Um Método Baseado em Reuso Para o Desenvolvimento de Sistemas Confiáveis Baseados em Componentes

Patrick H. da S. Brito[1]*, **Paulo Asterio de Castro Guerra, Cecília M. F. Rubira**[1]

[1]Institute of Computing – State University of Campinas (Unicamp)
P.O. Box 6176, 13084-971, Campinas, SP, Brazil

{pbrito, cmrubira}@ic.unicamp.br, asterio@acm.org

Abstract. *Nowadays, the software development demands each time more restrictions of time to market and low costs. Moreover, with the increase of the technological dependence, the software quality and dependability became obligatory requirements for developing computing systems. For speeding the development and reducing its cost, the component-based development (CBD) is being each time more adopted. This paper presents a CBD method which aims to maximise the reuse of existing components, improving at the same time the dependability of the final system. For achieving this, the proposed method adopts an architecture-based solution which acts in three complementary ways: (i) a uniform structure of all the critical architectural elements; (ii) separation of concern between the normal and abnormal (exceptional) behaviours; and (iii) a systematic modelling of the exceptional behaviour since the beginning of the software life cycle. The proposed solution was evaluated through a case study of a real financial application.*

Resumo. *O mercado de software está cada vez mais sujeito a fortes restrições de prazos e custos. Além disso, com a intensificação do uso de sistemas computacionais em atividades essenciais, exige-se cada vez mais qualidade e confiança no funcionamento dos sistemas. Por agilizar o desenvolvimento e reduzir o seu custo a longo prazo, o desenvolvimento baseado em componentes (DBC) vem sendo adotado largamente. Porém, se não for feita de forma sistemática, a reutilização pode comprometer a confiabilidade do sistema. Este artigo propõe um método de DBC que visa maximizar a reutilização de componentes prontos durante o desenvolvimento, aumentando ao mesmo tempo a confiabilidade do sistema final. Para isso, é adotada uma abordagem centrada na arquitetura que atua de três formas complementares: (i) uniformidade na estruturação dos elementos arquiteturais considerados críticos; (ii) separação de interesse entre os comportamentos normal e excepcional; e (iii) preocupação sistemática com o comportamento excepcional desde o início do ciclo de vida do software. A abordagem proposta nesse artigo foi avaliada através de um estudo de caso de uma aplicação financeira real.*

1. Introdução

Atualmente, sistemas de software são utilizados em uma variedade de aplicações onde o preço de uma falha pode ser muito alto. Programas que podem causar riscos a vidas humanas ou grandes perdas financeiras devem ser tolerantes a falhas. Isto é, esses sistemas devem ser capazes de oferecer os seus serviços especificados, ainda que parcialmente, mesmo na presença de falhas. Uma outra particularidade do mercado de software atual é o fato dele estar sujeito a fortes restrições de prazos e custos [23], o que estimula a utilização de tecnologias que facilitem a reutilização de artefatos de software, como por exemplo o desenvolvimento baseado em componentes (DBC). A reutilização de software pode impactar negativamente na confiabilidade dos sistemas devido a uma possível ausência de garantia da procedência dos componentes reutilizados [16]. No entanto, o DBC vem sendo utilizado para o desenvolvimento de sistemas computacionais com requisitos estritos de confiabilidade, tais como controladores de transportes em massa e dispositivos automotivos, para desenvolver software de forma mais barata e rápida. Os requisitos de confiabilidade do sistema e seu baixo custo de desenvolvimento podem ser conflitantes, o que demanda a utilização de técnicas e ferramentas que possibilitem o desenvolvimento de sistemas que sejam tanto tolerantes a falhas, quanto rápidos de construir e fáceis de manter.

*Apoiado pela FAPESP, projeto n^o 06/02116-2.

Entre as várias técnicas existentes para construir sistemas tolerantes a falhas, o tratamento de exceções é um mecanismo conhecido para estruturar a recuperação de erros em sistemas de software [5]. O fato do tratamento de exceções ser uma técnica específica de aplicação possibilita a implementação de medidas de recuperação de erros bastante sofisticadas, complementando outras técnicas conhecidas, tais como as transações atômicas [13]. Além do mais, em sistemas onde não é possível restaurar o estado do sistema para um estado anterior, o tratamento de exceções pode ser a única escolha disponível para recuperação de erro. Por outro lado, também é sabido que o uso de tratamento de exceções possui algumas desvantagens, por exemplo, uma parte considerável do código-fonte de sistemas confiáveis é dedicada à detecção e tratamento de erros [5], definido como sendo o comportamento excepcional do sistema. Isso acontece, em grande parte, devido à necessidade de se representar um grande de situações excepcionais em um sistema de software. Além de identificar os diferentes tipos de erros previstos para a aplicação, é necessário também identificar os componentes que podem detectá-los e tratá-los. Um outro fator que prejudica a qualidade do comportamento excepcional é o fato dos desenvolvedores priorizarem a implementação do comportamento normal do sistema e lidar com o código de recuperação de erros de maneira *ad hoc* durante a implementação. Apesar de complexo, o código que implementa o comportamento excepcional é normalmente o menos entendido, testado e documentado [5], o que tende a reduzir a confiabilidade geral do sistema. Uma tendência aceita recentemente é que, para alcançar os níveis desejados de confiabilidade, os mecanismos de detecção e tratamento de erros devem ser incorporados sistematicamente, desde a fase de especificação, passando pelo projeto e por fim na implementação do sistema [10, 21].

Para se desenvolver sistemas com reutilização, é imprescindível considerar a interação entre suas partes, que é representada através da sua arquitetura de software, que é o nível mais alto de abstração da estrutura de um sistema, sem detalhes de código. Dado que a arquitetura representa a estrutura global do sistema, os requisitos de qualidade, tais como confiabilidade e disponibilidade, são determinantes para a sua concepção [4]. O comportamento excepcional é também considerado uma qualidade sistêmica e portanto ele influencia a arquitetura de software do sistema. Além disso, os componentes reutilizados nem sempre obedecem aos mesmos contratos especificados no comportamento excepcional do sistema, isto é, podem ocorrer incompatibilidades entre o comportamento excepcional do componente e o comportamento excepcional do sistema. Nesse caso, a arquitetura deve se preocupar com a consistência da integração de componentes, de tal forma que o comportamento excepcional do sistema integrado seja confiável [15, 8].

Este trabalho propõe um método de DBC que visa maximizar a reutilização de componentes para a construção de sistemas tolerantes a falhas. O método guia o desenvolvedor durante as fases de especificação, projeto e implementação, baseando-se: (i) na definição do comportamento excepcional do sistema; (ii) na identificação de componentes prontos que podem ser reutilizados; e (iii) na adaptação do comportamento excepcional dos componentes reutilizados no contexto da arquitetura. Esse trabalho se baseia no Método para Definição do Comportamento Excepcional (MDCE+) [6], que define exceções e seus tratadores no nível da arquitetura de software. A principal contribuição deste trabalho é conciliar a máxima reutilização de componentes de software com a alta confiabilidade dos sistemas integrados. O método proposto possui atividades que especificam o comportamento excepcional e diretrizes que maximizam a reutilização de componentes prontos, bem como a adaptação desses componentes na arquitetura alvo.

O restante do artigo está organizado como segue. Seção 2 apresenta os conceitos necessários utilizados. Seção 3 apresenta o método proposto através de um diagrama de atividades. Seção 4 descreve um estudo de caso do método proposto utilizando um sistema bancário real. Finalmente, a Seção 5 apresenta as conclusões e direções para trabalhos futuros.

2. Fundamentos de Arquitetura de Software, Tratamento de Exceções e DBC

2.1. Arquiteturas de Software

A arquitetura de software, através de um alto nível de abstração, define o sistema em termos de **componentes arquiteturais**, a interação entre eles e os atributos e funcionalidades de cada um [4]. Além disso, a arquitetura representa essa interação de forma explícita, materializando-a através dos **conectores arquiteturais**. Por conhecerem o fluxo interativo entre os componentes do sistema, essas entidades são capazes, entre outras coisas, de estabelecer protocolos de comunicação e de coordenar a execução dos serviços que envolvam mais de um componente do sistema. A forma como os **elementos arquiteturais** (componentes e conectores) ficam dispostos logicamente é conhecida como **configuração arquitetural**.

A presença de uma determinada propriedade arquitetural pode ser obtida através da utilização de estilos arquiteturais [22]. Um **estilo arquitetural** caracteriza uma família de sistemas que são relacionadas pelo compartilhamento de suas propriedades estruturais e semânticas, influenciando inclusive na configuração arquitetural. Por influenciar os artefatos produzidos em todas as fases do desenvolvimento, a arquitetura de software é um artefato essencial nos processos modernos de desenvolvimento de software. Essa importância fica ainda mais evidente no contexto do desenvolvimento baseado em componentes, que possui uma etapa específica para integrar os componentes do sistema, de modo que eles interajam entre si para oferecer as funcionalidades especificadas.

2.2. Tratamento de Exceções

2.2.1. Componente de Software Tolerante a Falhas Ideal

O componente tolerante a falhas ideal[1] [17] (IFTC) é um conceito estrutural que utiliza o mecanismo de tratamento de exceções para a construção de sistemas tolerantes a falhas. Um IFTC promove uma separação de interesse explícita entre o comportamento normal de um sistema, e o seu comportamento excepcional, onde as medidas de tolerância a falhas são implementadas.

Componentes de software recebem requisições de serviços e produzem respostas. De acordo com a terminologia proposta por Lee e Anderson [17], essas respostas podem ser classificadas em duas categorias distintas: normal, que corresponde às situações onde o componente deve oferecer os seus serviços de forma satisfatória; e excepcional, normalmente sinalizada quando um erro é detectado e o componente não é capaz de oferecer o serviço requisitado. Respostas excepcionais são normalmente conhecidas como exceções [12].

As exceções podem ser classificadas em duas categorias distintas: internas e externas. Exceções internas são lançadas pelo componente com o intuito de acionar os seus próprios tratadores. Caso a exceção seja tratada com sucesso, o componente pode retornar ao estado normal e continuar a execução do serviço. Já as exceções externas são sinalizadas quando o componente determina que, por alguma razão, ele não é capaz de oferecer os serviços especificados. Exceções externas podem ser particionadas em exceções de interface, decorrentes de requisições inválidas de serviços, e exceções de defeito[2], que são provocadas por defeitos no processamento de requisições válidas de serviços. Neste sentido, exceções e tratamento de exceções oferecem um arcabouço para estruturar as atividades de tolerância a falhas de sistemas de software.

Vários IFTCs podem ser organizados em camadas, de modo que os componentes podem tratar as exceções propagadas por componentes localizados em outras camadas. Nessa abordagem, a arquitetura de software deve ser particionada em camadas que possuem diferentes níveis de abstração. Idealmente, cada camada deve ser responsável por tratar apenas as exceções lançadas pela camada imediatamente inferior a ela.

[1] do inglês *idealised fault-tolerant component*

[2] do inglês *failure exceptions*

2.2.2. iFTE: Elemento Arquitetural Tolerante a Falhas Ideal

O elemento arquitetural tolerante a falhas ideal (iFTE) [9] é uma abstração arquitetural para estruturar sistemas tolerantes a falhas que materializa os princípios associados ao conceito do componente tolerante a falhas ideal, apresentado na Seção 2.2.1. Para isso, ele inclui de uma maneira estruturada, responsabilidades para detecção, tratamento e propagação de erros nos elementos arquiteturais.

O modelo geral de um iFTE define quatro tipos de interfaces externas, que são particionadas claramente entre comportamento normal e excepcional: (i) I_iFTE_PS define um ponto de acesso para os serviços (tolerantes a falhas) oferecidos pelo iFTE; (ii) I_iFTE_PE define um ponto de acesso onde o iFTE sinaliza as suas exceções externas; (iii) I_iFTE_RS especifica os serviços requeridos para que o iFTE implemente o seu comportamento normal e os seus tratadores de exceções; e (iv) I_iFTE_RE especifica as exceções externas que o iFTE está preparado para tratar. Essas interfaces podem ser especializadas de acordo com os serviços e exceções de um elemento arquitetural específico.

Estrutura do iFTE. O projeto detalhado de um iFTE é apresentado na Figura 1: (i) o componente Normal implementa o comportamento normal do iFTE e pode ser associado a um componente já existente; (ii) o componente Abnormal trata as exceções lançadas pelo componente Normal, ou propagadas pelo ambiente externo, através do componente Required; (iii) o componente Provided atua como uma ponte entre os serviços oferecidos pelo iFTE e o ambiente externo, incluindo a sinalização de exceções; (iv) o componente Required atua como uma ponte entre os serviços requeridos do iFTE e o ambiente externo, que oferece esses serviços; e (v) o conector Coordinator coordena a interação entre os quatro componentes internos do iFTE.

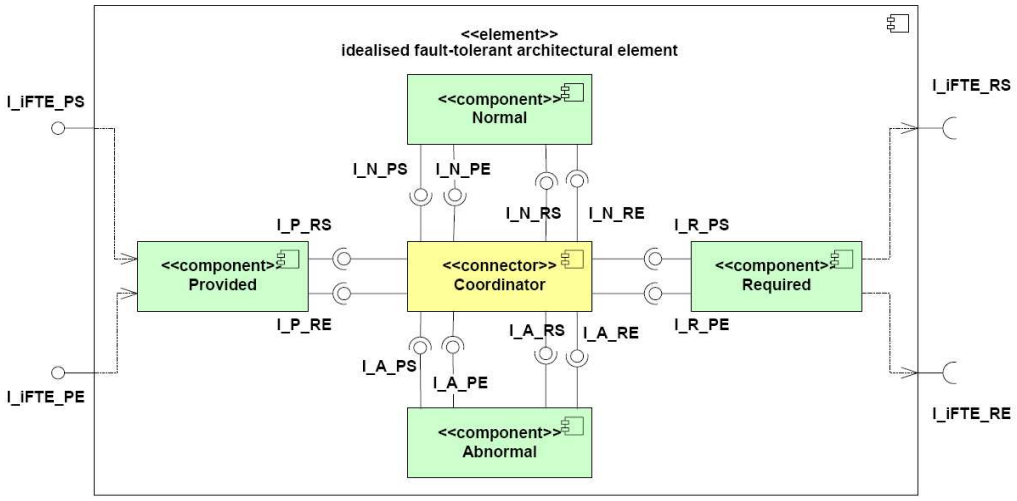

Figura 1. Estrutura interna do elemento arquitetural tolerante a falhas ideal (iFTE).

Os elementos arquiteturais internos do iFTE interagem através de interfaces internas, que também garantem a separação de interesse entre os comportamentos normal e excepcional.

Cenários de Execução. Do ponto de vista da abstração do iFTE em si, existem sete cenários distintos que descrevem os relacionamentos que podem ser estabelecidos entre as interfaces externas de um iFTE. Após receber uma requisição externa de serviço, através de I_iFTE_PS, o iFTE pode responder de três formas distintas: retorna normalmente através de I_iFTE_PS (1^o cenário); sinaliza uma exceção de interface através de I_iFTE_PE (2^o cenário); ou sinaliza uma exceção de defeito através de I_iFTE_PE (3^o cenário).

Além disso, o iFTE pode precisar de serviços externos a ele, que estão disponíveis através da interface I_iFTE_RS. Após requisitar um serviço externo, podem ocorrer quatro cenários distintos. Se o elemento arquitetural externo retornar normalmente através de I_iFTE_RS, o iFTE pode oferecer um retorno normal ao seu cliente, através de I_iFTE_PS (4^o cenário), ou pode sinalizar uma exceção de defeito através de I_iFTE_PE (5^o cenário). Se o elemento arquitetural externo sinalizar uma exceção, o iFTE pode tanto propagar o erro através de I_iFTE_PE (6^o cenário), quanto recuperar o seu estado e oferecer um retorno normal através de I_iFTE_PS (7^o cenário).

2.3. COSMOS: Um Modelo de Implementação de Componentes

Para concretizar os elementos arquiteturais em código-fonte, é necessário adotar algum modelo de implementação de componentes. Através de um conjuno de padrões de projeto, o modelo COSMOS [7] define três sub-modelos, que lidam com diferentes aspectos do desenvolvimento baseado em componentes: (i) o *modelo de especificação* especifica as interfaces providas e requeridas de cada componente, inclusive as excepcionais; (ii) o *modelo de implementação* define um conjunto de padrões de projeto que direcionam a implementação dos serviços oferecidos pelo componente, reduzindo o acoplamento com os respectivos serviços requeridos; e (iii) o *modelo de conectores* especifica a ligação entre as interfaces requeridas de um componente e as respectivas interfaces providas de outro. De um modo geral a única diferença entre a estrutura de um conector e um componente COSMOS, é a ausência de interfaces providas e requeridas, uma vez que as interfaces envolvidas pertencem a outros elementos da arquitetura. Por restrições de espaço, a estrutura do COSMOS só é apresentada na Seção 4.1, no contexto de uma aplicação financeira real.

3. O Método Proposto

A Figura 2 apresenta a visão geral do método proposto. Ele é composto de nove atividades que guiam o desenvolvimento de sistemas confiáveis, a partir da especificação dos requisitos, passando pelo projeto arquitetural, pela obtenção de componentes, até a implementação final do sistema baseado em componentes.

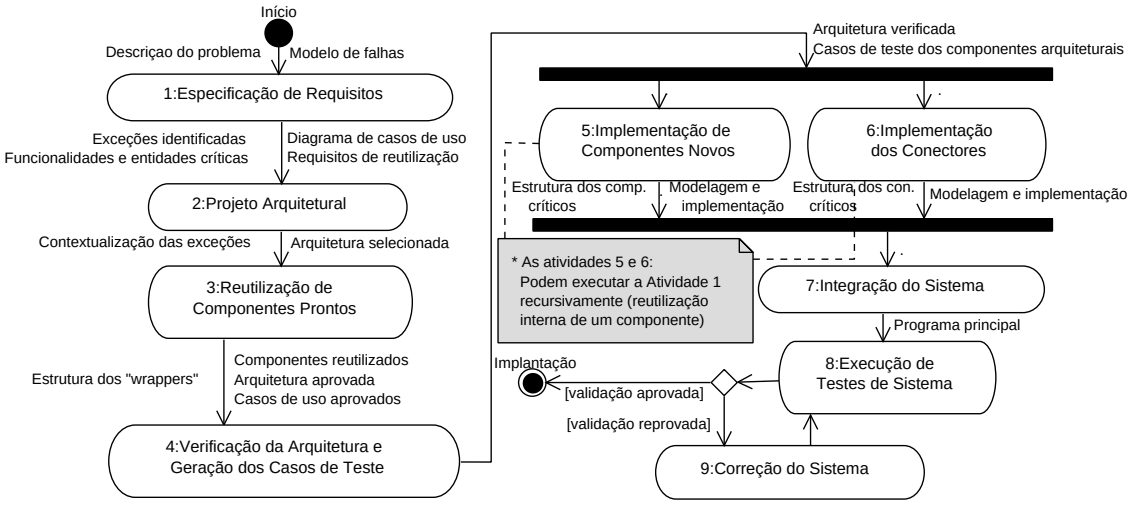

Figura 2. Diagrama de atividades em UML do método proposto

O desenvolvimento do comportamento normal do sistema é acompanhado pela especificação do comportamento excepcional, que inclui a identificação dos tipos de exceções e a especificação dos seus tratadores em cada elemento arquitetural. A especificação do comportamento excepcional está distribuída nas fases de desenvolvimento, com o objetivo de maximizar a reutilização em cada uma delas.

3.1. Ativ.1 Especificação de Requisitos

O principal objetivo desta fase é identificar e especificar os requisitos funcionais esperados para o sistema, assim como os requisitos de qualidade (requisitos não-funcionais).

Nessa fase, protótipos funcionais podem ser construídos com o objetivo de melhorar a compreensão dos requisitos e ao mesmo tempo auxiliar o entendimento da sua arquitetura [1]. É recomendado que esses protótipos sejam construídos a partir de componentes prontos.

Do ponto de vista da maximização da reutilização, as principais atividades executadas aqui são: análise e representação do domínio do problema, através de um modelo conceitual; e definição das restrições de reutilização, tais como restrições dos estilos arquiteturais. Essas restrições podem ser uma conseqüência da adoção de um *framework* específico ou da obrigatoriedade de se adotar alguma plataforma de desenvolvimento específica. Outros exemplos de restrições comuns são [23]: (i) existência de certificações específicas; (ii) custo do produto e forma de pagamento; (iii) prazo de entrega; e (iv) oferecimento de garantia/manutenção do componente.

O comportamento excepcional, relativo à identificação e tratamento de erros, é representado através de cenários excepcionais, que estendem a especificação dos casos de uso. Além disso, são definidas pré-condições, pós-condições e invariantes aos casos de uso tradicionais, seguindo a abordagem de projeto por contrato[3] [19]. A definição de contratos é uma atividade importante, uma vez que as exceções podem ser antecipadas através da análise das suas possíveis violações [21, 6].

Além das atividades para identificar e documentar exceções, a partir das funcionalidades especificadas e do domínio da aplicação, o método apresenta mais outras três atividades complementares: (i) identificação de exceções relativas ao domínio; (ii) definição de entidades críticas do domínio (modelo conceitual); e (iii) definição de funcionalidades críticas, cuja corretude ou disponibilidade é justificadamente desejada. Os especialistas do domínio podem identificar exceções recorrentes baseado em experiências anteriores. Além disso, algumas entidades do modelo conceitual podem ser consideradas críticas, tendo em vista a sua importância para o domínio.

Os principais artefatos finais dessa fase são: (i) diagramas de casos de uso com cenários normal, alternativo e excepcionais; (ii) exceções identificadas a partir da violação das pré-, pós-condições e invariantes dos casos de uso; (iii) exceções do domínio; (iv) funcionalidades e entidades conceituais críticas; e (v) requisitos de reutilização.

3.2. Ativ.2 Projeto Arquitetural

A fase de projeto arquitetural é composta de quatro etapas. Primeiramente, a partir de um catálogo de estilos arquiteturais, o arquiteto deve listar as arquiteturas compatíveis com as restrições definidas na fase de requisitos (Seção 3.1). Em especial, deve-se dar uma atenção maior aos aspectos tecnológicos, tais como *frameworks* e *middlewares*. Após essa triagem inicial, o arquiteto deve listar as principais arquiteturas candidatas para o domínio da aplicação. Com a lista dos principais candidatos disponíveis, o arquiteto deve especificar cenários que demonstrem como os requisitos de qualidade críticos são realizados por cada arquitetura candidata. Finalmente, em uma reunião de *brainstorming* envolvendo o líder do time de desenvolvimento, o gerente do projeto e o representante do cliente, o arquiteto apresenta os seus argumentos para cada um dos estilos pré-selecionados. Após escutar as argumentações de cada parte, o arquiteto escolhe a arquitetura mais apropriada, podendo ajustá-la de acordo com outras necessidades especiais discutidas na reunião.

Com os estilos arquiteturais selecionados, o próximo passo é identificar os principais componentes da arquitetura. O primeiro passo para isso é especificar os módulos funcionais do sistema, decorrentes tanto das funcionalidades esperadas para o sistema (casos de uso), quanto das funcionalidades inerentes ao domínio do negócio (modelo conceitual). Em seguida, os módulos identificados devem ser posicionados na arquitetura

[3] do inglês *design by contract*

especificada. Do ponto de vista de confiabilidade, é necessário identificar os elementos arquiteturais críticos, que são considerados os principais pontos de falha do sistema. Essa tarefa pode se basear nas funcionalidades e entidades críticas, definidas anteriormente. Tendo em vista a importância desses elementos, eles podem ser foco de redundância de projeto[4] [17].

Após a definição da arquitetura, é necessário classificar as exceções identificadas na fase de especificação de requisitos. Para isso, essas exceções, que são consideradas exceções do sistema, devem ser categorizadas de acordo com a hierarquia apresentada na Figura 3 [8]. Essa hierarquia define tipos de exceções que uniformizam os modelos de falhas e podem ser mapeados para os tipos de exceções do componente ideal, apresentado na Seção 2.2.1. Exceções de interface herdam de `RejectedRequestException`. Exceções de defeito (`FailureException`s) podem herdar de `DeclaredException`, caso se refiram a erros previstos na especificação do sistema; ou de `UndeclaredException`, caso se refiram a erros inesperados. `FailureException` possui dois subtipos que indicam variações no estado do componente após o lançamento da exceção. Uma exceção de defeito deve herdar de `RecoverableFailureException`, quando se sabe que o seu estado ainda pode ser recuperado, por exemplo, através de *rollback*. Conseqüentemente, uma exceção de defeito herda de `UnrecoverableFailureException` quando não se pode afirmar nada sobre a consistência do componente após o lançamento da exceção.

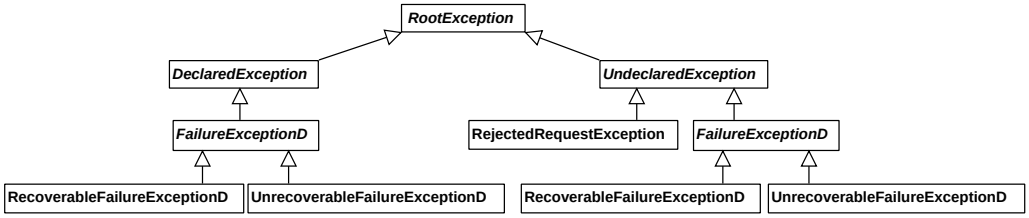

Figura 3. Hierarquia uniforme de tipos de exceções

Os principais artefatos finais dessa fase são: (i) arquitetura selecionada para o sistema; e (ii) classificação das exceções através da hierarquia proposta.

3.3. Ativ.3 Reutilização de Componentes Prontos

O método proposto considera três formas de obtenção de componentes: (i) reutilização interna (mesma organização); (ii) aquisição de terceiros; e (iii) implementação de um componente novo. Para cada elemento da arquitetura, são identificados os candidatos à reutilização, baseado nos requisitos e na similaridade existente entre componentes de um mesmo domínio, priorizando a reutilização interna à organização. Após essa triagem inicial, é iniciado um processo de negociação e ajuste dos requisitos, onde o arquiteto expõe as vantagens de se reutilizar o componente identificado (principalmente custo e prazo), e o impacto nos requisitos especificados. Se o cliente aceitar o impacto, o componente é reutilizado e os requisitos e a arquitetura são ajustados, se necessário.

Com a lista dos componentes candidatos a serem reutilizados, deve-se analisar cada caso e decidir qual o modo como o componente será obtido. Essa decisão consiste basicamente numa análise da relação custo x benefício entre as três formas apresentadas: reutilização interna, aquisição ou implementação. Além do esforço necessário para codificação, a análise de custos deve considerar limitações de tempo e recursos [23]. Na reutilização de componentes prontos, os componentes reutilizados nem sempre obedecem aos mesmos contratos especificados nos requisitos. Por esse motivo, nesses casos pode ser necessário adaptar o componente reutilizado a fim de satisfazer os contratos estabelecidos [15, 8]. O custo de implementação dessas adaptações também deve ser contabilizado como parte do custo total do componente. De maneira análoga, quando um componente é reutilizado exaustivamente, o seu custo de aquisição (compra) deve ser amortizado.

[4] do inglês *design diversity*

A Figura 4 ilustra a estrutura de um *wrapper* que envolve dois componentes. As requisições de serviços devem chegar ao *wrapper* seguindo o contrato especificado na fase de projeto arquitetural (Seção 3.2). A classe `InputAdapter` é responsável por converter os tipos dos parâmetros e repassa a solicitação ao componente propriamente dito. Ao receber a resposta, é realizada a conversão do tipo de retorno, inclusive os retornos excepcionais. A classe `OutputAdapter` realiza um papel semelhante, porém relativo às requisições de serviços externos realizadas pelos componentes reutilizados. Além desse papel conciliador entre as especificações da aplicação e do componente reutilizado, o *wrapper*, também é responsável por detectar novas exceções, desempenhando uma função semelhante aos componentes sentinelas[5], propostos por Dellarocas [11].

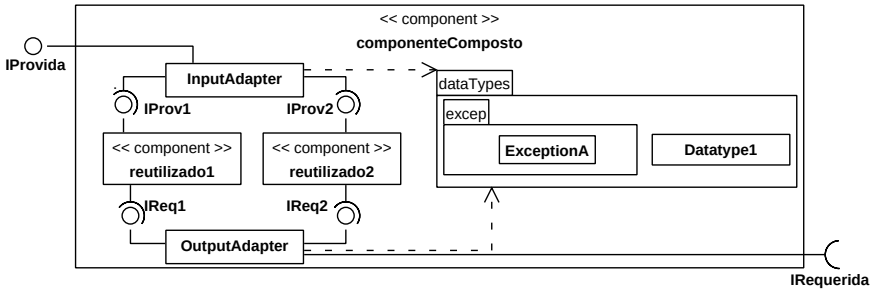

Figura 4. Ilustração de um *Wrapper* envolvendo dois componentes prontos

No caso dos componentes considerados críticos, eles devem ser estruturados de acordo com o elemento arquitetural tolerante a falhas ideal (iFTE), apresentado na Seção 2.2.2. Nesses casos, o componente reutilizado deve ser adaptado para desempenhar o papel do elemento Normal, interno ao iFTE.

Os principais artefatos finais dessa fase são: (i) componentes reutilizados; (ii) estrutura dos *wrappers* dos componentes reutilizados; (iii) arquitetura aprovada; e (iv) casos de uso aprovados.

3.4. Ativ.4 Verificação da Arquitetura e Geração dos Casos de Teste

Após a conclusão do projeto arquitetural, a arquitetura de software deve ser especificada formalmente. Com a especificação formal da arquitetura, é possível verificar propriedades que ajudam a identificar e remover falhas de forma antecipada, ainda nas fases de modelagem do sistema. Na abordagem proposta, o processo de verificação é feito em duas etapas seqüenciais [2]: (i) geração dos cenários de execução; e (ii) verificação do modelo para cada um dos cenários. Ambos os passos devem ser executados com o auxílio de ferramentas de apoio, que no caso particular da verificação formal (Etapa (ii)), é uma ferramenta de verificação de modelos.

As propriedades a serem verificadas na arquitetura são classificadas em duas categorias: (i) propriedades de configuração arquitetural, que verificam aspectos estruturais da arquitetura de software; e (ii) propriedades de propagação de exceções, que verificam a consistência dos cenários, a existência de exceções sem tratamento e de tratadores que nunca são ativados. Mais detalhes sobre a especificação formal da arquitetura de software, a verificação de propriedades e a geração de casos de teste estão disponíveis em outras publicações [2].

Os principais artefatos finais dessa fase são: (i) arquitetura com propriedades satisfeitas; e (ii) casos de teste dos componentes arquiteturais.

3.5. Ativ.5 Implementação de Componentes Novos

Quando não é possível reutilizar um determinado componente, o desenvolvedor pode proceder de duas formas: (i) tentar reutilizar componentes menores para construir um maior, utilizando o método de forma recursiva; ou (ii) implementar um novo componente

[5]do inglês *sentinel components*

do zero, nos casos onde a reutilização seja inviável, isto é, impossível ou muito cara. No segundo caso, o método proposto sugere a adoção do modelo de implementação de componentes COSMOS, apresentado na Seção 2.3. Para complementar a especificação interna do componente, deve-se utilizar algum processo de desenvolvimento orientado a objetos, e.g. o processo unificado [14].

Em relação às exceções especificadas para os componentes novos e conectores, elas devem ser organizadas de acordo com a hierarquia apresentada na Seção 3.2. Além disso, os elementos arquiteturais considerados críticos (Seção 3.2) devem ser estruturados de acordo com o iFTE, apresentado na Seção 2.2.2. A uniformidade de estruturação dos componentes críticos facilita a implementação e a manutenção dos componentes, o que reduz o número de falhas inseridas e aumenta o tempo de vida do sistema [17, 23].

Os principais artefatos finais dessa fase são: (i) arquitetura interna dos componentes novos; (ii) modelagem e implementação dos componentes novos.

3.6. Ativ.6 Implementação dos Conectores e Ativ.7 Integração do Sistema

Nessa fase, todos os componentes do sistema já estão disponíveis, mas para que possam ser integrados é necessário implementar as conexões entre eles. Essas conexões são realizadas através da ligação das interfaces requeridas de um componente às respectivas interfaces providas de outros. Porém, dependendo da similaridade ou não entre as interfaces envolvidas, pode ser necessário implementar algum procedimento de adaptação. Os tipos de adaptações mais comuns são três [23]: (i) conversão de tipos dos parâmetros e valores de retorno; (ii) diferença de assinatura das operações; e (iii) oferecimento de serviços incompletos, que devem ser complementados.

Por representar o elo de comunicação entre componentes arquiteturais, os conectores são os elementos da arquitetura que deveriam implementar os fluxos interativos entre os componentes. Dessa forma, eles são candidatos naturais para implementar os requisitos de qualidade do sistema, tais como confiabilidade e disponibilidade.

Após finalizar a especificação dos conectores do sistema, deve-se prosseguir com a sua implementação, que de acordo com método proposto, é aconselhável seguir o modelo COSMOS. Além disso, é necessário implementar as rotinas de "ligação" entre componentes e conectores. Finalmente, da mesma forma que acontece com os componentes novos, os conectores considerados críticos durante o projeto arquitetural (Seção 3.2) devem ser estruturados de acordo com o elemento arquitetural tolerante a falhas ideal (iFTE).

Os principais artefatos finais dessa fase são: (i) arquitetura interna dos conectores; (ii) modelagem e implementação dos conectores; e o (iii) programa principal que configura os elementos arquiteturais.

3.7. Ativ.8 Execução de Testes de Sistema e Ativ.9 Correção do Sistema

As atividades de testes visam reduzir o número de falhas existentes no sistema, principalmente as falhas adicionadas na fase de implementação, que provocam inconsistências entre o código-fonte e a especificação do sistema [20]. A execução dos testes consiste na aplicação dos casos de teste definidos a partir da especificação do sistema (Seção 3.4). Em sistemas reais, devido ao grande número de serviços que podem ser oferecidos, essa atividade deve ser executada preferencialmente de forma automática, com o auxílio de ferramentas CASE.

À medida que os testes são executados, as divergências entre os valores esperado e real são apontadas como erros e documentadas através de um relatório. Com o relatório dos testes em mãos, a equipe de desenvolvimento inicia um ciclo de manutenções corretivas, para finalmente implantar o sistema no ambiente real do cliente.

O principal artefato produzido nesta fase é o sistema aprovado nos casos de teste e pronto para implantação.

4. Estudo de Caso: Um Sistema Bancário Real

Para avaliar o método proposto, foi utilizado um estudo de caso de um sistema financeiro com requisitos críticos de disponibilidade [6]. O sistema pertence ao domínio de aplicações bancárias e foi desenvolvido em uma empresa de médio porte, especializada em automação bancária e situada em São Paulo. O estudo de caso se concentrou no subsistema responsável por registrar e controlar a distribuição e compensação de cheques, contratos bancários e limites de crédito.

4.1. Execução

Especificação de requisitos. A primeira atividade da fase de especificação de requisitos foi o desenvolvimento do modelo conceitual, que foi praticamente reutilizado de aplicações anteriores. Foram identificadas 22 entidades e quatro delas foram consideradas críticas: ContaBancária, ControleDeAgências, ParceirosDoBanco, e TransaçõesFinanceiras. Em relação às funcionalidades, o subsistema documentado neste artigo é composto de sete casos de uso: (i) requisitar talão de cheques; (ii) entregar talão de cheques; (iii) cancelar cheques; (iv) depositar um cheque; (v) cancelar contrato da conta; (vi) alterar limites de crédito; e (vii) alterar as regras econômicas. As operações de cancelamento de contrato, cancelamento de cheques e alterações nas regras econômicas possuem requisito crítico de disponibilidade e devem estar disponíveis 24/7. Neste artigo, dadas as restrições de espaço, será enfatizado o caso de uso Cancelar Contrato da Conta. Uma versão bastante detalhada do estudo de caso está disponível em outras publicações [6].

O caso de uso Cancelar Contrato da Conta foi especificado com um cenário principal, além de cenários alternativos e excepcionais. Para cada cenário, foram especificadas pré- e pós-condições. Por exemplo, uma das pré-condições do cenário principal é: "A agência e conta informadas precisam ser válidas e com contrato ativo". Baseado na violação dessa assertiva, foram derivadas as exceções `AgencyOrAccountIsInvalidException` e `AccountContractIsInactiveException`.

Projeto arquitetural. Entre as restrições consideradas para o projeto arquitetural, a principal delas foi o fato da empresa adotar uma arquitetura com duas camadas transversais, utilizadas em todos os projetos: camada Utilitários, que contém pequenas funcionalidades, tais como conversão de valores monetários; e a camada Conexão, responsável pela comunicação com outros sistemas existentes. Além das camadas impostas pela empresa, analisando-se o catálogo de arquiteturas disponíveis, foram identificadas outras duas camadas: camada de Sistema, responsável por implementar as funcionalidades especificadas nos casos de uso; e camada de Negócio, que contém as funcionalidades básicas, derivadas do modelo conceitual produzido nos requisitos. Devido ao requisito crítico de flexibilidade para evolução de regras econômicas, a arquitetura ganhou mais uma camada: RegrasNegócio, localizada entre as camadas de Sistema e de Negócio. Essa camada é exclusiva dos componentes controladores, que consultam periodicamente um arquivo descritor de regras de operação. Por restrições de espaço, a arquitetura final do sistema não é apresentada graficamente, mas de uma maneira geral ela segue uma postura relaxada do estilo arquitetural em camadas [22], que contém cinco camadas tradicionais: Interface, Sistema, RegrasNegócio, Negócio e Dados; além de duas camadas transversais: Utilitários e Conexão.

Foram identificados ao todo 33 elementos arquiteturais, sendo 19 componentes arquiteturais e 14 conectores arquiteturais. No caso específico do caso de uso Cancelar Contrato da Conta, foi identificado o componente Account_Ops, que entre outros, oferece o serviço cancelAccountContract(...), correspondente ao caso de uso. Para a camada de RegrasNegócio, foi identificado um componente EconomicRules_Ctr, responsável por manter o sistema atualizado, em caso de mudanças econômicas. Em relação aos componentes de Negócio, a entidade ContaBancária deu origem ao componente

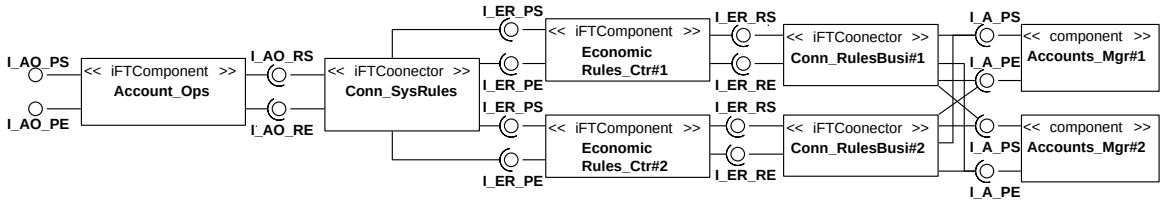

Figura 5. Arquitetura de Software parcial para o sistema financeiro

Account_Mgr. Esse componente implementa serviços relativos às entidades do modelo conceitual, tais como: getAccount(...) e verifyAccountState(...).

A Figura 5 apresenta parte dos elementos arquiteturais identificados para a aplicação. Accounts_Ops controla a execução dos casos de uso, repassando as requisições para uma instância de EconomicRules_Ctr. As duas instâncias de EconomicRules_Ctr são responsáveis por receber e processar as requisições de execução dos serviços (inclusive as mudanças de regras). O conector Conn_SysRules desempenha o papel de distribuidor de carga para melhorar o desempenho. Finalmente, duas instâncias de Accounts_Mgr oferecem os serviços de gerenciamento das contas bancárias e para isso, acessa o banco de dados da instituição (não representado na figura). Os conectores Conn_RulesBusi desempenham o papel de roteadores, com o intuito de aumentar a disponibilidade dos serviços.

O sistema pode falhar de várias maneiras e por várias razões. Por exemplo, o componente Account_Ops sozinho pode sinalizar sete tipos diferentes de exceções. Ao todo, analisando os 33 elementos arquiteturais, foram identificados 19 tipos de exceções que fluem na arquitetura. Apenas para exemplificar, os três tipos de exceções seguintes são sinalizados pelo componente Account_Ops: (i) `InvalidInputDataException`, uma exceção de interface que encapsula todas as exceções relacionadas ao oferecimento de dados inválidos; (ii) `CommunicationFailureException`, que representa um defeito no acesso a componentes distribuídos; e (iii) `InsufficientFundsException`, uma exceção específica do negócio, que é lançada quando o sistema tenta executar uma transação que necessita de fundos não disponíveis.

Em relação à classificação de acordo com a hierarquia apresentada na Seção 3.2, por ser uma exceção de interface, `InvalidInputDataException` foi classificada como sendo do tipo `RejectedRequestException`. A exceção `CommunicationFailureException` encapsula todas as exceções específicas de implementação relativas a falhas de comunicação, que não podem ser previstas em tempo de especificação (`UndeclaredException`). Além disso, por ser recuperável através de redundância, ela foi classificada como uma `RecoverableFailureExceptionU`. Já a exceção `InsufficientFundsException`, por ser uma exceção inerente às regras do negócio, foi prevista durante a especificação do sistema (`DeclaredException`). Mas o seu lançamento indica que o serviço não pode retornar normalmente, independente do tratamento que venha a ser executado; sendo assim, é considerada uma `UnrecoverableFailureExceptionD`.

Reutilização de componentes prontos. Primeiramente foram identificados os componentes das camadas Utilitários e Conexão, que já são reutilizados normalmente na empresa. Além desses componentes, percebeu-se a similaridade de domínio em relação a outros sistemas desenvolvidos previamente. Dessa forma, foi analisada a compatibilidade entre os componentes de sistemas distintos, que estão relacionados às mesmas entidades do modelo conceitual. Para esse estudo de caso, dos dez componentes de Negócio, oito possuíam candidatos à reutilização. Desses oito, sete foram reutilizados, entre eles o componente Accounts_Mgr.

Apesar da reutilização baseada em domínio ter se mostrado eficiente, isso se deve principalmente a duas peculiaridades do nosso cenário: (i) atuação em um domínio es-

pecífico e bem definido; e (ii) associação entre os componentes básicos e as entidades conceituais do domínio.

Verificação do sistema e Geração de casos de teste. Após a especificação formal da arquitetura do sistema, feita em teoria de conjuntos e álgebra de processos [2], a arquitetura foi verificada, principalmente do ponto de vista do seu comportamento excepcional: fluxo de exceções e tratamento de erros.

Após a instanciação do modelo formal da aplicação, foram identificados aproximadamente 980 cenários de execução possíveis, que variam de acordo com os serviços requisitados e os tipos de exceções propagadas. Cada um desses cenários tiveram as suas propriedades verificadas utilizando a ferramenta de checagem de modelos ProB [18]. Durante a verificação da arquitetura, foram detectados *deadlocks* e ocorreu a violação de algumas propriedades. Um exemplo foi o *deadlock* causado pela não-declaração de que o componente Account_Ops pode propagar a exceção `AccountContractIsInactiveException`, que é uma exceção de defeito não recuperável. Em relação à violação de propriedades de propagação de exceções, uma das violações identificadas foi causada pela ausência de dois tratadores no componente UserInterface.

Em um computador Pentium 4 com 512 MB de memória RAM, o processo de verificação demorou em média 22 segundos para cada cenário, totalizando aproximadamente seis horas. Após corrigir as falhas identificadas na verificação, todas as propriedades foram satisfeitas.

Em relação à geração de casos de teste, foram desenvolvidos diagramas de atividades para representar a seqüência de execução das operações do sistema. A partir dos possíveis caminhos desses diagramas, foram identificados os casos de teste do sistema. A Figura 6 apresenta parte da seqüência de execução da operação cancelAccountContract(...) do componente Account_Ops, como pode ser observado, a figura apresenta seis caminhos possíveis, gerando assim seis casos de teste [6].

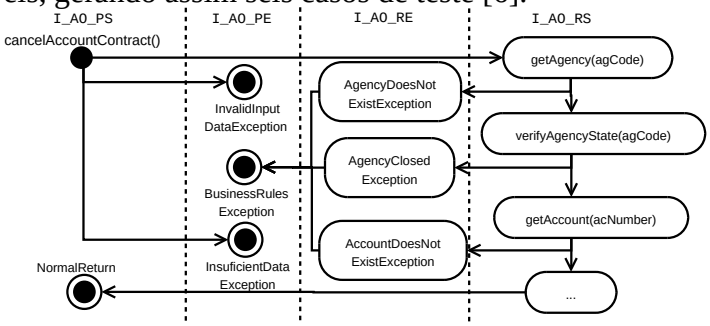

Figura 6. Seqüência de execução da operação cancelAccountContract(...)

Implementação de componentes novos e conectores. Após a especificação dos componentes e a verificação da arquitetura, os componentes que não foram reutilizados foram implementados em Java, de acordo com o modelo COSMOS. Por se tratar de um componente crítico, a Figura 7 ilustra a estrutura do elemento Normal do componente Account_Ops, e não o componente em si, que deve ser estruturado como um iFTE (Seção 2.2.2).

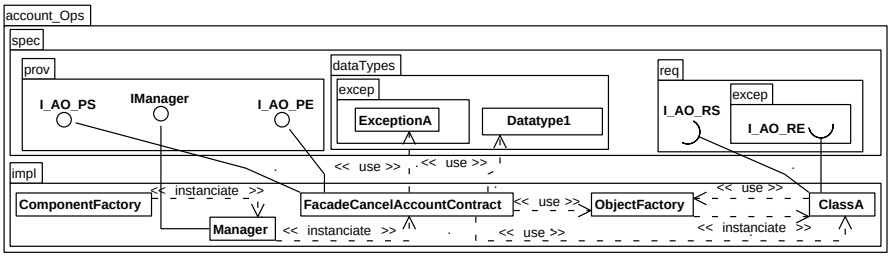

Figura 7. Elemento Normal **do componente** Account_Ops

4.2. Avaliação do Estudo de Caso

Apesar dessa ter sido a única experiência com o método proposto, foi possível realizar uma análise preliminar das suas características, pontos fortes e fracos. A análise foi realizada pelos autores e participantes do estudo de caso, avaliando os benefícios na qualidade final do produto e alguns outros critérios de avaliação de processos, propostos por Sommerville [23].

Em relação à qualidade final do produto, os desenvolvedores da empresa constataram um aumento de aproximadamente 20% na proporção de exceções por funcionalidade, quando comparado com sistemas similares desenvolvidos na própria empresa. De acordo com a análise realizada, esse acréscimo no número de exceções é uma conseqüência do maior refinamento dos tipos de erros, que aliado a uma maior informação contextual do erro na arquitetura, facilita a condução das manutenções corretivas. Antes, as exceções representavam erros genéricos, sem atributos internos que os especificassem melhor. Além dos benefícios relativos à qualidade e número de exceções identificadas antecipadamente, como o estudo de caso apresentou tanto casos de uso críticos quanto não críticos, foi possível avaliar como o método se adequa ao desenvolvimento de softwares com características de confiabilidade diferentes. Com os casos de uso não críticos, o número de exceções pôde ser reduzido através da utilização de tipos genéricos. Nesses casos, os supertipos da hierarquia de exceções foram utilizados para substituir vários tipos específicos. Mas mesmo nesses casos, a informação da natureza do erro foi preservada através de atributos.

Apesar dos benefícios no produto final e no processo de desenvolvimento, o método foi considerado burocrático, principalmente no que se refere à documentação dos requisitos (pré-condições, pós-condições e invariantes) e à especificação formal do sistema. Para contornar esse problema, dependendo da criticidade da aplicação a ser desenvolvida, os casos de uso podem ter uma especificação relaxada e a fase de especificação e verificação formal pode ser opcional.

5. Conclusões e Trabalhos Futuros

Este artigo apresentou um método de desenvolvimento de sistemas confiáveis baseado em componentes que visa maximizar a reutilização de componentes prontos. A principal contribuição do método proposto é a condução conjunta da modelagem dos comportamentos normal e excepcional. Dessa forma, os desenvolvedores de software são direcionadas a lidar com a detecção de exceções e a melhor forma de tratá-las desde o início do desenvolvimento. Uma característica do método proposto é a sua preocupação com a maximização da reutilização de componentes prontos, minimizando o comprometimento dos requisitos especificados e a possível redução da confiabilidade do sistema, proporcionando confiabilidade em um nível mais abstrato: na arquitetura de software. Além disso, do ponto de vista das fases de desenvolvimento, o método possui atividades comuns à maioria dos processos de DBC atuais [3, 24, 23], o que facilita a sua compreensão e utilização prática.

Além da modelagem dos comportamentos normal e excepcional e da especificação dos componentes, o método também cobre as atividades de implementação, que nos casos dos componentes reutilizados consiste na implementação de *wrapers* e tratadores de exceções. Além disso, o método sugere que os componentes considerados críticos sejam estruturados de acordo como um elemento tolerante a falhas ideal, que oferece uma separação de interesse explícita entre os comportamentos normal e excepcional, tanto internamente ao componente, quanto do ponto de vista da arquitetura do sistema.

Como trabalho futuro, é proposto um estudo de um ferramental para apoiar o método proposto de tal forma a automatizar parcialmente ou complementamente as atividades do método. Como por exemplo, a geração automática do modelo formal da arquitetura a partir de modelos UML, além da geração automática dos casos de teste.

Referências

[1] C. Alves, J. B. P. Filho, and J. Castro. Analysing the tradeoffs among requirements, architectures and cots components. In *WER*, pages 20–31, 2001.

[2] P. H. S. Brito, R. de Lemos, E. Martins, and C. M. F. Rubira. Verification and validation of a fault-tolerant architectural abstraction. In *DSN Workshop on Architecting Dependable Systems (WADS 2007)*, Edinburgh, Scotland - UK, 2007. Accepted for publication.

[3] J. Chessman and J. Daniels. *UML Components*. Addison-Wesley, 2000.

[4] P. Clements and R. Kazman. *Software Architecture in Practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[5] F. Cristian. Exception handling. In *Dependability of Resilient Computers*, pages 68–97. Blackwell, 1989.

[6] P. H. da Silva 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 II LADC*, LNCS 3747, pages 61–79, 2005.

[7] M. C. da Silva Jr., P. A. de C. Guerra, and C. M. F. Rubira. A java component model for evolving software systems. In *Proc. of the ASE*, pages 327–330, 2003.

[8] P. A. de C. Guerra, F. C. Filho, V. A. Pagano, and C. M. F. Rubira. Structuring exception handling for dependable component-based software systems. In *EUROMICRO*, pages 575–582, 2004.

[9] 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.

[10] R. de Lemos and A. Romanovsky. Exception handling in the software lifecycle. *IJCSE*, 16(2):167–181, 2001.

[11] C. Dellarocas. Toward exception handling infrastructures for component-based software. In *Proceedings of the International Workshop on Component-based Software Engineering, XX ICSE*, Kyoto, Japan, April 1998.

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

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

[14] I. Jacobson, G. Booch, and J. Rumbaugh. *The unified software development process*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[15] R. Kohavi and G. H. John. Wrappers for feature subset selection. *Artificial Intelligence*, 97(1-2):273–324, 1997.

[16] G. L. Lann. The ariane 5 flight 501 failure - A case study in system engineering for computing systems. Technical Report RR-3079, July 1996.

[17] P. A. Lee and T. Anderson. *Fault Tolerance: Principles and Practice*. Dependable computing and fault-tolerant systems. Springer-Verlag, Berlin ; New York, 2nd edition, 1990.

[18] M. Leuschel and M. J. Butler. Prob: A model checker for b. In *Proceedings of FME'2003*, LNCS 2805, pages 855–874. Springer-Verlag, Pisa, Italy, 2004.

[19] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1st edition, 1988.

[20] R. Pressman. *Software Engineering:a Practitioner's Approach*. McGraw-Hill, 5th edition, 2001.

[21] C. M. F. Rubira, R. de Lemos, G. Ferreira, and F. Castor Filho. Exception handling in the development of dependable component-based systems. *SPE*, 35(5):195–236, March 2005.

[22] M. Shaw and D. Garlan. *Software architecture: perspectives on an emerging discipline*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[23] I. Sommerville. *Software Engineering*. Addison-Wesley, 6th edition, 2001.

[24] C. Szyperski. Component software and the way ahead. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 1, pages 1–20. Cambridge University Press, 2000.