scalability of the Aereal approach.

The architecture of the financial system adheres to the layered architectural style. It has four layers: user interface, system, business, and database. The system and business layers implement the application-dependent and application-independent business rules, respectively. Components in the business layer can be reused across different applications from the same domain, since they are application-independent. The user interface and database layers are self-explanatory. Figure 3.19 presents part of the components and connectors view of the financial system. Following the previously adopted notation, components are represented by rectangles and normal connectors by circles. The diagram depicts the part of the system's architecture that is relevant to the Cancel Accounting Contract use case. Component AccountOps from the system layer implements the business logic of the use case. To provide its intended functionality, AccountOps interacts with

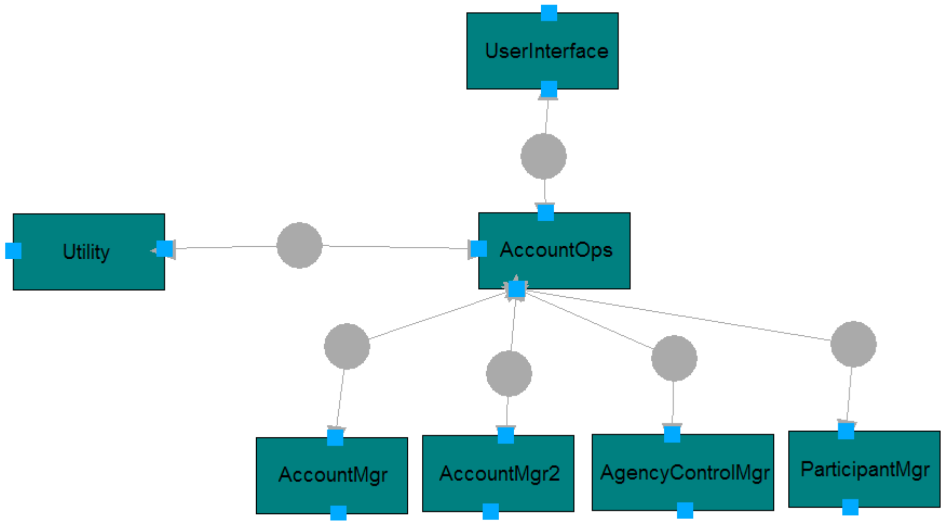


Figura 3.19: Partial components and connectors view of the architecture of the Financial System

three components from the business layer, AccountMgr, AgencyMgr, and ParticipantMgr. The existence of two AccountMgr components stems from availability requirements of the system.

The system can fail in several ways and for a number of different reasons. For example, the AccountOps component alone signals 15 different types of exceptions. In total, there are more than 50 types of exceptions that flow between architectural components. This large number of exceptions stems from a development policy adopted by the organization where the case study was conducted. According to this policy, even very similar errors and situations that are not normally treated as errors generate new exception types. Table 3.4 shows some of the exceptions that AccountOps may signal.

Tabela 3.4: Some exceptions signaled by the AccountOps component.

Exception	**Description**
InvalidAgencyException	The provided agency number does not belong to any valid agency.
InvalidAccountException	The provided account number does not match any of the accounts of the customer.
AlreadyCanceledException	The accounting contract for the provided account has already been canceled.

```
pred dp2() {
 all C: Component | let nonHandled = (C.catches - C.handles)
  | (all CF : C.catchesFrom | #(CF <: nonHandled) > 0 =>
     ((#nonHandled > 0 => #(C.propagates) > 0) &&
      all E: CF.nonHandled |  #(E.(CF.(C.propagates))) > 0))
}
```

Figura 3.20: Alloy specification of property DP2.

Applying Aereal

We created four different exception flow views for the architecture of the financial system. The exception flow views specify the exceptional activity of the system in the execution of use cases Solicitation of checkbooks, Delivery of checkbooks, Cancellation of checks, Retention of checks, and Cancellation of accounting contract. It was not necessary to define new ACME families for this case study because the architecture of the system adheres only to the layered architectural style. The effort necessary to describe the four exception flow views based on the development documentation of the system was approximately 8 developer hours.

Exception flow analysis was performed in two phases. In the first phase, we composed each exception flow view with the architecture description and generated an Alloy specification for each resulting extended architecture description. During exception flow analysis, the AA pointed out some misspellings and omissions in the Alloy specifications. These problems were a consequence of errors in the exception flow views. After fixing the errors, we attempted to verify the Alloy specifications again, but the AA produced counterexamples indicating that the specifications did not satisfy desired property DP2 (Section 3.4.4). Figure 3.20 shows the formal specification of this property in Alloy. The operators `-`, `=>`, and `#` mean set subtraction, logical implication, and set cardinality, respectively, and the declaration `let` associates an alias to an expression. For each component in the Alloy specification, predicate `dp2()` selects all the exceptions that the component catches but does not handle and checks if it propagates them. It is important to stress that the terms "handle" and "propagate" follow the terminology introduced in Table 3.2.

The fact that the Alloy specification does not satisfy property DP2 can be a problem or not, depending on (i) the implementation language of the system, and (ii) the design principles adopted for system implementation. In some languages, such as CLU [118] and Guide [112], a method must handle all the exceptions it catches, even if handling consists in simply re-raising the exceptions. Moreover, there are methodologies [54, 154] that advocate the use of explicit exception propagation even for languages that do not require it, such as Java and C#. In these two cases, a specification of the flow of exceptions in the system should satisfy property DP2. Otherwise, it will not faithfully represent the assumptions made about how the system will be implemented.

In the second phase of exception flow analysis, we composed all the exception flow views with the architecture description. The resulting extended architecture description provides a complete picture of how exceptions flow between architectural elements in the financial system. Moreover, it makes it easier to understand how the exception flow views interfere with one another. Figure 3.21 shows the result of composing the four exception flow views. For simplicity, components and connectors that are not related to the system's exceptional activity are not shown. From the extended architecture description, we generated the Alloy specification, which comprised 9 components, 8 ducts, and 40 exceptions, and employed the AA to perform exception flow analysis. To our surprise, the AA ran out of memory in a few minutes and terminated verification abnormally. We attempted to change some parameters of the AA, allocate more memory for the JVM (the AA is a Java application), and move verification to a PC with twice the amount of RAM memory (from 512MB to 1GB), but the AA still could not reach the end of the verification.

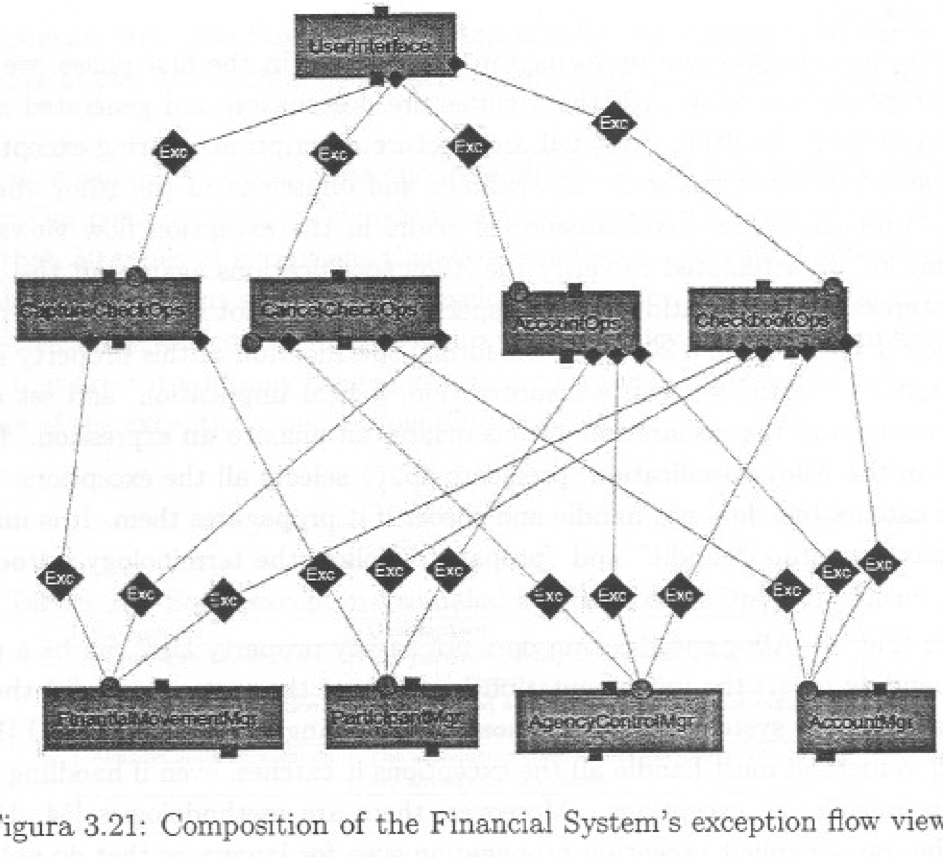


Figura 3.21: Composition of the Financial System's exception flow views.

Discussion

Aereal provided valuable assistance in the task of finding mistakes in the specification of the Financial System's exceptional activity. Although most of these problems are simple to correct, failure to address them can result in problems that are harder to correct in later phases of development. Moreover, it was possible to automatically validate a policy adopted by the EHSs of some programming languages without having to actually implement the system. The results we obtained from this case study and the one described in Section 3.5.1 do not demonstrate the universal usefulness of Aereal in the construction of software systems with strict dependability requirements. They do show, however, that the Aereal approach is useful in some cases, and justify further studies that can provide more substantial evidence.

This case study has shown that scalability is still a limitation of the Aereal approach. For an Alloy specification comprising 9 components, 8 ducts, and 40 exceptions, it was not possible to perform exception flow analysis. This problem does not represent a general trend, as we have successfully conducted five case studies so far and 40 is an unusually large number of exceptions visible at the architectural level. It requires immediate attention, nevertheless, because it was detected during the development of a real application, not a textbook example. Section 3.7 points directions for future work in this area.

3.6 Related Work

3.6.1 Architectural Analysis and ADLs

Several approaches for specifying software architectures so that they are passive to automated analysis have been proposed. Most of them define new ADLs that target specific aspects of a software system. These ADLs are usually based on some underlying formalism that is well-supported by tools. Wright [4] specifications can be translated to CSP and analyzed for deadlock freedom and interface compatibility. Rapide [120] is based on partially-ordered event sets. The language supports simulation of architecture descriptions and analysis of the event patterns produced by components. Darwin [122] is based on π-calculus and can be used to specify dynamic software architectures. Abowd and his coleagues [2] use Z to formalize and compare architectural styles.

ADLs such Wright and Rapide, which target the specification of system behavior, have many interesting features and could be used to analyze exception flow in architecture descriptions. However, these languages fail to address some important requirements. For instance, it is not possible to define a type hierarchy for events in Wright. Hence, it is not possible to check if a component catches exceptions by subsumption[3]. Moreover, Rapide does not support the notion of architectural style. Although the language allows certain styles to be simulated, this is not possible in many commonplace situations [137]. A more general problem is that these ADLs do not separate the specification of a system's normal and exceptional activities. This separation decreases the impact of error recovery mechanisms on the overall complexity of the system [5, 56, 146]. Finally, using these ADLs and associated tools to analyze exception flow in software architectures would require that developers specify the EHS to be supported from the ground up, a cumbersome and daunting task. In this work, we propose a more specific approach for analyzing exception flow in architecture descriptions. On the one hand, it does not present the aforementioned shortcomings. On the other hand, it has a narrower scope and is not intended to be a general-purpose solution for architecture design.

Some works have focused on formally characterizing architectural styles and using them as a basis for analysis of software architectures [76, 165]. Aereal builds upon these works but focuses on the analysis of exception flow at the architectural level. To the best of our knowledge, this is an aspect of software architecture that has not been addressed in the literature.

Approaches that take a formal description of a software architecture as starting point to produce code that preserves reliability-related properties achieved by the architecture have been proposed with different motivations. SADL [167] is an ADL for building hierarchies of architectural descriptions where descriptions located at lower levels of a hierarchy are refinements of descriptions at upper levels. Theorem proving techniques are used to show that refinements are valid and preserve safety, security, and fault tolerance properties. Saridakis and Issarny [155] attempt to characterize the semantics of software architectures from the standpoint of reliability properties. This characterization makes it possible to reuse software architectures in different systems while guaranteeing that the desired reliability properties are maintained in all refinement steps Our work differs from these works because it focuses on the verification of specific properties related to exception flow. These properties can be verified in the context of architectural styles, architectural elements, or whole systems. The aforementioned works emphasize more general fault tolerance properties related to the behavior of a system as a whole. In this sense, our work is complementary to previous efforts. Moreover, Aereal separates the definition of the normal and exception activities of a system, whereas previous works do not make this distinction. Finally, these works do not attempt to model the EHS of existing programming languages. This restricts their applicability because exception handling is usually employed in the implementation of fault tolerance mechanisms.

[3]An exception E is caught by subsumption if it is caught by a catch clause that targets a supertype E' of E.

3.6.2 Exceptions at the Architectural Level

In recent years, several works proposing the use of exception handling at the architectural level to build dependable systems have appeared in the literature. Bass et al [10] report that, during the development of an air-traffic control system, a system with high dependability requirements, it was necessary to devise a new architectural view that explicitly represented the flow of exceptions between components in the system. This view should provide enough information to help developers to understand how the system deals with errors. Section 3.3.1 described this work in detail.

Issarny and Banâtre [97] describe an extension to existing ADLs for specifying global invariants whose violations are called "configuration exceptions". This work emphasizes fault treatment [5] at the architectural level, by means of architecture reconfiguration. The concept of idealized C2 component [56] defines a structure for associating exception handlers to architectural components that adhere to the C2 architectural style [173]. Castor et al [30] have refined the notion of idealized C2 component and proposed an architectural EHS and implementation infrastructure addressing the specific concerns of component-based systems. Unlike the Aereal approach, these works do not provide means for defining how exceptions flow in different architectural styles. Moreover, they do not focus on the description and analysis of exceptions at the architectural level. Although the work on configuration exceptions presents a proposal for extending existing ADLs with information about exceptions, it focuses on a very specific type of exception that is not signaled or handled in the traditional sense (because exceptions do not flow between architectural elements). Hence, it is not straightforward, based on this proposal, to specify what exceptions are signaled or caught by a given component.

Some authors have proposed frameworks [162, 179] that support the detection and handling of exceptions in component-based platforms, such as J2EE. These frameworks provide means to introduce error detection mechanisms, such as monitors, pre, and post-conditions, in components in a non-invasive way. Moreover, they define hotspots that simplify the implementation of handlers and the association of these handlers with components. These implementation infrastructures are complementary to Aereal. They provide a link between architecture description and system implementation when designing for specific architectural styles and component platforms.

3.6.3 Exception Flow Analysis

Several works [64, 150, 156, 186] propose static analyses of source code that generate information about exception flow. Usually, this information consists in the exception propagation paths in a program and is used, for example, to discover uncaught exceptions in languages with polymorphic types, such as ML. Robillard and Murphy [150] present a brief survey of these techniques and tools.

Our approach leverages previous proposals for exception flow analysis, most notably Schaefer and Bundy's [156], but differs in focus. On the one hand, the Aereal approach targets the early phases of development and is broader in scope. It prescribes a conceptual framework for documenting and analyzing the flow of exceptions between architectural elements, and provides mechanisms to integrate this conceptual framework with existing tools for architecture-centric software development and software verification. Furthermore, since the emphasis of Aereal is on the architectural design phase, it explicitly takes into account the concept of architectural style. On the other hand, static analysis tools are employed when an application is already implemented, mainly to find bugs in exception handling code and to improve program understanding.

3.7 Conclusions and Future Work

In this paper we presented Aereal, a framework for analyzing exception flow in software architectures. Aereal works as an adaptable architectural-level EHS, that is, developers can add or remove constraints to the EHS according to their needs. Due to its combination of ACME and a structural approach for representing exception flow, it is possible to define how exceptions are propagated in a style-specific manner. This is a powerful feature since different architectural styles usually impose different constraints on how components communicate.

Developers of systems with strict dependability requirements benefit from using Aereal in the following ways: (i) by better documenting their decisions about exceptions that flow amongst architectural elements; (ii) by making explicit their assumptions about the EHS of the language(s) that will be used during system implementation; (iii) by checking structural constraints imposed by exceptional styles; and (iv) by verifying inconsistencies between the architecture description and the assumed EHS. Moreover, Aereal completely separates the exception flow view of a software architecture from its normal components and connectors view, and provides tools to compose these views automatically. This feature promotes better understandability and maintainability because concerns do not get cluttered in a single architecture description.

There are currently several limitations to the Aereal approach, besides the ones discussed in Section 3.5. The notion of architectural style is based on ACME's and, as such, focuses only on the structure of a system. However, an architectural style involves several other issues [129, 137], for example, communication, control flow, and data flow. Hence, our approach for describing exception flow is effective only as long as it is possible to describe exception flow abstractly in terms of system structure. Also, there are still no automated means for extracting an exception flow view from the implementation of a system. Tools such as Jex [150] provide some help in this task, but do not implement a complete solution and further studies are required to evaluate how productive this approach would be. Improving traceability between code and architecture is an active area of research [1, 81].

The evaluation of Aereal conducted so far indicates directions for future work. First, and most obvious, is improving scalability. We envision two complementary approaches to alleviate the scalability problems we discovered while conducting the Financial System case study. The first is to optimize Aereal's system model by removing redundant information, in order to decrease the amount of RAM memory required to analyze exception flow. For example, ACME properties signals and catches (Section 3.4.2) and their representations in Aereal's system model could be unified into a single abstraction. The two exist only because of historical reasons. The second is to implement a tool that checks if an Alloy specification satisfies all the basic properties of the EHS supported by Aereal. This would drastically reduce the complexity of the checks the AA performs, hence decreasing the amount of memory that verification requires. This change will not compromise the flexibility of the framework, since the basic properties do not change and any valid system must satisfy them.

An interesting issue that arises with the composition of exception flow views and architecture description is what exactly does it mean to adhere to an architectural style. For example, since one of the invariants of the ClientAndServerFam ACME family states that a client has exactly one port, does adding a catcher port to a client component in a Client/Server architecture makes that component conceptually invalid? In terms of ACME invariants, the answer is "yes". In this situation, it is necessary to modify violated invariants of the "normal" style in order for the extended architecture description to be valid. For the invariant above, this would mean stating that a client has exactly one port of type ClientPortT. This modified invariant expresses the intent of the original invariant without being overly restrictive. In the studies we conducted so far (five case studies involving 10 different ACME families, normal and exceptional), this kind of modification was sufficient to accommodate the introduction of exceptional styles. Further evaluation is required, though, to understand if it applies in general.

3.8 Resumo do Capítulo 3

Este capítulo apresentou uma abordagem para tornar mais rigoroso o projeto arquitetural do comportamento excepcional de sistemas de software. Essa abordagem complementa metodologias que prescrevem atividades para o projeto do comportamento excepcional do sistema e é centrada no arcabouço Aereal. Foi ilustrado o uso desse arcabouço para a descrição do comportamento excepcional de um sistema, através da Visão Arquitetural de Fluxo de Exceções, e para a verificação de propriedades relativas à maneira como exceções fluem entre componentes na arquitetura. Os dois estudos de caso realizados com o fim de avaliar a abordagem proposta mostraram seus benefícios, em especial a maneira como ajuda a tornar explícitas as suposições que arquitetos fazem sobre os componentes integrados no sistema, e colocaram em evidência suas limitações. Esse estudo de caso fornece evidência de que a Hipótese de Pesquisa 1 (Capítulo 1, Seção 1.3.1) pode ser válida.

Conforme mencionado anteriormente, Aereal funciona como um mecanismo de tratamento de exceções arquitetural. O arcabouço especifica regras sobre como exceções fluem entre elementos arquiteturais e o que esses elementos podem fazer com elas. Essas regras são descritas de maneira precisa em um modelo, representado na Figura 3.5 pelo oval rotulado *Definition of the Exception Handling Mechanism (in Alloy)*, usado como entrada para a atividade de análise de fluxo de exceções. O próximo capítulo descreve esse modelo em detalhes.

Capítulo 4

Análise de Fluxo de Exceções no Nível Arquitetural

Este capítulo se refere à Seção 1.3.4 do Capítulo 1 e apresenta um modelo genérico que define quais são responsabilidades de cada componente arquitetural, no tocante ao lançamento, recebimento e tratamento de exceções, e como exceções fluem entre esses componentes. O capítulo é uma continuação direta do Capítulo 3 e, consequentemente, tem seu foco na integração dos componentes de software de um sistema.

O modelo apresentado neste capítulo inclui um conjunto de regras formalmente especificadas que descrevem o funcionamento do mecanismo de tratamento de exceções. Esse modelo pode ser mapeado de forma quase direta para linguagens de especificação bem conhecidas, como Alloy [98] e B [3], e suas instâncias descrevem arquiteturas de sistemas de software. Através das ferramentas associadas a essas linguagens, é possível verificar de maneira automática se a arquitetura de um sistema satisfaz diversas propriedades de interesse relativas ao seu comportamento excepcional. Adicionalmente, essas propriedades tornam explícitas as suposições que arquitetos fazem sobre o comportamento excepcional dos componentes em um sistema, por exemplo, quais sãoas exceções que se espera que um determinado componente efetivamente mascare (e não apenas aquelas que ele é capaz de tratar). Este capítulo, em particular, foca no mapeamento do modelo proposto para a linguagem Alloy.

O artigo que este capítulo contém será publicado como um capítulo do livro *Rigorous Development of Complex Fault-Tolerant Systems* [27] da série *Lecture Notes in Computer Science.* Uma versão preliminar [26] foi apresentada no *FM'2005 Workshop on Rigorous Engineering of Fault-Tolerant Systems*, em julho de 2005.

Reasoning About Exception Flow at the Architectural Level

Fernando Castor Filho Patrick Henrique da S. Brito

Cecília Mary F. Rubira

Institute of Computing
State University of Campinas
Campinas - SP - Brasil
{fernando,pbrito,cmrubira}@ic.unicamp.br

4.1 Introduction

Exception handling [52] is a well-known mechanism for structuring error recovery in fault-tolerant software systems. Since exception handling is an application-specific technique, it complements other techniques for improving system reliability, such as atomic transactions [83], and promotes the implementation of sophisticated error recovery measures. Furthermore, in applications where backward error recovery is not possible, such as those that interact with mechanical devices, exception handling may be the only choice available.

Usually, a large part of a system's code is devoted to error detection and handling [52, 148, 182]. However developers tend to focus on the normal activity of applications and only deal with code responsible for error detection and handling at the implementation phase, in an ad hoc manner. Hence, this part of the code is usually the least understood, tested, and documented [52, 148]. To achieve the desired levels of reliability, mechanisms for detecting and handling errors should be developed systematically from early phases of software development [154], starting from requirements, and passing by analysis, architectural design, detailed design, and, finally, implementation.

The concept of software architecture [159] has been recognized in the last decade as a means to cope with the growing complexity of software systems. According to Clements and Northrop [48], software architecture is the structure of the components of a program/system, their interrelationships and principles, and guidelines governing their design and evolution over time. It is widely accepted that the architecture of a software system has a large impact on its capacity to meet its intended quality requirements, such as reliability, security, availability, and performance, amongst others [11, 47]. There are many proposals in the literature [4][76][120] of notations and techniques to formally describe software architectures to show how they achieve specific quality attributes, such as adherence to interaction protocols [4], and architectural styles [76]. These approaches are usually supported by tools that automatically or semi-automatically verify whether an architecture satisfies some previously-defined properties of interest.

An important challenge faced by developers of fault-tolerant systems is to build fault tolerance mechanisms that are reliable. If a system should be reliable and exception handling is one of the mechanisms that can be employed to achieve this goal, it may be beneficial to consider exception handling-related issues before the implementation phase; in particular during architectural design. This idea goes hand-in-hand with the notion that exception handling should be taken into account from early phases of software development, in order to define the exceptional activity of the system. However, to the best of our knowledge, there are currently no approaches for the specification and analysis of exception-related information at the architectural level. As pointed out by Bass and his coleagues [10], specifying how exceptions flow between architectural components is a real problem that appears in the development of systems with strict dependability requirements, such as air-traffic control and financial.

This paper proposes an approach for describing software architectures extended with information about exception flow based on a formal model that supports reasoning about exception flow-related properties of interest. Furthermore, our solution allows one to automatically verify whether an architecture satisfies these properties of interest. The ability to formally specify and verify the flow of exceptions in a system can help in the detection of ambiguities, mistakes, and incompletenesses, thus improving the system's overall reliablity. We present a model for reasoning about exception flow in software architectures. This model specifies the structuring of an architecture in terms of architectural components (loci of computation and data stores) and connectors (loci of interaction), as well as information relative to exception flow amongst these elements. We show how systems adhering to this model can be automatically verified using the Alloy [98] specification language and its associated tool set.

This work is organized as follows. Section 4.2 provides some background on exception handling and the Alloy design language. Section 4.3 describes the overall approach that we propose for extending architecture descriptions with exception flow-related information. Section 4.4 presents the proposed model for reasoning about the flow of exceptions at the architectural level. A mix of informal explanations and set theory notation is employed. Section 4.5 describes how the proposed model can be used to verify exception flow in architecture descriptions. Section 4.6 compares the proposed model with some related research. The last section rounds the paper and points directions for future works.

4.2 Background

4.2.1 Exception Handling

Exception handling [52] is a mechanism for structuring error recovery in software systems so that errors can be more easily detected, signaled, and handled. It is implemented by many mainstream programming languages, such as Java, Ada, C++, and C#. These languages allow the definition of exceptions and their corresponding handlers. The set of exceptions and exception handlers in a system define its abnormal or exceptional activity.

When an error is detected, an exception is generated, or *raised*. If the same exception may be raised in different parts of a program, different handlers may be executed, depending on the place where the exception was raised. The choice of the handler that is executed depends on the exception handling context (EHC) where the exception was raised. An EHC is a region of a program where the same exceptions are handled in the same manner. Each context has an associated set of handlers that are executed when the corresponding exceptions are raised. Typical examples of EHCs in object-oriented languages are blocks, methods, and classes [72]. At the architectural level, contexts are usually defined by architectural components and connectors [30].

The concept of *idealized fault-tolerant component* (IFTC) [5] defines a conceptual framework for structuring exception handling in software systems. An IFTC is a component (in a broader sense; an object, a software component, a whole system, etc.) in which the parts responsible for the normal and abnormal activities are separated and well-defined, within its internal structure. The goal of the IFTC approach is to provide means to structure systems so that the impact of fault tolerance mechanisms in the overall system complexity is minimized. This solution eases the detection and handling of errors. Figure 4.1 presents the internal structure of an IFTC and the types of messages it exchanges with other components in a system.

When an IFTC receives a service request, it produces a *normal response* if the request is successfully processed. If an IFTC receives an invalid service request, it *signals* an *interface exception*. If an error is detected during the processing of a valid request, the normal activity part of the IFTC *raises* an *internal exception*, which is received by the exceptional activity part of the IFTC. If the IFTC is capable of handling an internal exception properly, normal activity is resumed. If the IFTC has no handlers for an internal exception or is unable to handle an exception, it *signals* a failure exception. Interface and failure exceptions are collectively called *external exceptions*. An IFTC might also *catch* external exceptions signaled by other IFTCs and attempt to handle them. In this work, it is assumed that architectural elements behave like IFTCs. Hence, only external exceptions are taken into account, since internal exceptions are encapsulated inside components and connectors.

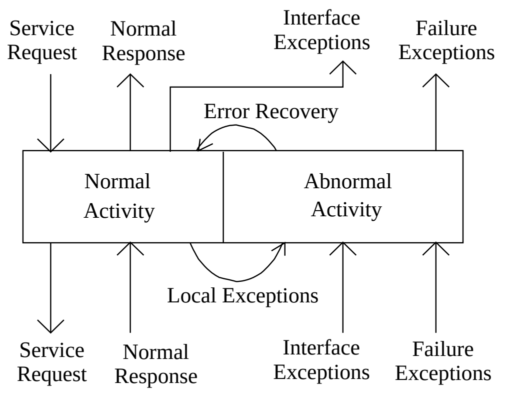


Figura 4.1: Idealized Fault-Tolerant Component.

4.2.2 Alloy

Alloy [98] is a lightweight modeling language for software design. It is amenable to a fully automatic analysis, using the Alloy Analyzer (AA) [100], and provides a visualizer for making sense of solutions and counterexamples it finds. Similarly to other specification languages, such as Z and B [3], Alloy supports complex data structures and declarative models.

In Alloy, models are analyzed within a given scope, or size. The analysis performed by the AA is sound, since it never returns false positives, but incomplete, since the AA only checks things up to a certain scope. However, it is complete up to scope; the AA never misses a counterexample which is smaller than the specified scope. As pointed out by the Alloy tutorial [100], small scope checks are still very useful for finding errors.

4.3 Proposed Approach

The construction of robust fault-tolerant systems requires that developers take fault tolerance-related issues into account since the outset of software development [154]. Our ultimate goal is to devise a general approach for the rigorous development of dependable software systems that use exception handling to implement forward error recovery at the architectural level. This work addresses specifically the issue of verifying properties of interest related to exception flow in software architectures. Our solution is supported by the Aereal [28] framework. This framework aims to assist in documenting, analyzing, and validating exception flow in software architectures.

Figure 4.2 presents a schematic description of our solution. Developers start by performing traditional activities of a software development process, namely, requirements

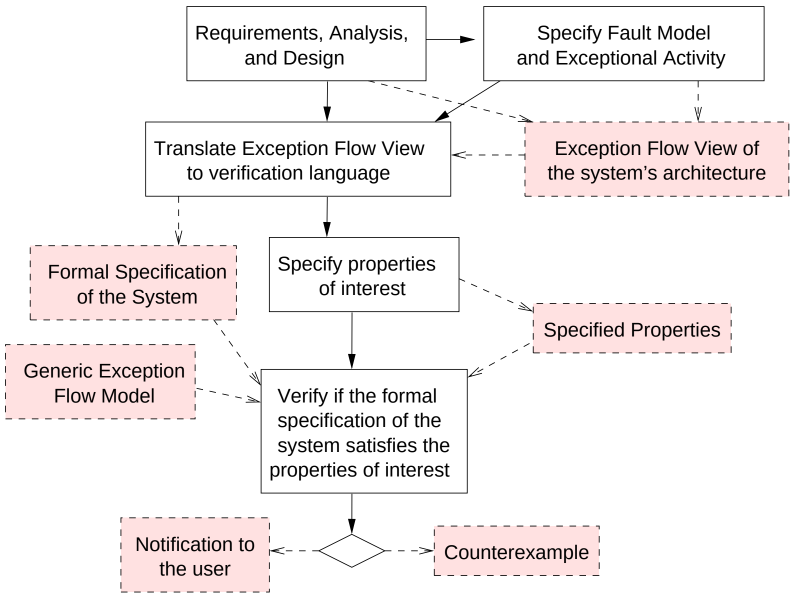

Requirements, Analysis, and Design
Specify Fault Model and Exceptional Activity
Translate Exception Flow View to verification language
Exception Flow View of the system's architecture
Formal Specification of the System
Specify properties of interest
Specified Properties
Generic Exception Flow Model
Verify if the formal specification of the system satisfies the properties of interest
Notification to the user
Counterexample

Figura 4.2: Overview of the proposed approach. White rectangles represent activities and shaded rectangles with dashed borders represent artifacts.

engineering, analysis, and design (both architectural and detailed) of the system. At the same time, they define the scenarios in which the system may fail (fault model), what exceptions correspond to each type of error, and where and how the exceptions are handled (exceptional activity). The specification of the system's fault model and exceptional activity can be conducted as prescribed by some works in the literature [154]. The result of these activities is an architecture description of the system that includes information about the exceptions that can be signaled by each architectural element and what elements are responsible for handling them. We refer to this specialized architecture description as the Architectural Exception Flow View [28]. *Architectural views* represent various aspects of the same architecture and each view shows how the architecture achieves a particular quality attribute [11]. In Aereal, exception flow views are specified using the ACME [76] architecture description language [128] (ADL).

To verify whether the exception flow view exhibits some properties of interest, it is necessary to translate this view to a formal language with adequate support for automated verification (*verification language*). The formal specification produced by this translation must adhere to a generic meta-model which specifies: (i) the elements that can be part of an architecture description; (ii) how exceptions flow amongst these elements; and (iii) how they relate to each other. Hereafter, we call this model *Generic Exception Flow Model*). Both the formal specification and the generic exception flow model are described in the verification language. Currently, we use Alloy to specify the generic exception flow model. A system is verified by providing its formal specification as input to a constraint solver for the verification language, together with the properties to be verified, and the definition of the generic exception flow model. We used the AA tool to verify formal specifications in Alloy. When a property of interest does not hold, the AA produces a counterexample.

In the rest of this paper, we focus on specifying the generic exception flow model. A detailed description of the general approach described in this section is available elsewhere [28].

4.4 Generic Exception Flow Model

In our model, special-purpose architectural connectors model exception flow between components. These connectors, called *exception ducts*, are unidirectional point-to-point links through which only exceptions flow. They are orthogonal to "normal" architectural connectors and do not constrain the way in which the architecture is organized [28]. Mehta and Medvidovic [129] argue that simple point-to-point connections are suitable general-purpose abstractions for modelling communication between architectural components independently of the architectural styles [159] to which an architecture adheres. We use this structural perspective on exception flow because it is intuitive to architects (who are used to thinking in terms of components and connectors), compatible with well-established views on what exception flow is [52], and it does not require the modeling of the complete activity of the application. Architectural elements are assumed to handle only one exception at a time. We address the case where an EHC might catch multiple exceptions concurrently elsewhere [35]. Upon receipt of an exception, the receiving element interrupts its execution and initiates exception handling. This is how exception handling works in most programming languages. Modeling issues such as data- and control-flow is beyond the scope of this work. Furthermore, we assume that the infrastructure supporting the exception handling mechanism (middleware, programming language, etc.) is correct. Therefore, exceptions are always delivered correctly and timely.

It is important to stress that exception ducts are not implementation-level connectors. They are just a high-level abstraction to describe exception flow. Architects often need to understand the architecture of a system from various perspectives, according to specific quality attributes. Since different architectural views target different aspects of an architecture, they usually employ different modeling constructs [109]. Hence, at the implementation level, exception flow is materialized by means of the constructs provided by the underlying programming language or infrastructure. For example, in a publisher/-subscriber architecture, exception flow can be materialized as events flowing through a message bus.

Tabela 4.1: Elements of the proposed exception flow model.

Set	**Description**
Element	The type of all architectural elements. Supertype of *Component* and *Duct*.
Component	The subtype of *Element* of which all components in a system are instances.
Duct	The subtype of *Element* of which all exception ducts in a system are instances.
RootException	The type of which all exceptions in a system are instances.

4.4.1 Representation of Components, Ducts, and Exceptions

In our model, components, exception ducts, and exceptions are represented by objects of a certain type. The proposed model employs a notion of type that is adopted by some modern formal specification languages, such as Alloy [98] and B [3]. Moreover, it is compatible with the notion of types used in OO languages such as Java and C#. A type T is a set of instances and its subtypes $T_1, T_2, ..., T_N$ of T are disjunct subsets of T. Only single inheritance is allowed. An exception is any instance of a type that is subtype of type *RootException*. The same applies to components and exception ducts, and the types *Component* and *Duct*, respectively. In our model, types *Component* and *Duct* are subtypes of *Element* (and collectively called "elements"). Table 4.1 lists the basic elements of the proposed model. The sets in the table can also be seen as unary relations and are, therefore, subject to operations that apply to relations, such as composition.

We represent exceptions as objects, instead of using symbols or global variables, mainly because objects are more flexible and can be used to encode arbitrary information regarding the cause of an exception [72]. Moreover, many large and complex software systems are developed nowadays using object-oriented languages, such as Java and C++, which represent exceptions as objects.

The supertype of all exceptions is called *RootException*, instead of a more usual name, such as *Exception* or *Error*, in order to provide to developers the flexibility to organize exceptions as required, for instance, based on the adopted programming language. For example, considering the EHS of Java, a developer can define at least four exception types: (i) *Throwable*, subtype of *RootException*; (ii) *Exception*, subtype of *Throwable*; (iii) *Error*, subtype of *Throwable*; and (iv) *RuntimeException*, subtype of *Exception*. Application-specific exception types would then be subtypes of one of these types.

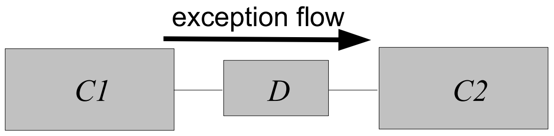


Figura 4.3: A trivial software architecture.

4.4.2 System Structure

We follow the general view of a system configuration as a finite connected graph of components and connectors [128]. We specialize this view, however, so that it can be used to reason about exception flow. In our model, a component is a structural architectural element that catches and/or signals exceptions and an exception duct is a structural element that represents flow of exceptions between two components. The structure of a system is defined in terms of connections between components and exception ducts. The relations $CatchesFrom \in Element \leftrightarrow Element$ and $SignalsTo \in Element \leftrightarrow Element$ specify these connections.

Given an element B, $\{B\}.CatchesFrom$ yields the set of elements that signal exceptions that B catches. Conversely, $\{B\}.SignalsTo$ yields the set of elements that catch exceptions that B signals. The "." operator represents relational composition (or join). Given two relations $A \subseteq T_1 \times T_2 \times ... \times T_n$ and $B \subseteq T_n \times T_{n+1} \times ... \times T_{n+m}$, $A.B$ yields a relation $C \subseteq T_1 \times T_2 \times ... \times T_{n-1} \times T_{n+1} \times ... \times T_{n+m}$. Relation C comprises all the tuples formed by combining tuples from A and B whenever the last element of a tuple from A is the same as the first element of a tuple from B. For example, given $A = \{(e_1, e_2), (e_2, e_3)\}$ and $B = \{(e_2, e_4), (e_2, e_5), (e_3, e_6), (e_7, e_8)\}$, $A.B$ yields $C = \{(e_1, e_4), (e_1, e_5), (e_2, e_6)\}$. Figure 4.3 illustrates relations *CatchesFrom* and *SignalsTo*. The figure depicts two components, $C1$ and $C2$, connected by an exception duct D. Since exceptions flow from $C1$ to C2, passing through D, we can say that $C1 \in \{D\}.CatchesFrom$ and $D \in \{C2\}.CatchesFrom$. Conversely, $D \in \{C1\}.SignalsTo$ and $C2 \in \{D\}.SignalsTo$.

Table 4.2 lists some constraints on relations *CatchesFrom* and *SignalsTo*. These constraints specify properties that a system specification adhering to our model should exhibit. Each one is identified by a name matching the pattern BPX, where "BP" stands for basic property and "X" is a positive integer. Properties $BP1$ and $BP2$ specify that the $CatchesFrom$ relation is not reflexive and it never associates elements of the same type, respectively. Properties $BP3$ and $BP4$ do the same for relation *SignalsTo*. Property $BP5$ states that exception ducts signal exceptions to exactly one element and catch exceptions from exactly one element. If B is a component, $\{B\}.CatchesFrom$ may yield an empty set, in which case B does not catch exceptions. If $\{B\}.SignalsTo$ yields an empty set, exceptions signaled by B, if any, are caught by an implicit component

Tabela 4.2: Contraints on the $CatchesFrom$ and $SignalsTo$ relations.

Property	**Constraint**
$BP1$	$\forall B \in Element \bullet B \notin \{B\}.CatchesFrom$
$BP2$	$\forall B' \in Element \bullet (B, B') \in CatchesFrom \Rightarrow \neg(B \in Duct \wedge B' \in Duct) \wedge \neg(B \in Component \wedge B' \in Component)$
$BP3$	$\forall B \in Element \bullet B \notin \{B\}.SignalsTo$
$BP4$	$\forall B' \in Element \bullet (B, B') \in SignalsTo \Rightarrow \neg(B \in Duct \wedge B' \in Duct) \wedge \neg(B \in Component \wedge B' \in Component)$
$BP5$	$\forall D \in Duct \bullet \lvert\{D\}.CatchesFrom\rvert = 1 \wedge \lvert\{D\}.SignalsTo\rvert = 1$
$BP6$	$\forall B_1, B_2 \in Element \bullet B_1 \in \{B_2\}.CatchesFrom \Rightarrow B_2 \in \{B_1\}.SignalsTo$
$BP7$	$\forall B_1, B_2 \in Element \bullet B_1 \in \{B_2\}.SignalsTo \Rightarrow B_2 \in \{B_1\}.CatchesFrom$
$BP8$	$\forall B \in Element \bullet \{B\}.CathcesFrom \cap \{B\}.SignalsTo = \{\}$
$BP9$	$\forall C_1, C_2 \in Component \bullet C_1 \in \{C_2\}. * ((SignalsTo \cup CatchesFrom).(SignalsTo \cup CatchesFrom))$

$OperatingSystem$. This is useful to model situations in which a system is not capable of handling a certain type of error and fails catastrophically by signaling an exception to the operating system. Properties $BP6$ states that the set of elements from which an element catches exceptions consists of all elements that signal exceptions to it. Property $BP7$ states that the set of elements to which an architectural element signals exceptions consists of all elements that catch exceptions from it, respectively. These two properties provide a link between $CatchesFrom$ and $SignalsTo$. Property $BP8$ specifies that the elements from which an architectural element catches exceptions are different from the ones to which it signals exceptions. For example, a configuration with only two elements, $C \in Component$ and $D \in Duct$, where $C \in \{D\}.CatchesFrom$ and $C \in \{D\}.SignalsTo$ is not valid.

For any valid system in the proposed model, the graph formed by using the components of the system as vertices and the ducts as edges is connected. More formally, let $G = (Component, Duct)$ be a graph, where $Component$ is the set of all vertexes and $Duct$ is the set of all edges. An edge $D \in Duct$ connects two vertexes $C_1 \in Component$ and $C_2 \in Component$ if $D \in \{C_1\}.CatchesFrom$ and $C_2 \in \{D\}.CatchesFrom$. In order for a graph to be connected, there must be a path between any two vertexes. Property $BP9$ specifies this constraint formally. It states that the reflexive transitive closure ($*$ operator) of any component with respect to the relation $(SignalsTo \cup CatchesFrom).(SignalsTo \cup CatchesFrom)$ contains all the other components of the system. In property $BP9$, the $*$ operator yields the set of all components reachable by composing component C_2 with $(SignalsTo \cup CatchesFrom).(SignalsTo \cup CatchesFrom)$ zero or more times.

Tabela 4.3: Contraints on the $PortMap$ relation.

Property	Constraint
$BP10$	$\forall B \in Element \bullet dom(\{B\}.PortMap) = \{B\}.CatchesFrom \wedge$ $ran(\{B\}.PortMap) = \{B\}.SignalsTo$

4.4.3 Exception Interfaces and Exception Handling Contexts

As mentioned in previously, we consider a component to be a structural element that catches and signals exceptions. Exception ducts are similar, but simpler, as they catch exceptions from exactly one component and signal exceptions to exactly one component. A component includes (i) a collection of exception interfaces, which specify the exceptions the component signals; and (ii) a collection of EHCs, which define regions where exceptions are always handled in the same way. Exception interfaces are associated to components by the $SignalsTo$ relation and, for each exception duct in the set $\{C\}.SignalsTo$, there is a corresponding exception interface. The same applies for the $CatchesFrom$ relation and EHCs. This represents the fact that a component may signal different exceptions to (or catch different exceptions from) the various exception ducts to which it is connected. As imposed by property $BP5$ (Table 4.2), each exception duct has exactly one exception interface and one EHC.

Models for reasoning about exception flow at the programming language level do not have an explicit separation between exception interfaces and EHCs. This separation is not necessary because these models focus on fine-grained programming constructs, like methods and procedures, where multiple contexts are associated to a single exception interface. At the architectural level, however, this separation is very important, since a component can have multiple access points (ports) and the latter are explicit in the system description.

The $PortMap \in Element \leftrightarrow Element \leftrightarrow Element$ relation associates exception interfaces and EHCs. When an element catches an exception and does not handle it, the $PortMap$ relation specifies the element to which the exception is signaled, based on the element that originally signaled it. $PortMap$ associates architectural elements to EHCs and exception interfaces. Table 4.3 lists the single property associated to the $PortMap$ relation. Property $BP10$ specifies that for every element B, $PortMap$ associates all the elements from which B catches exceptions to some element to which it signals exceptions and vice-versa.

4.4.4 Exception Flow

Exception flow is specified in terms of five relations: $Generates \in Element \leftrightarrow Element \leftrightarrow RootException$, $Masks \in Element \leftrightarrow Element \leftrightarrow RootException$, $DoesNotMask \in Element \leftrightarrow Element \leftrightarrow RootException \leftrightarrow RootException$, $Catches \in Element \leftrightarrow Element \leftrightarrow RootException$, and $Signals \in Element \leftrightarrow Element \leftrightarrow RootException$. The first two concern the exception interfaces of an architectural element, whereas the last three are related to EHCs. In the rest of this section, we describe these relations in more detail.

The $Signals$ relation defines the exception interfaces of an architectural element. This relation specifies which exceptions an architectural element signals and the elements that catch these exceptions. Let ES be a set of exceptions and B_1 and B_2 be architectural elements such that $B_2 \in \{B_1\}.SignalsTo$. If $\{B_2\}.(\{B_1\}.Signals) = ES$, we say that the element B_1 signals the exceptions in set ES to element B_2. Table 4.4 lists some properties associated to the $Signals$ relation. Property $BP11$ specifies that elements only signal exceptions to elements to which they are connected, as specified by the $SignalsTo$ relation.

Even though, for the sake of uniformity, we have called the second constraint on Table 4.4 a "property", it works more as a definition of the $Signals$ relation. Property $BP12$ states that the $Signals$ relation is derived from three other relations. Intuitively, the set of exceptions that a component signals depends on the exceptions it generates (raises) and on exceptions it catches that were signaled by other architectural elements. $Propagated$ and $Unhandled$ are auxiliary relations defined in terms of the relations that specify a component's EHCs (described in the following paragraphs). The $Generates$ relation specifies the exceptions that components generate when erroneous conditions are detected. These conditions are dependent on the semantics of the application and on the assumed failure model. For reasoning about exception flow, the fault that caused an exception to be raised is not important, just the fact that the exception was raised. Let ES be a set of exceptions and B_1 and B_2 be architectural elements such that $B_2 \in \{B_1\}.SignalsTo$. If $\{B_2\}.(\{B_1\}.Generates) = ES$, we say that the element B_1 raises exceptions ES to B_2. Property BP13 specifies that all the exceptions an element generates to another element are also signaled to the latter. This is coherent with the view that only external exceptions matter at the architectural level.

Exception handling contexts are defined in terms of three relations: $Catches$, $Masks$, and $DoesNotMask$. $Catches$ specifies, for an arbitrary element B, the exceptions B receives from the elements in the set $\{B\}.CatchesFrom$. Let ES be a set of exceptions and B_1 and B_2 be architectural elements such that $B_2 \in \{B_1\}.CatchesFrom$. If $\{B_2\}.(\{B_1\}.Catches) = ES$, we say that the element B_1 catches exceptions ES from element B_2. Table 4.4 shows the basic properties associated with $Catches$, $Masks$, and

Tabela 4.4: Contraints on the $Signals$, $Generates$, $Catches$, $Masks$, and $DoesNotMask$ relations.

Prop.	**Constraint**
$BP11$	$\forall B \in Element \bullet dom(\{B\}.Signals) \subseteq \{B\}.SignalsTo$
$BP12$	$Signals = Generates \cup Propagated \cup Unhandled$
$BP13$	$Generates \subseteq Signals$
$BP14$	$\forall B \in Element \bullet dom(\{B\}.Catches) \subseteq \{B\}.CatchesFrom$
$BP15$	$\forall B \in Element \bullet \forall B' \in \{B\}.CatchesFrom \bullet$ $\{B\}.(\{B'\}.Signals) = \{B'\}.(\{B\}.Catches)$
$BP16$	$\forall B \in Element \bullet dom(\{B\}.Masks) \subseteq \{B\}.CatchesFrom$
$BP17$	$\forall B \in Element \bullet dom((\{B\}.DoesNotMask).RootException) \subseteq$ $\{B\}.CatchesFrom$
$BP18$	$\forall B \in Element \bullet \forall B' \in \{B\}.CatchesFrom \bullet \lvert\{B'\}.(\{B\}.DoesNotMask)\rvert > 0$ $\Rightarrow (dom(\{B'\}.(\{B\}.DoesNotMask)) \cap \{B'\}.(\{B\}.Masks)) = \{\}$

$DoesNotMask$. Propety $BP14$ specifies that elements only catch exceptions from elements to which they are connected, as specified by the $CatchesFrom$ relation. Property $BP15$ states that the exceptions that an element catches are the exceptions signaled to it.

The $Masks$ relation specifies the exceptions that are masked by a component. By "masked", we mean that the component is capable of taking some action that stops the propagation of the exception and makes it possible for the system to resume its normal activity. Modeling the behavior of the exception handlers is beyond the scope of this work. We are just interested in the effect the handler has on the flow of exceptions. Let ES be a set of exceptions and B_1 and B_2 be architectural elements such that $B_2 \in \{B_1\}.CatchesFrom$. If $\{B_2\}.(\{B_1\}.Masks) = ES$, we say that the element B_1 handles exceptions ES from element B_2. Only property $BP16$ in Table 4.4 is directly associated to the $Masks$ relation. This property states that elements only handle exceptions signaled by elements to which they are connected. We could also restrict $Masks$ to be a subset of $Catches$ just like $Generates$ is a subset of $Signals$. We do not impose this restriction, however, because sometimes it is useful to specify general handlers, that is, handlers capable of dealing with any type of exception.

The $DoesNotMask$ relation describes exception handlers that do not stop the propagation of exceptions. These handlers end their execution by signaling the same exception or a new one. $DoesNotMask$ specifies a cause-consequence relationship between an exception that an element catches and an exception that it signals. Let E and E' be exceptions and B_1 and B_2 be architectural elements such that $B_2 \in \{B_1\}.CatchesFrom$.

Tabela 4.5: Properties that define auxiliary relations *Unhandled* and *Propagated*.

Property	**Constraint**
$BP19$	$Propagated = \{\ T \in Element \times Element \times RootException \mid\ s(T) \in \{f(T)\}.SignalsTo \wedge t(T) \in (((\{f(T)\}.PortMap).\{s(T)\}).(\{f(T)\}.Catches \backslash \{f(T)\}.Masks)).(((\{f(T)\}.PortMap).\{s(T)\}).(\{f(T)\}.DoesNotMask))\ \}$
$BP20$	$Unhandled = \{\ T \in Element \times Element \times RootException \mid\ s(T) \in \{f(T)\}.SignalsTo \wedge t(T) \in (((\{f(T)\}.PortMap).\{s(T)\}).(\{f(T)\}.Catches \backslash \{f(T)\}.Masks)) \backslash ((((\{f(T)\}.PortMap).\{s(T)\}).(\{f(T)\}.DoesNotMask)).(\{s(T)\}.(\{f(T)\}.Propagated)))\ \}$

If $\{B_2\}.(\{B_1\}.DoesNotMask) = (E, E')$, we say that the element B_1 **explicitly propagates** (or simply "propagates") exception E' from E, signaled by B_2. The last two properties in Table 4.4 are directly related to the $DoesNotMask$ relation. Property $BP17$ states that an element can only propagate exceptions signaled by an element from which it catches exceptions. Property $BP18$ specifies that an element can handle or propagate the exceptions it catches from another element, but not both.

Now we can go back to the definition of $Signals$ and define $Propagated$ and $Unhandled$. The $Propagated \in Element \leftrightarrow Element \leftrightarrow RootException$ relation specifies the subset of $Signals$ comprising exceptions that are explicitly propagated by an element. It associates an element to the exceptions it propagates explicitly and the elements that catch these propagated exceptions. The $Unhandled \in Element \leftrightarrow Element \leftrightarrow RootException$ relation associates the set of exceptions that an architectural element **implicitly propagates** and the elements to which these exceptions are signaled. An exception is said to be implicitly propagated when an architectural element catches it but the element does handle it or explicitly propagate an exception from it. Such an exception ends up being signaled to some other architectural element. Table 4.5 presents formal definitions for relations $Propagated$ and $Unhandled$. The constraints in Table 4.5 use three auxiliary functions, $f()$, $s()$, and $t()$, that take a triple as argument and return the first, second, and third elements of the triple, respectively.

4.4.5 Exception Propagation Cycles

An exception propagation cycle is a situation where an exception is propagated (implicitly or explicitly) indefinitely, without ever being handled, not even by the special $OperatingSystem$ component, in an architecture that adheres to the basic properties $BP1 - BP20$. Furthermore, exceptions in an exception propagation cycle might potentially have never been raised. For these reasons, valid exception flow views should not have exception propagation cycles.

A conservative way of preventing the occurrence of exception propagation cycles is to completely disallow structural cycles in the graph formed by the components and exception ducts in a system. For most systems, this solution is sufficient without being overly restrictive. However, for software architectures where components are peers, like multi-agent and publisher-subscriber, this approach is not acceptable. At the implementation level, exception propagation cycles are not a problem, since in exception handling mechanisms such as Java's and Ada's, each EHC is kept in the stack, which is finite, and removed from it when controls returns from the EHC (possibly due to exception propagation). Therefore, in a language-level exception propagation cycle, eventually all the EHCs will be removed from the stack and exception propagation will stop. At the architectural level, however, this is not always the case, as an exception is not necessarily implemented as a language-level exception [30, 28].

A simple way to avoid propagation cycles without removing structural cycles is to introduce a partial ordering on exceptions. This ordering could be introduced through additional information associated to each exception type. Then each element would be required to signal exceptions which are greater than the ones it catches. This solution has two shortcomings. The first is that it requires developers to be aware of this ordering, which is not required in any existing programming languages. The second problem is that it complicates the model and could have a negative effect on the performance of verification. Using the type hierarchy to represent the ordering of exceptions is also not adequate, since it also requires developers to be aware of the ordering and thus imposes constraints on the exception type hierarchy. Moreover, from a practical standpoint, this solution partially defeats the purpose of using different types of exceptions, as the topmost layers of a system would always receive general exceptions that do not provide accurate information about the specific error they represent. In the rest of this section, we propose an alternative solution that is a bit more complex, but avoids these shortcomings. We formalize the concept of exception propagation cycle and show how such cycles can be easily detected.

An **exception propagation** is a tuple $\phi = (B, E, E', B')$, with $B, B' \in \mathit{Element}$ and $E, E' \in \mathit{RootException}$. We use the functions $f()$, $s()$, $t()$, and $g()$ to obtain the first, second, third, and fourth elements of a propagation, respectively. Any propagation ϕ must satisfy the following well-formedness predicate:

$$\begin{aligned}(g(\phi) \neq f(\phi)) \wedge (g(\phi) \in \{f(\phi)\}.\mathit{SignalsTo}) \wedge (f(\phi) \in \{g(\phi)\}.\mathit{CatchesFrom}) \wedge \\ (t(\phi) \in \{f(\phi)\}.(\{g(\phi)\}.\mathit{Catches})) \wedge (t(\phi) \in \{g(\phi)\}.(\{f(\phi)\}.\mathit{Signals})) \wedge \\ \exists CF \in \{f(\phi)\}.\mathit{CatchesFrom} \bullet ((CF, g(\phi)) \in \{f(\phi)\}.\mathit{PortMap}) \wedge \\ (s(\phi) \in \{CF\}.(\{f(\phi)\}.\mathit{Catches})) \wedge (s(\phi) \notin \{CF\}.(\{f(\phi)\}.\mathit{Masks})) \wedge \\ (s(\phi) \in \mathit{dom}(\{CF\}.(\{f(\phi)\}.\mathit{DoesNotMask})) \Rightarrow\end{aligned}$$

```
void computePropagations(Element B) {
  foreach catchesP in B.Catches and not in B.Masks {
  // catchesP is an (Element, RootException) pair
    Propagation prop = new Propagation();
    foreach portMapP in B.PortMap such that portMapP.f() == catchesP.f() {
    // portMapP is an (Element, Element) pair
      if(there is a Triple propagatesT in B.DoesNotMask such that
        propagatesT.f() == catchesP.f() && propagatesT.s() == catchesP.s()) {
      // propagatesT is an (Element, RootException, RootException) triple
          prop.t = propagatesT.t();
      } else { prop.t = catchesP.s(); }
      prop.s = catchesP.s();
      prop.f = B;
      prop.g = portMapP.s();
    }
    B.propagations.add(prop);
  }
}
```

Figura 4.4: An algorithm to compute the exception propagations associated to an element.

$$t(\phi) \in \{s(\phi)\}.(\{CF\}.(\{f(\phi)\}.DoesNotMask)))$$

It is easy to compute the set of all exception propagations in a software architecture adhering to our model. Figure 4.4 presents an algorithm for computing all the exception propagations associated to an architectural element.

In order to impose a partial order between two exceptions propagations, we introduce the notion of consecutiveness. Two propagations ϕ_1 and ϕ_2 are said to be **consecutive** if they satisfy the following predicate:

$$\begin{aligned}(g(\phi_1) = f(\phi_2)) \wedge (f(\phi_1) \in \{f(\phi_2)\}.CatchesFrom) \wedge \\ (f(\phi_2) \in f(\phi_1).SignalsTo) \wedge (f(\phi_1) \neq f(\phi_2)) \wedge (t(\phi_1) = s(\phi_2)) \wedge \\ (t(\phi_1) \in \{f(\phi_1)\}.(\{f(\phi_2)\}.Catches)) \wedge (s(\phi_2) \in \{f(\phi_2)\}.(\{f(\phi_1)\}.Signals))\end{aligned}$$

In this case, ϕ_1 is said to be the predecessor of ϕ_2 and ϕ_2 is the successor of ϕ_1. We indicate that propagations ϕ_1 and ϕ_2 are consecutive with the notation $\phi_1 \rightharpoonup \phi_2$. A **sequence of propagations** φ of length n is a set of propagations $\phi_1, \phi_2, \phi_3 ..., \phi_{n-1}, \phi_n$ such that $\phi_1 \rightharpoonup \phi_2, \phi_2 \rightharpoonup \phi_3, ..., \phi_{n-1} \rightharpoonup \phi_n$. For simplicity, we assume that all sequences of propagations are finite. A sequence of propagations $\varphi = \phi_1, \phi_2, ..., \phi_n$ forms an **exception propagation cycle** iff $\phi_n \rightharpoonup \phi_1$.

To verify if a software architecture has exception propagation cycles, it is necessary to build the directed graph formed by all the exception propagations of the architecture. This graph is constructed in two steps: (1) compute the exception propagations for all the elements in the architecture and use them as vertexes; and (2) create a directed edge between two propagations whenever they are consecutive, from the predecessor to the sucessor. Detecting exception propagation cycles in the resulting graph is only a matter of using a regular algorithm for finding cycles in directed graphs.

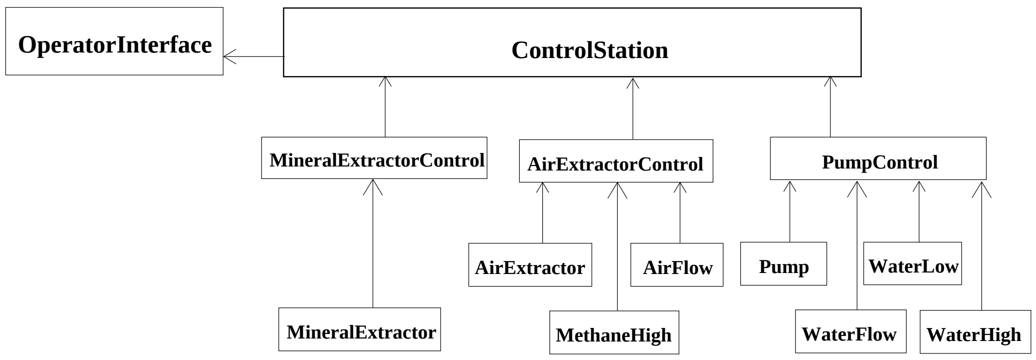


Figura 4.5: Layered architecture of a mining control system.

4.5 Materializing the Model

We have translated the generic exception flow model described in Section 4.4 to Alloy. In this section, we show how to specify systems based on this model and how to verify them. We use a well-known textbook example [163] to make the explanation more concrete. Addressing both the earlier (requirements definition and analysis) and later (detailed design, implementation, etc.) phases of software development is outside the scope of this paper.

Figure 4.5 shows the components and connectors view of a control system for the mining environment [154]. Rectangles represent architectural components and arrows represent exception flow. 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. The 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. For safety reasons, the extraction of minerals should be interrupted when the amount of methane in the atmosphere exceeds a safety limit. The air extractor control station 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 controlled from the surface via an operator console.

The system can fail in several ways. For simplicity, we only consider the case where the AirExtractorControl component fails by signaling the exception `AirExtractorOffException`. This exception is caught and handled by the `ControlStation` component. The handler ends its execution by signaling the exception `EmergencyException`. A detailed description of the exceptional activity of the mining system is available elsewhere [154].

Figure 4.6 shows the Alloy specification of the mining system. In Alloy, a signature (`sig` keyword) specifies a type. The `one` keyword indicates that a signature has exactly

```
 1 open ExceptionHandlingSystem
 2 one sig AirExtractorOffException, EmergencyException extends RootException{}
 3 one sig ControlStation, OperatorInterface, AirExtractorControl
 4     extends Component{}
 5 one sig CS_OI, AEC_CS extends Duct{}
 6 ...
 7 fact SystemStructure{ ...
 8   ControlStation.CatchesFrom = AEC_CS
 9   ControlStation.SignalsTo = CS_OI
10   AEC_CS.CatchesFrom = AirExtractorControl
11   AEC_CS.SignalsTo = ControlStation
12 } fact ExceptionFlow{ ...
13   AirExtractorControl.Signals = AEC_CS->AirExtractorOffException
14   AirExtractorControl.Generates = AEC_CS->AirExtractorOffException
15   AEC_CS.Catches = AirExtractorControl->AirExtractorOffException
16   AEC_CS.Signals = ControlStation->AirExtractorOffException
17   ControlStation.Catches = AEC_CS->AirExtractorOffException
18   ControlStation.Signals = CS_OI->EmergencyException
19   no ControlStation.Masks
20   ControlStation.DoesNotMask =
21       AEC_CS->AirExtractorControlOffException->EmergencyException
22 } fact PortMap{ ...
23   ControlStation.PortMap = AEC_CS->CS_OI
24 }
```

Figura 4.6: Partial Alloy specification of the mining control system.

one instance. We use signatures for modeling structural elements and exceptions. The `open` clause (Line 1) imports the definitions of the basic types of the generic exception flow model, `Element`, `Component`, `Duct`, and `RootException`. It also imports the predicates that specify the basic properties defined in Section 4.4. The relations defined in Section 4.4, such as *CatchesFrom*, *Masks*, etc., are explicitly instantiated by means of facts, predicates that the AA must assume to be true when evaluating constraints. For instance, the fact `SystemStructure` (Line 7) states, among other things, that component `ControlStation` catches exceptions from the exception duct `AEC_CS`. The latter connects `ControlStation` to the `AirExtractorControl` component. Moreover, the fact `ExceptionFlow` (Line 12) states that component `ControlStation` catches the exception `AirExtractorControl` (Line 17), signaled by the `AEC_CS` duct, and signals exception `EmergencyException` to the `CS_OI` duct (Line 18). `ControlStation` translates the former exception to the latter (Lines 20 and 21).

Verification consists in checking whether the Alloy specification of a system satisfies Alloy predicates corresponding to properties of interest. The properties of interest that a system must satisfy are split in three categories: basic, desired, and application-specific. Basic properties define the well-formedness rules of the model, that is, the characteristics of valid systems. These properties specify the functioning of the exception handling mechanism and how software architectures are structured. We have formally specified all the basic properties of the generic exception flow model in Section 4.4. Desired properties are general properties that are usually considered beneficial, although they are not part of the basic exception handling mechanism. They assume that the basic properties hold. Some examples are the following.

DP1. Architectural elements do not handle exceptions they do not catch.
DP2. All the exceptions caught by an architectural element are handled by it, even if some of its handlers end their execution by raising exceptions.
DP3. No unhandled exceptions.

Application-specific properties are rules regarding the flow of exceptions in a specific application. For the mining system, a possible application-specific property is one which guarantees that the OperatorInterface component does not receive domain-specific exceptions.

AP1. No architectural element signals to the OperatorInterface component an exception different from `EmergencyException`.

The Alloy definition of the generic exception flow model includes the specifications of several basic and desired properties that can be used "as-is". Developers only specify additional desired properties and application-specific properties, if any. The AA is employed to analyze exception flow. If a property of interest is violated, the AA generates a counterexample with a configuration of the system where the violated property does not hold. Otherwise it notifies the user that the system is valid.

Figure 4.7 defines four Alloy predicates named `bp13`, `dp2`, and `ap1`, formally specifying properties $BP13$, $DP2$, and $AP1$, respectively. Alloy predicates are logic sentences that must be checked by the AA. In the body of the predicates, `Generates`, `Signals`, `Catches`, `DoesNotMask`, `Masks`, and `CatchesFrom` are names of relations corresponding to the homonymous relations described in Section 4.4. Predicate `bp13()` states that the set of exceptions that a component raises is a subset of the exceptions it signals. Predicate `dp2()` selects, for each component in the Alloy specification, all the exceptions that the component catches but does not handle, and checks whether exceptions are propagated from them. The operators `all`, `<:`, `&&`, and `in` represent, respectively, universal quantification, domain restriction, logical conjunction, and subset. The operators `-`, `=>`, and `#` mean set subtraction, logical implication, and set cardinality, respectively, and the declaration `let` associates an alias to an expression. Predicate `ap1()` is a direct translation from the informal description of property $AP1$.

4.6 Related Work

Several works propose static analyses of source code that generate information about exception flow. Usually, this information consists in the exception propagation paths in

```
/* Basic property BP13 */
pred bp13() {  all C : Component | (C.Generates in C.Signals)  }
/* Desired property DP2 */
pred dp2() {
 all C: Component | let nonHandled = (C.Catches - C.Masks)
  | (all CF : C.CatchesFrom | #(CF <: nonHandled) > 0 =>
     ((#nonHandled > 0 => #(C.DoesNotMask) > 0) &&
      all E: CF.nonHandled |  #(E.(CF.(C.DoesNotMask))) > 0))
}
/* Application-specific property AP1 */
pred ap1() {
 all D: OperatorInterface.CatchesFrom |
   OperatorInterface.(D.Signals) = EmergencyException
}
```

Figura 4.7: Alloy specifications of properties $BP13$, $DP1$, $DP2$, and $AP1$.

a program and is used, for example, to identify uncaught exceptions in languages with polymorphic types, such as ML. Chang et al [38] present a set-based static analysis of Java programs that estimates their exception flows. This analysis is used to detect too general or unecessary exception specifications and handlers. Yi [186] proposes an abstract interpretation that estimates uncaught exceptions in ML programs. Fähndrich [64] and coleagues have employed their BANE toolkit to discover uncaught exceptions in ML. Schaefer and Bundy's [156] work describes a model for reasoning about exception flow in Ada programs. This model is used by a tool that tracks down uncaught exceptions and provides exception flow information to programmers. The JEX tool, proposed by Robillard and Murphy [150], analyzes exception flow in Java programs. The tool includes a GUI to display a program's exception propagation paths and detects handlers that are too general.

Our approach leverages previous proposals for exception flow analysis, most notably Schaefer and Bundy's [156], but it differs in focus. Out approach targets the early phases of development and is broader in scope. It describes how an architectural-level exception handling mechanism works and leverages existing verification tools to check for adherence to the rules prescribed by this mechanism. Furthermore, it supports the definition of new properties of interest and their automated verification. Moreover, as mentioned in Section 4.4.5, existing exception flow models do not take exception propagation cycles into consideration, as they are not a problem that occurs at the implementation level.

Jiang and coleagues [103] describe an approach for the analysis of exception propagation based on a data structure called exception propagation graph. The goal of the authors is to use exception propagation graphs as a basis for automatically generating structural tests. They do not address exception propagation cycles, as their work focuses on implementation-level exception flow analysis. Moreover, they do not show how the proposed approach can be employed to check whether a system exhibits some properties of interest, such as absence of useless handlers.

Several approaches for specifying software architectures so that they are passive to automated analysis have been proposed. Most of them define new ADLs that target specific aspects of a software system. These ADLs are usually based on some underlying formalisms that are well-supported by tools. Wright [4] specifications can be translated to CSP and analyzed for deadlock freedom and interface compatibility. Rapide [120] is based on partially-ordered event sets. The language supports simulation of architecture descriptions and analysis of the event patterns produced by components. We do not propose a new ADL. Instead, we use an existing ADL which supports extension, ACME, and a formal design language, Alloy, to specify and analyze exception flow at the architectural level. To the best of our knowledge, no ADLs currently available focus on the verification of properties related to exception flow.

In a previous work, Castor and coleagues [26] described an initial version of the model presented in this paper. This early work does not unify the definitions of components and ducts and is harder to use and less scalable. Furthermore, it does not take exception propagation cycles into account.

4.7 Concluding Remarks

This paper presented a model for reasoning about the flow of exceptions at the architectural level. This model is part of the Aereal framework and supports the specification of several properties of interest related to exception flow. We have described how systems adhering to it can be automatically analyzed using the AA, in order to verify whether they exhibit these properties. The main contributions of this paper are: (i) a formalization of exception flow in terms of elements that make sense at the architectural level; and (ii) a decomposition of this formalization in terms of a set of properties that can be easily verified through existing tools.

In another study [28], we assessed the scalability of the proposed model. We discovered that, for software architectures with a large number of exceptions (30+), it does not scale up well. Hence, our most immediate future work is to improve scalability. We envision two complementary approaches. The first is to optimize the system model by removing redundant information. The second is to implement a tool that checks if an Alloy specification satisfies all the basic properties of the EHS supported by Aereal. This would drastically reduce the complexity of the checks the AA performs, hence decreasing the amount of memory that verification requires. This change will not compromise the flexibility of the framework, since the basic properties do not change and any valid system must satisfy them.

4.8 Resumo do Capítulo 4

Este capítulo propôs um modelo que fornece um embasamento teórico para a abordagem proposta no Capítulo 3. Esse modelo especifica as responsabilidades dos elementos arquiteturais, no tocante ao tratamento e lançamento de exceções, e regras que estabelecem como exceções podem ser propagadas entre esses elementos. Através da materialização do modelo proposto na linguagem Alloy, torna-se possível verificar se arquiteturas aderem às regras que ele estabelece. Adicionalmente, desenvolvedores podem especificar propriedades adicionais, tanto gerais quanto específicas de aplicação, e verificá-las automaticamente usando o Alloy Analyzer.

O modelo apresentado neste capítulo se aplica a sistemas nos quais vigora a suposição de que erros são independentes. Consequentemente, é possível tratar cada exceção como se fosse a única em determinado instante de tempo. O próximo capítulo lida com sistemas que são concorrentes e cooperativos, o que quer dizer que os diversos componentes integrados no sistema interagem a fim de alcançar um objetivo comum e funcionam de maneira assíncrona, em *threads* separadas. Para tais sistemas, a suposição de independência de erros não é válida. Consequentemente, o tratamento de erros deve ser cooperativo e envolver todas as unidades de computação participantes, assim como o comportamento normal do sistema.

Capítulo 5

Verificação de Tratamento Coordenado de Exceções

A popularização de infra-estruturas para comunicação por eventos em sistemas baseados em componentes, como o *IBM Websphere MQ* [96] e a plataforma *Java Entreprise Edition* com seus *Message-Driven Beans* [92], exige que abordagens para a integração confiável de componentes de software contemplem os casos em que componentes se comunicam assincronamente e cooperam com o fim de atingir um objetivo comum. Conforme mencionado anteriormente, em sistemas com essa característica, chamados de concorrentes cooperativos, não é possível presumir que a manifestação simultânea de múltiplos erros decorre de falhas distintas.

Este capítulo se refere à Seção 1.3.5 do Capítulo 1 e estende algumas das idéias descritas nos Capítulos 3 e 4 e apresenta uma abordagem para a especificação e verificação de arquiteturas concorrentes e cooperativas que usam tratamento coordenado de exceções para estruturar mecanismos de tolerância a falhas. A abordagem é baseada em um modelo genérico que descreve como contextos de tratamento de exceções são organizados e como funcionam o tratamento, resolução e propagação de exceções nesse modelo de sistema. O modelo pode ser traduzido para linguagens de especificação bem amparadas por ferramentas de verificação, como Alloy e B, e é possível verificar diversas propriedades interessantes a partir de aplicações baseadas nele. A utilidade da abordagem proposta para encontrar falhas de projeto é demonstrada através de um estudo de caso.

O artigo que este capítulo contém foi apresentado no *21st ACM Symposium on Applied Computing* [35].

Verification of Coordinated Exception Handling

Fernando Castor Filho[1] Alexander Romanovsky[2]

Cecília Mary F. Rubira[1]

[1]Institute of Computing
State University of Campinas
Campinas - SP - Brasil
{fernando,cmrubira}@ic.unicamp.br

[2]School of Computing Science
University of Newcastle upon Tyne
Newcastle upon Tyne - UK
{alexander.romanovsky}@newcastle.ac.uk

5.1 Introduction

Applications that can cause risks for human lives or risk of great financial losses are usually made fault-tolerant [5], so that they are capable of providing their intended service, even if only partially, when faults occur. Fault-tolerant systems include mechanisms for detecting errors in their states and recovering from these errors. There are two main types of error recovery [5]: backward error recovery (based on rolling the system back to the previous correct state) and forward error recovery (which involves transforming the system into any correct state). The former usually uses either diversely implemented software or simple retry; the latter is typically application-specific and relies on an exception handling mechanism [52].

Usually, a large part of the system code is devoted to error detection and handling [52, 182]. However, since developers tend to focus on the normal activity of applications and only deal with the code responsible for error detection and handling at the implementation phase, this part of the code is usually the least understood, tested, and documented [52]. In order to achieve the desired levels of reliability, mechanisms for detecting and handling errors should be developed systematically from the early phases of development [154]. Ideally, the construction of system fault tolerance mechanisms should follow a rigorous or formal development methodology [12].

Error recovery in concurrent and distributed systems is known to be complicated by various factors, such as high cost of reaching an agreement, absence of a global view on the system state, multiple concurrent errors, difficulties in ensuring error isolation, etc. These systems require special error recovery mechanisms that suit their main characteristics. As it is not possible to develop general error recovery mechanisms applicable for all types of concurrent and distributed systems, two classes of techniques were developed to support recovery in competing and cooperating concurrent and distributed systems [184]. Distributed transactions [83] and atomic actions [20] are well-known examples of techniques for structuring competing and cooperative fault-tolerant distributed systems, respectively. In the rest paper, we concentrate mainly on cooperative distributed systems.

The Coordinated Atomic (CA) Actions concept [184] results from combining distributed transactions and atomic actions. Atomic actions are used to control cooperative concurrency and to implement coordinated exception handling [20] whilst distributed transactions are used to maintain the consistency of the resources shared by competing actions. CA actions function as exception handling contexts for cooperative systems and exceptions raised in an action are handled cooperatively by all the action's participants. If two or more exceptions are concurrently raised, an *exception resolution mechanism* [20] is employed to find an exception to be handled that represents all the exceptions raised concurrently (a *resolved exception*). Many case studies [152, 185] have shown that CA actions are a powerful and useful tool for structuring large, distributed fault-tolerant systems.

In order for CA actions to be applicable for the construction of complex, real-world systems with strict dependability requirements, software development based on CA actions has to be supported by rigorous models, techniques, and tools. Several approaches have been proposed for formalizing the CA action concept with the intention either to give a more complete and rigorous description of the concept [177] or to verify systems designed using CA actions [185]. However, an important aspect of CA actions that has not been addressed by existing work is coordinated exception handling. This is surprising, since exception handling complements other techniques for improving reliability, such as atomic transactions, and promotes the implementation of specialized and sophisticated error recovery measures. Moreover, in distributed applications where a rollback is not possible, such as those that interact with the environment, exception handling may be the only choice available.

Some authors [17] claim that mechanisms for involving multiple participants in coordinated exception handling are difficult to both implement and use. However, we believe that programmers will make more mistakes in an ad hoc implementation of coordinated exception handling than in applying the well-defined mechanisms that general frameworks such as CA actions provide. Therefore, techniques and tools that mitigate the inherent complexity of coordinated exception handling and help developers in the specification and design of systems that make use of this feature are required.

In this paper, we examine the problem of specifying CA action-based designs in a way that allows us to verify automatically if these designs exhibit certain properties of interest regarding coordinated exception handling. Moreover, since coordinated exception handling is strongly related with action structuring, it is also necessary to model how CA actions are nested and composed [152] to define multiple exception handling contexts. We present an approach to modeling CA action-based designs that makes it possible to verify these designs automatically using a constraint solver. The main component of the proposed approach is a formal model of CA actions that specifies the structuring of a system in terms of actions, as well as information relative to exception flow amongst these actions. This model can be directly specified using well-known specification languages, like Alloy [98] and B [3], and verified automatically using the tool sets associated with these languages. With the proposed approach, it is possible to check whether a CA-action based design satisfies several properties of interest.

This paper is organized as follows. Section 5.2 provides some background on CA actions and the Alloy design language. Section 5.3 presents an overview of the proposed approach, including some of the properties that it helps verifying. Section 5.4 presents a case study to illustrate the feasability and usefulness of the proposed approach. The last section rounds the paper.

5.2 Background

5.2.1 Coordinated Atomic Actions

A CA action is designed as a set of roles cooperating inside it and a set of resources accessed by them. An action starts when its roles are taken by participants (e.g. processes, threads, active objects, etc.). In the course of the action, participants can access resources that have ACID (atomicity, consistency, isolation, durability) properties. Action participants either reach the end of the action and produce a normal outcome or, if one or more exceptions are raised within the action, they all are involved in coordinated handling. If handling is successful the action completes by producing a normal outcome, but if handling is not possible then all responsibility for recovery is passed to the containing action where an external action exception is signaled.

The CA action scheme enforces a clear difference between *internal exceptions*, which are raised in the action and have handlers inside the action, and *external exceptions*, which are signaled outside the action when the action cannot deliver the results. The latter is used to report partial action outcomes, abort effect, failure to achieve a consistent result by action participants, etc. Internal exceptions are encapsulated in the action, whereas external ones are visible in the action interface as they have to be dealt with by the

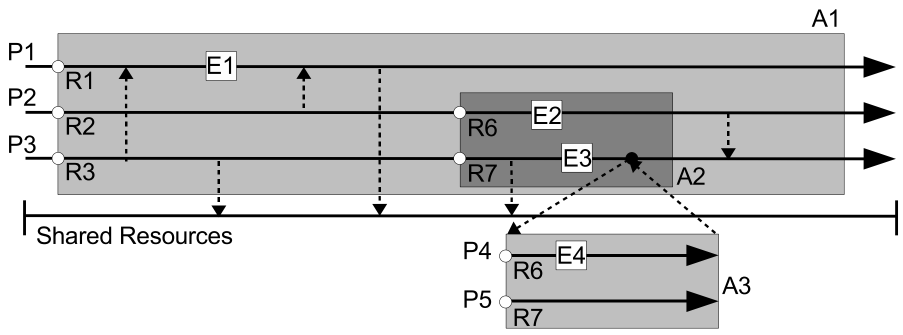


Figura 5.1: A CA action example.

containing action. When several exceptions are concurrently raised in a CA action, an exception resolution mechanism is used.

Figure 5.1 shows a graphical representation of a trivial system structured using CA actions. Top-level CA action A1 has three roles performed by participants P1, P2, and P3. Participants P2 and P3 also perform roles R4 and R5, respectively, in the *nested CA action* A2. Role R5 of A2 spawns *composed CA action* A3 at some point in time after the completion of A2. R5 is interrupted from the moment A3 starts until it completes. The internal exceptions of an action are represented by small squares (labeled E1, E2, E3, and E4 in the figure). Each exception is placed near the role that raises it.

5.2.2 Alloy Design Language

Alloy [98] is a lightweight modeling language for software design. It is amenable to a fully automatic analysis, using the Alloy Analyzer (AA), and provides a visualizer for making sense of solutions and counterexamples it finds. Alloy is based on first-order relational logic and, similarly to other specification languages, such as Z, Alloy supports complex data structures and declarative models. In this work, we use Alloy to formally specify CA action-based designs.

In Alloy, models are analyzed within a given scope, or size (the maximum number of instances of a type). The analysis performed by the AA is sound, since it never returns false positives, but incomplete, since the AA only checks things up to a certain scope. However, it is complete up to scope; the AA never misses a counterexample which is smaller than the specified scope.

5.3 Proposed Approach

The construction of robust fault-tolerant systems requires that developers take fault tolerance-related issues into account since the early phases of development. Our ultimate goal is to devise a general approach for the rigorous development of dependable distributed systems that use both cooperative and competitive concurrency.

This work addresses specifically the issue of verifying properties of interest related to system structuring and coordinated exception handling in CA action-based designs. The following sections present an overview of the proposed approach (Section 5.3.1) and briefly describe some of the properties that can be verified using this approach (Section 5.3.2).

5.3.1 Overview

Figure 5.2 presents a schematic description of the proposed approach for verifying CA action-based designs. Developers start by performing traditional activities of a software development process, namely, analysis and architectural design of the system, assuming that the system is concurrent and cooperative. At the same time, they define the scenarios in which the system may fail (fault model), what exceptions correspond to each type of error, and where and how the exceptions are handled (exceptional activity). The specification of the system's fault model and exceptional activity can be conducted as prescribed by some works in the literature [154]. The result of these activities is a CA action-based design of the system that includes a description of the exceptions that can be raised in each CA action and how they are handled. This design is usually described in a modeling language for CA actions (or simply *modeling language*), for example, informal diagrams (as presented in Section 5.2.1), the Coala [177] formal language, or the FTT-UML [86] profile for the UML.

To verify the CA action-based design, it is necessary to translate it to a formal language with adequate support for automated verification (*verification language*). If the modeling language has a well-defined semantics, like Coala, this translation can be completely automated by a tool. The translation can also be automated for informal notations, like UML profiles, but only partially. Usually, some manual intervention is required to resolve ambiguities. Developers used to formal methods can write the system descriptions directly in the verification language. The choice of using one or two specification languages is based solely on usability issues.

The formal specification produced by translating the CA action-based design to the verification language must adhere to a generic CA actions meta-model specifying the elements of CA actions and how they relate (hereafter called *generic CA actions model*). Table 5.1 lists the elements of the generic CA actions model. The elements of the generic CA actions model are Action, Role, Participant, and RootException. They are the main

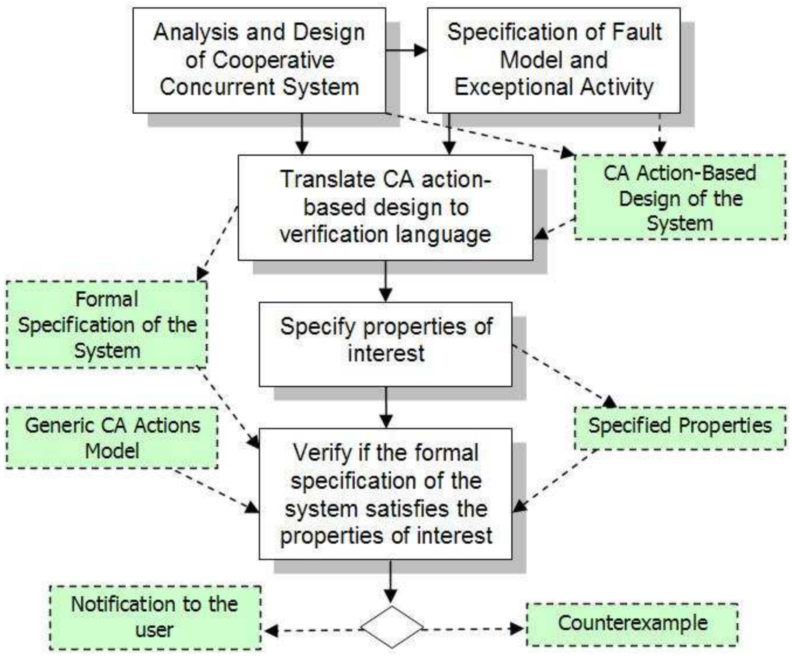

Analysis and Design of Cooperative Concurrent System
Specification of Fault Model and Exceptional Activity
Translate CA action-based design to verification language
CA Action-Based Design of the System
Formal Specification of the System
Specify properties of interest
Generic CA Actions Model
Specified Properties
Verify if the formal specification of the system satisfies the properties of interest
Notification to the user
Counterexample

Figura 5.2: Overview of the proposed approach. White rectangles represent activities and shaded rectangles with dashed borders represent artifacts.

concepts used in the definition of CA actions. Some of them, like Action and Role, include additional information represented through relations. For example, the set of roles of an action is defined by the Roles relation, which associates actions to their respective roles. Both the formal specification and the generic CA actions model are described in the verification language. Up to now, we specified generic CA actions models using B and Alloy as verification languages [34]. Developers can use either of them to formalize CA action-based designs.

A system is verified by providing its formal specification as input to a constraint solver for the verification language, together with the properties to be verified. We used the AA and ProB [115] constraint solvers to verify formal specifications in Alloy and B, respectively. If any of the properties of interest does not hold, the constraint solver produces a counterexample. Both constraint solvers, besides generating a counterexample, include a graphic visualizer that provides additional help in the identification of the problem. We used the two languages just to show that the proposed approach is not specific to a single specifrication language. In the rest of the paper, we focus on the last two activities of Figure 5.2, the ones directly related to system verification.

Element	**Description**
Action	Type that defines actions.
Role	Type that defines roles of actions.
Participant	Type that defines participants, units of computation that perform roles.
RootException	Type that defines exceptions.

Tabela 5.1: Basic elements of the proposed model. Each element is a type whose instances are used to specify CA action-based systems.

5.3.2 Properties of Interest

The properties of interest that a system must satisfy are split in three categories: basic, desired, and application-specific. Basic properties define the well-formedness rules of the model, the characteristics of valid CA actions. They specify the coordinated exception handling mechanism and how actions are organized. Examples of basic properties are presented below, stated informally.

BP1. If a participant performs a role in a nested action, it must also perform some role in the containing action.

BP2. No cycles in action nesting.

BP3. The exception resolution mechanism of an action resolves all possible combinations of concurrent internal exceptions, unless explicitly stated otherwise.

Desired properties are general properties that are usually considered beneficial, although they are not part of the basic mechanism of CA actions. In general, they assume that the basic properties hold. Some examples are the following:

DP1. Top-level CA actions have no external exceptions.

DP2. All internal exceptions of an action are handled in it (i.e. they are either masked or propagated).

DP3. Any role of an action has masking handlers for all of the action's internal exceptions, including all resolved ones.

Application-specific properties are rules regarding the flow of exceptions in a specific CA action-based application. Section 5.4.2 presents an example of application-specific property. The generic CA actions models we have specified so far include the specifications of several basic and desired properties that can be used "as-is". Developers only specify additional desired properties and application-specific properties, if any.

As mentioned in Section 5.3.1, properties of interest are specified in the verification language. The following snippet presents a formal specification for property BP1 in Alloy.

```
predicate parts_ok() { (all A:Action|
  (all NA:A.NestedActions| all NAR:NA.Roles|
  !(all P:Participant|!(NAR in P.RolesPlayed
   && some (P.RolesPlayed & A.Roles)))))
}
```

This snippet defines an Alloy predicate named `parts_ok`. Alloy predicates are logic sentences that must be checked by the AA. In the body of the predicate, `Roles`, `NestedActions`, and `RolesPlayed` are names of some relations that associate information to the elements of the system (actions, participants, etc.) and the "." operator is a generalized form of relational composition. For example, `A.NestedActions` yields the set of actions nested within action `A`, assuming that `A` $\in$ `Action`, where `Action` is a type (a set or, more generally, unary relation), and `NestedActions` is a relation associating actions to their nested actions. Predicate `parts_ok` states that every role of every nested action is performed by some participant that also performs some role in the enclosing action. The operators `all`, `some`, `!`, `&&`, and `&` represent, respectively, universal quantifier, existential quantifier, logical negation, logical conjunction, and set intersection.

The following snippet shows a formal specification in Alloy for property DP1. It states that all actions that are not nested within some other action and not composed by some role have no external exceptions. Operator `=>` represents logical implication.

```
 all A1:Action | ((all A2:Action |
  !(A1 in A2.NestedActions)) && (all R:Role |
    !(A1 in R.ComposedActions))) => (no A1.External)
```

5.4 Case Study

The Fault-Tolerant Insulin Pump Therapy [21] (FTIPT) is a control system with strict reliability requirements for treating patients with diabetes. This system is based on the Continuous Subcutaneous Insulin Injection technique [21] and involves several sensors and actuators that must function concurrently and continuously. These sensors and actuators are wearable devices put on by patients under treatment. The dose of insulin administered by the system includes two types of insulin: rapid action insulin (RAI) and long action insulin (LAI).

Sensors and actuators exchange information by means of wireless communication channels. Sensors send information about the vital signs of a patient to a server located in a hospital. The latter forwards this information to a doctor who defines the amount of insulin to inject. The server then communicates with the actuators which use pumps to administer the established dose of insulin.

Both sensors and actuators may fail. Sensors can fail by stopping to send information about a patient's vital signs. However, when they do send information, the latter is assumed to be correct. Actuators can also fail in the same manner. Moreover, they may fail because there is not enough insulin to apply the required dose. Whenever an error is detected, treatment is interrupted and an alarm located in a remote emergency room is activated. We assume that the wireless channels do not fail.

5.4.1 CA Action-Based Design

Capozucca et al [21] use CA actions to design and implement the FTIPT. The system is organized as a set of actions that structure the execution of sensors and actuators. Coordinated exception handling is used as the main fault tolerance mechanism, since it is not possible to roll back when insulin is administered to a patient. The CA action-design devised by the authors is informal and described by means of diagrams and textual descriptions.

Figure 5.3 shows the informal design of the system. For simplicity, it does not depict accesses to shared resources or interactions between participants. CA action CAA Cycle controls the overall execution of the system and determines the amount of insulin that must be injected for each pump based on the patient's vital signs. Actions CAA Sensors and CAA Actuators are spawned by roles ControllerChecking and ControllerExecuting of actions CAA Checking and CAA Executing, respectively. They are responsible, respectively, for collecting the vital signs of the patient and administering the insulin. Each of these composed CA actions has three roles. The roles A_RAIP and A_LAIP of CAA Actuators spawn the composed CA actions CAA RAIP and CAA LAIP, respectively. The latter two control the two pumps that will administer the two types of insulin.

Seven different types of exceptions can be raised in the system (Table 5.2). For most of these errors, exception handling consists in stopping the treatment and activating the alarm in the emergency room. In some cases, such as when the value of a sensor cannot be obtained, the handler will try again once before giving up.

5.4.2 Applying the Proposed Approach

We modeled the CA action-based design described in the previous section in Alloy. The following snippet shows part of the Alloy specification of the system. The complete specification of the system is available elsewhere [34].

```
// Imports generic CA actions model
```

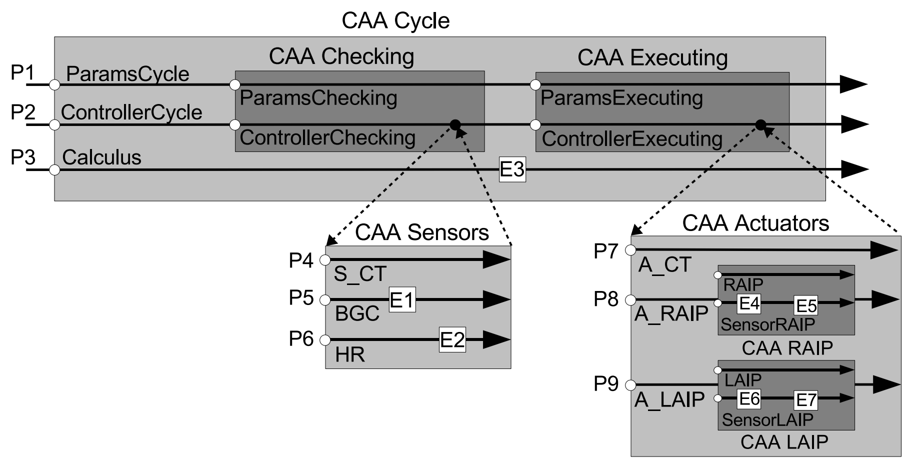


Figura 5.3: CA action-based design of the Fault-Tolerant Insulin Pump Therapy.

Exc.	Description
E1	Heart Rate (HR) sensor does not respond.
E2	Blood Glucose (BGC) sensor does not respond.
E3	Delivery limit reached.
E4	Rapid action insulin pump (RAIP) does not respond.
E5	Rapid action insulin pump (RAIP) stops during delivery.
E6	Long action insulin pump (LAIP) does not respond.
E7	Long action insulin pump (LAIP) stops during delivery.

Tabela 5.2: Exceptions in the CA action-based design.

```
open CoordinatedExceptionHandling

// CA actions extend ‘‘Action’’.
one sig CAACycle, CAAChecking, CAASensors,
  CAAExecuting extends Action{}
// Roles extend ‘‘Role’’.
one sig ControllerChecking, ParamsChecking,
  S_CT, BGC, HR extends Role {}
// Exceptions extend ‘‘RootException’’.
one sig E1, E2, E3 extends RootException {}
// Participants extend ‘‘Participant’’.
one sig P1, P2, P3, P4, P5 extends Participant {}
// Used for exception resolution.
one sig K1, K2, K3 extends Key {}
```

```
...     // Other declarations.
fact SystemStructure {
 CAACycle.NestedActions = CAAChecking
  + CAAExecuting
 CAACycle.Roles = ControllerCycle + ParamsCycle
  + Calculus
 CAAChecking.Roles = ControllerChecking
  + ParamsChecking
 no CAAChecking.NestedActions
 ControllerChecking.ComposedActions = CAASensors
 CAASensors.Roles = S_CT + BGC + HR
 P1.RolesPlayed = ControllerCycle + ControllerChecking
...// Other definitions. }
fact ExceptionFlow {
 CAASensors.Internal = E1 + E2
 && CAASensors.External = AlarmEXC
 && BGC.Generates = E1 && BGC.Raises = E1
 && HR.Raises = E2 && HR.Generates = E2
...// Other definitions. }
fact ExceptionResolution {
  CAASensors.ToResolve = E1->K1 + E2->K2
  CAASensors.ResolvedTo = K1->AlarmEXC + K2->AlarmEXC
...// Other definitions.}
```

In Alloy, a signature (`sig` keyword) specifies a type. The `one` keyword indicates that a signature has exactly one instance. We use signatures for modeling actions, roles, participants, and exceptions (signature `Key` is explained later on). Additional information is associated to these elements by means of relations. These relations are explicitly instantiated by facts, predicates that the AA must assume to be true when evaluating constraints. For instance, the fact `SystemStructure` in the snippet above states, among other things, that CA action `CAAChecking` has two roles, `ControllerChecking` and `ParamsChecking`, and no nested actions. It also states that participant P1 performs the roles `ControllerChecking` and `ControllerCycle` of actions `CAAChecking` and `CAACycle`, respectively. Moreover, the fact `ExceptionFlow` states, among other things, that roles `BGC` and `HR` raise exceptions `E1` and `E2`, and that the latter are internal exceptions of CA action `CAASensors`. The `open` clause in the beginning of the specification imports the definitions of the basic types of the proposed model, `Action`, `Role`, `Participant`, and `RootException`. Moreover, it imports the predicates that specify the basic properties of CA actions and some predefined desired properties.

Fact `ExceptionResolution` in the specification above describes the exception resolution tree of the FTIPT. It uses instances `K1` and `K2` of signature `Key` to associate internal and external exceptions of action `CAASensors`. `Key` is an auxiliary signature defined in the Alloy version of the generic CA actions model. It is necessary because the exception resolution tree of an action is a function from sets of exceptions to exceptions. Since it is not possible to define high order relations (or functions) in Alloy, we used a pair of relations, one associating internal exceptions to keys (`ToResolve`), one key for each mapping, and the other associating each key to an external exception (`ResolvedTo`). In the snippet above, fact `ExceptionResolution` states that exceptions `E1` and `E2` are both resolved to exception `AlarmEXC`. It is important to stress that this workaround for specifying the exception resolution tree of an action is not necessary in the B version of the generic CA actions model.

Let us now discuss some positive experience we had while developing the formal specification of the FTIPT case study. This work helped us in identifying a number of shortcomings in the original informal description of the system. These shortcomings were discovered when we were formalizing the system and during verification.

According to the original system description, the handlers for exceptions `E4` and `E6` "must stop the delivery of insulin and ring the danger alarm". Just by reading this statement, though, it is not possible to know the CA action that will be responsible for ringing the alarm when one of these exceptions is raised. Even though we are not explicitly modeling the actual alarm, this information is still relevant. If the alarm is to be activated by a CA action other than the one where the exception was raised, an exception should be propagated from the CA action where the error was detected to the one that will ring the alarm. However, no such exception exists in the original design of the system.

For simplicity, we could assume that some role in the CA action where an exception is raised is responsible for ringing the alarm. However, this is not the best option since it scatters the responsibility of activating the alarm throughout the whole application, partially defeating the purpose of decomposing the system into actions. In the end, we decided to add a new exception named `AlarmEXC` to the system specification. This exception is signaled by actions `CAASensors`, `CAARAIP`, and `CAALAIP` and propagated all the way up to `CAACycle`, where it is handled. Later, discussing the matter with the authors of the original case study, we discovered that, to our surprise, that was actually what they meant but forgot to state explicitly in the specification. To explicitly capture the idea that `AlarmEXC` can only be handled by `CAACycle`, we specified this constraint as the following application-specific property. Assuming the basic properties hold, it states that, for any action other than `CAACycle`, if `AlarmEXC` is an internal exception, it is also external. Moreover, it states that `CAACycle` handles `AlarmEXC`.

```
(AlarmEXC in CAACycle.Handles)
 && (all A:(Action - CAACycle)
  | AlarmEXC in A.Internal => AlarmEXC in A.External)
```

After finishing the specification of the system in Alloy, we tried to verify the basic CA action properties using the AA. In a couple of minutes, the latter presented a counterexample indicating that the specification failed to satisfy some property of interest. Careful study of the counterexample revealed that property **BP3** of Section 5.3.2 was being violated. This problem happened because the case where exceptions `E1` and `E2` are raised concurrently in action `CAASensors` was not covered by the exception resolution mechanism of the action. To fix the specification, we extended the action's resolution mechanism so that, when these two exceptions are raised concurrently, they are resolved to `AlarmEXC`. However, discussing this problem with the authors of the original case study we found out that these two exceptions are actually never raised concurrently. The authors unknowingly ommitted this information from the system specification. Hence, we modified the Alloy specification to say explicitly that `E1` and `E2` are never raised concurrently in CA action `CAASensors`. The generic CA actions model defines a relation, `Excluding`, whose goal is to exclude from the exception resolution tree of an action combinations of internal exceptions that are never raised concurrently. This relation is already taken into account by the basic CA properties defined by the generic CA actions model. By default, no combinations are excluded. The following line was introduced in the specification of the system:

```
CAASensors.Excluding = (E1 + E2) -> K3
```

5.5 Concluding Remarks

In this paper we presented an approach for specifying and verifying cooperative concurrent systems that use coordinated exception handling to achieve fault tolerance. This approach aims at guaranteeing that the fault tolerance mechanisms used to build a reliable system are also reliable. The main contribution of this paper is to provide a formalization of CA actions that makes it possible to check automatically whether a CA action-based design satisfies several properties of interest regarding coordinated exception handling. The usefulness of the proposed approach was demonstrated by a case study. Even for a very simple application, the proposed approach helped us to uncover some implicit assumptions in the original, informal design of the system. The problems we found were directly related to the use of coordinated exception handling. In other formal models for specifying CA actions, it would be harder to spot problems like the ones we found because they focus on different aspects of CA action-based systems, such as temporal ordering of events [185] and dynamic CA action structuring [172].

This work does not address some important aspects of systems structured as CA actions. For example, it is not possible to model consistent access to external resources or the dynamic structure of nested actions. In the near future, we intend to expand the system model used in our approach to address these issues and provide a more comprehensive framework for verifying CA action-based systems.

Another future work consists in extending the fault model of the FTIPT with the assumption that wireless communication channels can fail. In the least, this modification requires new types of errors and their corresponding exceptions to be defined. Furthermore, some peculiarities of wireless channels need to be taken into account. For example, sometimes wireless channels stop working for short periods of time because the signal is weak. This situation cannot immediately be treated as an error, though, since it is usually temporary.

5.6 Resumo do Capítulo 5

Este capítulo apresentou uma abordagem para a modelagem e verificação de sistemas concorrentes cooperativos que usam tratamento de exceções coordenado para estruturar seu comportamento excepcional. Usamos aplicações baseadas em *CA actions* como representantes dessa classe de sistema. O principal componente da abordagem proposta é um modelo formal de *CA actions* que especifica a estruturação de um sistema em termos de ações, assim como informações relativas ao fluxo de exceções entre essas ações. Esse modelo pode ser especificado de maneira direta usando linguagens bem conhecidas, como Alloy e B. Neste capítulo foi apresentada uma visão geral da abordagem proposta, sem descrever em detalhes o modelo de sistema no qual ela é baseada. Os Apêndices A, B e C complementam este capítulo e apresentam, respectivamente: (A) um modelo genérico de *CA actions* escrito em Alloy; (B) a especificação completa em Alloy do estudo de caso da Seção 5.4; e (C) especificações parciais do modelo genérico de *CA actions* e do estudo de caso na linguagem B.

Este capítulo encerra a primeira parte desta dissertação. Até agora este trabalho focou na integração dos componentes de um sistema de software. Foram propostas técnicas para a descrição e análise das arquiteturas desses sistemas, da perspectiva do seu comportamento excepcional. Essas técnicas visam reduzir o número de erros decorrentes de suposições conflitantes ou incompletas feitas por componentes sobre o comportamento excepcional dos outros componentes integrados no sistema. Os próximos dois capítulos da dissertação lidam com a construção dos componentes, mais especificamente, com a estruturação do comportamento excepcional desses componentes através do emprego de programação orientada a aspectos.

Capítulo 6

Um Estudo Quantitativo sobre a Modularização de Tratamento de Exceções com Aspectos

Este capítulo e o próximo focam na estruturação do comportamento excepcional de componentes de software usando programação orientada a aspectos (AOP). Em outras palavras, ele trata da construção de componentes de software nos quais o código de recuperação de erros é explicitamente separado do código responsável pelo comportamento normal. Este capítulo se refere à Seção 1.4.2 do Capítulo 1 e apresenta um estudo que avalia as vantagens de se usar AOP para modularizar tratamento de exceções. O estudo consistiu em refatorar para aspectos o código de tratamento de exceções de quatro aplicações diferentes, três delas baseadas em componentes. Três dessas aplicações foram implementadas originalmente em Java, enquanto a quarta foi escrita em AspectJ.

A linguagem AspectJ foi empregada como representante do paradigma orientado a aspectos para separar tratamento de exceções dos outros interesses de cada sistema. Para medir os atributos de qualidade das versões originais e refatoradas dos sistemas-alvo do estudo, foi usado um conjunto de métricas [73]. Este inclui métricas relativas a quatro atributos de qualidade: separação de interesses, concisão, coesão e acoplamento. A investigação incluiu também uma análise dos sistemas refatorados levando em conta (i) a reusabilidade do código de tratamento de exceções e (ii) a escalabilidade de AOP para modularizar tratamento de exceções na presença de outros interesses transversais.

O artigo que este capítulo contém foi publicado nos anais do *14th ACM SIGSOFT Symposium on Foundations of Software Engineering*, que foi realizado em novembro de 2006. Uma versão preliminar foi apresentada no *ECOOP'2005 Workshop on Exception Handling in Object-Oriented Systems* [32], em julho de 2005. Adicionalmente, uma versão estendida deste último apareceu no livro *Advanced Topics in Exception Handling Techni-*

ques [23], publicado este ano pela série *Lecture Notes in Computer Science.*

Exceptions and Aspects: The Devil is in the Details

Fernando Castor Filho[1] Nelio Cacho[2] Eduardo Figueiredo[2]
Raquel Ferreira[1] Alessandro Garcia[2] Cecília Mary F. Rubira[1]

[1]Institute of Computing
State University of Campinas
Campinas - SP - Brasil
fernando@ic.unicamp.br, raquelmaranhao@gmail.com, cmrubira@ic.unicamp.br

[2]Computing Department
Lancaster University
Lancaster - UK
{ncacho,emagno,garciaa}@comp.lancs.ac.uk

6.1 Introduction

Exception handling [80] mechanisms were conceived as a means to improve modularity of programs that have to deal with exceptional situations [51]. Their designs are aimed at promoting an explicit textual separation between normal and abnormal code, in order to support the construction of programs that are more concise, reusable, evolvable, and reliable [51, 80]. Several researchers [66, 117, 178] have explored new programming techniques in order to reap the promised benefits of existing exception handling mechanisms. In spite of this, achieving modular implementations of error handling code is still difficult for software engineers. The main problem is that realistic software systems exhibit very intricate relationships involving the normal-processing code and error recovery concerns. Moreover, exception handling is known to be a global design issue [72] that affects almost all the system modules [117], mostly in an application-specific fashion [5]. Also, a large part of the system code is usually devoted to error detection and handling [52, 182], but this part of the code is often the least understood, tested, and documented [52].

Given the broadly-scoped character of exception handling, aspect-oriented programming (AOP) techniques [106] emerge as a natural candidate to promote enhanced modularity, reusability, and conciseness of programs in the presence of exceptions. In fact, it is usually assumed that the exceptional behaviour of a system is a crosscutting concern that can be better modularized by the use of AOP [106, 114, 117]. Some recent research works [107, 117, 164] have investigated the degree to which AOP can improve the separation of concerns relative to some forms of fault tolerance mechanisms.

The most well-known study focusing specifically on exception handling was performed by Lippert and Lopes [117]. The authors had the goal of evaluating if AOP could be used to separate the code responsible for detecting and handling exceptions from the normal application code in a large object-oriented (OO) framework. According to this study, the use of AOP brought several benefits, such as less interference in the program texts and a drastic reduction in the number of lines of code (LOC). However, this first study has not investigated the "aspectization" of application-specific error handling, which is often the case in large-scale software systems. In addition, in spite of the assumption made by many authors that using AOP for separating exceptional code from the normal application code is beneficial, the involved trade-offs are not yet well-understood. For instance, previous investigations have not analyzed whether aspect-oriented (AO) solutions scale well in the presence of complex relationships involving the normal application code and error recovery code. Also, the interaction between exception handling aspects and aspects that implement other concerns still has not been explicitly studied. Hence, some important research questions remain unaddressed:

- Does AOP promote an improvement in well-accepted quality attributes other than separation of concerns, such as coupling cohesion, and size?
- Is exception handling a reusable aspect in real, deployable, software systems?
- When is it beneficial to aspectize exception handling? When is it not?
- How do exception handling aspects affect aspects implementing other concerns?

This paper presents an in-depth study that assesses the adequacy of AspectJ [114], a general-purpose aspect-oriented extension to Java, for modularizing exception handling code. The study consisted of refactoring four different applications so that the code responsible for handling exceptions was moved to aspects. Three of these applications were originally written in Java and one was implemented in AspectJ. This study differs from the Lippert and Lopes study for a number of reasons. First, the targets of the study are complete, deployable systems, not reusable infrastructures, such as a framework. Hence, the exception handling code also implements non-uniform, complex strategies, making it harder to move handlers to aspects. Second, we employ a metrics suite [73] to quantitatively assess attributes such as coupling, conciseness, cohesion, and separation of concerns in both the original and the refactored system. Third, we evaluate how exception handling aspects interact with aspects implementing other concerns. Fourth, we do not attempt to move error detection code to aspects. Error detection involves checking the state of a program against a certain predicate when its control flow graph reaches a certain node, at runtime [52]. Thus, in many cases, error detection code is very strongly coupled with the code that implements a system's normal behavior. We believe this subject deserves a separate in-depth study.

This paper is organized as follows. The next section describes the setting of our study. The results of the study are presented in Section 6.3. Section 6.4 analyzes the obtained results and points some constraints on the validity of our study. Section 6.5 discusses related work. The last section points directions for future work.

6.2 Study Setting

This section describes the configuration of our study. Section 6.2.1 briefly explains how we moved exception handling code to aspects. Section 6.2.2 describes the targets of our study. Section 6.2.3 presents the metrics suite we have used to quantitatively evaluate the original and refactored versions of each system.

6.2.1 Aspectizing Exception Handling

AspectJ [114] extends Java with constructs for picking specific points in the program flow, called join points, and executing pieces of code, called advice, when these points are reached. Join points are points of interest in the program execution through which crosscutting concerns are composed with other application concerns. AspectJ adds a few new constructs to Java. A *pointcut* picks out certain join points and contextual information at those join points. Join points selectable by pointcuts vary in nature and granularity. Examples include method call and class instantiation. Advice may be executed *before*, *after*, or *around* the selected join points. In the latter case, execution of the advice may potentially alter the flow of control of the application, and replace the code that would be otherwise executed in the selected join point. AspectJ also lets programmers suppress the static checks that the Java compiler makes regarding checked exceptions. This feature is called *exception softening* and is useful when it is necessary to move exception handlers to aspects. Exceptions are softened within a set of join points. When exceptions are thrown in these join points, they are automatically wrapped by a pre-defined unchecked exception called `SoftException`.

Our study focused on the handling of exceptions. We moved `try-catch`, `try-finally`, and `try-catch-finally` blocks in the four applications to aspects. Hereafter, we refer to these types of blocks collectively as `try-catch`, or handler, blocks, unless otherwise noted. We use the terms `try` block, `catch` block (or handler), and `finally` block (or clean-up action) to explicitly refer to the parts of a `try-catch` block. Method signatures (`throws` clauses) and the raising of exceptions (`throw` statements) were not taken into account in this study because these elements are related to exception detection.

We used the Extract Fragment to Advice [135] refactoring to move handlers to aspects. After extracting all the handlers to advice, we looked for reuse opportunities and eliminated identical handlers. The following code snippet shows a trivial example of aspectization of handlers using an around advice. Due to space constraints, we do not show how we extracted handlers to after advice.

```
// original code       // refactored code
class C {              class C {
 void m() {        ==>  void m() {
  try {...}               ... //former body of
  catch(E e) {...}       }    // the try block
 }                     }
}                      aspect A {
                        pointcut pcd :
                         execution(void C.m());
                        void around() : pcd() {
                         try { proceed(); }
                         catch(E e) {...}
                        } declare soft : E : pcd();
                       }
```

We implemented handlers in the aspects using after and around advice. Whenever possible, we used after advice, since they are simpler. After advice are not appropriate, though, for implementing handlers that do not raise an exception (handlers that mask the exception). AspectJ requires that an after advice end its execution in the same way as the join point to which it is associated. Therefore, if the code of an after advice is executed following the raising of an exception, the runtime system of AspectJ assumes that the advice ends its execution by raising an exception (either the original or a new one), even if that does not happen explicitly. Thus, it is not possible to implement as an after advice a handler that logs the caught exception and ignores it (the advice would have to raise some exception). In these cases, around advice were employed, since they do not have this restriction. Clean-up actions were implemented as after advice. New advice were created on a per-`try`-block basis, excluding cases where handlers could be reused. In situations where multiple `catch` blocks are associated to a single `try` block, we created a single advice that implements all the `catch` blocks. This helps decreasing the number of advice and, at the same time, avoids problems related to ordering multiple advice associated to the same join point.

For each target system, we employed a different strategy for organizing exception handling aspects. This approach helped us in understanding how different organizations influence handler reuse. Various approaches are possible. Extreme alternatives include putting all the exception handling code in a single aspect, creating several simple aspects that encapsulate the possible handling strategies for each type of exception, or creating a separate aspect for each handling strategy. More moderate approaches include creating a handler aspect per class that includes exception handling code or one aspect for each package. For a system where other concerns have been aspectized a priori, a feasible strategy would be to create one exception handling aspect per aspectized concern. Each organization has pros and cons that revolve around the code size vs. modularity trade-off. This is further discussed in Section 6.3.1.

Whenever possible, we associated exception handling advice to methods through `execution` pointcut designators. These pointcut designators have a simple semantics and, unlike the `call` pointcut designator, do not require that the exception handling advice deal directly with `SoftException` when it is necessary to soften exceptions. For cases where it was not possible to use the `execution` pointcut designator, we looked for alternate solutions, depending on the circumstances, usually employing the `call`, `new`, `withincode`, and `cflow` pointcut designators.

In several occasions, we modified the implementation of a method in order to expose join points that AspectJ could select more directly or contextual information required by exception handlers, for example, the values of local variables. Usually, this amounted to extracting new methods whose body is entirely contained within a `try` block, and whose parameters were the contextual information required by the handler. This was often necessary when `try-catch` blocks were tangled with the normal code, for example, nested within a loop. In these situations, using a pointcut to select the execution of the whole method might not be appropriate. After exception handling, when a tangled `try-catch` block does not end its execution by raising an exception, it is necessary to resume the execution of the normal code from the statement that textually follows the handler block. However, if no refactoring is performed, this behavior cannot be achieved for cases where it is imposible to specify a pointcut that captures exclusively the statements within the `try` part of the `try-catch` block.

6.2.2 Our Case Studies

We manually refactored four different applications in our study, three of them OO and one of them AO. Hereafter we call them “target systems”. We believe that these applications are representative of how exception handling is typically used to deal with errors in real software development efforts for several reasons. First, these systems were selected mainly because they include a large number of exception handlers that implement diverse exception handling strategies that range from trivial to sophisticated. Second, they encompass different characteristics, diverse domains, and involve the use of distinct real-world software technologies. Finally, they present heterogeneous crosscutting relationships involving the normal code, the handler code, the clean-up actions, and other crosscutting concerns. The rest of this subsection describes the four targets systems of the study.

The original implementation of the first three systems was written in Java and, afterwards, all the exception handling code was refactored to aspects. Telestrada is a traveler information system being developed for a Brazilian national highway administrator. For our study, we have selected some self-contained packages of one of its subsystems comprising approximately 3350 LOC (excluding comments and blank lines) and more than 200 classes and interfaces. Java Pet Store[1] is a demo for the Java Platform, Enterprise Edition[2] (Java EE). The system uses various technologies based on the Java EE platform and is representative of existing e-commerce applications. Its implementation comprises approximately 17500 LOC and 330 classes and interfaces. The third target system is the CVS Core Plugin, part of the basic distribution of the Eclipse[3] platform. The implementation of the plugin comprises approximately 170 classes and interfaces and approximately 19000 LOC. It is the target system with the most complicated exception handling scenarios.

Health Watcher was the only system originally implemented in AspectJ. It is a web-based information system that was developed for the healthcare bureau of the city of Recife, Brazil. The original system version involved the aspectization of distribution, persistence, and concurrency control concerns. Furthermore, the system includes some very simple exception handling aspects whose handling strategy consists in printing error messages in the user's web browser. The implementation of Health Watcher comprises 6630 LOC and 134 components (36 aspects and 98 classes and interfaces). The refactoring of Health Watcher consisted in moving exception handling code from classes to aspects. Moreover, we also moved exception handling code from aspects related to other concerns to aspects dedicated exclusively to exception handling.

6.2.3 Metrics Suite

The quantitative assessment was based on the application of a metrics suite to both the original and refactored versions of the target systems. This suite includes metrics for separation of concerns, coupling, cohesion, and size [73] to evaluate both original and refactored implementations and have already been used in several experimental studies [19, 32, 79, 84]. The coupling, cohesion, and size metrics were defined based on some classic OO metrics [43]. The original OO metrics were extended to be applied in a paradigm-independent way, supporting the generation of comparable results. Also, the metrics suite introduces three new metrics for quantifying separation of concerns. They

[1]http://java.sun.com/developer/releases/petstore/
[2]http://java.sun.com/j2ee
[3]http://www.eclipse.org

Attributes	Metrics	Definitions
Separation of Concerns	Concern Diffusion over Components	Counts the number of components that contribute to the implementation of a concern and other components which access them.
	Concern Diffusion over Operations	Counts the number of methods and advice which contribute to a concern's implementation plus the number of other methods and advice accessing them.
	Concern Diffusion over LOC	Counts the number of transition points for each concern through the LOC. Transition points are points in the code where there is a "concern switch".
Coupling	Coupling Between Components	Counts the number of components declaring methods or fields that may be called or accessed by other components.
	Depth of Inheritance Tree	Counts how far down in the inheritance hierarchy a class or aspect is declared.
Cohesion	Lack of Cohesion in Operations	Measures the lack of cohesion of a class or an aspect in terms of the amount of method and advice pairs that do not access the same field.
Size	Lines of Code (LOC)	Counts the lines of code.
	Number of Attributes	Counts the number of fields of each class or aspect.
	Number of Operations	Counts the number of methods and advice of each class or aspect.
	Vocabulary Size	Counts the number of components (classes, interfaces, and aspects) of the system.

Tabela 6.1: Metrics Suite

measure the degree to which a single concern (exception handling, in our study) in the system maps to the design components (classes and aspects), operations (methods and advice), and lines of code. For all the employed metrics, a lower value implies a better result. Table 6.1 presents a brief definition of each metric, and associates them with the attributes measured by each one. Detailed descriptions of the metrics appear elsewhere [73].

6.3 Study Results

This section presents the results of the measurement process. The data have been collected based on the set of defined metrics (Section 6.2.3). The presentation is broken in three parts. Section 6.3.1 presents the results for the separation of concerns metrics. Section 6.3.2 presents the results for the coupling and cohesion metrics. Section 6.3.3 presents the results for the size metrics.

We present the results by means of tables that put side-by-side the values of the metrics for the original and refactored versions of each target system. We break the results for the three OO target systems in two parts, in order to make it clear the contribution of classes and aspects to the value of each metric. For the AO application (Health Watcher), we break the results in three parts, in order to make it clear the contribution of classes, exception handling aspects, and aspects related to other concerns to the value of each metric. Hereafter, we use the term "class" to refer to both classes and interfaces. Rows labelled "Diff." indicate the percentual difference between the original and refactored verions of each system, relative to each metric. A positive value means that the original version fared better, whereas a negative value indicates that the refactored version exhibited better results.

6.3.1 Separation of Concerns Measures

Table 6.2 shows the obtained results for the separation of concerns metrics. In general, the refactored versions of the target systems performed better than the original ones. In the refactored versions, all the code related to exception handling that was not machine-generated was moved to aspects. We did not consider machine-generated code because it typically does not need to be maintained by developers. Among the target systems, only the Java PetStore includes machine-generated code, produced by the Java EE compiler.

Application		*Concern Diffusion over Components*		*Concern Diffusion over Operations*		*Concern Diffusion over LOC*	
		Original	Refactored	Original	Refactored	Original	Refactored
Telestrada	*Classes*	22	0	42	0	208	0
	Aspects	-	18	-	44	-	0
	Total	**22**	**18**	**42**	**44**	**208**	**0**
	Diff.	**-18.18%**		**+4.76%**		**-100%**	
Java Pet Store	*Classes*	110	20	256	21	1168	84
	Aspects	-	37	-	179	-	0
	Total	**110**	**57**	**256**	**200**	**1168**	**84**
	Diff.	**-48.18%**		**-21.88%**		**-92.81%**	
Eclipse CVS Core Plugin	*Classes*	59	0	236	0	1118	0
	Aspects	-	4	-	180	-	0
	Total	**59**	**4**	**236**	**180**	**1118**	**0**
	Diff.	**-93.22%**		**-23.73%**		**-100%**	
Health Watcher	*Classes*	35	0	115	0	488	0
	EH Aspects	5	10	9	70	0	0
	Other Aspects	7	0	12	0	48	0
	Total	**47**	**10**	**136**	**70**	**536**	**0**
	Diff.	**-78.72%**		**-48.53%**		**-100%**	

Tabela 6.2: Separation of Concerns Metrics.

Even though the measures of Concern Diffusion over Components diverged strongly amongst the four target systems, it is clear that the refactored solutions fared better. This divergence is a direct consequence of the adopted strategy for creating new handler aspects in each target system. In Telestrada, for complex classes with 7 or more `catch` blocks, we created a new aspect whose sole responsibility is to implement the handlers for that class. Furthermore, each package includes an aspect that modularizes exception handling code

Application		*Coupling between Components*		*Depth of Inheritance Tree*		*Lack of Cohesion in Operations*	
		Original	Refactored	Original	Refactored	Original	Refactored
Telestrada	*Classes*	179	142	186	186	408	524
	Aspects	-	39	-	2	-	0
	Total	**179**	**181**	**186**	**188**	**408**	**524**
	Diff.	**+1.12%**		**+1.08%**		**+28.43%**	
Java Pet Store	*Classes*	783	729	245	245	7095	7595
	Aspects	-	65	-	13	-	71
	Total	**783**	**794**	**245**	**258**	**7095**	**7666**
	Diff.	**+1.4%**		**+5.31%**		**+8.05%**	
Eclipse CVS Core Plugin	*Classes*	1481	1412	181	181	18326	19287
	Aspects	-	77	-	4	-	0
	Total	**1481**	**1489**	**181**	**185**	**18236**	**19287**
	Diff.	**+0.54%**		**+2.21%**		**+5.24%**	
Health Watcher	*Classes*	217	197	69	69	766	867
	EH Aspects	5	27	3	3	4	130
	Other Aspects	66	61	17	17	210	210
	Total	**288**	**285**	**89**	**89**	**980**	**1207**
	Diff.	**-1.04%**		**0%**		**+23.16%**	

Tabela 6.3: Coupling and Cohesion Metrics.

for simpler classes. In the Java Pet Store a single exception handling aspect was created per package. In Health Watcher, an exception handling aspect was created for each other crosscutting concern. For the CVS Plugin, the programmer was left free to create aspects as he deemed necessary. The results for Concern Diffusion over Components in Table 6.2 reflect these design choices. Telestrada is the system whose refactored version achieved the worst results (a reduction of 18.2%), since a great number of exception handling aspects were created. The refactored Java Pet Store achieved a middle ground between Telestrada and Health Watcher, with a reduction of 48.18%. In the refactored Health Watcher, the small number of exception handling aspects (10) represented a reduction of 78.7% in the value of the metric. For the CVS Plugin, only 4 exception handling aspects were created, resulting in a reduction of 93.22%.

Concern Diffusion over LOC was the metric where the refactored systems performed best, when compared to the original ones. The refactored versions of three out of four target systems did not have any concern switches and thus had value 0 for this metric. The only exception was Java Pet Store, because the machine-generated code was not moved to aspects. In spite of this, the measure for the refactored version was still more than 90% lower. Also, the measures of Concern Diffusion over LOC do not seem to be influenced by the size or characteristics of each target system. This can be seen as an indication that AOP scales up well when it comes to promoting separation of exception handlers in the program texts.

6.3.2 Coupling and Cohesion Measures

Table 6.3 shows the obtained results for the two coupling metrics, Coupling between Components and Depth of Inheritance Tree, and the cohesion metric, Lack of Cohesion in Operations. On the one hand, aspectizing exception handling did not have a strong effect on the coupling metrics. On the other hand, the measure of Lack of Cohesion in Operations for the refactored target systems was much worse than for the original ones.

The increase in the value of Depth of Inheritance Tree for some of the target systems was due to the creation of abstract aspects from which other handler aspects inherit. The greater the number of concrete aspects in a system that inherit from a newly created abstract aspect, the greater the value of the metric. Amongst all the metrics we employed, Coupling between Components was the least affected by the aspectization of exception handling. None of the target systems had a difference greater than 1.5% between the original and refactored versions. New couplings were introduced only when exception handling aspects had to capture contextual information from classes.

Lack of Cohesion in Operations was the metric for which the refactored target systems presented the worst results. The refactored versions of all target systems performed worse in this metric. For the refactored versions of Health Watcher and Telestrada, the measure of Lack of Cohesion in Operations was more than 20% higher than the corresponding original systems. In the Java Pet Store and the CVS Plugin, the increase was of approximately 8% and 5%, respectively. The main reason for the poor results is the large number of operations that were created to expose join points that AspectJ can capture. These new operations are not part of the implementation of the exception handling concern (and therefore do not affect Concern Diffusion over Operations), but are a direct consequence of using aspects to modularize this concern. Refactoring to expose join points is a common activity in AOP, since current aspect languages do not provide means to precisely capture every join point of interest.

Even though cohesion was worse in the refactored target systems, this was caused mostly by the classes. As shown in Table 6.3, the value of the cohesion metric for the aspects in the refactored version of Telestrada and the CVS Plugin was 0. In the Java Pet Store, the aspects accounted for less than 1% of the total value of the metric. Only Health Watcher was different. In this system exception handling aspects accounted for 10.8% of the total value.

6.3.3 Size Measures

Contradicting the general intuition that aspects make programs smaller [114, 117] due to reuse, the original and refactored versions of the four target systems had very similar results in two of the four size metrics: LOC and Number of Attributes. The measure of

Application		*Lines of Code*		*Number of Attributes*		*Number of Operations*		*Vocabulary Size*	
		Original	Refac.	Original	Refac.	Original	Refac.	Original	Refac.
Telestrada	*Classes*	3352	2885	127	127	423	437	224	224
	Aspects	-	459	-	0	-	44	-	18
	Total	**3352**	**3334**	**127**	**127**	**423**	**481**	**224**	**242**
	Diff.	**-0.54%**		**0%**		**+13.71%**		**+8.04%**	
Java Pet Store	*Classes*	17482	15593	542	542	2075	2135	339	339
	Aspects	-	2045	-	6	-	180	-	37
	Total	**17482**	**17638**	**542**	**548**	**2075**	**2315**	**339**	**376**
	Diff.	**+0.89%**		**+1.11%**		**+11.57%**		**+10.91%**	
Eclipse CVS Core Plugin	*Classes*	18876	17803	852	854	1832	1848	257	257
	Aspects	-	1620	-	0	-	180	-	4
	Total	**18876**	**19423**	**852**	**854**	**1832**	**2028**	**257**	**261**
	Diff.	**+2.82%%**		**+0.23%%**		**+9.66%**		**+1.43%**	
Health Watcher	*Classes*	5732	4641	152	152	542	553	98	98
	EH Aspects	86	853	3	7	9	73	5	10
	Other Aspects	812	701	12	12	104	104	31	31
	Total	**6630**	**6195**	**167**	**171**	**655**	**730**	**134**	**139**
	Diff.	**-6.56%**		**+2.4%**		**+11.45%**		**+3.73%**	

Tabela 6.4: Size Metrics.

Vocabulary Size grew as expected, due to the introduction of exception handling aspects. Moreover, the Number of Operations of the refactored versions of all the target systems grew significantly. Table 6.4 summarizes the results for the size metrics.

In Telestrada and Java Pet Store, the number of LOC of the original and refactored versions is similar (less than 1%). In Health Watcher there was a sensible decrease in the amount of exception handling code, even though the influence of this change on the overall number of LOC of the system was only modest (approximately -6.6%). In the CVS Plugin, there was an increase of 2.9% in the number of LOC of the refactored version. Although this is a small percentage of the overall number of LOC of the system, it accounts for almost 550 LOC introduced due to aspectization. The obtained values for LOC were expected. Although some reuse of handler code could be achieved, this was not anywhere near the results obtained by Lippert and Lopes in their study. Moreover, most handlers comprise a few (between 1 and 10) LOC and the use of AspectJ incurs in a slight implementation overhead because it is necessary to specify join points of interest and soften exceptions in order to associate handlers to pieces of code. In the end, the economy in LOC achieved due to handler reuse was more or less compensated by the overhead of using AspectJ.

The Number of Operations was sensibly higher in the refactored target systems. It grew 13.7% in the refactored version of Telestrada, 11.6% in the Java Pet Store, 10.7% in the CVS Plugin, and approximately %11.5 in Health Watcher. The main reason for this result was the creation of advice implementing handlers. Since there is a one-to-one correspondence between `try` blocks and advice (except for cases where handlers are reused) and handlers do not count as methods in the original systems, this increase was expected. Another reason for the increase in the Number of Operations was the refactoring of methods to expose join points that AspectJ can capture.

6.4 Discussion

This section makes a qualitative analysis of the obtained results (Section 6.3) focusing on the research questions posed in Section 6.1. We also base the analysis on our experience in modularizing exception handling in the four target systems. Furthermore, we discuss the constraints on the validity of our empirical evaluation.

6.4.1 Coupling, cohesion, and conciseness

Our empirical study confirms some of the findings of the study conducted by Lippert and Lopes [117], who claim that the use of aspects decreases interference between concerns in the program texts. The results achieved by the refactored versions of the target systems in the separation of concerns metrics (Section 6.3.1) provide convincing evidence for this.

When exception handling is aspectized using AspectJ, it is sometimes necessary to soften exceptions to suppress the static checks performed by the Java compiler. The issue with exception softening is that it creates an implicit, compile-time dependency of the base code on the exception handling aspect. The dependency is implicit because it cannot be inferred just by looking at the base code. Moreover, if the exception handling aspects are not present, the base code will not compile. An important benefit of aspectizing design patterns is the fact that dependencies are inverted and code implementing design patterns will depend on the participants of the pattern, but not the other way around [91]. This principle does not apply to the aspectization of exception handling with AspectJ because, in many situations, it is not possible to eliminate the dependency of the base code on the aspects. A direct consequence of this is that, in AspectJ, exception handling is not a pluggable aspect, differently from other concerns, such as distribution [164], assertion checking [117], and some design patterns [91].

As seen in Section 6.3.3, handler advice accounted for a significant increase in the Number of Operations of all the target systems (+10.4% in Telestrada +8.7% in the Java Pet Store, and +9.8 in the CVS Plugin and Health Watcher). As with all size metrics, this value cannot be evaluated in isolation. Although a developer getting acquainted to the refactored version will have to understand more operations, these operations are smaller and do not mix a system's normal activity with the code that handles exceptions. Therefore, the increase in the Number of Operations caused by the handler advice should not be seen as a negative factor.

Operations extracted in order to expose join points that AspectJ could capture corresponded to 3.3% of the total Number of Operations in Telestrada, 2.9% in the Java Pet Store, 0.79% in the CVS Plugin, and 1.7% in Health Watcher. Unlike the increase caused by handler advice, the increase caused by refactored operations, albeit small, is negative in most situations. These new operations are not part of the original design of the system and possibly do not clearly state the intent of the developer. In some cases, a refactored operation comprises just a few lines that do not make sense when separated from their original contexts.

The increase in Lack of Cohesion in Operations in the refactored versions of Java Pet Store and the CVS Plugin is much lower than the increase in the same metric in Telestrada and Health Watcher. In Telestrada, almost 90% of the increase in the cohesion metric is due to only three classes. These classes have a large number of complex methods, constrasting with the other classes of the system. Since the classes on the system are, in general, very simple (the Number of Operations/Vocabulary Size ratio of the original Telestrada is less than 2), we believe that the large increase in the cohesion metric exhibited by the refactored system was mainly due to the its small size. In Health Watcher, unlike the other target systems, the large increase in the cohesion metric was caused mainly by the exception handling aspects. Three such aspects can be accounted for almost 50% of the increase in the cohesion metric. It is important to stress that this result does not seem to be related to the existence of aspects in the system that implement other concerns.

6.4.2 Is exception handling a reusable aspect?

Reuse of handler code is not the main expected benefit of using aspects for modularizing crosscutting concerns in software systems. Rather, an implementation of concern code which is localized and consistent is a stronger reason for using aspects. However, some researchers [91, 114, 117, 164] claim that reuse is often a natural consequence of aspectization, specially when it comes to exception handling code [114, 117]. In our study, we found that reusing handlers is much more difficult than is usually advertised [114, 117]. This can be noticed by observing the measures for Concern Diffusion over Operations in Section 6.3.1. In the target system where the highest amount of reuse of exception handling code was achieved, a reduction of 48.5% was observed for this metric. Albeit very positive, this result still contrasts strongly with the findings of Lippert and Lopes, who claim to have achieved a reduction of more than 85% in the number of exception handlers in the target of their study.

Handler reuse directly depends on several factors. In our study, the factors that had the strongest influence on reuse were: (i) the type of exception being handled; (ii) what the handler does and whether it ends its execution by returning, raising an exception, etc.;

```
 1 // ADVICE #1
 2 boolean around() : ...  {
 3  try { return proceed(); }
 4  catch (CVSException e) { CVSProviderPlugin.log(e); }
 5  return false;
 6 }
 7 // ADVICE #2
 8 boolean around() : ... {
 9  try { return proceed(); }
10  catch (IOException e) { CVSProviderPlugin.log(
11          IStatus.ERROR,e.getMessage(),e); }
12  return false;
13 }
14 // ADVICE #3
15 boolean around() : ... {
16  try { proceed(); }
17  catch (CVSException e){ CVSProviderPlugin.log(e); }
18  return false;
19 }
```

Figura 6.1: Three similar advice that cannot be combined.

(iii) the kind of contextual information required, if any; and (iv) what the method that handles the exception returns and what exceptions appear in its `throws` clause. Some of these factors can assist in determining if aspectizing exception handling in a given context is beneficial or harmful (Section 6.4.3), independently of reuse.

The difficulty of reusing handler code is illustrated by Figure 6.1. The figure shows three advice that look similar, but cannot be merged into a single one because of small differences. Advice #1 and #2 cannot be combined because they log different error messages and handle different exceptions. A possible solution to the second problem is to implement a single advice that catches a supertype of both `CVSException` and `IOException`. Since the nearest common supertype is `Exception`, the advice should re-throw unchecked exceptions to avoid changing the system's behavior. It is also not possible to combine advice #1 and #3. The former returns a value that depends on the call to `proceed()` (Line 3) while the latter always returns `false` (Lines 17 and 19). For the same reasons, advice #2 and #3 cannot be combined.

The value of Concern Diffusion over Operations in the refactored version of Telestrada was almost 5% higher than in the original one (Section 6.3.1). This happened because, in some packages, reuse of handler code was virtually inexistent and some classes had operations with more than one `try-catch` block. Hence, when exception handling code in these classes was moved to aspects, each handler had to be put in a separate advice, contributing to the increase.

6.4.3 When to aspectize exception handling

From our experience in refactoring the four target systems, we derived a simple classification for exception handling code. This classification aims to help developers to identify the situations where moving exception handlers to aspects is beneficial and when it is not worth the effort. We use three categories to classify exceptional code in Java-like languages: (i) placement of `try-catch` blocks; (ii) dependency on local variables; and (iii) flow of control after handler execution. In the rest of this section, we describe these categories and show how they capture many of the situations that developers are likely to find when attempting to aspectize exception handling.

Placement of `try-catch` Blocks. The first category is related to where in the text of a method a handler block appears. This impacts the pointcut designators employed to capture the body of the `try` block and whether refactoring is required in order to expose join points of interest. If all the statements implementing the normal behavior of a method, including variable declarations, appear within a `try` block, we say that the placement of the containing `try-catch` block is *basic*. Aspectizing exception handling for a basic `try-catch` block is a simple matter of refactoring handlers and clean-up actions to advice, softening exceptions as necessary, and defining a pointcut to capture the whole method execution. No additional refactoring is required and the pointcut definition is usually quite simple.

If a `try-catch` block is not basic, it is *tangled*. It is usually much harder to modularize exception handling with aspects for tangled `try-catch` blocks. It is often necessary to perform some a priori refactoring that makes the `try-catch` block basic. Furthermore, depending on the context of the tangling, it may be necessary to define complex pointcuts to correctly capture the implementation of the `try` block. If all the statements that appear outside of a top-level (non-nested) tangled `try-catch` block can be moved to its corresponding `try` block without altering the behavior of the parent method, this `try-catch` block is considered basic.

A `try-catch` block can be further classified as *nested* or *top-level*. A nested `try-catch` block is contained within a `try` block, whereas a top-level `try-catch` block is not. A nested `try-catch` block is considered basic if it is the only statement in the `try` block of a basic `try-catch` block.

Dependency on Local Variables. The second category is used to separate exception handlers in two groups: those that do and those that do not depend on local variables. By "local variables" we mean variables defined within the containing context. Dependency on local variables hinders aspectization, as the joint point models of most AO languages (AspectJ included) cannot capture information stored by these variables. If a handler reads the value of one or more local variables, moving it to an aspect usually requires the use of the Extract Method refactoring, in order to expose the variables as parameters of a method. If the handler performs assignments to local variables, other refactorings might be necessary, depending on whether the variables are primitive or object types. Although ths discussion focuses on exception handlers, it also applies to clean-up actions.

Flow of Control after Handler Execution. The third category, flow of control after handler execution, is related to how an exception handler ends its execution. After a *termination* exception handler executes its last statement, the statement that textually follows the corresponding `try-catch` block is executed. A *propagation* exception handler finishes its execution by signaling an exception. In this case, system execution resumes when some other handler catches the signaled exception. A *return* exception handler is, in some ways, similar to a *propagation* handler. The difference is that, in the former, system execution resumes from site where the handler's method was called. Finally, a *loop iteration* exception handler occurs when a `try-catch` block is nested within a loop and at least one of its `catch` blocks executes a statement such as `break` or `continue`. Flow of control after handler execution affects several design choices related to the aspectization of exception handling. For example, propagation handlers can be easily implemented using after advice, whereas termination and return handlers cannot. Moreover, it is generally straightforward to define a pointcut that selects a tangled, top-level `try-catch` block if its handlers are all propagation or return. However, the same does not apply to termination handlers.

Using the proposed classification, it is possible to describe several interesting scenarios. These scenarios represent recurring situations with which a developer would have to deal if faced with the task of modularizing exception handling code using aspects. Table 6.5 enumerates the ones that we encountered the most frequently while conducting our study. Each row represents one or more scenarios. To avoid repetition, if a category is marked more than once in the same row (e.g. "basic" and "tangled" marked), the row represents more than one scenario and an OR semantics is adopted. For category Placement of `try-catch` Block, we assume that the two subcategories (basic/tangled and nested/top-level) count as different categories. For example, row 1 refers to scenarios where the placement of `try-catch` blocks is basic and either nested or top-level. Also, the `catch` block does not depend on local variables and can end its execution by terminating, propagating an exception, or returning. It is important to stress that the resulsts in Table 6.5 are directly dependent on the aspect-oriented language we employed, AspectJ.

The rightmost column of Table 6.5 indicates whether it is beneficial ("yes") or harmful ("no") to modularize exception handling with aspects in the presented scenarios. In general, we considered aspectization to be beneficial in a given scenario if it has a positive effect on the values of the metrics of Section 6.2.3, when comparing the original and aspectized code for instances of the scenario. Moreover, in some rows, the rightmost

#	Placement of `try-catch` blocks				Dependency on local vars.		Flow of control after handler execution				Should be aspectized?
	basic	tangled	nested	top-level	yes	no	term.	prop.	ret.	loop	
1	□		□	□		□	□	□	□		**Yes**.
2		□		□		□		□	□		**Yes**.
3		□		□		□	□				**Depends**.
4		□	□			□	□		□		**No**.
5		□	□			□		□			**Depends**.
6		□		□	□		□	□	□		**Depends**.
7		□	□		□		□	□	□		**No**.
8		□	□	□	□	□				□	**No**.

Tabela 6.5: Some exception handling scenarios according to the proposed classification

#	Reason
3	Aspectization is beneficial in this scenario if: (i) the code within the `try` block can be selected by a pointcut without the need for additional refactoring; or (ii) it is necessary to use Extract Method to expose a joint point that AspectJ can capture but the new method makes sense by itself, i.e., it could have been created by the developers of the system.
4 & 7	In our experience, combinations of tangling and nesting, and nesting and access to local variables usually result in complex code that needs to be refactored before it can be aspectized. In many cases more than one new operation needs to be created, negatively affecting the cohesion and conciseness of the code.
5	If the outer `try-catch` blocks do not have handlers for the exception caught by the innermost handler nor to the exceptions signaled by it, aspectization is beneficial because the advice implementing the handler can be associated to the execution of the whole method.
6	Aspectization is only beneficial if: (i) the handler accesses just a few variables (< 4) and only for reading; and (ii) the refactoring employed to expose these variables creates a method that makes sense by itself.
8	Loop iteration handlers are usually too strongly coupled with the context where they appear.

Tabela 6.6: Justification for the "no" and "depends" scenarios of Table 6.5.

column column indicates that the choice of aspectizing exception handling in a given scenario depends on factors that are not taken into account by the proposed classification. These cases are marked as "depends". We have chosen not to include these factors in the classification because they are very specific and subjective, and to keep it simple. Table 6.6 justifies the "no" and "depends" scenarios.

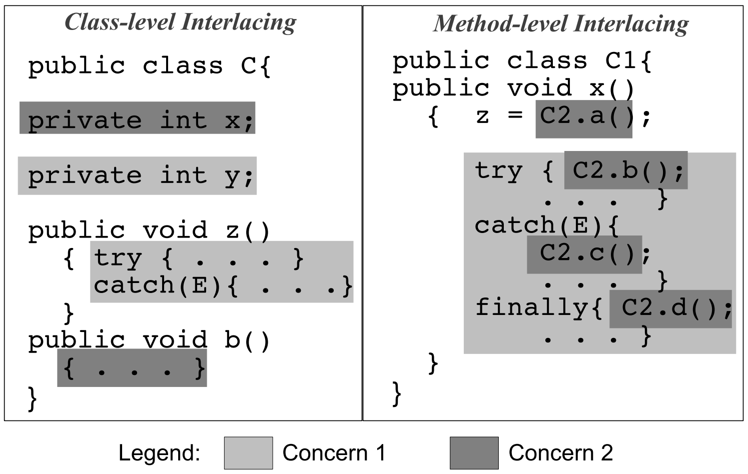
6.4.4 Exception handling and other aspects

This section analyzes the scalability of AOP when there are interactions between the implementation of exception handling and other crosscutting behaviors. The idea is to examine how easy it is to aspectize such crosscutting concerns in the presence of exception handling aspects. Our investigation was carried out mainly in the context of the Health Watcher system. However, as discussed in the previous sections and evidenced by the measurements (Section 6.3), this system has a simple exceptional behavior, when compared to the other three target systems. To obtain a more comprehensive perspective of the possible difficulties caused by interactions between exception handling and other aspects, we have also refactored part of the AspectJ version of the CVS Plugin, where the exception handling concern was already modularized with aspects. We have chosen this system, instead of the other two, because it was the case study that exhibited the most complex exception handling strategies. In the CVS Plugin, we have opted for aspectizing security (access control) and distribution (remote access through HTTP and SOCKS5 proxies) concerns, which would otherwise be naturally tangled and scattered through several classes. We selected these concerns because they are well-known as traditional crosscutting concerns in the literature, and tend to have a broadly-scoped influence in the system. More importantly, we have detected a number of different relationships between exception handling and these crosscutting concerns. These relationships were not captured in the Health Watcher system, which generally exhibited a loose coupling between the exception handling aspects and aspects implementing other concerns.

We have observed different categories of interactions involving the exception handling code and other crosscutting behaviors. They range from (i) simple invocations linking exceptional behaviors and methods relative to the other concern to (ii) the sharing of one or more module members by two different concerns. The set of interactions analyzed in this study was classified into 5 categories, which are described in the following. These categories involve either *class-level interlacing* or *method-level interlacing*. Our categorization is a specialization of interaction categories defined in a previous study where we have analyzed design pattern compositions [19]. Here we refine the previous categorization by also taking exception handling structures into account, namely protected regions (`try { }`), handlers (`catch (E) { }`), and clean-up actions (`finally { }`). To illustrate these categories, we use Figure 6.2. In the figure, Concern 1 (`C1`) corresponds to exception handling and Concern 2 (`C2`) is a second concern. In Health Watcher, `C2` can be part of the concurrency control, distribution, or persistence concerns, whereas in the Eclipse CVS plugin it is part of the security or distribution concerns.

Class-level Interlacing. The first category is concerned with class-level interlacing of exceptional behavior and other crosscutting behaviors. In this case, the implementations concerns `C1` and `C2` have one class in common. However, each concern encompasses disjoint sets of methods and attributes in the same class. As illustrated in the left-hand side of Figure 6.2, `C1` and `C2` have a coinciding participant class, but there is no method or attribute pertaining to the two concerns. This interaction category did not bring any kind of problem while aspectizing elements of `C2`. Hence we can say that AspectJ has scaled up well in scenarios involving class-level interlacing.

Method-level Interlacing. Four categories involve some form of *method-level interlacing*: *unprotected-region level*, *protected-region level*, *handler level*, and *cleanup-action level*. Differently from class-level interlacing, all these categories have a similar characteristic: the implementations of concerns `C1` and `C2` have one or more methods in common.



```
public class C{

private int x;

private int y;

public void z()
  { try { . . . }
    catch(E){ . . .}
  }
public void b()
  { . . . }
}
```

```
public class C1{
public void x()
  {  z = C2.a();

     try { C2.b();
         . . .  }
     catch(E){
         C2.c();
         . . .  }
     finally{ C2.d();
         . . . }
  }
}
```

Figura 6.2: Aspect interaction categories.

Hence exception handling code is interlaced at the method level with elements of C2. In the right-hand side of Figure 6.2, method x() has code pertaining both C1 and C2. In most of the situations in Health Watcher and the CVS plugin involved, interaction between concerns C1 and C2 consisted of calls to methods from C2 by code pertaining C1. The distinguishing feature of the four categories of method-level interlacing is where such a call is placed in terms of the exception handling elements.

The right-hand side of Figure 6.2 depicts all the 4 types of interactions encountered in the two systems. The interaction types influenced the way in which the AspectJ code of the two systems was refactored to expose the appropriate join points to the aspects of C2. The aspectization of crosscutting behaviors relative to C2 was straightforward when the situation exhibited interlacing at the unprotected- or protected-region level. The reason was that there was no explicit link between the exception handling aspects and the C2 code being aspectized. In Health Watcher, all the instances of method-level interlacing fit into one of these two categories. More explicit aspect interactions appear in methods with catch- and finally-level interlacings. These cases complicated the aspectization of distribution and security in the CVS Plugin: the advice in the exception handling aspects, which implemented handlers and clean-up actions, also contained calls to C2 methods that were being moved to aspects. In this case, we needed to change the implementation of the handler advice in order to (i) use reflective features of AspectJ to access the elements of C2, or (ii) use the execution of handler advice as join points of interest in poincuts of the C2-specific aspects.

The situation becomes more complicated when a handler advice depends on local variables (Section 6.4.3) that are initialized through calls to C2-specific methods, such as the z variable in Figure 6.2. After refactoring, exception handling aspects would be advising such a method call (`C2.a()`) in order to save the value being assigned to `z` in an aspect variable. However, with the aspectization of `C2`, the call `C2.a()` would either be moved to an aspect related to `C2` or be advised by such aspect. This would require the exception handling aspect to take this `C2`-specific aspect into account, creating a dependency between the two aspects.

6.4.5 Limitations of this Study

This study does not attempt to assess how different refactoring strategies for moving exception handlers to aspects affect the results. Furthermore, only one team of developers was responsible for conducting the study. More general results could be obtained by employing different teams of developers and performing measurements on the refactored systems produced by each team. Another limitation is that we have not evaluated how aspects affect execution time in the target systems.

Our study focuses on a single AO language, namely, AspectJ. Although many ideas presented here also apply to other AO languages, some surely do not. More powerful join point models would make it possible to deal more appropriately with some complicated cases, such as those where handler blocks are tangled or nested, thus affecting the study results. For example, using the `loop` pointcut designator of the LoopsAJ language [93] it is possible to associate handlers to exceptions raised and not handled within loops. Moreover, one of the extensions to AspectJ proposed by the developers of the abc AspectJ compiler [7] allows the selection of `throw` statements as join points. In some situations, this feature makes it possible to easily aspectize termination handlers (Section 6.4.3) that are nested or tangled in the original code (scenarions 3, 6, and 7 of Table 6.5).

Arguably, the employed metrics suite is a limitation of this work. There are a number of other existing metrics and other modularity dimensions that could be exploited in our study. We have to decided to focus on the metrics described in Section 6.2.3 because they have already been proved to be effective quality indicators in several case studies [19, 32, 79, 84]. In fact, despite the well-known limitations of these metrics, they complement each other and are very useful when analyzed together. In addition, there is no way to explore all the possible measures in a single study.

6.5 Related Work

Even though introductory texts [106, 114] often cite exception handling as an example of the (potential) usefulness of AOP, only a few works attempt to evaluate the suitability of this new paradigm to modularize exception handling code. The study of Lipert and Lopes [117] employed an old version of AspectJ to refactor exception handling code in a large OO framework, called JWAM, to aspects. The goal of this study was to assess the usefulness of aspects for separating exception handling code from the normal application code. The authors presented their findings in terms of a qualitative evaluation. Quantitative evaluation consisted solely of counting LOC. They found that the use of aspects for modularizing exception detection and handling in the aforementioned framework brought several benefits, for example, better reuse, less interference in the program texts, and a decrease in the number of LOC.

The Lippert and Lopes study was a important initial evaluation of the applicability of AspectJ in particular and aspects in general for solving a real software development problem. However, it has some shortcomings that hinder its results to be extrapolated to the development of real-life software systems. First, the target of the study was a system where exception handling is generic (not application-specific). However, exception handling is an application-specific error recovery technique [5]. In other words, the "real" exception handling would be implemented by systems using JWAM as an infrastructure and not by the framework itself. Most of the handlers in JWAM implemented policies such as "log and ignore the exception". This helps explaining the vast economy in LOC that was achieved by using AOP. Second, the qualitative assessment was performed in terms of quality attributes that are not well-understood, such as (un)pluggability and support for incremental development. The authors did not evaluate some attributes that are more fundamental and well-understood in the Software Engineering literature, such as coupling and cohesion. Third, quantitative evaluation was performed only in terms of number of LOC. Although the number of LOC may be relevant if analyzed together with other metrics, its use in isolation is usually the target of severe criticisms.

An initial assessment of the use of AspectJ for modularizing exception handling in software systems with non-trivial exception handling code has appeared elsewhere [32]. This previous assessment was based solely on a small part of Telestrada (+- 2000 LOC). Furthermore, it did not attempt identify the situations where modularizing exception handling with aspects is beneficial or harmful. Also, it did not investigate how exception handling aspects interact with aspects implementing other concerns.

One of the first studies of the applicability of AOP for developing dependable systems has been conducted by Kienzle and Guerraoui [107]. The study consisted of using AOP to separate concurrency control and failure management concerns from other parts of distributed applications. It employed AspectJ and transactions as a representative of AOP languages and a fundamental paradigm to handle concurrency and failures, respectively. This work is similar to ours in its overall goal, namely, to assess the benefits of using aspects to modularize error recovery code. However, there are some fundamental differences: (i) we use exception handling to deal with errors, instead of transactions; (ii) we substantiate our conclusions with measurements based on a metrics suite for AO software, instead of examples; (iii) we do not address concurrency; (iv) our study is more general and based on a varied set of applications with diverse error handling strategies.

Soares and his colleagues [164] employed AspectJ to separate persistence and distribution concerns from the functional code of a health care application written in Java. The authors found that, although AspectJ presents some limitations, it helps in modularizing the transactional execution of methods in many situations that occur in real systems. Furthermore, they employed aspects to modularize part of the exception handling code of an application, but did not attempt to assess the suitability of AspectJ for this task.

An early position paper by Fradet and Südolt [69] discusses the features that an AO language for detecting errors in numeric computations should provide. It proposes pointcut designators that work as global invariants whose violations trigger the execution of recovery code (advice). This work is complementary to ours because it focuses on error detection while ours emphasizes error recovery.

6.6 Concluding Remarks

In this paper, we presented an in-depth study to assess if AOP improves the quality of the application code when employed to modularize non-trivial exception handling. We found that, although the use of AOP to separate exception handling code and normal application code can be beneficial, that depends on a combination of several factors. As discussed in the previous sections, if exception handling code in an application is non-uniform, strongly context-dependent, or too complex, aspectization can bring more harm than good. We believe that effective use of AOP requires *a priori* planning and must be incorporated in the software development process. For exception handling, ad-hoc aspectization is beneficial only in simple scenarios. The main contributions of this work are: (i) a substantial improvement, based on experience acquired from refactoring four different applications, to the existing body of knowledge about the effects of AOP on exception handling code; (ii) a set of scenarios that can be used by developers to better understand when it is beneficial to aspectize exception handling and when it is not; and (iii) an initial assessment of the effects of aspect interaction when exception handling gets in the mix.

As mentioned in Section 6.1, this work does not attempt to measure the effects of aspectizing exception detection code. We believe that investigating if AOP can be employed to modularize error detection is an exciting direction for future work. We also intend to study the feasibility of devising a tool that (semi-)automatically detects and refactors exception handling code that is beneficial to aspectize.

Empirical studies about interactions between aspects have only now started to surface [19]. In the specific case of interactions between exception handling and other aspects, the presented empirical study provides an important starting point, but much remains to be done. For example, our study does not assess how handler and clean-up method-level interlacing (Section 6.4.4) affect the employed metrics. Also, the classification of Section 6.4.3 does not apply to systems where other concerns are modularized as aspects *a priori*.

6.7 Resumo do Capítulo 6

Este capítulo apresentou um estudo em profundidade com o objetivo de avaliar os benefícios trazidos pelo uso de AOP para modularizar o comportamento excepcional de sistemas de software, com uma ênfase especial em sistemas baseados em componentes. De acordo com os resultados desse estudo, o uso de AOP para separar os comportamentos normal e excepcional de um sistema é benéfico em diversas situações comuns no desenvolvimento de software, mas isso depende de uma combinação de diversos fatores. Quando o código de tratamento de exceções não é uniforme, tem um acoplamento forte com o contexto onde aparece ou envolve pontos de junção que a linguagem orientada a aspectos empregada não consegue capturar diretamente, aspectização *ad hoc* pode não ser possível ou piorar a qualidade do sistema.

Em especial, a coesão de todos os sistemas-alvo do estudo, conforme medida pela métrica que empregamos, piorou depois que tratamento de exceções foi modularizado com aspectos. Isso se deveu em grande parte à necessidade de, em alguns casos, modificar o código original antes de extrair o tratamento de exceções para aspectos, principalmente nos casos listados acima. Além disso, o grau de reuso de código de tratamento de exceções foi baixo. Embora esse resultado tenha sido esperado, tendo em vista a natureza específica de aplicação do código de tratamento de exceções [5], ele contradiz os resultados de um estudo anterior bastante conhecido [117]. Neste último, uma das consequências do uso de AOP foi uma grande economia de linhas de código devido ao reuso de tratadores. Finalmente, o estudo mostrou que interesses transversais que afetam uma mesma parte do código podem dificultar a aspectização. Isso é evidenciado nos casos de entrelaçamento no nível do método envolvendo blocos tratadores e ações de limpeza.

O estudo também propôs uma classificação para código de tratamento de exceções com base em alguns dos fatores que tiveram maior influência sobre a modularização com aspectos. Essa classificação e o conjunto de cenários derivados dela são um importante primeiro passo na direção de um catálogo para auxiliar desenvolvedores a decidir quando aspectizar o comportamento excepcional de um sistema. O próximo capítulo expande a classificação proposta e descreve em detalhes um catálogo de cenários que inclui diversas situações que não são contemplados por este capítulo. Esse catálogo visa fornecer o ferramental necessário para a construção de componentes de software onde os comportamentos normal e excepcional são devidamente separados.

Capítulo 7

Extração de Tratamento de Exceções para Aspectos: Um Catálogo de Práticas

A partir dos resultados do estudo apresentado no Capítulo 6, este capítulo propõe uma classificação para código de tratamento de exceções. Essa classificação leva em consideração os fatores que, no estudo que realizamos, tiveram maior influência sobre a extração do comportamento excepcional de um sistema para aspectos. Adicionalmente, propomos um catálogo de cenários que consiste de combinações desses fatores. O objetivo desse conjunto de cenários é guiar a tarefa de modularizar tratamento de exceções usando aspectos, tanto para sistemas já existentes quanto para sistemas novos, de modo a auxiliar na construção de componentes de software que são mais fáceis de manter, entendíveis e simples. Para atingir esse fim, o catálogo indica em quais cenários o uso de AOP melhora a qualidade do sistema e em quais piora.

Consideramos o uso de aspectos benéfico em determinado cenário quando: (i) o código normal, depois de ter os tratadores de exceções extraídos para aspectos, não exibe "maus cheiros" [68]; (ii) pouco ou nenhum redesenho do sistema *a priori* é necessário; (iii) a solução empregada para extrair os tratadores para aspectos é genérica, no sentido de que pode ser usada em todas os quase todas as instâncias do cenário; e (iv) o código extra introduzido devido ao uso de aspectos não é muito grande, ou seja, idealmente, para cada bloco `try-catch` extraído para um aspecto, no máximo um novo *advice* deve ser criado. Adicionalmente, considerando os fatores que influenciam a "aspectização", elaboramos um sistema de pontuação simples para decidir mais objetivamente se um dado cenário é benéfico ou prjudicial.

Este capítulo se refere à Seção 1.4.3 do Capítulo 1. O artigo que este capítulo contém foi submetido para a IEEE International Conference on Software Maintenance 2007. Além disso, uma versão preliminar aparece na série de relatórios técnicos do IC-UNICAMP [33].

Extracting Error Handling to Aspects: A Cookbook

Fernando Castor Filho[1] Alessandro Garcia[2]

Cecília Mary F. Rubira[1]

[1]Institute of Computing
State University of Campinas
Campinas - SP - Brasil
{fernando,cmrubira}@ic.unicamp.br

[2]Computing Department
Lancaster University
Lancaster - UK
garciaa@comp.lancs.ac.uk

7.1 Introduction

Exception handling [80] mechanisms were conceived as a means to improve modularity and maintainability of programs that have to deal with exceptional situations [51]. Ideally, an exception handling mechanism should make it possible to write programs where: (i) the code for error detection, error handling, and the normal behavior are lexically separate so that they can be modified independently [142]; (ii) the impact of the code responsible for error handling in the overall system complexity is minimized [146]; and (iii) an initial version that does little recovery can be evolved to one which uses sophisticated recovery techniques without a change in the structure of the system [142]. The use of exception handling in the construction of several real-world systems and its inclusion in many mainstream programming languages, such as Java, Ada, and C++, attest its importance to the current practice of large-scale software development.

In spite of the pivotal role of modular error handling in the overall system quality, there is not much implementation/design guidance in the literature on how to use exception handling mechanisms to develop error handling code that is modular. Most of the software development methodologies used in practice pay little attention to the design of a system's exceptional behavior. Furthermore, most of the good programming and design cookbooks and catalogues, like the refactoring [68] and design pattern [70] catalogues, either concentrate on the system's normal behavior or are too generic to be successfully applied to modular exception handling.

The popularization of new techniques for separation of concerns, such as aspect-oriented programming (AOP) [106], further aggravates this problem as new decompositions of the system's exceptional behavior become possible. Although it is usually assumed that the exceptional behavior of a system is a crosscutting concern that can be better modularized by the use of AOP [114, 117], the *ad hoc* use of this new paradigm is sometimes detrimental to the quality of a system [24, 85]. If developers do not receive the proper guidance, software systems whose exceptional behavior is refactored to aspects can manifest a number of problems that stem from poorly designed or implemented error handling code. Examples include limited reusability [24], swallowed exceptions [125], incorrect handler binding [24], and unforeseen control flow [150].

In this paper, we address this problem by providing design guidance regarding the use of AOP to modularize exception handling. We propose a classification for error handling code based on the factors that we found out have more influence on the task of refactoring exception handling to aspects. We use the proposed classification to devise a set of scenarios that comprise combinations of these factors and indicate whether aspectization is beneficial or harmful in each of these scenarios. Our goal is twofold: (i) to assist developers of new systems to modularize the exceptional behavior with aspects, so that they can focus on the beneficial scenarios and avoid the harmful ones; and (ii) to assist maintainers of existing systems in the task of refactoring error handling code to aspects by showing when aspectization is worth the effort and when it is not.

This paper is organized as follows. Section 7.2 provides background on exception handling, introduces the AOP language we have used, AspectJ [114], and discusses the interplay between aspects and exceptions. Section 7.3 presents the proposed classification for error handling code, whereas Section 7.4 introduces a catalog of scenarios that can be derived from the interactions among the elements of the classification. Section 7.5 presents an evaluation of our approach. Section 7.6 reviews some related work. The last section rounds the paper and points directions for future work.

7.2 Background

7.2.1 Exception Handling

Exception handling [52] is a technique for structuring the error recovery code of a system so that errors can be more easily detected, signaled, and handled. Many mainstream programming languages, such as Java, Ada, C++, and C#, implement exception handling mechanisms. These languages provide constructs to signal the occurrence of an error (raise or throw an exception) and to associate a set of recovery measures with the error, in order to remedy the problem (handle the exception).

When a part of a program raises an exception, the underlying exception handling mechanism is responsible for changing the normal control flow of the computation within the program to its exceptional control flow. Therefore, raising an exception results in the interruption of the normal activity of the component, followed by the search for an appropriate *exception handler* (or simply *handler*) to deal with the signaled exception. After the execution of an exception handler, control returns to the code that immediately follows the handler.

Exception handling contexts (EHCs) are regions in a program where the same exceptions are always treated in the same way. An EHC can have a set of associated handlers, among which a handler is chosen when exceptions are raised within the context. Additionally, a context may have an associated clean-up action, which is executed independently of exceptions being raised. In Java, `try` blocks define exception handling contexts, `catch` blocks define handlers, and `finally` blocks define clean-up actions.

7.2.2 AspectJ Basics

AspectJ [114] is a general purpose aspect-oriented extension to Java. It extends Java with constructs for picking specific points in the object call graph, called *join points*, and executing pieces of code, called *advice*, when these points are reached. A *pointcut* picks out certain join points and contextual information at those join points. AspectJ includes operators for the definition of pointcuts formed by the combination of various join points. Examples of join points include method call (the caller's context), method execution (callee 's context), and field access. Line 2 of Figure 7.1 declares a pointcut, named `pch`, that selects calls to the constructor of class `C`.

Advice are pieces of code executed *before*, *after*, or *around* a selected a join point. In the latter case, the advice may alter the flow of control of the application and replace the code that would be otherwise executed in the selected join point. Lines 3-6 declare an advice that is executed *after* the join points selected by `pch` when their execution ends by throwing `E1` (Line 3). The advice handlers the thrown exception (accessible in the advice through variable `e1`) by throwing a new exception, an instance of `E2`.

Aspects are units of modularity for crosscutting concerns. They are similar to classes, but may also include the AspectJ-specific elements. Figure 7.1 defines an aspect named `ConnectionPoolHandler`.

```
public aspect A {
  pointcut pch() : execution(* C.new(..));
  after() throwing (E1 e1) throws E2 : pch() {
    throw new E2(e1.getMessage());
  }
  ...
}
```

Figura 7.1: A simple exception handling aspect.

7.2.3 AOP and Exceptions

The ideas we present in this paper are derived from lessons learned from several systenatuc studies we have conducted to assess the benefits of modularizing various crosscutting concerns with aspects [24, 73, 85, 111]. In one of them, we have specifically targeted the aspectization of exception handling code [24]. This study consisted of refactoring some existing applications to move the code implementing heterogeneous error handling strategies to separate aspects. We have performed quantitative and qualitative assessments of four systems, three written in Java and one in AspectJ. These systems were developed by third parties from industry and academia.

Our study has focused on the handling of exceptions. We have moved `try-catch-finally` blocks in the four applications to aspects. Handlers in the aspects were implemented using *after* and *around* advice. Whenever possible, we have used *after* advice, since they are simpler. Clean-up actions were implemented exclusively as *after* advice. New advice were created on a per-`try`-block basis. We employed pointcuts to select the code that throws exception. Method signatures (`throws` clauses) and the raising of exceptions (`throw` statements) were not modified because they are part of the error detection concern. In several occasions, we have modified the implementation of a method in order to expose join points that AspectJ can select more directly or contextual information required by exception handlers, for example, the values of local variables. We restricted the "allowed" modifications to well-known refactorings [68] and their aspect-oriented counterparts [135] and did not use refactorings that modified more than one class.

One of the most important lessons we learned from the aforementioned study was that, although the use of AOP to separate exception handling from base code can be beneficial, that depends on a combination of several factors. In many common situations, in order to evolve a system to use aspects to implement error handling, some a priori redesign is necessary. If the exception handling code in an application is non-uniform, strongly context-dependent, or too complex, ad hoc aspectization might not be possible or lead to code which: (i) worsens the overall quality of the system; (ii) does not represent the original design of the system; and/or (iii) exhibits "bad smells" [68], such as temporary field and middle man.

7.3 Classification for Exception Handling Code

The existence of many situations where aspectization is not a good idea motivated us to try to understand precisely what factors make it easier or harder to modularize error handling with aspects. In this paper we propose a simple classification for exception handling code. This classification emphasizes factors that have the strongest impact on the aspectization of exception handling.

We classify exception handling code in Java-like languages according to the following categories: (i) tangling of `try-catch` blocks; (ii) nesting of `try-catch` blocks; (iii) dependency of exception handlers on local variables; (iv) placement of exception-throwing code; and (v) flow of control after handler execution. In the rest of this section we describe each of these categories.

Tangling of `try-catch` Blocks. The first category describes where in the body of a method a `try-catch` block appears. We consider a `try-catch` block *tangled* if it is textually preceded or followed by a statement in the body of the method where it appears. Declarations and statements that appear textually before a `try-catch` block make it *tangled by prefix*. Accordingly, statements that appear after a `try-catch` block make it *tangled by suffix*. A `try-catch` block appearing within a loop is considered tangled by prefix and suffix, independently of the placement of other statements. A `try-catch` block whose `try` block surrounds the whole method body is considered *untangled*. Figure 7.2(a) presents an untangled `try-catch` block. The shaded `try-catch` block in Figure 7.2(b) is tangled by suffix because it is followed by a call to `n()`. The `try-catch` block in Figure 7.2(c) is tangled by prefix, because the declaration of variable `i` precedes it.

Tangling of `try-catch` blocks impacts the selection of an exception handling context as a join point of interest. It also might make it difficult for an aspect to simulate the flow of control after handler execution of the original implementation. It is important to stress that, in our experience, tangling by prefix `try-catch` has no impact on aspectization. Hence, hereafter, whenever we say that a `try-catch` block is tangled, we mean that it is tangled by suffix.

Nesting of `try-catch` Blocks. A `try-catch` block can be also classified as *nested* or *non-nested*. A nested `try-catch` block is one that is contained within a `try` block. The shaded code snippet of Figure 7.2(d) is an example of nested `try-catch` block. Nesting of `try-catch` blocks can make it difficult to understand the flow of exceptions in a given context. Moreover, implementing nested `try-catch` blocks as advice may require these advice to be ordered, so that they mimic the behavior of the original code. This is necessary if some of these advice are associated with the same join points. We do not consider nested a `try-catch` block that appears within a `catch` or `finally` block, as such a `try-catch` block is naturally part of the system's exceptional behavior. The shaded

```
class C0 {
  void m(){
    try{...}
    catch(E e){...}
  } //m()
} //C0
```

(a)

```
class C1 {
  void m(){
    try{...}
    catch(E e){...}
    n();
  } //m()
} //C1
```

(b)

```
class C2 {
  void m(){
    int i = 0;
    try{...}
    catch(E e){...}
  } //m()
} //C2
```

(c)

```
class C3 {
  void m(){
    try{
      try{...}
      catch(E1){...}
    }catch(E2 e){...}
  } //m()
} //C3
```

(d)

```
class C4 {
  void m(){
    try{...}
    catch(E1 e){
      try{...}
      catch(E2){...}
    } //catch
  } //m()
} //C4
```

(e)

```
class C5 {
  void m(){
    try{
      n(); //throws E
      p(); // throws E
    }catch(E e){...}
    q();
  } //m()
} //C5()
```

(f)

```
class C6 {
  void m(){
    int x = 0;
    try{ x = n();//throws E
    }catch(E e){ p(x); }
  } // m()
} //C6
```

(g)

```
class C7 {
  void m(){
    int x = 0;
    try{ x = n(); //throws E
    }catch(E e){ x = p(); }
    q(x);
  } // m()
} //C7
```

(h)

```
class C8 {
  void m(){
    try{ n();//throws E
    }catch(E e){
      System.out.println(e);
    }
    p();
  } //m()
} //C8
```

(i)

```
class C9 {
  void m() throws E2{
    try{ n();//throws E1
    }catch(E1 e){
      throw new E2(e);
    }
  } // m()
} //C9
```

(j)

```
class C10 {
  void m(){
    try{ n();//throws E
    }catch(E e){
      return;
    }
  } // m()
} //C10
```

(k)

```
class C11 {
  void m(){
    for(int i=0;i++;i<10){
      try{ n();//throws E
      }catch(E e){break;}
    } //for
  } //m()
} //C11
```

(l)

Figura 7.2: Examples of the categories of the proposed classification for error handling code.

`try-catch` block in Figure 7.2(e) is **not** nested for the purposes of aspectizing error handling. Instead, it is considered part of the outer `catch` block.

Placement of Exception-Throwing Code. An exception-throwing statement is a statement within a `try` block that can throw exceptions that some handler wihin the same method catches. Exception-throwing code (ETC) is the set of exception throwing-statements within the same `try` block. An exception-throwing statement is *terminal* if it is not followed by any statements in the `try` block. When referring to a `try-catch` block, we consider its ETC terminal if all of its exception-throwing statements are terminal. In the general case, terminal ETC includes a single terminal exception-throwing statement. However, there are special cases such as when a `try-catch` has multiple `if` statements. In Figure 7.2(f), the call to method `n()`, which throws exception `E`, is *non-terminal* because it is followed by a call to `p()`. On the other hand, the call to `p()` is terminal because it is the last statement of the `try` block. This factor influences aspectization because it can make it difficult for aspects to simulate the control flow of the original program.

Dependency of Exception Handlers on Local Variables. This category classifies exception handlers in two groups: those that do and those that do not depend on local variables. By “local variables” we mean variables defined within the containing block of a given `try-catch` block. Dependency on local variables hinders aspectization, as the joint point models of the most popular AO languages (AspectJ included) cannot capture information stored by these variables. If a handler reads the value of one or more local variables, we say that it is `context-dependent`. Moving such a handler to an aspect often requires the use of the Extract Method refactoring [68] to expose the variables as parameters of a method. Often, it is possible to avoid such refactoring by selecting the join point where the value is generated (e.g. the return value of a method) and saving it for later retrieval during errror handling (e.g. in a table in the error handling aspect). In Figure 7.2(g), the `catch` block reads the value of local variable `x`. If a handler performs assignments to local variables, we call it a **context-affecting** handler. In this case, more radical refactoring may be necessary. The resulting code almost always exhibits bad smells such as “Temporary Field” [68]. The handler in Figure 7.2(h) performs an assignment to local variable `x`.

Flow of Control After Handler Execution. The fifth category describes how a `catch` block ends its execution. After the execution of a *masking* handler, control passes to the statement that textually follows the corresponding `try-catch` block. Figure 7.2(i) presents a masking handler. After its execution, method `p()` is invoked. A *propagation* handler finishes its execution by throwing an exception, as shown in Figure 7.2(j). In this case, control is transferred to the nearest handler that catches the exception. A *return* exception handler ends its execution with a `return` statement, as shown in Figure 7.2(k). After the handler returns, system execution resumes from the site where the handler’s method was called. Finally, a *loop iteration* handler is declared inside a loop and executes a statement such as `break` or `continue`. Figure 7.2(l) presents an example that ends with a `break` statement.

Flow of control after handler execution affects several design choices related to the aspectization of exception handling. For example, propagation handlers can be easily implemented using *after* advice, whereas masking and return handlers have to be implemented using *around* advice, as they stop the propagation of the exceptions they catch. Moreover, it is generally straightforward to simulate the flow of control of the original program (i.e. the program before error handling is aspectized) for a tangled `try-catch` block if its handlers are all propagation or return. These types of handlers ignore the code that follows the `try-catch` block, and thus the fact that it is tangled. However, the same does not apply to masking handlers.

7.4 A Catalog of Exception Handling Scenarios

Using the classification we presented in the previous section, it is possible to describe several exception handling scenarios representing recurring situations in software development, in the specific context of Java-like languages. We derive these scenarios by analyzing combinations of the factors of the proposed classification. Table 7.1 consolidates the most frequent ones we have identified. Each scenario is named after its most distinctive features. For example, the first scenario of Table 7.1 is simply named "Untangled handlers", because it deals with untangled `try-catch` blocks. The second one is named "Tangled, Non-masking handlers", because it refers specifically to `try-catch` blocks that include propagation or return handlers. To avoid repetition, if a category is marked more than once in the same row (e.g. tangled and untangled marked), the row represents more than one scenario and an OR semantics is adopted. For simplicity, hereafter we use the term "scenario" to refer to all the scenarios that each row represents. For example, scenario Untangled Handler describes situations where a `try-catch` block is untangled, independently of nesting and placement of ETC. Also, the corresponding `catch` block does not depend on local variables and can end its execution by masking, propagating an exception, or returning.

Scenario	**Tangled `try-catch` blocks**		**Nested `try-catch` blocks**		**Placement of exception-throwing code**		**Handler depends on local variables**			**Flow of control after handler execution**				**Score**	**Should extract?**
	yes	**no**	**yes**	**no**	**non-t.**	**term.**	**read**	**write**	**no**	**mask.**	**prop.**	**ret.**	**loop**		
Untangled Handler		X	X	X	X	X			X	X	X	X		0-1	**Yes**.
Tangled, Non-Mask. Handler	X			X	X	X			X		X	X		0	**Yes**.
Nested, Non-Mask. Handler	X		X		X	X			X		X	X		1	**Yes**.
Tangled Handler, Term. ETC	X			X		X			X	X				0	**Yes**.
Nested Handler,	X		X			X			X	X				1	**Yes**.
Term. ETC															
Block Handler,	X			X	X				X	X				2	**Depends**.
Nested Block	X		X		X				X	X				3	**Depends**.
Handler															
Context-Dependent Handler	X	X		X	X	X	X			X	X	X		2-4	**Depends**.
Nested, Context-Dependent Handler	X	X	X		X	X	X			X	X	X		3-5	**No**.
Context-Affecting Handler	X	X	X	X	X	X		X		X	X	X		3-6	**No**.
Loop Iter. Handler		X	X	X	X	X	X	X	X				X	5-9	**No**.

Tabela 7.1: Exception handling scenarios according to the proposed classification

Score	Description	Factors
1	Hinders aspectization in a general and non-limiting way.	- Nesting of `try` blocks
2	Some priori refactoring or a considerable implementation overhead are usually necessary. However, the process involves well-known refactorings and results in a generic solution that may or may not exhibit bad smells.	- Combinations of non-term. ETC, masking handler, and tangled `try-catch` block - Handler that reads from local variables
3	Refactoring is almost always necessary and the solutions that apply to each instance of the scenario are usually context-dependent.	- Handlers that write to local variables - Loop iteration handlers

Tabela 7.2: Scores for the factors that hinder the aspectization of error handling.

The rightmost column of Table 7.1 indicates whether it is beneficial ("Yes") or harmful ("No") to modularize exception handling with aspects in a given scenario. In some rows, it indicates that the choice of aspectizing exception handling in a given scenario depends on factors that are not taken into account by the proposed classification. These cases are marked as "Depends". We considered aspectization to be beneficial in a scenario if: (i) the code, after aspectization, does not generally exhibit bad smells; (ii) little or no a priori redesign is necessary; (iii) the solution we used to extract the handlers to aspects applies to most or all of the instances of the scenario; and (iv) the implementation overhead introduced by aspectization is not very large, i.e., ideally, for each `try-catch` block extracted to an aspect, at most one new advice should be created.

Additionally, considering the various factors that influence aspectization, we devised a simple scoring system to more objectively judge whether a given scenario is beneficial or harmful. The scores for each scenario appear in the column labeled "Score". They can be obtained by summing the score associated to each factor that is (potentially) present in each scenario. Table 7.2 provides a key for calculating the scores. Basically, a score between 0 and 1 indicates a "Yes" scenario, a score between 2 and 3 is a "Depends" scenario, and a score of 4+ is a "No" scenario. For example, scenario Untangled Handler has a score that ranges between 0 and 1. Its only complicating factor is nesting, whose score is 1, and even then it does not appear in all the instances of the scenario (hence the range, instead of a fixed value). In the rest of this section we explain each scenario of Table 7.1.

Untangled Handler. Since the `try-catch` block is not tangled, it is possible to select the entire execution of the method as the join point of interest. Moreover, it is straightforward to extract the handlers to an aspect. It does not matter whether the exception-throwing code is terminal or not. Flow of control after handler execution is not a problem, as long as it is not loop iteration. If there are nested `try-catch` blocks, it may be necessary to rank the handler advice (i.e. advice implementing exception handlers after aspectization has taken place) if some of them are associated with the same join points.

Tangled, Non-Masking Handler. In this scenario, the `try-catch` block is tangled but it is possible to specify a pointcut that captures the execution of the whole method and associate the handler advice with it. In the original code, the handler ends its execution by either returning or throwing an exception, thus ignoring whatever comes after the `try-catch` block. Hence, the tangling does not matter because the code that textually follows the `try-catch` block is never executed when an exception is thrown from the `try` block. The exception-throwing code does not matter in this case because the execution of the whole method is the join point of interest.

Nested, Non-Masking Handler. This scenario bears strong resemblance to the previous scenario, but the nesting of `try-catch` blocks introduces some complications that hinder aspectization. Since the `try-catch` block is nested, it may be necessary to order handler advice associated with the same join points. Moreover, many (3+) levels of nesting combined with tangled `try-catch` blocks often result in complex code and special care should be taken in order to avoid the introduction of subtle bugs and changing the system behavior. In spite of these complications, aspectization is still beneficial in this scenario. In the general case, no a priori refactoring has to be applied to the original code and the aspectized code often exhibits good structuring that reflects the original design of the exceptional behavior.

Tangled Handler, Terminal ETC. It is easy to aspectize exception handling in this case because it is possible to directly associate a handler advice with the exception-throwing statement. Since the latter is terminal and the handler has a termination behavior, after execution of the aspectized handler, control passes to the statement that follows the `try-catch` block in the original implementation. The code snippet below presents a simple instance of this scenario and a possible implementation of the handler as part of an aspect.

```
// within class C         // within aspect A
 void m(){                 pointcut callP():
  try{                      call(* C.p()) &&
   n();                     withincode(* C.m());
   p(); // throws E        around() :  callP() {
  }catch(E e){log(e);}      try { proceed();}
  q();                      catch(E e) {log(e);}
 } //m()                   } //advice
```

On the left-hand side, the call to `p()` is a terminal exception-throwing statement and the handler has a termination behavior. The right-hand side defines a pointcut that selects calls to method `p()` made within the code of method `m()`. An *around* advice implements the handler. The `proceed()` statement is defined by AspectJ and executes the selected join point from an *around* advice, in this example, the call to `p()`. After exception handling, control will return to the following statement, the call to `q()`. This mimics exactly the behavior of the original program.

Nested Handler, Terminal ETC. Nesting makes it slightly harder to modularize error handling with aspects in this scenario, when compared to the previous one. In general, though, it does not negatively affect the quality of the final code. Therefore, we believe aspectization is beneficial in this scenario.

Block Handler. Moving the error handling code to an aspect in this scenario is hard because of the combination of non-terminal ETC, tangled `try-catch` block, and a handler that has a masking behavior. These factors make it difficult to mimic the behavior of the original code using an aspectized handler. Figure 7.3 presents an example of this scenario.

```
void m(){
  try{  n(); // throws E
        p();
  }catch(E e){ log(e); }
  q();
} //m()
```

Figura 7.3: A Block Handler example

Based on Figure 7.3, we assume a situation where an `n()` raises an exception and control is transfered to the handler. If we refactor the handler to an advice and associate the latter with the call to method `n()`, after exception handling the call to method `p()` will be executed. However, in the original implementation, control should be passed to the statement following the `try-catch` block, in the example, the call to `q()`. On the other hand, if we associate the handler advice with the execution of method `m()`, control will return to `m()`'s caller after exception handling. This implies that the call to `q()` will not be executed. The bottom line is: we would like to associate a handler advice to a block containing more than one statement, just like a `try` block, instead of a single statement or a whole method. For this reason, we say that the combination of factors of this scenario results in a *block exception handler*, or simply *block handler*.

To the best of our knowledge, no current aspect-oriented languages provide constructs for implementing block handlers. A possible workaround to this limitation is to extract the code within the `try` block to a new method and associate a handler advice to the execution of the extracted method. This solution has drawbacks, however: (i) it modifies the original design of the system's normal behavior; (ii) it might result in methods that do not make sense by themselves; and (iii) it is not effective in all cases because it is not always possible to extract a piece of code to a new method [68]. In general, in this

Approach	Problems
Transform variable into field	- Conceptually bad because local variables only make sense in the method where they appear; - Must deal explicitly with multi-threaded programs; - Suffers from the "Temporary Field" bad smell;
Capture assignment to variable	- Not always possible to capture the join point of interest; - Imposes an additional implementation overhead; - Must also deal explicitly with multi-threaded programs.
Extract a new method	- Modifies the design of the system's normal behavior; - Might result in methods that do not make sense; - Not effective in all cases.
Extend exception class	- The resulting exception classes might include information specific to certain handling strategies; - Requires non-local changes to the program.

Tabela 7.3: Limitations of various approaches to expose local variables to handlers.

scenario, aspectization is only beneficial if it is possible to extract the code within the `try` block to a new method that could as well have been created by the developers of the system.

Nested Block Handler. This scenario presents the same complications as the scenario we just described, but includes nested `try-catch` blocks. Although nesting generally complicates aspectization, nested block handlers do not follow this trend. If one uses the aforementioned solution for aspectizing block handlers, the nesting hierarchy of `try` blocks can be decomposed according to the extracted methods. This strategy neutralizes the effects of nesting.

Context-Dependent Handler. Aspectizing handlers that use local variables of the enclosing method in AspectJ is difficult because the language does not make it possible for advice to access the local variables visible at a join point of interest. To try to address this problem, we analyzed four general ways of exposing a local variable to an advice: (#1) to transform the variable into a field of the enclosing class; (#2) to capture some use of the variable and store this information for later retrieval; (#3) to extract the piece of code where the variable is used to a new method and expose this variable as a parameter of the extracted method; and (#4) to extend the class that defines the thrown exception so that its instances can store the values of the variables. We concluded that none of them is ideal. Table 7.3 lists the drawbacks of each of these solutions.

The decision of refactoring the handler to an advice in this scenario depends on: (i) whether it is possible to expose the values of the local variables of interest in a way that is conceptually reasonable and does not impose a large implementation overhead; and (ii) whether it would be beneficial to aspectize the handler if we ignored the dependency of the handler on the method's local variables and classified the code scenarios previously described.

Nested Context-Dependent Handler. Combinations of nested `try-catch` blocks and handlers that read values of local variables are, in general, difficult to aspectize and not worth the effort. Code adhering to this scenario is often very complex and includes handlers accessing local variables defined various nesting levels above. In order to modularize exception handling, it is almost always necessary to intensively refactor the OO implementation. The resulting aspect-oriented code often includes a sensible implementation overhead. Moreover, the design of the normal code is often altered beyond recognition.

Context-Affecting Handler. In this scenario the `catch` block performs assignments to local variables of the containing method. Amongst the four solutions we presented for exposing local variables to advice, only the first one makes it possible for the aspect to change the value of the variable without duplicating code. As pointed out previously, this solution has some severe problems. The task of moving a `catch` block to an advice is described by the Extract Fragment to Advice [135] refactoring, an aspect-oriented adaptation of the Extract Method refactoring. Hence, similar constraints apply. In general, it is not possible to extract a code snippet to a new method if this snippet performs assignments to local variables of the containing method [68]. The same holds when one tries to extract a code snippet to an advice. Thus, as a rule, it is harmful to aspectize exception handling in this scenario.

Loop Iteration Handlers. In this scenario, the problem is that commands `break` and `continue` appear inside a `catch` block and are part of the error handling concern. However, the loop where the corresponding `try-catch` block appears is part of the normal behavior of the system. Since these commands have to be associated with a loop, otherwise a compile-time error occurs, it is not possible to extract the `catch` block to an advice "as-is". As a consequence, loop iteration handlers inevitably require some redesign of the original code before they can be aspectized.

Table 7.4 summarizes the general strategies that should be applied to extract error handling to aspects in each scenario. The second and third columns of the table specify the type of advice and the type of pointcut to use, respectively. The fourth column indicates whether it is necessary to order handler advice associated to the same join points in each scenario. In some rows, this column is marked "maybe", meaning that each instance of the scenario must be independently analyzed. The fifth column indicates the scenarios that exhibit block handlers. The sixth points out the scenarios where it is necessary to expose local variables read or written by the exception handlers. The rightmost column indicates the worst-case scenarios, those where some redesign of the system's normal behavior might be necessary.

Scenario	**Advice**	**Type of Pointcut**	**Order Advice**	**Block Handler**	**Expose Variables**	**Redesign**
Untangled Handlers	*after* or *around*	*execution*	no	no	no	no
Tangled, Non-Masking Handlers	*after*	*execution*	no	no	no	no
Nested, Non-Masking Handlers	*after*	*execution*	yes	no	no	no
Tangled Handlers, Terminal ETC	*around*	*call* and *withincode*	no	no	no	no
Nested Handlers, Terminal ETC	*around*	*call* and *withincode*	yes	no	no	no
Block Handler	*around*	*execution*	no	yes	no	no
Nested Block Handler	*around*	*call* or *execution*	yes	yes	no	no
Context-Dependent Handler	*after* or *around*	*call* or *execution*	no	maybe	yes	no
Nested, Context-Dependent Handler	*after* or *around*	*call* or *execution*	yes	maybe	yes	no
Context-Affecting Handler	*around*	*call* or *execution*	maybe	yes	yes	yes
Loop Iteration Handler	*around*	*call* or *execution*	maybe	yes	yes	yes

Tabela 7.4: General strategies for structuring error handling with aspects in each scenario.

Attributes	Metrics	Definitions
Separation of Concerns	Concern Diffusion over Components	Number of components that contribute to the implementation of a concern.
	Concern Diffusion over Operations	Number of methods and advice that contribute to a concern's implementation.
Coupling	Coupling Between Components	Number of components declaring methods or fields that may be called or accessed by other components.
Cohesion	Lack of Cohesion in Operations	Measures the lack of cohesion of a class or an aspect in terms of the number of method and advice pairs that do not access the same field.
Size	Lines of Code	Number of lines of code.
	Number of Operations	Number of methods and advice of each class or aspect.

Tabela 7.5: Metrics Suite

7.5 Evaluation

To evaluate the efficacy of the proposed catalog, we conducted a case study targeting two different systems. The main goal of the case study was to assess whether a developer following the recommendations of the catalog to aspectize error handling code produces code of higher quality than a developer not using it. We measured the quality of the target systems using the metrics described in Table 7.5. This metrics suite includes metrics that serve as indicators of measure attributes that are directly and indirectly related to reuse and maintainance [73, 111]. A detailed description is available elsewhere [73]. Based on these quality attributes, we compared three versions of the systems: (i) an object-oriented (OO) version, implemented in Java; (ii) an aspect-oriented version where all the `try-catch` blocks were refactored to aspects ; and (iii) an aspect-oriented version where all the instances of "Yes" scenarios (Section 7.4) were refactored to aspects.

The case study targeted two of the systems, Telestrada and the Java Pet Store. Telestrada is a traveler information system being developed for a Brazilian national highway administrator. For our study, we have selected some self-contained packages of one of its subsystems comprising approximately 3350 LOC (excluding comments and blank lines) and more than 200 classes and interfaces. The Java Pet Store[1] is a demo for the Java Platform, Enterprise Edition[2] (Java EE). The system uses various technologies based on the Java EE platform and is representative of existing e-commerce applications. Its implementation comprises more than 18000 LOC and approximately 300 classes and interfaces. In this case study, we considered each `try-catch` block a scenario instance.

The results of the evaluation for the three versions of the two systems are depicted

[1]http://java.sun.com/developer/releases/petstore/

[2]http://java.sun.com/j2ee

System	Metric Version	Concern Diffusion over Components	Concern Diffusion over Operations	Coupling between Components	Lack of Cohesion in Operations	Lines of Code	Number of Operations
Telestrada	Original	22	**42**	188	434	3424	**432**
	Entirely Aspectized	18	**42**	**182**	469	**3368**	471
	Partially Aspectized	**18**	**42**	185	**409**	3431	465
Java Pet Store	Original	110	256	**812**	**7258**	18442	**2174**
	Entirely Aspectized	57	200	820	7808	18544	2408
	Partially Aspectized	**54**	**197**	818	7328	**18247**	2316

Tabela 7.6: Results of measuring the three versions of the target systems.

in Table 7.6. For all the metrics, lower values imply in better results. The “Partially Aspectized” (PA) version of each system is the one where we used the catalog to guide aspectization. The “Entirely Aspectized” (EA) version is the one where aspectization was ad hoc. The measures for the two separation of concerns metrics (Concern Diffusion over Components and Concern Diffusion over Operations) highlight the main advantage of using aspects: to improve the textual separation between the base code and error handling. The aspect-oriented verions of the target systems performed substantially (up to 50%) better than the OO versions. For Coupling between Components and Lines of Code, the three versions exhibited similar results. The two metrics where the aspect-oriented versions performed worse were Number of Operations and Lack of Cohesion in operations. In the former, the two aspect-oriented versions performed worse because each handler advice counts as a new operation, unlike `try-catch` blocks in the OO version. This is a natural consequence of using aspects to modularize error handling.

The most interesting result that Table 7.6 presents pertains the cohesion metric. In our previous studies, systems where exception handling was modularized through aspects consistently exhibited worse results for this metric [24]. In some cases, the measure for the aspect-oriented version of a system was more than 25% higher than the corresponding OO implementation. This trend persists when we compare the OO and EA versions of the two target systems. The measures for Lack of Cohesion in Operations are 8.06 and 7.58% higher for the EA versions of Telestrada and the Java Pet Store, respectively.

The proposed scenario catalog had a significant positive impact on the cohesion of the PA versions of the systems. In general, the catalog marks as harmful the scenarios where a priori refactoring is necessary. In our experience, these are exactly the scenarios that increase Lack of Cohesion in Operations [24]. As a consequence, the measure of the cohesion metric for the PA version of the Java Pet Store is only 0.96% higher than the measure for the OO version. Comparing the two aspectized versions, the measure for the

EA version is 6.55% higher. For Telestrada, the results were even better. The measure for the EA version was almost 15% higher than the results for the EA version. Also, the value of the cohesion metric for the OO version of Telestrada was 6.11% higher that the value fot eh PA version. These results indicate that the proposed catalog may be advantageous. For this case study, the PA versions of the two systems exhibited the same benefits as the EA version while avoiding its drawbacks.

7.6 Related Work

Since the publication of the first design pattern catalog [70], several catalogs that try to amass design/implementation time-tested knowledge regarding OO software development activities have appeared. Most of them focus on general purpose refactorings [68] and design patterns [70]. A few describe specific solutions and guidelines, based on practical experience, concerning error handling [60, 125].

In recent years, some works have surfaced which try to gather together similar knowledge regarding aspect-oriented software development activities [49, 90, 114, 135]. Many authors have devoted attention to developing refactoring catalogs [49, 90, 113, 135] for aspect-oriented software. A few of them [113, 49] include specific procedures for moving exception handling code to aspects. Laddad presents the "Extract Exception Handling" refactoring. The refactoring centers around the effects of using it to extract trivial error handling code, but does not explain when it is useful (or possible) to apply it in practice. Cole and Borba describe a set of behavior-preserving programming laws that can be combined in order to specify a refactoring that works along the same lines as "Extract Exception Handling". These laws include preconditions that indicate when it is possible to apply them. While refactorings targeting error handling code concentrate on the mechanics of moving a `try-catch` block to an aspect, the work we present identifies situations where this is beneficial and where it is not.

Some authors describe their experience in the application of exception handling to real OO software systems in the form of best practices [60] and anti-pattern [22, 125] catalogs. An early paper by Cargill [22] describes some of the problems that often happen in C++ programs due to the complex control flow that exception handling creates. Doshi [60] presents two small catalogs of best practices: one for specifying APIs whose services can throw exceptions and another one for developing the client code of these APIs (the actual exception handlers). A recent article by McCune [125] presents a list of error handling anti-patterns, some of them well-accepted (e.g. "catch and ignore" and "throwing `Exception`"). Bruntink and colleagues [16] have analyzed how error handling is implemented in a large embedded software system that uses the return-code idiom for signaling exceptions. They have assessed the fault-proneness of this idiom and proposed:

(i) a characterization for it; (ii) a fault model; and (iii) a set of macros that summarize best practices for dealing with its most error-prone instances. Best practices and anti-pattern catalogs provide guidelines on what exception handlers should and should not do. Our work complements these approaches by providing design guidance to developers using AOP techniques to enhance the modularity of error handling code.

The most well-known study focusing on the interplay between exception handling and AOP was performed by Lippert and Lopes [117]. The authors used AOP to modularize exception handling in a large OO framework and also attempted to identify situations where it was easy to aspectize error handling code and highlighted a few where it was not. However, since the study targeted a reusable infrastructure, instead of a full-fledged application, it is difficult to extrapolate their findings to the development of real-world systems where error handling is application-specific.

7.7 Concluding Remarks

This paper makes two contributions to the current body of knowledge about the interplay between AOP and error handling. First, a classification of exception handling code in terms of factors that we found out have more influence on aspectization. Second, an analysis of the interactions amongst these factors, grouped as sets of scenarios. An initial evaluation has shown that the proposed approach may help to improve the quality of a system, when used to guide the refactoring of exception handling to aspects. The general conclusion that can be drawn from this work is that AOP does not fix poor designs. In other words, in OO systems where exception handling code is already well-structured, AOP further improves the structure by completely separating the system's normal and exceptional activities. However, for systems with very complex and/or poorly structured exception handling code, AOP merely adds insult to injury and worsens the overall quality of the system. A very preliminary version of the work we presented in this paper first appeared elsewhere [24].

This work focused on a single aspect-oriented language, namely, AspectJ. We have not attempted to assess whether the proposed classification and scenarios catalog apply to other aspect-oriented languages. More powerful join point models would make it possible to deal more appropriately with more complicated cases, such as those where handler blocks are tangled or nested, thus affecting the study results. In the future, we intend to experiment with other languages in order to understand: (i) to what extent the ideas we present here are generally applicable; and (ii) how other language features impact the aspectization of exception handling. A priori, we intend to evaluate the pros and cons of using CaesarJ [130] and the JBoss AOP framework[3] to modularize error handling code.

[3]http://labs.jboss.com/portal/jbossaop

7.8 Resumo do Capítulo 7

Este capítulo apresentou um catálogo de cenários referente a código de tratamento de exceções. O uso de AOP para modularizar o tratamento de exceções é benéfico em diversas situações, mas em outras resulta em código que: (i) não representa o projeto original do sistema; (ii) exibe maus-cheiros; e (iii) tem uma influência negativa na qualidade do programa, conforme medido pelo conjunto de métricas que empregamos. O catálogo proposto organiza essas situações de modo que desenvolvedores podem tomar a decisão de estruturar o comportamento excepcional de um programa usando AOP de maneira mais rápida e com um conhecimento objetivo sobre as consequências dessa decisão. O catálogo é baseado em uma classificação que indica os fatores que têm maior influência na tarefa de aspectizar tratamento de exceções. Essa classificação expande a que foi apresentada no Capítulo 6 com uma categoria adicional e refina duas das categorias já existentes, de modo a tornar mais precisas as diretrizes providas pelo catálogo. O catálogo foi avaliado através de um estudo de caso envolvendo dois sistemas. Os resultados do estudo de caso indicam que, através do uso do catálogo, desenvolvedores de componentes podem ganhar as principais vantagens de se usar AOP (melhor separação de interesses) evitando alguns de seus problemas (coesão piorada, aspectizações "artificiais").

Os resultados deste capítulo e do Capítulo 6 mostram que a Hipótese de Pesquisa 2 (Capítulo 1, Seção 1.4.1) é válida, embora isso não se aplique a todos as circunstâncias e não seja o caso para um uso *ad-hoc* de programação orientada a aspectos. Esses dois capítulos mostraram as situações em que o uso de AOP é prejudicial e apresentaram soluções para que AOP contribua de maneira positiva para a qualidade do código de tratamento de exceções.

Capítulo 8

Conclusões

Este capítulo revisita o problema que esta tese visa resolver, as soluções propostas e lista as contribuições deste trabalho para o estado da arte. O capítulo também propõe algumas extensões e descreve resumidamente os resultados obtidos ao longo do meu doutorado que não foram incluídos nesta tese.

8.1 Problema e Solução Proposta

Este trabalho se situa na interseção das áreas Engenharia de Software e Tolerância a Falhas. Da primeira lançamos mão de avanços recentes nas áreas de Arquitetura de Software, Verificação de Sistemas de Software e Programação Orientada a Aspectos. Da segunda usamos a idéia de Tratamento de Exceções como um meio de estruturar mecanismos de tolerância a falhas. Além disso, o trabalho é fortemente calçado num dos princípios básicos da ciência da computação: Dividir para Conquistar. Nesta pesquisa, esse princípio se manifestou através da idéia de que a separação entre os comportamentos normal e excepcional de um sistema resulta em sistemas que são mais fáceis de se construir, manter e compreender.

A tese abordou o problema de se construir sistemas tolerantes a falhas baseados em componentes. O enfoque do trabalho foi restrito a sistemas de software nos quais a separação entre comportamento normal e comportamento excepcional é feita através de tratamento de exceções. A investigação consistiu de duas partes, uma relativa à integração de componentes de software e outra relativa à sua construção, já que essa divisão de responsabilidades é inerente ao desenvolvimento baseado em componentes. Como consequência da dicotomia construção/integração, a tese propôs soluções para responder duas perguntas de pesquisa:

1. *Como reduzir o número de erros, em sistemas baseados em componentes, decorrentes*

de suposições conflitantes ou incompletas sobre o comportamento excepcional dos componentes integrados no sistema?

2. *Como melhorar a estruturação interna de componentes de software, de modo a minimizar o impacto do comportamento excepcional desses componentes sobre a sua complexidade total?*

As soluções propostas são resumidas pelas seguintes teses, detalhadas e justificadas ao longo dos seis capítulos anteriores:

> **Tese 1**: *A descrição rigorosa de como exceções fluem entre componentes no nível arquitetural e a verificação automática de propriedades relativas ao comportamento excepcional do sistema nesse nível de abstração aumentam a chance de que falhas de projeto relacionadas ao comportamento excepcional sejam detectadas antes que o sistema seja implementado.*
>
> **Tese 2**: *O uso de programação orientada a aspectos para modularizar o código de tratamento de exceções de um programa melhora a qualidade desse programa, tanto do ponto de vista do comportamento normal quanto do excepcional.*

8.2 Resumo das Contribuições

As principais contribuições desta tese são as seguintes:

1. Um mecanismo de tratamento de exceções que funciona no nível da arquitetura de software e leva em consideração algumas das particularidades de sistemas baseados em componentes. Esse mecanismo é centrado em um estilo arquitetural específico, C2 [173], e evidencia a influência que estilos arquiteturais têm na propagação de exceções (Seção 1.3.2 e Capítulo 2).

2. Uma abordagem para dar suporte à descrição e à análise de arquiteturas de software, da perspectiva do seu comportamento excepcional. Essa abordagem leva em consideração a maneira como diferentes estilos arquiteturais influenciam a propagação de exceções e complementa metodologias de desenvolvimento cujo foco é o comportamento excepcional do sistema (Seção 1.3.3 e Capítulo 3).

3. Um modelo formal que indica as responsabilidades de cada elemento arquitetural, no que se refere ao lançamento, recebimento e tratamento de exceções, e que permite

que certas propriedades úteis relativas ao fluxo de exceções entre esses elementos sejam verificadas de maneira automática. Essas propriedades foram organizadas de tal maneira que cada uma pode ser verificada independentemente e o conjunto de todas elas descreve um mecanismo arquitetural de tratamento de exceções (Seção 1.3.4 e Capítulo 4).

4. Uma abordagem para a especificação e verificação de sistemas concorrentes cooperativos da perspectiva do seu comportamento excepcional (Seção 1.3.5 e Capítulo 5).

5. Uma análise dos fatores que tiveram influência na modularização do código de tratamento de exceções usando aspectos, com base na experiência adquirida através da refatoração de quatro aplicações distintas (Seção 1.4.2 e Capítulo 6).

6. Uma avaliação inicial dos efeitos da estruturação de tratamento de exceções usando aspectos quando outros interesses transversais também são modularizados através de aspectos (Seção 1.4.2 e Capítulo 6).

7. Um catálogo de cenários de tratamento de exceções que indica, para cada cenário, se é vantajoso usar aspectos para separar comportamento normal e comportamento excepcional (Seção 1.4.3 e Capítulo 7).

8.3 Outros Resultados

Além dos resultados descritos em detalhes nos capítulos desta tese, diversos outros foram obtidos ao longo do meu doutorado. Quase todos resultaram de colaborações com colegas do IC-UNICAMP e de outras instituições. Esses trabalhos são sucintamente descritos a seguir.

8.3.1 Métodos para o Projeto e Implementação do Comportamento Excepcional

Conforme mencionado na Seção 1.3, a abordagem proposta para a descrição e análise do comportamento excepcional dos componentes integrados em um sistema exige uma metodologia de desenvolvimento que inclua atividades específicas para o levantamento do modelo de falhas do sistema e a especificação do seu comportamento excepcional. Tendo isso em vista, trabalhei junto com outros colegas do IC-UNICAMP e com o Prof. Rogério de Lemos, da Universidade de Kent at Canterbury, no refinamento e extensão da metodologia MDCE [65]. A MDCE complementa metodologias de desenvolvimento "tradicionais" com uma abordagem sistemática para o projeto dos mecanismos de detecção e o tratamento de erros. A MDCE é baseada no processo Catalysis [61] e o seu foco está nas atividades de especificação de requisitos e projeto detalhado. Um artigo [154] descrevendo a MDCE e apresentando algumas melhorarias com relação à sua versão original foi publicado no periódico *Software – Practice and Experience*. O título desse artigo é "Exception handling in the development of dependable component-based systems" e eu participei como co-autor.

Em um outro trabalho, a metodologia MDCE [154] foi estendida e adaptada ao processo UML Components [41] para desenvolvimento baseado em componentes. A metodologia resultante, batizada de MDCE+, apresenta algumas melhorias com relação à MDCE. Primeiro, ela inclui algumas atividades para o projeto da arquitetura do sistema [28]. Segundo, ela inclui diretrizes para a implementação desses sistemas. Essas diretrizes são o resultado de um outro trabalho, descrito no próximo parágrafo. Terceiro, a MDCE+ inclui atividades para testar o comportamento excepcional dos componentes de um sistema, baseadas no trabalho de outros pesquisadores do IC-UNICAMP [151]. Um artigo [54] descrevendo a MDCE+ foi apresentado no *2nd Latin-American Symposium on Dependable Computing*. O título do artigo é "A Method for Modeling and Testing Exceptions in Component-Based Software Development". Uma versão preliminar desse artigo foi apresentada no *IV Workshop Brasileiro de Desenvolvimento Baseado em Componentes*. Nos dois artigos eu participei como co-autor.

Foi elaborado também um conjunto de diretrizes e padrões para a implementação de componentes de software compatíveis com diferentes estratégias de tolerância a falhas. Esse trabalho argumenta que sistemas baseados em componentes exigem: (i) estratégias para tratamento de exceções decorrentes da composição de componentes; e (ii) estratégias locais a cada componente para lidar com erros internos ao componente. Dois artigos foram publicados como resultado dessa pesquisa. O primeiro [87] foi uma versão resumida apresentada na *30th Euromicro Conference*. O título desse artigo é "Structuring Exception Handling for Dependable Component-Based Software Systems". Neste trabalho eu participei como co-autor. O segundo artigo [29] foi publicado no periódico *Journal of the Brazilian Computer Society*, em uma edição especial sobre sistemas confiáveis, e seu título é "A Systematic Approach for Structuring Robust Component-Based Software". Eu fui o primeiro autor desse artigo.

8.3.2 Um Arcabouço Baseado em Aspectos para Sistemas Concorrentes Cooperativos

Implementamos um arcabouço baseado em aspectos para a construção de sistemas tolerantes a falhas orientados a objetos que apresentam concorrência e distribuição como requisitos não-funcionais. A principal característica desse arcabouço é a modularização de cada um desses requisitos como um conjunto de aspectos, classes e interfaces. Desse modo, é possível adicionar e remover realizações desses requisitos a uma aplicação baseada no arcabouço de maneira quase completamente transparente. Um artigo [37] descrevendo sua implementação foi apresentado no *VIII Simpósio Brasileiro de Linguagens de Programação*. O título desse artigo é "Implementing Coordinated Exception Handling for Distributed Object-Oriented Systems with AspectJ" e eu sou seu primeiro autor. Esse artigo também apareceu em uma edição especial do periódico *Journal of Universal Computer Science* [36].

8.3.3 Evolução de Sistemas Baseados em Componentes

Participei da elaboração de uma abordagem sistemática para a evolução de componentes de software. Essa abordagem visa melhorar a substitutabilidade de novas versões de componentes e facilitar a evolução dos sistemas nos quais esses componentes estão inseridos. A solução proposta consiste de três partes: (i) um modelo de versionamento para componentes de software; (ii) um conjunto de regras de evolução que limitam as formas como componentes podem evoluir; e (iii) um esquema de numeração para versões que leva em consideração o impacto das mudanças feitas ao longo da evolução do componente. Um artigo [119] descrevendo a este trabalho foi apresentado no *ECOOP'2005 Workshop on Architecture-Centric Evolution*. O título desse artigo é "A Systematic Approach for the Evolution of Reusable Software Components". Eu particiei como co-autor desse trabalho.

8.3.4 Um Ambiente para o Desenvolvimento de Sistemas Baseados em Componentes

Atualmente, o grupo de pesquisa de Engenharia de Software e Confiabilidade do IC-UNICAMP, do qual faço parte, está desenvolvendo, como parte de um grande projeto envolvendo outras instituições, como a UFPB, a empresa CiT e o Centro de Estudos e Sistemas Avançados do Recife (CESAR), um ambiente centrado na arquitetura para o desenvolvimento de sistemas baseados em componentes. Esse ambiente, batizado Bellatrix, está sendo implementado como um conjunto de *plugins* para a plataforma Eclipse [62]. O ambiente visa dar suporte às atividades definidas pelo processo UML Components e consiste de um conjunto de ferramentas, entre as quais: (i) editor de interfaces; (ii) editor de configurações arquiteturais; (iii) repositório de componentes; (iv) *framework* para testes unitários de componentes; e (v) gerador de esqueletos de código em conformidade com a arquitetura. Um trabalho [174] descrevendo o Bellatrix foi apresentado no IV Workshop Brasileiro em Desenvolvimento Baseado em Componentes. O título desse artigo é "Bellatrix: um Ambiente com suporte Arquitetural ao Desenvolvimento Baseado em Componentes" e eu sou co-autor.

8.4 Extensões

Há várias direções possíveis para trabalhos futuros no que concerne às duas partes desta tese. Algumas dessas propostas são listadas a seguir:

Formalização da execução de sistemas concorrentes cooperativos. A principal limitação da abordagem proposta nos Capítulos 3, 4 e 5 é que essa abordagem se baseia em uma visão estática da arquitetura do sistema. Embora os estudos de caso descritos nos Capítulos 3 e 5 tenham mostrado que essa visão estática permite que algumas propriedades úteis sejam verificadas e que algumas falhas de projeto sejam identificadas, as propriedades mais interessantes e difíceis de verificar estão relacionadas à execução do sistema. Isso é especialmente verdade para sistemas concorrentes, nos quais o não-determinismo torna uma verificação "manual" inviável. Para que tais propriedades possam ser verificadas, é necessário estender a formalização proposta anteriormente para que inclua uma noção de tempo relativo, de modo que seja possível modelar a ordem de eventos como trocas de mensagens, o lançamento de uma exceção e a execução de um tratador. Já há alguns trabalhos nessa linha na literatura [172, 185], mas eles não têm um foco tão forte em tratamento de exceções.

Prova de teoremas na verificação do comportamento excepcional. Conforme foi visto no Capítulo 3, uma limitação da abordagem proposta é escalabilidade. Este é um problema geral de técnicas automáticas de verificação [44], apesar dos muitos avanços da área nos últimos 20 anos. Consequentemente, uma outra direção possível para pesquisas futuras é empregar a técnica de prova de teoremas para verificar se a especificação do comportamento excepcional de um sistema satisfaz certas propriedades de interesse. Embora essa técnica exija intervenção manual, ferramentas modernas [45, 140] conseguem realizar uma grande parte dessas provas de forma automática. Essa linha de pesquisa poderia, inclusive, ser combinada com a anterior.

Refatoração automática de tratamento de exceções para aspectos. No tocante à estruturação de tratamento de exceções usando aspectos, um trabalho futuro consiste em automatizar duas tarefas: (i) detecção de situações nas quais a modularização do comportamento excepcional com AOP é benéfica; e (ii) refatoração do código de tratamento de exceções para aspectos nesses casos. Neste trabalho seria especialmente interessante elaborar estratégias para realizar a extração de tratadores de exceções em trechos de código envolvendo uma mistura de diversos cenários e separá-los de maneira coerente nos aspectos.

Novas construções para linguagens orientadas a aspectos. O estudo descrito no Capítulo 6 mostrou que é difícil modularizar com aspectos o código de tratamento de exceções em algumas situações comuns no desenvolvimento de software. Na maior parte dos casos, isso se deve a duas limitações da linguagem AspectJ: (i) a linguagem não permite a seleção de blocos arbitrários como pontos de junção; e (ii) a linguagem não permite que tratadores ou ações de limpeza refatorados para aspectos façam referência a variáveis locais dos métodos onde originalmente apareciam. Essas limitações na expressividade de AspectJ apontam na direção de duas linhas para pesquisas futuras: (1) propostas de novas construções para a linguagem AspectJ visando contornar essas limitações; e (2) a avaliação da adequação de outras linguagens orientadas a aspectos para modularizar tratamento de exceções.

Estruturação de detecção de erros usando aspectos. Finalmente, uma linha futura de pesquisa bastante interessante consiste em avaliar a adequação de AOP para modularizar o código responsável por detectar erros. Na literatura de Tolerância a Falhas [5], há um consenso de que o código de detecção de erros é dependente demais do código que implementa o comportamento normal. Logo, não vale a pena tentar separar essas duas partes pois a chance de se introduzir falhas de projeto no processo é grande demais. Embora AOP já tenha sido usada para modularizar a implementação de contratos [117] (pré- e pós-condições), não está claro se esse paradigma é apropriado para separar código de detecção de erro em outras situações comuns, quando esse código está misturado ao código responsável por outros interesses.

Apêndice A

Generic CA Actions Model in Alloy

```
module CoordinatedExceptionHandling

abstract sig RootException {}

abstract sig Keys {}

abstract sig Role {
  Signals : set RootException,
  Raises : set RootException,
  GeneratesSignaling : set RootException,
  GeneratesRaising : set RootException,
  Resolved : set RootException,
  // Encountered exceptions are assumed to be raised and never
  // signaled. On the one hand, this simplifies the semantics
  // of the model for cases where one wants to specify that an
  // encountered exception is directly signaled (without being
  // raised or resolved or propagated or anything). Moreover,
  // allowing encountered exceptions to be signaled requires
  // recursive definitions for Raises and Signals in order to
  // avoid ambiguity. On the other hand, designers have to be
  // slightly more explicit when specifying CA actions and roles
  // (because the encountered exception has to be raised, resolved
  // and, only then, signaled).
  Encounters : set RootException,

  // Two views on action-oriented exception handling: (i) only
  // actions are contexts; and (ii) both actions and roles may be
  // contexts. We support the second, but its easy to support the
  // first as well. No modifications to the model are required.
  Handles : set RootException,
  ComponentActions : set Action,
  Propagates : RootException->RootException,
  Aborts : set RootException
}

abstract sig Action {
  Internal : set RootException,
  // Alloy can not express a relation of the type
  // (set RootException)->RootException. The problem with this is
  // that we have to use RootException->RootException relations,
  // which are not adequate. For example, let E1, E2, and E3 be
  // exceptions. If E1 is raised in isolation and resolved to E1
  // and (E1 + E2) raised concurrently are resolved to E3, this
  // can not be expressed.
  ToResolve : RootException->Keys,
  ResolvedTo : Keys->RootException,
  // Sets of exceptions that do not need to be resolved because
  // the developer knows that the exceptions in each set will never
  // be raised concurrently.
  Excluding : RootException->Keys,
  External : set RootException,
  Roles : set Role,
  NestedActions : set Action,
  // The exceptions to be signaled when the action aborts or fails,
  // respectively.
  AbortException : lone RootException,
  FailException : lone RootException
}

abstract sig Participant {
  // The roles the participant plays.
  RolesPlayed : set Role
}

pred action_consistent(A : Action) {
  A.Internal = A.Roles.Raises + A.NestedActions.External
  A.External = A.Roles.Signals + A.AbortException + A.FailException
  (A.ResolvedTo) :> (A.Roles.Resolved) = A.ResolvedTo
  (let RES = (A.Internal).((A.ToResolve).(A.ResolvedTo)) |
    (let Aborted = {E:RES | all R:A.Roles | E in R.Aborts } |
          (#Aborted > 0 => (some A.AbortException))
          && (#Aborted = 0 => no A.AbortException)
    )
    &&
    // If the action signals at least two different exceptions and
    // the action does not abort nor handle all of its resolved
    // exceptions, then it may fail. A.Roles.Signal is used,
    // instead of A.External, in order to cover the case where
    // A.FailException is in A.External, because we do not want to
    // take into account A.FailException, unless it is one of the
    // exceptions signaled by the roles (independently of being
    // A.FailException).
    (#(RES - { E: RES | all R:A.Roles |
      E in R.Handles || E in R.Aborts}) > 0 =>
      ( (#(A.Roles.Signals + A.FailException) > 1 =>
          some A.FailException)
        &&
        (#A.Roles.Signals =< 1 => no A.FailException)
      )
    )
  )

  (all K_E:Keys | #(A.Excluding).K_E > 0 => (no K_TR:Keys |
    (A.ToResolve).K_TR = (A.Excluding).K_E))

  // Exception raising is consistent.
  all_keys_conc_raising_consistent(A, A.ToResolve)

  // Groups of exceptions that can be raised concurrently but
  // should not be taken into account by resolution are also
  // consistent.
  all_keys_conc_raising_consistent(A, A.Excluding)

  participants_consistent(A)

  // Requires high-order quantification.
/*  (all ES : some (A.Internal) |
    concurrent_raising_consistent(A, ES) => !(all K : Keys |
      (A.ToResolve).K != ES && (A.Excluding).K != ES)
  )
*/
}

pred participants_consistent(A:Action) {
  // Consistency of nested actions. All the roles in a nested
  // action are played by participants who also play some role
  // in the enclosing action.
  (all NA : A.NestedActions | all NAR : NA.Roles |
    !(all P : Participant |
    !(NAR in P.RolesPlayed && some (P.RolesPlayed & A.Roles))))
  // All roles of action A are associated to some participant
  (Participant.RolesPlayed & A.Roles) = A.Roles
  // No Participant can play two roles in the same action.
  (no P : Participant | #(P.RolesPlayed & A.Roles) > 1)
}

pred all_keys_conc_raising_consistent(A:Action,
    ExcMap:RootException->Keys){
  (all K:Keys | let ES = ExcMap.K |
    concurrent_raising_consistent(A, ES))
}

// Auxiliary predicate. Checks whether, for a set of exceptions
// ES in the domain of A.ToResolve, each exception E in ES was
// (could have been) raised by a different participant.
pred concurrent_raising_consistent(A:Action, ES:set RootException) {
    #ES =< #{R:A.Roles | #(R.Raises & ES) > 0}
  + #{NAA:A.NestedActions | #(NAA.External & ES) > 0} }

pred actions_consistent() {
  all A:Action | action_consistent(A)
}

assert ActionsConsistent {
  actions_consistent()
}

pred role_consistent(R:Role, A:Action) {
  R.Encounters = R.ComponentActions.External
  R.Resolved = Keys.(A.ResolvedTo)
  no (R.Handles & R.Aborts)
  R.Aborts in R.Resolved

  thrown_exceptions_set_consistent(R, R.Signals,
    R.GeneratesSignaling, R.Resolved)

//  role_is_context(R)
  role_is_not_context(R)
}

// Consistency checks for EHS where roles ARE exception
// handling contexts.
pred role_is_context(R:Role) {
  // Computes the Raises set of a role for EHS where both roles
  // and actions are EH contexts.
  thrown_exceptions_set_consistent(R, R.Raises, R.GeneratesRaising,
    R.Encounters)
  (R.Encounters + R.Resolved - R.Handles) <:
    (R.Propagates) = R.Propagates
  (R.Propagates) :> (R.Signals + R.Raises) = R.Propagates
  R.Handles in (R.Resolved + R.Encounters)
}

// Consistency checks for EHS where roles ARE NOT exception
// handling contexts.
pred role_is_not_context(R:Role) {
  // Computes the Raises set of a role for EHS where only actions
  // are EH contexts (therefore, encountered exceptions can not be
  // handled or explicitly propagated).
  R.Raises = R.Encounters + R.GeneratesRaising
  (R.Resolved - R.Handles) <: (R.Propagates) = R.Propagates
  (R.Propagates) :> (R.Signals) = R.Propagates
  R.Handles in R.Resolved
}

// This predicate checks whether a set of thrown exceptions is
// consistent. "Thrown" means either signaled or raised.
// "Generated" is the set of exceptions that are generated by
// role "R". "Received" means either "Encountered" or "Resolved",
// depending on the meaning of "Thrown".
pred thrown_exceptions_set_consistent(R:Role,
    Thrown:set RootException, Generated:set RootException,
    Received:set RootException) {
  (let RealRes = (Received - R.Handles) | Thrown = Generated
    +((RealRes - {E:RealRes|some E.(R.Propagates)})
    + RealRes.(R.Propagates)))
}

pred roles_consistent() {
  all A:Action | all R:A.Roles | role_consistent(R, A)
}

assert RolesConsistent {
  roles_consistent()
}

// Desired property -> No top-level (non-nested, non-component)
// CA action has external exceptions. This is not a fundamental
// property of CA actions, but a desirable property that indicates
// a safe use of CA actions.
pred top_level_actions_safe() {
  all A1:Action | ((no A2:Action | A1 in A2.NestedActions)
    && (no R:Role | A1 in R.ComponentActions)) => (no A1.External)
}

assert TopLevelActionsSafe {
  top_level_actions_safe()
}

// Desired property -> No internal exception of a CA action is
// visible to the outside world. "Internal", in this case,
// includes the action's Internal set and the Resolved sets of
// its roles. All internal exceptions are either handled (and
// normal activity is resumed) or propagated to something else.
// It must be stressed that this predicate does not impose any
// restrictions on an the external exceptions of an action.
pred internal_exceptions_not_visible_outside() {
  all A:Action | no (A.External & A.Internal) &&
    no (A.External & A.Roles.Resolved)
}

assert InternalExceptionsNotVisibleOutside {
  internal_exceptions_not_visible_outside()
}

// Basic-property -> No action nesting/composition cycles. This
// property must take into account situations where action and
// composition are combined. It is not enough to just verify if the
// action A is in the (non-reflexive) transitive closure of
// A.NestedActions and A.Roles.ComponentActions (a top-down
// approach). The check must be performed bottom-up.
pred no_action_nesting_cycles() {
  no A:Action| A in A.^(~(Roles.ComponentActions + NestedActions))
}

assert NoActionNestingCycles {
  no_action_nesting_cycles()
}

pred basic_properties() {
  all A:Action | action_consistent(A) && all R:A.Roles |
    role_consistent(R, A)
}

assert BasicProperties {
  basic_properties()
}
```

Apêndice B

Fault-Tolerant Insulin Pump Therapy in Alloy

```
module FTIPT

open CoordinatedExceptionHandling

// ACTIONS
one sig CAACycle, CAAChecking, CAAExecuting extends Action {}
one sig CAASensors extends Action {}

one sig CAAActuators extends Action {}
one sig CAARAIP, CAALAIP extends Action {}

// ROLES
one sig ParamsCAACycle, ParamsCAAChecking, ParamsCAAExecuting
  extends Role {}
one sig ControllerCAACycle, ControllerCAAChecking,
  ControllerCAAExecuting extends Role {}
one sig Calculus extends Role {}
one sig S_CT, BGC, HR extends Role {}

one sig A_CT, A_RAIP, A_LAIP extends Role {}
one sig RAIP, SensorRAIP extends Role {}
one sig LAIP, SensorLAIP extends Role {}

one sig K1, K2, K3 extends Keys {}

one sig Params_Part, Controller_Part, Calculus_Part
```

```
  extends Participant {}
one sig S_CT_Part, BGC_Part, HR_Part extends Participant {}
one sig A_CT_Part, A_RAIP_Part, A_LAIP_PArt extends Participant {}
one sig RAIP_Part, SensorRAIP_Part extends Participant {}
one sig LAIP_Part, SensorLAIP_Part extends Participant {}

// EXCEPTIONS
one sig E1, E2, E3, AlarmEXC extends RootException {}
one sig E4, E5, E6, E7 extends RootException {}

fact SystemStructure {
  CAACycle.Roles = ParamsCAACycle + ControllerCAACycle + Calculus
  CAAChecking.Roles = ParamsCAAChecking + ControllerCAAChecking
  CAAExecuting.Roles = ParamsCAAExecuting + ControllerCAAExecuting
  CAASensors.Roles = S_CT + BGC + HR
  CAACycle.NestedActions = CAAChecking + CAAExecuting
  no CAAChecking.NestedActions
  no CAAExecuting.NestedActions
  no CAASensors.NestedActions

  Params_Part.RolesPlayed = ParamsCAAExecuting + ParamsCAAChecking
    + ParamsCAACycle
  Controller_Part.RolesPlayed = ControllerCAACycle
    + ControllerCAAChecking + ControllerCAAExecuting
  Calculus_Part.RolesPlayed = Calculus
  S_CT_Part.RolesPlayed = S_CT
  BGC_Part.RolesPlayed = BGC
  HR_Part.RolesPlayed = HR
  A_CT_Part.RolesPlayed = A_CT
  A_RAIP_Part.RolesPlayed = A_RAIP
  A_LAIP_PArt.RolesPlayed = A_LAIP
  RAIP_Part.RolesPlayed = RAIP
  SensorRAIP_Part.RolesPlayed = SensorRAIP
  LAIP_Part.RolesPlayed = LAIP
  SensorLAIP_Part.RolesPlayed = SensorLAIP

  CAAActuators.Roles = A_CT + A_RAIP + A_LAIP
  CAARAIP.Roles = RAIP + SensorRAIP
  CAALAIP.Roles = LAIP + SensorLAIP
  no CAAActuators.NestedActions
  no CAARAIP.NestedActions
  no CAALAIP.NestedActions

  ControllerCAAChecking.ComponentActions = CAASensors
  ControllerCAAExecuting.ComponentActions = CAAActuators
  no ControllerCAACycle.ComponentActions
  no ParamsCAACycle.ComponentActions
  no ParamsCAAExecuting.ComponentActions
  no ParamsCAAChecking.ComponentActions
  no Calculus.ComponentActions
  no S_CT.ComponentActions
  no BGC.ComponentActions
  no HR.ComponentActions

  no A_CT.ComponentActions
  A_RAIP.ComponentActions = CAARAIP
  A_LAIP.ComponentActions = CAALAIP
  no RAIP.ComponentActions
  no LAIP.ComponentActions
  no SensorLAIP.ComponentActions
  no SensorRAIP.ComponentActions
}

fact ExceptionFlow {
  no CAACycle.External
  CAACycle.Internal = E3 + AlarmEXC
  CAACycle.ToResolve = E3->K1 + AlarmEXC->K2 + (AlarmEXC + E3)->K3
  CAACycle.ResolvedTo = K1->AlarmEXC + K2->AlarmEXC + K3->AlarmEXC
  no CAACycle.AbortException
  no CAACycle.FailException
  no CAACycle.Excluding

  CAAChecking.External = AlarmEXC
  CAAChecking.Internal = AlarmEXC
  CAAChecking.ToResolve = AlarmEXC->K1
  CAAChecking.ResolvedTo = K1->AlarmEXC
  no CAAChecking.AbortException
  no CAAChecking.FailException
  no CAAChecking.Excluding

  CAAExecuting.External = AlarmEXC
  CAAExecuting.Internal = AlarmEXC
  CAAExecuting.ToResolve = AlarmEXC->K1
  CAAExecuting.ResolvedTo = K1->AlarmEXC
  no CAAExecuting.AbortException
  no CAAExecuting.FailException
  no CAAExecuting.Excluding

  CAASensors.External = AlarmEXC
  CAASensors.Internal = E1 + E2
  CAASensors.ToResolve = E1->K1 + E2->K2 + (E1 + E2)->K3
  CAASensors.ResolvedTo = K1->AlarmEXC + K2->AlarmEXC + K3->AlarmEXC
  no CAASensors.AbortException
  no CAASensors.FailException
  no CAASensors.Excluding

  CAAActuators.External = AlarmEXC
  CAAActuators.Internal = AlarmEXC
  CAAActuators.ToResolve = AlarmEXC->K1
  CAAActuators.ResolvedTo = K1->AlarmEXC
  no CAAActuators.AbortException
  no CAAActuators.FailException
  no CAAActuators.Excluding

  CAARAIP.External = AlarmEXC
  CAARAIP.Internal = E4 + E5
  CAARAIP.ToResolve = E4->K1 + E5->K2
  CAARAIP.ResolvedTo = K1->AlarmEXC + K2->AlarmEXC
  no CAARAIP.AbortException
  no CAARAIP.FailException
  no CAARAIP.Excluding

  CAALAIP.External = AlarmEXC
  CAALAIP.Internal = E6 + E7
  CAALAIP.ToResolve = E6->K1 + E7->K2
  CAALAIP.ResolvedTo = K1->AlarmEXC + K2->AlarmEXC
  no CAALAIP.AbortException
  no CAALAIP.FailException
  no CAALAIP.Excluding

  no Calculus.Signals
  Calculus.Raises = E3
  Calculus.GeneratesRaising = E3
```

```
no Calculus.GeneratesSignaling
Calculus.Resolved = AlarmEXC
no Calculus.Encounters
Calculus.Handles = AlarmEXC
no Calculus.Propagates
no Calculus.Aborts

no ControllerCAACycle.Signals
no ControllerCAACycle.Raises
no ControllerCAACycle.GeneratesRaising
no ControllerCAACycle.GeneratesSignaling
ControllerCAACycle.Resolved = AlarmEXC
no ControllerCAACycle.Encounters
ControllerCAACycle.Handles = AlarmEXC
no ControllerCAACycle.Propagates
no ControllerCAACycle.Aborts

no ParamsCAACycle.Signals
no ParamsCAACycle.Raises
no ParamsCAACycle.GeneratesRaising
no ParamsCAACycle.GeneratesSignaling
ParamsCAACycle.Resolved = AlarmEXC
no ParamsCAACycle.Encounters
ParamsCAACycle.Handles = AlarmEXC
no ParamsCAACycle.Propagates
no ParamsCAACycle.Aborts

ControllerCAAChecking.Signals = AlarmEXC
ControllerCAAChecking.Raises = AlarmEXC
no ControllerCAAChecking.GeneratesRaising
no ControllerCAAChecking.GeneratesSignaling
ControllerCAAChecking.Resolved = AlarmEXC
ControllerCAAChecking.Encounters = AlarmEXC
no ControllerCAAChecking.Handles
no ControllerCAAChecking.Propagates
no ControllerCAAChecking.Aborts

ParamsCAAChecking.Signals = AlarmEXC
no ParamsCAAChecking.Raises
no ParamsCAAChecking.GeneratesRaising
no ParamsCAAChecking.GeneratesSignaling
```

```
ParamsCAAChecking.Resolved = AlarmEXC
no ParamsCAAChecking.Encounters
no ParamsCAAChecking.Handles
no ParamsCAAChecking.Propagates
no ParamsCAAChecking.Aborts

ControllerCAAExecuting.Signals = AlarmEXC
ControllerCAAExecuting.Raises = AlarmEXC
no ControllerCAAExecuting.GeneratesRaising
no ControllerCAAExecuting.GeneratesSignaling
ControllerCAAExecuting.Resolved = AlarmEXC
ControllerCAAExecuting.Encounters = AlarmEXC
no ControllerCAAExecuting.Handles
no ControllerCAAExecuting.Propagates
no ControllerCAAExecuting.Aborts

ParamsCAAExecuting.Signals = AlarmEXC
no ParamsCAAExecuting.Raises
no ParamsCAAExecuting.GeneratesRaising
no ParamsCAAExecuting.GeneratesSignaling
ParamsCAAExecuting.Resolved = AlarmEXC
no ParamsCAAExecuting.Encounters
no ParamsCAAExecuting.Handles
no ParamsCAAExecuting.Propagates
no ParamsCAAExecuting.Aborts

S_CT.Signals = AlarmEXC
no S_CT.Raises
no S_CT.GeneratesRaising
no S_CT.GeneratesSignaling
S_CT.Resolved = AlarmEXC
no S_CT.Encounters
no S_CT.Handles
no S_CT.Propagates
no S_CT.Aborts

BGC.Signals = AlarmEXC
BGC.Raises = E1
BGC.GeneratesRaising = E1
no BGC.GeneratesSignaling
BGC.Resolved = AlarmEXC
```

```
no BGC.Encounters
no BGC.Handles
no BGC.Propagates
no BGC.Aborts

HR.Signals = AlarmEXC
HR.Raises = E2
HR.GeneratesRaising = E2
no HR.GeneratesSignaling
HR.Resolved = AlarmEXC
no HR.Encounters
no HR.Handles
no HR.Propagates
no HR.Aborts

A_CT.Signals = AlarmEXC
no A_CT.Raises
no A_CT.GeneratesRaising
no A_CT.GeneratesSignaling
A_CT.Resolved = AlarmEXC
no A_CT.Encounters
no A_CT.Handles
no A_CT.Propagates
no A_CT.Aborts

A_RAIP.Signals = AlarmEXC
A_RAIP.Raises = AlarmEXC
no A_RAIP.GeneratesRaising
no A_RAIP.GeneratesSignaling
A_RAIP.Resolved = AlarmEXC
A_RAIP.Encounters = AlarmEXC
no A_RAIP.Handles
no A_RAIP.Propagates
no A_RAIP.Aborts

A_LAIP.Signals = AlarmEXC
A_LAIP.Raises = AlarmEXC
no A_LAIP.GeneratesRaising
no A_LAIP.GeneratesSignaling
A_LAIP.Resolved = AlarmEXC
A_LAIP.Encounters = AlarmEXC
```

```
no A_LAIP.Handles
no A_LAIP.Propagates
no A_LAIP.Aborts

RAIP.Signals = AlarmEXC
no RAIP.Raises
no RAIP.GeneratesRaising
no RAIP.GeneratesSignaling
RAIP.Resolved = AlarmEXC
no RAIP.Encounters
no RAIP.Handles
no RAIP.Propagates
no RAIP.Aborts

SensorRAIP.Signals = AlarmEXC
SensorRAIP.Raises = E4 + E5
SensorRAIP.GeneratesRaising = E4 + E5
no SensorRAIP.GeneratesSignaling
SensorRAIP.Resolved = AlarmEXC
no SensorRAIP.Encounters
no SensorRAIP.Handles
no SensorRAIP.Propagates
no SensorRAIP.Aborts

LAIP.Signals = AlarmEXC
no LAIP.Raises
no LAIP.GeneratesRaising
no LAIP.GeneratesSignaling
LAIP.Resolved = AlarmEXC
no LAIP.Encounters
no LAIP.Handles
no LAIP.Propagates
no LAIP.Aborts

SensorLAIP.Signals = AlarmEXC
SensorLAIP.Raises = E6 + E7
SensorLAIP.GeneratesRaising = E6 + E7
no SensorLAIP.GeneratesSignaling
SensorLAIP.Resolved = AlarmEXC
no SensorLAIP.Encounters
no SensorLAIP.Handles
```

```
  no SensorLAIP.Propagates
  no SensorLAIP.Aborts

}

// Application-specific property. For all the actions except for
// the top-level CA action CAACycle, if AlarmEXC is an internall
// exception, it is also one external exception.
pred app_specific_properties() {
  all A: (Action - CAACycle) |
    AlarmEXC in A.Internal => AlarmEXC in A.External
}

assert AppSpecificProperties {
  app_specific_properties()
}
```

Apêndice C

Partial Specifications of Generic CA Actions Model and Fault-Tolerant Insulin Pump Therapy in B

```
MACHINE           CoordinatedExceptionHandling
/*
  Static B machine for Coordinated Exception Handling.
*/

SETS              ACTION = {CAACycle, CAAChecking};
                  PARTICIPANT = {P1, P2, P3};
                  ROLE = {Calculus, Controller, Params,
                    ControllerChecking, ParamsChecking};
                  ROOT_EXCEPTION = {E3, E4}; OK = {yes, no}

                  /* Variables of Action */
VARIABLES         Internal, External, Roles, NestedActions,
                  AbortException, FailException, Resolution,
                  Excluding,

                  /* Variables of Role */
                  Signals, Raises, Generates, Resolved, Encounters,
                  Handles, ComponentActions, Propagates, Aborts,

                  /* Variables of Participant */
                  RolesPlayed,
                  /* Results of operations */
                  ActionsConsistent, ParticipantsConsistent,
                  RolesConsistent

INVARIANT          Roles : ACTION <-> ROLE
                & Internal : ACTION <-> ROOT_EXCEPTION
                & External : ACTION <-> ROOT_EXCEPTION
                & NestedActions : ACTION <-> ACTION
                & AbortException : ACTION +-> ROOT_EXCEPTION
                & FailException : ACTION +-> ROOT_EXCEPTION
                & Resolution : ACTION <-> (POW(ROOT_EXCEPTION)
                  +-> ROOT_EXCEPTION)
                & Excluding : ACTION <-> POW(ROOT_EXCEPTION)
                & card(AbortException) <= 1 & card(FailException) <= 1
                & Signals : ROLE <-> ROOT_EXCEPTION
                & Raises : ROLE <-> ROOT_EXCEPTION
                & Generates : ROLE <-> ROOT_EXCEPTION
                & Resolved : ROLE <-> ROOT_EXCEPTION
                & Encounters : ROLE <-> ROOT_EXCEPTION
                & Handles : ROLE <-> ROOT_EXCEPTION
                & Aborts : ROLE <-> ROOT_EXCEPTION
                & ComponentActions : ROLE <-> ACTION
                & Propagates : ROLE <->
                  (ROOT_EXCEPTION +-> ROOT_EXCEPTION)
                & RolesPlayed : PARTICIPANT <-> ROLE
                & ActionsConsistent : OK
                & ActionsConsistent = yes
                & ParticipantsConsistent : OK
                & ParticipantsConsistent = yes
                & RolesConsistent : OK
                & RolesConsistent = yes

INITIALISATION     Roles := { CAACycle |-> Calculus,
                    CAACycle |-> Controller, CAACycle |-> Params,
                    CAAChecking |-> ControllerChecking,
                    CAAChecking |-> ParamsChecking}
               || External := {} || Internal := { CAACycle |-> E3}
               || NestedActions  := { CAACycle |-> CAAChecking}
               || AbortException := {}
               || FailException := {}
               || Resolution := {CAACycle |-> {{E3} |-> E3}}
```

```
                  || Signals := {} || Excluding := {}
                  || Raises := {Calculus |-> E3}
                  || Generates := {Calculus |-> E3}
                  || Resolved := {Calculus |-> E3, Controller |-> E3,
                    Params |-> E3}
                  || Encounters := {} || Handles := {Calculus |-> E3,
                    Controller |-> E3, Params |-> E3}
                  || Aborts := {} || ComponentActions := {}
                  || Propagates := {}
                  || RolesPlayed := {P1 |-> Controller, P2 |-> Calculus,
                    P3 |-> Params, P1 |-> ControllerChecking,
                    P2 |-> ParamsChecking}
                  || ActionsConsistent := yes
                  || ParticipantsConsistent := yes
                  || RolesConsistent := yes

OPERATIONS

actionConsistent =
  IF !Act.( (Act : ACTION) =>
         (ran({Act} <| Internal) = ran(ran({Act} <| Roles) <| Raises)
           \/ ran(ran({Act} <| NestedActions) <| External))
       & (ran({Act} <| External) = ran(ran({Act} <| Roles)
           <| Signals))
       & (ran(union(Resolution[{Act}])) <: Resolved[Roles[{Act}]])
       & (card({E | E:ROOT_EXCEPTION & E:ran(union(Resolution[{Act}]))
           & (!R.((R:ROLE & R:Roles[{Act}])=> E:Aborts[{R}]))}) > 0 =>
               (#AE.(AE:ROOT_EXCEPTION & AE = AbortException(Act)))
         )
       & (card({E | E:ROOT_EXCEPTION & E:ran(union(Resolution[{Act}]))
           & (!R.((R:ROLE & R:Roles[{Act}])=> E:Aborts[{R}]))}) = 0 =>
               not(#AE.(AE:ROOT_EXCEPTION & AE = AbortException(Act)))
         )
       & (#E.(E:ROOT_EXCEPTION & E:ran(union(Resolution[{Act}]))
            & !R.((R:ROLE & R:Roles[{Act}]) => not(E:Handles[{R}]
              or E:Aborts[{R}]))
          ) =>
           ( (card(Signals[Roles[{Act}]] \/
               ran({Act} <| FailException)) > 1 =>
                 (#FE.(FE:ROOT_EXCEPTION & FE = FailException(Act)))
           &
```

```
              (card(Signals[Roles[{Act}]]
                \/ ran({Act} <| FailException)) <= 1 =>
                  not(#FE.(FE:ROOT_EXCEPTION & FE = FailException(Act))))
            )
          )
        & ((dom(union(Resolution[{Act}])) /\ Excluding[{Act}]) = {})
        & (!ES.((ES:POW(ROOT_EXCEPTION) &
          (ES:(dom(union(Resolution[{Act}])) \/ Excluding[{Act}]))) =>
                 card(ES) <=
                   ( card({R|R:ROLE & R:Roles[{Act}]
                     & card(Raises[{R}] /\ ES) > 0})
                   + card({NA|NA:ACTION & NA:NestedActions[{Act}]
                     & card(External[{NA}] /\ ES) > 0})
                   )
              )
          )
      )
  THEN ActionsConsistent := yes
  ELSE ActionsConsistent := no
  END;

roleConsistent =
  IF !Act.((Act:ACTION) =>
       !R.((R:ROLE & R:Roles[{Act}]) =>
           (Encounters[{R}] = External[ComponentActions[{R}]])
         & (Resolved[{R}] = ran(union(Resolution[{Act}])))
         & ((Handles[{R}] /\ Aborts[{R}]) = {})
         & (Aborts[{R}] <: Resolved[{R}])
         & (Signals[{R}] = Resolved[{R}] - Handles[{R}]
                         - ((Resolved[{R}] - Handles[{R}])
                           /\ dom(union(Propagates[{R}])))
                         \/ (union(Propagates[{R}]))[(Resolved[{R}]
                           - Handles[{R}])]

           )
         & (Raises[{R}] = Encounters[{R}] \/ Generates[{R}])
         & (dom(union(Propagates[{R}])) <: Resolved[{R}]
           - Handles[{R}])
         & (ran(union(Propagates[{R}])) <: Signals[{R}])
         & (Handles[{R}] <: Resolved[{R}])
       )
```

```
     )
   THEN RolesConsistent := yes
   ELSE RolesConsistent := no
   END;

 participantsConsistent =
   IF !Act.((Act:ACTION) =>
        (!NA.((NA:ACTION & NA:NestedActions[{Act}]) =>
           (!P.((P:PARTICIPANT) => card(RolesPlayed[{P}]
             /\ Roles[{Act}]) <= 1))
          & (Roles[{Act}] <: RolesPlayed[PARTICIPANT])
          & (!NAR.((NAR:ROLE & NAR:Roles[{NA}]) =>
                  #P.(P:PARTICIPANT & NAR:RolesPlayed[{P}]
                    & card(RolesPlayed[{P}] /\ Roles[{Act}]) > 0)
                )
           )
        ))
     )
   THEN ParticipantsConsistent := yes
   ELSE ParticipantsConsistent := no
   END

 END;


 END
```

Referências Bibliográficas

[1] M. Abi-Antoun, J. Aldrich, D. Garlan, B. R. Schmerl, N. H. Nahas, and T. Tseng. Modeling and implementing software architecture with acme and archjava. In *Proceedings of the 27th International Conference in Software Engineering (ICSE'2005)*, pages 676–677, St. Louis, USA, May 2005.

[2] G. Abowd, R. Allen, and D. Garlan. Formalizing style to understand descriptions of software architecture. *ACM Transactions on Software Engineering and Methodology*, 4(4):319–364, October 1995.

[3] J. R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

[4] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997.

[5] T. Anderson and P. A. Lee. *Fault Tolerance: Principles and Practice.* Springer-Verlag, 2nd edition, 1990.

[6] P. C. Attie, D. H. Lorenz, A. Portnova, and H. Chockler. Behavioral compatibility without state explosion: Design and verification of a component-based elevator control system. In *Proceedings of the 9th International Symposium on Component-Based Software Engineering*, volume 4063 of *LNCS*, pages 33–49, 2006.

[7] P. Avgustinov, A. S. Christensen, L. J. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: an extensible aspectj compiler. In *Proceedings of 4th ACM Conference on Aspect-Oriented Software Development*, pages 87–98, March 2005.

[8] A. Avizienis. Towards systematic design of fault-tolerant systems. *IEEE Computer*, 30(4):51–58, April 1997.

[9] F. Bachman, L. Bass, C. Buhman, F. Long, J. Robert, R. Seacord, and K. Wallnau. Volume ii: Technical concepts of component-based software engineering. Technical

Report CMU/SEI-2000-TR-008, Software Engineering Institute, Carnegie-Mellon University, April 2000.

[10] L. Bass, P. C. Clements, and R. Kazman. Air traffic control: A case study in designing for high availability. In *Software Architecture in Practice*, chapter 6. Addison-Wesley, 2nd edition, 2003.

[11] L. Bass, P. C. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 2nd edition, 2003.

[12] C. Bernardeschi, A. Fantechi, and S. Gnesi. Model checking fault tolerant systems. *Software Testing, Verification, and Reliability*, 12:251–275, December 2002.

[13] S. Bodoff. *The J2EE Tutorial*. Addison-Wesley, 2004.

[14] P. H. S. Brito. A method for modeling exceptions in component-based development (in portuguese). Master's thesis, State University of Campinas, Brazil, 2005.

[15] A. Brown and K. Wallnau. The current state of CBSE. *IEEE Software*, 15(5):37–46, September/October 1998.

[16] M. Bruntink, A. van Deursen, and T. Tourwé. Discovering faults in idiom-based exception handling. In *Proc. of the 28th ICSE*, pages 242–251, Shangai, China, 2006.

[17] P. A. Buhr and R. Mok. Advanced exception handling mechanisms. *IEEE Transactions on Software Engineering*, 26(9):1–16, September 2000.

[18] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley and Sons, West Sussex, England, 1996.

[19] N. Cacho, C. Sant'Anna, E. Figueiredo, A. Garcia, and T. B. C. Lucena. Composing design patterns: A scalability study of aspect-oriented programming. In *Proceedings of 5th ACM Conference on Aspect-Oriented Software Development*, pages 109–121, March 2006.

[20] R. H. Campbell and B. Randell. Error recovery in asynchronous systems. *IEEE Transactions on Software Engineering*, SE-12(8):811–826, 1986.

[21] A. Capozucca, N. Guelfi, and P. Pelliccione. The fault-tolerant insulin pump therapy. In *Proceedings of FM'2005 Workshop on Rigorous Engineering of Fault-Tolerant Systems*, pages 33–42, Newcastle upon Tyne, UK, July 2005.

[22] T. Cargill. Exception handling: A false sense of security. *C++ Report*, 6(9), November-December 1994.

[23] F. Castor Filho, R. M. ao Ferreira, C. M. F. Rubira, and A. Garcia. Aspectizing exception handling: A quantitative study. In C. Dony et al., editors, *Advanced Topics in Exception Handling Techniques*, volume 4119 of *LNCS*, pages 255–274. Springer-Verlag, 2006.

[24] F. Castor Filho, N. Cacho, E. Figueiredo, R. Ferreira, A. Garcia, and C. M. F. Rubira. Exceptions and aspects: The devil is in the details. In *Proceedings of the 14th ACM SIGSOFT Symposium on Foundations of Software Engineering*, November 2006. To appear.

[25] F. Castor Filho, P. H. da S. Brito, and C. M. F. Rubira. A framework for analyzing exception flow in software architectures. In *Proceedings of the ICSE'2005 Workshop on Architecting Dependable Systems*, May 2005.

[26] F. Castor Filho, P. H. da S. Brito, and C. M. F. Rubira. Modeling and analysis of architectural exceptions. In *Proceedings of the FM'2005 Workshop on Rigorous Engineering of Fault-Tolerant Systems*, pages 112–121, July 2005.

[27] F. Castor Filho, P. H. da S. Brito, and C. M. F. Rubira. Reasoning about exception flow at the architectural level. In M. Butler et al., editors, *Rigorous Development of Complex Fault-Tolerant Systems*, volume 4157 of *LNCS*, pages 80–99. Springer-Verlag, 2006.

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

[29] F. Castor Filho, P. A. de C. Guerra, V. A. Pagano, and C. M. F. Rubira. A systematic approach for structuring exception handling in robust component-based software. *Journal of the Brazilian Computer Society*, 10(3):5–19, April 2005.

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

[31] F. Castor Filho, P. A. de C. Guerra, and C. M. F. Rubira. FaTC2: An object-oriented framework for developing fault-tolerant component-based systems. In *Proceedings of the ICSE'2003 Workshop on Software Architectures for Dependable Systems*, pages 13–18, May 2003.

[32] F. Castor Filho, A. Garcia, and C. M. F. Rubira. A quantitative study on the aspectization of exception handling. In *Proceedings of ECOOP'2005 Workshop on Exception Handling in Object-Oriented Systems*, 2005.

[33] F. Castor Filho, A. Garcia, and C. M. F. Rubira. Implementing modular error handling with aspects: Best and worst practices. Technical Report IC-06-22, Institute of Computing, State University of Campinas, 2006.

[34] F. Castor Filho, A. Romanovsky, and C. M. F. Rubira. Verification of coordinated exception handling. Technical Report CS-TR-927, School of Computing Science, University of Newcastle upon Tyne, 2005.

[35] F. Castor Filho, A. Romanovsky, and C. M. F. Rubira. Verification of coordinated exception handling. In *Proceedings of the 21st ACM Symposium on Applied Computing*, pages 680–685, Dijon, France, April 2006.

[36] F. Castor Filho and C. M. F. Rubira. Implementing coordinated error recovery for distributed object-oriented systems with AspectJ. *Journal of Universal Computer Science*, 10(7):843–858, July 2004.

[37] F. Castor Filho and C. M. F. Rubira. Implementing coordinated exception handling for distributed object-oriented systems with AspectJ. In *Proceedings of the VIII Brazilian Symposium on Programming Languages*, pages 128–142, May 2004.

[38] B.-M. Chang et al. Interprocedural exception analysis for java. In *Proceedings of the 16th ACM Symposium on Applied Computing*, pages 620–625, March 2001.

[39] M. Chapman, A. Vasseur, and G. Kniesel (eds.). Aosd 2006 - industry track proceedings. Technical Report IAI-TR-2006-3, Computer Science Department III, University of Bonn, March 2006. ISSN 0944-8535.

[40] C. Chavez. *A Model-Driven Approach for Aspect-Oriented Design*. PhD thesis, Pontiffficia Universidade Católica do Rio de Janeiro, April 2004.

[41] J. Cheesman and J. Daniels. *UML Components*. Addison-Wesley, 2000.

[42] S. Chiba. Load-time structural reflection in java. In *Proceedings of the 14th European Conference on Object-Oriented Programming*, volume 1850 of *LNCS*, pages 313–336, 2000.

[43] S. Chidamber and C. Kemerer. A metrics suite for oo design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[44] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, January 2000.

[45] ClearSy. ClearSy AtelierB v3.6, October 26th 2006. Address: http://www.atelierb.societe.com/index_uk.htm.

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

[47] P. C. Clements, R. Kazman, and M. Klein. *Evaluating Software Architectures*. Addison-Wesley, 2003.

[48] P. C. Clements and L. Northrop. Software architecture: An executive overview. Technical Report CMU/SEI-96-TR-003, SEI/CMU, February 1996.

[49] L. Cole and P. Borba. Deriving refactorings for aspectj. In *Proceedings of the 4th ACM Conference on Aspect-Oriented Software Development*, pages 123–134, March 2005.

[50] J. E. Cook and J. A. Dage. Highly reliable upgrading of components. In *Proceedings of the 21st International Conference on Software Engineering*, pages 203–212, May 1999.

[51] F. Cristian. A recovery mechanism for modular software. In *Proceedings of the 4th International Conference on Software Engineering*, pages 42–51, 1979.

[52] F. Cristian. Exception handling. In T. Anderson, editor, *Dependability of Resilient Computers*, pages 68–97. Blackwell Scientific Publications, 1989.

[53] Q. Cui and J. Gannon. Data-oriented exception handling. *IEEE Transactions on Software Engineering*, 18(5):393–401, May 1992.

[54] P. H. da S. Brito, C. R. Rocha, F. Castor Filho, E. Martins, and C. M. F. Rubira. A method for modeling and testing exceptions in component-based software development. In *Proceedings of the 2nd Latin American Symposium on Dependable Computing*, volume 3747 of *LNCS*, pages 61–79, 2005.

[55] E. M. Dashofy, N. Medvidovic, and R. N. Taylor. Using off-the-shelf middleware to implement connectors in distributed software architectures. In *Proceedings of the 21st International Conference on Software Engineering*, pages 3–12, May 1999.

[56] P. A. de C. Guerra, C. M. F. Rubira, and R. de Lemos. An idealized fault-tolerant architectural component. In *Proceedings of ICSE Workshop on Architecting Dependable Systems*, May 2002.

[57] R. de Lemos. Describing evolving dependable systems using co-operative software architectures. In *Proceedings of the 2001 IEEE International Conference on Software Maintenance*, pages 320–331, Florence, Italy, November 2001.

[58] R. de Lemos, P. A. de Castro Guerra, and C. M. F. Rubira. A fault-tolerant architectural approach for dependable systems. *IEEE Software*, 23(2):80–87, 2006.

[59] R. de Lemos and A. Romanovsky. Exception handling in the software lifecycle. *International Journal of Computer Science and Engineering*, 16(2):167–181, 2001.

[60] G. Doshi. Best practices for exception handling, 2003. Address: http://www.onjava.com/pub/a/onjava/2003/11/19/exceptions.html.

[61] D. D'Souza and A. C. Wills. *Objects, Components and Frameworks with UML: The Catalysis Approach.* Addison-Wesley, 1998.

[62] Eclipse. The eclipse project. http://www.eclipse.org, 2003.

[63] T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming: Introduction. *Communications of the ACM*, 44(10):28–32, 2001.

[64] M. Fahndrich et al. Tracking down exceptions in standard ml. Technical Report CSD-98-996, University of California, Berkeley, 1998.

[65] G. R. M. Ferreira, C. M. F. Rubira, and R. de Lemos. Explicit representation of exception handling in the development of dependable component-based systems. In *Proceedings of International Symposium on High Assurance Software Engineering*, Boca Raton, FL, USA., October 2001.

[66] C. Fetzer, K. Högstedt, and P. Felber. Automatic detection and masking of nonatomic exception handling. *IEEE Transactions on Software Engineering*, 30(8):547–560, 2004.

[67] R. Filman and D. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Proceedings of the OOPSLA'2000 Workshop on Advanced Separation of Concerns*, October 2000.

[68] M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[69] P. Fradet and M. Südolt. An aspect language for robust programming. In *Proceedings of the ECOOP'99 Workshop on Aspect-Oriented Programming*, June 1999.

[70] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Software Systems*. Addison-Wesley, 1995.

[71] A. Garcia, D. Beder, and C. Rubira. An exception handling mechanism for developing dependable object-oriented software based on a meta-level approach. In *Proceedings of the 10th IEEE International Symposium on Software Reliability Engineering*, pages 52–61, November 1999.

[72] A. Garcia, C. Rubira, A. Romanovsky, and J. Xu. A comparative study of exception handling mechanisms for building dependable object-oriented software. *Journal of Systems and Software*, 59(2):197–222, November 2001.

[73] A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. J. P. de Lucena, and A. von Staa. Modularizing design patterns with aspects: A quantitative study. *Transactions on Aspect-Oriented Programming*, 1:36–74, 2006.

[74] A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena, and A. von Staa. Modularizing design patterns with aspects: A quantitative study. In *Proceedings of the 4th ACM Conference on Aspect-Oriented Software Development*, pages 3–14, March 2005.

[75] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6):17–26, 1995.

[76] D. Garlan et al. Acme: Architectural description of component-based systems. In *Foundations of Component-Based Systems*, chapter 3. Cambridge University Press, 2000.

[77] D. Garlan and Z. Wang. A case study in software architecture interchange. In *Proceedings of COORDINATION'99*, 1999.

[78] J. Gleick. A Bug and a Crash, December 1st 1996. Disponível em http://www.around.com/ariane.html.

[79] I. Godil and H. Jacobsen. Horizontal decomposition of prevlayer. In *Proceedings of CASCON 2005*, 2005.

[80] J. B. Goodenough. Exception handling: Issues and a proposed notation. *Communications of the ACM*, 18(12):683–696, December 1975.

[81] I. Gorton and L. Zhu. Tool support for just-in-time architecture reconstruction and evaluation: an experience report. In *Proceedings of the 27th International*

Conference in Software Engineering (ICSE'2005), pages 514–523, St. Louis, USA, May 2005.

[82] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

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

[84] P. Greenwood and L. Blair. A framework for policy-driven auto-adaptive systems using dynamic framed aspects. *Trans. AOSD*, 2006. To appear.

[85] P. Greenwood and et al. On the impact of aspectual decompositions on design stability: An empirical study. In *Proc. of the 21st ECOOP*, July 2007. To appear.

[86] N. Guelfi, G. L. Cousin, and B. Ries. Engineering of dependable complex business processes using uml and coordinated atomic actions. In *Proceedings of International Workshop on Modeling Inter-Organizational Systems*, pages 468–482, 2004.

[87] P. Guerra, F. Castor Filho, V. A. Pagano, and C. Rubira. Structuring exception handling for dependable component-based software systems. In *Proceedings of 30th Euromicro Conference*, pages 575–582, Rennes, France, September 2004.

[88] P. Guerra, C. Rubira, A. Romanovsky, and R. de Lemos. Integrating COTS software componentes into dependable software architectures. In *Proceedings of the 6th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 139–142, May 2003.

[89] P. A. C. Guerra, C. M. F. Rubira, and R. de Lemos. A fault-tolerant architecture for component-based software systems. In R. de Lemos, C. Gracek, and A. Romanosvsky, editors, *Architecting Dependable Systems*, LNCS 2677. Springer-Verlag, 2003.

[90] S. Hanenberg, C. Oberschulte, and R. Unland. Refactoring of aspect-oriented software. In *Proceedings of 4th Net.ObjectDays Conference*, pages 19–35, September 2003.

[91] J. Hannemann and G. Kiczales. Design pattern implementation in java and AspectJ. In *Proceedings of 17th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 161–173, November 2002.

[92] M. Hapner, R. Burridge, R. Sharma, J. Fialli, and K. Stout. Java Message Service specification, April 2002. Version 1.1.

[93] B. Harbulot and J. R. Gurd. A join point for loops in aspectj. In *Proceedings of the 5th ACM Conference on Aspect-Oriented Software Development*, pages 63–74, Bonn, Germany, March 2006.

[94] Headquarters US Air Force Inspection and Safety Center. Software system safety, September 1985. AFISC SSH 1-1.

[95] A. Hejlsberg, S. Wiltamuth, and P. Golde. *The C# Programming Language*. Addison-Wesley, Reading, USA, 2003.

[96] IBM. IBM Software: Websphere MQ v6.0, October 26th 2006. Address: http://www-306.ibm.com/software/integration/wmq/v60/.

[97] V. Issarny and J. P. Banatre. Architecture-based exception handling. In *Proceedings of the 34th Annual Hawaii International Conference on System Sciences*, 2001.

[98] D. Jackson. Alloy: A lightweight object modeling notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290, April 2002.

[99] D. Jackson. Alloy home page, 2004. Available at `http://sdg.lcs.mit.edu/alloy/default.htm`.

[100] D. Jackson. Alloy home page, March 2006. Available at `http://sdg.lcs.mit.edu/alloy/default.htm`.

[101] D. Jackson, I. Schechter, and I. Shlyakhter. Alcoa: The alloy constraint analyzer. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 730–733, Limerick, Ireland, June 2000.

[102] J.-M. Jézéquel and B. Meyer. Design by contract: The lessons of ariane. *IEEE Computer*, 30(2):129–130, January 1997.

[103] S. Jiang et al. An approach to analyzing exception propagation. In *Proceedings of the 8th IASTED International Conference on Software Engineering and Applications*, November 2004.

[104] S. Johannes. *Software Engineering with Reusable Components*. Springer-Verlag, 1997.

[105] K. Kanoun, Y. Crouzet, A. Kalakech, A.-E. Rugina, and P. Rumeau. Benchmarking the dependability of windows and linux using postmark workloads. In *Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering*, pages 11–20, November 2005.

[106] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, LNCS 1271, pages 220–242, 1997.

[107] J. Kienzle and R. Guerraoui. AOP: Does it make sense? the case of concurrency and failures. In *Proceedings of 16th European Conference on Object-Oriented Programming*, pages 37–61, June 2002.

[108] A. Koening and B. Stroustrup. Exception handling for c++. *Jornal of Object-Oriented Programmings*, 3(2):16–33, July/August 1990.

[109] P. Krüchten. The 4+1 view model of software architecture. *IEEE Software*, pages 42–50, November 1995.

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

[111] U. Kulesza et al. Quantifying the effects of aspect-oriented programming: A maintenance study. In *Proc. of the 22nd ICSM*, pages 223–233, 2006.

[112] S. Lacourte. Exceptions in guide, an object-oriented language for distributed applications. In *Proceedings of the 5th European Conference on Object-Oriented Programming*, LNCS 512, pages 268–287, July 1991.

[113] R. Laddad. Aspect-oriented refactoring, parts 1 and 2, 2003. The Server Side, www.theserverside.com.

[114] R. Laddad. *AspectJ in Action*. Manning, 2003.

[115] M. Leuschel and M. J. Butler. Prob: A model checker for b. In *Proceedings of the International Symposium of Formal Methods Europe (FME'2003)*, LNCS 2805, pages 855–874, Pisa, Italy, 2003.

[116] N. G. Leveson. *Safeware: System Safety and Computers*. Addison-Wesley, 1995.

[117] M. Lippert and C. V. Lopes. A study on exception detection and handling using aspect-oriented programming. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 418–427, June 2000.

[118] B. Liskov and A. Snyder. Exception handling in clu. *IEEE Transactions on Software Engineering*, 6(5):546–558, 1979.

[119] A. E. C. Lobo, P. A. de C. Guerra, F. Castor Filho, and C. M. F. Rubira. A systematic approach for the evolution of reusable software components. In *Proceedings of ECOOP'2005 Workshop on Architecture-Centric Evolution*, Glasgow, UK, July 2005.

[120] D. Luckham and J. Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9):717–734, April 1995.

[121] P. Maes. Concepts and experiments in computational reflection. *ACM SIGPLAN Notices*, 22(12):147–155, March 1987.

[122] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings of the 5th European Software Engineering Conference*, pages 137–153, September 1995.

[123] E. Martins, C. M. Toyota, and R. L. Yanagawa. Constructing self-testable software components. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 151–160, Göteborg, Sweden, July 2001.

[124] J. Matharu. Reusing safety-critical software components. *COTS Journal*, 7(8), July/August 2005.

[125] T. McCune. Exception-handling antipatterns, 2006. Address: http://today.java.net/pub/a/today/2006/04/06/exception-handling-antipatterns.html.

[126] M. D. McIlroy. Mass-produced software componentes. In P. Naur and B. Randell, editors, *Software Engineering, Report on a conference sponsored by the NATO Science Committee*, pages 138–155, Garmisch, Germany, October 1969.

[127] N. Medvidovic, P. Oreizy, and R. N. Taylor. Reuse of off-the-shelf components in c2-style architectures. In *Proceedings of the 1997 ACM SIGSOFT Symposium on Software Reusability*, May 1997.

[128] N. Medvidovic and R. N. Taylor. A framework for classifying and comparing architecture description languages. In *Proceedings of Joint 5th ACM SIGSOFT Symposium on Foundations of Software Engineering/6th European Software Engineering Conference*, pages 60–76, September 1997.

[129] N. R. Mehta and N. Medvidovic. Composing architectural styles from architectural primitives. In *Proceedings of Joint 9th European Software Engineering Conference/11th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 347–350, September 2003.

[130] M. Mezini and K. Ostermann. Conquering aspects with caesar. In *Proceedings of the 2nd ACM Conference on Aspect-Oriented Software Development*, pages 90–99, March 2003.

[131] Microsoft Corporation. Distributed component object model, 2002. http://www.microsoft.com/com/tech/DCOM.asp.

[132] M. Mikic-Rakic and N. Medvidovic. Adaptable architectural middleware for programming-in-the-small-and-many. In *ACM/IFIP/USENIX International Middleware Conference*, pages 162–181, Rio de Janeiro, Brazil, June 2003.

[133] S. E. Mitchell, A. Burns, and A. J. Wellings. Mopping up exceptions. *ACM SIGAda Ada Letters*, 21(3):80–92, September 2001.

[134] R. Monroe. Capturing architecture design expertise with armani. Technical Report CMU-CS-96-163, Carnegie Mellon University, School of Computer Science, 1998.

[135] M. P. Monteiro and J. M. Fernandes. Towards a catalog of aspect-oriented refactorings. In *Proceedings of the 4th ACM Conference on Aspect-Oriented Software Development*, pages 111–122, March 2005.

[136] G. Murphy, R. Walker, E. Baniassad, M. Robillard, and M. Kersten. Does aspect-oriented programming work? *Communications of the ACM*, 44(10):75–77, October 2001.

[137] E. D. Nitto and D. Rosenblum. Exploiting adls to specify architectural styles induced by middleware infrastructures. In *Proceedings of the 21st International Conference on Software Engineering*, pages 13–22, May 1999.

[138] A. Oliva and L. E. Buzato. The design and implementation of guaraná. In *Proceedings of the 5th USENIX Conference on Object-Oriented Technologies & Systems*, pages 203–216, San Diego, USA, May 1999.

[139] OMG. The corba component model. http://www.omg.org/, 2003.

[140] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[141] D. M. Papurt. The use of exceptions. *Journal of Object-Oriented Programming*, 11(2):13–17, 32, May 1998.

[142] D. L. Parnas and H. 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.

[143] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992.

[144] H. Rajan and K. Sullivan. Eos: Instance-level aspects for integrated system design. In *Proceedings of Joint 9th European Conference on Software Engineering/11th SIGSOFT Symposium on Foundations of Software Engineering*, pages 291–306, Helsinki, Finland, September 2003.

[145] M. Rakic and N. Medvidovic. Increasing the confidence in off-the-shelf components: A software connector-based approach. In *Proceedings of the 2001 ACM SIGSOFT Symposium on Software Reusability*, pages 11–18, May 2001.

[146] B. Randell and J. Xu. The evolution of the recovery block concept. In *Software Fault Tolerance*, chapter 1, pages 1–21. John Wiley Sons Ltd., 1995.

[147] A. Regalado et al. 10 emerging technologies that will change the world. *Technology Review*, pages 97–113, January/February 2001.

[148] 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.

[149] R. A. Riemenschneider and V. Stavridou. The role of architecture description languages in component-based development: The sri perspective. In *Proceedings of the 21st International Conference on Software Engineering*, 1999.

[150] M. P. Robillard and G. C. Murphy. Static analysis to support the evolution of exception structure in object-oriented systems. *ACM Transactions on Software Engineering and Methodology*, 12(2):191–221, April 2003.

[151] C. R. Rocha and E. Martins. A strategy to improve component testability without source code. In S. Beydeda et al., editors, *SOQUA/TECOS*, LNI 58. GI, 2004.

[152] A. Romanovsky, P. Periorellis, and A. F. Zorzo. Structuring integrated web applications for fault tolerance. In *Proceedings of the 6th IEEE ISADS*, pages 99–106, 2003.

[153] R. Roshandel, B. R. Schmerl, N. Medvidovic, D. Garlan, and D. Zhang. Understanding tradeoffs among different architectural modeling approaches. In *Proceedings of the 4th Working IEEE / IFIP Conference on Software Architecture*, pages 47–56, Oslo, Norway, June 2004.

[154] C. M. F. Rubira, R. de Lemos, G. Ferreira, and F. Castor Filho. Exception handling in the development of dependable component-based systems. *Software – Practice and Experience*, 35(5):195–236, March 2005.

[155] T. Saridakis and V. Issarny. Developing dependable systems using software architecture. In *Proceedings of the 1st Working IFIP Conference on Software Architecture*, pages 83–104, February 1999.

[156] C. F. Schaefer and G. N. Bundy. Static analysis of exception handling in ada. *Software: Practice and Experience*, 23(10):1157–1174, October 1993.

[157] B. Schmerl and D. Garlan. Acmestudio: Supporting style-centered architecture development. In *Proceedings of the 26th International Conference on Software Engineering*, pages 704–705, May 2004.

[158] M. Shaw and P. Clements. A field guide to boxology: Preliminary classification of architectural styles for software systems. In *Proceedings of COMPSAC'96*, Washington, DC, USA, August 1996.

[159] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Addison-Wesley, 1996.

[160] A. Shui, S. Mustafiz, and J. Kienzle. Exception-aware requirements elicitation with use cases. In *Recent Advances in Exception Handling Techniques*, LNCS 4119, pages 221–242. Springer-Verlag, 2006.

[161] J. Siedersleben. Errors and exceptions - rights and obligations. In A. Romanovsky et al., editors, *Recent Advances in Exception Handling Techniques*, LNCS 4119. Springer-Verlag, 2006.

[162] K. Simons and J. 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. Springer-Verlag, May 2004.

[163] M. Sloman and J. Kramer. *Distributed Systems and Computer Networks.* Prentice-Hall, 1987.

[164] S. Soares, E. Laureano, and P. Borba. Implementing distribution and persistence aspects with AspectJ. In *Proceedings of the 17th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 174–190, 2002.

[165] B. Spitznagel and D. Garlan. Architecture-based performance analysis. In *Proceedings of the 10th International Conference on Software Engineering and Knowledge Engineering*, June 1998.

[166] D. Sprott. Componentizing the enterprise application packages. *Communications of the ACM*, 43(4):63–69, April 2000.

[167] V. Stavridou and A. Riemenschneider. Provably dependable software architectures. In *Proceedings of the Third ACM SIGPLAN International Software Architecture Workshop*, pages 133–136. ACM, 1998.

[168] Sun Microsystems. Enterprise javabeans specification v2.1 - proposed final draft, 2002. http://java.sun.com/products/ejb/.

[169] C. Szyperki. Component technology - what, where, and how? In *Proceedings of the 25th International Conference on Software Engineering*, pages 684–693. IEEE Computer Society Press, May 2003.

[170] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, USA, second edition edition, November 2002.

[171] S. Taft and R. Duff. *Ada 95 Reference Manual: Language and Standard Libraries, International Standard Iso/Iec 8652:1995(e)*, volume 1246 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[172] F. Tartanoglu, N. Levy, V. Issarny, and A. Romanovsky. Using the b method for the formalization of coordinated atomic actions. Technical Report CS-TR: 865, School of Computing Science, University of Newcastle, October 2004.

[173] R. N. Taylor et al. A component- and message- based architectural style for GUI software. In *Proceedings of the 17th International Conference on Software Engineering*, pages 295–304, April 1995.

[174] R. T. Tomita, F. Castor Filho, P. A. de C. Guerra, and C. M. F. Rubira. Bellatrix: An environment with arquitectural support for component-based development (in portuguese). In *Proceedings of the IV Brazilian Workshop on Component-Based Development (WDBC'2004)*, pages 43–48, September 2004.

[175] UCI. ArchStudio 3.0 homepage. http://www.isr.uci.edu/projects/archstudio.

[176] US Department of Transportation. Reusable software components, December 2004. Advisory Circular # AC 20-148.

[177] J. Vachon and N. Guelfi. Coala: a design language for reliable distributed system engineering. In *Proceedings of the Workshop on Software Engineering and Petri Nets*, pages 135–154, Aarhus, Denmark, June 2000.

[178] M. van Dooren and E. Steegmans. Combining the robustness of checked exceptions with the flexibility of unchecked exceptions using anchored exception declarations. In *Proceedings of 20th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 455–471, October 2005.

[179] G. Vecellio, M. Thomas, and R. Sanders. Container services for high confidence software. In *Proceedings of the 7th ECOOP Workshop on Component-Oriented Programming*, June 2002.

[180] J. Viega and J. M. Voas. Can aspect-oriented programming lead to more reliable software. *IEEE Software*, 17(6):19–21, 2000.

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

[182] W. Weimer and G. Necula. Finding and preventing run-time error handling mistakes. In *Proceedings of the 19th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 419–433, October 2004.

[183] Y. Wu, D. Pan, and M.-H. Chen. Techniques for testing component-based software. In *Proceedings of the 7th International Conference on Engineering of Complex Computer Systems*, pages 222–232, Skövde, Sweden, June 2001.

[184] J. Xu, B. Randell, A. B. Romanovsky, C. M. F. Rubira, R. J. Stroud, and Z. Wu. Fault tolerance in concurrent object-oriented software through coordinated error recovery. In *Proceedings of the 25th Symposium on Fault-Tolerant Computing Systems*, pages 499–508, Pasadena, USA, June 1995.

[185] J. Xu, B. Randell, A. B. Romanovsky, R. J. Stroud, A. F. Zorzo, E. Canver, and F. W. von Henke. Rigorous development of an embedded fault-tolerant system based on coordinated atomic actions. *IEEE Transactions on Computers*, 51(2):164–179, February 2002.

[186] K. Yi. An abstract interpretation for estimating uncaught exceptions in standard ml programs. *Science of Computer Programming*, 31(1):147–173, 1998.