Amanda Sávio Nascimento e Silva

"Uma Infraestrutura Autoadaptativa Baseada em Linhas de Produtos de Software para Composições de Serviços Tolerantes a Falhas"

CAMPINAS
2013

Universidade Estadual de Campinas
Instituto de Computação

Amanda Sávio Nascimento e Silva

"Uma Infraestrutura Autoadaptativa Baseada em Linhas de Produtos de Software para Composições de Serviços Tolerantes a Falhas"

Orientador(a): **Prof^a. Dr^a. Cecília Mary Ficher Rubira**

Tese de Doutorado apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Computação da Universidade Estadual de Campinas para obtenção do título de Doutor em Ciência da Computação.

Este exemplar corresponde à redação final da Tese devidamente corrigida e defendida por Amanda Sávio Nascimento e Silva e aprovada pela Banca Examinadora.

Cecília Mary Ficher Rubira
Assinatura do Orientador(a)

CAMPINAS
2013

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Maria Fabiana Bezerra Muller - CRB 8/6162

N17e

Nascimento e Silva, Amanda Sávio, 1982-
Uma infraestrutura autoadaptativa baseada em linhas de produtos de software para composições de serviços tolerantes a falhas / Amanda Sávio Nascimento e Silva. – Campinas, SP : [s.n.], 2013.

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

1. Tolerância a falha (Computação). 2. Arquitetura orientada a serviços (Computação). 3. Linhas de produto de software. I. Rubira, Cecília Mary Fischer,1964-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: A self-adaptive infrastructure based on software product line for fault-tolerant composite services
Palavras-chave em inglês:
Fault-tolerant computing
Service-oriented network architecture (Computer science)
Software product lines
Área de concentração: Ciência da Computação
Titulação: Doutora em Ciência da Computação
Banca examinadora:
Cecília Mary Fischer Rubira [Orientador]
Patrick Henrique da Silva Brito
Itana Maria de Souza Gimenes
Ariadne Maria Brito Rizzoni Carvalho
Maria Beatriz Felgar de Toledo
Data de defesa: 15-10-2013
Programa de Pós-Graduação: Ciência da Computação

TERMO DE APROVAÇÃO

Tese Defendida e Aprovada em 15 de outubro de 2013, pela Banca examinadora composta pelos Professores Doutores:

[illegible]

Prof. Dr. Patrick Henrique da Silva Brito
IC / UFAL

Itana Gimenes

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

Ariadne M. B. R. Carvalho

Profª. Drª. Ariadne Maria Brito Rizzoni Carvalho
IC / UNICAMP

Maria Beatriz Felgar de Toledo

Profª. Drª. Maria Beatriz Felgar de Toledo
IC / UNICAMP

Uma Infraestrutura Autoadaptativa Baseada em Linhas de Produtos de Software para Composições de Serviços Tolerantes a Falhas

Amanda Sávio Nascimento e Silva[1]

15 de outubro de 2013

[1]Suporte financeiro: CNPQ (processo 141253/2009-6) 03/2009 – 02/2012; CAPES-PDEE (processo 5363-10-1) 01/2011–08/2011; UOL Bolsa Pesquisa (processo 20120217172801) 04/2012 – 03/2013.

Abstract

Nowadays, society depends on systems based on Service-Oriented Architecture (SOA) for its basic day-to-day functioning. As a consequence, these systems should be reliable. Fault-tolerant service compositions encompass a set of services, each with equivalent functionality yet different designs, called alternate services, that are used to implement fault tolerance techniques. A particular technique, for example, Recovery Blocks or N-version Programming, might be more suitable in a context than in another one, depending on non-functional requirements of an application, for example, performance or reliability. SOA-based applications often rely in an environment that is highly dynamic and several decisions should be postponed until runtime, where we have different stakeholders with conflicting requirements, and fluctuations in the quality of services (QoS) are recurrent. Therefore, a fault-tolerant service composition should adapt itself to meet the dynamically and widely changing context. Nevertheless, the existing diversity-based solutions for fault-tolerant service compositions present some drawbacks: (i) they do not support the selection of alternate services that are in fact efficient to support a reliable service composition; (ii) they usually support only one fault tolerance technique, thus not being able to face various clients' requirements; (iii) they do not support an adaptive fault tolerance mechanism able to instantiate different fault tolerance strategies at runtime to cope with dynamic changes in the context. In this thesis, we present a solution based on software product line, which explores the variability among various software fault tolerance techniques and changes in the execution environment, to implement fault-tolerant and self-adaptive service compositions. The proposed solution encompasses: (a) a set of directives to investigate to what extent alternate services are able to tolerate software faults; (b) a family of software fault tolerance techniques to support reliable service compositions, such as the most suitable technique can be chosen according to the context; (c) a self-adaptive infrastructure to instantiate at runtime appropriate fault tolerance techniques in response to changes in the context, through dynamic management of software variability. Results from empirical studies suggest that the proposed solution is efficient to support fault-tolerant and self-adaptive service compositions. Directions for future work are also presented.

Resumo

A confiabilidade é um requisito de qualidade indispensável a muitos sistemas orientados a serviços, cada vez mais disseminados em várias atividades humanas. Composições confiáveis de serviços são formadas por um conjunto de serviços com diversidade de projetos, isto é, um conjunto de serviços funcionalmente equivalentes, ou serviços alternativos, usados para implementar técnicas de tolerância a falhas. Uma determinada técnica, como por exemplo, Recovery Blocks ou N-version Programming, pode ser mais adequada para um contexto específico de execução do que outra, dependendo dos requisitos exigidos pela aplicação, como por exemplo, desempenho. Sistemas orientados a serviços são usualmente implantados num ambiente altamente dinâmico, em que são comuns alterações nos requisitos dos clientes e flutuações na qualidade de serviços. Portanto, uma composição de serviços confiável deveria poder modificar seu próprio comportamento dinamicamente em resposta a essas mudanças. Entretanto, as soluções existentes, que usam diversidade de projetos para implementar composições confiáveis, apresentam algumas limitações: (i) não apóiam a seleção de serviços alternativos adequados que garantam que a composição realmente tolere falhas de software; (ii) em geral implementam uma única técnica de tolerância a falhas, não apoiando os requisitos diversos de clientes; e (iii) não apoiam um mecanismo autoadaptativo capaz de mudar a estratégia de tolerância a falhas em tempo de execução. Nessa tese, é apresentada uma solução baseada em linhas de produtos de software, que explora a variabilidade de software existente nas técnicas de tolerância a falhas e nas mudanças ocorridas no ambiente de execução, para a implementação de composições de serviços tolerantes a falhas e autoadaptativas. A solução encompassa: (a) um conjunto de diretrizes para investigar até que ponto serviços alternativos são realmente diversos entre si para tolerar falhas de software; (b) uma família de técnicas de tolerância a falhas para construir composições confíaveis que permite a escolha de uma técnica mais adequada para o contexto; e (c) uma infraestrutura autoadaptiva que apoia a instanciação de técnicas diferentes de tolerância a falhas como resposta a mudanças ocorridas no contexto, baseando-se no gerenciamento dinâmico de variabilidades de software. Resultados de estudos empíricos sugerem que a solução é eficiente para apoiar composições de serviços tolerantes a falhas e autoadaptativas. Direções para trabalhos futuros são apresentadas.

Às pessoas que amo.

Agradecimentos

Primeiramente, agradeço à minha orientadora Profa. Dr^a. Cecília M. F. Rubira e ao Prof. Dr. Fernando Castor, um grande colaborador neste projeto, pelas oportunidades, dedicação, compreensão e ensinamentos passados. São incontáveis as contribuições dadas não somente a minha formação enquanto pesquisadora mas também enquanto pessoa. Gratidão.

I would like to thank Dr. Jaejoon Lee, my supervisor during my PhD internship in Lancaster University (UK), and Dr. Rachel Burrows, a work colleague. They were always supportive.

Agradeço aos funcionários e pesquisadores do Instituto de Ciências de Computação da Universidade Estadual de Campinas, que me auxiliaram em diferentes etapas deste projeto.

Meus sinceros agradecimentos ao CNPq (processo: 141253/2009-6), à CAPES (processo: 5363-10-1), e à UOL Bolsa Pesquisa (processo: 20120217172801), pelo apoio financeiro para o desenvolvimento deste projeto de pesquisa.

À minha família, não há palavras que expressem a minha gratidão pelo apoio, compreensão, incentivo e ensinamentos. Amo e sinto-me amada, certamente o principal alicerce em tudo que realizo. É um privilégio viver em harmonia com vocês diariamente. Gratidão.

Finalmente, agradeço aos meus amigos e colegas pelo apoio, consciente ou inconsciente, nesta etapa. A special thanks to the new friends I could meet during my stay in Lancaster, UK.

"A writer doesn't solve problems. He allows them to emerge".
Friedrich Dürrenmatt

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Em 1968, na primeira conferência de engenharia de software apoiada pela OTAN[1], foi cunhada a expressão *crise de software* em referência ao problema de desenvolver sistemas de software grandes e confiáveis de maneira sistemática e efetiva [6]. Convidado a escrever um artigo para a conferência, Mcllroy [7] enfatizou a importância da utilização de técnicas para produção em massa de software, dado que fomentar o reúso de software seria uma forma de superar a crise [8]. Segundo Mcllroy, a replicação de software tem uma vantagem em relação às demais indústrias por não demandar matéria-prima. De maneira complementar, Randell [6], editor e participante da conferência, pontuou que possivelmente haveria um incentivo para melhorar a qualidade inicial do software caso seus custos de replicação fossem similares aos custos provenientes da replicação de hardware.

Uma das técnicas que surgiram para promover o reúso foi o desenvolvimento baseado em componentes (DBC), que visa a construção rápida de sistemas a partir de componentes pré-fabricados [9]. Conforme a definição proposta por Szyperski [10], um componente de software é uma unidade com interfaces e dependências de contexto explicitamente especificadas, que pode ser fornecido isoladamente para integrar sistemas de software desenvolvidos por terceiros. Por meio de interfaces providas e interfaces requeridas, os componentes de software, respectivamente, disponibilizam seus serviços e explicitam os serviços dos quais dependem. Essas propriedades contribuem para aumentar a coesão dos componentes de software e diminuir o acoplamento entre eles [11].

Na década de noventa, a disciplina chamada de arquitetura de software foi um dos grandes focos da atenção dos pesquisadores. De acordo com Bass et al. [12], a arquitetura de software define uma estrutura de alto nível do sistema composta por elementos arquiteturais e pelo relacionamento entre eles. Elementos arquiteturais podem ser compostos por componentes arquiteturais e conectores arquiteturais. Um componente arquitetural é responsável pelo processamento e armazenagem de dados relacionados às funcionalidades do

[1]sigla para Organização do Tratado do Atlântico Norte

sistema [13]. Já um conector arquitetural é responsável por intermediar a comunicação entre componentes arquiteturais. Uma configuração arquitetural, por sua vez, define um conjunto de componentes arquiteturais ligados por conectores arquiteturais [14]. Dessa forma, o conceito de arquitetura de software pode ser vista como complementar ao desenvolvimento baseado em componentes.

No cenário atual, de aplicações dinâmicas e distribuídas, a interoperabilidade, definida como a capacidade de dois ou mais sistemas heterogêneos compartilharem informação de maneira eficiente [15], é um requisito fundamental. A adoção de Arquiteturas Orientadas a Serviços (SOA[2]) popularizou-se a partir do ano de dois mil e cinco como uma das principais abordagens para aumentar o grau de interoperabilidade de sistemas. SOA é um modelo de componentes de software que interrelaciona diferentes unidades funcionais, chamadas serviços, por intermédio de interfaces bem definidas, que são independentes de plataformas e linguagens de implementação [16]. A computação orientada a serviços pode ser considerada uma abordagem de reúso interorganizacional bem sucedida [17]. Serviços permitem às organizações exporem suas competências e capacidades de negócios na Internet (ou Intranet) programaticamente ao utilizar padrões e protocolos abertos [17].

Desta forma, serviços que são construídos sobre uma gama heterogênea de sistemas interagem entre si de maneira uniforme por intermédio de composições de serviços [17]. Uma composição de serviços, ou um serviço composto, pode ser definida como uma combinação de atividades invocadas numa ordem predefinida e executadas como um todo [16]. Serviços Web constituem uma das tecnologias que apoia a realização de SOAs e são usualmente descritos, publicados e invocados mediante padrões XML, eXtensible Markup Language, (*e.g.* SOAP - *Simple Object Access Protocol*, WSDL - *Web Services Description Language*). Além do mais, recentemente, serviços Rest (*Representational State Transfer*) tem sido amplamente utilizados como uma alternativa para a tecnologia de serviços Web mais eficiente. Dentre os diversos tipos de serviços, no presente trabalho focamos Serviços Web. Por essa razão, de agora em diante nos referimos a serviços Web apenas como serviços.

Um dos desafios computacionais inerente à heterogeneidade, distribuição e autonomia das diferentes entidades que participam em *sistemas orientados a serviços* refere-se ao fato que estes sistemas devem apoiar a seleção, composição e invocação dos seus serviços em tempo de execução. Além disso, estes sistemas, de modo frequente, são implantados num ambiente altamente dinâmico em que *(i)* é comum a existência de vários clientes com, possivelmente, requisitos conflitantes; *(ii)* flutuações nos valores de atributos de qualidade de serviços (QoS[3]) são recorrentes; *(iii)* instabilidades no ambiente de execução são usuais; e *(iv)* frequentemente algumas decisões de projeto devem ser selecionadas em tempo de

[2]Do inglês, *Service-Oriented Architecture* - SOA
[3]Do inglês, *Quality of Services (QoS)*

execução [17, 16]. Em outras palavras, sistemas orientados a serviços, de modo usual, compõem um conjunto emergente de sistemas de software conhecidos como *sistemas de software autoadaptativos* footnoteDo inglês, *Self-adaptive systems* [17, 18, 16].

Um sistema de software autoadaptativo modifica seu próprio comportamento em resposta à percepção de mudanças no *ambiente de execução* e no sistema em si, ou seja, mudanças no contexto [19, 18]. Em dois mil e dois, Garlan, Wolf e Kramer organizaram o primeiro workshop [20] (1st ACM Workshop on Self-Healing Systems (WOSS 02)) direcionado à discussão acerca de sistemas autocuráveis[4] e, de uma forma mais abrangente, sistemas dinamicamente adaptáveis[5]. Eles enfatizaram que, no passado, os sistemas que apoiavam a autoadaptação eram raros, e, situavam-se, essencialmente, em áreas de domínio em que a intervenção humana em sistemas de software não era viável (*e.g.* sistemas de software embarcados em equipamentos lançados ao espaço para análises diversas). Entretanto, outros sistemas passaram a apresentar requisitos de autoadaptação, incluindo sistemas de *e-commerce* e sistemas embarcados em dispositivos móveis. Consequentemente, sistemas autoadaptativos passaram a ser estudados em áreas diversas de conhecimento, tais como, sistemas de controle, arquiteturas de software, computação tolerante a falhas e redes neurais [20, 21].

Na década de setenta, Parnas [22] apresentou o desenvolvimento de artefatos de software reutilizáveis pertencentes a um domínio específico ao propor o conceito de famílias de produtos, visando prover variabilidades de requisitos não-funcionais. Variabilidade de software é a capacidade que um sistema de software ou artefato tem de ser modificado ou configurado para ser usado em contexto específico [23]. A variabilidade de software é composta por pontos de variação, os locais dos artefatos de um software em que decisões de escolha do produto podem ser tomadas, e variantes, as alternativas associadas aos pontos de variação. Kang et al. [24], nos anos noventa, propuseram a análise de domínio orientada a características com o propósito de modelar variabilidades de software. Característica é uma propriedade de sistema que é relevante para alguma parte interessada [25]. Um modelo de características[6] pode ser usado para representar as características de um domínio, e é definido como um modelo gráfico e hierárquico que representa as características como obrigatórias, opcionais ou alternativas [5].

Esses trabalhos de Parnas [22] e Kang et al. [24] foram essenciais para o surgimento do conceito de linhas de produtos de software (SPL[7]), um conjunto de sistemas de software que compartilham características comuns e possuem características distintas visando satisfazer as necessidades de um nicho de mercado [26]. O principal propósito da engenharia

[4]Do inglês, *Self-healing systems*

[5]O segundo workshop neste tópico, *2nd ACM SIGSOFT Workshop on Self-Managed Systems (WOSS 04)*, teve seu nome alterado para refletir essa visão mais ampla sobre a adaptação dinâmica.

[6]Do inglês, *Feature model*

[7]Do inglês, *Software Product Line (SPL)*

de linhas de produtos é oferecer produtos personalizados a um custo razoável [27]. O reúso de software em linhas de produtos cria uma infraestrutura de apoio necessária para que possa ser aplicado na prática. Em face disso, ao apoiar o reúso de software planejado, linhas de produtos diferem das abordagens anteriores [28]. As arquiteturas de linhas de produtos (PLAs[8]) são artefatos que ajudam a lidar com a complexidade das linhas de produtos abstraindo seus detalhes de implementação [12]. Uma arquitetura da linha de produto representa as variabilidades de software envolvendo os elementos arquiteturais do sistema: componentes, conectores e configuração arquitetural [27].

A variabilidade arquitetural reflete a existência de alternativas de projeto arquitetural e é expressa por meio de um conjunto de pontos de variação e variantes arquiteturais [29]. Pontos de variação arquiteturais são locais na arquitetura nos quais decisões relacionadas à escolha de produtos são tomadas. Elas representam as variações do modelo de características no modelo de arquitetura [29]. As variantes arquiteturais representam as alternativas associadas a um determinado ponto de variação arquitetural. A partir da resolução de pontos de variação arquitetural, é possível escolher quais variantes vão fazer parte de um determinado produto de software num processo chamado de configuração do produto [27].

Embora engenharias tradicionais de SPLs reconheçam a possibilidade de postergar tomadas de decisão para tempo de execução, SPLs geralmente gerenciam suas variabilidades em tempo de projeto. Em contraste, *Linhas de Produtos de Software Dinâmicas* (DSPLs[9]), uma proposta que popularizou-se com Hallsteinsen et al. [30] em dois mil e oito, estendem SPLs a fim de apoiar tomadas de decisões dinamicamente [18, 31, 32]. DSPLs são usualmente utilizadas para projetar e implementar sistemas autoadaptativos, uma vez que apoiam a identificação e representação explícita das partes desses sistemas e seleção de suas partes variantes em tempo de execução a partir do gerenciamento dinâmico de variabilidades de software.

1.1 Contexto

Na atualidade, a sociedade é dependente de sistemas orientados a serviços para realizar tarefas rotineiras, tais como recuperar valores de câmbios de moedas, efetuar compras em lojas virtuais, consultar possíveis rotas entre duas coordenadas geográficas, comprar passagens de voos e efetuar reservas diversas [17, 16, 33, 34]. Ademais, diferentes organizações cooperam a fim de oferecer serviços mais elaborados ou sofisticados para seus clientes, possibilitando, assim, a integração de seus negócios [16]. Exemplos típicos deste cenário são: *(i)* o setor financeiro global, em que diferentes bancos se fundem para se tornarem mais

[8]Do inglês, *Product Line Architecture (PLA)*
[9]Do inglês, *Dynamic Software Product Lines (DSPLs)*

competitivos; *(ii)* a integração de diferentes agências do setor de viagens; *(iii)* pequenas e médias empresas que, apesar de cada uma ter suas próprias estratégias de negócios, decidem se unir em uma rede para se tornarem mais competitivas; e *(iv)* a adoção de Governo Eletrônico, ou *e-Government*, que integra aplicações dos setores públicos a fim de oferecer serviços públicos tradicionais também por canais eletrônicos [35]. Essa disseminação de sistemas orientados a serviços em várias atividades humanas torna a confiabilidade[10] uma característica indispensável a esses sistemas [36, 37, 38, 39, 34, 40, 41].

Confiabilidade é um dos atributos de dependabilidade[11], que involve outros atributos, tais como, disponibilidade, segurança, integridade e manutenibilidade [1]. Confiabilidade pode ser definida como a capacidade do sistema de oferecer suas funcionalidades conforme a especificação por um tempo determinado [15]. Quando um sistema é executado sob vários casos de entrada, a sua taxa, ou frequência, de execuções bem sucedidas pode ser utilizada como uma estimativa de sua confiabilidade [42]. De modo geral, o grau de confiança de um sistema depende principalmente do número de falhas contidas no mesmo e da maneira como ele se comporta na presença delas [42, 1]. Falhas[12] são eventos que causam erros[13]. Se a falha não for tolerada ou o erro correspondente tratado, ocorrerá um defeito[14], que se manifestará pela mudança indesejada na funcionalidade [1].

As principais técnicas utilizadas para o desenvolvimento de sistemas de software com requisitos de confiabilidade são prevenção de falhas, detecção de falhas e tolerância a falhas [1]. Em particular, estamos interessados em tolerância a falhas, definida como a capacidade do sistema se comportar de maneira bem-definida uma vez que falhas ocorram [43, 44]. Ao projetar um sistem tolerante a falhas, o primeiro pré-requisito é especificar, por meio de um *modelo de falhas*, as falhas que devem ser toleradas [44]. O próximo passo é adicionar redundância ao sistema, isto é, recursos adicionais utilizados para detectar e tolerar as falhas identificadas [42, 44]. Tais recursos não seriam necessários se a tolerância a falhas não fosse implementada [43, 42, 1]. Redundância pode ser inserida de diferentes formas, por exemplo, redundância de hardware, de software e de tempo [1]. Em particular, a redundância de software pode ser empregada de forma implícita, mediante mecanismos de tratamento de exceções, e de forma explícita, mediante técnicas de tolerância a falhas baseadas em diversidade de projetos [45].

Nesta tese, focamos o uso de técnicas de tolerância a *falhas de software* baseadas em *diversidade de projetos*. Falhas de software, conforme definição de Pullum [1], podem ocorrer por dois principais motivos: *(i)* devido a especificação incorreta de requisitos - o desenvolvimento do sistema de software satisfaz os requisitos, no entanto, os requisitos não

[10]Do inglês, *reliability*
[11]Do inglês, *dependability*
[12]Do inglês, *Fault*
[13]Do inglês, *Error*
[14]Do inglês, *Failure*

foram especificados de forma apropriada; e *(ii)* devido a uma implementação inadequada que não satisfaz os requisitos - tal implementação é resultante de um projeto incorreto ou uma codificação incorreta [1]. Falhas de software são também conhecidas como falhas de projeto ou *bugs* [43, 1]. As técnicas baseadas em diversidade de projetos para tolerar falhas de software são implementadas a partir do uso de múltiplas, ou alternativas, versões de componentes de software que são funcionalmente equivalentes e, no entanto, projetados e desenvolvidos de forma independente [46, 1]. A partir de agora, para simplificar, os termos falhas e *falhas de software* são utilizados de forma intercambiáveis *neste* texto, embora, conforme discutido, o termo *falha* contenha um significado mais abrangente do que *falha de software*.

O mecanismo básico das técnicas de tolerância a falhas baseadas em diversidade de projetos para satisfação de confiabilidade é o seguinte [1]: valores de entradas são providos para os componentes de software alternativos. Estes componentes executam suas operações sob estes valores. Ao término das execuções, possivelmente, haverá múltiplos valores de retorno. Deste conjunto de valores, um resultado considerado correto, aceitável ou mais promissor deve ser escolhido, caso exista. Esta tarefa de decisão é realizada por um mecanismo chamado juiz (*e.g.* teste de aceitação ou votadores [1]). O resultado escolhido é, então, retornado a unidade subsequente de execução do sistema de software. Cabe ressaltar que uma determinada técnica para tolerância a falhas baseada em diversidade de projetos, como por exemplo, *Recovery Blocks* ou *N-version Programming*, pode ser mais adequada para um contexto específico de execução do que outra, dependendo dos requisitos exigidos pela aplicação, como por exemplo, desempenho [47, 48, 46]. Além disso, ao se empregar diversidade de projetos, é esperado que os componentes de software alternativos falhem raramente para os mesmos casos de entrada, pois isso aumenta a probabilidade de falhas serem toleradas [49, 50, 51, 52, 1]. Quando componentes de software alternativos falham para um mesmo caso de entrada ocorre uma *falha coincidente*, o que pode levar a uma mudança indesejada do comportamento do juiz [1].

Na década de noventa, o uso de técnicas baseadas em diversidade de projetos para tolerar falhas de software foi bastante criticado pois, a partir de uma especificação comum, componentes de software alternativos eram desenvolvidos basicamente a partir do zero - um processo bastante custoso [53, 54]. À vista disso, tais técnicas eram geralmente utilizadas em sistemas altamente críticos, em que a ocorrência de falhas acarretaria em grandes prejuízos financeiros ou, até mesmo, perda de vida [1]. Não obstante, atualmente, no contexto de arquitetura orientadas a serviços, é possível encontrar na Internet, um ambiente aberto e dinâmico, serviços funcionalmente equivalentes [55, 40, 56, 16]. Estes serviços, aqui chamados de serviços alternativos, podem estar disponíveis gratuitamente ou serem oferecidos por diferentes organizações para seus parceiros de negócios, inclusive, a um custo financeiro pré-determinado [17, 17, 57].

Devido à facilidade de reutilização de serviços alternativos, vários pesquisadores propuseram a adoção de técnicas de tolerância a falhas baseadas em diversidade de projetos com a finalidade de aumentar a confiabilidade de sistemas orientados a serviços [36, 37, 38, 33, 58, 39, 34, 40, 41, 59]. Nessas soluções, composições confiáveis de serviços são formadas por um conjunto de serviços alternativos, usados para implementar técnicas de tolerância a falhas. Em linhas gerais, estas soluções atuam como conectores arquiteturais entre clientes e serviços alternativos. A facilidade de reutilização de serviços alternativos para tolerar falhas permite que sistemas orientados a serviços sejam confiáveis a fim de evitar também pequenos contratempos, além de prejuízos e danos maiores. Cabe ressaltar que, em geral, os usuários estão cada vez mais exigentes no que se refere aos requisitos de qualidade e eles são pouco tolerantes com os defeitos de sistemas de software [34, 33, 58].

1.2 Definição do Problema e Questões de Pesquisa

Sistemas orientados a serviços diferem de sistemas tradicionais devido a suas características de heterogeneidade, distribuição, autonomia e dinamismo [17, 60, 16]. Consequentemente, esses sistemas fomentam novos desafios de pesquisas no que tange a utilização de soluções baseadas em diversidade de projetos para tolerar falhas de software. O problema que esta tese se propõe a resolver é dividido em três subproblemas. Primeiro, não há diretrizes para apoiar a seleção de serviços alternativos adequados e que garantam que uma composição realmente tolere falhas de software. Segundo, há uma lacuna no que tange a existência de soluções capazes de apoiar estratégias de tolerância a falhas apropriadas para requisitos diversos de clientes e contextos de uso - a flexibilidade é um requisito de qualidade fundamental em sistemas orientados a serviços [17, 60, 16]. Terceiro, há uma carência por infraestruturas autoadaptivas que apoiem várias técnicas de tolerância a falhas simultaneamente e a escolha da técnica mais apropriada para os diferentes contextos em tempo de execução - a autoadptação é outro requisito fundamental desses sistemas [18, 61, 62]. Esses problemas são motivados e descritos nas próximas subseções. Além disso, a cada um desses problemas são atribuídos uma ou mais questões de pesquisa.

1.2.1 Insuficiência de Evidências Relativas a Eficiência de Serviços Alternativos para Tolerar Falhas de Software

Ao utilizar técnicas baseadas em diversidade de projetos para tolerar falhas de software, o projetista assume que falhas coincidentes entre os componentes de software alternativos são raras, e, se existirem, os resultados das execuções serão suficientemente diferentes para apoiar a detecção de erros e distinção de um resultado correto ou aceitável [1]. Quanto mais diversos são os componentes de software alternativos (*e.g.* desenvolvidos a partir de diferentes linguagens e plataformas ou diferentes algoritmos), menor a probabilidade deles falharem para um mesmo caso de entrada, e, consequentemente, maior a probabilidade de falhas de um ou mais destes componentes serem toleradas, se necessário. Neste sentido, análises empíricas de diversidade de projetos em componentes de software funcionalmente equivalentes constituíram um importante tópico de pesquisa entre meados da década de oitenta e início da década de noventa [1].

Um número considerável de estudos e experimentos foram realizados a fim de mensurar quão diversos e eficientes componentes de software alternativos eram para tolerar falhas [49, 50, 51, 52]. Na maioria das pesquisas, os times de desenvolvimento e a plataforma adotada para implementar os componentes de software alternativos, bem como seus códigos fonte, eram bem conhecidos. Os resultados obtidos sugerem que alcançar os benefícios advindos da utilização de diversidade de projetos para apoiar sistemas de software confiáveis pode não ser tão simples. Em muitos casos, contraditoriamente, é possível obter maiores índices de confiabilidade ao adotar um componente de software simples do que ao usar múltiplos componentes de software na tentativa de tolerar falhas [49, 52].

No contexto de sistemas orientados a serviços, as autores de soluções baseadas em diversidade de projetos assumem de modo intrínseco que serviços alternativos são sempre eficientes para tolerar falhas de software [36, 37, 38, 39, 34, 40, 41]. No entanto, não há na literatura evidências suficientes para ratificar esta suposição. Serviços são caixas-pretas de reúso e estão usualmente sob controle de terceiros, isto é, de organizações independentes [17, 16, 60]. O termo caixa-preta refere-se ao fato que somente as interfaces de requisições dos serviços são disponíveis e usualmente não há detalhes acerca de como foram projetados, implementados, ou mesmo especificados [17]. Por consequência, torna-se difícil extrapolar conclusões dos estudos prévios sobre diversidade de projetos para o contexto de serviços alternativos. Dessa forma, é essencial buscar evidências que corroboram ou refutam a hipótese de que serviços alternativos são sempre eficientes para apoiar o aumento da confiabilidade de sistemas orientados a serviços. À vista disso, definimos as nossas primeiras questões de pesquisa.

Questão de Pesquisa 1 (QP1) Como mensurar se serviços alternativos são diversos e eficientes para tolerar falhas de software quando alavancados por técnicas baseadas em diversidade de projetos?

Questão de Pesquisa 2 (QP2) Quais as implicações de se utilizar serviços alternativos para tolerar falhas de software?

Até onde sabemos, dado um requisito funcional e seus respectivos serviços alternativos, não há diretrizes para analisar *(i)* se estes serviços são de fato providos por diferentes projetos; e *(ii)* a frequência com que estes serviços falham para um mesmo caso de entrada.

Ao estudarmos de maneira adequada a diversidade de serviços, seremos capazes de apoiar um maior grau de confiabilidade ao implantar uma composição que alavanca serviços alternativos para tolerar falhas, ou ao implantar um serviço simples que exiba maior grau de confiabilidade do que seria o caso se diversidade de projetos fosse adotada.

1.2.2 Ausência de Soluções que Apoiam Diferentes Técnicas Baseadas em Diversidade de Projetos

As várias técnicas de tolerância a falhas baseadas em diversidade de projetos apresentam diferentes valores para requisitos de qualidade (*e.g.* confiabilidade, tempo de resposta e consumo de memória) [1] e facilidades para julgar diferentes tipos de resultados retornados após a execução dos componentes de software alternativos (*e.g.* um tipo de dado complexo ou simples). Em linhas gerais, estas técnicas diferem em termos de escolhas de projetos no que tange *(i)* esquemas para executar componentes de software alternativos (*e.g.* execução em sequencial ou em paralelo); *(ii)* tipos juízes (*e.g.* votadores, testes de aceitação ou variações desses); e *(iii)* componentes de software alternativos que são utilizados [46, 1]. Consequentemente, uma determinada técnica pode ser mais adequada para um contexto específico de execução do que outra, dependendo dos requisitos exigidos pela aplicação.

Apesar da exigência por sistemas orientados a serviços confiáveis, o mercado de software atual apresenta outros requisitos aparentemente antagônicos, como a necessidade desses sistemas serem flexíveis a fim de facilmente acomodarem requisitos, possivelmente conflitantes, dos seus diversos clientes [17, 16]. Conforme mencionado, variando os requisitos funcionais e de qualidade, pode ser necessário utilizar técnicas mais adequadas de tolerância a falhas. Além disso, sistemas orientados a serviços devem ser flexíveis de modo a apoiar diversos requisitos de clientes. Por consequência, soluções baseadas em diversidade de projetos para composições confiáveis de serviços, idealmente, deveriam apoiar técnicas de tolerância a falhas customizadas para diferentes clientes e contextos. No entanto, não há na literatura *(i)* um mapeamento de quais escolhas de projetos são apoiadas pelas soluções existentes para composições de serviços tolerantes a falhas; *(ii)* diretrizes indicando quais escolhas de projetos são mais eficientes para determinados requisitos da aplicação; e *(iii)* soluções que apoiem várias técnicas de tolerância a falhas simultaneamente, e, consequentemente, que apoiem requisitos diversos de clientes. Nesse sentido, surgem as seguintes questões de pesquisas.

Questão de Pesquisa 3 (QP3) Quais as escolhas de projetos que são apoiadas por soluções existentes para composições de serviços tolerantes a falhas de software? Quais as principais diferenças entre essas escolhas de projetos no que tange os requisitos de qualidade?

Questão de Pesquisa 4 (QP4) Como apoiar uma infraestrutura que acomode de forma planejada diferentes técnicas de tolerância a falhas?

Ao respondermos a QP3, seremos capazes de pontuar as principais similaridades e diferenças entre as soluções existentes, bem como apoiar a escolha de soluções mais adequadas para diferentes contextos conforme requisitos de diversos clientes. Ademais, a partir da análise das soluções existentes e suas implicações, é possível identificar oportunidades de progressos no que se refere a propostas de composições de serviços tolerantes a falhas. Referente a QP4, tal infraestrutura permitiria reutilizar as escolhas de projetos apoiadas por soluções existentes para composições de serviços tolerantes a falhas, como também acomodar outras escolhas de projetos não contempladas por tais soluções.

1.2.3 Ausência de Mecanismos Adequados para Adaptação Dinâmica das Composições Confiáveis de Serviços

Outro requisito de qualidade essencial para sistemas orientados a serviços é a autoadaptação em resposta a mudanças dinâmicas nos requisitos dos clientes, variações na disponibilidade de recursos de hardware e de software, e flutuações nos valores de atributos de qualidade dos serviços (QoS) [63, 19, 21, 64]. Em face disso, soluções para composições de serviços tolerantes a falhas, idealmente, deveriam apoiar técnicas de tolerância a falhas adaptadas para o contexto corrente. Entretanto, a adoção de um mecanismo autoadaptativo para tolerância a falhas, de um modo geral, implica em alguns desafios de pesquisa, conforme descrito a seguir [63]. Primeiro, a lógica de adaptação idealmente deve ser separada da lógica de tolerância a falhas [63, 65], de modo a facilitar a compreensão, manutenção e modularidade do mecanismo como um todo. Segundo, além desta separação de interesses, é importante manter uma separação explícita dos componentes de software responsáveis por realizar a lógica de adaptação, dos componentes que realizam a lógica de tolerância a falhas [65]. Terceiro, esta separação, por sua vez, requer o uso de um conjunto bem estruturado de modelos de alto nível que são utilizados para abstrair detalhes do ambiente de execução, do estado do sistema propriamente dito, e de requisitos dos usuários [66, 67]. Estes modelos de abstração idealmente deveriam compor uma base para a análise e tomadas de decisões acerca do comportamento dinâmico da lógica de tolerância a falhas em conformidade com o contexto [68, 67]. Finalmente, alterações em tais abstrações de alto nível deveriam ser refletidas para o sistema em execução, a fim de efetivamente adaptar o mecanismo de tolerância a falhas dinamicamente [66, 67]. Neste sentido, a quinta questão de pesquisa é:

Questão de Pesquisa 5 (QP5) Como apoiar de forma adequada uma infraestrutura autoadaptiva para apoiar a instanciação de técnicas diferentes de tolerância a falhas em resposta a mudanças ocorridas no contexto?

Referente a QP5, a partir da análise da literatura relacionada [34, 41, 58, 33], observamos que há uma lacuna no que se refere a existência de soluções baseadas em diversidade de projetos que apoiem, de forma adequada, composições de serviços tolerantes a falhas e autoadaptáveis.

1.3 Solução Proposta

Esta tese descreve uma solução que visa avançar o estado da arte no que tange a concepção e implementação de sistemas confiáveis orientados a serviços mediante técnicas de tolerância a falhas de software baseadas em diversidade de projetos. A solução proposta contempla: *(i)* uma infraestrutura que apoia a seleção de serviços alternativos eficientes que garantam que a composição realmente tolere falhas de software; *(ii)* uma família de técnicas de tolerância a falhas para construir composições confiáveis que permita a escolha de uma técnica mais adequada para o contexto; e *(iii)* uma infraestrutura capaz de apoiar várias técnicas de tolerância a falhas de software simultaneamente, de modo que a técnica mais adaptada é instanciada em tempo de execução, baseando-se no gerenciamento dinâmico de variabilidades de software. A escolha da técnica mais adequada é realizada em conformidade com políticas de alto nível pré-estabelecidas e percepção do contexto corrente (*e.g.* flutuações na qualidade de serviços e mudanças de requisitos de clientes).

Para alcançar estes três objetivos, respondemos cada uma das questões de pesquisa, apresentadas na seção anterior, combinando técnicas de quatro disciplinas: estudos empíricos em engenharia de software, linhas de produtos de software (dinâmicas), sistemas autoadaptativos, desenvolvimento centrado na arquitetura de software e desenvolvimento baseado em componentes, conforme descrito nas seções subsequentes. Até onde sabemos, o uso de linhas de produtos de software (dinâmicas) para tolerar falhas de software em sistemas orientados a serviços é uma das contribuições originais desta pesquisa. Além disso, soluções existentes baseadas em linhas de produtos de software (dinâmicas) foca variabilidade funcional (*e.g.* [69, 18, 70, 71, 72, 73]). Ao contrário, neste trabalho, focamos variabilidade não-funcional ao explorar a variabilidade de software existente nas técnicas de tolerância a falhas e nas mudanças ocorridas no ambiente de execução.

1.3.1 Uma Infraestrutura para Mensurar Diversidade de Serviços Alternativos e suas Implicações

Questão de Pesquisa 1 (QP1) Como mensurar se serviços alternativos são diversos e eficientes para tolerar falhas de software quando utilizados para implementar técnicas baseadas em diversidade de projetos?

Questão de Pesquisa 2 (QP2) Quais as implicações de se utilizar serviços alternativos para tolerar falhas de software?

Com o objetivo de responder a QP1, propomos uma infraestrutura que engloba um conjunto de diretrizes e ferramentas para apoiar a preparação e a execução de estudos empíricos para investigar, dado a especificação de um requisito funcional, até que ponto seus serviços alternativos são eficientes para tolerar falhas de software quando estruturados mediante técnicas baseadas em diversidade de projetos. Uma vez que serviços são caixas-pretas, toda a investigação é realizada a partir do ponto de vista do cliente [16, 60]. Primeiro, a infraestrutura, através do uso de testes estatísticos, permite verificar se os serviços alternativos apresentam diferenças significativas em seus valores de saída e frequência de falhas quando executados sob uma mesma sequência de valores de entrada. Se os serviços alternativos apresentarem diferentes comportamentos observados, isso sugere que eles são de fato providos por diferentes implementações, e, possivelmente, projetos. Caso contrário, consideramos que não temos evidências suficientes para afirmar se eles são ou não diversos. Segundo, a infraestrutura apoia o cálculo estimado da confiabilidade alcançada por serviços individualmente (*i.e.* serviços simples) e pela composição que usa tais serviços alternativos para tolerar falhas de software. Resultados de estudos empíricos, nos quais analisamos a eficiência de serviços alternativos gratuitos, baseados em SOAP/WSDL e disponíveis na Internet para tolerar falhas de software, sugerem a viabilidade e eficiência da solução proposta.

Conforme mencionado, as soluções existentes para mensurar diversidade de componentes de software alternativos não são aplicáveis a serviços alternativos que mantêm somente suas interfaces de requisições disponíveis (*i.e.* são caixas pretas) [49, 50, 51, 52]. Portanto, uma contribuição original desta tese é uma infraestrutura que apoia um conjunto de diretrizes para investigar a diversidade de serviços alternativos e a eficiência dos mesmos para tolerar falhas de software. Essa infraestrutura resultou na seguinte publicação:

Nascimento A.S.; Castor, F.; Rubira, C. M. F; Burrows, R. *An experimental setup to assess design diversity of functionally equivalent services.* 16th International Conference on Evaluation & Assessment in Software Engineering, 2012, Ciudad Real, Spain.

Posteriormente, a fim de responder a QP2, a infraestrutura proposta foi utilizada em estudos empíricos mais abrangentes com a finalidade de obtermos um maior entendimento das implicações de se empregar diversidade de projetos para apoiar composições de serviços tolerantes a falhas. Os resultados obtidos sugerem que pode existir diversidade na implementação de serviços alternativos. Entretanto, em algumas circunstâncias, a frequência com que os serviços alternativos falham num mesmo caso de entrada pode ser tão alta que usar um serviço isoladamente pode produzir melhores resultados de confiabilidade do que empregar técnicas de diversidade de projetos. Estes resultados reforçam a relevância da infraestrutura proposta para mensurar a diversidade de serviços alternativos. Conforme mencionado, as soluções existentes partem do pressuposto que serviços alternativos são sempre eficientes para tolerar falhas de software. Entretanto, obtivemos evidências que refutam esta hipótese por intermédio de estudos empíricos. Um maior entendimento das implicações de se utilizar serviços alternativos para tolerar falhas de software é uma contribuição original desta tese. Esses estudos empíricos resultaram nas seguintes publicações.

- **Nascimento A.S.**; Castor, F.; Burrows, R.; Rubira, C.M.F. *An Empirical Study on Design Diversity of Functionally Equivalent Web Services.* 7th International Conference on Availability, Reliability and Security, 2012, Prague, Czech Republic.
- **Nascimento A.S.**; Castor, F.; Burrows, R.; Rubira, C.M.F. *An Empirical Study on Design Diversity of Functionally Equivalent Web Services.* IC, UNICAMP, Tech. Rep. IC-12-18, 2012. (versão estendida do artigo anterior)

1.3.2 Uma Família de Técnicas de Tolerância a Falhas de Software Baseadas em Diversidade de Projetos

Questão de Pesquisa 3 (QP3) Quais escolhas de projetos são apoiadas por soluções existentes para serviços compostos tolerantes a falhas de software? Quais as principais diferenças entre essas escolhas de projetos no que tange os requisitos de qualidade?

Questão de Pesquisa 4 (QP4) Como apoiar uma infraestrutura de reúso que acomode de forma planejada diferentes técnicas de tolerância a falhas?

Com o objetivo de responder a QP3, realizamos uma revisão sistemática para indentificarmos as semelhanças e diferenças entre as várias soluções existentes para composições de serviços tolerantes a falhas. Tais soluções constituem nossos estudos primários. A revisão sistemática foi estruturada a partir de diretrizes propostas por Kitchenham e Charters [74]. Estas diretrizes constituem três fases da revisão sistemática: planejamento, execução, e apresentação dos resultados obtidos. Os estudos primários analisados foram classificados com base numa taxonomia que identifica as principais escolhas de projetos referentes às diversas técnicas de tolerância a falhas (*e.g.* os diferentes esquemas de execução de serviços alternativos e os diferentes juízes) [1]. Realizamos também um levantamento inicial de quais requisitos de qualidade são apoiados pelas escolhas de projetos identificadas (*i.e.* a execução sequencial de serviços alternativos requer menos memória do que a execução paralela de tais serviços), facilitando a escolha de soluções mais apropriadas para diferentes contextos. Além disso, apontamos referências para estudos mais aprofundados acerca da utilização das diferentes escolhas de projetos e suas implicações [49, 50, 51, 52]. Esses estudos são baseados em diferentes suposições relativas a modelos de falhas. Logo, tais estudos compõem um importante ponto de partida para elaboração de modelos de falhas também adequados para diferentes contextos.

Cabe ressaltar que as publicações relativas às soluções existentes para composições de serviços tolerantes a falhas são baseadas em diferentes contextos conceituais e técnicos [36, 37, 38, 39, 34, 40, 41], o que dificulta a escolha entre as soluções existentes. Portanto, uma outra contribuição desta tese é uma descrição padronizada das soluções identificadas. Enfatizamos, ainda, que estudos empíricos em *Engenharia de Software* são essenciais, pois permitem a construção de uma base de conhecimento científico sobre o quão eficientes são diferentes soluções para engenharia de software, permitindo uma escolha fundamentada [75]. A revisão sistemática resultou na seguinte publicação:

- **Nascimento A.S.**; Rubira, C. M. F.; Burrows, R.; Castor, F. *A systematic review of design diversity-based solutions for fault-tolerant SOAs.* 17th International Conference on Evaluation and Assessment in Software Engineering, 2013, Porto de Galinhas, PE, Brazil.

Com o objetivo de responder a QP4, projetamos uma família de técnicas de tolerância a falhas baseadas em diversidade de projetos para apoiar composições confiáveis de serviços. Conforme mencionado, linhas de produtos apoiam o reúso de software de forma planejada e a criação de produtos personalizados a um custo razoável [28]. A partir da análise do domínio de tolerância a falhas de software [45, 1, 46, 48, 76] e de composição de serviços [17, 16, 60], extraímos as características comuns e variáveis entre as técnicas de interesse. A análise de domínio também contemplou: *(i)* o levantamento de características peculiares a sistemas orientados a serviços, como interoperabilidade e descoberta e invocação dinâmicas de serviços alternativos [17, 16, 60]; e *(ii)* as semelhanças e diferenças, identificadas a partir da revisão sistemática das soluções existentes para composições confiáveis de serviços [57].

As características identificadas foram inicialmente estruturadas num modelo de características e posteriormente mapeadas para uma arquitetura de linha de produtos baseada em componentes de software [2]. Mais especificamente, o modelo de características proposto estende o modelo previamente apresentado por Brito et al. [45], a fim de explicitamente apoiar características peculiares a sistemas orientados a serviços. Para a implementação da arquitetura da linha de produtos, utilizamos um modelo independente de plataforma tecnológica para projeto e implementação de arquiteturas de software baseadas em componentes. Este modelo de implementação de componentes oferece diretrizes para materializar elementos arquiteturais em código-fonte [77]. É importante notar que num primeiro momento temos somente o '*esqueleto*' da arquitetura da família de técnicas de tolerância a falhas implementada. Nesta arquitetura, as conexões entre os componentes arquiteturais são exercitadas através de suas interfaces providas e requeridas, contudo é preciso implementar o comportamento dos diversos métodos (*e.g.* reutilizando implementações das soluções existentes na literatura). Cabe ressalar que a partir da análise da literatura relacionada [36, 37, 38, 33, 58, 39, 34, 40, 41, 59], observamos que há uma lacuna no que tange a existência de soluções flexíveis baseadas em diversidade de projetos para composições confiáveis de serviços. Portanto, a concepção de uma família de técnicas baseadas em diversidade de projetos para composições de serviços tolerantes a falhas é também uma contribuição desta tese.

Além disso, propomos uma infraestrutura dirigida por modelos para apoiar o desenvolvimento semi-automatizado de arquiteturas de linhas de produtos de software em geral. Utilizamos a infraestrutura proposta para especificar e validar o modelo de características e o modelo arquitetural da família de técnicas de tolerância a falhas. A infraestrutura dirigida por modelos encompassa um conjunto de ferramentas e métodos para garantir que modelos de mais alto nível de abstração sejam mapeados para modelos de mais baixo nível de abstração, até a implementação da arquitetura da linha de produtos, de forma semi-automatizada, coordenada e consistente. As soluções existentes para a engenharia de linhas de produtos dirigida por modelos apresentam algumas lacunas, como por exemplo, *(i)* não definem de forma clara a sequência de modelos a serem produzidos e as transformações executadas entre os diferentes modelos [78, 79]; e *(ii)* não apresentam um suporte ferramental para apoiar a automatização das transformações entre os modelos [80, 81]. Portanto, uma infraestrutura dirigida por modelos e semi-automatizada para apoiar o desenvolvimento sistemático de arquiteturas de linhas de produtos de software é mais uma contribuição desta tese.

O projeto da família de técnicas de tolerância a falhas para serviços compostos confiáveis e a proposta da infraestrutura dirigida por modelos para desenvolver arquiteturas de linhas de produtos de software, resultaram na seguinte publicação.

- **Nascimento A.S.**; Rubira, C. M. F.; Burrows, R.; Castor, F. *A systematic review of design diversity-based solutions for fault-tolerant SOAs.* 17th International Conference on Evaluation and Assessment in Software Engineering, 2013, Porto de Galinhas, PE, Brazil.

1.3.3 Uma Infraestrutura Baseada em Linhas de Produtos de Software (Dinâmicas)

Questão de Pesquisa 5 (QP5) Como apoiar de forma adequada uma infraestrutura autoadaptiva que apoia a instanciação de técnicas diferentes de tolerância a falhas em resposta a mudanças ocorridas no contexto?

A fim de respondermos a QP5, propomos uma infraestrutura baseada em linhas de produtos de software (dinâmicas), chamada ArCMAPE, para composições de serviços tolerantes a falhas e autoadaptativos. ArCMAPE é composta por uma família de técnicas de tolerância a falhas e apoia a instanciação da técnica mais adaptada em tempo de execução conforme mudanças nos requisitos de clientes e valores dos atributos de qualidade de serviços (QoS). A técnica mais adaptada é aquela que *(i)* satisfaz regras pré-estabelecidas (*e.g.* serviços alternativos que executam pagamentos de cartão de créditos devem ser executados de forma sequencial e seus resultados analisados por testes de aceitação para evitar cobranças duplicadas); e *(ii)* maximiza uma função objetivo. Esta função objetivo leva em consideração tanto os valores correntes de QoS quanto prioridades pré-estabelecidas por diversos clientes (*e.g.* para alguns, maximizar tempo de resposta, para outros, a disponibilidade).

ArCMAPE foi projetada numa arquitetura reflexiva a fim de apoiar a separação de interesses. O nível meta baseia-se no *loop* autonômico [82] para discorrer sobre o comportamento dinâmico da estratégia de tolerância a falhas e o contexto corrente. Para tomadas de decisão, o nível meta analisa e manipula modelos da linha de produtos (*i.e.* o modelo de características e a arquitetura da linha, referentes a família de técnicas de tolerância a falhas), e modelos contendo outras informações relevantes do contexto (*e.g.* valores de QoS e requisitos dos clientes). Estes modelos são mantidos dinamicamente e constantemente refletem a configuração da técnica de tolerância a falhas em execução. Planos de adaptação são gerados de forma automática, reduzindo a complexidade referente à especificação da lógica de adaptação dinâmica do mecanismo de tolerância a falhas. O nível base é composto por componentes de software implementando a arquitetura da linha de produtos. Mais especificamente, no nível base, componentes de software implementando as características mandatórias são fornecidos, enquanto que componentes de software implementando características variáveis podem ser facilmente inseridos em pontos de extensão, no caso, pontos de variação, definidos na arquitetura. A técnica de tolerância a falhas mais adaptada é instanciada automaticamente em tempo de execução a partir da configuração dinâmica de produtos, isto é, mediante gerenciamento dinâmico de variabilidades de software.

Cabe ressaltar que os modelos mantidos em tempo de execução podem ser especificados e validados a partir da infraestrutura proposta dirigida por modelos para desenvolvimento de arquiteturas de linhas de produtos de software. Os serviços aternativos podem ser escolhidos mediante a infraestrutura proposta para mensurar diversidade de serviços. As demais características alternativas podem ser realizadas a partir da análise e reúso das escolhas de projetos apoiadas por soluções existentes para composições de serviços confiáveis, descritas na revisão sistemática (*e.g.* diferentes esquemas para executar serviços alternativos e diferentes juízes para julgar seus resultados). Resultados de estudos empíricos sugerem que ArCMAPE é eficiente para apoiar composições de serviços confiáveis e autoadaptivas e não insere um *overhead* excessivo para apoiar o gerenciamento dinâmico de variabilidades de software.

A partir da análise da literatura relacionada, obsevamos que composições de serviços autoadaptáveis e tolerantes a falhas não apoiam soluções para os desafios de pesquisa (Seção 1.2.3) relacionados aos mecanismos de tolerância a falhas autoadaptativos [34, 41, 58, 33]. A proposta de uma infraestrutura facilmente extensível e que apoia o gerenciamento dinâmico de variabilidades de software a fim de instanciar em tempo de execução técnicas de tolerância a falhas apropriadas para diferentes contextos é uma contribuição desta tese. ArCMAPE resultou nos seguintes artigos:

- **Nascimento A.S.**; Rubira, C. M. F.; Castor, F. *A Comprehensive Solution for Fault-Tolerant Composite Services*, 2013. (a ser submetido numa revista)
- **Nascimento A.S.**; Rubira, C. M. F.; Castor, F. *ArCMAPE: A Software Product Line Infrastructure to Support Fault-Tolerant Composite Services.* 15th IEEE International Symposium on High Assurance Systems Engineering, 2014, Miami, Florida, USA.
- **Nascimento A.S.**; Rubira, C. M. F.; Castor, F. *Using CVL to Support Self-Adaptation of Fault-Tolerant Service Compositions.* 7th IEEE International Conference on Self-Adaptive and Self-Organizing Systems, 2013, Philadelphia, USA. (resumo expandido)
- **Nascimento A.S.**; Castor, F.; Rubira, C. M. F. *Identifying Modelling Dimensions of a Self-Adaptive Framework for Fault-Tolerant SOAs - An Experience Report.* 1st Workshop on Dependability in Adaptive and Self-Managing Systems co-located LADC, 2013, Rio de Janeiro.
- **Nascimento A.S.**; Rubira, C. M. F.; Lee, J. *An SPL approach for adaptive fault tolerance in SOA.* 1st International Workshop on Services, Clouds and Alternative Design Strategies for Variant-Rich Software Systems co-located with SPLC, 2011, Munich, Germany.
- **Nascimento A.S.**; Rubira, C. M. F. *Tolerância a Falhas em Linhas de Produto de Software Baseadas em Serviços Web.* V Workshop de Teses, Dissertações e Trabalhos de Iniciação Científica em Andamento IC-UNICAMP, 2009, Campinas, SP, Brazil. (resumo)
- **Nascimento A.S.**; Rubira, C. M. F. *Especificação de uma abordagem sistemática para gerenciar e implementar Linhas de Produtos Dinâmicas e baseadas em serviços Web.* V Workshop de Teses, Dissertações e Trabalhos de Iniciação Científica em Andamento IC-UNICAMP, 2009, Campinas, SP, Brazil. (resumo)

1.4 Estrutura da Tese

Esta tese foi escrita baseada em uma coleção de artigos científicos *publicados em* ou *submetidos a* conferências internacionais, simpósios nacionais e internacionais, e periódicos internacionais. Por esta razão, com exceção dos capítulos de introdução e conclusão, foram preservados os textos originais dos artigos em inglês. Cada capítulo é autocontido, apresentando problemas e motivações, solução proposta e sua avaliação, e trabalhos relacionados. Com o propósito de evitar textos redundantes, especialmente sobre motivação e fundamentos teóricos, os conteúdos dos artigos originais foram adaptados com o intuito de serem inseridos sem os seus fundamentos teóricos, introduções e suas conclusões. Além disso, nos Capítulos 3 a 6, foi acrescentada uma seção chamada *Overview*, que apresenta uma visão geral do capítulo com base nos artigos originais associados ao capítulo em questão. Nesses capítulos, também foi adicionada uma seção chamada *Summary*, que apresenta considerações relevantes acerca do capítulo em questão. No início de cada capítulo, adicionamos um prólogo indicando a referência da qual o respectivo conteúdo foi extraído.

O restante desta tese está organizado como segue: O Capítulo 2 apresenta alguns fundamentos teóricos sobre os conceitos contidos nesta tese. O Capítulo 3 descreve a infraestrutura proposta para avaliar quão eficientes um dado conjunto de serviços alternativos é para tolerar falhas de software. Nesse capítulo, também apresentamos estudos empíricos realizados, mediante o uso da infraestrutura proposta, com a finalidade de melhor entendermos as implicações de se utilizar diversidade de projetos para tolerar falhas de software de serviços. Portanto, no Capítulo 3, apresentamos soluções para o primeiro subproblema que esta tese visa resolver. O Capítulo 4 apresenta a revisão sistemática que conduzimos com o propósito de analisar e classificar as soluções existentes baseadas em diversidade de projetos para composições de serviços tolerantes a falhas. O Capítulo 5 descreve uma infraestrutura dirigida a modelos e semi-automatizada para desenvolver arquiteturas de linhas de produtos de software diversas. Neste capítulo, é apresentado como a infraestrutura foi utilizada para propormos uma família de técnicas de tolerância a falhas de software. Portanto, nos Capítulos 4 e 5, apresentamos soluções para o segundo subproblema que esta tese visa resolver. O Capítulo 6 descreve a infraestrutura baseada em linhas de produtos de software dinâmicas para apoiar composições de serviços confiáveis e autoadaptáveis. Portanto, no capítulo 6, apresentamos soluções para o terceiro subproblema que esta tese visa resolver. Finalmente, no Capítulo 7, apresentamos as conclusões e identificamos direções de pesquisas futuras.

Capítulo 2

Fundamentos Teóricos de Reúso de Software, Tolerância a Falhas, e Sistemas de Software Autoadaptativos

Este capítulo apresenta alguns conceitos de fudamentos teóricos sobre desenvolvimento baseado em componentes, arquiteturas orientadas a serviços, linhas de produtos de software (dinâmicas), técnicas de tolerância a falhas de software baseadas em diversidade de projeto, e sistemas autoadaptativos. O conteúdo deste capítulo foi retirado de três publicações: um artigo publicado no *17th International Conference on Evaluation and Assessment in Software Engineering - EASE '13*, que apresenta a revisão sistemática de soluções baseadas em diversidade de software para composições de serviços tolerantes a falhas; um artigo publicado no *7th Brazilian Symposium on Software Components - SBCARS '13*, que apresenta uma infraestrutura dirigida por modelos para apoiar o desenvolvimento de arquiteturas de linhas de produtos de software; e um artigo publicado no *1st Workshop on Dependability in Adaptive and Self-Managing Systems (WDAS) em conjunto com o 6th Latin-American Symposium on Dependable Computing - LADC'13*, que descreve de forma sistemática o comportamento dinâmico da infraestrutura proposta para apoiar composições de serviços tolerantes a falhas e autoadaptativas. Como o conteúdo do capítulo foi extraído na íntegra desses artigos, foi preservado o idiomal original.

2.1 Software Reuse

Currently, there is an increasing need to address software evolvability in order to gradually cope with different stakeholders' needs [22, 10]. At the same time, the software system's desired time-to-market is ever decreasing. This reality demands on software systems' capability of rapid modification and enhancement to achieve cost-effective software evolution [22, 10, 27]. To face these needs, advanced software paradigms have emerged. Service-oriented computing, component-based development and software product line engeneering are three promising cases in particular as they increase software reuse at the architectural level of design [10, 16, 27].

In this section, we briefly describe service-oriented architectures; COSMOS*, a component implementation model; and (dynamic) software product lines.

2.1.1 Service-Oriented Architecture (SOA)

SOA is described as a component-based model which interrelates different functional units (or services) by means of well-defined interfaces that should be neutral, platform- and language-independent [16]. Services running over heterogeneous systems may then interact and be used as building blocks for composite services [16]. Due of the specifics of the SOA scenarios, to aggregate multiple services into a single composite services it is necessary to support a series of functional and quality requirements, for example, interoperability capabilities; autonomic composition of services; QoS-aware service compositions; and business-driven automated compositions, as discussed in the literature (e.g. [83, 16]).

SOA is often found in web service applications. *Web Services*, called as services for simplicity, often rely on XML-based standards such as SOAP (*Simple Object Access Protocol*) and WSDL (*Web Services Description Language*) to exchange information with other applications over the Internet. Services can be read-only, which means that, given a request, these services provide access to data that may be read but not changed or deleted [15]. They can also be classified as stateless or stateful. Stateless services support no mechanism within themselves to handle state across requests [84]. Stateful services keep state information across requests [84]. To evaluate our solution, in our empirical studies we reuse third-party, stateless, read-only, SOAP/WSDL-based Web Services. It was a decision that stemmed from the availability of cost-free services and the possibility of comparing their results. It is important to mention that, recently, Representational State Transfer (REST) has gained widespread acceptance across the Web as a simpler alternative to SOAP- and WSDL-based Web services [85].

2.1.2 Cosmos* Implementation Model

The COSMOS* model [77] is a generic and platform-independent implementation model, which uses object-oriented structures, such as interfaces, classes and packages, to implement component-based software architectures. The main advantages of COSMOS*, when compared with other component models such as Corba Component Model (CCM), Enterprise Java Beans, and .NET, is threefold. COSMOS* provides traceability between the software architecture and the respective source code by explicitly representing architectural units, such as components, connectors and configuration. Second, COSMOS* is based on a set of design patterns, thus, it is considered a platform-independent model. Thirdly, the set of design patterns employed by COSMOS* can be automatically translated to source code. In particular, our solution employs Bellatrix [86], an Eclipse Plug-in that translates graphically specified software architectures (*using Unified Modelling Language - UML*) to Java source code in COSMOS*. Bellatrix allows the creation of a 'skeletal' system in which the communication paths are exercised but which at first has a minimal functionality. This 'skeletal' system can then be used to implement the system incrementally, easing the integration and testing efforts [12].

To address different perspectives of component-based systems, COSMOS* defines five sub-models: *(i)* a specification model specifies the components; *(ii)* an implementation model explicitly separates the provided and required interfaces from the implementation; *(iii)* a connector model specifies the link between components using connectors, thus enabling two or more components to be connected in a configuration; *(in)* a composite components model specifies high-granularity components, which are composed by other COSMOS* components; and *(v)* a system model defines a software component which can be executed, thus encapsulating the necessary dependencies. Figure 2.1 (a) shows an architectural view of a COSMOS* component called *CompA* and Figure 2.1 (b) shows the detailed design of the same COSMOS* component [77]. We have used UML notation in this work to model architecture and detailed design and, in Figure2.1 (a) , we have omitted operations and attributes for the sake of clarity.

The component is internally divided into specification (*CompA.specpackage*) and implementation (*CompA.implpackage*). The specification is the external view of the component, which is also sub-divided into two parts: one that specifies the provided services (*CompA.spec.prov* package) and the other makes dependencies explicit (*CompA.spec.req* package). For instance, $IManager$ and IA are provided interfaces and IB is a required interface. The COSMOS* implementation model also defines classes to support component instantiation and to implement provided interfaces. The *CompA.impl* package has three mandatory classes: (i) a $ComponentFactory$ class, responsible for instantiating the component; (ii) a $Facade$ class that realizes provided interfaces, following the Facade design pattern; and (iii) a $Manager$ class that realizes $IManager$ interface and provides meta-information about the component. *ClassA* and *ClassB* are examples of auxiliary classes.

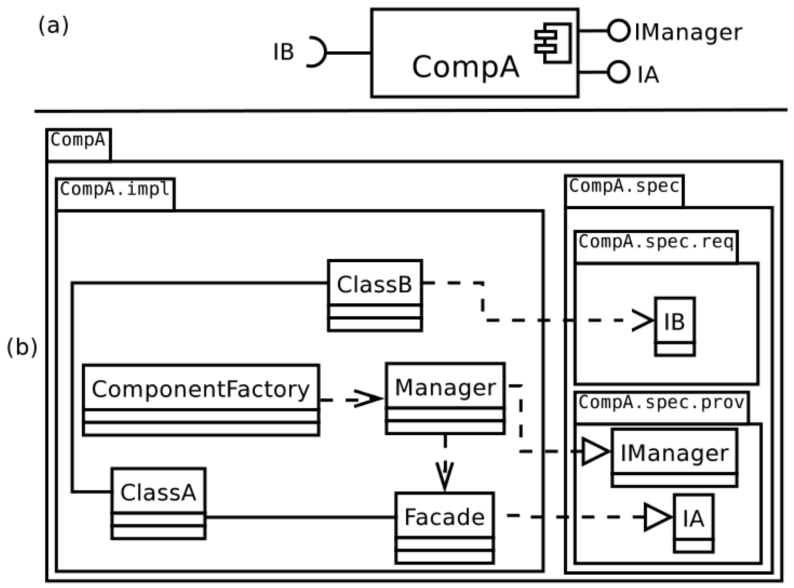

Figure 2.1: (a) An architectural view and a detailed design of a COSMOS* component; (b) A detailed design of the COSMOS* component

2.1.3 (Dynamic) Software Product Lines

Software product line (SPL) is a systematic software reuse approach that promotes the generation of specific products from a set of core assets for a given domain, exploiting the commonalities and variabilities among these products [23, 87]. *Software variability* is defined as the capacity that a software system or artefact has to be modified for use in a particular context at some point in its life-cycle [23]. Although traditional SPL engineering recognizes late variability, it typically selects, deploys, and acts on their features before delivery of the software [18]. In contrast, Dynamic Software Product Lines (DSPLs) extend SPLs to support late variability [18]. In the following, we briefly describe some background on software product line engeneering.

Use Case Modelling for Software Product Lines

To specify the functional requirements of a SPL, it is important to capture the requirements that are common to all members of the family as well as the variable ones. Gomaa [88] identifies different kinds of use cases: kernel use cases, which are needed by all members of the product line; optional use cases, which are needed by only some members of the product line; and alternative use cases, which are usually mutually exclusive and where different versions of the use case are required by different members of the product line.

According to Gomaa [88], product line use cases can be described by its *(i)* name; *(ii)* reuse category (whether the target use case is kernel, optional, or alternative); *(iii)* summary description; *(iv)* actors - the only external entities that interact with the system; *(v)* dependency - this optional section describes whether the use case includes or extends another use case; *(vi)* preconditions - one or more conditions that must be true at the start of the use case; *(vii)* description - a narrative description of the main sequence of the use case; *(viii)* alternatives - a narrative description of alternative branches off the main sequence; *(ix)* variation points; and *(x)* postcondition - the condition that is always true at the end of the use case if the main sequence has been followed [88]. Regarding the item *(ix)*, variation points identify places in the use case description where different functionality can be introduced for the different members of the product line, i.e., a variation point is a location in a use case where a change can take place [89]. We refer to Gomaa [88] for further details on how to identify and specify use cases.

Feature Models

A feature is a system property that is relevant to some stakeholder [25]. A *feature model* represents the commonalities among all products of a product line as mandatory features, while variabilities among products are represented as variable features. Variable features extensively fall into three categories: *(i)* optional, which may or may not be present in a product; *(ii)* alternative, which indicates a set of features, from which only one must be present in a product; and *(iii)* multiple features, which represents a set of features, from which at least one must be present in a product [5, 45]. The feature model can also represent additional constraints between features. Some examples of constraints are mutual dependency, when a feature requires another, and mutual exclusion, when a feature excludes another [5, 45]. Furthermore, use cases should be mapped to the features on the basis of their reuse properties [88].

Product Line Architectures (PLAs)

One of the main artifacts of a SPL is the product line architecture (PLA) [90], which explicitly represents the commonalities and variabilities of architectural elements and their configurations [27]. The commonalities are reused in different products, while the variabilities are resolved through design decisions related to the choices at the PLA [27]. In PLAs, software variability can be reached by delaying certain architectural design decisions, which are described through variation points. A variation point is the place at the software architecture where a design decision can be made. Variants are the different possibilities that exist to satisfy a variation point. Binding the variant is the selection of some variant supported by a variation point. A well-known technique to efficiently and rapidly derive products from a set of reusable assets it the combination of SPL and Component-based Development (CBD) [90]. Furthermore, a component framework [27, 2] can be used to guarantee that components that need to be configured are successfully coordinated. In this case, variation points are represented by locations in the framework where plug-in components may be added and variants are realized by specific choices of the plug-in [27]. The explicit integration of plug-in mechanisms in the PLAs reduces the effort for the composition of the final products [2]. We emphasize that a PLA is a key artefact to achieve a controlled evolution and it should be consistent with the feature model.

Feature-Architecture Mapping (FArM)

By iteratively refining the initial feature model, the FArM method enables the construction of a transformed feature model containing exclusively functional features, whose business logic can be implemented into architectural components [2]. It is largely accepted that FArM improves maintainability and flexibility of PLAs [2]. The FArM transformation flow is shown in Figure 2.2 and is briefly described in the following.

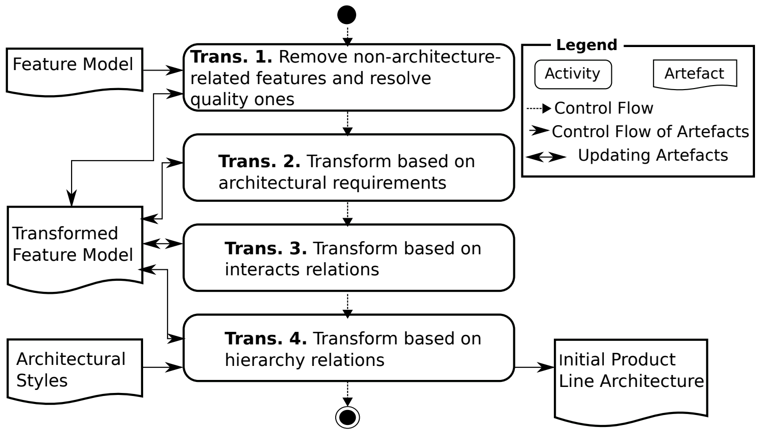
Feature Model
Trans. 1. Remove non-architecture-related features and resolve quality ones
Legend
Activity
Artefact
Control Flow
Control Flow of Artefacts
Updating Artefacts
Trans. 2. Transform based on architectural requirements
Transformed Feature Model
Trans. 3. Transform based on interacts relations
Trans. 4. Transform based on hierarchy relations
Architectural Styles
Initial Product Line Architecture

Figure 2.2: The FArM method [2]

Firstly, non-architecture-related features should be removed from the initial FM. Quality features, in turn, can be resolved by integrating them into existing functional features by enhancing their specification or by adding new features providing functional solutions that fulfil the quality features and their specifications. Secondly, architectural requirements (AR) are identified (*e.g.* authentication and interoperability). AR can be handled through direct resolution, integration in existing functional features and addition of new functional features. The third interacts relation is used to model the communication of features. A valid FArM interacts relation connects two features where one feature uses the other feature's functionality; and the correct operation of one feature either alters or contradicts the behaviour of the other feature. Once all interacts relations between features are identified, features are then transformed based on the type and the number of interacts relations.

The last transformation is based on the analysis of hierarchy relations between super-features and their sub-features. The sub-feature can specialize, be part of or present alternatives to their super-feature. At this transformation, invalid hierarchy relations are removed and their features remain without any sub-features. Whenever it is necessary, new hierarchy relations may be created [2]. In the last FArM iterations the system architects commit to certain architecture and they can also employ a specific architectural style by adapting components to a particular style (*e.g.* a plug-in or a layered architecture).

Decision Models

For generating a specific product, it is necessary to instantiate the PLA considering the design choices associated with particular variants [90, 27]. To support these choices, a decision model should be constructed in order to relate the possible choices of variants in the feature model, to high-level decisions of the software architecture in the form of variation points [68, 27]. In other words, the decision model documents the decisions that need to be made at the context of a PLA, and which are related to the variants of the feature model, thus providing support for tuning the software architecture according to the requirements of the system [90]. The decision model provides a mapping from the requirements (feature model) to the design (PLA). Through this artefact, the client can directly participate of the decision process of a specific product, since it is possible to abstract away about technical details.

Common Variability Language (CVL)

CVL is a domain-independent language for specifying and resolving variability being considered for standardization at Object Management Group [3] (OMG). CVL allows the specification of variability over models of any language defined using a MOF-based meta-model, including Unified Modelling Language (UML) and Domain Specific Languages (DSLs) [3]. As illustrated in Figure 2.3, in CVL approach we have three models:

- *Base Model:* a model described in a DSL.
- *Variability Model:* the model that defines variability on the base model.
- *Resolution Model:* the model that defines how to resolve the variability model to create a new model in the base DSL.

With variability model and resolution model properly defined, it is possible to run CVL model-to-model transformation to generate new resolved models, or product models, in the

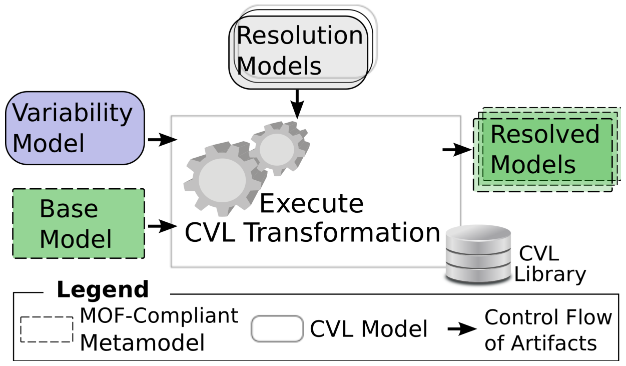

Figure 2.3: CVL transformation (or materialisation) [3]

base language [3]. This process is also called *materialisation*. In general terms, a resolution model specifies a feature configuration (*i.e.* a set of selected features), and a base model specifies a PLA model. The core concepts of the CVL materialisation are *substitutions*. A *substitution* replaces *base model elements* defined as a *placement* by *base model elements* defined as a *replacement*. CVL creates a product model by copying the base model and performing the selected substitutions. Under some circunstances, some replacing model elements, not already in the base model, may have to be added. These additional model elements can be found at the CVL library.

More specifically, in CVL, a variability model consists of three main parts [3]:

- *Variation points*: Define the points of the base model that are variable and can be modified during the CVL execution. For instance, some of the variation points supported by CVL are the existence of elements of the base model or substitutions of elements of the base model.

- *Variability Specification Tree (VSpec tree)*: Tree structures whose elements represent choices bound to variation points. These choices are resolved by a resolution model and propagated to variation points and the base model, generating the resolved model without variability. In general terms, VSpec trees are similar to feature models and deciding choices is similar to selecting features [3].

- *OCL Constraints*: CVL supports the definition of OCL constraints among elements of a VSpec tree, providing a highly flexible mechanism for delimiting the bounds of variability, being able to discard invalid configurations.

It is important to notice that the CVL variability model defines a decision model, because CVL specifies *(a)* variabilities and commonalities in domain requirements; and *(b)* how requirements choices map to choices of decisions that need to be made at the context of a PLA [91]. A product model is in fact a *decision model instance*, in which all decisions are resolved. A product model can be further used to instantiate a specific product from SPL artefacts [91]. The base, variability, and resolution models are described in details in Section 5.3.1, in the context of the SPL proposed to support software fault tolerance techniques.

2.2 Fault Tolerance

A fault is the identified or hypothesized cause of an error [92, 93]. An error is part of the system state that is liable to lead to a failure [92, 93]. A failure, in turn, occurs when the service delivered by the system deviates from the specified service [1, 93]. So, with software fault tolerance, we want to prevent failures by tolerating faults whose occurrences are known when errors are detected [1]. When designing fault tolerance, a first prerequisite is to specify, by means of fault models, the faults that should be tolerated [44]. The next step is to enrich the system under consideration with components or concepts that provide protection against faults from the fault models [44].

2.2.1 Redundancy

A key supporting concept for fault tolerance is *redundancy*, that is, additional resources that would not be required if fault tolerance was not being implemented [1]. Redundancy can take several forms, for example, hardware and software. *Software redundancy* includes the additional programs, modules, or objects used in the system to support fault tolerance [46]. Redundant or diverse software can reside on the redundant hardware to tolerate both hardware and software faults. *Hardware redundancy* includes replicated and supplementary hardware added to the system to support fault tolerance [46].

For instance, we are interested in software faults. According to Pullum [1], '*software faults may be traced to incorrect requirements (where the software matches the requirements, but the behavior specified in the requirements is not appropriate) or to the implementation (software design and coding) not satisfying the requirements*'. Software faults are also called design faults or bugs [1]. To protect against software faults, we cannot simply duplicate identical software units, as is typically done for hardware units to tolerate hardware faults, because doing so will simply duplicate the problem [1]. A solution to the problem of replicating design and implementation faults is to introduce diversity into the software replicas [94]. In the following, we present the basic design diversity concept.

2.2.2 Basic Design Diversity Concept

Design diversity is the provision of functionally equivalent services through separate design and implementations [46, 1, 42]. When diversity is used, the redundant software components are termed variants, versions, or alternates[1] [46]. Figure 2.4 illustrates the basic design diversity concept. Inputs are distributed to multiple software components, each with equivalent functionality yet different designs, the alternates. The alternates execute their operations and produce their results, from which a single correct or acceptable result must be derived, if any [1]. The mechanism responsible for this task is called an *adjudicator*. The input to the adjudicator function consisting of at least the alternate outputs is called *syndrome* [95]. To execute the alternates, it is necessary to provide all of them with exactly the same experience of the system state when their respective executions start in order to ensure consistency of input data [94, 46].

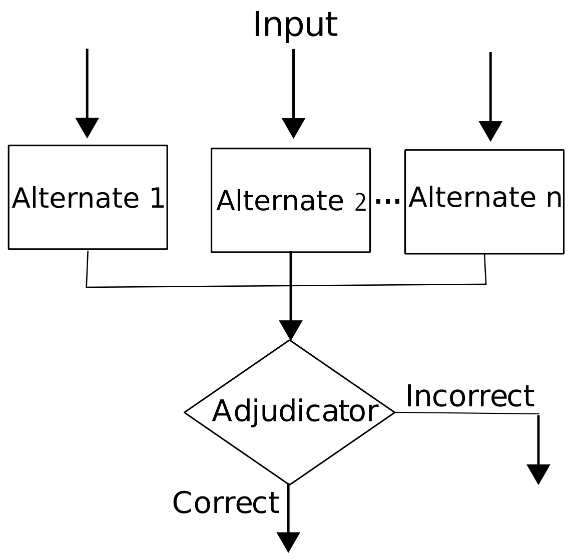

Figure 2.4: Basic Design Diversity [1]

If alternates fail on the same input case, then a *coincident failure* [96] is said to have occurred. The goals of increasing diversity in alternates are to decrease the probability of coincident failures and to increase the probability that the alternates fail on disjoint subsets of the input space, when they do fail [1]. The achievement of these goals increases the system's reliability and increases the ability to detect failures when they occur [96, 97]. In other words, the more diverse alternates are, the less likely coincident failures are to occur and the more failures of alternates is likely to be detectable [1, 42, 49, 96]. In fact,

[1]Although Laprie et al. [46] justifies the usage of the term 'variants' to identify the multiple functionally equivalent software components, we use 'alternates' instead to avoid conflict with the notion of variant from SPL.

adjudicators might fail when coincident failures occur [1].

Adjudicators

Adjudicators generally comes in two flavours, voters and Acceptance Tests (ATs). We refer to Pullum [1] for a description about the various types of adjudicators and their operations (pages 269-324) - *e.g.* exact majority, consensus, formal consensus, formal majority, median, mean, weighted, and dynamic voters; acceptance tests based on satisfaction of requirements, accounting tests, computer run-time acceptance tests and reasonableness acceptance tests. Ideally, the adjudication mechanism should be free from errors itself since the adjudicator is not replicated and typically does not have an alternate [1]. Adjudicators encompass three possible states, or *status*, as illustrated in Figure 2.5 [1] and briefly described in the following. The correct result, if any, and the staus indicator are returned to the subsequent program modules.

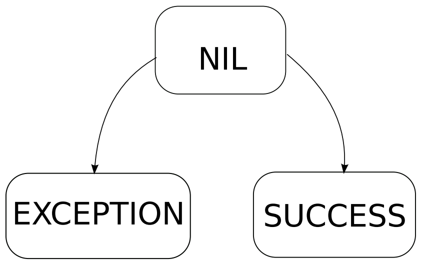

Figure 2.5: Status Indicating possible *State* of adjudicators [1]

Status = NIL *Status* is initialized to this value. If the *Status* returned from the adjudicator is NIL then an error ocurred during adjudication. That is, this status indicates that the adjudicator has not completed examining the alternate results [1]. Ignore the returned 'correct' output, if any.
Status = EXCEPTION The adjudicator was not able to find the presumed correct output given the alternate results, although it did complet processing [1].
Status = SUCCESS The adjudicator did complete processing and found the assumed correct, adjudicated result [1].

Voter Adjudicators

In particular, for the empirical studies on service diversity, we are interested in a three-alternate voting system since this is the minimum number of alternates that allows a service composition to tolerate faults of one of its services. In a three-alternate system, three kind of results are possible: *(i) correct result*- executions of a majority of the alternates in the triplet result in identical or similar correct results from which a single correct result is adjudicated [98]; *(ii) failure*- a majority of alternates fail on the same input case (*i.e.* coincident failures) resulting in similar or identical incorrect results, therefore the voter returns either an incorrect result as the presumably *'correct one'*, or no output [1, 98]; *(iii) failure exception*- three results that are not within an acceptable range (*i.e.* they are not similar) are returned by execution of alternates, therefore, a voter is not able to decide whether results were successful or not [98].

2.2.3 Structuring Software Redundancy

As discussed by Laprie et al. and Pullum [46, 1], software redundancy can be structured in many forms, as illustrated in Figure 2.6. The final fault-tolerant system can have (a) all replicas and adjudicators on a single hardware component, (b) replicas on multiple hardware components, or (c) the adjudicator on a separate hardware component. Moreover, the software that is replicated can range from an entire program to a program segment as shown in Figure 2.6(d). According to Pullum [1], the choices to be made in structuring software redundancy are based on available resources (e.g. the underlying hardware) and the specific application.

2.2.4 Error Recovery

The fault tolerance process is that set of activities whose goal is to remove errors and their effects from the computational state before a failure occurs [1]. In particular, error recovery is the process in which the erroneous state is substituted with an error-free state [1]. Error recovery is performed using either backward recovery or forward recovery. On one hand, *backward recovery* attempts to return the system to a correct or error-free state by restoring or rolling back the system to a previously saved state, which is assumed to be error-free [1]. On the other hand, *forward recovery* attempts to return the system to a correct or error-free state by finding a new state from which the system can continue operation. Compared with backward error recovery, the forward recovery is usually more efficient in terms of the overhead (*e.g.* time and memory) it requires [1].

2.2.5 Design Diversity Software Fault Tolerance Techniques

This section covers the original and basic design diverse software fault tolerance techniques - recovery blocks (RcB) and N-version programming (NVP) [1]. Each one of these techniques can be realised by a reference architecture, which provides a proven template solution for a particular domain [45, 99].

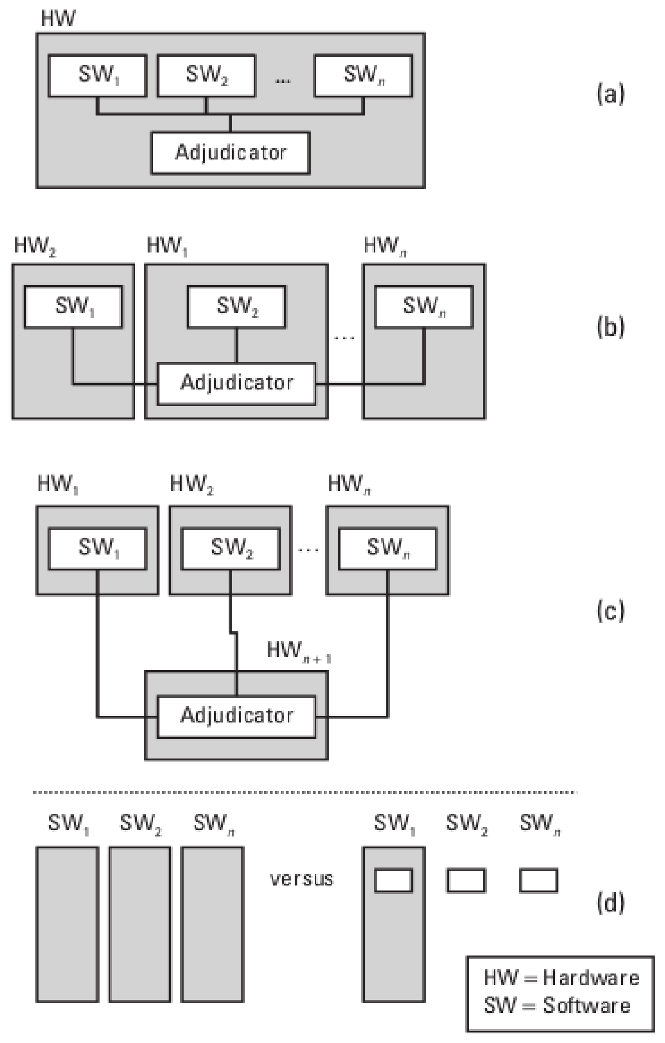

Figure 2.6: Various Views of Redundant Software (figure extracted from [1]-page 20)

Recovery Block

The recovery block (RcB) technique was introduced by Horning et al. [47], with early implementation by Randell [76]. RcB combines the basics of the checkpoint and restart approach with multiple versions of a software component such that a different version is tried after an error is detected [46, 45]. Checkpoints are created before a version is executed for providing an operational state for recovering after a version fails. A reference architecture for tolerating a single fault using the RB technique was proposed by Brito et al. [45], and is shown in Figure 2.7. The *Switch* is responsible for choosing a proper *Alternate* to execute the service, in case an error is detected by the *AcceptanceTest* (*i.e.* any one of the various acceptance tests [1]). The data integrity between two executions is guaranteed by the *Checkpoint* element.

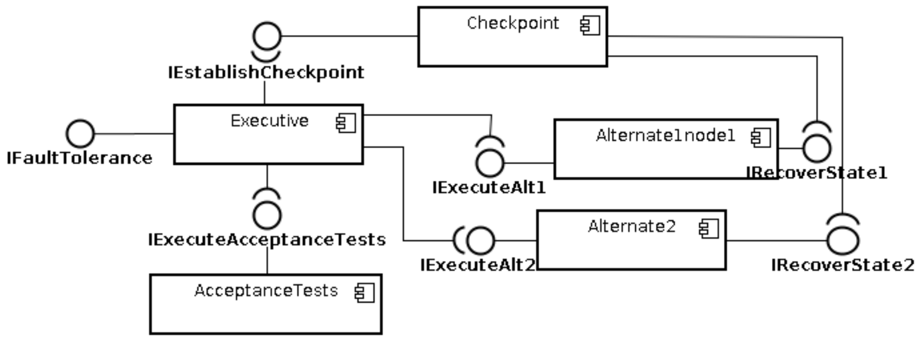

Figure 2.7: Reference Architectures For Recovery Block Techniques

Wilfredo [100] claims that in RcB, the execution of the multiple versions can be also in parallel, depending on the available processing capability and performance requirements. Under this circunstance, it is not necessary to employ rollback recovery since parallel execution provides a valid operational starting point for the multiple versions through a synchronization regime [1, 100]. RcB is typically uniprocessor implemented, with the executive and all components residing on a single hardware unit [1] (Figure 2.6 (a)). However, when employing RcB to support fault-tolerant service compositions, we can have n alternate services residing on n hardware units and the executive residing on a different hardware unit. Therefore, we can have $n+1$ hardware units (Figure 2.6 (c)). Under this circunstance, communications between software components are done through remote methods calls or method invocations.

N-Version Programming

N-version programming (NVP) was suggested by Elmendorf [48] and developed by Aviziens and Chen [101, 102]. NVP is a multi-version technique in which all the versions are designed to satisfy the same specification. The outcome is obtained using compensation by comparing the outputs of the versions through voting [101, 102] (*i.e.* any one of the various voters [1]). Figure 2.8 presents a reference architecture proposed by Brito et al. [45] for tolerating a single fault using the NVP technique. In this architecture, the *Switch* connector is responsible to receive the results of all the three versions of components and then judge if there is a reliable result based on voting.

Wilfredo claims [100] that actual execution of the multiple versions could be also sequential. Under this circunstance, it may be required the usage of checkpoints to reload the state before an alternate version is executed [100]. According to Pullum [1], the NVP is typically multiprocessor implemented with components residing on n hardware units and the executive together with the voter residing on one of the processors [1]. However, when employing NVP to support fault-tolerant service compositions, the executive together with the voter can reside on a separate hardware unit, therefore, we can have $n + 1$ hardware units (Figure 2.6 (c)). Under this circunstance, communications between software components are done through remote methods calls or method invocations.

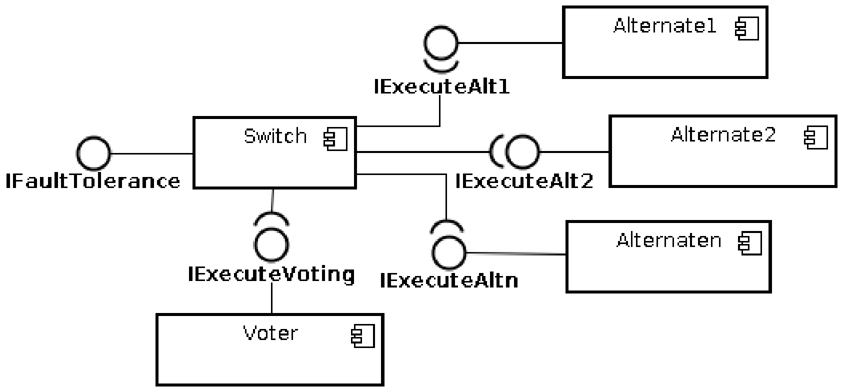

Figure 2.8: Reference Architectures For N-Version Programming

2.2.6 Reliability

Reliability is defined as *'the ability of a system or component to perform its required functions under stated conditions for a specified period of time'* [15]. When the execution time is not readily available, approximations such as the number of test cases executed may be used [42]. In this sense, the successful execution rate can be adopted to estimate the reliability of a system. The value of the success rate $0 \leq q_{rate}(s) \leq 1$ of a system s is computed from data of past invocations [103] $q_{rate}(s) = N_c(s)/k(s)$, where $N_c(s)$ is the number of successful results provided by a system s within the maximum expected time frame, and $K(s)$ is the total number of invocations of the system.

Moreover, by means of successful execution rate, we are able to estimate the difference in reliability of a fault-tolerant architecture based on design diversity, or FT-architecture, and single non-fault-tolerant alternates, in order to find out evidence on whether the first supports a reliability improvement over the latter. A positive difference in reliability indicates an increase in reliability [42], that is, the FT-architecture tolerated faults of its alternates, which rarely fail on the same input cases [1]. On the other hand, a negative difference indicates a reliability decrease [42]. The introduction of design diversity might lead to occurrence of coincident failures, which might defeat most adjudicators [1, 42, 49].

2.3 Self-Adaptive Systems and the Autonomic Control Loop

Software-based systems today increasingly operate in changing environments with variable user needs. Consequently, systems are increasingly expected to dynamically self-adapt to accommodate such changes [82, 4, 66]. The autonomic control loop provides the generic mechanism for self-adaptation. This loop, shown in Figure 2.9, typically involves four key activities: collect, analyze, decide and act [4].

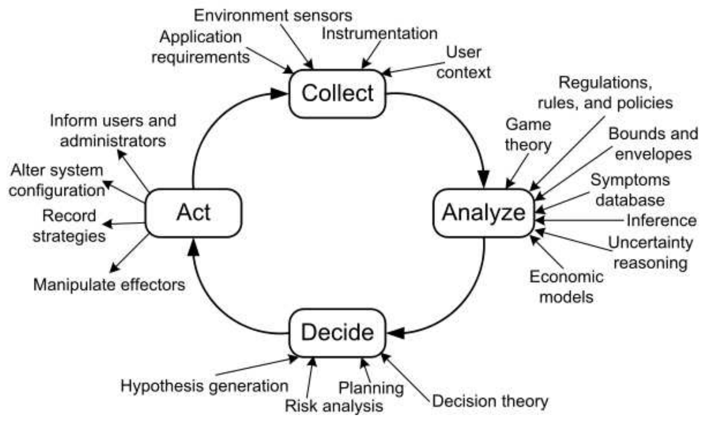

Figure 2.9: Autonomic Control Loop [4]

Sensors collect data from the executing system and its context about its current state [4]. The accumulated data is then cleaned, filtered, and pruned and, finally, stored for future reference to portray an accurate model of past and current states. Then, the data is analysed to infer trends and identify symptoms. Subsequently, the planning (decide) attempts to predict the future to decide on how to act on the executing system and its context through actuators or effectors [4].

For instance, we focus on the feedback control loop proposed by IBM's autonomic computing initiative, also known as the MAPE-K control loop. MAPE-K includes the Monitoring, Analyzing, Planning and Executing functions and is fed with knowledge, the K on MAPE-K. We refer to the IBM's architectural blueprint for details on fundamental concepts, constructs and architectural building blocks of the MAPE-K loop [82].

2.4 Synergies between DSPLs and Self-Adaptative Systems

Self-adaptive systems can be built based on software product line principles by means of feature-based runtime adaptation. From the perspective of runtime adaptation, well-established variability modelling in the SPL domain promises to be a valuable basis for the definition of appropriate models at runtime. These models are required in adaptive systems [104]. Under these circunstances, although variability analysis and design can be performed at development time, the variability binding and reconfiguration is performed at runtime. Thus it requires some kinds of variability mechanisms to map environment variants to variable features, which, in turn, should be mapped to variants in the PLA level; and support runtime reconfiguration [105]. As a result, the self-adaptation strategies can be obtained and specified in a higher feature level rather than the lower program level, which makes it easily validated and understood by the system users [105]. On the other hand, the decision of producing a new product (*i.e.* to adapt to another configuration) can follow, among other solutions, the autonomic MAPE-k loop, a mechanism to fully automate the derivation process into self-adaptive systems [82].

Capítulo 3

An Experimental Setup to Assess Design Diversity of Alternate Services

Este capítulo apresenta uma infraestrutura que apoia a preparação e a execução de estudos empíricos para investigar o quão eficiente um conjunto de serviços alternativos é para tolerar falhas de software. A fim de avaliar a solução proposta e analisarmos as implicações de se utilizar diversidade de projetos para apoiar composições de serviços tolerantes a falhas, a infraestrutura foi utilizada para analisar a diversidade de vários serviços alternativos disponíveis na Internet. O conteúdo deste capítulo foi retirado de dois artigos, um publicado no *16th International Conference on Evaluation and Assessment in Software Engineering - EASE '12*, que apresenta as diretrizes para avaliar diversidade de projeto; o outro publicado no o *7th International Conference on Availability, Reliability and Security - ARES'12*, que apresenta os resultados de um estudo empírico abrangente sobre diversidade de serviços alternativos e suas implicações. Como o conteúdo deste capítulo foi extraído na íntegra de tais artigos, foi preservado o idioma original.

3.1 Overview

Software faults cannot be tolerated by simple replication of identical software components since the same mistake will exist in each copy of the components [1]. A solution to this problem is to introduce diversity into the software replicas [1, 100, 42]. Design diversity is the provision of functionally equivalent software components, called alternates, through different design and implementations [1]. The main goal of increasing diversity is to increase the probability that alternates fail on disjoint subsets of the input space, when they do fail [1, 42]. In software fault-tolerant architectures based on design diversity, called FT-architectures, a task is executed by several alternates. The results of these executions are provided to an adjudicator, which operates upon them to determine which one to output as the presumably correct result [1, 106]. Adjudicators might fail, when coincident failures occur. A *coincident failures* is said to have occurred if a majority of alternates fail on the same input case [1, 96]. Hence, FT-architectures might not provide an improvement in reliability over a single software component [1, 49, 107].

Today's society is highly dependent on systems based on Service-Oriented Architectures (SOA) for its basic day-to-day functioning [16, 57]. A composite service, the basis for the construction of applications in the SOA world, can be regarded as a combination of activities invoked in a predefined order and executed as a whole [16]. Nevertheless, it is unlikely that services, usually controlled by third parties, will ever be completely free from software faults arising out of wrong specifications and incorrect coding [34, 1]. In the past, employing design diversity used to be costly [53, 54] because the need for two or more alternates implied implementing the same system more than once, almost from scratch [49, 54]. However, nowadays, in the context of SOA, on the web a number of functionally equivalent web services, or alternate services, usually exist to achieve a particular task [55, 40, 56, 16]. These alternate services might be simply cost-free and open access or even offered by different organizations to their own business partners to cope with changes in the level of user requirements and quality of services (QoS) [16]. Due to the low cost of reusing alternate services, several diversity-based solutions have been developed [57]. These solutions operate as mediators between clients and alternate services. The latter are structured in fault-tolerant composite web services [40, 56], called *FT-compositions* for simplicity. From the clients' viewpoint, the FT-composition works as a single, reliable service.

solutions operate in the communication between clients and alternate services. The latter are structured in fault-tolerant composite services [40, 56, 33], called *FT-compositions* for simplicity. From the clients' viewpoint, the FT-composition works as a single, reliable service.

A considerable number of studies and experiments aim to assess design diversity [49, 50, 51, 52]. In most of existing research, the development teams, the adopted platform to implement alternates and their source code are well known. However, services are black boxes and independently developed by different organizations. Therefore, it is difficult to extrapolate the results of previous studies about design diversity to the context of SOA. Although diversity-based solutions exist in SOA, to the best of our knowledge, given a requirements specification, there are no directives to assess *(i)* whether its alternate services are actually provided by means of design and implementations that are sufficiently different, *i.e.* usable for fault tolerance [96, 97]; and *(ii)* how often its alternates fail on the same input case, i.e., how effective service diversity is for tolerating faults [1, 49, 107]. With proper assessment of service diversity, we are able to achieve higher levels of reliability by employing either *(i)* the FT-composition with selected alternate services; or *(ii)* a single non-fault-tolerant service, called *NFT-service*, that can exhibit higher reliability than would be the case if diversity was employed.

We propose a set of directives to organize the preparation and execution of an experiment to investigate, given a requirements specification, to what extent its alternate services are able to tolerate software faults. The investigation is performed from clients' viewpoint because services are black boxes [16, 60]. Firstly, the proposed solution supports an investigation on whether alternate services are diverse. This is achieved through the adoption of statistical tests to check whether alternate services present difference in their outputs and their failure behaviours. If alternates have different observed behaviour, it suggests that they are in fact provided by different design and implementations. Secondly, the solution also supports the analysis of if and by how much the use of the FT-composition improves reliability when compared to a NFT-service. For this, we estimate the achieved reliability of these two architectural-solutions.

We evaluated the applicability and usefulness of the proposed experimental setup by employing it to assess diversity of alternate services adhering to a number of requirements specifications. These empirical studies also provided more insight on the effectiveness of design diversity in service-oriented applications and its implications. We found out that, for some requirements, coincident failures of two services are frequent enough that using the most reliable service in isolation yields the best results. Data and observations regarding our studies are available at our study webpage [108]. We emphasize that empirical studies are needed from a Software Engineering (SE) perspective because they enable the development of scientific knowledge about how useful different SE solutions are, thus allowing for informed and well-grounded decision [75].

3.2 An Infrastructure to Assess Service Diversity

Figure 3.1 presents an overview of the proposed experimental setup. Given a requirements specification (*Activity I*), the investigation of service diversity is based on the execution of alternate services under the same sequence of input cases (*Activities II, III, IV*). Littlewood and Miller [97] have demonstrated that the probabilities of coincident failures are decreased when alternates exhibit diversity at the level of design, implementation and also in terms of failure behaviour. Therefore, first, we examine whether alternate services are sufficiently diverse [1, 42] (*Activities V, VI, VII*). Second, it is necessary to perform an empirical investigation to truly assess whether FT-composition is tolerating faults, if alternate services fail on disjoint subsets of the input space [1, 96, 97]. Furthermore, through the comparison of different architectural solutions (*Activities VIII,IX,X*), we are able to achieve higher levels of reliability by employing either *(i)* FT-composition (*Activity XI*); or *(ii)* the most reliable NFT-service (*Activity XII*).

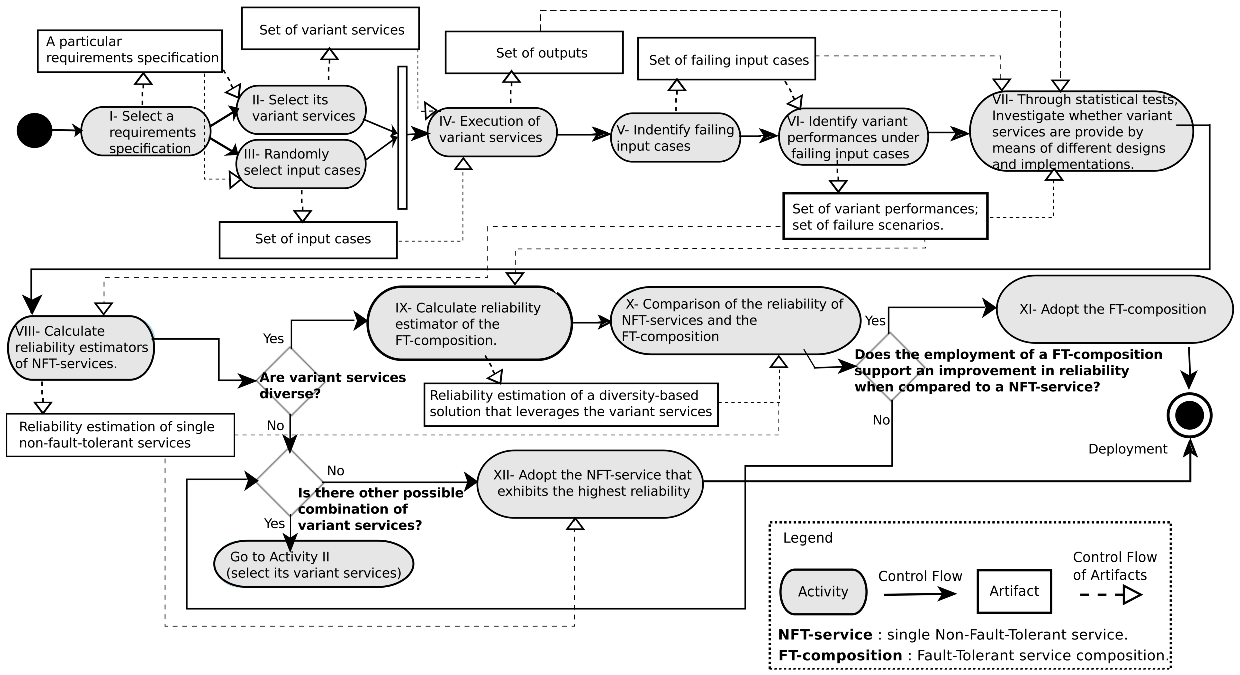
A particular requirements specification
Set of variant services
Set of outputs
Set of failing input cases
I- Select a requirements specification
II- Select its variant services
III- Randomly select input cases
IV- Execution of variant services
V- Indentify failing input cases
VI- Identify variant performances under failing input cases
VII- Through statistical tests, Investigate whether variant services are provide by means of different designs and implementations.
Set of input cases
Set of variant performances; set of failure scenarios.
VIII- Calculate reliability estimators of NFT-services.
Yes
Are variant services diverse?
No
IX- Calculate reliability estimator of the FT-composition.
X- Comparison of the reliability of NFT-services and the FT-composition
Yes
Does the employment of a FT-composition support an improvement in reliability when compared to a NFT-service?
No
XI- Adopt the FT-composition
Reliability estimation of a diversity-based solution that leverages the variant services
Reliability estimation of single non-fault-tolerant services
No
Is there other possible combination of variant services?
Yes
XII- Adopt the NFT-service that exhibits the highest reliability
Go to Activity II (select its variant services)
Deployment
Legend
Activity
Control Flow
Artifact
Control Flow of Artifacts
NFT-service : single Non-Fault-Tolerant service.
FT-composition : Fault-Tolerant service composition.

Figure 3.1: A method to assess design diversity of alternate services.

3.2.1 Research Questions and Hypotheses

Given a requirements specification and its alternate services, the proposed experimental setup aims to provide directives to answer the following research questions:

RQ_0: **Are alternate services diverse?**

To empirically investigate research question RQ_0, we hypothesize the following:

- *Null hypothesis (H_{01})*: There is no difference in observed outputs with respect to alternate services.
- *Null hypothesis (H_{02})*: There is no difference in observed frequency (or proportion) of failures with respect to alternate services.

RQ_1: **Does the use of a FT-composition support an improvement in reliability when compared to a NFT-service?**

3.2.2 Detailed Description of the Activities

In this section, we describe all activities presented in Figure 3.1.

Activity I

First of all, select a particular requirements specification. We call the selected specification r.

Activity II

Identify alternate services that realize the requirements specification r. In this work, V_r is the set of such alternate services. It contains exactly three elements, the minimum number of alternates employed by diversity-based techniques that leverage voters [1]. It is easy to obtain precise estimates for more than three alternates using the same experimental setup. We define V_r as follows:

$$V_r = \{v_1, v_2, v_3\}, \tag{3.1}$$

where v_1, v_2, v_3 are alternates implementing requirements specification r.

Activity III

Given the requirements specification r, select input cases at random from the input space [97]. The same inputs should be supplied to each of alternate $v \in V_r$ in order to improve the precision of the experiment [109, 75]. We define the X_r set of input cases as follows:

$$X_r = \{x_1, x_2, ..., x_m\}, \tag{3.2}$$

where x_i, $1 \leq i \leq m$, are inputs. The bigger the number of analysed input cases m, the more precise the obtained measurements [97, 109, 75].

We emphasize that minor adaptations in input data supplied to each alternate service might be required according to its expected formats of inputs. For example, suppose that a service is responsible for *Validating an email address*. For one of the alternates, data provided is *'a@a.com, validationOperator'*, while for others alternates, the data provided should be *'a@a.com'*.

Activity IV

After the sets of alternate services (*i.e.* V_r) and of input cases (*i.e.* X_r) have been identified, we are able to execute alternate services. Each input case $x \in X_r$ is distributed to each alternate $v \in V_r$, which then executes its operation. Since we are interested in investigating the output space, we define a function, called $exe(v, x)$, that returns the output value resulting from the execution of an alternate service v under input x.

Then, for each alternate service $v \in V_r$, we define the O_{r_v} set, which associates inputs, alternate services, and the corresponding outputs, as follows:

$$O_{r_v} = \Big\{(v, x, exe(v, x))|x \in X_r\Big\} \tag{3.3}$$

It should be noticed that the whole analysis is made based on the comparison of third elements of identified triples. The first two elements of the triples are only identifiers adopted to guarantee that there is no duplicate elements, which are not allowed (or are ignored) in sets. Moreover, different alternates may return output formats that are equivalent but not the same, therefore output sets need to be standardized before the analysis commences. For example, for the *Email Validation* task, given an valid email address, one alternate returns as output the *'It is valid'* string, while the other one returns *'1'*.

Activity V

Identify input cases under which at least one alternate fails. The identification of failing input cases is essential to analyse and understand the failure behaviour of alternate services [97, 110]. To identify failing inputs, we propose the adoption of the performance function $p(v, x)$, defined by Littlewood and Miller [97]. This function indicates whether an alternate v, when executed under the input x, returns either a correct result or a failure within the maximum expected time frame TF (*Section 2.2.6*). The performance function $p(v, x)$ is defined as follows [97]:

$$p(v, x) = \begin{cases} 1 & \text{if } v \text{ does } \textbf{not} \text{ fail on } x \text{ within } TF \\ 0 & \text{otherwise} \end{cases} \tag{3.4}$$

This function requires a means to identify whether a alternate fails or not under a certain input. For example, adoption of assertions on the anticipated system state [1]; adoption of a gold version [49, 53]; a consistency relationship between inputs and outputs that is sufficient to assert correction [42]; and comparison of outputs of a large number of versions [42].

Let $FX_r \subseteq X_r$, be the set of failing input cases. We define FX_r as follows:

$$FX_r = \{x | x \in X_r \ \wedge (\exists v \in V_r \mid p(v,x) = 0)\} \tag{3.5}$$

where $|FX_r| \geq 1$. If $|FX_r| = 0$, it means that none of the alternates failed on any of the analysed input cases. If that is the case, it is not necessary to conduct the remaining activities, since using any single alternate would yield the highest reliability.

Activity VI

For each input case under which a alternate fails, if any other alternate does not fail on the same input case, there is evidence that *(i)* a diversity-based technique might tolerate that fault [110]; *(ii)* alternate services fail on disjoint subsets of the input space [110]; *(iii)* alternate services have distinguishable frequency of failures. In order to look for such evidence, we should analyse the performance of each alternate under failing input cases. The analysis consists observing the outputs produced by all the alternates under the inputs from FX_r and determining which alternates failed under those inputs. For each alternate $v \in V_r$, we define PV_{r_v} as the set of performances under failing inputs:

$$PV_{r_v} = \Big\{(v, x, p(v,x)) | x \in FX_r\Big\} \tag{3.6}$$

In a complementary way, for each requirements specification $r \in R$, let $FS(r)$ be the set of all alternate performances under failing inputs in specification r. That is, the $FS(r)$ associates all alternate services belonging to $V(r)$, failing input cases from $FX(r)$ and alternate performances under such failing inputs. The $FS(r)$ set is called as failure scenario, for simplicity.

$$FS(r) \ = \ PF(r, v_1) \cup PF(r, v_2) \cup PF(r, v_3) \tag{3.7}$$

Activity VII

At this point, we have the data that is necessary to answer research question RQ_0 (*Section 3.2.1*), that is, whether alternate services belonging to V_r are diverse or not. For that, we need to test null hypotheses H_{01} and H_{02}, which go on to state that alternate services have no distinguishable effect on the outputs and frequency of failures. Therefore, to check H_{01} and H_{02}, it is necessary to determine whether, respectively, the sets of outputs (*i.e.* O_{r_1}, O_{r_2},O_{r_3}) and sets of alternate performances (*i.e.* FS_{r_1}, FS_{r_2}, FS_{r_3}) are significantly different among themselves. Consequently, we need to find out probabilities p_{01} and p_{02} of rejecting, respectively, hypotheses H_{01} and H_{02} at the α significance level [109, 75], where $\alpha = 0.05$, for instance. That is, given probabilities p_{01} and p_{02}, if any probability is less than α, we can reject its related null hypothesis at a 0.05 significance level [109, 75].

In this context, we adopt non-parametric statistical tests to investigate both hypotheses H_{01} and H_{02}. In general, nonparametric statistical tests typically are more accessible than parametric ones [109, 75], which further motivates our work. On the one hand, nonparametric tests are based on a model that specifies only very general conditions and none regarding the specific form of the distribution from which the sample was drawn [109]. On the other hand, parametric tests require complex assumptions about analysed data [109].

In particular, our solution is based on nonparametric statistical tests for the case of k related or matched samples [109]. In these tests, k related samples of equal size (n) are matched according to some criteria, sometimes called treatments, which may affect the values of the observations [109]. These tests are suitable for our experimental setup due to the following reasons. Given a particular requirements specification r, the same input cases, which belong to the X_r set, are subjected to different treatments, that is, it is processed by different alternate services belonging to the V_r set. We want to investigate whether these different treatments have distinguishable effects on observed output space of alternates (*i.e.* in their sets of outputs and performances). The adopted tests are summarized in Table 3.1. Each test is suitable for specific sampling types of variables under study. The supported sampling types also show the capabilities and limitations of our solution.

Table 3.1: Selected Statistical Tests

Nonparametric Statistical Test	**Brief Description**	**Required sampling type**
Cochran Q test	It provides a method for testing whether three or more matched sets of frequencies or proportions differ significantly among themselves [109].	Categorical (specifically, in a dichotomized ordinal scale or in a nominal scale with two levels) [109]
Friedman two-way analysis of variance by ranks	It tests whether k samples are from the same population or population with the same median. [109].	At least an ordinal scale (*i.e.* variable under study are at least ordered) [109].

In particular, regarding the H_{01} hypothesis, we should first analyse the type of output in order to choose a suitable statistical test. Whereas, for the H_{02} hypothesis, we always adopt the *Cochran Q* test, because data is dichotomized as '1 '(*i.e.* success) and '0'(*i.e.* failure). When one or both of the hypotheses H_{01} and H_{02} are rejected, we have enough evidence to conclude that diversity is applied to alternate services. Otherwise, they should have similar or identical behaviour when executed under the same sequence of inputs. However, we should prefer a set of alternate services that present difference in outputs and in terms of failures [97]. The more diverse alternates are, the less likely coincident failures are to occur [97]. Nevertheless, given any individual requirements specification, if we cannot reject at least one of these hypotheses, there is not enough evidence suggesting that alternate services are diverse and, thus, *usable* for fault tolerance. In these scenarios, if there are no more alternate services to be analysed, we should employ the NFT-service that exhibits the highest reliability (Activities VIII,XII). We refer to Siegel and Castellan Jr. [109] for further details on both the *Cochran Q test* and the *Friedman analysis*.

Activities VIII, IX

In order to measure the reliability of NFT-services and of FT-compositions, we define, respectively, the reliability estimators $Rel\_Est_{NFT\_S_{rv}}$ and $Rel\_Est_{FT\_SOA_r}$. We estimate both $Rel\_Est_{NFT\_S_{rv}}$ and $Rel\_Est_{FT\_SOA_r}$ by means of the successful execution rate, which calculates the probability that a request is correctly responded (*Section 2.2.6*). Furthermore, to improve the precision of the obtained measures, all the reliability estimators we define are based on the analysis of failing inputs, which belong to the FX_r set.

Estimator for NFT-services

We define the $Rel\_Est_{NFT\_S_{rv}}$ estimator for NFT-services as follows:

$$Rel\_Est_{NFT\_S_{rv}} = \frac{\sum_{x \in FX_r} p(v,x)}{|FX_r|} \tag{3.8}$$

Where, $|FX_r| \geq 1$, as mentioned, and $0 \leq Rel\_Est_{NFT\_S_{rv}} \leq 1$. If $Rel\_Est_{NFT\_S_{rv}} = 1$, it implies that the analysed alternate service is able to return correct results for all failing input cases from FX_r. If $Rel\_Est_{NFT\_S_{rv}} = 0$, the analysed alternate service failed on all failing inputs.

Estimators for a FT-Composition

To define the reliability estimator for a FT-composition $Rel\_Est_{FT\_SOA_r}$, we considered the general definitions of voters (Section 2.2.2). Specifically, we are interested in cases where a majority of the alternates were successful and a voter is able to adjudicate a correct result [1, 42]. Also, we assume that one is able to develop a perfect voter. Previous studies evaluating design diversity as a technique to improve overall system reliability have made the same assumption [1, 106, 49, 98, 111]. We define the reliability estimator $Rel\_Est_{FT\_SOA_r}$ for a FT-composition, as follows:

$$Rel\_Est_{FT\_SOA_r} = \frac{\sum_{x \in FX_r} geq\Big((p(v_1,x) + p(v_2,x) + p(v_3,x)), 2\Big)}{|FX_r|} \tag{3.9}$$

where, *geq* is an operator that returns 1 if the first argument is greater than or equal to the second argument and 0 otherwise, $|FX_r| \geq 1$, and $0 \leq Rel\_Est_{FT\_SOA_r} \leq 1$. If $Rel\_Est_{FT\_SOA_r} = 1$, it implies that voters are always able to adjudicate a single correct result from alternate results, where alternates were executed under input cases belonging to the FX_r set. Hence, voters were able to tolerate all faults whose activation has led to failure of alternates. If $Rel\_Est_{FT\_SOA_r} = 0$, voters are not able to tolerate any faults whose activation has led to coincident failures.

It is important to notice that the reliability estimation of a FT-composition should take into account the adopted adjudicator [112]. If instead of using voters, we adopt acceptance tests, then the FT-composition will be successful under an input case if at least one of the results from the target alternate services satisfies the assertion employed by the acceptance test [1]. At this point, our solution can be extended to support other adjudicators.

Activity X

Finally, we check whether the FT-composition is more reliable than each NFT-service in isolation in order to answer research question RQ_1 (*Section 3.2.1*). Given the reliability estimators $Rel\_Est_{NFT\_S_{rv}}$ (the alternate with the highest reliability) and $Rel\_Est_{FT\_SOA_r}$, we estimate the overall differences in reliability achieved by adopting different architectural solutions by calculating $(Rel\_Est_{FT\_SOA_r} - Rel\_Est_{NFT\_S_{rv}})$. As discussed in *Section 2.2.6*, a positive difference indicates that the FT-composition exhibits higher reliability (*Activity XI*). On the other hand, a negative difference indicates that, if there are no more alternate services to analyse, we should employ the most reliable NFT-service (*Activity XII*).

3.3 Evaluation

We have applied the proposed directives and executed the experimental procedures on a number of alternate services adhering to different requirements specifications. Our goal was twofold: (i) to exemplify the use of the experimental setup to assess alternate services developed by third-parties; (ii) to show its effectiveness to evaluate the reliability of FT-compositions and NFT-services; (iii) and to provide more insight on the effectiveness of design diversity in service-oriented applications and its implications.

3.3.1 Research Questions and Hypotheses

To conduct our investigation, we analysed alternate services adhering to seven different requirements specifications. This study aims to answer the following research questions.

$\mathbf{RQ_0}$: *Do functionally equivalent services (i.e. alternate services) present diversity in their design and implementations?*

To empirically investigate research question RQ_0, for each specification, we hypothesize the following:

Null hypotheses ($H_{01...07}$): There is no difference in observed outputs with respect to alternate services.

Null hypotheses ($H_{11...17}$): There is no difference in observed frequency of failures with respect to alternate services.

Although hypotheses are assigned to each requirements specification, we do not focus on understanding the real impact of each specification individually, only of the results. Given any individual requirements specification, if we cannot reject at least one of its related hypotheses, within the scope of this work, we will report that there is not *enough evidence* to either confirm or deny the existence of diversity in the implementation of its alternate services.

$\mathbf{RQ_1}$: *Does the use of a FT-compositions, built with alternate services and voters, support an improvement in reliability when compared to single services?*

3.3.2 Target Requirements Specifications and Alternate Services

We selected seven requirements specifications that are performed by a population of third-party, stateless, read-only and SOAP/WSDL-based Web services. We will refer to these as services, for simplicity. The selection of these services required cost-free alternate availability of services and the possibility of comparing alternate results. Table 3.2 briefly describes the requirements specifications (*Activity I*). Alternate services adhering to these specifications were selected at online services repositories [113, 114] (*Activity II*). For all alternates, we identified their corresponding descriptions, specified in *WSDL* documents. Based on these descriptions, we created Web service clients by adopting the *Java API for XML Web Services (JAX-WS)*, a Java programming language API for creating and invoking Web services [84].

3.3.3 Target Input Cases

For each requirements specification, we generated the X_r set of inputs *(Activity III)* as follows.

(1) Email Validations: We choose three popular domains (*i.e.* Yahoo, Gmail and Hotmail) and two unpopular domains (*i.e.* ic.unicamp.br, ige.unicamp.br). Each one of these mail

Table 3.2: Target Requirements Specifications.

Requirements Specification (Functionality)
(1) *Email Validations* validates email addresses for client applications.
(2) *Credit Card Validations* validates credit card numbers.
(3) *Distance By ZIP Codes* finds the distance between any two ZIP Codes.
(4) *Currency Trading* returns 'up-to-date' currency rates.
(5) *Temperature Conversion* converts temperature from Kelvin to Celsius.
(6) *Weather Forecast* provides weather forecast for cities.
(7) *ZIP code geocoding* converts ZIP codes into geographic coordinates.

servers specifies its restrictions on login specifications. Based on these restrictions, we randomly generate 1149 mail addresses as inputs to our experiment. We generated valid email addresses including existing ones and not-valid addresses.

(2) Credit Card Validations: We randomly generated 1046 credit card numbers. The valid credit card numbers were generated conforming to the Luhn formula (MOD 10 check) [115]. *MOD 10 check* is a simple checksum formula used to validate a variety of identification numbers, including credit card numbers [115]. The adopted valid credit card types were *MasterCard, VISA 16 digit, VISA 13 digit, American Express, Discover, Diners Club, enRoute, JCB 15 digit, JCB 16 digit* and *Voyager*. Invalid credit card numbers were randomly generated and they did not conform to the Luhn formula.

(3) Distance By Zip Codes: We generated 1021 input pairs comprising of an origin Zip code and a destination one: $< Starting\ ZIP\ code, Ending\ ZIP\ code >$. Both codes were randomly selected from a database that contains 13137 Zip Codes within the United States [116].

(4) Currency Trading: The following currencies compose the input set used to get 'up-to-date' currency exchange rates: AUD *(Australian Dollars)*, CAD *(Canadian Dollars)*, CHF *(Swiss Francs)*, DKK *(Danish Kroner)*, EUR *(Euros)*, HKD *(Hong Kong Dollars)*, JPY *(Japanese Yen)*, NOK *(Norwegian Kroner)*, NZD *(New Zealand Dollars)*, SEK *(Swedish Krona)*, SGD *(Singapore Dollars)*, TWD *(Taiwan Dollars)*, USD *(US Dollars)* and ZAR *(South African Rand)*.

(5) Temperature Conversion: We randomly generated 1049 real numbers as input case including values below absolute zero, the lowest possible temperature [117].

(6) Weather Forecast: We randomly selected 926 Zip Codes from the database that contains 13137 US Zip Codes [116].

(7) US Zip code geocoding: We randomly selected 1200 Zip codes from a database with 13137 US Zip Codes [116].

For each requirements specification, we distributed input cases to its alternates, which then execute their operations by means of the Java clients. Therefore, we identified, for each alternate service $v \in V_r$, its set of outputs O_{r_v} (*Activity IV*).

3.3.4 Identification of Failures

For each requirements specification $r \in R$, given its alternate services $v \in V(r)$ and its inputs $x \in X(r)$, we identified whether an alternate service fails or not under certain input. Usually, in this type of experiment, the experiment administrator employs a 'gold' version as a means to identify failures [49, 98]. It is used by comparing the output of the alternate service, for a given input case, to the output that is known to be correct, the one returned by the gold version [49, 98]. For each particular set of requirements $r \in R$, we either wrote or employed a third-party gold version $g(r)$. In a complementary way, for some alternate services, we adopted acceptance criteria, which is used to evaluate whether a alternate result is acceptable compared to the one returned by the gold version, as follows.

Gold Versions: For the *(1) Email validations*, we implemented a program based on the restrictions on login specifications defined by the adopted mail servers. For *(2) Credit card validations*, we implemented a routine based on the *Luhn* algorithm. This algorithm is described in ISO/IEC 7812-1[118]. For the *(3) Distance by ZIP codes*, we implemented an algorithm based on Google API [119]. The gold version for *(4) Currency trading* is based on the values provided by *CNN Money* [120]. For the *(5) Temperature conversion*, our implementation was based on the relationship between Kelvin and Celsius scales [117]. For *(6) Weather forecast*, we implemented a routine based on information for weather forecast provided by *Weather.com*. The *(7) Geocoder gold version* is based on the Google API [119].

Acceptance Criteria The *(3) Distances by ZIP codes*, *(4) Currency Trading*, *(5) Temperature Conversions* and *(6) Code Geocoding* are realized by services that use floating-point arithmetic (FPA). The use of FPA in general computing produces a result that is accurate only within a certain range [1]. The use of design diversity, especially if FPA is used, can also produce individual alternate results that differ within a certain range [1]. In this way, Pullum defines a tolerance as a variance allowed by a decision algorithm to decide whether results were successful [1]. Consequently, for the services whose outputs use FPA, we adopted acceptance criteria to check whether alternate services 'disagree'or not with gold versions. We specify tolerance ranges based on measures of dispersion under all alternate results and the gold version result, thus, reducing subjectiveness of the specified criterion.

For each requirement specification, we identified all input cases under which at least one of its alternate service fails, *i.e.*, we identified its FX_r set (*Activity V*). For instance, the maximum expected time frame (TF) is of 5 minutes. For each requirements specification, Table 3.3 shows the absolute sizes of X_r and FX_r, *i.e.*, the number of inputs and the number of failing inputs. For *(1) Email Validation*, for example, we have identified 454 input cases under which at least one alternate failed, out of a total of 1449 input cases. Afterwards, based on the FX_r set, we generated for each alternate $v \in V_r$ its set of performances PV_{r_v} (*Activity VI*).

Table 3.3: Number of input and failing inputs.

Requirements Specification	$\|X_r\|$	$\|FX_r\|$
(1) Email validations	1449	454
(2) Credit Card Validations	1046	262
(3) Distance by ZIP Codes	1021	238
(4) Currency Trading	15	1
(5) Temperature Conversion	1049	765
(6) Weather Forecast	925	224
(7) ZIP code geocoding	1200	210

Details on *(i)* alternate descriptions; *(ii)* input cases; *(iii)* gold versions; *(iv) Java clients*; are available at our study webpage [108].

3.3.5 Study Results and Discussion

Once all data have been collected, we analysed them to provide an in-depth analysis of design diversity in service-oriented applications. In this section, we present and discuss the results and their implications.

Investigating Research Question RQ_0

For each requirements specification, the set of its input cases is subjected to different treatments, that is, it is processed by different alternate services [112]. We investigated whether different treatments have distinguishable effect on the observed behaviour from client's viewpoint, in particular, on the output values and frequency of failures [112] (*Activity VII*). To find out all probabilities, which are estimated based on observed data, we adopted R, a language and environment and programming language for statistical computing and graphics [121, 122]. Details on the calculations and our R working environment are available at our study webpage [121].

Comparison of resulting outputs

For each requirements specification, we checked whether sets of alternate outputs are significantly different among themselves. That is, we test the null hypotheses $H_{01,\ldots,07}$ (*Section 3.3.1*). On one hand, for *(1) Email Validations* and *(2) Credit Card Validations*, the output values returned by their alternates are categorical data. On the other hand, for *(3) Distances by Zip Codes, (4) Currency Trading, (5) Temperature Conversions, (6) Weather Forecasts* and *(7) ZIP code geocoding*, the output values are data in at least an ordinal scale. Therefore, we adopted different statistical tests for checking $H_{01,02}$ and $H_{03\ldots07}$.

To test the null hypotheses H_{01} and H_{02}, we adopted the *Cochran Q test*. This test is suitable for out study because *(i)* the data are from $|V(r)| > 2$ related groups; *(ii)* the data are dichotomized as *valid* and *not-valid*; *(iii)* we want to examine whether the frequency of both 'valid' and 'not-valid' as outputs are the same for all alternate services when they are executed under the same sequence of inputs. To test the null hypotheses $H_{03\ldots07}$, we adopted the *Friedman two-way analysis of variance by ranks*. This analysis is suitable for our study because *(i)* we want to find out whether alternate services are from the same population or not, that is, have similar output values when executed under the same input cases; *(ii)* the analysed data are in at least an ordinal scale [112, 109]. We rejected the hypotheses $H_{01,02}$ and $H_{03\ldots07}$ for $p - value < 0.05$ [112, 109]. These results suggest that diversity is applied to alternate services, otherwise, they would have had similar or identical outputs values when executed under the same sequence of inputs [112].

Comparison of frequencies of failures

For each requirements specification $r \in R$, given its alternate services $v \in V(r)$, we checked whether its set of performances PV_{r_v} are significantly different among themselves. We test the null hypotheses $H_{11\ldots17}$ at the significance level α ($\alpha = 0.05$). We adopted the *Cochran Q test* which is suitable for this study because *(i)* the data are from ($|V(r)| > 2$) related groups; *(ii)* the data are dichotomized as '1 '(*i.e.* success) and '0'(*i.e.* failure); and *(iii)* we want to examine whether the frequency of both correct results and failures are the same for all alternate services, which are executed under the same input cases [112]. According to the results the probabilities p_{11}, p_{12}, p_{13}, p_{15}, p_{16} and p_{17} were less than the significance level (α). Therefore, the results provide us evidence to reject the null hypotheses $H_{11,12,13,15,16,17}$. Hence, for all requirements specifications, except for *(4) Currency Trading*, we conclude that frequencies of failures are dependent on their alternate services with 95% confidence [109]. That is, alternate services have a distinguishable effect on the distribution of failures. For *(4) Currency Trading*, there is only one failing input, thus we did not adopt the *Cochran Q test* to either accept or reject H_{14}.

For each specification $r \in R$, we graphically illustrate the relative frequency of both correct results and failures of each individual alternate service $v \in V(r)$ by means of $|R|$ bar charts. These charts are represented in Figure 3.2. Failures and correct results are represented, respectively, under label *Failure* and *Success*. Alternate service identifiers (*i.e.* $v1$, $v2$, $v3$) and *Total* represent the analysed categories. Overall success rate, under label *'Total'*, give us the total frequency of failures and successes presented into a failure scenario (the $FS(r)$ set, Definition 3.7). For each particular specification, differences regarding the observed behaviour of its alternate services and the grand total might suggest that the proportion of failures is dependent on the alternate services. Otherwise, proportions of outputs of a particular kind would be similar or identical for all bars in the related chart [123]. In Figure 3.2, we can notice that differences among frequencies of failures are less pronounced for alternate services belonging to *(5) Temperature Conversions* (Figure 3.2(5)). For other requirements specifications, at least two of their alternates have marked differences on their proportions of failures (*e.g.* Figure 3.2(1); Figure 3.2(2); Figure3.2(7)). Moreover, Figure 3.2(4) suggests that we can reject the hypothesis H_{14}, otherwise its alternates should present the same proportion of failures and correct results. All these observations reinforce our conclusions that the target alternate services are diverse (Activity VII).

Investigating Research Question RQ_1

We examined which alternates fail coincidentally and which ones return similar (or identical) correct results coincidentally (*Section 2.2.2*). Figure 3.3 presents the relative frequency of both coincident failures and similar correct results in each failure scenario. Coincident failures are represented under the label *'Failure'*, while similar correct results are represented under label *'Success'*. We represented all possible coincident failures, that is: $v1v2$ ($v1$ and $v2$ fail, $v3$ does not fail), $v2v3$ ($v2$ and $v3$ fail,$v1$ does not fail), $v1v3$ ($v1$ and $v3$ fail, $v2$ does not fail) and $v1v2v3$ (all alternates fail). In a similar way, we represented similar correct results, that is, $v1v2$ ($v1$ and $v2$ are successful, $v3$ fails), $v2v3$ ($v2$ and $v3$ are successful, $v1$ fails) and $v1v3$ ($v1$ and $v3$ are successful, $v2$ fails). The case where $v1$, $v2$ and $v3$ are correct does not belong to the set of failure scenarios, which is composed by alternate performances under failing input cases. Moreover, we represented, under label *'Total'*, the total proportion of coincident failures and coincident similar results into a failure scenario.

By analysing Figure 3.3, we can notice that frequencies of coincident failures are not the same under different matched subsets of alternate services. Otherwise, for each particular specification, in its related chart, proportions of coincident failures would be similar or identical for all five bars [123]. For example, regarding coincident failures, in Figure 3.3(1), the alternates $v1$ and $v2$ fail coincidentally in approximately 20% of the input cases, while

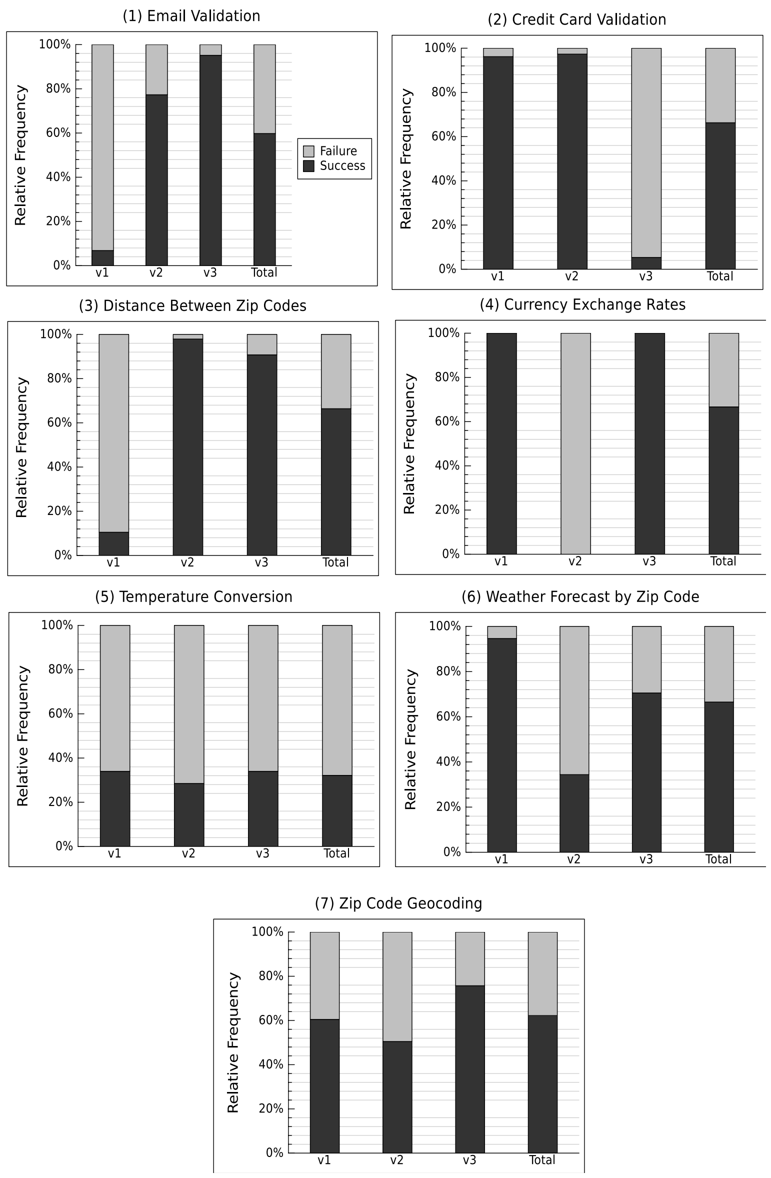

Figure 3.2: Joint frequency distribution of failures and successes in failure scenarios

$v1$ and $v3$ do not fail on the same input case. Related to similar correct results, e.g., in Figure 3.3(3), the alternates $v1$ and $v3$ return similar correct results coincidentally in approximately 4% of the input cases, while the alternates $v2$ and $v3$ are coincidentally successful in approximately 88% of the input cases which belong to the failure scenario.

Moreover, we can notice that behaviours in terms of both coincident failures and similar correct results differ among failure scenarios belonging to different requirements specifications. For example, in some circumstances, failure scenarios are composed predominantly by coincident failures, *e.g.* Figure 3.3(5), while other scenarios are mainly

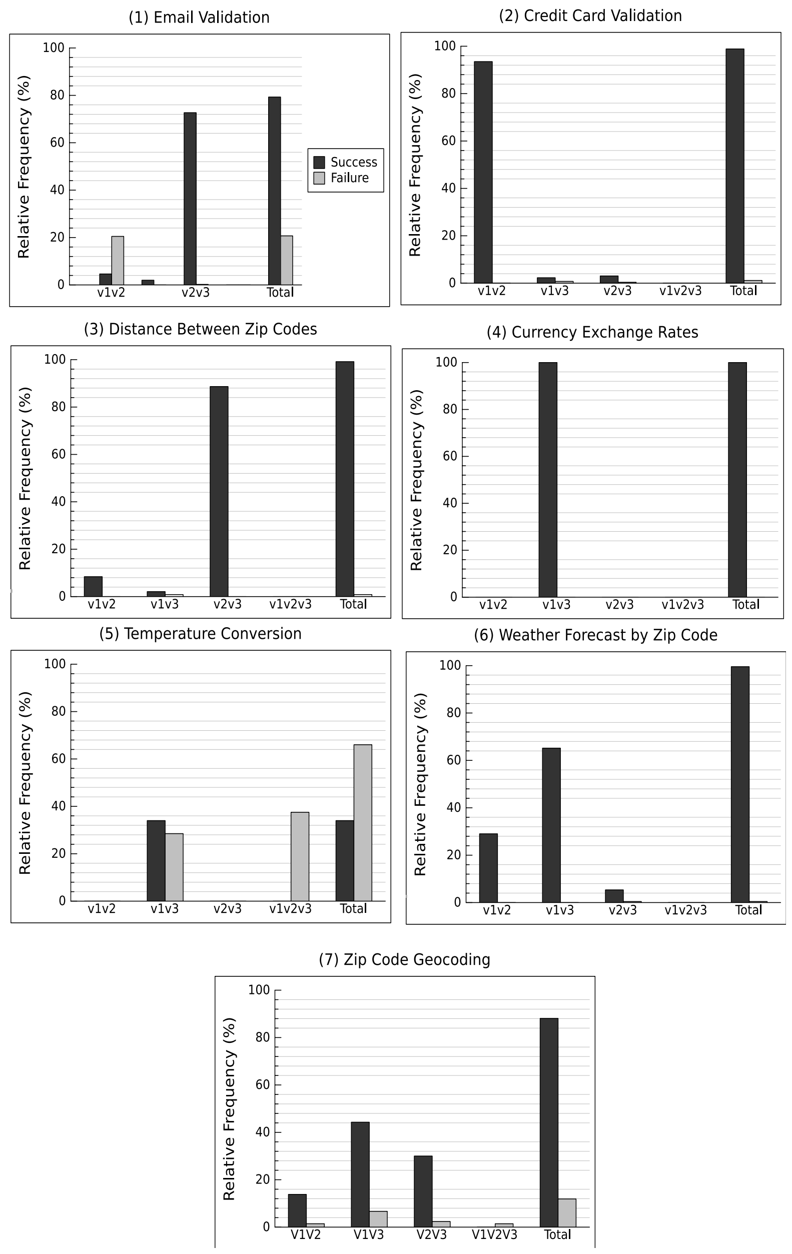

Figure 3.3: Relative frequency of coincident failures and similar correct results in failure scenarios.

composed by similar correct results, *e.g.* Figure 3.3(2) and Figure 3.3(3). Moreover, for some requirement specifications, similar or identical correct results represent approximately 100% of failures scenarios (*e.g.* Figure 3.3(2); Figure 3.3(3) and Figure 3.3(4)). It is rare for all three alternates to fail coincidentally. However, related to the *(5) Temperature Conversions*, in Figure 3.3(5), we can notice that all alternates fail in some 36%

of input cases. All these observations suggest that frequency of coincident failures is dependent on both the requirements specifications and their alternate services.

On the basis of the data represented in Figure 3.2 and Figure 3.3, for each requirements specification r, we also estimated the reliability of each NFT-service $v \in V_r$ and of the FT-composition (*Activities VIII and IX*). These reliability measurements are calculated by means of the estimators, respectively, $Rel\_Est_{NFT\_S_{rv}}$ and $Rel\_Est_{FT\_SOA_r}$, whose values are between 0 and 1. For each analysed requirements specification, we also identified its alternate service that exhibits the greatest reliability. Figure 3.4 summarized the obtained values for such estimators. For example, for *(1) Email Validations*, the reliability estimated for the most reliable single service is about 0.95, while the one estimated for the FT-Composition is about 0.80.

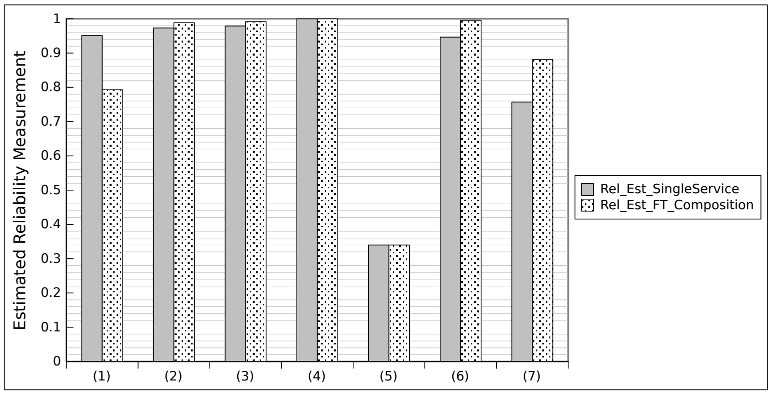

Figure 3.4: Reliability estimations for the different architectural solutions

Based on the values represented in Figure 3.4, we estimated the overall differences in reliability achieved by adopting different architectural solutions (*Activity X*). The obtained values, expressed as a percentage, are summarized in Table 3.4. A positive difference in reliability indicates an increase in reliability [42], i.e., the FT-composition tolerated faults of its alternate services, which rarely fail on the same input cases [1]. A negative difference indicates a reliability decrease [42]. The introduction of design diversity might lead to the occurrence of coincident failures, which might defeat most adjudicators [1, 42]. In Figure 3.4 we can observe that, for three specifications (*i.e.* Figure 3.4 (1); Figure 3.4 (4); Figure 3.4 (5)) the reliability estimation of fault-tolerant composite service is equal to or less than the one achieved by a single service (*i.e.* the overall reliability either remains the same or decreases). Such fact is confirmed across the second column of Table 3.4. Therefore, regarding research question RQ_1, we cannot be confident that service diversity is always *efficient* to tolerate software faults.

Table 3.4: Estimation of the overall difference in reliability

Requirements Specification	**Percentage of reliability improvement**
(1) Email validations	*(-15.90)*
(2) Credit Card Validations	1.50
(3) Distance By ZIP Codes	1.30
(4) Currency Trading	0.00
(5) Temperature Conversions	0.00
(6) Weather Forecasts	4.90
(7) ZIP code geocoding	12.30

3.3.6 Study Limitations

We briefly discuss the limitations of our study based on the categories of validity threats presented by Wohlin et al. [75]. For each category, we identified the possible threats to validity and, whenever it is applicable, the measures we took to reduce the risks.

Internal Validity: One threat to internal validity we identified is guaranteeing that gold versions result in correct results under all input cases. Regarding identification of failures, the gold version actually just provides another version to check against [98]. It is, of course, possible that failures common to all of the versions, including the gold one, were not detected [42, 49, 98]. This is an unavoidable consequence of this type of experiment as pointed out in the related literature [42, 49, 98]. Hence, both alternate services and analysed FT-compositions might produce results that are more or less reliable than the measured ones. To mitigate this risk, as part of the experiment, the gold version has been subjected to several test cases.

Construct Validity: This work does not address at all the aspects related to voter implementation problems (*e.g.* synchronization of the alternates, delays due to communication between the end-user servers and the various remote servers, maintainability issues of fault-tolerant compositions). That is, the adoption of voters might affect other constructs negatively. Since we do not observe these unintended side effects of voters, we identify one more threat to the construct validity: the restricted generalizability across constructs, as suggested by Wohlin et al. [75]. However, it should be noticed that the study of side effects of voters is outside the scope of this paper. We performed an investigation, specifically, on design diversity of alternate services and we assume that one is able to develop a perfect voter, as already mentioned [1, 106, 49, 98, 111].

External Validity: We identified one threat to external validity. The alternate services may not be representative of industrial practice since all of them are based on simple functionality. Regarding such risk, since we were looking for evidence on whether alternate services are able to face software faults, the complexity of service functionality would have no negative effect on our final conclusions because more complex systems have larger design spaces. Therefore, there are more opportunities for the introduction of problems that have different causes [1, 49]. Furthermore, while the empirical analysis of design diversity was a hotly-debated topic in the mid-1980s and early 1990s [1], no studies so far have been carried out in the context of Web services and this study represents a step stone in this direction.

Conclusion Validity: We identified three threats to conclusion validity: *(i)* the number of requirements specifications *i.e.*, sample size; *(ii)* the homogeneity of input cases; and *(iii)* the time-out setting for the request. Risk *(i)* cannot be completely avoided due to the lack of requirements specifications implemented by cost-free, functionally equivalent SOAP/WSDL-based Web Services. Moreover, existing empirical studies on effectiveness of design diversity for fault tolerance are based on the analysis of only one requirements specification, which is implemented by several variant components [49, 52, 110]. Therefore, seven requirements functionalities seem to be sufficient to derive preliminary conclusions about the general design diversity of services. Regarding risk *(ii)*, according to Littlewood and Miller [97], in order to assess whether alternates fail independently, it is necessary to guarantee that input cases are also independent, *i.e.*, heterogeneous. In this way, to mitigate risk *(ii)*, inputs from the input space were chosen at random for each requirements specification. Related to the risk *(iii)*, it is well known that services might take a variable amount of time to respond requests due to the dynamic and unpredictable nature of communication links. Consequently, some of the failures might be observed because services do not respond within the expected time frame. To mitigate this risk, we specified a high value of the time-out setting for requests (*i.e.* 5 minutes).

services do not respond within the expected time frame. To mitigate this risk, we specified a high value of the time-out setting for requests (*i.e.* 5 minutes).

3.4 Lessons Learned

We proposed an assessment approach to investigate design diversity of alternate services. The feasibility of the proposed approach was assessed using third-party, stateless, read-only, SOAP/WSDL-based alternate services adhering to seven different requirements specifications. First, diversity-based fault tolerance techniques require some form of diversity among alternates [1]. For all analysed requirements specifications, we found out that output samples returned by the execution of different alternate services differ significantly among themselves. That is, the output values are dependent on the alternate services. Therefore, we have sufficient evidence to conclude that alternate services in fact seem to be diverse and *usable* for software fault tolerance. Second, design diversity aims to make alternates as diverse as possible in order to minimize identical design faults and implementation mistakes. However, for some requirements specifications, the number of input cases under which most or all alternates failed coincidentally was high enough that using the most reliable service in isolation yields the best results. Therefore, this study has shown that diverse designs in SOAs do not always result in increased overall service reliability, which also reinforces the usefulness of our assessment framework.

For all analysed requirements specifications, the results also suggest that the frequency of failures depends on the alternate service. Therefore, since there is difference in observed proportion of failures with respect to alternate services, we can also conclude that the frequency of coincident failures seems to be dependent on adopted alternate services. That is, different combination of alternate services, which are structured in FT-compositions, might imply in different measures of enhanced reliability. Consequently, in order to try to achieve higher measures of reliability by adopting diversity-fault tolerance techniques, we should first observe effects of combining different alternate services.

Furthermore, care needs to be taken by software developers when selecting alternate services based solely on diversity of their failure rates. In fact, the FT-composition that exhibited the highest increase in reliability (*i.e. (7) Zip code geocoding*) was composed of alternates that had only $\approx 30\%$ difference in the failure-rates between the most faulty and least faulty alternate. For five of the requirements specifications, alternates with higher variance in failure-rate actually displayed a lower improvement in reliability. For this reason, additional indicators must be utilized to assess how effective service diversity is to tolerate software faults, such as taking into consideration the individual failure rate of each alternate. By applying a failure-rate threshold that a alternate service must satisfy may help software designers pinpoint alternates that are likely to negatively contribute to the overall service reliability. Once potentially problematic alternates have been isolated, appropriate action can be taken such as eliminating the alternate from the FT-composition or, alternatively, by increasing the total number of alternates to the design so that the higher coincident failure-rates have less influence on the overall FT-composition. We emphasize that in studies on the effectiveness of voting algorithms, it was shown that voters have a high probability of selecting the correct result value when the reliability estimated for each alternate is greater than 0.5 [1].

3.5 Related Work

Although there is much related research in the literature, we are not familiar with any work that proposes a general experimental setup to evaluate design diversity of alternates, when they are black boxes, and its effectiveness for fault tolerance. We address, in turn, work related to (i) effectiveness of design diversity, (ii) design diversity of alternates, and

(iii) design diversity in SOAs.

Empirical Studies on Effectiveness of Design Diversity for Fault Tolerance

Knight and Leveson [49] and Eckhardt et al. [52], describe an experiment to investigate whether it is valuable to use N-version programming, a design-diversity technique based on voters, to achieve high levels of reliability (*Section 2.2.5*). Their experiments are based on the analysis of the failure behaviour of several alternates of a program. All alternates were developed and validated according to a common specification using independent programming teams [49, 52]. These authors conclude that that N-version programming must be used with care because the number of input cases under which most or all alternate failed coincidentally was more than expected [49, 52].

Gashi et al. [110] studied design diversity as a means for tolerating design faults of four popular off-the-shelf SQL servers. Their study is based on an analysis of the bug reports available for the SQL servers. They conclude that design diversity is effective in this category of products since none of identified bugs affected more than two products.

Findings from existing work reveal threats to the effectiveness of software that relies on design diversity to tolerate faults [49, 52, 110]. This in turn reinforces the necessity for a thorough methodology to support an assessment of the reliability of FT-compositions that leverage alternate services.

Studies on Design Diversity of Alternates

Lyu et al. [50] quantified software diversity by proposing qualitative and quantitative metrics. The first metric, assumes that diversity may be applied at the specification, design, coding and the testing phases. The intensity of diversity at each identified phase might vary according to the adoption of different implementers, languages, tools, algorithms and methodologies. The proposed metrics quantify, based on characteristics of alternate implementations, structural diversity, tough-spot diversity and failure diversity. These metrics are subsequently used as indicators of which alternates may be unsuitable for use.

Hilford et al. [51] proposed a method of incorporating diversity in the development of alternates in order to eliminate as many software faults as possible before the testing phase. They proposed the pipeline approach, in which different teams work on different phases of the development process of each alternate. According to Chen et al. [124] in order to estimate software diversity it is not enough to know only the failure probability of each single alternate. It is necessary analyse the failure distribution patterns of each single version. To analyse the failures of alternates, the authors adopted *Data flow Perturbation* and *Constant Perturbation*, which are fault injection techniques.

These previous works [50, 51, 124] cannot be adopted in the context of SOAs, since the only known information about services is their interfaces; neither source code, nor information about deployment environment are available (*i.e.* services are black boxes) [16, 60, 125].

Solutions based on Design Diversity in SOAs.

Nascimento et al. [40] present a software product line infrastructure for adaptive fault tolerance in service-oriented applications. This solution supports different types of adjudicators (*e.g.* majority election and acceptance test) and different schemes to execute the alternate services (e.g. sequential and parallel schemes). Chen [126] proposed an infrastructure, called WS-Mediator that aligns the fault tolerance techniques based on design diversity with service resilience information, that can be provided by both client application itself and third-party tools. Gonçalves and Rubira [33] described a software infrastructure that leverages voters and operates in the communication between a web service's clients and alternate service in order to provide fault tolerance. Zheng and Lyu [56] proposed an adaptive QoS-aware fault tolerance strategy based on design diversity, voters and acceptance tests for Web services. Nevertheless, for this previous work, there is an underlying assumption that alternate services can always be efficiently employed by means of diversity-based technique. As we discussed, there is no guarantee that this assumption is always true. Therefore, we feel our work is complementary to theirs. One could investigate groups of alternate services by adopting our proposed experimental setup in order to either select alternate services that are in fact able to tolerate software faults or choose to employ the most reliable NFT-service.

3.6 Summary

Several solutions propose the use of functionally equivalent services (*i.e* alternate services) to tolerate software faults in service-oriented architecture (SOA). However, given a particular requirements specification, it is unclear whether its alternate services are improving the fault tolerance of the software application. This is because the reliability of fault-tolerant composition depends upon design diversity of their alternate services to increase the probability that they fail on disjoint input spaces. One contribution of this chapter is an assessment approach to investigate design diversity of alternate services. First, based on the analysis of the output space, we check, by means nonparametric statistical tests, whether alternate services are provided by design and implementations that are sufficiently different. If they are in fact diverse, we investigate if and by how much service diversity is in fact effective for tolerating software faults. The feasibility of the proposed approach was assessed using third-party, stateless, read-only, SOAP/WSDL-based alternate services adhering to four different requirements specifications.

Based on the proposed solution, we presented the results of a novel study, in the context of SOAs, to investigate whether functionally equivalent services (*i.e.* alternate services) are able to tolerate software faults. We discussed in detail our findings and lessons learned from this study. We concluded that the benefits of diversity-based solutions applied to SOAs are not straightforward. Even when alternate services present design diversity, in some cases, this diversity might not be sufficient to improve system reliability. These results also reinforces the usefulness of our assessment framework. Existing work also reveals threats to the effectiveness of software that relies on design diversity to tolerate software faults. However, to the best of our knowledge, this is the first study assessing how effective is *service diversity* for tolerating software faults.

In Chapter 4, we present a systematic literature review of design diversity-based solutions for fault-tolerant composite services.

Capítulo 4

A Systematic Review of Design Diversity-Based Solutions for Fault-Tolerant SOAs

Uma vez que obtivemos evidências de que serviços alternativos são eficientes para tolerar falhas de software, desde que devidamente selecionados, faz-se necessário estudar as soluções existentes para composições de serviços tolerantes a falhas, a fim de *(i)* identificar as similaridades e diferenças entre as soluções existentes; *(ii)* identificar suas principais contribuições e limitações, *(iii)* prover diretrizes que apoiem a escolha de soluções apropriadas para diferentes contextos; e *(iv)* identificar oportunidades para avançar o estado da arte no que tange o projeto e implementação de composições de serviços tolerantes a falhas. Neste capítulo, descrevemos a revisão sistemática das soluções existentes baseadas em diversidade de projetos para composições de serviços tolerantes a falhas que alavancam técnicas baseadas em diversidade de projetos. O conteúdo deste capítulo é baseado no artigo publicado no *17th International Conference on Evaluation and Assessment in Software Engineering - EASE '13*, que apresenta todo o processo de preparação e execução da revisão e os resultados obtidos. Como o conteúdo deste capítulo foi extraído na íntegra de tal artigo, foi preservado o idioma original.

4.1 Overview

service.

In general terms, three major design issues need to be considered while building software fault-tolerant architectures based on design diversity, namely, *(i)* selection of alternates that are sufficiently diverse and able to tolerate software faults; *(ii)* execution of alternates; and *(iii)* selection of an adjudicator to determine the acceptability of the results obtained from the alternates [1, 127, 43]. Each design issue can be realized by a set of alternative design decisions, which, in turn, imply in different degrees of quality requirements (e.g. memory consumption, financial cost, response time and reliability). For example, alternates can be executed either sequentially or in parallel, and alternate outputs can be adjudicated by adopting different voting and acceptance algorithms [1, 127].

Nevertheless, it is unclear the extent to which existing approaches for fault-tolerant composite web services [40, 56], called *FT-compositions* for simplicity, support the above mentioned design issues of a software fault-tolerant architecture based on design diversity. Moreover, the publications regarding FT-compositions are written from different viewpoints and rely on different technical backgrounds [33, 58, 39, 34, 40]. As a result, it is hard to compare them and choose from these solutions. In this sense, we conducted a systematic literature review that sheds light on the similarities and differences among various diversity-based approaches for FT-compositions, which compose our primary studies. A taxonomy is developed to help address the three major design issues and their respective alternative design solutions, and to classify the primary studies. General remarks on the effectiveness of the design solutions are also presented, which might be a good starting point to choose proper design solutions in accordance with difference clients requirements.

In this chapter, we present details regarding the specified systematic literature review method, including, our research question; search strategy; inclusion and exclusion criteria to assess each potential primary study; data collection and data analysis. Finally, we report our main findings and identify gaps in current approaches in order to suggest opportunities for greater progress related to the design and implementation of reliable SOA-based applications.

4.2 A General Taxonomy for Software Fault Tolerance Techniques based on Design Diversity

Three major design issues need to be considered while building software fault-tolerant architectures based on design diversity, namely, selection of alternates; selection of alternate execution schemes; and judgement on result acceptability [1, 43, 127]. We define a general taxonomy, represented in Figure 4.1, for these common design issues and their different design solutions. The proposed taxonomy is adopted to discuss and classify the analysed primary studies. Both design issues and decisions were derived from the analysis of largely adopted software fault tolerance techniques based on design diversity (*e.g. Recovery Blocks*, *N-Version Programming*, *N-Self Checking Programming*, *Consensus Recovery Block* and *Acceptance Voting*) and adjudicators [47, 48, 128, 46, 129, 42, 43, 1]. We also considered the reliable hybrid pattern structure proposed by Kim and Vouk [127]. In comparison with their work, our work *(i)* identifies different types of voters and acceptance tests based on the general taxonomy of adjudicators presented by Pullum[1]; and *(ii)* explicitly distinguishes the different schemes of alternate execution (i.e. sequentially or in parallel). Different design decisions employ different measures of quality requirements [1]. These differences make each design solution suitable for a particular situation. In Section 4.2.1, we briefly compare the described design solutions and present some general remarks about their effectiveness.

The elements of the taxonomy are described in the following.

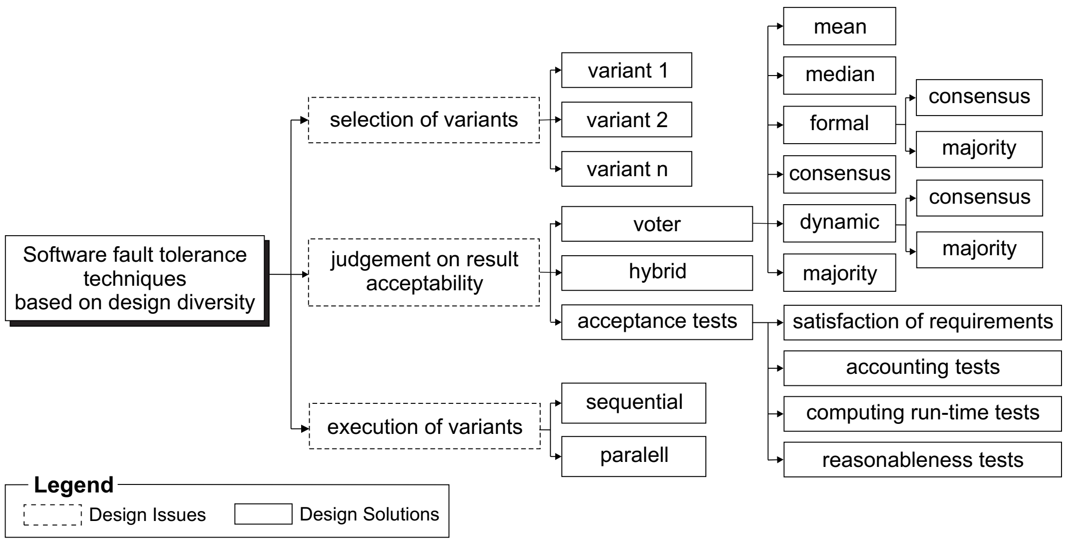

Figure 4.1: General Taxonomy of Design Issues and Solutions

Design Issue I - Selection of Alternates: The number of alternate software components can range from two alternates to n and they must be provided by different software design and implementations. The main goal of increasing diversity is to increase the probability that alternates fail on disjoint subsets of the input space [1, 42]. The reliability of the alternates should be as high as possible, so that at least one alternate will be operational at all times [1]. Finally, alternates might be chosen at different points during the software lifecycle.

Design Issue II - Execution of Alternates: Alternates can be executed either sequentially or in parallel. The execution schemes should provide all alternates with exactly the same experience the system state when their respective executions start to ensure consistency of input data [31], which be can be achieved by employing backward recovery or forward recovery (*Section 2.2.4*). Sequential execution often requires the use of checkpoints (it usually employs backward recovery), and parallel execution often requires the use of algorithms to ensure consistency of input data (it usually employs forward recovery by invoking all the alternates and coordinating their execution through a synchronization regime) [1, 100].

- **Sequential:** in implementing a sequential execution scheme the alternates are executed one at a time. Generally, in the sequential execution scheme, the most efficient alternate (*e.g.* in terms of response time or financial cost) is located first in the series, and is termed *primary alternate*. The less efficient alternates are placed serially after the primary alternate and are referred to as (secondary) alternates. Thus, the resulting rank of the alternates reflects the graceful degradation in the performance of the alternates [1].
- **Parallel:** in the parallel execution scheme, alternates are executed concurrently. The resulting outputs can be provided to the adjudicator in an *asynchronous* fashion as each version completes, or in a *synchronous* manner [1, 127].

Design Issue III - Judgement on Result Acceptability: Adjudicators, or decision mechanisms, generally come in two flavours, voters and Acceptance Tests (ATs).
Voters: Voters are based on a relative judgement on result acceptability by comparison of alternate results [1]. We present an overview of voters that are mostly described in the literature [1, 43]. We refer to Pullum [1] for further details on voter procedures and pseudocodes.

- **Exact Majority Voter:** The exact majority selects the value of the majority of the alternates as its presumably correct result [130]. This voter is also known as the m-out-of-n voter [1]. The agreement number, m, is the number of versions required to match for system success [96]. The total number of alternatives, *i.e.* n, is rarely more than 3. Consequently, the majority voter is generally seen as a 2-out-of-3 voter.
- **Consensus Voter:** This voter allows the selection of a consensus or set of matching alternate results as the adjudicated result if no majority exists [1]. That is, this voter is a generalization of the majority voter [53].
- **Formal Consensus** and **Majority Voter**: The *Formal Consensus* and *Majority* voter is a variation of, respectively, the consensus and the exact majority voters [1]. Basically, the formal voter uses a comparison tolerance indicating the maximum distance allowed between two correct output values for the same input. In this way, alternates results that are different, but quite close together, are the adjudicated correct answers.
- **Median Voter:** The median voter selects the median of the alternate' output values as its adjudicated result. Alternate outputs must be in an ordered space [1].
- **Mean an Weighted Voter:** The mean and weighted voter selects, respectively, the mean or weighted average of the alternates' output values, which are in an ordered

space, as the adjudicated result [131]. Additional information related to the trustworthiness of the alternatives might be used to assign weights to the alternate outputs, if using the weighted average voter [1].

• **Dynamic Majority** and **Consensus Voters:** Unlike previously described voters, dynamic voters are not defeated when any alternate fails to provide a result [1]. Dynamic majority and consensus voters operate in a way similar to, respectively, majority and consensus voters, with the exception that dynamic voters can handle a varying number of inputs [1]. When the dynamic voter adjudicates upon two results, a comparison takes place. When comparing, if the results match, the matching value will be output as the correct result. Otherwise, no selected output will be returned.

Acceptance Tests (ATs): ATs relies on an absolute judgement with respect to a specification [1]. With ATs, only one alternate is executed at a time. The AT is responsible for checking whether the produced result is correct. In case it is not, another alternate is executed until a correct result is obtained, if possible. Pullum [1] claims that ATs can also be used not to determine if a result is correct, but to indicate if something blatantly incorrect has resulted from the execution of an alternate.

• **Acceptance Tests Based on Satisfaction of Requirements**: When conditions that must be met at the completion of alternate execution are used to construct AT, we have a 'satisfaction of requirements' type AT [1]. These conditions might arise from the problem statement of the software specifications.

• **Accounting Tests**: Accounting ATs are suitable for transaction-oriented applications with simple mathematical operations [1]. For example, when a large number of records are reordered or transmitted, a tally is made of both the sum over all records and the total number of records of a particular data field. These results can be compared between the source and the destination to implement an accounting check AT [1].

• **Computer Run-Time Tests**: Run-time tests detect anomalous states such as overflow, undefined operation code, underflow, write-protection violations or end of file [1].

• **Reasonableness Tests**: These ATs are used to determine if the state of an object in the system is reasonable (*e.g.* precomputed ranges or expected sequences of program states [1]).

Hybrid Adjudicators: A hybrid adjudicator generally incorporates a combination of AT and voter characteristics. For example, alternate results are evaluated by an AT, and only accepted results are sent to the voter [1].

4.2.1 A Comparison of Design Solutions

To provide an exhaustive discussion into the performances of the described design solutions is outside the scope of this work. We refer to Pullum[1] for a list of references for efficiency and capability investigations on design solutions of software fault tolerance techniques based on design diversity ([1]:page 120:Table 4.2). As emphasized by Pullum, each study has different underlying assumptions, thus it is difficult to compare the results across experiments [1]. The fault assumptions used in the experiments and studies are important and if changed or ignored can alter the interpretation of the results [1]. Therefore, the reader is encouraged to examine all references identified by Pullum [1] (page 120:Table 4.2) for details on assumptions made by the researchers, experiment design, and results interpretation. We summarize some of the findings from these empirical studies on design diversity, in particular the findings related to non-functional characteristics of the described design solutions (*Figure 4.1*). The analysis of these characteristics might be a good starting point to choose a proper design solution and to specify the system fault model (*Section 2.2*). The fault taxonomy for service-oriented architectures proposed by Bruning et al. [132] can also be useful to define the fault model.

Firstly, as already discussed (*Section 3.4*), it is essential to select alternates that are sufficiently diverse in order to decrease the probability of occurrence of coincident failures [49, 50, 51, 52, 133]. With respect to the execution of alternates, in the parallel scheme, there is an underlying assumption that sufficient hardware resources are available to enable the execution of alternates concurrently. Even with sufficient parallelism, the execution time of this scheme will be constrained by the slowest version - there may be a substantial difference in the execution speeds of the fastest and slowest version because of the need to generate independent designs [43]. When alternates are executed in parallel, there is also a synchronization time overhead. The time required to execute alternates in a sequential way will range from the execution time of the primary alternate (if acceptance tests are employed and the primary result is acceptable) to the sum of execution time of all alternates (*e.g.* if all alternate results are subjected to ATs [37]). Nevertheless, this time will normally be constrained by the execution time of the primary alternate [43]. Furthermore, under some circumstances, a specific execution scheme is not applied, *e.g.*, when there is a processing cost charged for the use of alternate services, invoking them in parallel might incur in greater actual cost (*i.e.* in sequential schemes, not all alternates are necessarily executed) [34].

The adjudicator would run its decision-making algorithm on the results and determine which one (if any) to output as the presumably correct result. Just as we can imagine different specific criteria for determining the 'best' item depending on what that item is, so we can use different criteria for selecting the 'correct'or 'best' result to output. The probabilites of activation of related faults between alternates are likely to be greater for voters than for acceptance tests (ATs) [134]. If one could develop a perfect AT and a perfect voter and if we assume failure independence, then ATs with three alternates is a better solution than the three-alternate voting system [130]. In general, ATs are more difficult to construct in practice because they are strongly application-dependent and because it is not always possible to determine a criterion to judge variant results [1, 106]. As a consequence, voting is a more useful technique in a practical setting, because voting adjudicators are easier to develop [1, 100].

The exact mojority voter is most appropriately used to examine integer or binary results, but can be used on any type of input [95]. The majority voter has a high probability of selecting the correct result value when the probability of an alternate failure is less than 50% and the number of processes, n, is 'large ' [135] (Blough and Sullivan [135] used n = 7 and n = 15 in the study). However, when the probability of an alternate failure exceeds 50%, then the majority voter performs poorly [135]. In fact, all voters have a high probability of selecting the correct result value when the probability of alternate failures is less than 50% [135]. A median voter can be defined for alternate outputs consisting of a single value in an ordered space (*e.g.* real number) [1]. Median voter is a fast algorithm and is likely to select a corret result in the correct range [135]. If it can be assumed that, for each input value, no incorrect result lies between two correct results, and that a majority of the replica outputs are correct, then the median voter produces a correct output [1].

Consensus voting is more stable than majority voting and always offers reliability at least equivalent to majority voting [53, 130]. Nevertheless, in terms of implementation, the consensus voting algorithm is more complex than the majority one, since the consensus voting algorithm requires multiple comparisons [1]. Blough performed a study on the effectiveness of voting algorithms [135]. He states that the median voter is expected to perform better than the mean voting strategy. He also shows the overall superiority of the median strategy over the majority voting scheme [135]. Furthermore, under circumstances in which some or all alternates might not produce their results (*e.g.* some or all alternates not providing their results within the maximum expected time frame; catastrophich failure of some or all of the alternates), dynamic voters are the best option since they can process zero to n inputs [1]. Finally, acceptance tests, exact majority, consensus and dynamic voters can process any type of alternate outputs, while the remaining adjudicators must receive inputs in a ordered space [1].

We summarize some details of the described voters in Table 4.1, which is based on a summary table fashioned by Pullum [1]. We have added the type of alternate results the voter is able to judge. The table states the resulting output of a fault tolerance technique, given the type of alternate results provided to the voter and the type of voter. To use this table, we must consider the primary concerns surrounding the software's application and details about the output space. For example, if safety is the primary concern, it is recommended to adopt the voter that would rather raise an exception and produce no selected output than present an incorrect output as presumably correct one (*e.g.* the exact majority or dynamic majority voters) [1]. If the primary goal is to avoid cases in which the voter does not reach a decision, i.e., *an* answer is better than no answer, than it is sufficient to adopt the voter that reaches a 'No output' result least often (*e.g.* the median voter) [1]. As emphasized by Pullum [1], based on this criterion, exact majority voter, formal majority voter, and dynamic majority voter can be considered the safest voters, because they produce incorrect output 'only' in cases where most or all of the variants produce identical and wrong results.

Table 4.1: Voter Results Given Details About Alternate Output Space([1] - page 310)

ALTERNATE RESULTS	**VOTER**							
	Exact Majority	**Median**	**Mean**	**Weighted Average**	**Consensus**	**Formal Majority**	**Dynamic Majority**	**Dynamic Consensus**
All outputs identical and correct	Correct	Correct	Correct	Possibly correct	Correct	Correct	Correct	Correct
Majority Identical and correct	Correct	Correct	Possibly correct	Possibly Correct	Correct	Correct	Correct	Correct
Plurality identical and correct	No output	Possibly correct	Possibly Correct	Possibly correct	Correct	No output	No output	Correct
Distinct outputs, all correct	No output	Correct	Possibly correct	Possibly correct	No output	No output	No output	No output
Distinct outputs, all incorrect	No output	Incorrect	Possibly incorrect	Possibly incorrect	No output	No output	No output	No output
Plurality identical and wrong	No output	Possibly incorrect	Possibly incorrect	Possibly incorrect	Incorrect	No output	No output	Incorrect
Majority identical and wrong	Incorrect	Incorrect	Possibly incorrect	Possibly incorrect	Incorrect	Incorrect	Incorrect	Incorrect
All outputs identical and wrong	Incorrect	Incorrect	Incorrect	Incorrect	Incorrect	Incorrect	Incorrect	Incorrect
ALTERNATE RESULT TYPE	Any Type	In an ordered Space	In an ordered Space	In an ordered Spac	Any Type	Floating-point arithmetic	Any Type	Any Type

The voter outputs are [1]:

- Correct: The voter outputs a correct result;
- Possibly correct: The voter outputs a result that may be correct;
- Possibly incorrect: The voter outputs a result that may be incorrect;
- Incorrect: The voter outputs an incorrect result;
- No output: The voter does not output a result; an exception is raised.

4.3 Research Method

This study has been conducted as a systematic literature review based on guidelines proposed by Kitchenham and Charters [74]. The guidelines cover three phases of a systematic review: planning, executing and reporting the review. In our work, the goal of the review is to provide a better understanding of diversity-based approaches for FT-compositions, which compose our primary studies [74].

4.3.1 Research Question

This work aims to answer the following research question.
RQ What design issues and respective design solutions related to fault tolerance based on software diversity (Figure 4.1) are being supported by existing approaches for FT-compositions?

To address the research question, we classify the primary studies using the proposed taxonomy for software fault tolerance (Figure 4.1) and present a discussion regarding solutions proposed to address the listed design issues.

4.3.2 Search Process

Searches for primary studies were executed using searching databases of software engineering research that met the following criteria [136]:

- Contains peer-reviewed software engineering journals articles, conference proceedings, and book chapters.
- Contains multiple journals and conference proceedings, which include volumes that range from 2000 to 2012.
- Used in other software engineering systematic reviews (e.g. [137, 136, 138, 139]).

The resulting list of databases was: *(i)* ACM Digital Library; *(ii)* IEEE Electronic Library; *(iii)* SpringerLink; *(iv)* Scopus; and *(v)* Scirus (Elsevier).

A search string was created to extract data from each database. We adopted various combinations of terms from *(i)* the main purpose of this review; *(ii)* the research question [74, 140] and *(iii)* meaningful synonyms and alternate spellings. The resulting search string is summarized in Table 4.2. Whenever it was necessary, this search string was decomposed into several search terms (*e.g.* Recovery Block AND Service-Oriented Architectures) due to restrictions imposed by some of the search engines.

Table 4.2: Search String

(*fault tolerance OR diversity OR fault-tolerant OR redundancy OR Recovery block OR N-Version Programming OR Distributed Recovery Blocks OR N Self-Checking Programming OR Consensus Recovery Block OR Acceptance Voting OR dependability OR dependable OR reliable OR reliability*) $< AND >$ (*service-oriented architecture OR SOA OR service computing OR SOC OR web services*)

4.3.3 Study Selection

The database searches resulted in a large number of candidate papers. We adopted study selection criteria to identify those studies that provide direct evidence about the research question.
Inclusion Criteria:

• Approaches based on software diversity for FT- compositions, specifically focused on web services, that support one or more issues identified in the taxonomy (Figure 4.1), *i.e.* selection of alternate services, execution of alternates and judgement on result acceptability.

Exclusion Criteria:

• Solutions for reliable SOA-based applications employing solely replicas of identical services - although the adoption of identical replicas can improve system availability, they are not able to tolerate *software* faults [1].

• Solutions for fault-tolerant SOAs based on data diverse software fault tolerance techniques.

• Solutions for fault-tolerant SOA relying solely on exception handling - alternate services are not employed as part of the exception handling mechanism.

• Duplicate reports of the same solution - when several reports of the proposed solution exist in different papers the most complete version of the study was included in the review.

• Short papers, introductions to special issues, tutorials, and mini-tracks.

• Studies presented in languages other than English.

• Papers addressing empirical studies on fault tolerance based on design diversity applied to *SOAs* - not proposing any particular solution to employ FT-compositions.

- Grey literature, that is, informally published written material.

These criteria were applied as performed in [74, 136]:

1. Reading the title in order to eliminate any irrelevant papers.
2. Reading the abstract and keywords to eliminate additional papers whose title may have fit, but abstract did not relate to the research question.
3. Reading the introduction and, whenever it is necessary, the conclusion to eliminate additional papers whose abstract was not enough to decide whether the inclusion/exclusion criteria are applicable.
4. Reading the remainder of the paper and including only those that addressed the research question.

4.3.4 Data Collection and Synthesis

We adopted a data extraction form to collect all the information needed to address the review question [74]. The contents of the designed data form are composed by [74, 140]: the source (*e.g.* journal, conference) and full reference; date of extraction; summary of the proposed solution; supported design issues/solutions (Figure 4.1); and space for additional notes.

4.4 Results

In this section, we present the results obtained.

4.4.1 Search Results

The searches returned thousands of papers that were filtered down to 17 primary studies [36, 37, 141, 142, 112, 38, 40, 143, 59, 144, 41, 145, 34, 33, 58, 146, 147]. In the solutions by Dillen et al. [38], Santos et al. [59] and Mansour and Dillen [146], despite each service being named a replica by the authors, these solutions were designed to tolerate different responses by means of adjudicator mechanisms. This suggests these solutions could be also implemented as a diversity-based solution. Therefore, these solutions were included as primary studies.

Although we identified 23 articles by this search process, the articles [56, 148] are short versions of another article [34], the article [143] is also an extended version of the article [149]. The study presented by Gorbenko et al. in [125] and by Nascimento et al. [133] are based, respectively, on the solutions proposed in [145] and in [112] - therefore, we consider the proposed solutions, i.e. [145, 112], as primary studies. Xu [150] examined challenges in the fields of dependability and security that need to be addressed carefully in order to provide sufficient support to enable service-oriented systems to offer non-trivial Quality of Service guarantees. Then, Xu presents several advanced techniques developed at the University of Leeds to achieve dependability and security in service-oriented systems and applications, including, the solution proposed by Townend et al. [142]. Therefore, only the work by Townend et al. [142] is included as a primary study. Milanovic and Miroslaw [151] also highlight the use of techniques to tolerate software faults in SOA, including techniques based on software diversity (e.g. N-Versions); however, no specific solution is proposed in this direction.

4.4.2 A Classification of the Primary Studies

In Table 4.3, we present the summary of the design solutions supported by the analysed primary studies. Each primary study was classified as follows:

Y(yes), the design solution is supported by a primary study; **N(no)**, no information at all about the design solution is specified; **U(unknown)** according to the authors the design issue is supported, however, what design solutions are supported cannot be readily inferred.

It is important to emphasize that the Table 4.3 was fashioned to show which design issues have been addressed by existing approaches for FT-compositions. It might be meaningless to rank the primary studies based solely on their 'quantity of *Yes*' since the studies present different purposes (e.g. some solutions are mainly focused on supporting the selection of alternate services [112, 142], while other ones are focused on executing alternate services [33, 58]). Although it's difficult to compare these solutions, Section 4.2.1 presents non-functional characteristics related to the design solutions supported by the authors - thus supporting researchers' decision making when selecting design solutions more adjusted to different clients requirements.

In the next subsections, we discuss the answers to our research question **RQ** (*Section 4.3.1*) and present the main proposed solutions regarding the design issues (Figure 4.1). Because of space limitations, summaries of solutions are representative rather than exhaustive.

4.4.3 Selection of Alternate Services

The aims of selecting alternates are twofold: *(i)* to increase the probability of selecting alternates that are provided by different designs and implementations; *(ii)* to determine an appropriate degree and/or selection of alternate services targeting an optimal trade-off

Table 4.3: Classification of the Primary Studies

PRIMARY STUDIES	Gotze et al. [36]	Buys et al. [37]	Nourani et al. [152, 141]	Townend et al. [142, 150]	Nascimento et al. [112, 133]	Dillen et al. [38]	Nascimento et al. [40]	Abdeldjelil et al. [149, 143]	Santos et al. [59]	Looker et al. [144]	Kotonya and Stephen [41]	Gorbenko et al. [145, 39]	Zheng and Lyu [148, 56, 34]	Goncalves and Rubira [33]	Chen and Romanovsky [58]	Mansour and Dillon [146]	Laranjeiro and Vieira [147]
DESIGN ISSUES AND SOLUTIONS																	
Selection of Alternate Services	Y	Y	N	Y	Y	Y	Y	Y	N	N	Y	Y	Y	N	Y	Y	N
Execution of Alternate Services	U	Y	Y	U	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Sequentially	-	Y	Y	-	-	Y	Y	Y	N	N	N	Y	Y	Y	Y	Y	Y
Parallely	-	Y	Y	-	-	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Judgement on Result Acceptability	Y	Y	Y	Y	N	Y	Y	Y	Y	Y	Y	U	Y	N	N	Y	Y
Voter	Y	Y	Y	Y	-	Y	Y	Y	Y	N	U	-	Y	-	-	U	Y
Median Voter	N	N	N	N	-	N	N	N	N	N	-	-	N	-	-	-	N
Mean Voter	N	N	N	N	-	N	N	N	N	N	-	-	N	-	-	-	N
Majority Voter	N	N	Y	N	-	N	Y	N	N	Y	-	-	Y	-	-	-	N
Consensus Voter	N	N	Y	N	-	N	N	N	Y	N	-	-	N	-	-	-	Y
Formal Voter	N	N	N	N	-	Y	N	Y	N	N	-	-	N	-	-	-	N
Dynamic Voter	Y	Y	Y	Y	-	N	N	N	N	N	-	-	N	-	-	-	N
Acceptance Tests	N	N	U	N	-	Y	U	N	N	N	N	-	N	-	-	U	N
Satisf. Requir.	-	-	-	-	-	Y	-	-	-	-	-	-	-	-	-	-	-
Account. Tests	-	-	-	-	-	N	-	-	-	-	-	-	-	-	-	-	-
Comp. Run-Time Tests	-	-	-	-	-	N	-	-	-	-	-	-	-	-	-	-	-
Reasonableness Tests	-	-	-	-	-	N	-	-	-	-	-	-	-	-	-	-	-
Hybrid Adjudicator	N	N	Y	N	-	N	N	N	N	N	N	-	N	-	-	N	N

between reliability measures as well as performance-related factors such as timeliness, cost and resource consumption. In general, alternates have been chosen at different points during the software lifecycle. For instance, they can be chosen at design time by the engineer, configured manually once the software is deployed, or even be discovered and selected at run-time by the software itself.

The solutions by Townend et al. [142] and by Nascimento et al. [112] address the issue *(i)*: ensuring designs are diverse. Townend et al. [142] aims to detect diverse designs during runtime. This is achieved by monitoring previous results and flow of data from a alternate service using interaction provenance in order to reveal evidence that two alternates share similar services or workflows. Such evidence may include matching common-mode failures that have propagated back from two alternate services. Nascimento et al. [112] propose an experimental setup to investigate, from clients' viewpoint and by means of statistical tests, whether alternate services present a difference in their outputs and their failure behaviours. Their solution also investigates if and by how much the use of FT-composition improves reliability when compared to a single non-fault-tolerant service [112].

Regarding issue (ii): the degree of alternates necessary, the responsibility for deciding which alternates are necessary may be based on different measurements and different points throughout the execution. Buys et al. [37] and Dillen et al. [38] propose a fault tolerance strategy that autonomously changes the amount of redundancy or the selection of alternate services. The architecture proposed by Buys et al. [37] bases this decision upon the current execution context and clients' requirements at the time of request. They propose a measure to infer combinations of alternate services that are, in fact, effective. This measure quantifies the historical effectiveness of each alternate service by penalising or rewarding it when it disagrees or complies with the majority decision respectively. Gotze et al. [36] propose a solution where every atomic service provides information about its dependability attributes and every composite service has to provide additional information about their external services that are used to provide the desired functionality. The calculated dependability attributes and probability values of the resulting composite service are then used to manually optimize the composite service towards the user's expectations [36], differing from the solution by Buys [37] and Dillen et al. [38] in which alternate services are automatically selected.

The solution by Nascimento et al. [40], Abdeldjelil et al. [149, 143], Chen and Romanovsky [58] and Zheng and Lyu [148, 56, 34] allows the dynamic selection of alternate services based on a priority schemes where client defines requirements in terms of QoS. QoS values are updated by monitoring procedures in [40, 149, 143, 58] and by encouraging users to contribute their individually-obtained QoS information of the target Web services in [148, 56, 34]. The solution by Kotonya and Stephen [41] and by Mansour and Dillon [146] support a QoS matching scheme that will prioritize services based on reliability and performance metrics. These metrics are not shared with the client, i.e., a client cannot express preference for specific QoS metrics. The solution by Gorbenko [145] monitors dependability attributes and selects an appropriate service based on them. However, the motivation behind this solution was to manage service upgrades, i.e. switching from an old version to a new version online when the level of dependability of the new service is acceptable.

4.4.4 Execution of Alternate Services

Both parallel and sequential execution schemes have been addressed. The solution by Nascimento et al. [112] does not aim to support execution schemes. The solutions by Gotze et al. [36], Townend et al. [142] supports execution of alternates; however, they do not specify how alternates might be executed in parallel, sequentially or both. Most solutions support both execution schemes [37, 141, 38, 40, 143, 145, 33, 58, 147, 148, 56, 34, 146], the solution by Looker et al. [144], Santos et al. [59] and by Kotonya and Stephen [41] support the parallel execution of alternates. An important characteristic of the execution scheme proposed by Laranjeiro and Vieira [147] is that it can use functionally equivalent web services that have different interfaces, providing developers with more options to build their solutions (*i.e.* input/ output adapters). However, their solution operates on a static connectivity mode, requiring static generation of local proxy classes for each alternate service [147]. The solution by Abdeldjelil et al. [143] allows the execution of alternate services that present diversity also in their interfaces and results by mapping operations and parameters in the domain ontology.

4.4.5 Judgement on Result Acceptability

A variety of decision mechanisms were found amongst the target papers. The main advantage of diversity-based solutions is that they allow for alternate service responses to be compared using a large choice of voter and acceptance test techniques. For instance, majority, consensus, formal and dynamic voters are addressed by some of the existing approaches for FT-compositions. Acceptance tests (ATs) are also supported; however, as they are strongly application-dependent there are fewer approaches supporting ATs.

Majority Voters

Majority voters are supported by Nourani et al. [152, 141], Looker et al. [144], Nascimento et al. [40] and Zheng and Lyu [148, 56, 34]. The solution by Nourani et al. [152, 141] supports the implementation of FT-compositions by means of the WS-BPEL. In this sense, the authors present details on the prototype implementation by identifying the BPEL structured activities adopted. For the adjudication mechanism, which is also implemented as web service and invoked from a BPEL process, the authors mention the adoption of voters, including majority, dynamic and consensus ones, acceptance test and hybrid adjudicator. Nevertheless, no details on voting procedures are provided, except for the majority one. Differently from other solutions, they adopt diversity to allow two majority voting mechanisms to be applied, if necessary, to the same input set. In BPEL process, if for any reason the voter faces faults or the result shows the absence of consensus, the second version of voter is invoked using the same inputs. This avoids the voting process to become a single point of failure [152, 141]. The Web Service-Fault Tolerance Mechanism (WS-FTM) proposed by Looker et al. [144] supports the majority voter that allows generic result comparison. Nascimento et al. [40] and Zheng and Lyu [148, 56, 34] do not present additional information on voting procedures. We emphasize, however, that the solution by Zheng and Lyu [148, 56, 34] supports different fault tolerance strategies and the authors described in detail a dynamic fault tolerance strategy selection algorithm.

Consensus Voters

Nourani et al. [152, 141] argues that their solution support the consensus voters. In the solution by Santos et al. [59], there is a component responsible for arbitrating the adjudicated output based on the output with the highest number of occurrences. Therefore, we inferred that their decision mechanism is based on the consensus voter. The benefits of diverse components is not only limited to the services, in the solution proposed by Laranjeiro and Vieira [147] the voter protocol supports two voting mechanisms to be utilized at the same time: a unanimous voter and a consensus voter. Their solution also supports an evaluation mechanism that performs a continuous assessment of the quality of services. During the voting process, if an impasse occurs the QoS values of the alternates are used to select one response. According to the authors, the impasse occurs whenever different alternate results present the same number of occurrences [147]. However, in their solution, it is not clear how they judge alternate result acceptability when alternates are executed sequentially.

Formal Voters

When dealing with greater variability in alternate service interfaces, a straightforward consensus voting mechanism may not suffice. Some solutions accept a certain amount of variability and agree that a consensus is formed with multiple equivalent results. The solution by Dillen et al. [38] supports the formal consensus voting, in the paper, called plurality voting. For each invocation of the scheme, the alternate services will be partitioned based on the equivalence of their results, and the result associated to the largest cluster will be accepted as the correct result [38]. Abdeldjelil et al. [149, 143] describe a specific voting algorithm, called equivalence vote, that will decide if multiple concurrent service responses are equivalent or not. This is based on a pre-agreed amount of deviation for answers to be considered equivalent. Their algorithm requires as input an ordered set of equivalent results and the functional deviation to authorize among them.

Dynamic Voters

Dynamic consensus and dynamic majority voters are supported by a number of solutions [36, 37, 152, 141, 142]. This may be more suitable for applications where it is acceptable for some alternate services to fail and an answer still be returned from the remaining alternate services. Due to the intricacies of dynamic decision mechanisms defined in our taxonomy some classifications in our review have been changed from the classifications used in the target papers. The solution by Gotze et al. [36] define three FT-protocols, namely, *one*, *any* and *majority*. Only the majority operator allows for the enhancement of reliability and availability. The remaining operators are mainly focused on achieving high levels of availability and are out of the scope of this work. They define the majority as an operation that schedules the same request to all defined services. Afterwards this operation uses the results of all services that did not fail and chooses the most common result. Based on this general definition, we inferred that their solution (according to our taxonomy) in fact supports the dynamic consensus voter instead of the majority one that has been specified by the authors [36].

According to Buys et al. [37], their solution, called A-NVP composite, supports majority voting. However, for similar reasons to the Gotze et al. classification we decided their solution instead supports a dynamic majority voter. Moreover, the response latency of the A-NVP composite is guaranteed not to exceed a maximum response time defined by the client. If no absolute majority could be established before the maximum response time, an exception will be issued to signal that consensus could not be found [37]. In addition, according to Townend et al. [142], their solution supports different types of voting algorithms to choose from. However, we inferred that the dynamic consensus voting is the only decision mechanism supported. The voter is dynamic as it can process 0 to n results, where n is the number of executed alternate services. Specifically, the voting discards results of any alternate service whose weighting, which is defined based on the confidence of that service returning a correct result, falls below a user-defined value, and performs consensus voting on the remaining results.

Acceptance Tests

Acceptance tests give the client the freedom to specify accepted values according to pre-defined requirements. Nascimento et al. [40] propose a solution that supports a mechanism responsible for the error-processing technique, which, in turn, supports acceptance tests, voters and comparisons. The solution by Dillen et al. [38] has been designed so that acceptance tests are no longer hardwired within the FT-composition. ATs can be configured at runtime through a parameterised assertion holding an $XPath$ expression that will be used to assess the acceptability of the response returned by each of the probed alternates. The solution by Mansour and Dillon [146] propose a scheme to combine alternate web services into parallel and serial configurations with centralized coordination. In this case, the broker has an acceptance testing mechanism that examines the results returned from a particular web service. The acceptance test is conducted using the broker, which might be a single point of failure. To increase the reliability of the broker introduced in their systems and mask out errors at the broker level they suggest a modified general scheme based on triple modular redundancy and N-version programming, which also includes a voting algorithm. The ATs could be specified as examining a post-condition or inalternate association with the service. The solution by Zheng and Lyu [148, 56, 34] and by Chen and Romanovsky [58] supports recovery block strategy (RB), nevertheless, they do not mention any acceptance tests, the adjudicator employed in RB.

Other Remarks

The solution by Gorbenko et al. [145] supports an adjudicator, whose type is not explicitly specified, that is also responsible for reconfiguration (switching the releases on or off), recovery of the failed releases and for logging the information which may be needed for further analysis [145]. Extensibility is an important feature, and is important in SOA architecture when the context and available services are constantly changing. In the solution by Kotonya and Stephen [41] different adjudicators might be plugged to their solutions. Also, new web services may be discovered and combined with the existing set of services. A particularly robust protocol detailed in this paper, Andros, provides a three-step consensus and authentication solution to tolerate Byzantine faults at a trade-off with system resources. Chen and Romanovsky [58] claim that although N-Version programming techniques require voting on results, in a real world application, it is not always possible to vote on results received from different services. In this sense, their solution for fault-tolerant SOAs supports well defined extension-points in which voting implementation might be included. The solution by Gonçalves and Rubira [33] encapsulates the WS-Mediator proposed by Chen and Romanovsky [58], therefore, it is possible to include voting implementations in extension-points of the WS-Mediator.

4.5 Discussion

Related to the selection of alternate services, we identified its two main purposes, to select diverse alternates and to determine an appropriate degree and/or selection of alternate services. We should emphasize that these two different purposes are complementary. This is because the reliability of fault-tolerant compositions depends upon design diversity of their alternate services to increase the probability that they fail on disjoint input spaces. For most of the proposed approaches for FT-compositions there is an underlying assumption that alternate services can always be efficiently employed by means of diversity-based techniques [148, 56, 34, 41, 36]. However, Nascimento et al. [133] presents an empirical study to investigate whether alternate services are able to tolerate software faults. They concluded that the benefits of diversity-based solutions applied to SOAs are not straightforward. Even when alternates seem to present design diversity, in some cases, this diversity might not be sufficient to improve system reliability. That is, the chosen set of alternates will impact on the success of the FT-strategy used (*Section 3.4*).

Runtime decisions regarding which alternate services are used require trade-offs according to which specific QoS attributes to use, and their feasibility of obtaining them. There are important considerations for delegating QoS responsibility to different parts of the architecture. It may be less process intensive to require each atomic service to provide quality measurements of themselves, however, lack of trust or need to ensure data integrity may mean that QoS is monitored from the client. For instance, many approaches measure availability by monitoring the alternate service 'heartbeat' - this is certainly feasible with most available services. However, a particular challenge for fault tolerance is to find a feasible way of measuring another QoS attributes, for example, the security of potential services, an issue tackled by Gotze et al. [36] where various levels of service transparency are taken into account. Moreover, while many solutions provide means of optimizing service selection once the alternate services have been chosen, we feel there is a growing need to integrate tests to ensure a specific service as being, indeed, suitable as a candidate alternate service. This is particularly needed with growing capabilities of fault-tolerant SOA in terms of autonomic searching, discovering and selection of alternate services.

With respect to the execution scheme, both parallel and sequential execution schemes have been addressed. This is particularly important, as the type of execution scheme will affect important QoS attributes such as execution time and resource consumption (*Section 4.2.1*). Related to the decision mechanisms, it is important to notice that the expected behaviour of alternate services is likely to affect the complexity of the chosen decision mechanism. For example, to choose among voting algorithms presented in the proposed taxonomy (Figure 4.1), we should consider the primary concerns surrounding the software's application and details about the output space (*Section 4.2.1*). As we can observe in the proposed classification (Table 4.3) there are still some adjudicators not addressed by current approaches for FT-compositions, e.g., median, mean voters and acceptance tests. One can claim that existing solutions might be easily extended.

On one hand, when looking beyond the implementation of solely the decision mechanism, we can also find interesting architectural solutions that provide additional functionality at this point. For instance, by defining key extension points or ability for pluggable FT strategies enhances the flexibility and interoperability of fault-tolerant SOA design. On the other hand, the authors do not explicitly describe how to extend their solutions, for example, what interfaces must be implemented when inserting a custom adjudicator [40, 41, 147, 58]. Secondly, in general, adjudication procedures are marginally described, for example authors do not specify which type of alternate outputs their solutions are able to process or how to navigate through elements and attributes in messages returned by the alternate services in order to adjudicate the acceptability of specific fragments from these messages [38, 40, 59, 41]. In other words, most authors do not specify clear guidelines on the reuse and, in particular, customization of their decision mechanisms in practical settings.

In addition, related with the decision mechanism, many FT-protocols within diversity-based fault tolerance solutions frequently selected results based on properties other than the actual response values, such as response time or likelihood of failure. This perhaps reflects the reality that even when reliability of results is uncertain, the fastest response time remains one of the main sought-after service qualities. Finally, it is interesting to observe that although various design issues and respective design solutions related to software fault tolerance techniques have been supported, they are spread among existing approaches for FT-compositions. That is, there is no a single solution able to cope with conflicting client requirements by employing at the same time a wide variety of schemes to select and execute alternate services and to determine the adjudicated result from the alternate services. There is a lack of solutions able to bring out a set of closely related fault tolerance techniques based on design diversity in close accordance with customers' requirements and high-level policies (e.g. to adopt a fault tolerance technique based on parallel execution scheme for better response time).

4.6 Threats to Validity

We identified some possible threats to validity [75] and the measures we took to reduce the risks.

Internal Validity: In terms of internal validity, our study is based on 17 papers that matched our criteria (*Section 4.3.2*). This number is not high, nevertheless, it is representative of this area of research [140, 153]. To mitigate this risk, we adopted a search strategy that aims to detect as much of the relevant literature as possible [74]. Nevertheless, the size of the sample should be kept in mind when assessing the generality of our results.

Construct validity: We identified two threats to construct validity: the study selection and data extraction are error-prone activities. Related to the study selection, this activity was performed by one of the researches at two different points in time, thus reducing the risk of having the inclusion/exclusion criteria applied inconsistently. Related to the data extraction, data might have been extracted in an inconsistent manner and to reduce this risk, as suggested by Kitchenham and Charters [74], all primary studies were assigned to one of the researchers, responsible for extracting the data [74]. Another researcher was asked to perform data extraction on a subset of primary studies chosen at random (for instance, on 6 studies). Data from the researches were compared and disagreements were resolved by consensus among researchers.

Conclusion validity: We identified one threat to conclusion validity, which is the reliability of the taxonomy itself used to classify the primary studies. To mitigate the risks of employing an inadequate taxonomy, before it was built, we had analysed the domain knowledge of software fault tolerance techniques based on design diversity in depth (e.g. [47, 48, 128, 46, 129, 42, 43, 1]).

4.7 Related Work

We are not familiar with any work that surveys diverse fault-tolerant SOAs. As a consequence, we address, in turn, work related to literature review of fault tolerance techniques in general.

Garcia et al. [154] present a comparative study of exception handling mechanisms for building dependable object-oriented software. The authors define a taxonomy to help address main basic technical aspects for a given exception handling proposal. By means of the proposed taxonomy, the authors survey various exception mechanisms implemented in different object-oriented languages, evaluates and compares different designs. Our classification of software fault tolerance solutions is also based on a general taxonomy of design issues. However, compared to their work, we do not provide a rating of the primary studies according to a quality assessment.

Carzaniga et al [155] identify some key dimensions upon which they define a taxonomy of fault tolerance and self-healing techniques in order to survey and compare the different ways redundancy has been exploited in the software domain. These are the intention of redundancy (deliberate or opportunistic), the type of redundancy (code, data, environment), the nature of triggers and adjudicators that can activate redundant mechanisms and use their results (preventive - implicit adjudicator or reactive or- implicit/explicit adjudicator), and lastly the class of faults addressed by the redundancy mechanisms (Bohrbugs or Heisenbugs). The proposed taxonomy is used to classify well known techniques, for example, N-version programming, exception handling and data diversity. The concepts presented in their taxonomy and the ones presented in the taxonomy we employed are orthogonal. In fact, our classification is performed in lower level of abstraction.

Ammar et al. [156] propose a survey of the different aspects of system fault tolerance and discuss some issues that arise in hardware fault tolerance and software fault tolerance. In this context, the authors distinguish information, spatial and temporal redundancy; present the three fundamental concepts of fault tolerance (i.e. failure, error, fault); describe the four steps of fault tolerance (i.e. error detection, damage assessment, error recovery, and fault removal) and relate these to the differences of redundant techniques for handling hardware as well as software faults. According to the authors, since redundancy may be used under a variety of forms to achieve fault tolerance, the design of a fault-tolerant system involves a set of trade-offs between redundancy requirements (imposed by the need for fault tolerance) and requirements of economy (economy of the process, and the product) [156]. The authors also emphasize that program fault tolerance is no panacea, like almost everything in software engineering. We refer to their work for an interesting discussion on reasons to support this claim.

Florio and Blonda [157] present a survey of linguistic structures for application-level fault tolerance (ALFT). The authors emphasize the importance of employing appropriate structuring techniques to support an adequate separation between the functional and fault tolerance concerns. They claim the design choice of which fault tolerance provisions to support can be conditioned by the adequacy of the syntactical structure at 'hosting' the various provisions, called *syntactical adequacy.* Moreover, offline and online (dynamic) management of fault tolerance provisions and their parameters may be an important requirement for managing the fault-tolerant code in an ALFT, called *adaptability.* These three properties, separation of concerns, adaptability and syntactical adequacy are referred as the *structural attributes* of ALFT. The structural attributes are adopted to classify and analyse a number of ALFTs, including, recovery blocks and n-version programming. This classification is also orthogonal to the classification we have provided.

4.8 Summary

Due to the low cost of reusing existing functionally equivalent services, called alternate services, several solutions based on design diversity exist to support fault-tolerant Service-Oriented Architecture (SOA). Regarding fault tolerance based on software diversity, three major design issues need to be considered, namely, selection of alternate services; alternate execution schemes; and judgement on result acceptability. These design issues may be realized by different design solutions. Different design decisions imply in different measures of quality requirements (e.g. memory utilization, execution time, reliability, financial costs and availability). In this chapter, we define a general taxonomy for these common design issues and their different design solutions. We also provide an initial comparison of the various design solutions in terms of their quality requirements. Based on this information and by means of systematic literature review method, we present a comprehensive survey of existing solutions for fault-tolerant SOAs and discuss their main contributions and limitations, thus, suggesting directions for future work.

Because a SOA differs from conventional architectures, it poses new challenges for diversity-based solutions. Applications based on SOA rely in a singular scenario, where the environment is highly dynamic, the uncertainty is high and several decisions cannot be taken at design time but must be postponed until runtime, where the control is highly distributed, and we have different stakeholders with possibly conflicting requirements [17, 16, 60, 158]. Because of that, solutions should provide greater flexibility in terms of alternate selection, execution of alternate services and the decision mechanisms. For examplo, decision mechanisms need to be flexible in order to take into account the wide range of types of adjudicators so that a specific adjudicator could be adopted accordingly to high-level policies specifying what is desired.

In Chapter 5, we design a family of software fault tolerance techniques based on design diversity to support FT-compositions. The proposed solution provides an infrastructure that leverages software product line engineering to facilitate the development of fault tolerance strategies tailored to individual clients' needs. The proposed family was specified and designed by means of a model-driven infrastructure, presented in Chapter 5, for developing product line architectures. We adopted COSMOS* (*Section 2.1.2*) to implement the resulting product line architecture, thus creating a 'skeletal' system in which the communication paths are exercised but which at first has a minimal functionality. This 'skeletal' system can then be used to implement the final products, *i.e.* the software fault tolerance techniques, incrementally, easing the integration and testing efforts [12]. Implementation of the design solutions supported by the analysed primary studies (*Table 4.3*) could also be reused to implement the family of fault tolerance techniques for FT-compositions (*i.e.* by wrapping such implementations into COSMOS* implementation packages).

Capítulo 5

A Model-Driven Infrastructure for Developing Product Line Architectures

Neste capítulo, apresentamos uma infraestrutura dirigida por modelos para implementar de forma consistente e coordenada arquiteturas de linhas de produtos de software em geral. Utilizando esta infraestrutura, criamos uma família de técnicas de tolerância a falhas baseadas em diversidade de projetos para apoiar composições de serviços tolerantes a falhas. O conteúdo desse capítulo foi retirado do artigo publicado no *7th Brazilian Symposium on Software Components, Architecture, and Reuse - SBCARS'13*, que apresenta a infraestrutra dirigida por modelos para apoiar o desenvolvimento de arquiteturas de linhas de produtos de software e sua avaliação. Como o conteúdo do capítulo foi extraído na íntegra desse artigo, foi preservado o idiomal original.

5.1 Overview

Currently, there is an increasing need to address software architecture evolvability in order to gradually cope with new stakeholders' needs [27]. At the same time, the software system's desired time-to-market is ever decreasing [27, 79, 159]. This reality demands on software architectures' capability of rapid modification and enhancement to achieve cost-effective software evolution. To face these needs, advanced software paradigms have emerged. For instance, Model Driven Engineering (MDE) and Software Product Line Engineering (SPLE) are two promising cases in particular as they increase software reuse at the architectural level of design [79, 159, 45].

MDE conceives software development as transformation chains where higher level models are transformed into lower level models [160]. In MDE, reuse is mainly achieved by means of model transformations, which are built once but can be enacted for a variety of input models that yield different results [160]. Models are considered as first-class entities, are used to capture every important aspect of a software system for a given purpose, and are not just auxiliary documentation artefacts; rather, they are source artefacts and can be used for automated analysis and/or code generation [159, 81].

SPLE systematises the reuse of software artefacts through the exploration of commonalities and variabilities among similar products [27, 161]. Feature modelling is a technique for representing the commonalities and the variabilities in concise, taxonomic form [2, 159] (*Section 2.1.3*). A key to the success of SPL implementation is structuring its commonalities and variabilities in Product Line Architectures (PLAs) in terms of variable architectural elements, and their respective interfaces, which are associated with variants [27] (*Section 2.1.3*). Furthermore, SPLE encompasses two sub-processes [27]: *(i)* domain engineering aims to define and realize the commonality and the variability of the SPL; and *(ii)* application engineering aims to derive specific applications by exploiting the variability of the SPL.

The combination of MDE and SPLE, referred to as Model-Driven Product Line Engineering (MD-PLE), integrates the advantages of both [79]. SPLE provides a well-defined application scope, which enables the development and selection of appropriate modelling languages [159]. On the other hand, MDE supports the different facets of a product line more abstractly. For instance, architectural variability can be specified systematically by appropriate models, thus enabling *(i)* an automated product derivation process in accordance with feature configurations, also called resolution models [3]; and *(ii)* an automated generation of source code to implement PLA models [78, 159]. Model-driven engineering method to develop SPLs should describe systematically how to refine models to decrease the level of abstraction to reach a code implementation. Nevertheless, existing solutions for MD-PLE either do not define clearly the sequence of models to be developed at each SPLE sub-process and how models transformations are performed between the different process phases [78, 79], or, if they do, they lack of automation support [80, 81].

In this sense, we propose a model-driven infrastructure to implement PLAs through a step-by-step refinement from high-level models into lower level abstractions. The infrastructure employs a set of existing process, tools and models, such as model transformations are semi-automated. More specifically, to specify common and variable requirements of the product line we adopt use case models, as suggested by Gomaa [88] (*Section 2.1.3*), and feature models [2] (*Section 2.1.3*). Second, the FArM (Feature-Architecture Mapping [2]) method is adopted to perform a sequence of transformations on the initial feature model to map features to a PLA (*Section 2.1.3*). Third, we explicitly and systematically specify architectural variability such that executing the variability specification model with a resolution model will yield a product model. In particular, our solution is based on the Common Variability Language (CVL), a generic variability modelling language being considered for standardization at the Object Management Group (OMG) [3] (*Section 2.1.3*). Finally, in accordance with the PLA model, the source code generation is guided by component-based development process and uses COSMOS*, a component implementation model that materializes the elements of a software architecture using the concepts available in object-oriented programming languages [77] (*Section 2.1.2*).

We have identified a set of supporting tools to specify and validate the target models and automate model-to-model transformations. We show the feasibility and effectiveness of the proposed solution by employing it to implement a PLA to support a family of fault tolerance techniques applied to service-oriented architectures (SOAs). The resulting models are described in detail and lessons learned from executing this case study are also presented. Contributions of this work are threfold *(i)* a model-driven, systematic and semi-automated engineering method to develop PLAs; *(ii)* to promote the incorporation of CVL and its supporting tools in a comprehensive engineering method for PLA implementation, ensuring that CVL models are effectively coordinated; and *(iii)* a family of software fault tolerance techniques to support fault-tolerant service-orietend systems.

5.2 A Model-Driven Infrastructure for Product Line Architecture Development

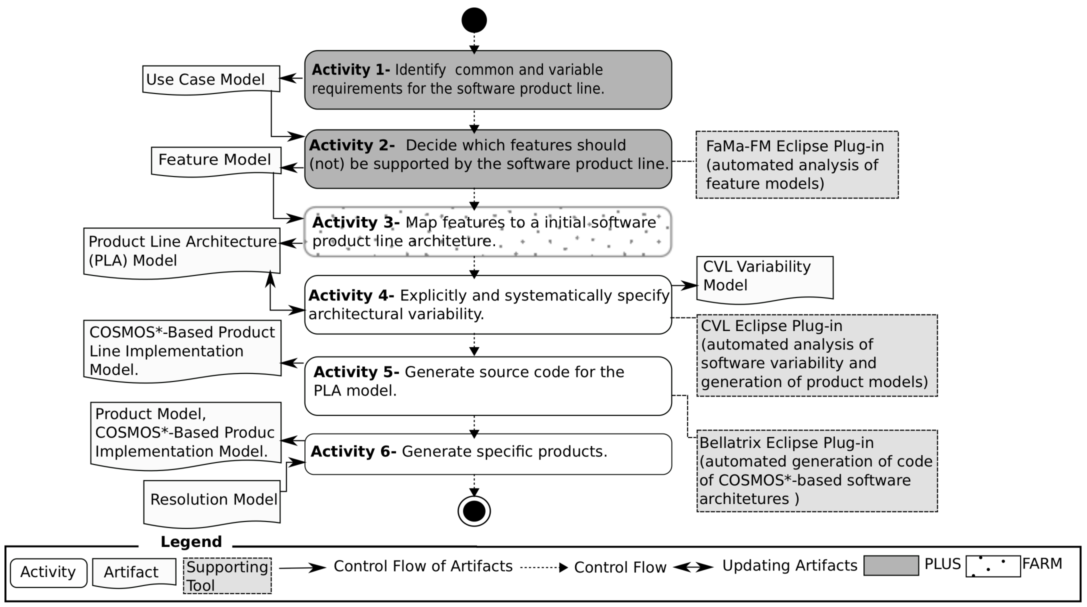
Use Case Model
Activity 1- Identify common and variable requirements for the software product line.
Activity 2- Decide which features should (not) be supported by the software product line.
Feature Model
FaMa-FM Eclipse Plug-in (automated analysis of feature models)
Activity 3- Map features to a initial software product line architeture.
Product Line Architecture (PLA) Model
Activity 4- Explicitly and systematically specify architectural variability.
CVL Variability Model
CVL Eclipse Plug-in (automated analysis of software variability and generation of product models)
COSMOS*-Based Product Line Implementation Model.
Activity 5- Generate source code for the PLA model.
Product Model, COSMOS*-Based Produc Implementation Model.
Activity 6- Generate specific products.
Bellatrix Eclipse Plug-in (automated generation of code of COSMOS*-based software architetures)
Resolution Model
Legend
Activity
Artifact
Supporting Tool
Control Flow of Artifacts
Control Flow
Updating Artifacts
PLUS
FARM

Figure 5.1: A semi-automated model-driven method for developing product line architectures

We present a model-driven infrastructure that combines existing methods, tools, models and languages to implement PLAs, as illustrated in Figure 5.1. The infrastructure encompasses a semi-automated engineering method that supports a step-by-step high-level refinement of models into lower level abstractions. In summary, the infrastructure firstly enables the identification and specification of common and variable requirements and features of a SPL (Activities 1 and 2). Secondly, features are realized by a component-based PLA (Activities 3-5). Finally, features and their respective components are chosen which allowing for instantiation of a final product (Activity 6). The resulting feature model, PLA model and variability model are persisted in XMI (XML Metadata Interchange) format, the standard proposed by the OMG for MOF metadata exchange. The main activities of the proposed method are described in detail in the following.

5.2.1 Activities 1-2: To Specify Use Case and Feature Models

These activities are concerned with the domain requirements engineering which encompasses all activities for eliciting and documenting the common and variable requirements of the SPL [27].

Specification of Use Case Models

The input for this activity consists of the product roadmap and domain knowledge. This may include, for example, specification, (pseudo) codes and data flow diagrams related to the products needed. The output comprises of reusable, textual and model-based requirements. In particular, for specification of use cases, we suggest the template proposed by Gomaa [88] (*Section 2.1.3*).

Specification and Validation of Feature Models

The input for this activity is mainly composed by the use case descriptions. The output comprises of a feature model and feature specifications. At this stage, a feature model is specified in order to differentiate among members of the SPL [5, 88] (*Section 2.1.3*). Use cases and features can be used to complement each other, thus enhancing variability descriptions [88]. A software engineer is responsible for specifying a feature model that is consistent with the use cases [88]. In particular, use cases should be mapped to the features on the basis of their reuse properties [88]. To specify and validate features models, we adopted *FaMa-FM*, a largely accepted test suite to support automated analysis of feature models [162]. For instance, we test whether the feature model is valid and we verify the number of products it encompasses.

5.2.2 Activity 3: To Map from Features to a PLA Model

The input of this activity consists of the feature model and feature specifications. The output comprises of an initial PLA. To specify the PLA, we adopt the FArM method (*Section 2.1.3*). We emphasize that a correct specification of feature realizations is essential for the derivation of correct and intended products [2]. This specification is frequently carried out by a software engineer who has a great understanding of the product line domain, making automation extremely difficult. Although the transformation from features to a PLA model is not automated, FArM encompasses a series of well-defined transformations on the initial feature model in order to achieve a strong mapping between features and the architecture [2]. Consequently, the FArM method ensures that the feature model is consistent with the PLA. Variability at the PLA implies using variable configurations of components and interfaces (*e.g.* common components can use required interfaces for accessing functionality provided by variant components [27]).

5.2.3 Activity 4: To Specify Architectural Variability

The input of this activity comprises of the transformed feature model and the PLA model (*i.e.* the models resulting from FArM transformations - Activity 3). The output consists of a CVL variability model specifying architectural variability as first-class entities. Although FArM is essential to drive the specification of the PLA, it does not systematically define the details of the transformations required to get a specific product model according to a resolution model. To overcome this gap, our solution employs CVL to specify software variability explicitly and systematically at the PLA model *(Section 2.1.3*). CVL ensures the consistency among the PLA model and its products [3, 73, 163]. By means of the CVL-Enable Papyrus Eclipse Plug-in, the initial PLA is firstly specified in UML, a MOF-based language. The resulting PLA model encompasses the base and variant models, according to the CVL nomenclature (*Figure 2.3*). Second, the transformed feature model is used to specify the CVL variability model by using the CVL diagram, which is encompassed by the CVL Eclipse Plug-in. This plug-in also enables the definition of how model elements of the base model shall be manipulated to yield a new product model [3], as exemplified in *Section 3.3*.

5.2.4 Activity 5: To Translate PLAs to Implementation Models

The input of this activity comprises of the UML-based PLA model. The output consists of a Java source model encompassing the COSMOS* sub-models (*Section 2.1.2*) for each architectural component and connector of the PLA model. The PLA model (from Activity 3) is used as the basis for model transformation from component-based software architectures to detailed design of software components. In particular, we adopt the Bellatrix case tool [86] to automate the generation of Java source code in COSMOS* from UML component based-models specified in XMI. By automatically generating Java source code in COSMOS*, we ensure that the PLA implementation is consistent with the PLA model.

5.2.5 Activity 6: To Generate Product Models

The input of this activity comprises of the UML-based PLA, the CVL variability model and the COSMOS*-based source code. The output consists of a product model architecture and its respective COSMOS*-based implementation. Software components, interfaces, and other software assets are not created anew. Instead, they are derived from the platform by binding variability [27, 3]. At this stage, a software engineer is able to configure a specific product by choosing the set of features required. For instance, it is necessary to create a CVL resolution model (*Section 2.1.3*). The realizations of the chosen features are then applied to the base model to derive the product model by using the CVL Eclipse Plug-in [3]. That is, by using the product line and resolution models, specifics products are obtained by CVL model-to-model transformation, which is in charge of exercising the built-in variability mechanisms of the PLA model [3, 73, 163]. As already mentioned, Bellatrix creates a 'skeletal' PLA implementation in which the communication paths are exercised but which at first has a minimal functionality. In this way, the product architecture implementation can be derived by specific choices of the COSMOS*-based components and connectors realising the product model (*e.g.* by connecting the required interface of each common component to exactly one, matching provided interface of a variant component [27]).

5.3 Evaluation

We present an illustrative application of the proposed solution to demonstrate its feasibility, effectiveness and present a concrete example of our activities in practice.

5.3.1 A Family of Software Fault Tolerance Techniques for SOAs

We show the development of a PLA to support a family of software fault tolerance techniques based on design diversity applied to SOAs [57]. In summary, these techniques are used to create fault-tolerant composite services, called *FT-compositions*, that leverage functionally equivalent services, or *alternate services*, to tolerate software faults [40]. We felt this example was well suited to demonstrate our solution as existing software fault tolerance techniques differ in terms of *(i)* quality requirements (*e.g.* execution time and reliability), and *(ii)* types of alternates' results that they are able to adjudicate (*e.g.* simple or complex data types) - that makes different techniques more suitable in different contexts [57, 1]. Mass customization using a PLA to support multiple fault tolerance techniques can be employed to address different demands on solutions for FT-compositions.

We consider the following fault tolerance techniques: Recovery Block (RcB), N-Version Programming (NVP) and N-Self-Checking Programming (NSCP) [1]. In addition, we also take into account the RcB' and NVP' techniques, indicating, respectively, the adoption of parallel execution of the multiple alternate services combined with Acceptance Tests, and, the fully sequential execution of the multiple alternate services combined with voters [1, 57]. In the following, we describe the activities and models presented in Figure 5.1, but individual use case descriptions are omitted due to space constraints. For the specification of use cases, we took into account a set of roles and functionalities that composite services, in general, should encompass for the aggregation of multiple services into a single composite service (*e.g.* interoperability capabilities and autonomic composition of services) [16]. Additional information for use case specification was taken from domain knowledge of software fault tolerance techniques based on design diversity, such as the various types of adjudicators and their operations [45, 46, 1, 57].

A Feature Model (FT-FM) for Fault-Tolerant Composite Services

Features were derived from use cases descriptions (*Figure 5.1: Activity 1*). In Figure 5.2(a), we show an excerpt of the resulting feature model (*Figure 5.1: Activity 2*). The feature model is an extension of the model proposed by Brito et al. [45] and the one of our previous work [40]. In comparison with these previous models, our revised feature model *(i)* identifies different types of adjudicators [57] (these sub-features were omitted in the figure to avoid an unreadable feature model); *(ii)* explicitly distinguishes between different sequential execution schemes; *(iii)* allows the combination of a parallel execution scheme with acceptance tests (for instance, the RcB' technique) and the combination of a fully sequential scheme with voters (for instance, the NVP' technique); and *(iv)* explicitly states some aspects that are specifics to FT- compositions.

The *Executive* feature aims to orchestrate a software fault tolerance technique operation. It is composed of seven mandatory features. The *Consistency of input data* feature presents how the consistency of data is achieved, either implicitly through backward error recovery, or explicitly through a synchronization regime [1, 45]. The *Execution scheme* feature represents the three possible ways for executing the alternate services: conditionally or fully sequential execution and parallel execution [1, 57]. The *Judgement on result* feature presents how the judgement on result should be performed, either with an *absolute criteria* (involving the result of only one alternate service), or a *relative criteria* (involving the results of more than one alternate service) [45, 46]. The *Adjudicator* feature captures a set of alternative features related to the different ways that can be employed for detecting errors: by emphacceptance testing (ATs), *voting* or *comparison* [45, 1, 57].

The *Number of Alternate Services for tolerating f sequential faults* represents the number of alternate services to tolerate *f* solid faults [1, 45, 46]. A solid software fault is recurrent under normal operation or cannot be recovered [46]. In contrast, a soft software fault is recoverable and has a negligible likelihood of recurrence. Provided there are no fault coincidences, an architecture tolerating a solid fault can also tolerate a (theoretically) infinite sequence of soft faults [46]. The *Dynamic Selection* represents which alternate services will be dynamically employed as part of the FT-composition. Alternate services are defined by a common shared terminology so that service selection can be achieved automatically [16]. The *Suspension of operation delivery during error processing* feature indicates whether the error recovery technique suspends the execution when an error is detected [1, 45, 46]. In case the execution is suspended, it is necessary to define what the purpose of the suspension is: either for re-executing the target operation or only for switching to another result [45, 46].

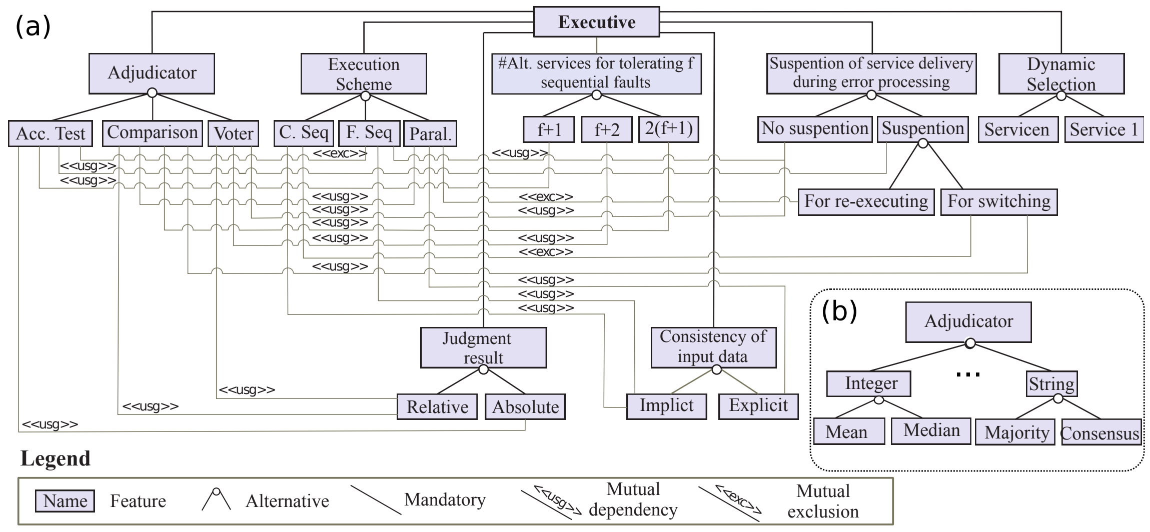

Figure 5.2: A Partial Feature Model for Software Fault Tolerance Techniques Applied to SOA (notation proposed by Ferber et al. [5])

According to the FaMa-FM [162] tool, the proposed feature model (*Figure 5.2*(a)) is a valid one and it encompasses 36 products (taking into account the different subtypes of adjudicators).

A Product Line Architecture for Fault-Tolerant Composite Services

We adopt the FArM (*Section 2.1.3*) to map from features to a PLA (*Figure 5.1: Activity 3*). Regarding the first transformation, our initial feature model contains neither architecture-related nor quality features; consequently, no transformations were performed at this stage. With respect to the second transformation, we identified one architectural requirement: interoperability. Products should provide its main functionalities via services that are defined by means of well-defined interfaces that should be neutral, platform- and language-independent. This requirement was used to enhance the description of the *Executive* feature that should provide interoperable services. Related to the third transformation, we analysed interacts relations, as follows.

The *Judgement on Result* and *Suspension of Service Delivery* are used, in fact encompassed, by *Adjudicator* and *Execution Scheme*, respectively. As a consequence, we integrated *Judgement on Result* and *Suspension of Service Delivery*, into, respectively, *Adjudicator* and *Execution Scheme*. We also integrated the features related to the consistency of input data into features related to the execution of the alternate services, due to their usage interacts relations, represented in Figure 5.2(a). Therefore, execution schemes must provide all alternate services with exactly the same experience the system state when their respective executions start [1]. Regarding the fourth FArM transformation, the *# of alternative services for tolerating f sequential faults* feature and its sub-features do not present a valid hierarchy relation - they simply represent a value assignment. We decided to remove theses features from the transformed model. Due of this removal, we enhance the specification of the adjudicators by explicitly specifying the number of alternate services each type of adjudicator should ideally employ to tolerate f sequential faults.

CVL Models related to the Feature and Product Line Architecture Models

At this stage, CVL was adopt to explicitly specify architectural variability (*Figure 5.1: Activity 4*). Figure 5.3 shows an excerpt of the transformed feature model and PLA (from FArM transformations). We have omitted architectural connectors and interface names for the sake of clarity. More specifically, the transformed feature model is represented by means of a CVL variability model, which is also composed by the variability specification tree (VSpec). In the VTSpec, in Figure 5.3(a), choices are represented by rounded rectangle with their names inside. A non-dashed link and dashed link between choices indicates that the child choice is, respectively, resolved according to its parent [3] and is not implied by its parent. For example, if the *Executive* is resolved to true, its sub-choices are also resolved to true, whereas when *Execution Scheme* is true, its sub-choices can be resolved either positively or negatively. Choices have a group multiplicity, indicated by a triangle, with a lower and an upper bound (*e.g.* we can select up to n alternate services and exactly one adjudicator). VSpec trees are similar to feature models and deciding choices is similar to selecting features [3].

The resulting FT-PLA model is represented by means of the base model and variant models using UML, as illustrated in Figure 5.3(d) and Figure 5.3(c), respectively. In Figure 5.3(b), a *variation point* is a specification of concrete variability in the base model and it defines specific modifications to be applied to the base model during materialisation [3, 73]. Variation points, defined at the product realization layer, refer to base model elements via base model handles and are bound to elements of the VSpec [3]. For instance, we represent variability on the base model by means of *FragmentSubstituition*. A *FragmentSubstitution* is a choice variation point which specifies that a fragment of the base model, called the placement, may be substituted for another fragment, called the replacement. As illustrated in Figure 5.3(d), we created auxiliary object models, which are not transformed to source code, to define placements. The replacement objects, or variant models, can be found into the CVL Library (*Figure 5.3(c)*).

The *resolution model* is a structural replica of the corresponding VSpec model such that choices are positively or negatively resolved [3], as exemplified in Figure 5.3(e). In particular, we always assume that a resolution model starts with a set of resolutions - choices related to the mandatory features (*i.e.* Executive, Adjudicator, Selector and Execution Scheme) are always set to true, whereas alternative and multiple choices need to be resolved to true or false. To generate product models, these choices are resolved by a resolution model and propagated to variation points and the base model [3]. In Figure 5.3(e), for example, we select the *Parallel* execution scheme, the *Service 1*, *Service 2* and *Service 3* as alternate services and we choose to judge the results from the alternate services by using a median voter (*Median Vt*). Therefore, after executing the CVL transformation, as a resolved model, we have an architectural specification in UML of the N-Version Programming (NVP) with a median voter, as illustrated in Figure 5.3(f) (*Figure 5.1: Activity 6*).

By means of the CVL variability model, CVL also supports the definition of OCL constraints among elements of a VSpec to define choices constraints [3]. For example, we should specify that to choose the conditionally sequential execution scheme implies choosing an acceptance test as an adjudicator, thus, being able to discard invalid configurations - these features are mutually inclusive as represented in Figure 5.2(a).

In *Figure 5.4*, which is semantically equivalent to Figure 5.3, there is an excerpt of the models we generated by means of CVL Eclipse Plug-ins. From left to right, we have the following models. First, we defined elements of the CVL library, for instance, variant models related to the different types of adjudicators, execution schemes and alternate services. Secondly, we have the base model, which mainly encompasses the mandatory features (or choices in CVL notation), with the exception of auxiliary elements defined

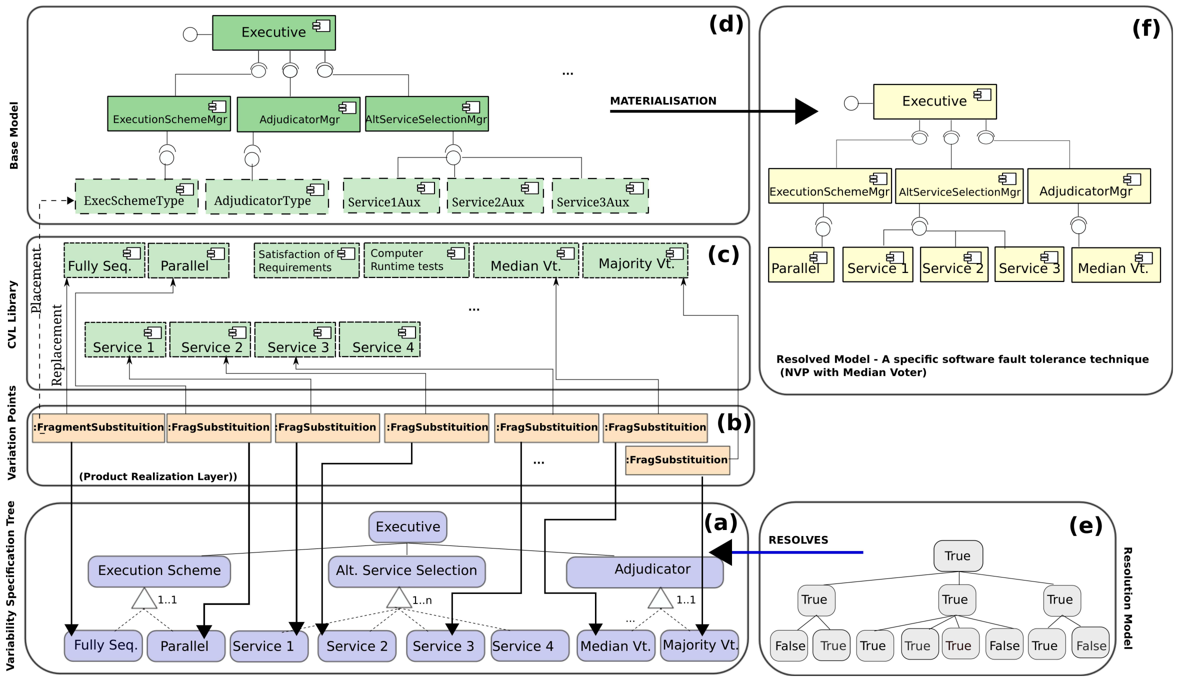

Figure 5.3: Using CVL to explicitly and systematically specify and resolve variability at the PLA model

solely to support the specification of placement fragments. Thirdly, we have the model representing the software architecture of the NVP software fault tolerance technique employing a median voter as an adjudicator. Finally, we have the variability specification tree in which variation points are defined and linked to the base model and the elements from the CVL library. In the variability specification tree, the highlighted elements are part of the resolution model in which is specified the choices related to the instantiation of the NVP with median voter (*Figure 5.1: Activity 6*). The CVL tool also supports validation operations and the defined variability model was validated successfully.

The PLA Implementation

Bellatrix was used to materialise through model-to-model transformation the elements of the UML-based PLA to JAVA source code in COSMOS* (*Figure 5.1: Activity 5*).

5.4 Lessons Learned

We proposed a model-driven and semi-automated infrastructure to define and manage PLAs. Our solution encompasses a model-based method *(i)* to specify common and variable use cases as proposed by Gomaa [88]; *(ii)* to specify a feature model [5]; *(iii)* to

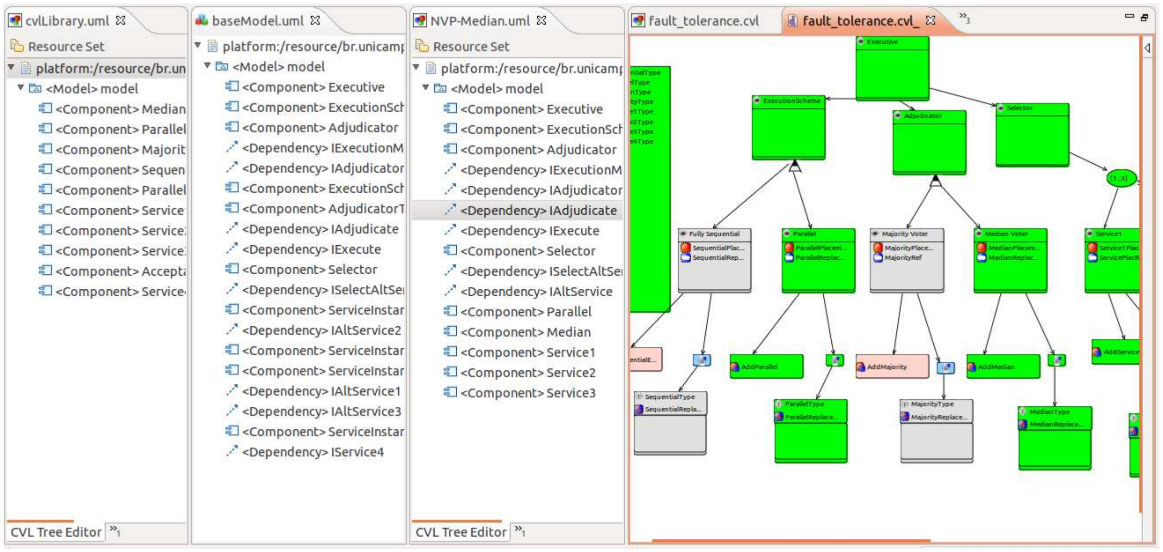

Figure 5.4: Using CVL Eclipse Plug-in to specify architectural variability

map features to the PLA using FArM [2]; *(iv)* to specify variability at the architectural level using CVL [3]; *(v)* and to automate the generation of code related to the software architecture using COMOS* [77]. These reused approaches have been validated, are well-documented and explored in other domains [2, 3, 73, 163, 88]. Differently, in this work we document and explore how these approaches and related tools can be used together to develop PLAs. In particular, we employ this method to define a family of software fault tolerance techniques for fault-tolerant service compositions. We discuss lessons learned from executing this case study.

Activities 1 and 2 (*Figure 5.1*) were adopted to specify common and variable requirements, which was essential to achieve a better understanding of the commonality and the variability of the intended SPL and to define the set of applications the SPL is planned for. To map from use cases to features is an error-prone and time-consuming task. Although it is a research challenge to fully automate this task, it could be greatly benefitted if an initial feature model could be automatically generated by analysing reuse properties of the use cases. Under this circumstance, a software engineer could incrementally refine the initial feature model by specifying appropriate many-to-many associations [88].

Activity 3 specifies a feature-oriented development method to map features to architectural components using FArM. The transformation from the feature model to the PLA was smoothly, although it is not automated. We could observe that to map a feature to the PLA can be systematically performed by carrying out the series of FArM transformations for the feature at hand [2]. Furthermore, due to the encapsulation of the feature business logic in one architectural component, the effects of removing and updating features were localized. As claimed by Sochos et al. [2], we noticed that FArM improves maintainability and flexibility of PLAs. Nevertheless, FArM does not *(i)* define the details of the transformations required in order to generate product models in accordance with feature configurations; and *(ii)* provide clear guidelines on how to describe all the traceability links among the initial and transformed feature models and the resulting PLA.

Activity 4 uses CVL to specify software variability as a first-class concern [73]. The CVL Eclipse Plug-in greatly facilitates the specification and resolution of software variability. By specifying variability systematically, we are able to produce the right product models automatically given a resolution model encompassing a set of preselected features (**Activity 6**). This formalization of product creation process improves analyzability and maintainability [3, 73, 163]. In addition, by using CVL, the specification of the PLA using UML does not have to be extended or overloaded with variability. The base and variant models are oblivious to the CVL model while the CVL model refers to objects of those models [3]. Because of this separation of concerns, there may be several variability models applying to the same architectural model which also facilitates the evolution of these product line assets [3].

We noticed that there are several strategies for defining a base model (*Figure 2.3*). One obvious choice is to define a complete model where CVL can remove features to get a specific product model [163]. Another choice is to adopt a minimal CVL model such that a product model will be generated by adding features to the base model [163]. We preferred to use neither maximum nor minimum, but somewhere in between. We created auxiliary elements in the base model to represent placement fragments. The variant models (found in the CVL Library) were then used as replacements for these placement fragments. This seemed to be the most practical way to specify fragment substitutions in our software solution.

Using the transformed feature model and its related PLA to define, respectively, the variability specification tree and the base and variant models, was essential to ease the selection of products and maintenance of the product line. By using models from FArM, it is possible to define a compact product realization layer regarding the number of substitutions and the complexity of the substitutions. This is due to the strong mapping from features to software components [2]. Based on the choice specification layer, the product realization layer further defines low level, fairly detailed, operations required to transform the base model to a resolved product model [73].

The learning curve of using the proposed method is an important consideration due to the different types of tools and documentation that is available for the activities. The specification of use cases was easily performed as Gomaa [88] presents a template for use case specifications. With respect to the FaMa-FM, XML (eXtensible Markup Language), Lisp or plain text can be used to load and save feature models [162]. Although there are many examples available illustrating the feature model notations employed by FaMa-FM, in fact, a GUI editor tool support is missing. With respect to FArM, the transition from a feature model to a PLA is explained by the authors in a short but comprehensive way as they provide a running example illustrating each FArM transformation [2].

CVL is a relatively small language with clearly defined semantics making it less time-consuming to learn it [73]. By being able to use the base MOF-compliant language editor to select and highlight fragments, it is not required to know the lower level implementation details of CVL [73]. In our case, the definition of fragments and substitutions on the target base model was intuitive as we were familiar with our architectural model and the Papyrus Eclipse Plug-in. CVL Plug-in requires '*an old*' Eclipse version and some compatible libraries [3]. To find some of the required libraries were a time-consuming task. The COSMOS* implementation model also defines clear semantics which are in fact representative of software component models. It was also straightforward to implement component-based software architectures specifically due to the support of Bellatrix (**Activities 5 and 6**).

The tasks to ensure consistency among models are not fully automated. Under some refinements from high-level models into lower level abstractions, a software engineer is responsible for ensuring that models are consistent. Because it is an error-prone activity, we identified useful guidelines to facilitate the accomplishment of this task (*e.g.* the method by Gomaa [88] to map from use cases to features and FArM [2] to map from features to PLAs). Nevertheless, despite the smooth transition from the feature model to the architecture, the process could be greatly facilitated if there was one tool that provided support for all the activities. This would achieve greater consistency, traceability among models, efficiency and familiarity within the development environment which would lessen the learning curve. Unfortunately, it is difficult to find the support for specific modelling needs (*e.g.* to model optional/ mandatory features and architectural variability etc.) at the same time as finding compatible tools that support a large range of the specified activities.

5.5 Related Work

Zhang et al. [78] propose a tool to compare the existing potential product models to identify reusable assets such as model fragments. A preliminary PLA model and a CVL variability model can be automatically specified based on the comparison results. Our solutions complement each other as we can employ the *CVL Compare* tool to generate the initial PLA model and they can use Bellatrix to generate code automatically. The solution by Chastek el al. [80], encompasses a set of process, and tools to ensure that models are effectively coordinated. Our solution can be regarded as an instantiation of their method engineering, which is described in a higher level of abstraction. Their solution does not allow the automatic generation of product models given a resolution model, neither the automatic translation of the PLA model to an implementation model. Unlike our solution, Chastek el al. [80] present general guidelines to create test assets by using the JUnit test framework.

Azanza et al. [79] present a MD-PLE, but they focus on extracting product line artefacts from existing software applications. Our solution, in fact, can be regarded as a proactive approach for mass-customization [161], which is appropriate when the requirements for the set of products needed are well defined and stable [161]. Buchmann et al. [164] propose a MD-PLE for software configuration management systems. They are able to instantiate an application-specific system, which is executable, according to the selection of features. We intend to reuse their solution to refine the COSMOS*-based implementation of PLAs. To specify architectural variability they develop a tool that maps features to corresponding domain model elements by manually annotating elements of the domain model with feature expressions. To specify and manage these annotations is an error-prone activity. At this point, we adopt CVL, a well-defined language for specifying and resolving software variability [3, 73, 163].

A number of works have been using CVL to specify and resolve software variability [3, 73, 163]. In particular, we refer to the work by Svendsen et al. [73] for a detailed description of how to use CVL, its main concepts, the CVL tool support and a discussion on different strategies to choose a base model and to create the CVL Library. One of the main purposes of that work [73] is to show the applicability of the CVL to support the transformation of an original (*e.g.* a product line model) into a configured, new product model.

5.6 Summary

In this chapter, we present a comprehensive, semi-automated, systematic and model-driven engineering method for Product Line Architecture (PLA) development ensuring that models are effectively coordinated. The method is supported by an infrastructure that encompasses a set of existing processes, languages and tools, which are consistent and interoperable. The process involves extracting common and variable functional requirements of the product line using use case models. To differentiate among members of the product line, features are extracted to form a feature model and subsequently mapped to a component-based PLA model. Finally, using preselected features, individual component-based product models are generated through model-to-model transformations. Our solution employs Common Variability Language to specify and resolve architectural variability and COSMOS*, a component implementation model, to realize the PLA. We presented well known Eclipse Plug-ins that can be used to manage and validate the models and to support model-to-model transformations.

To exemplify and evaluate the proposed solution, we employ it to develop a PLA to support family of software fault tolerance techniques for fault-tolerant composite services. The proposed family can be extended by either adding or removing features in order to cope with different clients' requirements. As discussed in this chapter, the proposed model-driven infrastructure supports the generation of 'skeletal' systems, consequently, to have executable fault tolerance strategies, it is necessary to incrementally implement the behaviour of methods defined within COSMOS*-based implementation models. The provisioning of components to meet this specification can be achieved either by directly implementing the specification or by finding an existing component that fits the specification [165] (*i.e.* to reuse the implementation provided by existing solutions for FT-compositions - Table 4.3).

In Chapter 6, we present ArCMAPE, an infraestrutura based on dynamic software product lines to support self-adaptation of fault-tolerant service compositions. ArCMAPE relies on a reflective architecture to support separation of concerns. The base-level is composed by a framework implementing the FT-PLA. Software components implementing the mandatory features are already provided, whereas variant software components can be added at predefined places. The meta-level leverages the FT-FM and FT-PLA models at runtime and resolves variants dynamically in order to support the dynamic instantiation of fault tolerance techniques more adapted to the current context.

Capítulo 6

ArCMAPE: A Software Product Line Infrastructure to Support Self-Adaptation of Fault-Tolerant Composite Services

Neste capítulo propomos ArCMAPE, uma solução baseada em linhas de produtos de software (dinâmicas) para apoiar uma família de técnicas de tolerância a falhas baseadas em diversidade de projetos para composições de serviços confiáveis, tal que a técnica mais adaptada ao contexto seja instanciada em tempo de execução. ArCMAPE é uma extensão da nossa implementação preliminar para composições de serviços autoadaptativas e confiáveis, publicada no 1st Workshop on Services, Clouds, and Alternative Design Strategies for Variant-Rich Software Systems em conjunto com SPLC'11. A solução inicial foi estendida para apoiar uma separação clara de interesses entre a lógica de tolerância a falhas e lógica de adaptação, especificar e resolver variabilidades arquiteturais de forma sistemática, e apoiar a geração dinâmica de planos de adaptação. Estas necessidades de melhorias foram identificadas a partir de um estudo do comportamento dinâmico da solução preliminar, que foi apresentado num artigo publicado no *1st Workshop on Dependability in Adaptive and Self-Managing Systems em conjunto com LADC'13.* Parte do conteúdo deste capítulo foi retirado de um artigo a ser publicado no *15th IEEE International Symposium on High Assurance Systems Engineering - HASE'14*, que descreve detalhes acerca da especificação, projeto e implementação do ArCMAPE. Como a maior parte conteúdo deste capítulo foi extraído na íntegra de tal artigo, foi preservado o idioma original. Enfatizamos que um resumo expandido sobre ArCMAPE foi publicado no *7th IEEE International Conference on Self-Adaptive and Self-Organizing Systems - SASO' 13*

6.1 Overview

A systematic review of solutions for fault-tolerant composite web services [40, 56], called *FT-compositions* for simplicity, has shown us that *(i)* few solutions support various software fault tolerance techniques at the same time [34, 41]; and *(ii)* solutions do not support an adaptable architecture at which changes can be made easily, specially, at runtime [33, 126] (*Section 4.5*). Systems based on Service-Oriented Architectures (SOA) often rely in an environment that is highly dynamic and several decisions should be postponed until runtime, where we have different stakeholders with conflicting requirements, and fluctuations in the quality of services (QoS) are recurrent [17, 16]. Consequently, solutions for FT-compositions ideally should support an adaptive fault tolerance mechanism to dynamically bring out fault tolerance strategies in close accordance with clients' requirements and the environment.

Adaptive Fault Tolerance Mechanisms (AFTMs) are able to meet the dynamically and widely changing fault tolerance requirements [63]. Nevertheless, AFTMs raise a number of research and engineering challenges. In AFTMs, the adaptation logic should be separated from the fault tolerance logic to improve the comprehension, maintainability, testability and modularity of such mechanisms [63, 67]. Furthermore, it is important to clearly separate the components realizing the fault tolerance logic from the components realizing the dynamic adaptation [67]. This separation requires using a set of well-structured and clearly organized high-level abstractions to reason about the fault tolerance mechanism's dynamic behaviour and its environment [67, 66]. Finally, it is necessary to link the high-level abstractions to the running fault tolerance strategy (to actually adapt it) [68, 67].

Dynamic Software Product Lines (DSPLs) extend Software Product Lines (SPLs) to support late variability [18]. DSPLs provide the modelling framework to understand a self-adaptive system by highlighting the relationships among its parts [18] (*Section 2.1.3*). Furthermore, by leveraging the product line assets to runtime, DSPLs ensure that a self-adaptive system moves from a consistent configuration to another in accordance with feature constraints [18] (*Section 2.4*). We propose an DSPL infrastructure, called ArCMAPE, to support self-adaptation of FT-compositions, by means of feature-based runtime adaptations. In particular, ArCMAPE *(i)* supports a family of software fault tolerance techniques based on design diversity; and *(ii)* is able to dynamically instantiate fault tolerance techniques suitable to the context in accordance with high-level policies.

The target family of fault tolerance techniques can be developed on the basis of the FT-FM and FT-PLA models described in Section 5.3.1 (*i.e.* removal or addition of variable features, but the mandatory features remain unchanged). Hereafter, the configured models are referred to as FT-FM' and FT-PLA' to avoid misunderstandings. Common Variability Language (CVL) (*Section 2.1.3*) needs to be adopted to specify architectural variability as a first-class concern at the FT-PLA' model (*Section 5.2*). By specifying variability systematically, we are able to produce the right product models automatically given a resolution model encompassing a set of preselected features, through CVL model-to-model transformation (*Figure 2.3* and *Figure 5.3*). This facility is mainly used by ArCMAPE to generate adaptation plans dynamically.

To overcome the research challenges related to AFTMs [63], ArCMAPE was designed on a reflective architecture to support separation of concerns between the adaptation and fault tolerance logics. The meta-level relies on an autonomic loop [82] (*Section 2.3*) and leverages the FT-FM' and FT-PLA' models at runtime for reasoning on the fault tolerance's dynamic behaviour and its environment. The base-level is composed by the FT-PLA' implementation. ArCMAPE provides *(i)* software components implementing the mandatory features; and *(ii)* a foundation on which variation points are represented by locations in the FT-PLA' where plug-in components, or variant components, may be added. The variant components should implement the target variable features. At runtime, depending on the context, ArCMAPE dynamically chooses suitable variant components to realize those variation points. The chosen variant components may provide better quality of service (QoS) or offer new services that did not make sense in the previous context (*e.g.* due to changes in client requirements).

We present two empirical studies to exemplify the use of ArCMAPE in practical settings, show its feasibility and to assess the time overhead it imposes to support late variability. If we have a high overhead it could outweigh the benefits of the dynamic adaptation itself [32]. The first empirical study, referred to as *e-credit*, is based on an enterprise application, described in [166], that provides instant credit decisions for in-store purchases. For this study we provide scripts implementing the target alternate services. The second empirical study, referred to as *e-tour*, is based on a web store for packages tour and leverages alternate services available on the Web to tolerate software faults. Outcomes from these studies suggest that ArCMAPE *(i)* is efficient to support and dynamically derive software fault tolerance techniques for FT-compositions tailored to the specific needs of different clients and contexts; *(ii)* can be easily extended and reused; and *(iii)* does not introduce an excessive time overhead to support dynamic management of software variability.

Finally, it is important to notice that the models leveraged at runtime by ArCMAPE can be specified and validated by using the model-driven infrastructure for developing PLAs (*Section 5.2*). The target alternate services, which are also variable features, can be selected by means of the guidelines to asses service diversity (Section 3.2). The study presented in Chapter 4 can also be a good starting point to make choices related to the other variable features and to implement them (*i.e.* different adjudicators and execution schemes) (*Section 4.4*).

6.2 ArCMAPE

ArCMAPE mediates the communication between clients and alternate services. When using ArCMAPE to tolerate software faults by leveraging n alternative services, we can have $n+1$ hardware units (Figure 2.6 (c)), and the software replicated is a service operation, that is, a program segment (Figure 2.6(d)). As illustrated in Figure 6.1, ArCMAPE relies on a dynamic component framework. Furthermore, ArCMAPE was designed on a reflective architecture. The base-level, or running system, is composed by software components implementing the FT-PLA' model. The meta-level relies on a MAPE-K loop (*Section 2.3*) and manipulates models to resolve variability at runtime by specific choices of the variant components at the base level. These models belong to the knowledge base. A dynamic adaptation corresponds to the dynamic instantiation of a software fault tolerance technique, *i.e.* a product, that satisfies high-level policies while maximizes the utility value, or subjective preferences, which is calculated based on pre-defined weights for QoS. The base-level is oblivious to the adaptation level, which is modularized as a stand-alone module, or *aspect* (*i.e.* Aspect-Oriented Programming) [167]. At runtime, the adaptation logic intercepts the running system when a requisition is sent by a client to ArCMAPE, since such logic decides which fault tolerance strategy will be provided to this requisition. Sensors and effectors are used, respectively, to gather details about and change the behaviour of the running system [82]. In the following subsections, we go into detail on the knowledge base, and the base and meta levels.

The Knowledge Base

The knowledge base is composed by the FT-FM'; FT-PLA' (including the *CVL variability* model); high-level policies defined as Event-Condition-Action (ECA) rules; reflection product model, which abstracts the running software fault tolerance technique; auxiliary mappings (mapping from features from the FT-FM' to choices of the CVL variability model); and details on the current context. These models are not causally connected with the runtime system, that is, they are not be directly modified to adapt the running system, but they reflect what happens at runtime (in the execution context) [68, 67]. On the contrary, these models serve as a basis for reasoning about the environment and determining a new configuration more adapted to the current context, if necessary [68]. The knowledge base offers enough details to fully automate the dynamic adaptation process, as described in Section 6.2. In the following, we describe the ECA rules in more detail.

Event-Condition-Action (ECA) Rules

ECA rules can be specified to handle known and common changes in a deterministic way [91, 21]. These rules allow us to map general conditions to actions, which represent a set of pre-selected features [21]. A rule is composed of several *clauses* [158], as illustrated in

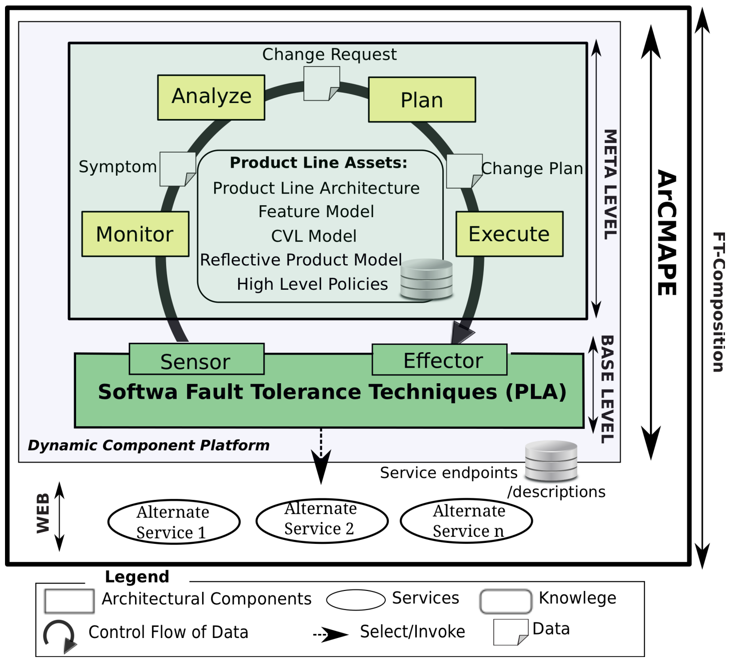

Figure 6.1: An Overview of ArCMAPE

Figure 6.2. Each *clause* is composed of one or more *test* and one or more *body*. The *clause* test is a classical condition expressed on *observables* that are objects abstracting a state of the current context. *Condition* is used to specify conditions. The clause body is a *dynamic adaptation* composed of two parts: places and changes. The *place* defines a variation point in the FT-PLA'. The *change* defines the chosen variable components(s), which should not violate feature constraints of the FT-FM'. To define the *Observable* concept, we have specified a context model. This model describes from a high level of abstraction the structure of the proposed fault tolerance mechanism itself. The goal is to enable the fault tolerance mechanism to observe its own structure and behaviour at runtime and to represent explicitly every entity that can be used as observables (to specify rules). The context model, which can be regarded as an excerpt of the ArCMAPE conceptual model, is illustrated in Figure 6.3.

In summary, *Adjudicator*, *Executor*, *Selector*, and *AlternateService* are specializations of *Service*, which is composed by a set of operations (*Operation*). Each operation is specified by means of IOPEs (Inputs, Outputs, Preconditions, Effects) (*Profile*) and presents a set of values for its quality of service attributes (*QoS*). The description of inputs and outputs encompasses their data types (either primitive or composed types) and should be based on a shared name convention (*DataTypeDesc*). A *Reliable Requisition*, defined as a requisition sent by a client to ArCMAPE, is associated to client preferences (*UserPreferences*), which is specified in terms of pre-defined weights for QoS (*QoSWeight*).

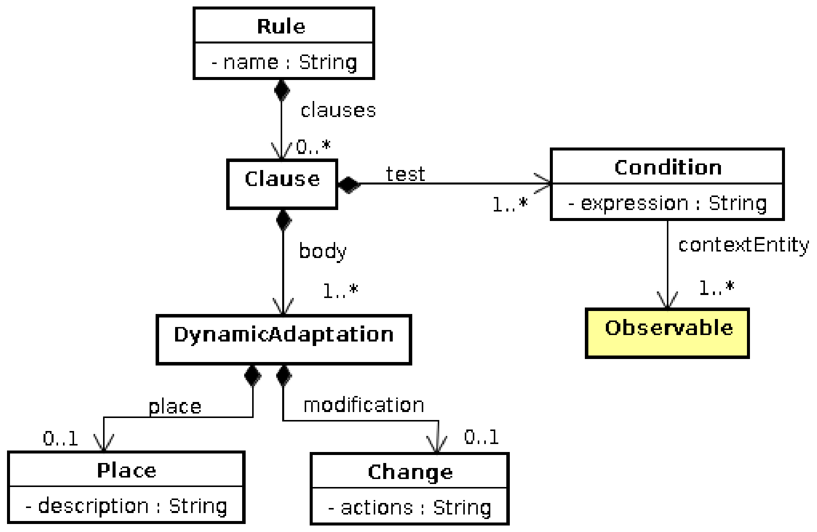

Figure 6.2: An overview of the rule metamodel

Furthermore a reliable requisition is also described by means of IOPEs. This common terminology is used by *Selector* to search for concrete alternate services (*AlternateService*) by matching IOPEs, in particular, by analysing client requests and descriptions of the services registered in the repository (*Repository*). Concrete alternate services are structured in a *NTuple*, the input of the execution scheme (*Executor*). After service executions, outputs are produced (*AlternateRes*). The alternate services and the results from their executions are structured in a *NTupeExecution*. From a *NTupeExecution*, a syndrome, the input to the adjudicator function consisting of at least the alternate outputs, is extracted (*Syndrome*) - Section 2.2.2. Given a syndrome, an adjudicator (*Adjudicator*) is responsible for identifying the correct or most acceptable result (*AdjudicatedResult*), if any.

Therefore, rules can be specified in terms of client preferences; number of alternate services realizing a requisition; result data types that adjudicators are able to judge; service descriptions; availability of adjudicators, execution schemes, and means to select alternate services; and so on. For example, to avoid inconsistencies, we could specify that *credit card payment* operations should always be invoked by means of a *conditionally sequential execution scheme* and their results judged by *acceptance tests*.

The Fault Tolerance Level

The fault tolerance level, also called base-level or running system, encompasses the implementation of the FT-PLA'. Implementation of its common components are already provided (*i.e. Executive*, *ExecutionMgr*, *AdjudicatorMgr* and *AltServiceSelectionMgr*) (*Fig-*

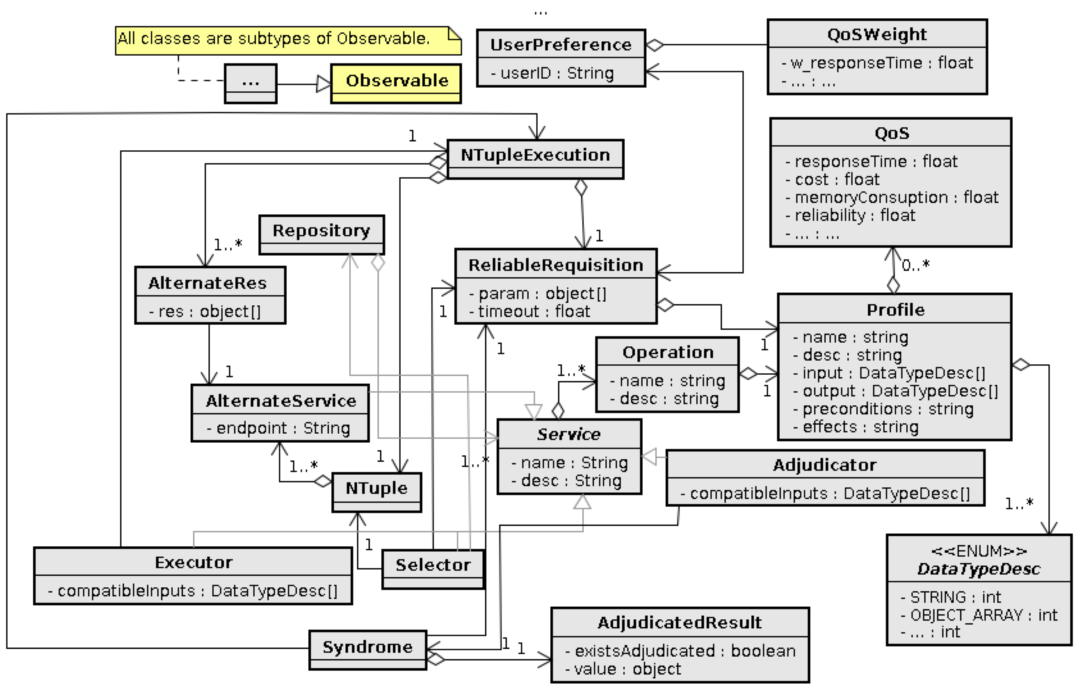

Figure 6.3: An overview of the context model

ure 6.4). The variable features are usually dependent on the domain - *e.g.* an adjudicator able to judge an application specific data type; an execution scheme to invoke SOAP (or JAVA RMI) services; and a set of alternate services realizing an application specific task. To support the variable features, ArCMAPE restricts the variability to adding and using variant components at predefined places. The variant components should realise the target variable features. More specifically, variation points are represented by locations in the FT-PLA' where one or more plug-in components, *i.e.* variant components, may be added [27].

At runtime, architectural variabilities are realised by specific choices of the variant component. The underlying dynamic component framework itself has a registration interface to give the variant component access to the framework through the registration of the access interface [27]. Through registration, the variant components make themselves known to the framework and, consequently, to ArCMAPE, which afterwards is able to access functionality from the variants [2]. Under these circumstances, the combination of a required access interface and a provided registration interface makes up the plug-in location [27]. Figure 6.4 shows an excerpt of the FT-PLA model in order to highlight the interface names, which were omitted in the previous figure (*Figure 5.3*) for the sake of clarity. For instance, as illustrated in Figure 6.4, the plug-in locations are required interfaces (*i.e. IR_ExecuteAltService*, textitIR_AdjudicateResults, and *IR_SelectService*). As a consequence, the variant components should be connected to these locations through their own provided interfaces. To fit, each variant component has to obey rules defined by ArCMAPE.

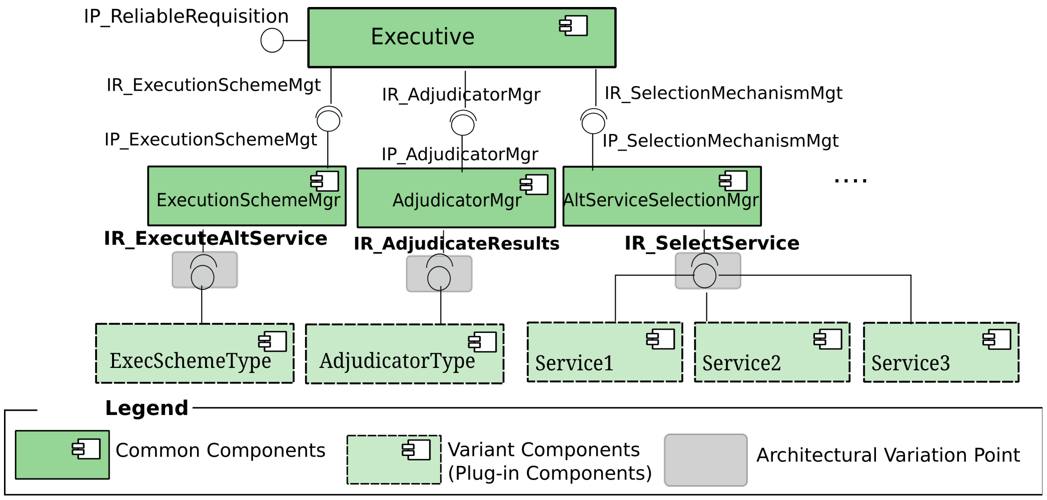

Figure 6.4: An excerpt of FT-PLA

Excerpts of the Detailed Design of the FT-PLA

In Figures 6.5-6.8, we show excerpts of the detailed design of *Executive*, *ExecutionSchemeMgr*, *AdjudicatorMgr*, and *AltServiceSelectionMgr*. These components are based on COSMOS* (*Section 2.1.2*). For instance, data types and exceptions are classes with private attributes and get/set operations, containing only the information needed by other components. Instances of data types and exceptions are used to exchange information between architectural elements, thus providing information hiding of the implementation classes [45]. Both packages *br...spec.dataTypes* and *br...spec.excep* were not specified by the general structure of the COSMOS* implementation model (Figure 2.1). We refer to the description of the context model (*Figure 6.3*) for further details on the specified data types. The implementation models (*i.e.* the *br...impl* packages) have been omitted from Figures 6.5-6.8 for the sake of clarity. Furthermore, each one of the components (*Executive*, *ExecutionSchemeMgr*, *AdjudicatorMgr*, and *AltServiceSelectionMgr*) provide meta-information by means of its *I...Manager* interface, as specified by COSMOS*.

The *Executive* component, illustrated in Figure 6.5, is responsible for orchestrating the execution of a fault tolerance strategy. It provides an interface used to configure the fault tolerance mechanism (the *definePolicies(...)* operation), and to execute a fault strategy itself (the *executeFaultTolerance(...)* operation). It requires interfaces to manage the selection of alternate services (*IAdjucatorMgt*), to execute alternate services (*IExecutionSchemeMgt*), and to judge results from these alternate services (*ISelectionMech-*

anismMgt). These required interfaces are realised by, respectively *AltServiceSelectionMgr*, *AdjudicatorMgr*, and *ExecutionMgr*. The execution of the *Executive* component can propagate exceptions to clients - the exceptions thrown by the execution of *AltServiceSelectionMgr*, *AdjudicatorMgr*, and *ExecutionMgr*. If an undeclared exception is caught after the execution of these components, *Executive* throws an *UndeclaredException* to clients. The *UndeclaredException* exception is used to allow us to attach failure semantics to exceptions associated with exceptional conditions that were not anticipated by the component's specification. In this way, clients can also deal with the undeclared exceptions in a systematic way [168]. The execution of the *Executive* component can also throw the *SelectorNotAvailable*, *ExecutorNotAvailable* and *AdjudicatorNotAvailable*, indicating that the there is no variant components implementing, respectively, the required selector, execution scheme and adjudicator.

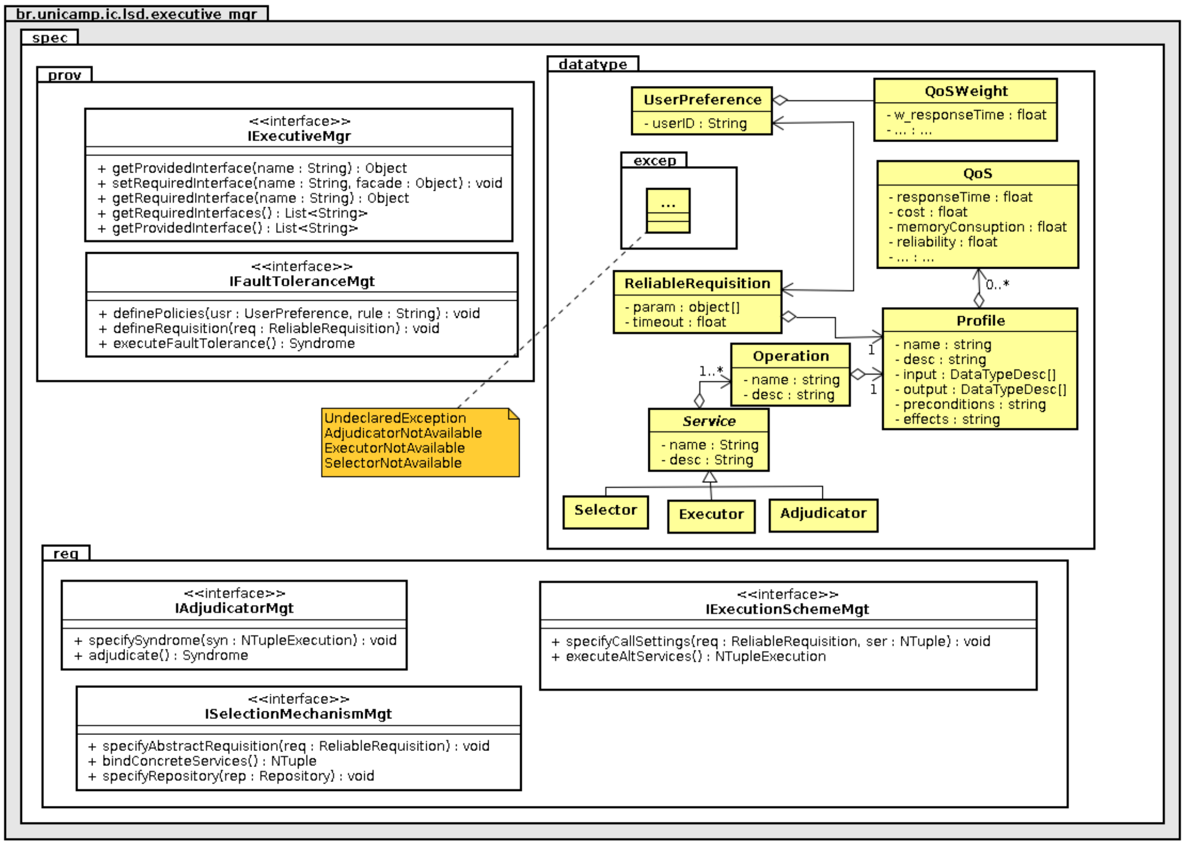
br.unicamp.ic.lsd.executive_mgr
spec
prov
<<interface>>
IExecutiveMgr
+ getProvidedInterface(name : String) : Object
+ setRequiredInterface(name : String, facade : Object) : void
+ getRequiredInterface(name : String) : Object
+ getRequiredInterfaces() : List<String>
+ getProvidedInterface() : List<String>
<<interface>>
IFaultToleranceMgt
+ definePolicies(usr : UserPreference, rule : String) : void
+ defineRequisition(req : ReliableRequisition) : void
+ executeFaultTolerance() : Syndrome
UndeclaredException
AdjudicatorNotAvailable
ExecutorNotAvailable
SelectorNotAvailable
datatype
UserPreference
- userID : String
QoSWeight
- w_responseTime : float
- ... : ...
excep
...
QoS
- responseTime : float
- cost : float
- memoryConsuption : float
- reliability : float
- ... : ...
0..*
ReliableRequisition
- param : object[]
- timeout : float
Profile
- name : string
- desc : string
- input : DataTypeDesc[]
- output : DataTypeDesc[]
- preconditions : string
- effects : string
1
1
Operation
- name : string
- desc : string
1..*
Service
- name : String
- desc : String
Selector
Executor
Adjudicator
req
<<interface>>
IAdjudicatorMgt
+ specifySyndrome(syn : NTupleExecution) : void
+ adjudicate() : Syndrome
<<interface>>
IExecutionSchemeMgt
+ specifyCallSettings(req : ReliableRequisition, ser : NTuple) : void
+ executeAltServices() : NTupleExecution
<<interface>>
ISelectionMechanismMgt
+ specifyAbstractRequisition(req : ReliableRequisition) : void
+ bindConcreteServices() : NTuple
+ specifyRepository(rep : Repository) : void

Figure 6.5: *Executive*: An excerpt of its detailed design

The *AltServiceSelectionMgr* component, illustrated in Figure 6.6, is responsible for managing the selection of alternative services. It defines a required interface, a variation point, called *ISelectService*. One or more variant components realising this required interface should be added. These variant components should provide concrete services matching the profiles contained by reliable requisitions (*Figure 6.3*). The *bindConcreteService(...):...* operation, from both *ISelectionMechanismMgt* and *ISelectSerivce* interfaces, raises the *NoConcreteServiceException* and *InvalidRepositoryException* exceptions, indicating, respectively, that no[1] services have been found and no repository has been found. These exceptions are declared in these operation signatures. The *description()* operation provides a description of the *Selector* plugged in.

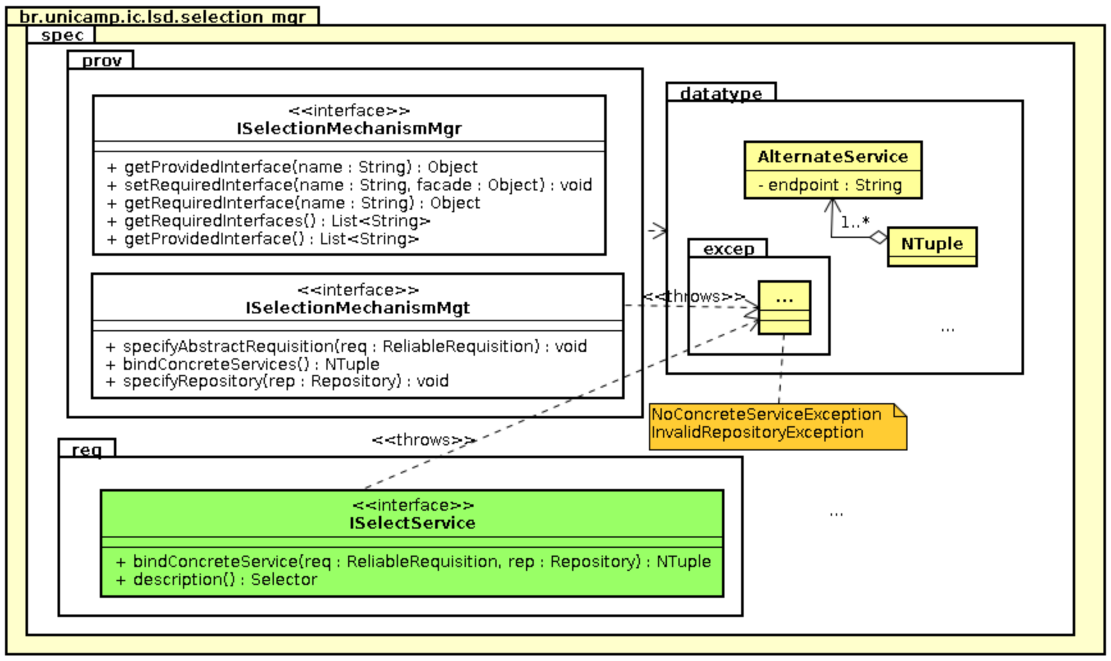
br.unicamp.ic.lsd.selection_mgr
spec
prov
<<interface>>
ISelectionMechanismMgr
+ getProvidedInterface(name : String) : Object
+ setRequiredInterface(name : String, facade : Object) : void
+ getRequiredInterface(name : String) : Object
+ getRequiredInterfaces() : List<String>
+ getProvidedInterface() : List<String>
<<interface>>
ISelectionMechanismMgt
+ specifyAbstractRequisition(req : ReliableRequisition) : void
+ bindConcreteServices() : NTuple
+ specifyRepository(rep : Repository) : void
datatype
AlternateService
- endpoint : String
1..*
NTuple
excep
...
<<throws>>
...
NoConcreteServiceException
InvalidRepositoryException
<<throws>>
req
<<interface>>
ISelectService
+ bindConcreteService(req : ReliableRequisition, rep : Repository) : NTuple
+ description() : Selector
...

Figure 6.6: *AltServiceSelectionMgr*: An excerpt of its detailed design - Extension Points (required interfaces)

The *ExecutionSchemeMgr* component, illustrated in Figure 6.7, is responsible for managing the execution of alternate services give a *ReliableRequisition* and a *NTuple*. It defines a required interface, a variation point, called *IExecuteAltService*. One or more variant components realising this required interface should be added. These variant components should execute the target alternate services in order to obtain their results and return the respective *NTupeExecution*. The *executeAltServices(..)* operation, from both *IExecutionSchemeMgt* and *IExecuteAltService* interfaces, raises the *TimeOutException*, and *ServicesNotAvailableException* exceptions, indicating, respectively, that no services have executed within the time allotted and no services were available. These exceptions are

[1]Notice that if at least one service has been found, depending on the adjudicator, a correct result might still be identified, consequently no exception should be thrown under this circumstance.

declared in these operation signatures. The *description()* operation provides a description of the execution scheme, or *Executor*, plugged in.

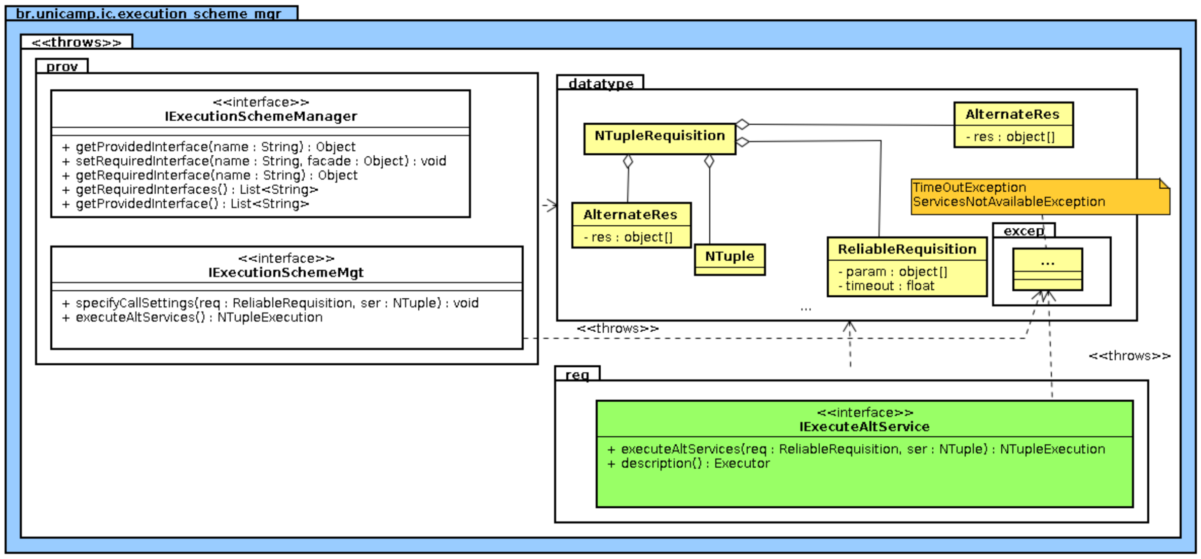
br.unicamp.ic.execution scheme mgr
<<throws>>
prov
<<interface>>
IExecutionSchemeManager
+ getProvidedInterface(name : String) : Object
+ setRequiredInterface(name : String, facade : Object) : void
+ getRequiredInterface(name : String) : Object
+ getRequiredInterfaces() : List<String>
+ getProvidedInterface() : List<String>
<<interface>>
IExecutionSchemeMgt
+ specifyCallSettings(req : ReliableRequisition, ser : NTuple) : void
+ executeAltServices() : NTupleExecution
datatype
NTupleRequisition
AlternateRes
- res : object[]
AlternateRes
- res : object[]
NTuple
ReliableRequisition
- param : object[]
- timeout : float
TimeOutException
ServicesNotAvailableException
excep
...
<<throws>>
<<throws>>
req
<<interface>>
IExecuteAltService
+ executeAltServices(req : ReliableRequisition, ser : NTuple) : NTupleExecution
+ description() : Executor

Figure 6.7: *ExecutionSchemeMgr*: An excerpt of its detailed design - Extension Points (required interfaces)

The *AdjudicatorMgr* component, illustrated in Figure 6.8, is responsible for managing the adjudication of results returned from the execution of alternative services. It defines a required interface, a variation point, called *IAdjucateResults*. One or more variant components realising this required interface should be added. These variant components should find out a *AdjudicatedResult*, if any, given a *Syndrome*. The *adjudicate(..)* operation, from both *IAdjucatorMgt* and *IAdjudicateResults* interfaces, raises the *NoAdjudicatedResultException* and *InternalErrorException* exceptions, indicating, respectively, that the adjudicator did complete processing, but was not able to find an adjudicated result given the syndrome; and the adjudicator has not completed examining the alternate results and an error occurred during adjudication (Figure 2.5). These exceptions are declared in these operation signatures. The *description()* operation provides a description of the *Adjudicator* plugged in.

The Adaptation Level

Unlike the FT-PLA, the adaptation level, or meta-level, is independent on the application, thus it can be reused across different domains. The input for the execution of the adaptation logic consists of the assets from the knowledge base. These assets serve as a basis for reasoning about the environment and determining a software fault tolerance technique

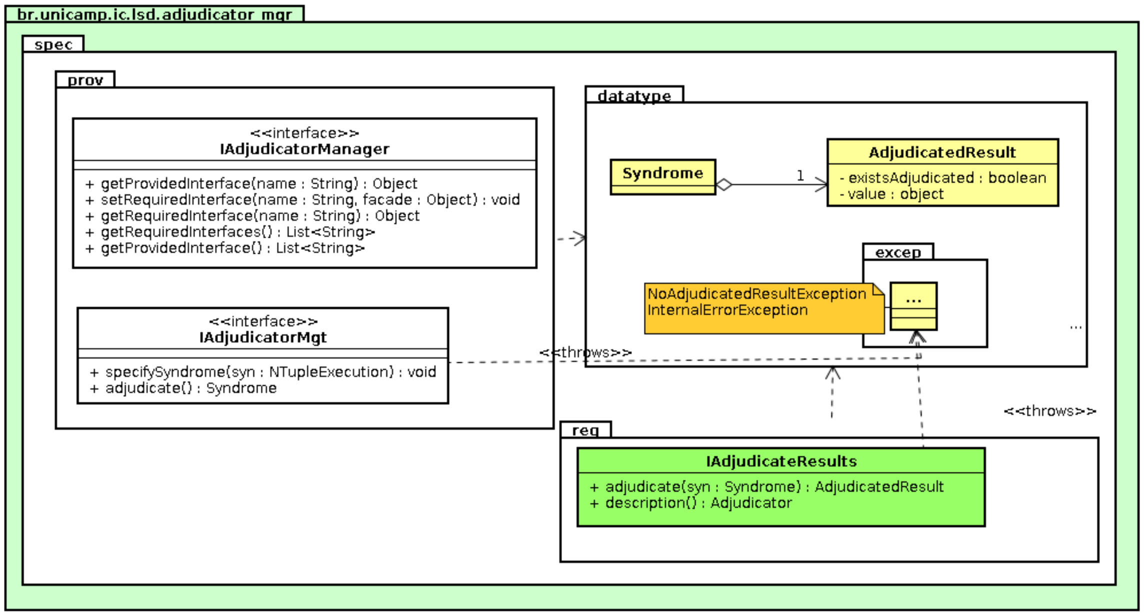
br.unicamp.ic.lsd.adjudicator_mgr
spec
prov
<<interface>>
IAdjudicatorManager
+ getProvidedInterface(name : String) : Object
+ setRequiredInterface(name : String, facade : Object) : void
+ getRequiredInterface(name : String) : Object
+ getRequiredInterfaces() : List<String>
+ getProvidedInterface() : List<String>
<<interface>>
IAdjudicatorMgt
+ specifySyndrome(syn : NTupleExecution) : void
+ adjudicate() : Syndrome
datatype
Syndrome
AdjudicatedResult
- existsAdjudicated : boolean
- value : object
1
excep
NoAdjudicatedResultException
InternalErrorException
...
<<throws>>
<<throws>>
req
IAdjudicateResults
+ adjudicate(syn : Syndrome) : AdjudicatedResult
+ description() : Adjudicator

Figure 6.8: *AdjudicatorMgr*: An excerpt of its detailed design - Extension Points (required interfaces)

suitable to the current context, if necessary. Therefore, the effort to design adaptation logic explicitly is decreased. The adaptation level, which is already implemented, relies on an autonomic control loop [82], for instance, the MAPE-K loop (*Section 2.3*).

The *Monitor* Component updates run-time objects related to the context model (*Figure 6.3*) when relevant changes appear in the running system's execution context. Details from the base-level are collected through *Sensors*. Sensors should log any significant changes that appear in the running system (*e.g.* addition/removal of components). For instance, when software components are removed, the *Monitor* is in charge of disabling selection of its related features to avoid inconsistencies. On the other hand, when software components are added, a software engineer is in charge of keeping the knowledge base up-to-date. In particular, *Monitor* monitors QoS values of software components implementing the variable features, including the alternate services. *Monitor* also computes the utility of the functional variable features based on pre-defined weights for QoS, as follows.

We adopt the concept of sub-trees, denoted as ST, to group a collection of features with a common (either identical or similar) functionality but possibly different QoS. These sub-trees are automatically identified by analysing the FT-FM'. For example, in Figure 5.2(b), we have two sub-trees: ST(String)={Majority, Consensus}, ST(Integer)={Mean, Median}. For each feature of a sub-tree (ST) is associated a QoS vector $[q_1, ..., q_m]$ (*e.g.* [availability, reliability, memory consumption, financial cost]). Suppose there are α QoS values to be maximized and β QoS values to be minimized. The utility function for a feature k in a sub-tree ST is defined as proposed by Yo and Lin [169]:

$F(k)_{ST} = \sum_{i=1}^{\alpha} w_i * \frac{q_{ai}(k)-\mu_{ai}}{\sigma_{ai}} + \sum_{j=1}^{\beta} w_j * (1 - \frac{q_{bj}(k)-\mu_{bj}}{\sigma_{bj}})$

where w is the weight for each QoS parameter set by a client ($0 < w_i, w_j < 1, \sum_{i=1}^{\alpha} w_i + \sum_{j=1}^{\beta} w_j = 1, \alpha + \beta = m$). μ and σ are, respectively, the average value and the standard deviation of the QoS attribute for all candidates in the ST sub-tree.

The ***Analyse*** **Component** selects the most suitable features if adaptation is required. First, *Analyse* ensures that clients and applications requirements are addressed by executing at runtime pre-defined ECA rules. Second, it selects a set of variable features that present the highest utility value while satisfying feature constraints from the FT-FM. More specifically, executing ECA rules might result in leaf features or features identifying sub-trees. For example, in Figure 5.2(b), if a *String* adjudicator is selected, we identify a sub-tree composed by *Majority Voter* and *Consensus Voter*, whereas if *Consensus* is selected, we have a leaf feature. In the first case, it is necessary to find out the feature that present the highest utility of the identified sub-tree. At this stage we handle unknown changes since QoS values vary over time. We adopt a greedy algorithm to recursively find out the most valuable feature of each sub-tree - a different maximization model could be adopted.

Once we have identified the most valuable adjudicator, execution scheme and alternate services in accordance with the predefined ECA-rules, if these features are different from the ones encompassed by the currently running fault tolerance strategy, a dynamic adaptation is required. As a consequence, the set of required features is passed to the plan function.

The ***Plan*** **Component** identifies the set of software components to be (un)bound in order to actually adapt the running system in agreement with the requisition sent by the *Analyse* component. First, it dynamically generates a *CVL resolution model (Figures 2.3*) in accordance with the set of required features. Subsequently, a new product model is generated at runtime accordingly by executing *CVL model-to-model transformation* (see example in Figure 6.4). By comparing the *new product model* with the *reflection product model*, the plan function dynamically generates a change plan [68]. Models comparisons are specified as follows. The TM and RM are defined as the set of, respectively, all elements of the *new model* and all elements of the *reflection model.* The corresponding configuration change, ΔC, is defined as a par $(\Delta C_{\oplus}, \Delta C_{\ominus})$, such as, $\Delta C_{\oplus}$ is the set of all the elements of the new model that have no matching counterpart in the reflection model, whereas $\Delta C_{\ominus}$ is the set of all the elements of the reflection model that have no matching counterpart in the new model [170]. Architectural components related to the model elements belonging to the $\Delta C_{\oplus}$ should be bound at runtime. On the other hand architectural components related to the model elements belonging to the $\Delta C_{\ominus}$ should be unbound at runtime. The $\Delta C_{\oplus}$ and $\Delta C_{\ominus}$ set are specified as follows:

$\Delta C = (\Delta C_{\oplus}, \Delta C_{\ominus})$

$\Delta C_{\oplus} := TM \setminus RM$

$\Delta C_{\ominus} := RM \setminus TM$

If ($\Delta C_{\oplus} = \emptyset \ and \ \Delta C_{\ominus} = \emptyset$), than no adaptation is required, otherwise it is necessary to update the reflection model ($RM = TM$) [170]. The change plan, composed by $\Delta C_{\oplus}$ and $\Delta C_{\ominus}$, is logically passed to the *Execute* component.

The *Execute* Component generates reconfiguration scripts on the fly in accordance with the adaptation plan in order to realize the causal connection between high-level abstractions from the meta-level and the running system (base-level). Effectors execute these scripts in order to finally instantiate the most adapted software fault tolerance technique. In particular, effectors use APIs for introspection and reconfiguration provided by the underlying dynamic component framework. To correctly handle the life-cycle of the running components we adopt the topological sorting of command suggested in [68]: components, which should be stopped, are stopped and then bindings are removed before removing components. Components are added before adding bindings and components, which should be (re-)started, are (re-)started.

The ArCMAPE Implementation

The ArCMAPE Implementation relies on Equinox, an implementation of the OSGi R4 core framework specification [171]. OSGi, also known as SOA in VM (virtual machine), is a module system and service platform for the Java programming language that implements a complete and dynamic component model [171]. The OSGi specifications enables a development model where applications are (dynamically) composed of many different (reusable) components. It also enables components to hide their implementations from other components while communicating through services.

To simplify the component configuration and assembly, at the PLA implementation level, we restrict the use of concrete types in favour of interfaces. In particular, common components, through their variation points, depend on interface types. Implementations of these interfaces are injected into the common component instances at runtime. As a consequence, the plug-in components, *i.e.* variant components, know statically to which variation points they have to connect, whereas the variation points, which provide the connection facility, do not know the variant components statically. For instance, the meta-level assumes the responsibility of locating and/or instantiating the variant components and simply supplying them to the variation points when needed. Therefore, our implementation combines inversion of control and dependency injection to allow late binding of variants [171]. We refer to the OSGi specification for further details on how the variant components can make themselves known to the OSGi runtime environment [171], and, consequently, to ArCMAPE.

Furthermore, *Sensors* and *Effectors* were implemented by means of APIs for introspection and reconfiguration provided by the OSGi platform [171]. Software components were implemented according to COSMOS* [77]. At runtime, to bind software components in the base-level, we employ a *service locator*, instead of traditional architectural connectors, that are used to break a component's dependency on component implementations. The 'service locator' is used by *Effectors* and acts as a simple run-time linker.

To programmatically and dynamically create *CVL resolution models* and invoke *CVL model-to-model transformation*, we extended a headless (a non-GUI) CVL Eclipse Plug-in. For product models comparisons, we use *EMF Compare* [172]. *EMF Compare* is a comparison engine that compares any kind of models producing as results a *diff* and a *match* model that specifies, respectively, the differences and the similarities between the source and the target models [68]. ECA-based rules are defined and managed by employing a rule engine, for instance, the *JRuleEngine engine* [173]. We modularize the meta-level as an *aspect*, which intercepts the *running system* when a requisition is sent by a client to ArCMAPE. For instance, we use Aspectj, an aspect-oriented extension for the Java programming language [167].

6.3 Evaluation

To exemplify the use of the proposed solution and to assess its effectiveness, we present two case studies.

6.3.1 E-credit

We present a case study, called *e-credit*, based on an enterprise application, described in [166], that provides instant credit decisions for in-store purchases. Among other functionalities of *e-credit*, we focus on the approval of credit line. When a customer in a retail store wishes to purchase an expensive item, the store clerk offers the customer an instant line of credit to make the purchase and pay later. If the customer is interested in obtaining the line of credit, the customer's information and a request for credit is sent to the remote *e-credit* server for approval. *E-credit*, in turn, invokes remote web services realizing the credit decision, which is based on the analysis of the customer's credit card records. This information can be purchased from multiple vendors at varying prices based on volume. Different vendors employ different algorithms to make a credit decision. A failure to make a decision could result in a customer not making a purchase or in approving a credit line that would have to be denied. To face software faults, we develop the *FT-e-credit*, which leverages ArCMAPE to support FT-compositions. An excerpt of the high-level architecture of *FT-e-credit* is shown in Figure 6.9. ArCMAPE acts as an architectural connector between *e-credit* and alternate services, used to implement fault tolerance techniques.

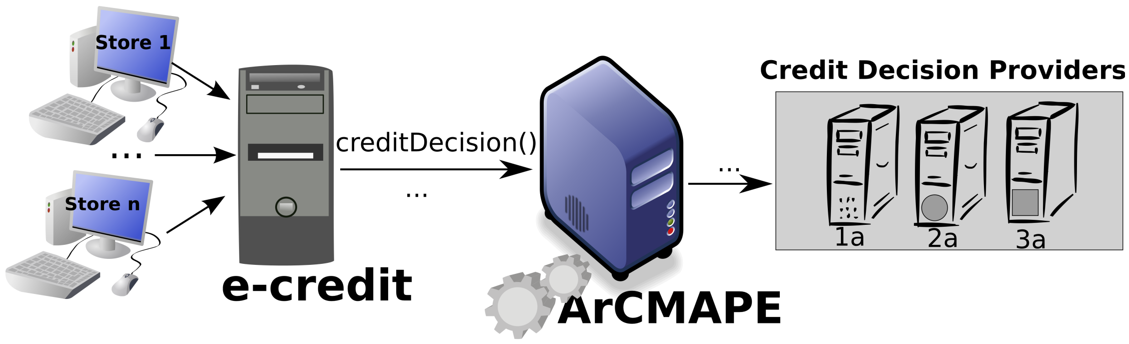

Figure 6.9: An excerpt of the high-level architecture of *FT-e-credit*.

Related to ArCMAPE, besides its mandatory features, we taken into account *(i)* three alternate services; *(ii)* three executions schemes (fully sequential, parallel and conditionally sequential ones); *(iii)* one majority voter able to judge *Boolean* data type; and *(iv)* one computer run-time acceptance tests, which detects anomalous answers (*e.g.* timeout exceptions or undefined operation code) [1]. To have an executable instance of ArCMAPE, we provide software components implementing these variable features. The target adjudicators were implemented based on pseudocode samples provided by Pullum [1]. To implement the variant execution schemes, we reused the implementation provided by Chen and Romanovsky [126] (*Table 4.3*). All these variant components were implemented in accordance with COSMOS* and were plugged in the base level. To implement the alternate services we have adopted scripts in order to random generate 1 or 0 (*i.e.* credit requisition has been approved or denied) and exceptions, for instance, undefined operation code [1]. The alternate services rely on the SOAP/WSDL protocol and were deployed in a remote server.

We customized the FT-FM (Figure 5.2) and the FT-PLA (Figure 6.4) in accordance with the target variable features. For instance, *FT-FM-ecredit* and *FT-PLA-ecredit* support 5 different software fault tolerance techniques, or products. To define the target fault model, we taken into account a fault taxonomy for service-oriented architectures defined by Bruning et al. [132], and general remarks about the target design solutions [1] (*Section 4.2.1*). The target fault models includes *discovery faults*, including no service found and timeout; and *execution fault*, including incorrect results, undefined operation codes and time out. To calculate the utility of each one of the functional variable features at runtime, we mocked the *Monitor* component implementation. For that, we programmatically simulated changes in QoS of variable features, for instance, realiability, memory consumption and financial costs. Changes on user preferences were also simulated (weights for QoS). To programmatically simulate these inherent instabilities of a real execution scenario is a sufficient condition to trigger dynamic adaptations.

We implemented a client application (a store) to send requisitions for credit decisions to *e-credit*, which, in turn, invokes ArCMAPE. For instance, 5000 requisitions were sent and the amount required was generated at random ($200 - $10000). Finally, we taken into account one ECA rule, to know: *if* the amount of credit line required is *greater than* $5000 *then* the alternate services should be executed in parallel *or* in a fully sequential way *and* results obtained should be judged by the majority voter; *otherwise* the alternate services should be executed in a conditionally sequential way *and* the results from services should be judged by the acceptance test.

During adaptation processes on DSPLs, the cost to monitor the context data and execute plans in order to deploy new configurations can be considered fixed [174]. However, it is critical to make the plan task as efficient as possible because it depends on the number of variable features [174]. As a consequence, we measured the time required to generate adaptation plans dynamically in order to analyse the overhead imposed by ArCMAPE to support late variability. In Table 6.1, we present the approximate time obtained to *(i)* execute each one of the target software fault tolerance techniques (which includes the time required to execute the alternate services - about 10 ms); and *(ii)* generate adaptation plans at runtime.

Table 6.1: Time overhead imposed by ArCMAPE (in milliseconds ms)

Activity Specification	**Mean**	**Median**	**St.Deviation**	**# Executions**
Tec. 1: Parallel + Majority + Alt. Services	1578.80	388.00	24566.26	902
Tec. 2: Fully Sequential + Majority + Alt. Services	1767.60	842.50	17202.45	889
Tec. 3: Parallel + ATs + Alt. Services	1597.594	409.00	24566.35	724
Tec. 4: Fully Sequential + ATs + Alt. Services	1829.58	863	17592.91	1002
Tec. 5: Conditionally Sequential + ATs + Alt. Services	453.36	445.6	176.15	1483
ArCMAPE: Generate Adaptation Plans Dynamically	82.58	73	38.93	2742

The median time required to generate adaptation plans at runtime is about 73 ms, which tends to be very close to the mean time (82.58). The slowest and the highest median time to execute a fault tolerance is, respectively, about 388 ms (Technique 1) and 846.00 ms (Technique 4). In the worst case, the median time to execute ArCMAPE is about 18.04% of the time to execute a software fault tolerance technique (Technique 1). Furthermore, the execution of the fault tolerance techniques might present higher values for the standard deviation due to inherent instabilities of the *Internet*, as the alternate services are available remotely. These results suggest that ArCMAPE does not introduce an excessive time overhead.

6.3.2 E-tour

We present a case study based on a web store for packages tour. Hereafter, we refer to this study application as *e-tour*. The *e-tour*, among other functionalities, sales packages for sightseeing. For transportation purposes, a tourist region is virtually divided into sectors by defining concentric circles, or zones, taking into account clients' interests. The tour price is automatically calculated based on the size of the radius encompassed by the zone a client is interested in. Under some circumstances, *e-tour* can automatically offer substantial discounts to packages purchased at least in 3 days in advance. The percentage discount can vary depending on the weather forecast for the places of interest. In the case of bad weather some destinations should be avoided (*e.g.* scenic overlooks), whereas packages for other destinations can benefit from a higher discount (*e.g.* museums and castles).

To qualify for discounts, clients should register themselves in *e-tour* by providing, among other information, a credit card number. The *e-tour* system invokes remote web services realizing the three described functional requirements, to known: *distance between two zip codes*, used to delimit zones, *weather forecast*, and *credit card number validation.* A failure to get this information could result in *(i)* customers not purchasing a package tour; *(ii)* in calculating prices and discounts improperly; or *(iii)* making difficult to create new accounts. To face software faults we develop the *FT-e-tour*, which employ ArCMAPE to support FT-compositions. An excerpt of the high-level architecture of *FT-e-tour* is shown in Figure 6.10. ArCMAPE acts as an architectural connector between *e-tour* and alternate services, used to implement fault tolerance techniques.

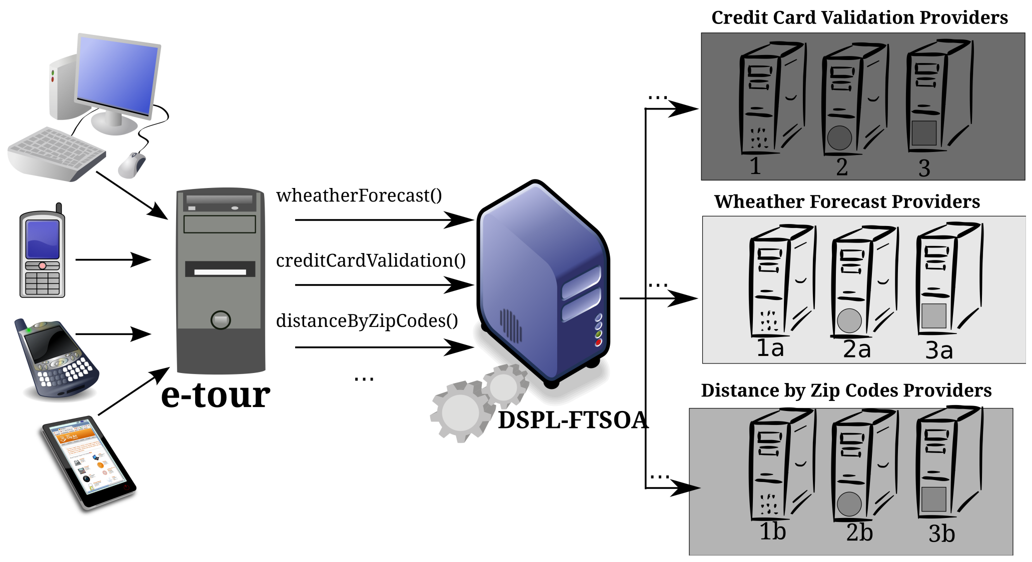

Figure 6.10: An excerpt of the high-level architecture of *FT-e-tour*.

To evaluate our solution within the scope of *FT-e-tour*, firstly, we identified multiple alternate services adhering to the three described functional requirements. Second, we investigated whether the target alternate services are in fact able to tolerate software faults. Third, we specify a family of software fault tolerance techniques supported ArCMAPE. We also provide software components implementing the target variable features. Fourth, we configure ArCMAPE accordingly and employ it to support the *FT-e-tour* (*Figure 6.10*). Subsequently, we performed experiments to evaluate the overhead imposed by ArCMAPE to support late variability.

Service Diversity Assessment

To implement *FT-e-tour* we leverage, for each required operation, three alternate services that are cost-free, third-party, stateless, read-only and based on SOAP/WSDL (*Section 2.1.1*). Such alternate services were selected from online services repositories [113, 114] and are representative of real scenarios (*i.e.* they are under control of third-parties and available on the Web). As described in Section 3.2, for each one of the requirements, its alternate services were executed under the same sequence of inputs. Based on the analysis of the output space, we check, by means nonparametric statistical tests, whether alternates are provided by different design and implementations [112]. Furthermore, for each functional requirement, we estimated the reliability achieved by its single services and by the FT-composition. In particular, we are interested in three-alternate voting systems since this is the minimum number of alternates that allows a service composition to tolerate faults from one of its services (*Section 2.2.2*). Outcomes from the investigation of service diversity are summarized in Table 6.2.

For the three functional requirements, we conclude that their alternate services are in fact diverse at the significance level α ($\alpha = 0.05$). To employ FT-compositions leveraging the target alternate services for *credit card validation*, *weather forecast*, and *distance by zip codes* improve the overall reliability, when compared to its respective single non-fault-tolerant services, in, respectively, 1.50%, 4.90%, and 1.30%. These outcomes suggest that the target alternate services are *efficient* to tolerate software faults.

Configuration of ArCMAPE

Besides the mandatory features (Figure 5.2), we taken into account *(i)* one majority voter able to judge Boolean data type; *(ii)* one formal voter able to judge whether integer values are either identical or similar (*i.e.* values are within an acceptable range [1]); *(ii)* one parallel execution scheme; *(iii)* one fully sequential execution scheme; and *(iv)* the selected alternate services. We customized the FT-FM (Figure 5.2) and the FT-PLA (Figure 5.3) accordingly by using the model-driven infrastructure for developing PLAs

Table 6.2: Reliability measurements of single non-fault-tolerant services ($Rel\_Est_{NFT\_S_{rv}}$) and FT-Composition employing voters and the target alternate services ($Rel\_Est_{FT\_SOA_r}$) - the reliability estimator values are between 0 and 1.

Requirements	Are services diverse?	$Rel\_Est_{NFT\_S_{rv}}$			$Rel\_Est_{FT\_SOA_r}$	Reliability Gains (%)
		v1	v2	v3		
Credit Card Validation	Yes	0.961	0.973	0.534	0.988	1.50%
Weather Forecast	Yes	0.946	0.343	0.705	0.995	4.90%
Distance By ZIP Codes	Yes	0.105	0.978	0.907	0.991	1.30%

(*Section 5.2*). The resulting assets are called as *FT-FM-etour* and *FT-PLA-etour*, for simplicity. For instance, *FT-FM-etour* and *FT-PLA-etour* support 12 different software fault tolerance techniques, or products.

To have an executable instance of ArCMAPE, we provide software components implementing the target variable features by reusing existing solutions (*Table 4.3*). The implementation of the formal voter is based on the pseudodoce provided by Abdeldjelil et al. [143]. To implement the majority voter we leveraged the *Java Comparable*, as suggested by Looker et al. [144]. To execute the target alternate services, we wrapped the execution schemes provided by Chen and Romanovsky [58]. All these variant components are based on COSMOS* and were plugged in the *FT-FM-etour*. To the best of our knowledge only the solutions by Chen and Romanovsky [58] and by Goncalves and Rubira [33] are available on the Web.

To define the target fault model, we taken into account a fault taxonomy for service-oriented architectures defined by Bruning et al. [132], and general remarks about the target design solutions [1] (*Section 4.2.1*). The target fault models includes *discovery faults*, including no service found and timeout; and *execution fault*, including incorrect results and time out. Furthermore, for this empirical study, we defined two ECA rules, to know *(i) credit card validation* requires adjudicators able to judge *Boolean* data types; and *(ii) weather forecast* and *distance by zip codes* requires adjudicators able to judge *integer* data type. We updated the ArCMAPE knowledge base by providing the FT-FM-etour and FT-PLA-etour models and the ECA rules.

Overhead Imposed by ArCMAPE

To evaluate the overhead imposed by ArCMAPE to support feature-based runtime adaptations, we implemented a client application to send requisitions to *FT-e-tour*, which, in turn, invokes ArCMAPE. For instance, 7000 requisitions were sent. The operation required by the client application (*credit card validation, weather forecast or distance by zip* *codes*) is specified randomly. To calculate the utility of each one of the functional variable features at runtime, we mocked the *Monitor* component implementation. For that, we programmatically simulated changes in QoS of the target variable features, for instance, reliability, memory consumption and financial costs. Changes on user preferences were also simulated (weights for QoS). To programmatically simulate these inherent instabilities of a real execution scenario is a sufficient condition to trigger dynamic adaptations.

To estimate the time and memory required to execute a software fault tolerance technique, we measured the time and memory taken to invoke the target alternate services. The overhead imposed to judge results from the target alternate services can be considered fixed and presents low values (we employ voters able to judge simple data types). In Figure 6.11 and Table 6.3 we illustrate, respectively, the approximate time and memory required to *(i)* invoke the alternate services by employing different execution schemes; and *(ii)* plan adaptations dynamically. For instance, in total, ArCMAPE adapted itself 2154 times to face changes in QoS related to the variable features. Each one of the execution schemes represented in Figure 6.11 was invoked around 1000 times.

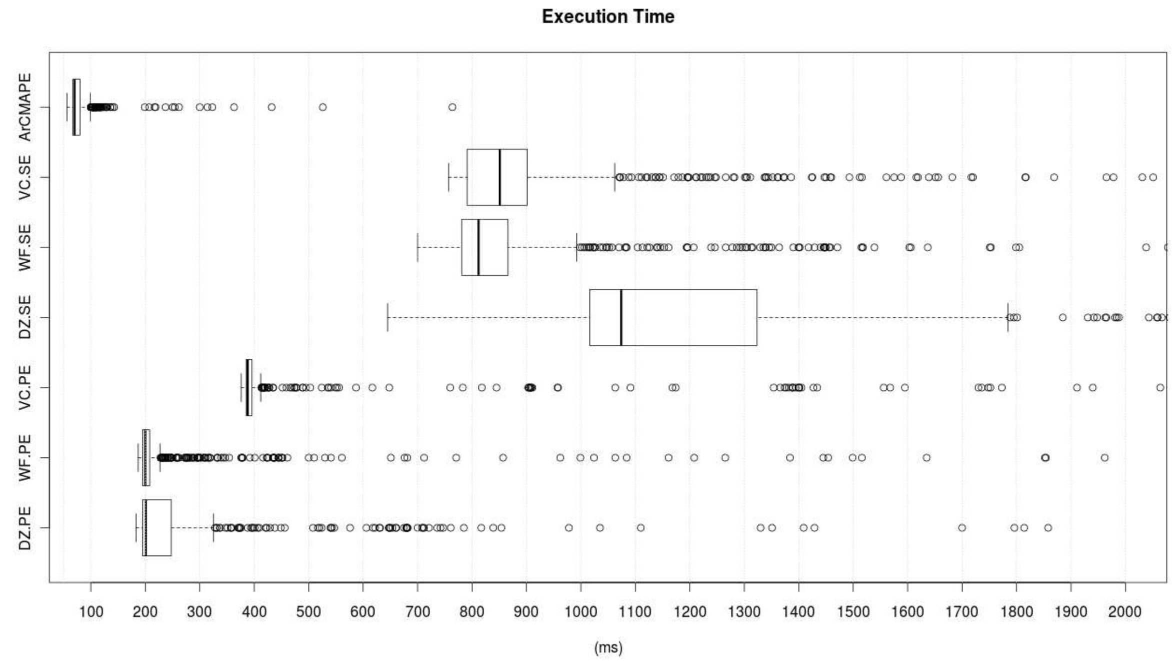

Figure 6.11: Execution Time (ms-milliseconds): DZ (Distance By Zip Codes), WF (Weather Forecast), VC (Validate Credit Card), PE (Parallel Execution Scheme), SE(Sequential Execution Scheme), ArCMAPE (specifically, time to execute the plan function)

As we can observe in Figure 6.11, the median time required to generate adaptation plans at runtime is about 70 *ms*. In half of the cases in which adaptations were performed, the time required to generate the respective plans range of 50 *ms* to 80 *ms*. The lowest 25% of execution times range from 50 *ms* to 70 *ms*. The top 25% of the time values are in the range of 80 *ms* to 100 *ms* with few exceptions (the outliers). As the median is nearer to the lower quartile (Q1) it suggests that to support late variability maintains, in the case, a low overhead cost. Furthermore, because the box is very thin relative to the whiskers, it suggests that a very high number of cases are contained within a very small segment of the sample. Consequently, the time required by ArCMAPE to generate adaptations plans tend to be very close to the mean.

Contrary to the time required to execute the target alternate services sequentially, the cost to execute them in parallel seems to be spread out over a large range of values. The slowest median times obtained for alternate service executions is around 200 *ms* and belongs to $DZ.PE$ and $WF.PE$. The slowest execution time to invoke the services in sequential and in parallel is, respectively, about 640 *ms* ($DZ.SE$) and 180 *ms* ($DZ.PE$). On the other hand, the highest execution time to invoke the alternate services in sequential and in parallel is, respectively, about 550000 *ms* ($VC.SE$) and 550000 *ms* ($VC.PE$) - we have omitted values higher than 2000 *ms* for the sake of clarity. Finally, it is interesting to notice that, in general, the time taken to execute services, even in parallel or in sequential, varies more than the one required by ArCMAPE to generate plans at runtime. The time to execute services also presents more extreme values that deviate significantly from the rest of the sample. These results might be related to inherent instabilities of the *Internet* - as the alternate services are available remotely.

As described in Table 6.3, ArCMAPE takes more memory to generate adaptation plans than the amount required to execute the alternate services. For instance, the mean and median memory consumed by ArCMAPE is about, respectively, $13,67$ MB and $13,49$ MB. Furthermore, ArCMAPE presents higher values for standard deviations, which suggest that the memory required to generate adaptation plans seems to be more widely spread. To execute services use about 9 MB of memory. We could observe by means of this empirical study that the highest amount of memory is consumed by ArCMAPE during the execution of *CVL model-to-model* transformation.

Table 6.3: Memory Consumption (MB - Megabytes): DZ (Distance By Zip Codes), WF (Weather Forecast), VC (Validate Credit Card), PE (Parallel Execution Scheme), SE(Sequential Execution Scheme), ArCMAPE (specifically, time to execute the plan function)

	DZ-PE	**WF-PE**	**VC-PE**	**DZ-SE**	**WF-SE**	**VC-SE**	**ArCMAPE**
Mean	8.50	9.58	9.60	8.50	9.52	9.58	13.67
Median	8.48	9.52	9.57	8.49	9.52	9.57	13.49
Standard Variation	0.071	0.15	0.13	0.073	0.068	0.10	2.70

6.3.3 Study Limitations

In this section we discuss some limitations of our empirical studies based on categories of validity threats presented by Wohlin et al. [75]. We identified possible threats to validity and, whenever it is applicable, the measures we took to reduce the risks.
External Validity: We identified one major threat to external validity: the target alternate services may not be representative of industrial practice since all of them are based on simple functionalities. This risk cannot be completely avoided due to the lack of functional requirements implemented by cost-free, functionally equivalent SOAP/WSDL-based Web Services [57, 112]. Regarding such risk, since we were looking for evidence on whether ArCMAPE imposes an excessive overhead to support late variability, the complexity of service functionality would have no negative effect on our final conclusions, because more complex systems usually take larger time to execute and consume more memory.
Conclusion Validity: We identified one main threat to conclusion validity: the number of employed variable features. Firstly, the low number of alternate services, as already mentioned, cannot be completely avoided due to the lack of cost-free web services. To mitigate this risk, we leveraged a number of different execution schemes and different adjudicators. As a result, for e-credit and e-tour, we obtained, respectively, 5 and 12 different products, which is representative of this area of research [18, 62, 68, 175].

6.4 Discussion

For the *e-tour*, to execute an experiment to assess service diversity was essential to allow us to improve the overall reliability by employing appropriate alternate services as part of the FT-compositions. On the other hand, measuring service diversity was time-consuming and a great deal of work. Nevertheless, while the empirical analysis of design diversity was a hotly-debated topic in the mid-1980s and early 1990s [1], no alternative guidelines to support diversity assessment so far have been proposed in the context of Web services. Furthermore, outcomes from our previous work point out the efficiency of the employed method [57, 133].

To specify a family of software fault tolerance techniques to develop *FT-e-credit* and *FT-e-tour* was greatly facilitated by two main facts. First, a general feature model and product line model was previously defined in order to identify commonalities and variabilities among various software fault tolerance techniques (*i.e.* models in Figure 5.2 and Figure 5.3). Secondly, the employed models are supported by existing tools, identified by the proposed model-driven infrastructure for developing PLAs (*Section 5.2*). This model-driven infrastructure further facilitates the execution of offline activities, as they are required when design models are leveraged at runtime [68] (*e.g.* modelling, validation and evolution). Consequently, it was straightforward to specify customized models. Furthermore, it is important to emphasize that the offline activities are required to guarantee efficiency and correctness at runtime [68].

By instantiating ArCMAPE, we realized that the effects of changes, such as the removal and addition of features, were localized. This mainly happened due to two reasons. First, ArCMAPE explicitly supports well-defined extension points in which software components implementing variable features can be plugged in. The explicit integration of plug-in mechanisms in the PLAs reduces the effort for the composition of the final products [2]. Second, ArCMAPE supports an explicit separation of concerns between the adaptation and the fault tolerance logic. By reusing models from design time to automatically generate adaptation plans at runtime is a suitable approach to decrease the effort to design adaptation logic explicitly. Therefore, ArCMAPE supports maintainability and flexibility of software fault tolerance techniques applied to SOAs.

Outcomes from the empirical studies suggest that ArCMAPE seems to introduce a considerable overhead related to the memory usage when compared to the amount taken to execute a fault tolerance technique. Nevertheless, it is important to notice that although ArCMAPE implies in higher memory consumption, it does not impact negatively the execution time. Furthermore, while the execution of the alternate services manipulates, for instance, simple data types (*i.e.* Boolean, integer and string), the generation of adaptation plans takes into account the feature model, the product line architecture and ECA rules. Consequently, these results for memory consumption were already expected. Nevertheless, to investigate eventual memory bottlenecks is a fundamental issue for future work.

In fact, we have already identified that the *CVL model-to-model* transformation is the most costly function employed to generate adaptation plans dynamically. As already mentioned, our solution reuses a third *CVL implementation* to generate product models [3]. Although we could employ more lightweight approaches to specify and generate these models at runtime, CVL ensures that software variability is systematically defined and resolved. To ensure the correctness of all abstraction models leveraged at runtime is particularly critical issue for a solution that should protect against software design faults instead of inserting new ones.

6.5 Related Work

We address work related to existing solutions for FT-compositions; and solutions for DSPLs.

Solutions for FT-compositions

A systematic literature review of diversity-based solutions for FT-compositions has shown us the main drawbacks and contributions of these solutions. Firstly, some solutions support only one software fault tolerance technique which is not customizable to face changing requirements [144, 59, 36, 152, 141]. Secondly, even when authors claim that their solutions can be customized, they not explicitly describe how to extend their solutions, for example, what interfaces must be implemented when inserting a custom adjudicator [40, 41, 147, 126, 33]. Furthermore, outcomes from the systematic literature review suggest that there is a lack of solutions able to manage the whole process of building FT-compositions according to clients' specific requirements. Our solution supports selection of alternate services, specification and design of software fault tolerance techniques, and a framework to manage these techniques dynamically in accordance with high-level policies and the current context.

Some solutions are able to face changing requirements at runtime by selecting and executing the most appropriate software fault tolerance technique [34, 33, 126]. Nevertheless, these solutions do not considerer separation of concerns between adaptation and fault tolerance logics. The solution by Zheng and Lyu [148, 56, 34] supports different fault tolerance strategies and the authors described in detail a dynamic fault tolerance strategy selection algorithm. Their selection algorithm could be incorporated into ArCMAPE instead of selecting a software fault tolerance technique that maximizes the utility function.

Finally, it is important to emphasize that ArCMAPE is an extension of our very preliminary solution for FT-compositions [40], which presents some drawbacks: *(i)* models leveraged to runtime were specified in an ad hoc way, thus, it is difficult and error prone to specify and validate them; *(ii)* fault tolerance and adaptation logics were not clearly separated (*e.g.* the monitor component was employed at the base-level);*(iii)* adaptations were defined by ECA rules (it might be not scalable [91]).

Solutions for DSPL

MADAM [176] leverages architectural models at runtime to reason about adaptation and uses the adaptation capabilities offered by dynamic platforms to supports runtime variability [176]. The EU MUSIC project [61] extends the solution developed in MADAM to ubiquitous computing and SOA (e.g. to discover services dynamically). At this point, our solution is similar to the MUSIC solution. Morin et al [68, 67] use model-driven engineering (MDE) and aspect-oriented modelling techniques to support runtime variability and dynamic generate product models. These solutions [68, 176, 61] also rely on models comparisons to generate adaptation plans. Nevertheless, in our work, product models are systematically and dynamically generated by *CVL transformations*.

Baresi and Pasquale [18] enriches Business Process Execution Language (BPEL) compositions with software variability by using CVL, which makes it possible to easily generate a DSPL. They employ a dynamic version of BPEL, called DyBPEL, to manage and run the DSPL. Ayora et al. [62] manage software variability at design time and runtime by using the CVL. Software variability is defined in business processes [62] and the authors also describes how the base, variability and resolution models are leveraged at runtime by a MAPE-K loop to adapt business processes. These solutions [18, 62] do not use CVL transformation dynamically neither support comparisons between reflection and target models.

6.6 Summary

Within distributed SOA applications, the bulk of the complexity is situated in the application layer, and there will often remain design faults which eluded detection despite rigorous and extensive testing and debugging [37]. Due to the low cost of reusing functionality equivalent services, or alternate services, several solutions based on design diversity have been proposed to support fault-tolerant composite services, or FT-compositions. Nevertheless, it is challenging to apply existing solutions for FT-compositions due to the lack of capabilities to adapt themselves at runtime to cope with dynamic changes of *(a)* user requirements and *(b)* the level of quality of services (QoS).

In this chapter we propose ArCMAPE, an DSPL infrastructure to support self-adaptation of FT-compositions in close accordance with high-level policies and the current context. ArCMAPE allows leveraging a family of software fault tolerance techniques based on design diversity at runtime and instantiates the most suitable one through dynamic management of software variability. The instantiation of suitable fault tolerance techniques is achieved by means of feature-based runtime adaptations, thus the effort to design adaptation logic explicitly is decreased. In this chaper, we provided a detailed design of ArCMAPE in order to facilitate its reuse in practical settings. Furthermore, we exemplified the use of the proposed solution and evaluated it by employing ArCMAPE to support FT-compositions within two web applications: one for instant credit decisions, and the other one for selling packages tour. Outcomes from these empirical studies suggest that ArCMAPE *(i)* is efficient to support self- adaptation of fault-tolerant service compositions; *(ii)* does not introduce an excessive overhead to dynamically adapt itself; and *(iii)* is easily reused and customized to be used in practical settings. To the best of knowledge, to employ (dynamic) software product line to support software fault tolerance for service oriented systems is a novel solution.

In Chapter 7, we present the final conclusions of this thesis.

Capítulo 7

Conclusões e Trabalhos Futuros

Neste capítulo, apresentamos as conclusões desta tese (*Seção 7.1*), suas principais contribuições (*Seção 7.2*) e publicações correspondentes (*Seção 7.3*). Por fim, discutimos direções para trabalhos futuros (*Seção 7.4*).

7.1 Conclusões

Esta tese apresentou estudos empíricos, infraestruturas e métodos para apoiar o projeto e implementação de sistemas confiáveis orientados a serviços que usam técnicas de tolerância a falhas de software baseadas em diversidade de projetos. Em linhas gerais, serviços Web funcionalmente equivalentes, aqui chamados de serviços alternativos, são estruturados em composições de serviços que alavancam técnicas baseadas em diversidade de projetos para tolerar falhas de software. A seguir, descrevemos com mais detalhes os desafios e soluções apresentados nesta tese. As questões de pesquisa são apresentadas a seguir.

Questão de Pesquisa 1 (QP1) Como mensurar se serviços alternativos são diversos e eficientes para tolerar falhas de software quando alavancados por técnicas baseadas em diversidade de projetos?

Questão de Pesquisa 2 (QP2) Quais as implicações de se utilizar serviços alternativos para tolerar falhas de software?

Referente a QP1, propusemos uma infraestrutura, apresentada no Capítulo 3, para apoiar a organização e execução de estudos empíricos que visam investigar se um dado conjunto de serviços alternativos são eficientes para tolerar falhas de software quando executados por intermédio de técnicas de tolerância a falhas baseadas em diversidade de projetos. A solução proposta encopassa diretrizes para analisar *(i)* se os serviços alternativos são de fato providos por diferentes projetos; e *(ii)* a confiabilidade alcançada por estes serviços quando estruturados em serviços compostos para tolerar falhas de software. Utilizamos estas diretrizes para avaliar a diversidade de serviços alternativos baseados em SOAP/WSDL e gratuitamente disponíveis na Internet. Resultados obtidos apontam a viabilidade e eficiência da solução proposta. Num segundo momento, para responder a QP2, conforme descrito no Capítulo 3, utilizamos esta infraestrutura num estudo mais abrangente a fim de obtermos um maior entendimento das implicações de se utilizar técnicas de diversidade de projetos para apoiar serviços compostos confiáveis. A partir deste estudo, obtivemos evidências que sugerem que os benefícios de se utilizar diversidade de projetos para apoiar maiores índices de confiabilidade em sistemas orientados a serviços não são tão triviais, embora sim, sejam alcançáveis (Seção 3.4).

Questão de Pesquisa 3 (QP3) Quais escolhas de projetos são apoiadas por soluções existentes para composições de serviços tolerantes a falhas de software? Quais as principais diferenças entre essas escolhas de projeto no que tange os requisitos de qualidade?

Realizamos um estudo sistemático, descrito no Capítulo 4, que apoiou a classificação das soluções existentes para composições de serviços tolerantes a falhas conforme uma taxonomia, que lista as principais decisões de projetos e suas respectivas soluções de projeto referentes às diversas técnicas de tolerância a falhas de software (Seção 4.2). Além disso, apresentamos algumas considerações no que tange as diferenças em termos de requisitos de qualidade das diferentes soluções de projeto identificadas (Seção 4.2.1). A partir dos resultados obtidos é possível identificar mais facilmente as principais contribuições e limitações das soluções existentes, bem como escolher soluções mais apropriadas para diferentes contextos. Ademais, essa comparação das soluções de projetos pode ser útil na concepção de modelos de falhas conforme as soluções escolhidas.

Questão de Pesquisa 4 (QP4) Como apoiar uma infraestrutura que acomode de forma planejada diferentes técnicas de tolerância a falhas?

Como apresentado no Capítulo 5, propusemos uma família de técnicas de tolerância a falhas de software para apoiar serviços compostos confiáveis. É sabido que linhas de produtos de software apoiam o reúso planejado e customização de produtos similares a um custo razoável [28]. Em particular, os artefatos desta família foram especificados e projetados de forma sistemática, consistente e coordenada mediante o uso de uma infraestrutura dirigida por modelos para apoiar a implementação de arquiteturas de linhas de produtos em geral. Como resultado, obtivemos a implementação de uma arquitetura de linha de produtos que contempla o projeto de diferentes técnicas de tolerância a falhas. Essa implementação inicial pode ser refinada a fim de apoiar técnicas de fato executáveis, inclusive a partir da reutilização da implementação de soluções existentes.

Questão de Pesquisa 5 (QP5) Como apoiar de forma adequada uma infraestrutura autoadaptiva que apoia a instanciação de técnicas diferentes de tolerância a falhas em resposta a mudanças ocorridas no contexto?

Como descrito no Capítulo 6, propusemos uma infraestrutura, baseada em linhas de produtos de software dinâmicas, que apoia uma família de técnicas de tolerância a falhas (*Seção 6.2*), de forma que a técnica mais adaptada seja instanciada em tempo de execução conforme a percepção do contexto, incluindo mudanças nos requisitos de clientes e flutuações nos valores de qualidade de serviços. Para apoiar a configuração dinâmica de produtos, utilizamos o *loop* autonômico, tal que a técnica mais apropriada é instanciada por intermédio de gerenciamento dinâmico de variabilidades de software (*Seção 6.2*). Os estudos de caso *e-credit* (*Secção 6.3.1*), uma linha de aprovação de crédito *online*; e do *e-tour* (*Seção 6.3.2*), uma loja virtual para vendas de pacotes turísticos, mostraram que a infraestrutura proposta é eficiente para apoiar composições de serviços confiáveis e autoadaptativas e não implica em um *overhead* excessivo para apoiar gerenciamento dinâmico de variabilidades de software.

A avalição das soluções propostas para responder às questões de pesquisas também apontaram direções para trabalhos futuros, conforme descrito em detalhe na Seção 7.4.

7.2 Contribuições

As contribuições desta tese situam-se em algumas subáreas de engenharia de software a saber: projeto e implementação de sistemas tolerantes a falha de software, em particular, sistemas confiáveis orientados a serviços; estudos empíricos em engenharia de software; desenvolvimento de sistemas de software dirigido a modelos; e projeto e implementação de sistemas autoadaptativos. A seguir, apresentamos as contribuições principais e secundárias desta tese. Apresentamos também outras contribuições não relacionadas ao foco princial deste trabalho, no entanto, relacionadas a trabalhos desenvolvidos no grupo de pesquisa em Engenharia de Software do IC-UNICAMP.

7.2.1 Contribuições Principais

As principais contribuições relacionadas ao trabalho desenvolvido são:

- *Utilização de linhas de produtos de software para apoiar requisitos não-funcionais em sistemas orientados a serviços.* As soluções existentes baseadas em linhas de produtos de software geralmente apoiam variabilidades em requisitos funcionais [18, 67, 69, 71, 62]. As soluções existentes para apoiar composições de serviços tolerantes a falhas que alavancam técnicas baseadas em diversidade de projetos não apoiam requisitos de reusabilidade e manutenção e não apoiam a adaptação dinâmica do mecanismo de tolerância a falhas de forma adequada [36, 37, 38, 33, 58, 39, 34, 40, 41, 59]. Nesta tese, apresentamos uma solução baseada em linhas de produtos de software, que explora a variabilidade de software existente nas técnicas de tolerância a falhas e nas mudanças ocorridas no ambiente de execução, para a implementação de composições de serviços tolerantes a falhas e autoadaptativas.

- *Um maior entendimento sobre as implicações de se utilizar técnicas baseadas em diversidade de projetos para tolerar falhas de software de serviços (Capítulo 3).* As soluções existentes para serviços compostos confiáveis que alavancam técnicas de tolerância a falhas baseadas em diversidade de projetos assumem, intrinsicamente, que serviços alternativos são sempre eficientes para tolerar falhas de software. A partir dos estudos empíricos que realizamos, constatamos que em algumas situações, contrariando o esperado, utilizar serviços simples pode gerar melhores resultados do que adotar técnicas de diversidade de projetados. À vista disso, é extremamente importante selecionar um conjunto de serviços alternativos adequados para compor uma composição de serviços efetivamente confiável. Embora existam estudos semelhantes realizados nas décadas de oitenta e noventa [49, 50, 51, 52], seria complicado extrapolar as conclusões desses estudos para o contexto de sistemas orientados a serviços, que apresentam características bastante peculiares. Este estudo foi possível mediante a utilização de um conjunto de diretrizes (Capítulo 3) para avaliar o quão eficiente um conjunto de serviços alternativos é para tolerar falhas de software.

- *Uma solução abrangente que avança o estado da arte no que tange a concepção e implementação de sistemas confiáveis orientados a serviços mediante o uso de técnicas de tolerância a falhas de software baseadas em diversidade de projetos.* A solução proposta contempla desde a seleção de serviços funcionalmente equivalentes, que são eficientes para tolerar falhas de software; até a especificação, projeto e implementação de uma família de técnicas de tolerância a falhas; e instanciação dinâmica da técnica mais apropriada conforme políticas de alto nível pré-definidas e percepção do contexto corrente. Até onde sabemos, não há uma solução na literatura que contemple todas estas etapas relativas ao desenvolvimento de sistemas confiáveis orientados a serviços [36, 37, 38, 33, 58, 39, 34, 40, 41, 59].

- *Uma revisão sistemática das principais soluções de projeto apoiadas por soluções para serviços compostos confiáveis que alavancam serviços alternativos para tolerar falhas de software (Capítulo 4).* A partir da análise das soluções existentes para serviços compostos confiáveis, identificamos de forma sistemática e efetiva suas principais contribuições e limitações. Tais soluções, até então, eram descritas a partir de diferentes contextos conceituais e técnicos, dificultando o seu entendimento e comparação. Nosso estudo facilita a escolha de soluções mais apropriadas para determinados requisitos, além de apontar direções para pesquisas futuras no que tange a concepção e desenvolvimento de sistemas confiáveis orientados a serviços.

- *Uma família de técnicas baseadas em diversidade de projetos para tolerar falhas de software de serviços (Capítulo 5).* A linha de produtos proposta contempla tanto requisitos de técnicas de tolerância a falhas baseadas em diversidade de projetos, quanto requisitos inerentes a sistemas orientados a serviços. Ao adotar abordagens de linhas de produtos, uma técnica que apoia a reutilização de forma estruturada e planejada [27, 87], obtivemos uma solução que apoia requisitos de extensão, manutenção e evolução. Tais requisitos de qualidade são essenciais para sistemas orientados a serviços, que em geral apoiam vários clientes, comumente com requisitos conflitantes [17, 16]. Até onde sabemos, não há na literatura uma solução propondo uma família de técnicas de tolerância a falhas para sistemas orientados a serviços.

• *Uma infraestrutura dirigida por modelos para o desenvolvimento de arquiteturas de linhas de produto (Capítulo 5)* Esta infraestrutura garante que modelos referentes à concepção e desenvolvimento de arquiteturas de linhas de produtos são coordenados de forma eficiente e são consistentes entre si. Além disso, a infraestrutura apoia um conjunto de atividades, tipicamente executadas em tempo de projeto, que são essenciais para garantir a corretude de arquiteturas de linhas de produto como: especificação de modelos valendo-se de um apoio ferramental; validação de modelos; e transformação automatizada de modelos de alto nível de abstração para modelos de nível de abstração mais baixo [67]. No contexto deste trabalho, em particular, estas atividades são particularmente importantes, pois aumentam a corretude de modelos que são utilizados num processo de tolerar falhas de software. Caso todos estes modelos fossem especificados e mantidos manualmente, poderia haver uma maior inserção de erros, contrariando o propósito principal da solução. Além disso, a junção de ferramentas e diretrizes num método sistemático para especificar e implementar arquiteturas de linhas de produtos de forma semi-automatizada corresponde a uma contribuição original.

• *Uma infraestrutura baseada em linhas de produtos de software dinâmicas para apoiar serviços compostos confiáveis e autoadaptativos que alavancam serviços alternativos para tolerar falhas de software (Capítulo 6).* A infraestrutura para serviços compostos confiáveis apoia soluções para os diferentes desafios de pesquisas inerentes aos mecanismos autoaptativos para tolerância a falhas em geral, a saber: *(i)* separação de interesses entre a lógica de adaptação dinâmica e a lógica de tolerância a falhas; *(ii)* uso de modelos de abstração para facilitar a especificação do comportamento dinâmico da estratégia de tolerância a falhas; *(iii)* configuração da estratégia de tolerância a falhas em tempo de execução em conformidade com alterações realizadas nos modelos de abstração; e *(iv)* alterações dos modelos de abstração em conformidade com percepção do contexto. Estas soluções foram propostas mediante utilização de linhas de produtos de software dinâmicas. Em particular, foi utilizado um loop autonômico [82] para gerenciamento dinâmico das variabilidades de software. Até onde sabemos, não há na literatura mecanismos autoadaptativos de tolerância a falhas baseada em diversidade de projetos para sistemas orientados a serviços que contemplam soluções para todos estes desafios [36, 37, 38, 33, 58, 39, 34, 40, 41, 59].

- *Uma solução abrangente para uma família de técnicas de tolerância a falhas baseadas em diversidade de projetos para apoiar serviços compostos confiáveis.* Considerando a utilização conjunta da infraestrutura dirigida por modelos para implementar arquiteturas de linhas de produtos e da infraestrutura para apoiar o gerencialmente dinâmico de variabilidade de software, temos uma solução abrangente para especificar, implementar e executar uma família de técnicas para tolerar falhas de software de serviços. Esta solução abrangente contempla desde atividades tipicamente executadas em tempo de projeto, até tomadas de decisão em tempo de execução com diretrizes de reúso, extensão e evolução é uma contribuição original no que tange à concepção e desenvolvimento de sistemas confiáveis orientados a serviços.

7.2.2 Contribuições Secundárias

As principais contribuições secundárias relacionadas à esta tese são:

- *Uma infraestrutura dirigida por modelos para linhas de produtos de software dinâmicas.* Embora a infraestrutura dirigida por modelos e a infraestrutura para gerenciamento dinâmico de variabilidade de software tenham sido exemplificadas e validadas no contexto de sistemas confiáveis orientados a serviços, é de suma importância salientar que estas infraestruturas poderiam ser reutilizadas em outros domínios. Elas, essencialmente, processam modelos estruturados, que poderiam conter características referentes a quaisquer outras linhas de produtos de software.

- *Geração automática de planos de adaptação dinâmica em sistemas autoadaptativos valendo-se de uma especificação sistemática de variabilidades arquiteturais.* Até onde sabemos, gerar planos de adaptação dinamicamente a partir da especificação e resolução sistemática de variabilidades arquiteturais é uma contribuição original. Mais especificamente, utilizamos CVL, uma linguagem genérica para especificação de variabilidades de software, que apoia a geração automática de modelos de produtos em conformidade com uma configuração de características. Embora algumas soluções diversas existentes utilizem CVL para gerar modelos de produtos estaticamente [18, 62, 73, 163], nossa solução inova ao gerar estes modelos também em tempo de execução. Cabe ressaltar que CVL tem sido considerada pela OMG para se tornar uma linguagem padrão de especificação de variabilidades de software.

7.2.3 Contribuições Não Relacionadas

As contribuições não relacionadas referem-se aos trabalhos relativos à área de pesquisa da tese, mas que estão fora do seu foco principal. Essas contribuições não foram apresentadas no contexto da tese e são descritas brevemente a seguir. As contribuição não relacionadas consistem normalmente de trabalhos envolvendo outros membros do grupo de pesquisa em engenharia de software do IC/UNICAMP.

- *Variabilidade em tratamento de exceções em linha de produtos de software.* Nesta tese, focamos o uso de técnicas de tolerância a falhas de software baseadas em diversidade de software. A fim de aumentar a confiabilidade geral de um sistema de software, é possível utilizar, de forma complementar a estas técnicas, o tratamento de exceções. Neste sentido, assim como exploramos variabilidades de software referentes às diferentes técnicas baseadas em diversidade projeto, um outro trabalho do grupo explorou variabilidades no que tange o comportamento excepcional de sistemas de software [175]. Alguns resultados iniciais foram publicados no Workshop on Exception Handling (WEH'12), em conjunto com ICSE.

- *Uma infraestrutura para apoiar o tratamento de exceção coordenado em composições de serviços.* Também com a finalidade de aumentar a confiabilidade de sistemas orientados a serviços, foi proposta uma infraestrutura que gerencia o tratamento de exceções de forma coordenada em composições de serviços mediante a utilização de conceitos de ações atômicas coordenadas [177]. Alguns resultados iniciais foram publicados no 5th Latin American Symposium on Dependable Computing Workshops (LADCW'11), em conjunto com o LADC.

- *Uma família de técnicas de monitoramento de atributos de QoS em Sistemas Orientados a Serviços* Este projeto tem como principal objetivo apoiar uma família de técnicas de monitoramento de QoS em sistemas orientado a serviços. Para tanto, a infraestrutura dirigida por modelos proposta nesta tese está sendo reutilizada para apoiar o desenvolvimento da arquitetura correspondente à família de interesse. Além disso, as diretrizes aqui seguidas para uma revisão sistemática foram reutilizadas a fim de obtermos as principais contribuições e limitações no que se refere a soluções existentes para monitoramento de QoS de serviços.

7.3 Publicacações

Trabalhos publicados no decorrer desta tese, ordenados pela relevância para a tese:

- **Nascimento A.S.**; Rubira, C. M. F.; Burrows, R.; Castor, F. A systematic review of design diversity-based solutions for fault-tolerant SOAs. In: 17th International

Conference on Evaluation and Assessment in Software Engineering, 2013, Porto de Galinhas, PE, Brazil.

• **Nascimento A.S.**; Castor, F.; Rubira, C. M. F; Burrows, R. An experimental setup to assess design diversity of functionally equivalent services. In: 16th International Conference on Evaluation & Assessment in Software Engineering, 2012, Ciudad Real, Spain.

• **Nascimento A.S.**; Rubira, C. M. F.; Castor, F. *ArCMAPE: A Software Product Line Infrastructure to Support Fault-Tolerant Composite Services.* 15th IEEE International Symposium on High Assurance Systems Engineering, 2014, Miami, Florida, USA.

• **Nascimento A.S.**; Castor, F.; Burrows, R.; Rubira, C.M.F. An Empirical Study on Design Diversity of Functionally Equivalent Web Services. In: 7th International Conference on Availability, Reliability and Security, 2012, Prague, Czech Republic.

• **Nascimento A.S.**; Rubira, C. M. F.; Burrows, R.; Castor, F. A Model-Driven Infrastructure for Developing Product Line Architectures Using CVL. In: 7th Brazilian Symposium on Software Components, Architecture, and Reuse., 2013, Brasília, DF, Brazil.

• **Nascimento A.S.**; Rubira, C. M. F.; Castor, F. Using CVL to Support Self-Adaptation of Fault-Tolerant Service Compositions. In: Seventh IEEE International Conference on Self-Adaptive and Self-Organizing Systems, 2013, Philadelphia, USA.

• **Nascimento A.S.**; Rubira, C. M. F.; Lee, J. An SPL approach for adaptive fault tolerance in SOA. In: 1st International Workshop on Services, Clouds and Alternative Design Strategies for Variant-Rich Software Systems co-located with SPLC, 2011, Munich, Germany.

• **Nascimento A.S.**; Castor, F.; Rubira, C. M. F. Identifying Modelling Dimensions of a Self-Adaptive Framework for Fault-Tolerant SOAs - An Experience Report. In: 1st Workshop on Dependability in Adaptive and Self-Managing Systems co-located with LADC, 2013, Rio de Janeiro.

• **Nascimento A.S.**; Castor, F.; Burrows, R.; Rubira, C.M.F. *An Empirical Study on Design Diversity of Functionally Equivalent Web Services.* IC, UNICAMP, Tech. Rep. IC-12-18, 2012.

• **Nascimento A.S.**; Rubira, C. M. F. *Tolerância a Falhas em Linhas de Produto de Software Baseadas em Serviços Web.* V Workshop de Teses, Dissertações e Trabalhos de Iniciação Científica em Andamento IC-UNICAMP, 2009, Campinas, SP, Brazil. (resumo)

- **Nascimento A.S.**; Rubira, C. M. F. *Especificação de uma abordagem sistemática para gerenciar e implementar Linhas de Produtos Dinâmicas e baseadas em serviços Web.* V Workshop de Teses, Dissertações e Trabalhos de Iniciação Científica em Andamento IC-UNICAMP, 2009, Campinas, SP, Brazil. (resumo)
- Iizuka, B.; **Nascimento A.S.**; Tizzei, L.P.; Rubira, C. M. F.. Supporting the evolution of exception handling in component-based product line architecture. In: 5th International Workshop on Exception Handling co-located with ICSE, 2012, Zurich.
- Manzoni, R. J.; **Nascimento A.S.**; Rubira, C. M. F. WSCA-DRIP: An Infrastructure to Web Service Composition Actions. In: 5th Latin American Symposium on Dependable Computing Workshops co-located LADC, 2011, São José dos Campos, SP, Brazil.

7.4 Trabalhos Futuros

A seguir são listados alguns tópicos que podem dar continuidade a esta pesquisa:

- *Um estudo para avaliar a eficiência de serviços REST funcionalmente equivalentes para tolerar falhas de software.* Este estudo teria como objetivo analisar se nossas principais conclusões acerca das implicações de se utilizar serviços web alternativos para tolerar falhas de software também se aplicam para serviços REST. A infraestrutura proposta nesta tese (Capítulo 3) poderia ser reutilizada nesta direção, modificando, no entanto, os objetos de estudo.
- *Extensão do ArCMAPE para que este possa apoiar uma família de estratégias de monitoramento de atributos de qualidade de serviços (QoS).* Conforme mencionado, no momento, o componente de monitoramento do ArCMAPE é *mocked* (Seção 6.3.2). No entanto, está sendo desenvolvido pelo grupo de pesquisa em engenharia de software do IC/UNICAMP uma infraestrutura para apoiar uma família de estratégias de monitoramento de QoS. Tal infraestrutura pode ser incorporada ao ArCMAPE.
- *Estudos empíricos para avaliar a usabilidade das infraestruturas propostas.* Foram propostras três principais infraestruturas: para avaliar diversidade de serviços, para projetar e implementar arquiteturas de linhas de produtos de software, e para apoiar uma família de técnicas de tolerância a falhas, de forma que a técnica mais apropriada seja dinamicamente instanciada conforme percepção do contexto. Poderia ser realizado um estudo de caso no qual, a partir de diretrizes gerais de uso, um grupo de usuários teria como objetivo projetar e implementar uma família de técnicas de tolerância a falhas utilizando as três infraestruturas propostas. Possibilitando, assim, o estudo mais efetivo da usabilidade de tais infraestruturas [75]. A usabilidade é a medida pela qual

um produto pode ser usado por usuários específicos para alcançar objetivos específicos com efetividade, eficiência e satisfação em um contexto de uso específico [75, 15].

• *Uma solução de tolerância a falhas que contemple tanto técnicas de diversidade de projetos, quanto tratamento de exceção.* A nossa solução poderia ser estendida a fim de apoiar variabilidades de software relativas a diferentes estratégias de tratamento de exceção [175, 177]. Ao combinar técnicas baseadas em diversidade de projeto e tratamento de exceção seria possível apoiar maiores índices de confiabilidades em sistemas orientados a serviços e maior flexibilidade do mecanismo de tolerância a falhas como um todo. Cabe ressaltar que até o momento, conforme descrito no Capítulo 6, as exceções são simplesmente propagadas para os clientes. Um ponto de partida seria inserir tais exceções no modelo contextual (*Figura 6.3*) e prover tratadores alternativos para as mesmas.

• *Estudos empíricos para analisar requisitos de qualidade (e.g. confiabilidade, tempo de resposta, consumo de memória e disponibilidade) ao empregar diferentes técnicas de tolerância a falhas de software baseadas em diversidade de projetos que executam serviços alternativos.* Nesta tese, assumimos que diferentes técnicas de tolerância a falhas apresentam diferentes requisitos de qualidade, valendo-nos de estudos já existentes [1, 49, 50, 51, 52, 133]. Entretanto, seria bastante útil um estudo atual que forneça um mapeamento adequado das técnicas mais apropriadas para diferentes prioridades de requisitos de qualidade no contexto de sistemas orientados a serviços. Tal estudo, facilitaria também a criação de modelos de falhas apropriados.

• *Uma evolução da nossa solução a fim de apoiar a modularização de interesses transversais.* Para o projeto e implementação da arquitetura da família de técnicas baseadas em diversidade de projetos para apoiar serviços compostos confiáveis, as variabilidades de software são resolvidas a partir de componentes que podem ser plugados em pontos pré-estabelecidos. Para tanto, não levamos em consideração os interesses transversais. Tizzei et al. [178] propõem uma visão de característica orientada a aspectos que tem um papel complementar ao modelo de características, ao permitir analisar como as características transversais afetam outras características. Esta visão de características orientada a aspectos é então utilizada para guiar o projeto de uma arquitetura de LPS baseada em componentes e aspectos. Dias et al. [71] propõem uma extensão do modelo COSMOS* (Seção 2.1.2), chamado COSMOS*-VP, que visa modularizar variabilidade arquitetural e características transversais. Este modelo usa técnicas de aspectos para apoiar a modularização e técnicas de componentes para minimizar o acoplamento entre os módulos. Desta forma, ao refatorar nossa solução a partir desta abordagem orientada a aspectos, seria possível obter uma arquitetura que seja mais facilmente evoluída. Estudos empíricos deveriam ser realizados para analisar esta hipótese.

- *Um ambiente integrado para desenvolvimento de arquitetura de linhas de produtos dirigido por modelos.* A infraestrutura dirigida por modelos que propusemos para desenvolver arquiteturas de linhas de produto de software engloba diferentes técnicas, ferramentas e métodos (Capítulo 5). Tal infraestrutura poderia ser bastante beneficiada se existisse um ambiente integrado para apoiar todas as atividades incluídas no processo de desenvolvimento de arquiteturas de linhas de produtos. Este ambiente facilitaria a especificação, validação e transformação de modelos, que também seriam mais consistentes. Ainda, diminuiria a curva de aprendizagem de utilização da infraestrutura, uma vez que não seria necessário aprender diversas técnicas e métodos independentes.

- *Uma solução para o desenvolvimento de linhas de produtos de software dinâmicas.* Embora uma das contribuições desta tese seja uma solução para projetar e implementar linhas de produtos de software dinâmicas, esta solução foi descrita e validada de forma marginal, uma vez que o foco principal da tese é uma solução para tolerância a falhas em sistemas orientados a serviços. Neste sentido, valendo-se dos resultados e soluções iniciais apresentados nesta tese, seria viável especificar uma solução abrangente para apoiar linhas de produtos de software dinâmicas.

Referências Bibliográficas

[1] L. L. Pullum, *Software fault tolerance techniques and implementation.* Norwood, MA, USA: Artech House, Inc., 2001.

[2] P. Sochos, M. Riebisch, and I. Philippow, "The feature-architecture mapping (farm) method for feature-oriented development of software product lines," in *Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems (ECBS'06)*, March 2006, pp. 308 – 318.

[3] O. M. Group, "Welcome to the common variability language," Last access: September, 2013. [Online]. Available: ⟨http://www.omgwiki.org/variability/doku.php?id=start&rev=1351084099⟩

[4] S. Dobson, S. Denazis, A. Fernández, D. Gaiti, E. Gelenbe, F. Massacci, P. Nixon, F. Saffre, N. Schmidt, and F. Zambonelli, "A survey of autonomic communications," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 1, no. 2, pp. 223 – 259, 2006.

[5] S. Ferber, J. Haag, and J. Savolainen, "Feature interaction and dependencies: Modeling features for reengineering a legacy product line," in *Proceedings of the 2nd International Conference on Software Product Lines (SPLC'02)*, August 2002, pp. 235 – 256.

[6] P. Naur and B. Randell, *Proceedings of the NATO software engineering conference.* Garmisch, Germany: Van Nostrand Reinhold, 1968.

[7] M. D. Mcilroy, "Mass produced software components," Tech. Rep., October 1968.

[8] M. Anastasopoulos and C. Gacek, "Implementing product line variabilities," in *Proceedings of Symposium on Software Reusability (SSR'01)*, May 2001, pp. 109 – 117.

[9] F. Bachmann, L. Bass, C. Buhman, S. C. Dorda, F. Long, J. Robert, R. Seacord, and K. Wallnau, "Technical concepts of component-based software engineering,"

Software Engineering Institute, Carnegie Mellon University, Tech. Rep. Technical Report CMU/SEI-2000-TR-008, May 2000.

[10] C. Szyperski, *Component software: Beyond object-oriented programming*, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[11] J. Hopkins, "Component primer," *Communications of the ACM*, vol. 43, no. 10, pp. 27 – 30, 2000.

[12] L. Bass, P. Clements, and R. Kazman, *Software architecture in practice*, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[13] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *ACM SIGSOFT Software Engineering Notes*, vol. 17, no. 4, pp. 40 – 52, 1992.

[14] P. A. Guerra, "Uma abordagem arquitetural para tolerância a falhas em sistemas de software baseados em componentes," PhD thesis, Institute of Computing, University of Campinas, 2004.

[15] A. Geraci, *IEEE standard computer dictionary: Compilation of IEEE standard computer glossaries.* Piscataway, NJ, USA: IEEE Press, 1991.

[16] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann, "Service-oriented computing: State of the art and research challenges," *Computer*, vol. 40, no. 11, pp. 38 – 45, 2007.

[17] M. P. Papazoglou, "Service-oriented computing: Concepts, characteristics and directions," in *Proceedings of the 4th International Conference on Web Information Systems Engineering (WISE '03)*, December 2003, pp. 3 – 12.

[18] L. Baresi, S. Guinea, and L. Pasquale, "Service-oriented dynamic software product lines," *Computer*, vol. 45, no. 10, pp. 42 – 48, 2012.

[19] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimhigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf, "An architecture-based approach to self-adaptive software," *IEEE Intelligent Systems*, vol. 14, no. 3, pp. 54 – 62, 1999.

[20] D. Galan, A. Wolf, and J. Kramer, *Proceedings of the 1st ACM workshop on self-healing systems.* Charleston, SC, USA: ACM Press/Addison-Wesley Publishing Co, 2002.

[21] B. H. Cheng, R. Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, G. Marzo Serugendo, S. Dustdar, A. Finkelstein, C. Gacek, K. Geihs, V. Grassi, G. Karsai, H. M. Kienle, J. Kramer, M. Litoiu, S. Malek, R. Mirandola, H. A. Muller, S. Park, M. Shaw, M. Tichy, M. Tivoli, D. Weyns, and J. Whittle, "Software engineering for self-adaptive systems: A research roadmap," in *Software Engineering for Self-Adaptive Systems*, B. H. Cheng, R. Lemos, H. Giese, P. Inverardi, and J. Magee, Eds. Springer Berlin Heidelberg, 2009, pp. 1 – 26.

[22] D. L. Parnas, "On the design and development of program families," *IEEE Transactions on Software Engineering*, vol. 2, no. 1, pp. 1 – 9, 1976.

[23] J. Van Gurp, J. Bosch, and M. Svahnbergm, "On the notion of variability in software product lines," in *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA '01)*, August 2001, pp. 45 – 54.

[24] K. C. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson, "Feature-oriented domain analysis (foda) feasibility study," Software Engineering Institute, Carnegie Mellon University, Tech. Rep. Technical Report CMU/SEI-90-TR-21, November 1990.

[25] K. Czarnecki and U. W. Eisenecker, *Generative programming: Methods, tools, and applications.* New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000.

[26] P. C. Clements and L. M. Northrop, *Software product lines: Practices and patterns.* Boston, MA, USA: Addison Wesley Professional, 2002.

[27] K. Pohl, G. Böckle, and F. J. van der Linden, *Software product line engineering: Foundations, principles and techniques.* Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.

[28] L. M. Northrop, "Sei's software product line tenets," *IEEE Software*, vol. 19, no. 4, pp. 32 – 40, 2002.

[29] S. Thiel and A. Hein, "Systematic integration of variability into product line architecture design," in *Proceedings of the 2nd International Conference on Software Product Lines (SPLC'02)*, August 2002, pp. 130 – 153.

[30] S. Hallsteinsen, M. Hinchey, P. Sooyong, and K. Schmid, "Dynamic software product lines," *Computer*, vol. 41, no. 4, pp. 93 – 95, 2008.

[31] A. S. Nascimento, C. M. F. Rubira, R. Burrows, and F. Castor, "A model-driven infrastructure for developing product line architectures using cvl," in *Proceedings of the 7th International Conferences on Self-Adaptative and Self-Organizing Systems (SBCARS'13)*, September 2013, p. Accepted for publication.

[32] A. S. Nascimento, C. M. F. Rubira, and F. Castor, "Identifying modelling dimensions of a self-adaptive framework for fault-tolerant soas - an experience report," in *Proceedings of 1st Workshop on Dependability in Adaptive and Self-Managing Systems (WDAS-LADC'13)*, April 2013, pp. 23 – 30.

[33] E. M. Gonçalves and C. M. F. Rubira, "Archmeds: An infrastructure for dependable service-oriented architectures," in *Proceedings of the 17th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'10)*, March 2010, pp. 371 – 378.

[34] Z. Zheng and M. R. Lyu, "An adaptive qos-aware fault tolerance strategy for web services," *Empirical Software Engineering*, vol. 15, no. 4, pp. 323 – 345, 2010.

[35] I. J. G. dos Santos and E. R. M. Madeira, "A semantic-enabled middleware for citizen-centric e-government services," *IJIIT*, vol. 6, no. 3, pp. 34–55, 2010.

[36] J. Gotze, J. Muller, and P. Muller, "Iterative service orchestration based on dependability attributes," in *Proceedings of the 34th Euromicro Conference on Software Engineering and Advanced Applications (SEAA'08)*, September 2008, pp. 353–360.

[37] J. Buys, V. De Florio, and C. Blondia, "Towards context-aware adaptive fault tolerance in soa applications," in *Proceedings. of the 5th ACM International Conference on Distributed Event-Based System (DEBS'11)*, July 2011, pp. 63 – 74.

[38] R. Dillen, J. Buys, V. Florio, and C. Blondia, "Wsdm-enabled autonomic augmentation of classical multi-version software fault-tolerance mechanisms," in *Computer Safety, Reliability, and Security*, F. Ortmeier and P. Daniel, Eds. Springer Berlin Heidelberg, 2012, pp. 294 – 306.

[39] A. Gorbenko, A. Romanovsky, V. Kharchenko, and O. Tarasyuk, "Dependability of service-oriented computing: Time-probabilistic failure modelling," in *Software Engineering for Resilient Systems*, P. Avgeriou, Ed. Springer Berlin Heidelberg, 2012, pp. 121 – 133.

[40] A. S. Nascimento, C. M. F. Rubira, and J. Lee, "An spl approach for adaptive fault tolerance in soa," in *Proceedings of the 15th International Software Product Line Conference (SPLC'11)*, August 2011, pp. 1 – 8.

[41] G. Kotonya and S. Hall, "A differentiation-aware fault-tolerant framework for web services," in *Service-Oriented Computing*, P. P. Maglio, M. Weske, J. Yang, and M. Fantinato, Eds. Springer Berlin Heidelberg, 2010, pp. 137 – 151.

[42] M. R. L., *Handbook of software reliability engineering.* Hightstown, NJ, USA: McGraw-Hill, Inc., 1996.

[43] P. A. Lee and T. Anderson, *Fault tolerance: Principles and practice*, 2nd ed. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1990.

[44] F. C. Gärtner, "Fundamentals of fault-tolerant distributed computing in asynchronous environments," *ACM Comput. Surv.*, vol. 31, no. 1, pp. 1–26, March 1999.

[45] P. H. S. Brito, C. M. F. Rubira, and R. Lemos, "Verifying architectural variabilities in software fault tolerance techniques," in *Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture (WICSA/ECSA'09)*, September 2009, pp. 231 – 240.

[46] J. C. Laprie, C. Béounes, and K. Kanoun, "Definition and analysis of hardware and software-fault-tolerant architectures," *Computer*, vol. 23, no. 7, pp. 39 – 51, 1990.

[47] J. J. Horning, H. C. Lauer, P. M. Melliar-Smith, and B. Randell, "A program structure for error detection and recovery," in *Proceedings of an International Symposium on Operating Systems: Theoretical and Practical Aspects*, April 1974, pp. 171 – 187.

[48] W. R. Elmendorf, "Fault-tolerant programming," in *Proceedings of the 2nd IEEE International Symposium on Fault Tolerant Computing (FTCS'2)*, June 1972, pp. 79 – 83.

[49] J. C. Knight and N. G. Leveson, "An experimental evaluation of the assumption of independence in multiversion programming," *IEEE Transactions on Software Engineering*, vol. 12, no. 1, 1986.

[50] M. R. Lyu, J. H. Chen, and A. Avizienis, "Experience in metrics and measurements of n-version programming," *International Journal of Reliability, Quality and Safety Engineering*, vol. 1, no. 1, pp. 41 – 62, 1994.

[51] V. Hilford, M. R. Lyu, B. Cukic, A. Jamoussi, and F. B. Bastani, "Diversity in the software development process," in *Proceedings of the 3rd Workshop on Object-Oriented Real-Time Dependable Systems (WORDS,97)*, February 1997, pp. 129 – 136.

[52] D. E. Eckhardt, A. K. Caglayan, J. C. Knight, L. D. Lee, D. F. McAllister, M. A. Vouk, and J. P. J. Kelly, "An experimental evaluation of software redundancy as a strategy for improving reliability," *IEEE Transactions on Software Engineering*, vol. 17, no. 7, pp. 692 – 702, 1991.

[53] M. A. Vouk, D. F. Mcallister, D. E. Eckhardt, and K. Kim, "An empirical evaluation of consensus voting and consensus recovery block reliability in the presence of failure correlation," *Journal of Computer and Software Engineering*, vol. 1, pp. 364 – 388, 1993.

[54] T. Anderson, P. A. Barrett, D. N. Halliwell, and M. R. Moulding, "Software fault tolerance: An evaluation," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 12, pp. 1502 – 1510, 1985.

[55] T. Mollerand and H. Schuldt, "Osiris next: Flexible semantic failure handling for composite web service execution," in *Proceedings of the 4th IEEE International Conference on Semantic Computing (ICSC'10)*, September 2010, pp. 212 – 217.

[56] Z. Zheng and M. R. Lyu, "Collaborative reliability prediction of service-oriented systems," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE'10)*, May 2010, pp. 35 – 44.

[57] A. S. Nascimento, C. M. F. Rubira, R. Burrows, and F. Castor, "A systematic review of design diversity-based solutions for fault-tolerant soas," in *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering (EASE'13)*, April 2013, pp. 107 – 118.

[58] C. Yuhui and A. Romanovsky, "Improving the dependability of web services integration," *IT Professional*, vol. 10, no. 3, pp. 29 –35, 2008.

[59] G. T. Santos, L. C. Lung, and C. Montez, "Ftweb: A fault tolerant infrastructure for web services," in *Proceedings of the 9th IEEE International EDOC Enterprise Computing Conference (EDOC '05)*, September 2005, pp. 95 – 105.

[60] M. N. Huhns and M. P. Singh, "Service-oriented computing: Key concepts and principles," *IEEE Internet Computing*, vol. 9, no. 1, pp. 75 – 81, 2005.

[61] R. Rouvoy, P. Barone, Y. Ding, F. Eliassen, S. Hallsteinsen, J. Lorenzo, A. Mamelli, and U. Scholz, "Music: Middleware support for self-adaptation in ubiquitous and service-oriented environments," in *Software Engineering for Self-Adaptive Systems*, B. H. Cheng, R. Lemos, H. Giese, P. Inverardi, and J. Magee, Eds. Springer Berlin Heidelberg, 2009, pp. 164 –182.

[62] C. Ayora, V. Torres, V. Pelechano, and G. H. Alférez, "Applying cvl to business process variability management," in *Proceedings of the VARiability for You Workshop: Variability Modeling Made Useful for Everyone (VARY'12)*, September 2012, pp. 26 – 31.

[63] K. H. Kim and T. F. Lawrence, "Adaptive fault tolerance: Issues and approaches," in *Proceedings of the 2nd IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS'90)*, October 1990, pp. 38 – 46.

[64] A. Papageorgiou, T. Krop, S. Ahlfeld, S. Schulte, J. Eckert, and R. Steinmetz, "Enhancing availability through dynamic monitoring and management in a self-adaptive soa platform," *International Journal on Advances in Software*, vol. 3, no. 3-4, pp. 434 – 446, 2011.

[65] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053 – 1058, 1972.

[66] D. Garlan, S. W. Cheng, A. C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-based self-adaptation with reusable infrastructure," *Computer*, vol. 37, no. 10, pp. 46 – 54, 2004.

[67] B. Morin, "Leveraging models from design-time to runtime to support dynamic variability," PhD thesis, School for Mathematics, Computing, Signal, Electronics, Telecommunications, University of Rennes, 2010.

[68] B. Morin, O. Barais, J. Jezequel, F. Fleurey, and A. Solberg, "Models@ run.time to support dynamic adaptation," *Computer*, vol. 42, no. 10, pp. 44 – 51, 2009.

[69] L. P. Tizzei, M. Dias, C. M. F. Rubira, A. Garcia, and J. Lee, "Components meet aspects: Assessing design stability of a software product line," *Information and Software Technology*, vol. 53, no. 2, pp. 121 – 136, 2011.

[70] C. Ayora, V. Torres, V. Pelechano, and G. H. Alférez, "Applying cvl to business process variability management," in *Proceedings of the VARiablity for You Workshop: Variability Modeling Made Useful for Everyone (VARY'12)*, September 2012, pp. 26 – 31.

[71] M. O. Dias, L. Tizzei, C. M. F. Rubira, A. F. Garcia, and J. Lee, "Leveraging aspect-connectors to improve stability of product-line variabilities."

[72] K. C. Kang, M. Kim, J. Lee, and B. Kim, "Feature-oriented re-engineering of legacy systems into product line assets: a case study," in *Proceedings of the 9th International Conference on Software Product Lines (SPLC'05)*, September 2005, pp. 45 – 56.

[73] A. Svendsen, X. Zhang, R. Lind-Tviberg, F. Fleurey, O. Haugen, B. Moller-Pedersen, and G. K. Olsen, "Developing a software product line for train control: a case study of cvl," in *Proceedings of the 14th International Conference on Software Product Lines (SPLC'10)*, September 2010, pp. 106 – 120.

[74] K. B. and S. Charters, "Guidelines for performing systematic literature reviews in software engineering," Department of Computer Science, University of Durham, Tech. Rep. Technical Report EBSE 2007-001, July 2007.

[75] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering: An introduction.* Norwell, MA, USA: Kluwer Academic Publishers, 2000.

[76] B. Randell, "System structure for software fault tolerance," in *Proceedings of the 1st International Conference on Reliable Software*, April 1975, pp. 437 – 449.

[77] L. A. Gayard, C. M. F. Rubira, and P. A. C. Guerra, "Cosmos*: a component system model for software architectures," Institute of Computing, University of Campinas, Tech. Rep. Technical Report IC-08-04, February 2008.

[78] X. Zhang, O. Haugen, and B. Moller-Pedersen, "Model comparison to synthesize a model-driven software product line," in *Proceedings of the 15th International Software Product Line Conference (SPLC'11)*, August 2011, pp. 90 – 99.

[79] M. Azanza, J. De Sosa, S. Trujillo, and O. Diaz, "Towards a process-line for mdple," in *2nd International Workshop on Model-Driven Product Line Engineering (MDPLE'10)*, June 2010, pp. 3 – 12.

[80] G. Chastek, P. Donohoe, J. D. McGregor, and D. Muthig, "Engineering a production method for a software product line," in *Proceedings of the 15th International Software Product Line Conference (SPLC'11)*, August 2011, pp. 277 – 286.

[81] A. P. Magalhaes, J. M. N. David, R. S. P. Maciel, B. C. Silva, and F. A. Silva, "Modden: An integrated approach for model driven development and software product line processes," in *Proceedings of 5th Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS'11)*, September 2011, pp. 21 – 30.

[82] I. Corporation, "An architectural blueprint for autonomic computing," Department of Computing Science, University of Newcastle upon Tyne, Tech. Rep. n/a, October 2004.

[83] M. P. Papazoglou and W. J. Heuvel, "Service oriented architectures: Approaches, technologies and research issues," *The International Journal on Very Large Data Bases*, vol. 16, no. 3, pp. 389 – 415, 2007.

[84] JAX-WS, "Jax-ws reference implementation," Last access: December, 2011. [Online]. Available: ⟨https://jax-ws.dev.java.net⟩

[85] L. Richardson and S. Ruby, *Restful web services*, 1st ed. O'Reilly, 2007.

[86] R. T. Tomita, F. Castor, P. A. d. C. Guerra, and C. M. F. Rubira, "Bellatrix: An environment with arquitectural support for component-based development (in portuguese)," in *Proceedings of the 4th Brazilian Workshop on Component-Based Development (WDBC'04)*, September 2004, p. 01 – 10.

[87] S. E. Institute, "Framework for software product line practive," Last access: September, 2013. [Online]. Available: ⟨http://www.sei.cmu.edu/productlines/framework.html⟩

[88] H. Gomaa, *Designing software product lines with UML: From use cases to pattern-based software architectures.* Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 2004.

[89] I. Jacobson, M. Griss, and P. Jonsson, *Software reuse: architecture, process and organization for business success.* New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1997.

[90] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wüst, and J. Zettel, *Component-based product line engineering with UML.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[91] N. Bencomo, S. Hallsteinsen, and E. Almeida, "A view of the dynamic software product line landscape," *Computer*, vol. 45, no. 10, pp. 36 – 41, 2012.

[92] K. S. Trivedi, M. Grottke, and E. Andrade, "Software fault mitigation and availability assurance techniques," *International Journal of Systems Assurance Engineering and Management*, vol. 1, no. 4, pp. 340 – 350, 2010.

[93] A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11 – 33, 2004.

[94] M. Hiller, "Software fault-tolerance techniques from a real-time systems point of view - an overview," Department of Computer Engineering, Chalmers University of Technology, Tech. Rep. Technical Report 98-16, November 1998.

[95] F. Saglietti, "The impact of voter granularity in fault-tolerant software on system reliability and availability," in *Software Fault Tolerance*, M. Kersken and F. Saglietti, Eds. Springer Berlin Heidelberg, 1992, pp. 199 – 212.

[96] D. E. Eckhardt and L. D. Lee, "A theoretical basis for the analysis of multiversion software subject to coincident errors," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 12, pp. 1511 – 1517, 1985.

[97] B. Littlewood and D. R. Miller, "Conceptual modeling of coincident failures in multiversion software," *IEEE Transactions on Software Engineering*, vol. 15, no. 12, pp. 1596 –1614, 1989.

[98] T. J. Shimeall and N. G. Leveson, "An empirical comparison of software fault tolerance and fault elimination," in *Proceedings of the 2nd Workshop on Software Testing Verification and Analysis (WST'98)*, July 1998, pp. 180 – 187.

[99] P. Kruchten, *The rational unified process: An introduction*, 3rd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[100] T. Wilfredo, "Software fault tolerance: A tutorial," National Aeronautics and Space Administration (NASA), Tech. Rep., 2000.

[101] A. Avizienis and L. Chen, "On the implementation of n-version programming for software fault tolerance during execution," in *Proceedings of the 1st IEEE International Computer Software and Applications Conference (COMPSAC)*, November 1977, pp. 149 – 155.

[102] L. Chen and A. Avizienis, "N-version programming: A fault-tolerance approach to reliability of software operation," in *Digest of papers of the 8th Annual International Conference on Fault-Tolerant Computing*, June 1978, pp. 21 – 23.

[103] L. L. Zeng, B. Benatallah, A. H. H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang, "Qos-aware middleware for web services composition," *IEEE Transactions on Software Engineering*, vol. 30, no. 5, pp. 311 – 327, 2004.

[104] V. Alves, D. Schneider, M. Becker, and N. Bencomo, "Comparitive study of variability management in software product lines and runtime adaptable systems," in *3rd International Workshop on Variability Modelling of Software intensive Systems (VaMoS'09)*.

[105] L. Shen, X. Peng, J. Liu, and W. Zhao, "Towards feature-oriented variability reconfiguration in dynamic software product lines," in *Proceedings of the 12th International Conference on Top Productivity through Software Reuse (ICSR'11)*, June 2011, pp. 52 – 68.

[106] F. Di Giandomenico and L. Strigini, "Adjudicators for diverse-redundant components," in *Proceedings of the 9th Symposium on Reliability in Distributed Software and Database Systems (SRDS'90)*, October 1990, pp. 114 –123.

[107] J. L. Gersting, R. L. Nist, D. B. Roberts, and R. L. van Valkenburg, "A comparison of voting algorithms for n-version programming," in *Proceedings of the 24th Annual Hawaii International Conference on System Sciences (HICSS'91)*, January 1991, pp. 253 – 262.

[108] S. Setting, "An empirical study on design diversity of variant services," Last access: September, 2013. [Online]. Available: ⟨https://sites.google.com/site/variantservices2s11/home⟩

[109] S. Siegel and N. J. Castellan, *Nonparametric statistics for the behavioral sciences*, 2nd ed. New York, NY, USA: McGraw–Hill, Inc., 1988.

[110] I. Gashi, P. Popov, and L. Strigini, "Fault tolerance via diversity for off-the-shelf products: A study with sql database servers," *IEEE Transactions on Dependable and Secure Computing*, vol. 4, no. 4, pp. 280 – 294, 2007.

[111] D. Partridge and W. Krzanowski, "Software diversity: Practical statistics for its measurement and exploitation," *Information and Software Technology*, vol. 39, no. 10, pp. 707 – 717, 1997.

[112] A. S. Nascimento, F. Castor, C. M. F. Rubira, and R. Burrows, "An experimental setup to assess design diversity of functionally equivalent services," in *Proceedings of the 16th International Conference on Evaluation and Assessment in Software Engineering (EASE'12)*, May 2012, pp. 177 – 186.

[113] S. W. Services, "Seekda's web services portal," Last access: December, 2011. [Online]. Available: ⟨http://webservices.seekda.com⟩

[114] ProgrammableWeb, “Programmableweb repository,” Last access: September, 2013. [Online]. Available: ⟨http://www.programmableweb.com/apis/directory⟩

[115] H. P. Luhn, “A statistical approach to mechanized encoding and searching of literary information,” *IBM Journal of Research and Development*, vol. 1, no. 4, pp. 309 – 317, 1957.

[116] U. S. C. Bureau, “Us zip codes,” Last access: December, 2011. [Online]. Available: ⟨http://www.census.gov/tiger/tms/gazetteer/zips.txt⟩

[117] F. A. Bettelheim, W. H. Brown, and M. K. Campbell.

[118] I. O. for Standardization (ISO), “Iso/iec 7812-1:2006: Identification cards,” Last access: December, 2011. [Online]. Available: ⟨http://www.iso.org/iso/isocatalogue/cataloguetc/catalogue/detail.htm?csnumber=39698⟩

[119] G. Developers, “A google geocoding api,” Last access: September, 2013. [Online]. Available: ⟨http://code.google.com/apis/maps/documentation/geocoding⟩

[120] C. Money, “Cnn money,” Last access: September, 2013. [Online]. Available: ⟨http://money.cnn.com/data/currencies⟩

[121] E. Studies, “An experimental setup to assess diversity in soa,” Last access: September, 2013. [Online]. Available: ⟨https://sites.google.com/site/assessingdiversityinsoa⟩

[122] R. Project, “The r project for statistical computing,” Last access: September, 2013. [Online]. Available: ⟨http://www.r-project.org⟩

[123] D. S. Moore, *The basic practice of statistics*, 2nd ed. New York, NY, USA: W. H. Freeman & Co., 1999.

[124] L. Chen, J. May, and G. Hughes, “Assessment of the benefit of redundant systems,” in *Computer Safety, Reliability and Security*, S. Anderson, S. Bologna, and M. Felici, Eds. Springer-Verlag London, 2002, pp. 151 – 162.

[125] A. Gorbenko, V. Kharchenko, and A. Romanovsky, “Using inherent service redundancy and diversity to ensure web services dependability,” in *Methods, Models and Tools for Fault Tolerance*, M. Butle, C. Jones, A. Romanovsky, and E. Troubitsyna, Eds. Springer Berlin Heidelberg, 2009, pp. 324 – 341.

[126] C. Yuhui, “Ws-mediator for improving dependability of service composition,” PhD thesis, Department of Computing Science, University of Newcastle upon Tyne, 2008.

[127] F. Daniels, K. Kim, and M. A. Vouk, "The reliable hybrid pattern: a generalized software fault tolerant design pattern," in *Proceedings of the 4th Conference of Patter Languages of Programming Conference (PloP'97)*, September 1997, pp. 1 – 9.

[128] K. H. Kim, "Distributed execution of recovery blocks: An approach to uniform treatment of hardware and software faults," in *Proceedings of 4th the International Conference on Distributed Computing Systems (ICDSC'84)*, May 1984, pp. 526 – 532.

[129] R. K. Scott, J. W. Gault, and D. F. Mcallister, "Fault-tolerant software reliability modeling," *IEEE Transactions on Software Engineering*, vol. SE-13, no. 5, pp. 582 – 592, 1987.

[130] D. F. McAllister and M. A. Vouk, "Fault-tolerant software reliability engineering."

[131] R. B. Broen, "New voters for redundant systems," *Journal of Dynamic Systems, Measurement, and Control*, vol. 97, no. 1, pp. 41 – 45, 1975.

[132] S. Bruning, S. Weissleder, and M. Malek, "A fault taxonomy for service-oriented architecture," in *Proceedings of the 10th IEEE High Assurance Systems Engineering Symposium*, 2007, pp. 367–368.

[133] A. S. Nascimento, F. Castor, C. M. F. Rubira, and R. Burrows, "An empirical study on design diversity of functionally equivalent web services," in *Proceedings of the 7th International Conference on Availability, Reliability and Security (ARES'12)*, August 2012, pp. 236 – 241.

[134] J. Arlat, K. Kanoun, and J. C. Laprie, "Dependability evaluation of software fault-tolerance," in *Digest of Papers of the 18th International Symposium on Fault-Tolerant Computing (FTCS'18)*, June 1988, pp. 142 – 177.

[135] D. M. Blough and G. F. Sullivan, "A comparison of voting strategies for fault-tolerant distributed systems," in *Proceedings of the 9th Symposium Reliable Distributed Systems (SRDS'09)*, October 1990, pp. 136 – 145.

[136] B. J. Williams and J. C. Carver, "Characterizing software architecture changes: A systematic review," *Information and Software Technology*, vol. 52, no. 1, pp. 31–51, 2010.

[137] E. S. F. Cardozo, J. B. F. Araújo Neto, A. Barza, A. C. C. França, and F. Q. B. da Silva, "Scrum and productivity in software projects: a systematic literature

review," in *Proceedings of the 14th International Conference on Evaluation and Assessment in Software Engineering (EASE'10)*, April 2010, pp. 131 – 134.

[138] M. Jorgensen and M. Shepperd, "A systematic review of software development cost estimation studies," *IEEE Transactions on Software Engineering*, vol. 33, no. 1, pp. 33 – 53, 2007.

[139] B. A. Kitchenham, E. Mendes, and G. H. Travassos, "Cross versus within-company cost estimation studies: A systematic review," *IEEE Transactions on Software Engineering*, vol. 33, no. 5, pp. 316 – 329, 2007.

[140] B. A. Kitchenham, O. Pearl Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman, "Systematic literature reviews in software engineering - a systematic literature review," *Information and Software Technology*, vol. 51, no. 1, pp. 7–15, 2009.

[141] E. Nourani, "A new architecture for dependable web services using n-version programming," in *Proceedings of 3rd International Conference on Computer Research and Development (ICCRD'11)*, March 2011, pp. 333 – 336.

[142] P. Townend, P. Groth, and J. Xu, "A provenance-aware weighted fault tolerance scheme for service-based applications," in *Proceedings of the 8th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*, March 2005, pp. 258 – 266.

[143] H. Abdeldjelil, N. Faci, Z. Maamar, and D. Benslimane, "A diversity-based approach for managing faults in web services," in *Proceedings of the IEEE 26th International Conference on Advanced Information Networking and Applications*, March 2012, pp. 81 – 88.

[144] N. Looker, M. Munro, and J. Xu, "Increasing web service dependability through consensus voting," in *Proceedings of the 29th annual International Conference on Computer Software and Applications (COMPSAC-W'05)*, July 2005, pp. 66 – 69.

[145] A. Gorbenko, V. Kharchenko, P. Popov, and A. Romanovsky, "Dependable composite web services with components upgraded online," in *Architecting Dependable Systems III*, R. Lemos, C. Gacek, and A. Romanovsky, Eds. Springer-Verlag Berlin Heidelberg, 2005, pp. 92 – 121.

[146] H. Mansour and T. Dillon, "Dependability and rollback recovery for composite web services," *IEEE Transactions on Services Computing*, vol. 4, no. 4, pp. 328 – 339, 2011.

[147] N. Laranjeiro and M. Vieira, "Towards fault tolerance in web services compositions," in *Proceedings of the 2nd International Workshop on Engineering Fault Tolerant Systems (EFTS'07)*, September 2007, p. n/a.

[148] Z. Zheng and M. R. Lyu, "Ws-dream: A distributed reliability assessment mechanism for web services," in *Proceedings of the International Conference on Dependable Systems and Networks*, June 2008, pp. 392 – 397.

[149] N. Faci, H. Abdeldjelil, Z. Maamar, and D. Benslimane, "Using diversity to design and deploy fault tolerant web services," in *Proceedings of the 20th IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE'11)*, June 2011, pp. 73 – 78.

[150] X. Jie, "Achieving dependability in service-oriented systems," in *Dependable and Historic Computing*, C. B. Jones and J. L. Lloyd, Eds. Springer Berlin Heidelberg, 2011, pp. 504 – 522.

[151] N. Milanovic and M. Malek, "Service-oriented operating system: A key element in improving service availability," in *Proceedings of the 4th International Symposium on Service Availability (ISAS '07)*, May 2007, pp. 31 – 42.

[152] E. Nourani and M. A. Azgomi, "A design pattern for dependable web services using design diversity techniques and ws-bpel," in *Proceedings of the 6th International Conference on Innovations in Information Technology (IIT'09)*, December 2009, pp. 290 – 294.

[153] R. Burrows, A. Garcia, and F. Taiani, "Coupling metrics for aspect-oriented programming: A systematic review of maintainability studies," in *Evaluation of Novel Approaches to Software Engineering*, L. Maciaszek, C. Gonzalez-Perez, and S. Jablonski, Eds. Springer Berlin Heidelberg, 2010, pp. 277 – 290.

[154] A. F. Garcia, C. M. F. Rubira, A. B. Romanovsky, and J. Xu, "A comparative study of exception handling mechanisms for building dependable object-oriented software," *Journal of Systems and Software*, vol. 59, no. 2, pp. 197 – 222, 2001.

[155] A. Carzaniga, A. Gorla, and M. Pezze, "Handling software faults with redundancy," in *Architecting Dependable Systems VI*, R. Lemos, J. C. Fabre, C. Gacek, F. Gadducci, and M. Beek, Eds. Springer-Verlag, 2009, pp. 148 – 171.

[156] H. H. Ammar, B. Cukic, A. Mili, and C. Fuhrman, "A comparative analysis of hardware and software fault tolerance: Impact on software reliability engineering," *Annals of Software Engineering*, vol. 10, no. 1-4, pp. 103 – 150, 2000.

[157] V. D. Florio and C. Blondia, "A survey of linguistic structures for application-level fault tolerance," *ACM Computing Surveys*, vol. 40, no. 2, pp. 6:1 – 6:37, 2008.

[158] C. Parra, X. Blanc, and L. Duchien, "Context awareness for dynamic service-oriented product lines," in *Proceedings of the 13th International Software Product Line Conference (SPLC'09)*, August 2009, pp. 131 – 140.

[159] K. Czarnecki, M. Antkiewicz, C. H. P. Kim, S. Lau, and K. Pietroszek, "Model-driven software product lines," in *Proceeding of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*, October 2005, pp. 126 – 127.

[160] X. Zhang and B. Moller-Pedersen, "Towards correct product derivation in model-driven product lines," in *Proceedings of the 7th International Conference on System Analysis and Modeling: Theory and Practice (SAM'12)*, October 2012, pp. 179 – 197.

[161] C. W. Krueger, "Easing the transition to software mass customization," in *Software Product-Family Engineering*, F. van der Linden, Ed. Springer-Verlag Berlin Heidelberg, 2002, pp. 282 – 293.

[162] I. R. Group, "Fama framework," Last access: September, 2013. [Online]. Available: ⟨http://www.isa.us.es/fama⟩

[163] E. Rouille, B. Combemale, O. Barais, D. Touzet, and J. M. Jezequel, "Leveraging cvl to manage variability in software process lines," in *Proceedings of the 19th Asia-Pacific Software Engineering Conference (APSEC '12)*, December 2012, pp. 148 – 157.

[164] T. Buchmann, A. Dotor, and B. Westfechtel, "Mod2-scm: A model-driven product line for software configuration management systems," *Information and Software Technology*, vol. 55, no. 3, pp. 630 – 650, 2013.

[165] J. Cheesman and J. Daniels, *UML components: A simple process for specifying component-based software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

[166] J. White, B. Dougherty, H. D. Strowd, and D. C. Schmidt, "Using filtered cartesian flattening and microrebooting to build enterprise applications with self-adaptive healing," in *Software Engineering for Self-Adaptive Systems*, B. H. C. Cheng, R. Lemos, H. Giese, P. Inverardi, and J. Magee, Eds. Springer Berlin Heidelberg, 2009, pp. 241 – 260.

[167] L. Ramnivas, *AspectJ in action: Enterprise AOP with spring applications*, 2nd ed. Greenwich, CT, USA: Manning Publications Co., 2009.

[168] F. Castor, P. A. 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*, vol. 10, no. 3, pp. 5 – 19, 2005.

[169] Y. Tao and L. Kwei-Jay, "Service selection algorithms for composing complex services with multiple qos constraints," in *Proceedings of the 3rd International Conference on Service-Oriented Computing (ICSOC'05)*, December 2005, pp. 130 – 143.

[170] M. Rosenmuller, N. Siegmund, M. Pukall, and S. Apel, "Combining runtime adaptation and static binding in dynamic software product lines," School of Computer Science, University of Magdeburg, Tech. Rep. Technical Report FIN-02-2011, February 2011.

[171] O. Alliance, "Osgi - the dynamic module system for java," Last access: September, 2013. [Online]. Available: ⟨http://www.osgi.org/Main/HomePage⟩

[172] Eclipse, "Eclipse modeling framework technology (emft)," Last access: September, 2013. [Online]. Available: ⟨http://www.eclipse.org/modeling/emft/⟩

[173] JRuleEngine, "Jruleengine – project description," Last access: September, 2013. [Online]. Available: ⟨http://jruleengine.sourceforge.net⟩

[174] G. Brataas, S. Hallsteinsen, R. Rouvoy, and F. Eliassen, "Scalability of decision models for dynamic product lines," in *Proceedings of the 1st International Workshop on Dynamic Software Product Lines (DSPL'07)*, September 2007, pp. 23 – 32.

[175] B. Iizuka, A. Nascimento, L. Tizzei, and C. Rubira, "Supporting the evolution of exception handling in component-based product line architecture," in *Exception Handling (WEH), 2012 5th International Workshop on*, June 2012, pp. 62–64.

[176] K. Geihs, P. Barone, F. Eliassen, J. Floch, R. Fricke, E. Gjorven, S. Hallsteinsen, G. Horn, M. U. Khan, A. Mamelli, G. A. Papadopoulos, N. Paspallis, R. Reichle, and E. Stav, "A comprehensive solution for application-level adaptation," *Software – Practice and Experience*, vol. 39, no. 4, pp. 385 – 422, 2009.

[177] R. de Jesus Manzoni, A. Nascimento, and C. Rubira, "Wsca-drip: An infrastructure to web service composition actions," in *Dependable Computing Workshops (LADCW), 2011 Fifth Latin-American Symposium on*, April 2011, pp. 29–32.

[178] L. P. Tizzei, C. M. F. Rubira, and L. Jaejoon, "An aspect-based feature model for architecting component product lines," in *Proceedings of the 38th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA'12)*, September 2012, pp. 85 – 92.