Universidade Federal da Bahia

Universidade Salvador

Universidade Estadual de Feira de Santana

TESE DE DOUTORADO

Projeto Arquitetural Automatizado de Sistemas Self-Adaptive Uma Abordagem Baseada em Busca

Sandro Santos Andrade

Programa Multiinstitucional de Pós-Graduação em Ciência da Computação – PMCC

Salvador - BA

2014

PMCC-DSc-0016

Programa Multiinstitucional de
Pós-Graduação em Ciência da Computação – PMCC

PMCC-IM-UFBA, Campus de Ondina
Av. Adhemar de Barros S/N, Salvador - Bahia, CEP 40.170-110
dmcc@ufba.br http://dmcc.dcc.ufba.br

SANDRO SANTOS ANDRADE

Projeto Arquitetural Automatizado de Sistemas Self-Adaptive

Uma Abordagem Baseada em Busca

Tese de Doutorado apresentada ao Programa Multi-institucional de Pós-Graduação em Ciência da Computação da Universidade Federal da Bahia, Universidade Salvador e Universidade Estadual de Feira de Santana como requisito parcial para obtenção do grau de Doutor em Ciência da Computação.

Orientador: Prof. Dr. Raimundo José de Araújo Macêdo

SALVADOR
Dezembro/2014

Sistemas de Bibliotecas - UFBA

Andrade, Sandro Santos.
Projeto arquitetural automatizado de sistemas self-adaptive: uma abordagem baseada em busca / Sandro Santos Andrade. - 2015.
278 f.: il.

Inclui apêndices.
Orientador: Prof. Dr. Raimundo José de Araújo Macedo.
Tese (doutorado) - Universidade Federal da Bahia, Instituto de Matemática, Universidade Salvador, Universidade Estadual de Feira de Santana, Salvador, 2014.

1. Software - Desenvolvimento. 2. Otimização combinatória. 3. Algoritmos genéticos. 4. Engenharia de Software. I. Macedo, Raimundo José de Araújo. II. Universidade Federal da Bahia. Instituto de Matemática. III. Universidade Salvador. IV. Universidade Estadual de Feira de Santana. V. Título.

CDD - 005.1
CDU - 004.4

TERMO DE APROVAÇÃO

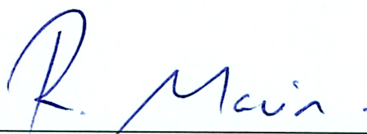
SANDRO SANTOS ANDRADE

PROJETO ARQUITETURAL AUTOMATIZADO DE SISTEMAS SELF-ADAPTIVE

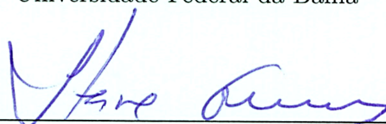
Uma Abordagem Baseada em Busca

Esta tese foi julgada adequada à obtenção do título de Doutor em Ciência da Computação e aprovada em sua forma final pelo Programa Multi-institucional de Pós-Graduação em Ciência da Computação da UFBa/UNIFACS/UEFS.

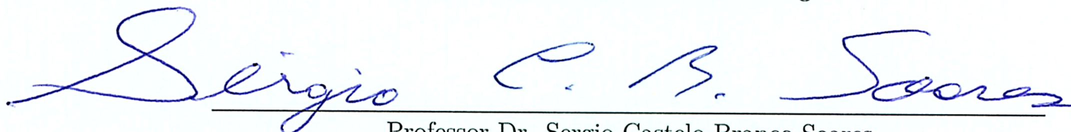
Salvador, 19 de Dezembro de 2014



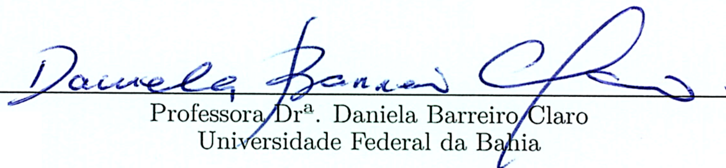
Professor Dr. Raimundo José de Araújo Macêdo (orientador)
Universidade Federal da Bahia



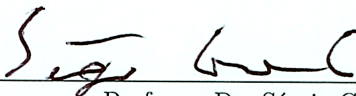
Professora Dr^{a}. Itana Maria de Souza Gimenes
Universidade Estadual de Maringá



Professor Dr. Sergio Castelo Branco Soares
Universidade Federal de Pernambuco



Professora Dr^{a}. Daniela Barreiro Claro
Universidade Federal da Bahia



Professor Dr. Sérgio Gorender
Universidade Federal da Bahia

A meu pai, Geraldo

Agradecimentos

A conclusão de um trabalho de doutorado certamente não seria possível sem pessoas queridas e singulares, que acabam vivendo um pouco das dores e dos prazeres presentes nesta caminhada. Um agradecimento especial aos meus pais – Dilce e Geraldo – e aos meus irmãos – Marcelo, Eudes e Márcio – pelo suporte, carinho e compreensão ao longo de toda uma vida. Ao meu filho, Benjamin, alegria constante da minha vida e razão da busca incessante pela iluminação. Um agradecimento especial à minha esposa, Gabi, por todo o amor e compreensão que a fizeram vivenciar de perto minhas conquistas e incertezas.

Este trabalho não seria realizado sem o apoio, amizade e dedicação do meu orientador – Raimundo Macêdo – sempre disponível e generoso nessa minha jornada pelo mundo acadêmico. Gostaria de agradecer também a todos os professores do PMCC, em especial a José Suruagy e Christina Chaves pelo apoio e ajuda fornecidos em alguns momentos mais difíceis. Aos funcionários do CEAPG, sempre dispostos a ajudar no que fosse necessário. Aos colegas de doutorado Alírio Sá, Paul Regnier, Antonio Maurício Pitangueira e Roberto Figueiredo, pelas conversas reconfortantes.

Um agradecimento especial aos meus colegas do GSORT/IFBa: Manoel Neto, Renato Novais, Flávia Nascimento, Romildo Martins, Allan Freitas, Pablo Florentino, Frederico Barbosa e Grinaldo de Oliveira. A José Mário Araújo por toda a consultoria em Teoria de Controle gentilmente prestada. Gostaria de agradecer a inúmeros membros da comunidade Qt e KDE, em especial a Tomaz Canabrava, pela ajuda dispensada durante as atividades de desenvolvimento realizadas neste trabalho.

Finalmente, mas não menos importante, um agradecimento especial aos amigos da música: André, Alex, Neri, Érica, Tati, Mana, Cassio, Robertinha, Eli, Juvino e Ana Flávia. Foram longos anos de ausências mas espero estarmos em breve de volta aos nossos desvarios sonoros.

Senta-te sem ansiedades.
Acalma-te, ama, perdoa, renuncia,
medita e guarda silêncio.
Aguarda.
Os frutos vão amadurecer.

(Hermógenes)

Resumo

A construção de objetos artificiais, intencionalmente projetados para exibir propriedades desejadas, é o objetivo principal das atividades de engenharia. O projeto de sistemas de *software* – enquanto representantes legítimos de objetos artificiais – ao mesmo tempo em que impõe desafios particulares à representação sistemática de conhecimento de projeto, se beneficia direta e substancialmente dos produtos de pesquisas em áreas tais como *Design Science*. Ao longo dos últimos anos, uma série de novas demandas contribuiram para aumentar ainda mais a complexidade essencial de sistemas de *software*. Como consequência, desenvolver sistemas computacionais que atendam satisfatoriamente a essas demandas requer um controle intelectual que rapidamente se aproxima dos limites humanos. Os Sistemas Self-Adaptive (SSA) representam uma solução promissora para este problema. Um SSA é caracterizado pela transferência – para *run-time* – de uma ou mais atividades do processo de desenvolvimento de *software*, suportadas por uma infraestrutura de adaptação que permite o raciocínio e execução automáticos de atividades antes realizadas *off-line* pelos desenvolvedores. Nos últimos anos, diversas técnicas refinadas para implementação desta capacidade de autogerenciamento em sistemas computacionais foram propostas. Consequentemente, conhecer as abordagens disponíveis, seus fundamentos arquiteturais, implicações e *trade-offs* envolvidos não é uma tarefa trivial. Esta tese apresenta o projeto, implementação e avaliação de uma abordagem para projeto arquitetural automatizado de SSA, baseada na representação sistemática de conhecimento refinado de projeto e na aplicação de técnicas de otimização multiobjetivo. A meta foi viabilizar uma exploração mais efetiva do espaço de projeto envolvido na construção de SSA e suportar, de forma mais rigorosa, as decisões referentes aos *trade-offs* presentes. Para isso, uma nova linguagem – denominada DuSE – para especificação de espaços de projeto e de métricas de qualidade particulares a um determinado domínio de aplicação foi proposta, associada a uma plataforma de otimização de arquiteturas de *software*. A linguagem DuSE viabilizou a infraestrutura de metamodelagem necessária para a especificação do SA:DuSE – espaço de projeto concreto responsável pela captura sistemática das principais dimensões de projeto relacionadas à implementação de SSA baseados em *feedback control loops*. A abordagem foi avaliada em relação à sua capacidade de manifestação de *trade-offs* (através de indicadores de desempenho de modelagem em algoritmos de otimização), potencial de antecipação dos atributos de qualidade evidenciados (através de comparação com protótipos funcionais de SSA) e impacto na efetividade e complexidade dos projetos quando comparada a abordagens alternativas (através de experimento controlado). Resultados indicam que a abordagem efetivamente captura o caráter multiobjetivo do projeto de SSA e que promove um melhor suporte à investigação de alternativas e tomada de decisão no projeto de SSA.

Palavras-chave: sistemas *self-adaptive*, arquitetura de *software*, otimização multiobjetivo, engenharia de *software* baseada em busca, modelagem de *software*.

Abstract

The conceiving of artificial objects, intentionally designed to exhibit desired properties, is the main goal of engineering activities. The design of software systems – as legitimate representatives of artificial objects – not only brings new challenges to the systematic representation of design knowledge but also directly benefits from research discoveries in fields such as Design Science. Over the past few years, a number of new demands have contributed to further increase the essential complexity of software systems. As a consequence, the intellectual control involved in the development of systems that fulfill such demands appears to be approaching the limits of human capability. Self-Adaptive Systems (SAS) have currently be advocated as a promising solution for such a problem. A SAS is characterized by the adoption of a run-time infrastructure for reasoning and adaptation, which automates one or more software development process acitivities previously undertaken offline by the developers. In recent years, several mechanisms for endowing software systems with such self-management capabilities have been proposed. Consequently, being familiar with the available approaches, their architectural foundations, implications and involved trade-offs is a challenging task. This work presents the design, implementation and evaluation of an automated approach for designing SAS architectures, which relies on the use of systematic knowledge representation and multi-objective optimization techniques. The goal was to enable a more effective exploration of the design space involved in developing SAS and provide accurate support for decision making with respect to architectural trade-offs. For that purpose, we created a new modeling language – named DuSE – for the definition of domain-specific design spaces and architecture evaluation metrics. Furthermore, we devised a multi-objective optimization infrastructure which automatically searches for effective solutions in a given design space. The DuSE language provided the underpinnings which enabled the specification of SA:DuSE – a concrete design space that systematically captures the most prominent design dimensions involved in the development of SAS based on feedback control loops. Our approach was evaluated regarding its ability to reveal design trade-offs (by using particular optimization performance indicators), its potential for anticipating the exhibited quality attributes (by comparison with functional prototypes of SAS) and its impact on the effectiveness and design complexity when compared to alternative approaches (by a controlled experiment). Results indicate that our approach effectively captures the multi-objective nature of SAS design and improves the support for comparing design alternatives and decision making in such an application domain.

Keywords: self-adaptive systems, software architecture, multi-objective optimization, search-based software engineering, software modeling.

Sumário

Lista de Figuras v

Lista de Tabelas vii

Lista de Algoritmos ix

Lista de Siglas xi

1. Introdução 1
- 1.1. Motivação 1
- 1.2. Questões de Pesquisa 2
- 1.3. Contribuições da Tese 3
- 1.4. Visão Geral da Tese 3
  - 1.4.1. Como Ler esta Tese 5
- 1.5. Publicações 6
- 1.6. Convenções Tipográficas 8

I. Fundamentos 9

2. Projeto e Análise de Arquiteturas de Software 11
- 2.1. Arquiteturas de Software 11
- 2.2. Atributos de Qualidade 15
- 2.3. Processos de Projeto Arquitetural 17
  - 2.3.1. Gestão de Conhecimento de Projeto Arquitetural 19
- 2.4. Modelagem e Metamodelagem 20

3. Sistemas Self-Adaptive 27
- 3.1. Motivação para Sistemas Self-Adaptive 28
- 3.2. Fundamentos 30
  - 3.2.1. Espaço de Problema 32
  - 3.2.2. Espaço de Solução 33
- 3.3. Mecanismos para Implementação de Comportamento Self-Adaptive 40
  - 3.3.1. Abordagens Baseadas em Agentes Inteligentes 40
  - 3.3.2. Abordagens Baseadas em Busca 42
  - 3.3.3. Abordagens Baseadas em Formalização de Estrutura 43
  - 3.3.4. Abordagens Baseadas em Teoria de Controle 44
    - 3.3.4.1. Fundamentos 45
    - 3.3.4.2. Modelagem do Sistema 50

3.3.4.3. Estratégias para Feedback Control . . . 54
3.3.4.4. Técnicas para Projeto de Controladores . . . 59
3.4. Discussão Final . . . 63

4. Otimização Multiobjetivo 65
4.1. Otimização . . . 65
4.2. Otimização Multiobjetivo . . . 67
4.2.1. Dominância e Pareto-Optimality . . . 70
4.3. Principais Abordagens para Otimização Multiobjetivo . . . 73
4.3.1. Abordagens Clássicas . . . 74
4.3.2. Abordagens Evolucionárias . . . 75
4.3.2.1. Algoritmos Genéticos . . . 76
4.3.2.2. O NSGA-II . . . 80
4.4. Métricas de Desempenho . . . 83
4.4.1. Métricas para Avaliação de Convergência . . . 84
4.4.2. Métricas para Avaliação de Diversidade . . . 85
4.4.3. Métricas para Avaliação Conjunta de Convergência e Diversidade . . . 86

II. Projeto Arquitetural Automatizado de Sistemas Self-Adaptive 89

5. Projeto Arquitetural Automatizado de Sistemas Self-Adaptive 91
5.1. Motivação . . . 92
5.1.1. Levantamento de Requisitos . . . 92
5.1.2. Tomada de Decisão . . . 93
5.1.3. Análise Arquitetural . . . 94
5.2. Objetivos . . . 95
5.3. Aplicações Exemplo . . . 96
5.3.1. Servidor Web SISO . . . 97
5.3.2. Servidor Web MIMO . . . 99
5.3.3. Cluster Elástico para Aplicações MapReduce . . . 101
5.4. Processo Automatizado de Projeto Arquitetural de Sistemas Gerenciadores . . . 104
5.5. Premissas e Limitações . . . 106
5.6. Resumo do Capítulo . . . 107

6. Infraestrutura Independente de Domínio para Espaços de Projeto 109
6.1. Requisitos para Projeto Arquitetural Automatizado . . . 110
6.2. Formalização do Espaço de Projeto . . . 111
6.3. Algoritmos para Projeto Arquitetural Automatizado . . . 122
6.3.1. Algoritmo de Derivação do Espaço de Projeto Específico de Aplicação . . 122
6.3.2. Algoritmo de Projeto Automatizado de Arquiteturas Candidatas . . . 122
6.4. Premissas e Limitações . . . 124
6.5. Resumo do Capítulo . . . 127

7. Espaço de Projeto Arquitetural para Sistemas Self-Adaptive 131
7.1. Esquema para Descrição das Dimensões de Projeto . . . 132
7.2. Dimensões de Projeto do SA:DuSE . . . 133
7.2.1. DD1) Modelagem da Dinâmica do Sistema Gerenciado . . . 133
7.2.2. DD2) Cardinalidade de Controle . . . 135
7.2.3. DD3) Lei de Controle . . . 140
7.2.4. DD4) Técnica de Sintonia . . . 144
7.2.5. DD5) Grau de Adaptabilidade do Controlador . . . 147

7.2.6. DD6) Forma de Cooperação entre Múltiplos Loops 153
7.3. Métricas de Qualidade do SA:DuSE 161
7.3.1. Tempo Médio de Estabilização 161
7.3.2. Sobressinal Médio 162
7.3.3. Efetividade de Controle 163
7.3.4. Overhead de Controle 164
7.4. Resumo do Capítulo 164

8. Otimização no Projeto Arquitetural de Sistemas Self-Adaptive 167
8.1. Definição do Problema de Otimização 167
8.2. Otimização Multiobjetivo de Arquiteturas 169
8.2.1. Representação das Soluções 170
8.2.2. Operadores de Seleção, Recombinação e Mutação 170
8.3. Premissas e Limitações 172
8.4. Resumo do Capítulo 173

9. Suporte por Ferramenta 175
9.1. Requisitos e Diretrizes Arquiteturais 175
9.2. Projeto e Implementação 177
9.2.1. O Qt5Modeling e o Qt5ModelingWidgets 179
9.2.2. O Qt5Duse e o Qt5SADuseProfile 183
9.2.3. O Qt5Optimization 184
9.2.4. A Ferramenta DuSE-MT 185
9.3. Resumo do Capítulo 187

III. Avaliação e Conclusão 189

10. Avaliação 191
10.1. Metas de Avaliação 192
10.2. Estudos de Avaliação 194
10.2.1. Estudo 1: Efetividade da Captura de Trade-offs no SA:DuSE 194
10.2.1.1. Servidor Web MIMO 195
10.2.1.2. Cluster Elástico para Aplicações MapReduce 200
10.2.1.3. Discussão 203
10.2.2. Estudo 2: Acurácia das Métricas de Qualidade do SA:DuSE 204
10.2.2.1. Servidor Web MIMO 204
10.2.2.2. Cluster Elástico para Aplicações MapReduce 210
10.2.2.3. Discussão 217
10.2.3. Estudo 3: Benefícios do Uso de Abordagens Baseadas em Busca no Projeto de Sistemas Self-Adaptive 218
10.2.3.1. Definição do Experimento 218
10.2.3.2. Planejamento do Experimento 219
10.2.3.3. Análise dos Resultados e Discussão 224
10.2.3.4. Ameaças à Validade 227
10.3. Resumo do Capítulo 228

11. Trabalhos Correlatos 231
11.1. Dimensões de Correlação 231
11.1.1. Engenharia de Software para Sistemas Self-Adaptive 232
11.1.2. Projeto Automatizado de Arquiteturas de Software 236
11.1.3. Processos de Projeto Arquitetural Baseados em Atributos de Qualidade . 238

11.2. Discussão Final 240

12. Conclusão **243**
12.1. Resumo da Tese 243
12.2. Contribuições 245
12.3. Limitações 246
12.4. Trabalhos Futuros 247

Referências Bibliográficas **249**

A. Especificação do SADuSE UML Profile **269**
A.1. Visão Geral 269
A.2. Descrição dos Elementos do Profile 269
A.2.1. Componentes e Interfaces de Processo 269
A.2.1.1. ProcessComponent 269
A.2.1.2. SISOProcessComponent 270
A.2.1.3. TFProcessComponent 271
A.2.1.4. FOPDTProcessComponent 272
A.2.1.5. MIMOProcessComponent 272
A.2.1.6. SSProcessComponent 274
A.2.1.7. ProcessInterface 274
A.2.1.8. MonitorableInterface 275
A.2.1.9. ControlableInterface 275

Índice Remissivo **277**

Lista de Figuras

1.1. Áreas de conhecimento envolvidas nesta tese . . . 3

2.1. Roteiro do capítulo 2 . . . 12
2.2. Atividades gerais de um processo de projeto arquitetural . . . 18
2.3. Acrônimos para diferentes abordagens de uso de modelos de *software* . . . 20
2.4. Visão geral da MDE . . . 21
2.5. Hierarquia de metamodelagem definida na MDA . . . 23
2.6. Modelo conceitual das definições apresentadas no capítulo 2 . . . 24

3.1. Roteiro do capítulo 3 . . . 28
3.2. Dimensões do espaço de problema de sistemas *self-adaptive* . . . 32
3.3. Estrutura básica de um *loop* externo de adaptação . . . 34
3.4. Dimensões do espaço de solução de sistemas *self-adaptive* (1º nível) . . . 35
3.5. Dimensões do espaço de solução de sistemas *self-adaptive* (*loops* de adaptação) 35
3.6. Dimensões do espaço de solução de sistemas *self-adaptive* (monitoramento) . . 37
3.7. Dimensões do espaço de solução de sistemas *self-adaptive* (análise) . . . 37
3.8. Dimensões do espaço de solução de sistemas *self-adaptive* (planejamento) . . . 38
3.9. Dimensões do espaço de solução de sistemas *self-adaptive* (atuação) . . . 40
3.10. Sistema dinâmico e estruturas de controle em malha aberta e malha fechada . 46
3.11. Propriedades de sistemas de controle e ações realizadas por diferentes controladores . . . 49
3.12. Parâmetros coletados via ensaio ao degrau . . . 54
3.13. Diferentes graus de robustez apresentado por controladores . . . 55
3.14. Esquemas MIAC e MRAC para controle adaptativo . . . 58
3.15. Lugar das raízes em função do parâmetro K_P . . . 60

4.1. Roteiro do capítulo 4 . . . 66
4.2. Espaço de decisão viável, espaço-objetivo viável e exemplos de *Pareto-front* e de solução dominada . . . 68
4.3. Soluções com diferentes graus de convergência e diversidade . . . 71
4.4. Exemplo de problema de otimização multiobjetivo com objetivos não-conflitantes . . . 71
4.5. Atividades gerais de um algoritmo genético . . . 76
4.6. Exemplos dos níveis de não-dominância e cuboides da distância de aglomeração utilizados no NSGA-II . . . 80
4.7. Hipervolume gerado pelas soluções do *Pareto-front* e o ponto de referência $\mathcal{W}$ 86

5.1. Roteiro do capítulo 5 . . . 92

5.2. Exemplo de modelo para servidor *web* SISO, apresentando suas interfaces monitorável e controlável 97
5.3. Duas alternativas de controle do servidor *web* SISO: controle P e controle PI. Interface para ajuste do valor de referência utilizado pelo controlador 98
5.4. Exemplo de modelo para servidor *web* MIMO com suas interfaces monitoráveis e controláveis 99
5.5. Duas alternativas de controle do servidor *web* MIMO: controle MIMO *dynamic state* e múltiplos controles SISO via PI. Interfaces para ajuste dos valores de referência utilizados pelos controladores 100
5.6. Exemplo de modelo para um *cluster* de computadores utilizado como infraestrutura para execução de aplicações *MapReduce* e suas interfaces monitoráveis e controláveis 102
5.7. Controle do *cluster* elástico através da utilização de controladores PI aninhados 103
5.8. Processo automatizado de projeto arquitetural proposto nesta tese 104

6.1. Roteiro do capítulo 6 109
6.2. Representação gráfica (sintaxe concreta) do modelo arquitetural utilizado para exemplificar a formalização do espaço de projeto 113
6.3. Representação gráfica (sintaxe concreta) do modelo arquitetural resultante da aplicação do ponto de variação apresentado nas Tabelas 6.3 e 6.4 117
6.4. Modelo conceitual das definições apresentadas no capítulo 6 129

7.1. Roteiro do capítulo 7 132
7.2. Exemplo de modelagem de sistemas via função de transferência, FOPDT e espaço de estados 135
7.3. Exemplos de arquiteturas candidatas geradas para o ponto de variação VP21 e VP22 140
7.4. Arquiteturas resultantes da combinação dos pontos de variação VP33 e VP46 e dos pontos de variação VP35 e VP42 150
7.5. Arquiteturas resultantes da combinação dos pontos de variação VP33, VP46 e VP52 e dos pontos de variação VP35, VP42 e VP53 153
7.6. Arquiteturas resultantes da aplicação dos pontos de variação VP62 e VP64 . . 160
7.7. Influência de α_r^E na efetividade de controle gerada pelos pontos de variação da dimensão DD5 164

8.1. Roteiro do capítulo 8 167

9.1. Roteiro do capítulo 9 175
9.2. Visão originada do estilo de uso de módulos (*module uses style*) para os artefatos utilizados neste trabalho 178
9.3. Visão originada do estilo de decomposição (*module decomposition style*) para o módulo `Qt5Modeling` 179
9.4. Visão originada do estilo de decomposição (*module decomposition style*) para o módulo `Qt5ModelingWidgets` 181
9.5. *Widgets* implementados no módulo `Qt5ModelingWidgets` 182
9.6. Metamodelo da linguagem DuSE para especificação de espaços de projeto para domínios específicos 183
9.7. Visão originada do estilo de decomposição (*module decomposition style*) para o módulo `Qt5Optimization` 184
9.8. Visão originada do estilo de decomposição (*module decomposition style*) para a ferramenta DuSE-MT 185

9.9. Visão geral da interface gráfica de usuário do DuSE-MT 186

10.1. Roteiro do capítulo 10 191
10.2. Matriz de gráficos de dispersão do *Pareto-front* de referência P^*_{SW} obtido para o modelo do servidor *web* MIMO 197
10.3. Gráfico de convergência da métrica *hypervolume* durante as iterações de otimização do modelo do servidor *web* MIMO 198
10.4. Matriz de gráficos de dispersão do *Pareto-front* de referência P^*_C obtido para o modelo do *cluster* elástico para aplicações *MapReduce* 201
10.5. Gráfico de convergência da métrica *hypervolume* durante as iterações de otimização do modelo do *cluster* elástico para aplicações *MapReduce* 202
10.6. Dinâmica do `httpd`, em malha aberta, para a utilização de CPU, número de *threads worker* e consumo de memória, em função dos parâmetros `MaxRequestWorkers` e `KATimeout` 206
10.7. Dinâmica do `httpd`, em malha fechada, utilizando os controladores que implementam as arquiteturas candidatas c_1, c_2 e c_d 209
10.8. Componentes fundamentais do HDFS e do YARN (*Hadoop* 2.3.0) 212
10.9. Dinâmica do *Hadoop*, em malha fechada, utilizando os controladores que implementam as arquiteturas candidatas c_1, c_2 e c_d 216
10.10. Objetos de projeto arquitetural utilizados no experimento. 220
10.11. Exemplo de *Pareto-front* de referência P^*e *Generational Distance* 221
10.12. Gráficos "*box and whispers*" para efetividade de projeto, complexidade de projeto e potencial de aquisição de conhecimento refinado de projeto 225

11.1. Roteiro do capítulo 11 231

Lista de Tabelas

2.1. Escolas de gestão de conhecimento propostas por Earl 19

3.1. Dimensões de caracterização de ambientes de operação de agentes inteligentes 41
3.2. Diferentes tipos (arquiteturas) de agentes inteligentes 42
3.3. Comparação entre controle em malha aberta e malha fechada 47
3.4. Indicadores para avaliação de modelos obtidos via identificação de sistema . . 51
3.5. Análise de propriedades do sistema a partir da função de transferência 52
3.6. Método empírico de Ziegler-Nichols 62
3.7. Método empírico de Chien-Hrones-Reswick 62
3.8. Método empírico de Cohen-Coon 62

6.1. Metamodelo do modelo apresentado na Figura 6.2 114
6.2. Modelo (sintaxe abstrata) da representação gráfica apresentada na Figura 6.2 . 114
6.3. Exemplo de modificações de inclusão de elementos na especificação de ponto de variação para inclusão de controlador 116
6.4. Exemplo de modificações de alteração de propriedades na especificação de ponto de variação para inclusão de controlador 116

7.1. Modificações do ponto de variação VP21 137
7.2. Modificações do ponto de variação VP22 139
7.3. Modificações do ponto de variação VP31 141
7.4. Modificações do ponto de variação VP36 143
7.5. Modificações do ponto de variação VP41 146
7.6. Modificações do ponto de variação VP46 148
7.7. Modificações do ponto de variação VP47 149
7.8. Modificações do ponto de variação VP52 151
7.9. Modificações do ponto de variação VP62 157
7.10. Modificações do ponto de variação VP64 158
7.11. Modificações do ponto de variação VP65 159

10.1. Resultados dos testes de hipótese para verificação de convergência do *hypervolume* na otimização do modelo do servidor *web* MIMO 199
10.2. Resultados dos testes de hipótese para verificação de convergência do *hypervolume* na otimização do modelo do *cluster* elástico para aplicações *MapReduce* 203
10.3. Parâmetros do `httperf` utilizados na experimentação com o servidor *web* `httpd` 205
10.4. Arquiteturas implementadas nos protótipos reais de sistemas gerenciadores para o `httpd` 207

10.5. Resultados dos testes de hipótese para verificação da acurácia das predições de tempo de resposta e sobressinal no `httpd` 211
10.6. Arquiteturas implementadas nos protótipos reais de sistemas gerenciadores para o *Hadoop* 214
10.7. Resultados dos testes de hipótese para verificação da acurácia das predições de tempo de resposta e sobressinal no *Hadoop* 217
10.8. Cronograma de preparação e realização do experimento 219
10.9. Testes definidos para o experimento 224
10.10. Estatísticas descritivas para as variáveis dependentes do experimento 225
10.11. Resultados do teste de normalidade de Anderson-Darling e teste de heteroscedasticidade de Brown-Forsythe 226
10.12. Resultados do teste de normalidade de Anderson-Darling e teste de heteroscedasticidade de Brown-Forsythe 226

11.1. Abordagens de engenharia de *software* para sistemas *self-adaptive* 236
11.2. Abordagens de projeto automatizado de arquiteturas de *software* 238
11.3. Abordagens de processo de projeto arquitetural baseado em atributos de qualidade 239

Lista de Algoritmos

4.1. Algoritmo simples para obtenção das soluções não-dominadas 72
4.2. Algoritmo para obtenção das soluções não-dominadas com atualização contínua . 73
4.3. Algoritmo de Kung et al. para obtenção das soluções não-dominadas 74
4.4. Algoritmo rápido para ordenação por não-dominância 81
4.5. Algoritmo para cálculo da distância de aglomeração 82
4.6. Algoritmo NSGA-II para otimização multiobjetivo 82

6.1. Algoritmo para derivação do espaço de projeto específico de aplicação 122
6.2. Algoritmo para geração de arquiteturas 123

8.1. Operador de recombinação com detecção de indivíduos *offspring* inválidos 171

Lista de Siglas

ADD	Attribute-Driven Design
ADL	Architecture Description Language
AG	Algoritmo Genético
ARX	AutoRegressive eXogeneous
ASC	Architectural Separation of Concerns
ATAM	Architectural Trade-off Analysis Method
BAPO	Business, Architecture, Process and Organization
BIBO	Bounded Input Bounded Output
CBAM	Cost Benefit Analysis Method
CC	Cohen-Coon
CCS	Calculus of Communicating Systems
CHR	Chien-Hrones-Reswick
COCOMO	COnstructive COst MOdel
COTS	Commercial Off-the-Shelf
CSP	Communicating Sequential Processes
CWM	Common Warehouse Metamodel
DA	Distância de Aglomeração
DASBSE	Dynamic Adaptive Search-Based Software Engineering
DSL	Domain-Specific Language
ECA	Event Condition Action
FOPDT	First-Order Plus Dead Time
GPL	General-Purpose Language
IPDT	Integrator Plus Dead Time
JMX	Java Management eXtension
JVMTI	JVM Tool Interface
LIT	Linear e Invariante no Tempo
LQR	Linear-Quadratic Regulator
MBE	Model-Based Engineering
MDA	Model-Driven Architecture
MDD	Model-Driven Development
MDE	Model-Driven Engineering
MDL	Many-Domains Language
MIAC	Model Identification Adaptive Control
MIMO	Multiple-Input Multiple-Output
MOF	Meta Object Facility

MPC	Model Predictive Control
MRAC	Model Reference Adaptive Control
NND	Nível de Não-Dominância
NSGA	Non-dominated Sorting Genetic Algorithm
OCL	Object Constraint Language
ODM	Ontology Definition Metamodel
PID	Proportional-Integral-Derivative
POFOD	Probability Of Failure On Demand
QoS	Quality of Service
QVT	Query/View/Transformation
RMSE	Root Mean-Square Error
RUP	Rational Unified Process
SBSE	Search-Based Software Engineering
SISO	Single-Input Single-Output
SLA	Service Level Agreement
SPEM	Software Process Engineering Metamodel
UML	Unified Modeling Language

Capítulo 1

Introdução

Simple pleasures are the last healthy refuge in a complex world.
Oscar Wilde

Esta tese apresenta o projeto, implementação e avaliação de uma abordagem para projeto arquitetural automatizado de sistemas *self-adaptive*, baseada na representação sistemática de conhecimento refinado de projeto e na aplicação de técnicas de otimização multiobjetivo. Este capítulo apresenta a motivação, as questões de pesquisa envolvidas, as contribuições realizadas e uma visão geral do conteúdo apresentado nesta tese.

1.1. Motivação

Ao longo dos últimos anos, uma série de novas demandas [141, 224, 267] contribuíram para aumentar ainda mais a complexidade essencial de sistemas de *software*. A disponibilização de computadores com melhor desempenho e menor custo, redes de comunicação mais velozes e dispositivos com maior capacidade de armazenamento de dados traz, a cada dia, novas oportunidades e desafios ao uso de *software* como elemento estratégico central de negócio. Sistemas altamente distribuídos, heterogêneos e com demandas rigorosas por escalabilidade (*ultra-large scale*), baixo consumo de energia, *dependability*, facilidade de integração, flexibilidade e autogerenciamento na presença de ambientes incertos têm sido o foco de muitas pesquisas atuais. O objetivo é a proposta de soluções de engenharia de *software* que gerenciem a complexidade essencial e reduzam a complexidade acidental em tais cenários.

Mesmo utilizando tecnologias poderosas para gerência de complexidade, acredita-se que a capacidade humana de compreensão e manipulação de artefatos de *software* implicará, em poucos anos, em limites na escalabilidade de aplicação destas tecnologias [145]. A complexidade essencial continuará crescendo até o ponto em que os profissionais mais habilidosos e as tecnologias mais efetivas não serão suficientes para o desenvolvimento de produtos de qualidade. Uma possível solução – investigada por uma série de pesquisas recentes [145, 168, 266] – é a transferência de certas atividades de processo para o próprio *software*, dotando-o com alguma capacidade de autogerenciamento ou auto-adaptação.

Ao mesmo tempo, a transformação de conhecimento tácito de projeto de arquiteturas de *software* em conhecimento estruturado de forma sistemática é fundamental para alavancar o desenvolvimento de soluções mais efetivas. A prática comum envolve o uso de arquiteturas de referência e catálogos de estilos e padrões arquiteturais [297]. Embora tais abordagens já tragam benefícios

para uma formação abreviada de bons arquitetos, a falta de suporte para manipulação direta deste conhecimento por ferramentas e a dificuldade de associação e comparação de arquiteturas candidatas limita os benefícios destas abordagens no uso rotineiro pela indústria.

Arquiteturas de notória efetividade são frequentemente caracterizadas por uma combinação sensata de estilos arquiteturais, conectores sofisticados e aspectos comportamentais bem elaborados. Conhecer e documentar quais táticas arquiteturais são mais adequadas à indução de determinados atributos de qualidade são atividades valiosas pois favorecem o reuso de soluções efetivas e tornam o processo de projeto mais previsível.

Adicionalmente, a natureza frequentemente conflitante dos atributos de qualidade envolvidos no projeto de arquiteturas torna mais difícil uma ponderação bem informada e criteriosa das soluções alternativas disponíveis. *Trade-offs* (soluções de compromisso) não conhecidos ou a ausência de mecanismos para manifestação explícita de tais *trade-offs* constituem obstáculos ao projeto de arquiteturas efetivas pois tornam mais provável a adoção de soluções inferiores, decorrentes de algum viés tecnológico, falsa intuição ou conhecimento parcial do espaço de solução em questão.

No caso particular de sistemas *self-adaptive*, identificar as principais dimensões de modelagem e apresentar diretrizes para um levantamento efetivo de requisitos de adaptação não é uma tarefa fácil. Além disso, um amplo conjunto de técnicas para implementação de capacidades de adaptação em sistemas computacionais está disponível atualmente. A ausência de mecanismos mais disciplinados e sistematizados para organização de conhecimento de projeto e para geração e avaliação facilitadas de alternativas arquiteturais é fator impeditivo para uma maior adoção e completa realização dos sistemas *self-adaptive* na indústria de *software*.

Conhecer as diferentes problemáticas, metas, estratégias e consequências da adoção de uma determinada abordagem para implementação de comportamento *self-adaptive* requer, atualmente, uma formação especializada que é demorada, custosa e que demanda um amplo conjunto de pré-requisitos. Ao mesmo tempo em que parte desta dificuldade é característica de um domínio inerentemente complexo, acredita-se que outra parcela é decorrente da falta de organização sistemática de conhecimento especializado que, se disponível, barreiras conceituais e técnicas entre o engenheiro de *software* e o engenheiro de controle poderiam ser minimizadas. Este cenário é o ponto central das motivações deste trabalho.

1.2. Questões de Pesquisa

As seguintes questões de pesquisa foram identificadas para este trabalho:

1. Como representar – de forma mais sistemática e estruturada – o conhecimento refinado utilizado no projeto de arquiteturas de sistemas *self-adaptive* de alta qualidade, com o objetivo de alavancar sua utilização na prática corrente ?
2. Quais mecanismos podem ser utilizados para representar conflitos de atributos de qualidade deste domínio como elementos de primeira classe e disponibilizar uma abordagem mais rigorosa para a análise destes *trade-offs* ?
3. Como tornar esta infraestrutura genérica o suficiente de modo a ser passível de uso em outros domínios de aplicação ?
4. O uso destas representações sistemáticas de conhecimento de projeto arquitetural de sistemas *self-adaptive* conduz a soluções mais efetivas em relação ao atendimento dos atributos de qualidade frequentemente desejados neste domínio de aplicação ?

1.3. Contribuições da Tese

Conforme apresentado na Figura 1.1, as abordagens apresentadas nesta tese estão associadas à utilização de mecanismos existentes e à contribuição de novas soluções em três amplas áreas de pesquisa: *i)* engenharia de *software*; *ii)* sistemas *self-adaptive*; e *iii)* otimização.

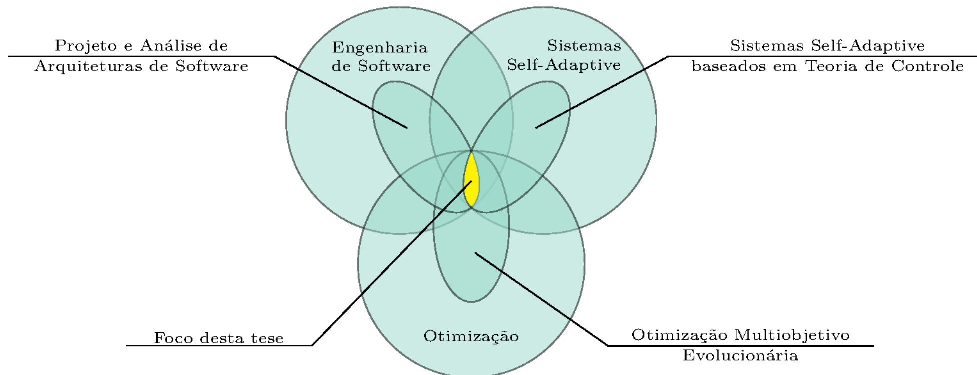


Figura 1.1.: Áreas de conhecimento envolvidas nesta tese.

Os produtos desta tese podem ser classificadas em dois grupos: contribuições para projeto e análise arquitetural independentes de domínio e contribuições para projeto e análise de arquiteturas para sistemas *self-adaptive*. As principais contribuições independentes de domínio são:

- A formalização de uma infraestrutura para espaços de projeto (Capítulo 6);
- Uma linguagem para modelagem de espaços de projeto de domínios específicos – DuSE (Capítulo 9);
- Um mecanismo para otimização multiobjetivo de arquiteturas de *software* (Capítulo 8);
- Uma biblioteca para operações de modelagem e metamodelagem – `Qt5Modeling` (Capítulo 9);
- Um *application framework* para otimização multiobjetivo – `Qt5Optimization` (Capítulo 9);
- Uma ferramenta de suporte ao processo de projeto automatizado de arquiteturas aqui apresentado – DuSE-MT (Capítulo 9).

As contribuições específicas da área de projeto de sistemas *self-adaptive* são:

- Um espaço de projeto para sistemas *self-adaptive* – SA:DuSE (Capítulo 7);
- Avaliação experimental do SA:DuSE: estudos de avaliação de desempenho e qualidade das otimizações arquiteturais realizadas, experimentos com protótipos reais de *feedback control loops* para o servidor *Apache HTTP* e para a plataforma de computação distribuída *Apache Hadoop* e experimento controlado comparando a abordagem aqui proposta a métodos clássicos de projeto arquitetural (Capítulo 10).

1.4. Visão Geral da Tese

Esta tese está dividida em três partes. A Parte I apresenta os fundamentos das áreas de conhecimento envolvidas neste trabalho e está organizada como segue.

Capítulo 2. Apresenta os fundamentos sobre: arquiteturas de *software*; processos de desenvolvimento de *software* e processos de projeto arquitetural em particular; análise arquitetural e

atributos de qualidade e; finalmente, linguagens de modelagem e metamodelagem. As informações apresentadas neste capítulo subsidiam uma melhor compreensão do processo automatizado de projeto arquitetural, das métricas de avaliação de atributos de qualidade e da nova linguagem de modelagem para especificação de espaços de projeto de domínios específicos propostos neste trabalho.

Capítulo 3. Apresenta os fundamentos sobre sistemas *self-adaptive* em geral e, em particular, sobre os mecanismos viabilizadores de autogerenciamento baseados em Teoria de Controle. Uma discussão sobre as dimensões mais prevalentes do espaço de problema e espaço de solução deste domínio de aplicação é inicialmente realizada. Com isso, apresenta-se uma visão geral do espectro atual de mecanismos e tecnologias utilizados na implementação de sistemas *self-adaptive* e motiva-se a necessidade de representação sistemática de conhecimento de projeto arquitetural. Em seguida, uma visão mais detalhada sobre o mecanismo particular de viabilização de autogerenciamento adotado nesta tese – Teoria de Controle – é apresentada. Discute-se as principais técnicas para modelagem da dinâmica de sistemas computacionais, as leis de controle e técnicas de sintonia de controladores mais utilizadas e as ferramentas para análise das propriedades apresentadas pelo sistema final. Parte dos assuntos discutidos neste capítulo representam exatamente o conhecimento refinado de projeto – particular da área de sistemas *self-adaptive* – que foi sistematicamente capturado na abordagem aqui apresentada.

Capítulo 4. Apresenta os fundamentos sobre problemas de otimização em geral e, em particular, sobre o uso de algoritmos evolucionários para resolução de problemas de otimização multiobjetivo. São apresentados os conceitos básicos de dominância entre soluções e *Pareto-optimality*, seguidos de uma visão geral sobre a abordagem evolucionária particularmente adotada nesta tese – o NSGA-II. Finalmente, os principais indicadores para avaliação do desempenho das execuções de otimização são apresentados. As informações apresentadas neste capítulo subsidiam uma melhor compreensão do mecanismo de busca por arquiteturas efetivas proposto nesta tese e dos estudos de avaliação conduzidos.

A Parte II reúne as contribuições desta tese e está organizada como segue.

Capítulo 5. Apresenta uma visão geral da abordagem de projeto arquitetural automatizado de sistemas *self-adaptive* proposta. Inicialmente, expõe-se as motivações e requisitos que guiaram o projeto da solução aqui apresentada e discute-se os exemplos – utilizados ao longo da tese – de modelos de sistemas aos quais se deseja adicionar capacidades de autogerenciamento. Ao final, uma visão geral do processo automatizado proposto é apresentada.

Capítulo 6. Apresenta a formalização do mecanismo para representação sistemática de conhecimento refinado de projeto proposto neste trabalho. A solução foi concebida de modo a não estar restrita ao uso somente no domínio de sistemas *self-adaptive*. Os requisitos levantados para o mecanismo e os constructos desenvolvidos para viabilizar a captura de conhecimento são discutidos. Finalmente, os algoritmos para derivação do espaço de projeto associado a um problema particular de projeto arquitetural e para automação da geração das diversas arquiteturas do espaço são apresentados e debatidos.

Capítulo 7. Apresenta como o mecanismo genérico para captura de conhecimento de projeto arquitetural apresentado no Capítulo 6 foi instanciado com o objetivo de representar sistematicamente as técnicas e formalismos para *feedback control loops* apresentados no Capítulo 3. São apresentadas as dimensões de projeto que viabilizam a produção automática de arquiteturas representando diferentes estratégias para construção de sistemas *self-adaptive*, bem como as métricas que avaliam as arquiteturas geradas em relação a propriedades particulares da área de Teoria de Controle.

Capítulo 8. Apresenta como as técnicas de otimização multiobjetivo discutidas no Capítulo 4 foram utilizadas para viabilizar a busca automática por arquiteturas efetivas em espaços de

projeto construídos com base no formalismo apresentado no Capítulo 6. São discutidos os aspectos técnicos que viabilizaram a representação de arquiteturas de forma adequada à manipulação pelos algoritmos evolucionários de otimização.

Capítulo 9. Apresenta os aspectos de projeto e implementação da ferramenta desenvolvida, como parte deste trabalho, para suportar o processo automatizado de projeto arquitetural proposto. São apresentados os requisitos funcionais e não-funcionais da aplicação, os requisitos e diretrizes arquiteturais, os módulos (bibliotecas) desenvolvidos e algumas questões técnicas sobre a implementação.

Finalmente, a Parte III apresenta as validações realizadas, os trabalhos correlatos e as conclusões desta tese.

Capítulo 10. Apresenta as metas de avaliação do trabalho, os instrumentos de investigação utilizados em cada meta e os resultados obtidos. A abordagem foi avaliada em relação a: *i)* sua capacidade de identificação dos *trade-offs* comumente encontrados no projeto de sistemas *self-adaptive* (via indicadores de desempenho da área de otimização); *ii)* acurácia com a qual os valores de atributos de qualidade obtidos com a abordagem aqui proposta são de fato observados em sistemas reais (via comparação com protótipos funcionais); e *iii)* o impacto na efetividade e complexidade das arquiteturas produzidas quando comparado com técnicas convencionais de projeto arquitetural (via experimento controlado).

Capítulo 11. Apresenta os trabalhos correlatos à abordagem aqui proposta. Tais trabalhos são classificados em três diferentes dimensões de correlação: abordagens de engenharia de *software* para sistemas *self-adaptive*, abordagens de projeto automatizado de arquiteturas de *software* e processos de projeto arquitetural baseados em atributos de qualidade.

Capítulo 12. Apresenta as conclusões e as potenciais oportunidades de trabalhos futuros.

1.4.1. Como Ler esta Tese

Conforme mencionado anteriormente, as contribuições apresentadas nesta tese resultam da intersecção de três áreas de conhecimento: engenharia de *software*, sistemas *self-adaptive* e otimização multiobjetivo. De modo a facilitar a leitura deste texto e direcionar o leitor aos conteúdos mais relacionados aos seus interesses, quatro diferentes guias são abaixo apresentados.

Leitura Rápida. Adequada para obter uma visão geral e superficial das questões de pesquisa investigadas, métodos adotados, soluções propostas e avaliações realizadas. O seguinte roteiro deve ser seguido neste caso:

- Capítulo 1 (pág. 1);
- Capítulo 5: processo automatizado de projeto arquitetural de sistemas gerenciadores (pág. 104) + resumo (pág. 107) + premissas e limitações (pág. 106);
- Capítulo 6: resumo (pág. 127) + premissas e limitações (pág. 124);
- Capítulo 7: resumo (pág. 164);
- Capítulo 8: resumo (pág. 173) + premissas e limitações (pág. 172);
- Capítulo 10: resumo (pág. 228);
- Capítulo 12 (pág. 243).

Leitura com Foco em Engenharia de Software. Adequada para leitores interessados em arquiteturas de *software*, processos de desenvolvimento de *software* e linguagens de modelagem. O seguinte roteiro deve ser seguido neste caso:

- Capítulo 1 (pág. 1);
- Capítulo 2 (pág. 11);
- Capítulo 5 (pág. 91);
- Capítulo 6 (pág. 109);
- Capítulo 9 (pág. 175);
- Capítulo 10: estudo 3 (pág. 218);
- Capítulo 12 (pág. 243).

Leitura com Foco em Sistemas Self-Adaptive. Adequada para leitores interessados em modelagem de sistemas dinâmicos, Teoria de Controle, sistemas autonômicos, *cyber-physical computing* e sistemas autogerenciáveis. O seguinte roteiro deve ser seguido neste caso:

- Capítulo 1 (pág. 1);
- Capítulo 3 (pág. 27);
- Capítulo 5 (pág. 91);
- Capítulo 7 (pág. 131);
- Capítulo 10: estudo 2 (pág. 204);
- Capítulo 12 (pág. 243).

Leitura com Foco em Otimização Multiobjetivo. Adequada para leitores interessados em algoritmos genéticos, otimização evolucionária, engenharia de *software* baseada em busca e uso de metaheurísticas. O seguinte roteiro deve ser seguido neste caso:

- Capítulo 1 (pág. 1);
- Capítulo 4 (pág. 65);
- Capítulo 5 (pág. 91);
- Capítulo 8 (pág. 167);
- Capítulo 10: estudo 1 (pág. 194);
- Capítulo 12 (pág. 243).

1.5. Publicações

As seguintes publicações foram produzidas ao longo da realização desta tese, todas de autoria de Sandro S. Andrade e Raimundo José de Araújo Macêdo:

1) *A Non-Intrusive Component-Based Approach for Deploying Unanticipated Self-Management Behaviour*

Local de publicação: ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS). 2009. Vancouver - Canadá.

Tópicos abordados: experimentação preliminar com sistemas *self-adaptive*.

2) *Architectural Design Spaces for Feedback Control Concerns in Self-Adaptive Systems*

Local de publicação: 25th International Conference on Software Engineering and Knowledge Engineering (SEKE). 2013. Boston - MA - EUA.

Tópicos abordados: fundamentos para espaços de projeto arquitetural e realização de exploração manual de soluções (Capítulo 6).

3) *DuSE-MT: From Design Spaces to Automated Software Architecture Design*

Local de publicação: 25th International Conference on Software Engineering and Knowledge Engineering (SEKE) – Demo Session. 2013. Boston- MA - EUA. Best Demo Award.

Tópicos abordados: aspectos de projeto e implementação da ferramenta, referentes à construção de novos espaços de projeto e exploração manual de soluções (Capítulo 9).

4) *A Search-Based Approach for Architectural Design of Feedback Control Concerns in Self-Adaptive Systems*

Local de publicação: 7th IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO). 2013. Philadelphia - PA - EUA.

Tópicos abordados: espaço de projeto para sistemas *self-adaptive* (Capítulo 7) e mecanismos para otimização multiobjetivo de arquiteturas de *software* (Capítulo 8).

5) *Toward Systematic Conveying of Architecture Design Knowledge for Self-Adaptive Systems*

Local de publicação: 7th IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO) – Demo Session. 2013. Philadelphia - PA - EUA.

Tópicos abordados: aspectos de projeto e implementação da ferramenta referentes à busca automática por soluções não-dominadas (Capítulo 9).

6) *A Systematic Model-Based Approach for Feedback Control Trade-Off Evaluation in Autonomic Systems*

Local de publicação: 4° Workshop de Sistemas Distribuídos Autonômicos (WoSiDA) – 32° Simpósio Brasileiro de Redes de Computadores (SBRC). 2014. Florianópolis - SC - Brasil.

Tópicos abordados: visão geral dos experimentos do Estudo 2 (Capítulo 10).

7) *Do Search-Based Approaches Improve the Design of Self-Adaptive Systems ? A Controlled Experiment*

Local de publicação: Congresso Brasileiro de Software: Teoria e Prática (CBSoft) – 28° Simpósio Brasileiro de Engenharia de Software (SBES). 2014. Maceió - AL - Brasil. Prêmio de 2° melhor artigo do SBES 2014.

Tópicos abordados: visão geral dos experimentos do Estudo 3 (Capítulo 10).

8) *Assessing the Benefits of Search-Based Approaches when Designing Self-Adaptive Systems Architectures: A Controlled Experiment*

Local de publicação: Journal of Software Engineering Research and Development 2015, 3:2 (24 de março de 2015).

Tópicos abordados: versão ampliada da descrição dos experimentos do Estudo 3 (Capítulo 10).

1.6. Convenções Tipográficas

As seguintes convenções tipográficas foram adotadas nesta tese:

- São apresentadas em *itálico* as palavras da lingua inglesa cuja grafia não foi aportuguesada. Exemplos: *software* e *role*. Contra-exemplo: contêiner (palavra da lingua inglesa, porém com grafia aportuguesada). Todos as traduções de termos em inglês passíveis de serem realizadas sem prejuízos para a compreensão do texto foram adotadas. O primeiro uso do termo traduzido é acompanhado do termo original em inglês entre parênteses.
- São apresentados em **negrito** as primeiras ocorrências de termos que representam conceitos importantes (definições) nesta tese. Tais termos devem ser compreendidos segundo as definições aqui apresentadas.
- São apresentados com fonte `Typewriter` os constructos referentes a artefatos de projeto e implementação das soluções propostas nesta tese. Exemplos: nomes de classes, interfaces, componentes e conectores.

Todos os gráficos e ilustrações desta tese foram criados com as ferramentas TikZ (`http://www.texample.net/tikz/`), PGFPlots (`http://pgfplots.sourceforge.net/`) e Color Brewer (`http://colorbrewer2.org/`).

Parte I

Fundamentos

Capítulo 2

Projeto e Análise de Arquiteturas de Software

The paradox of technology should never be used as an excuse for poor design.
Donald A. Norman (The Design of Everyday Things)

A prospecção de objetos artificiais (artefatos), intencionalmente projetados para exibir propriedades desejadas, é o objetivo principal das atividades de engenharia. Com a publicação do livro *The Sciences of the Artificial* [279], de Herbert Simon, em 1969, inicia-se a busca por uma doutrina intelectualmente planejada, analítica, parcialmente formalizável, parcialmente empírica e lecionável sobre o processo de projeto (*design*) de tais artefatos. O objetivo é a definição de um conjunto de teorias para avaliação e representação formal de projetos, bem como mecanismos de busca por soluções alternativas e abordagens para estruturação e organização destes projetos.

O projeto de sistemas de *software* – enquanto representantes legítimos de objetos artificiais – ao mesmo tempo em que impõe desafios particulares à representação sistemática de conhecimento de projeto, se beneficia direta e substancialmente dos produtos de tais pesquisas. A alta complexidade de algumas aplicações e a presença de requisitos não-funcionais rigorosos demandam a derivação de projetos sofisticados, não-óbvios e obtidos através da aplicação de conhecimento altamente especializado.

A utilização de projetos inferiores geralmente acarreta maiores custos (devido a constantes correções), consumo ineficiente de recursos e atendimento parcial de requisitos não-funcionais [48]. Dessa forma, acredita-se que a sistematização do conhecimento e do processo de projeto de *software* constitui atividade importante para melhorar a eficácia da prática diária da construção de tais artefatos.

Este capítulo apresenta os fundamentos sobre arquiteturas de *software* e sua relação com atributos de qualidade, bem como os mecanismos atualmente utilizados para captura de conhecimento específico de projeto arquitetural de *software*. Adicionalmente, são apresentadas as técnicas para representação de modelos arquiteturais e como tais modelos podem subsidiar operações valiosas de análise – responsáveis por estimar o grau de indução dos atributos de qualidade desejados. A Figura 2.1 apresenta o roteiro deste capítulo.

2.1. Arquiteturas de Software

Embora as primeiras ideias de arquitetura de *software* como organização estrutural básica de aplicações estejam presentes em algumas pesquisas das décadas de 70 e 80, somente a partir

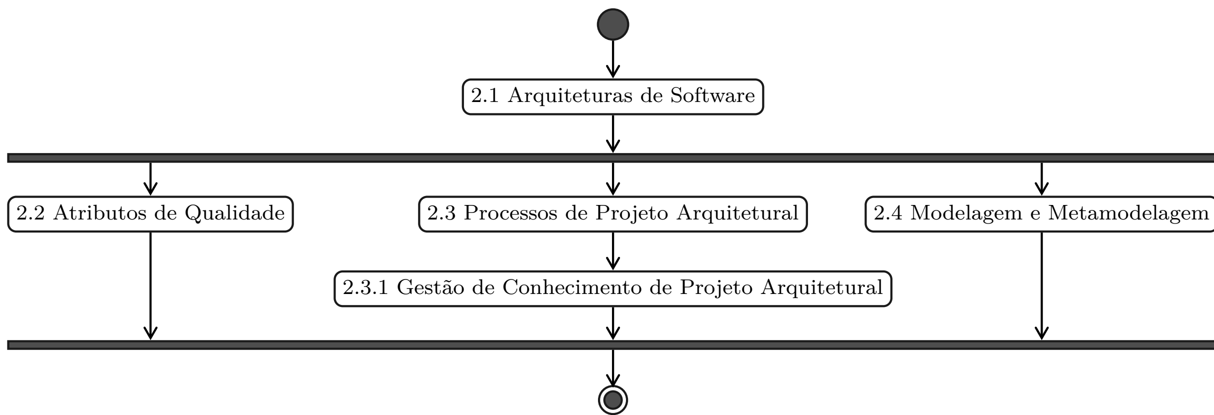


Figura 2.1.: Roteiro do capítulo 2.

de 1991 – com a publicação dos trabalhos de Winston Royce & Walker Royce [258] e Perry & Wolf [243] – passou-se a conceber a área de Arquitetura de Software como disciplina particular. De lá até os dias atuais, um vasto conjunto de conhecimentos para projeto, análise, realização (implementação) e modelagem (representação) de arquiteturas de *software* foi desenvolvido [181, 278, 297].

Ainda que amplamente reconhecida a importância da arquitetura de *software* no desenvolvimento de sistemas complexos [64, 277], uma série de requisitos para uma adoção mais efetiva pela indústria ainda representam desafios. A definição de relacionamentos formais entre decisões de projeto e atributos de qualidade, uso de notações efetivas para modelagem de arquiteturas, organização de conhecimento de projeto arquitetural para uso rotineiro e mecanismos de verificação de conformidade entre arquiteturas e artefatos de implementação representam algumas das frentes atuais de pesquisa [275]. Por outro lado, a real natureza sociotécnica das arquiteturas de *software* e os limites alcançáveis de qualidade em projetos continuam sendo atualmente questionados [42, 43, 44].

Apesar da maturidade reconhecidamente alcançada pela disciplina nos últimos anos, não há consenso sobre uma definição satisfatória e concisa sobre o que é uma arquitetura de *software*. Uma ampla compilação de definições pode ser encontrada em [150]. O conceito de **arquitetura de software** utilizado neste trabalho baseia-se na definição proposta por Perry & Wolf, em 1992:

Definição 2.1: Arquitetura de Software (Perry & Wolf, 1992, pág. 5)

Uma arquitetura de *software* é um conjunto formado por: *i)* elementos (de processamento, de armazenamento de dados e de conexão); *ii)* propriedades e relacionamentos que definem uma forma arquitetural; e *iii)* descrição das intenções, premissas e restrições que justificam a arquitetura (*rationale*).

A dificuldade na transmissão da real natureza das arquiteturas de *software* e dos benefícios da sua aceitação como artefato de fundamental importância nos projetos exige, da literatura, a utilização de artifícios didáticos para a realização desta tarefa. Analogias com arquiteturas na construção civil e diferenciação em relação a artefatos produzidos por outras atividades de projeto ajudam no entendimento das características, do papel desempenhado e das habilidades de projeto requeridas na construção das arquiteturas de *software*. Acredita-se, entretanto, que a completa assimilação desta pauta depende fortemente de uma exposição a vivências reais de atividades de projeto e análise arquitetural, catalisando a produção de experiência refinada.

Arquiteturas de *software* representam decisões de projeto de âmbito estratégico, responsáveis por viabilizar as macrocaracterísticas estruturais e comportamentais que irão induzir (ou invia-

bilizar), por fim, os atributos de qualidade desejáveis para o sistema em questão. Reconhecer a importância devida às arquiteturas dentro de um processo de desenvolvimento de *software* viabiliza um melhor controle intelectual e integridade conceitual em projetos de larga complexidade, uma base adequada e efetiva para reuso e uma comunicação mais facilitada no projeto.

Sistematizar as relações entre decisões arquiteturais e suas consequentes influências nos benefícios acima descritos é de fundamental importância para a completa realização de processos de desenvolvimento de *software* centrados em arquiteturas. Projetar e analisar arquiteturas é somente um meio para a obtenção, no produto final, de características desejadas [30].

Um determinado elemento arquitetural é denominado componente – se responsável por processamento das regras de negócio e/ou armazenamento de dados – ou conector – se responsável pela mediação e viabilização de interações entre componentes. O termo “componente” é geralmente sobrecarregado dentro da Ciência da Computação, assumindo diferentes significados dependendo da época e da área específica de uso. Em particular, muitas definições são ainda encontradas nas áreas de Engenharia de Software Baseada em Componentes [134] e Arquitetura de Software [297]. Neste trabalho, o conceito de **componente de software** segue a definição apresentada por Taylor et al., em 2009:

Definição 2.2: Componente de Software (Taylor et al., 2009, pág. 69)

Um componente de *software* é um elemento arquitetural que: *i)* encapsula um subconjunto das funcionalidades e/ou dados do sistema; *ii)* restringe o acesso a este subconjunto através de interfaces explicitamente definidas; e *iii)* possui dependências – explicitamente definidas – em relação ao seu contexto de execução.

Embora a definição acima ainda permita a derivação de diferentes modelos (semi-)formais de componentes de *software* [191] – com diferentes variações na sintaxe, semântica e no nível de expressividade do relacionamento com o contexto de execução – entende-se, neste trabalho, um componente de *software* (daqui pra frente, simplesmente **componente**) como uma unidade encapsulada de composição, independentemente compilável e implantável, cujos serviços de processamento e armazenamento de dados providos e requeridos são explicitamente definidos através de interfaces.

Tal entendimento é fundamentado nos trabalhos realizados por Heineman & Councill em 2001 [136] e Szyperski et al. em 2002 [294]. Um componente pode representar elementos arquiteturais de diferentes granularidades de composição, variando, por exemplo, de simples *plugins* com algumas poucas funcionalidades até subsistemas completos em aplicações mais complexas.

Um **conector de software** é um elemento arquitetural responsável por garantir uma interação apropriada entre os componentes. Embora sua importância tenha sida reconhecida há alguns anos [274], são ainda geralmente subapreciados na prática diária e não modelados explicitamente. A definição de conector de *software* utilizada neste trabalho é aquela proposta por Mary Shaw & David Garlan, em 1996:

Definição 2.3: Conector de Software (Shaw & Garlan, 1996, pág. 165)

Um conector de *software* é um elemento arquitetural responsável por mediar a interação entre dois ou mais componentes de *software*, estabelecendo as regras que governam esta interação e especificando quaisquer mecanismos auxiliares necessários.

Conectores de *software* (daqui pra frente, simplesmente **conectores**) passam a desempenhar um papel crucial à medida em que interações sofisticadas entre componentes passam a ser fator dominante para a obtenção dos atributos de qualidade desejados. Pode-se citar, como exemplos, o projeto de sistemas altamente heterogêneos, sistemas com demandas rigorosas por

escalabilidade (*ultra-large scale systems*), sistemas confiáveis (*dependable systems*) e aplicações que realizam adaptações em tempo de execução (*run-time*).

Nestes casos, os conectores são os elementos arquiteturais onde são definidos, por exemplo, o número de componentes envolvidos na interação, o protocolo de comunicação utilizado e o estado interno necessário para a implementação de mecanismos de suporte tais como *buffers* e algoritmos de mediação. A sofisticação desta mediação pode variar desde uma simples chamada de procedimento (*procedure call*) até mecanismos complexos para balanceamento de carga, sincronização de acesso a recursos compartilhados, tratamento de falhas e provisionamento dinâmico de recursos [215]. Diferente dos componentes, conectores são geralmente independentes de aplicação e, portanto, passíveis de serem facilmente realizados, por exemplo, através do uso de alguma solução de *middleware*.

Sob esta perspectiva, reconhecer os conectores como elementos arquiteturais de primeira classe – com características, critérios de seleção e consequências de uso próprios – é indispensável para a construção de modelos arquiteturais mais expressivos e com possibilidades mais complexas de análise. Parte deste trabalho já vem sendo realizada, em pesquisas com foco na definição de taxonomias [215], modelos formais [12, 98], conectores de mais alta ordem [204] e catálogos de conectores para domínios específicos [211]. Entretanto, ainda existem controvérsias sobre a real natureza dos conectores e sua relação com outros elementos arquiteturais estruturais [166].

Componentes e conectores são interligados para formar sistemas completos, em estruturas denominadas – por Taylor et al. – **configurações arquiteturais**:

Definição 2.4: Configuração Arquitetural (Taylor et al., 2009, pág. 72)

Uma configuração arquitetural é um conjunto de associações específicas entre os componentes e os conectores de uma arquitetura de *software*.

Da mesma forma que a correta seleção dos componentes e conectores de uma arquitetura definem diretamente o grau de obtenção dos atributos de qualidade requeridos, configurações arquiteturais bem planejadas viabilizam estruturas menos acopladas, mais escaláveis e com eventuais possibilidades de re-estruturação em tempo de execução. Outros autores utilizam os termos topologia [278], *attachments* [105] e *component assembly* [294] para se referir às configurações arquiteturais.

Associações entre componentes e conectores são realizadas através de **interfaces**, denominadas – em algumas literaturas [105] – portas (*ports*), quando presentes em componentes; e papéis (*roles*), quando presentes em conectores. Neste trabalho, utiliza-se o conceito genérico de interface como local de interligação entre componentes e conectores. Vale ressaltar que o uso do termo interface não implica, necessariamente, em uma interação baseada em chamada de procedimento (também conhecida como *call-return*). Mecanismos de comunicação com diferentes formas de transferência de dados e de controle, como o uso de eventos ou fluxos de dados (*data streams*), podem também ser implementados pelas interfaces [215, 297].

Processos de desenvolvimento de *software* centrados em arquiteturas [297] são caracterizados pela utilização de representações arquiteturais como artefatos-guia para a definição de diretrizes de suporte às atividades de implementação, testes, implantação e evolução. A relevância da arquitetura no suporte a estas atividades depende, dentre outros fatores, da reificação dos seus conceitos fundamentais em um **modelo arquitetural** e do grau de formalidade da notação utilizada na construção deste modelo.

Definição 2.5: Modelo Arquitetural (Taylor et al., 2009, pág. 75)

Um modelo arquitetural é um artefato que captura/documenta algumas ou todas as decisões de projeto que compõem a arquitetura do sistema.

Definir critérios para classificar uma decisão como arquitetural ou não-arquitetural é uma tarefa difícil. Decisões arquiteturais referem-se a aspectos do sistema de natureza diversa [155, 178, 301] e encontrar um equilíbrio adequado entre guiar e restringir as atividades subsequentes do processo é uma das habilidades esperadas de um bom arquiteto. Uma boa recomendação geral é não classificar como arquitetural (e portanto não impor restrições) uma decisão que não tem impacto direto em um ou mais atributos de qualidade do sistema.

Mesmo buscando um número mínimo de decisões arquiteturais que induzam os atributos de qualidade desejados, sistemas complexos geralmente têm a sua arquitetura descrita por vários modelos, que descrevem visões diferentes, porém complementares, da arquitetura do mesmo sistema. Modelos arquiteturais são descritos em alguma **notação de modelagem arquitetural**.

Definição 2.6: Notação de Modelagem Arquitetural (Taylor et al., 2009, pág. 185)

Uma notação de modelagem arquitetural é a linguagem ou os meios utilizados na captura (documentação) das decisões arquiteturais de projeto.

Diversas notações de modelagem arquitetural são utilizadas atualmente [297], desde o uso de *templates* de documentos em linguagem natural e gráficos informais [259] até representações descritas em UML (*Unified Modeling Language*) [106, 213] ou algum tipo particular de ADL (*Architecture Description Language*) [214]. Outras literaturas utilizam os termos “descrição arquitetural” [259] e “documento de arquitetura” [105] para se referir a modelos arquiteturais. Neste trabalho, a adoção de uma definição mais genérica de “notação de modelagem arquitetural” permite o uso mais abrangente do termo “modelo arquitetural” como qualquer tipo de documento que capture decisões arquiteturais de projeto.

A motivação para a construção de modelos arquiteturais pode variar desde a necessidade de uma melhor comunicação em âmbito estratégico até a utilização de arquiteturas como artefatos centrais para análises sofisticadas e automação das atividades subsequentes de implementação, testes, implantação e evolução. Decidir o quê, quanto e como modelar – ponderando os custos e benefícios da criação destes modelos – é um importante papel realizado pelos arquitetos de *software*.

2.2. Atributos de Qualidade

Projetar sistemas computacionais complexos envolve não só a correta implementação das funcionalidades esperadas mas também a obtenção de atributos de qualidade, tais como interoperabilidade, alto desempenho, segurança (*security*), alta disponibilidade, capacidade de adaptação em tempo de execução e facilidade de modificação (*modifiability*). Embora tais atributos dependam de fatores presentes em várias etapas de um processo de desenvolvimento de *software*, é atualmente evidente que grande parte do mérito do alcance destas propriedades deve-se a projetos arquiteturais criteriosos, geralmente frutos do uso de experiência refinada e da aplicação de táticas de indução dos atributos desejados [30].

A definição do termo “qualidade” e, em particular, “qualidade de *software*”, não é precisa. Qualidade é um conceito multidimensional, com interpretações diversas dependendo da parte interessada (*stakeholder*) em questão, do ponto de vista por ela utilizada e dos atributos de qualidade considerados. Além disso, a possibilidade de discussão sobre qualidade em diferentes níveis de abstração e o uso popular do termo, em contraponto a uma definição mais rigorosa e técnica, tornam os trabalhos ainda mais difíceis.

Duas definições técnicas de qualidade, amplamente adotadas na indústria, são: *i)* qualidade significa conformidade com os requisitos [69]; e *ii)* qualidade significa adequação ao uso [121].

A primeira definição requer que os requisitos sejam claramente especificados e compreendidos ("qualidade de conformidade"); não-conformidades são consideradas ausência de qualidade. A segunda definição leva em consideração as expectativas e necessidades das partes interessadas do produto sendo desenvolvido, dando ênfase ao que é conhecido como "qualidade de projeto" [223]. Visto que diferentes partes interessadas possuem diferentes visões sobre o que significa "adequação ao uso", a qualidade de projeto é geralmente descrita através do uso de vários **atributos de qualidade**.

Definição 2.7: Atributo de Qualidade

Um atributo de qualidade é uma expressão de qualidade do sistema que é de interesse de alguma parte interessada e cuja adequação ao uso pode ser estimada ou comparada.

Em particular, qualidade de *software* [162] envolve não só conformidade com os requisitos (ausência de *bugs*) mas também a satisfação de atributos específicos, tais como usabilidade, desempenho, confiabilidade, facilidade de instalação, boa documentação, etc. A importância cada vez maior do alcance destes atributos, bem como as dificuldades geradas por atributos conflitantes, requer que tais aspectos sejam antecipadamente considerados e discutidos durante o projeto arquitetural; e continuamente analisados nas fases subsequentes do processo.

Atributos de qualidade são diretamente influenciados por decisões tanto arquiteturais, quanto de implementação e implantação. Por exemplo, bom desempenho depende de aspectos arquiteturais tais como atribuição de funcionalidades a componentes, utilização de recursos compartilhados e características de comunicação (conectores); mas é também influenciado pela escolha dos algoritmos e estruturas de dados utilizados e pela disposição destes elementos na rede, no caso de sistemas distribuídos.

Sob esta perspectiva, acredita-se que atributos de qualidade podem e devem ser considerados durante o projeto e análise de arquiteturas, definindo os pilares que sustentam a complementação da sua realização em fases subsequentes [30, 116]. Ampliando a taxonomia presente em [30], pode-se classificar os atributos de qualidade em quatro principais categorias:

- Atributos de qualidade de sistema: propriedades referentes a aspectos não-funcionais gerais de uma aplicação, tais como disponibilidade, facilidade de modificação, desempenho e segurança. Embora haja divergência sobre como estes atributos devam ser denominados e principalmente mensurados, é senso comum evitar descrições não-operacionais e/ou imprecisas tais como "o sistema deve ser escalável". Uma solução é a utilização de requisitos de qualidade estruturados como, por exemplo, os cenários de atributo de qualidade propostos em [30].
- Atributos de qualidade de domínio específico: propriedades que abordam aspectos de qualidade específicos de um determinado domínio e que são impactados diretamente por atributos de qualidade de outras categorias. Por exemplo, sistemas multimídia para transmissão de vídeos sob demanda podem apresentar diferentes graus de cobertura de dispositivos (*device coverage*), dependendo do número de formatos de vídeo disponíveis para serem visualizados em resoluções de tela distintas. Diferentes arquiteturas para *cloud computing* podem apresentar flexibilidades variadas para a implementação de mecanismos de cobrança (*billing*). Embora estes atributos representem interpretações menos abstratas e que derivam dos atributos de qualidade de sistema, sua utilização em domínios de aplicação já maduros e consolidados pode tornar o processo de análise arquitetural mais efetivo. Atributos de qualidade de domínio específico são amplamente utilizados neste trabalho.
- Atributos de qualidade de negócio: propriedades referentes a aspectos econômicos, público-alvo e tempo de vida do *software*. Como exemplos, pode-se citar o tempo para comercialização (*time to market*), relação custo-benefício, amplitude de mercado a ser coberta e capacidade de integração com sistemas legados.

- Atributos de qualidade arquitetural: propriedades inerentemente relacionadas à arquitetura do *software* e que impactam diretamente o atendimento de atributos de qualidade de outras categorias. Como exemplos, pode-se citar integridade conceitual (presença de um único tema, visão ou ideia de projeto ao longo de toda a arquitetura), corretude/completude e facilidade de realização (*buildability*).

Por fim, é importante ressaltar que atributos de qualidade representam noções abstratas de qualidade e não disponibilizam, por si só, os meios para quantificar a qualidade de um sistema. É necessário a escolha criteriosa da métrica de qualidade mais adequada para estimar o atributo de qualidade em questão no sistema/cenário sendo especificamente estudado. Um amplo conjunto de técnicas qualitativas e quantitativas para mensuração de atributos de qualidade de *software* está atualmente disponível na literatura [152, 162, 185], abordando desde estimativas gerais de qualidade do produto e do processo até modelos específicos para estimativa de confiabilidade, complexidade e produtividade.

2.3. Processos de Projeto Arquitetural

Uma das principais causas de falhas na gerência de complexidade em projetos de sistemas computacionais modernos é a execução *ad-hoc* de atividades produtivas imaturas e de alto risco. A adoção sistemática de um **processo de desenvolvimento de software** ajuda a minimizar os riscos e potencializar o uso dos recursos disponíveis para a obtenção dos atributos de qualidade desejados.

Definição 2.8: Processo de Desenvolvimento de Software

Um processo de desenvolvimento de *software* é uma estrutura organizacional e processual que define, dentre outras coisas, as partes interessadas envolvidas, a dinâmica (atividades) e os produtos gerados em um projeto de desenvolvimento de sistemas computacionais.

Embora o objetivo final seja sempre o desenvolvimento de *software* de qualidade, diferentes processos de desenvolvimento de *software* apresentam diferentes graus de burocracia, agilidade, imunidade a riscos e centralidade arquitetural. Em particular, **processos de projeto arquitetural** tentam sistematizar a utilização de insumos e experiência refinada em atividades que conduzem à criação de arquiteturas de *software*. Funcionam como subprocessos dentro do processo geral de desenvolvimento de *software* e buscam disponibilizar diretrizes para esclarecer como e de onde surgem as boas arquiteturas.

Definição 2.9: Processo de Projeto Arquitetural

Um processo de projeto arquitetural define as atividades, habilidades e insumos que viabilizam o projeto e evolução de arquiteturas de *software*.

Processos de desenvolvimento de *software* e de projeto arquitetural são aglomerados complexos de atividades intelectuais e criativas. A escolha de um processo adequado a uma determinada realidade organizacional, técnica e de mercado, bem como a adoção de programas de especialização e melhoria contínuas, são fundamentais para uma comunicação mais efetiva, melhor produtividade e maior viabilidade de automação do processo.

A Figura 2.2 apresenta as atividades gerais de um processo de projeto arquitetural. A maioria dos processos de projeto arquitetural é formado por atividades iterativas (de maior ou menor agilidade) onde um subconjunto particular de requisitos é analisado em busca de aspectos arquiteturalmente significativos. Tais aspectos irão fundamentar a escolha de elementos arquiteturais

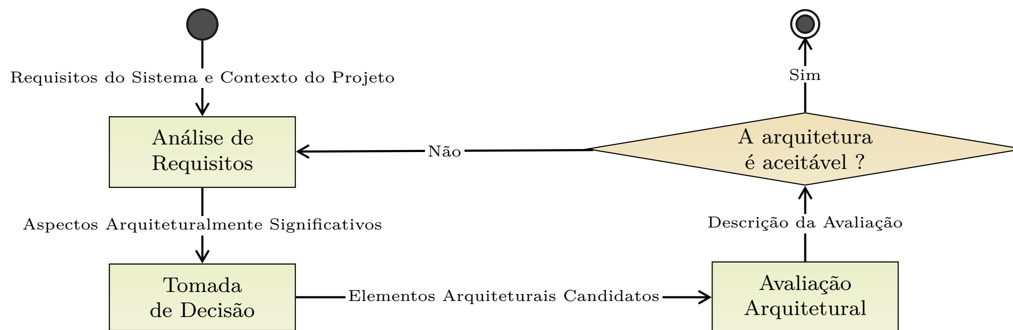


Figura 2.2.: Atividades gerais de um processo de projeto arquitetural (adaptada de [95]).

que, estrategicamente, irão constituir a arquitetura candidata. Esta arquitetura é então analisada para estimar o grau de atendimento dos atributos de qualidade relacionados.

A atividade "tomada de decisão" certamente requer a aplicação de experiência refinada e conhecimento específico de projeto arquitetural de modo a produzir soluções de boa qualidade. É também típica a situação onde projeta-se mais de uma arquitetura candidata de boa qualidade, possibilitando ao arquiteto uma posterior articulação de preferências sobre o grau de atendimento de atributos de qualidade conflitantes. Este cenário é parte central da ideia proposta nos próximos capítulo deste trabalho.

Embora não exista um processo "ideal" e sua adoção não seja fator determinante para a produção de *software* de qualidade, a definição de processos que identifiquem as principais fontes de conhecimento, táticas e elaborações utilizadas no projeto de arquiteturas tem sido o foco de muitas pesquisas recentes [17, 111, 120, 142, 156, 200, 201, 323]. Outras pesquisas investigam até que ponto a adoção de um processo de projeto arquitetural atende as reais necessidades dos arquitetos [95].

As diferentes intenções, forças condutoras e escopos dos processos de projeto arquitetural atualmente disponíveis permitem a sua classificação em três principais categorias:

- Processos guiados por visões: caracterizam-se pela especificação de atividades dedicadas à derivação de modelos que descrevem subconjuntos particulares das decisões de projeto que compõem a arquitetura do sistema (visões arquiteturais). Diferentes visões capturam decisões arquiteturais referentes a aspectos particulares, tais como estrutura, concorrência, distribuição de informação e implantação. Dentre os processos representantes desta categoria pode-se citar: a abordagem 4+1 presente no RUP (*Rational Unified Process*) [179] e o Siemens 4V [143].

- Processos guiados por atributos de qualidade: baseiam-se na identificação antecipada de atributos de qualidade a partir de requisitos arquiteturalmente significativos e na seleção de táticas de projeto arquitetural que induzem o atendimento destes atributos. Tais táticas são geralmente disponibilizadas sob a forma de estilos e padrões arquiteturais, arquiteturas de referência ou diretrizes gerais de uso de componentes, conectores e configurações arquiteturais [101, 297, 105]. Como representantes desta categoria pode-se citar o ADD (*Attribute-Driven Design*) [30, 323], o CBAM (*Cost Benefit Analysis Method*) [164] e o ATAM (*Architectural Trade-off Analysis Method*) [165].

- Processos guiados por *commonality/variability*: possuem como objetivo a produção de arquiteturas caracterizadas pela facilidade de acomodação de mudanças referentes a funcionalidades de um domínio de aplicação específico. São amplamente utilizados em linhas de produtos de *software* [65]. O BAPO/CAFCR [17] e o ASC (*Architecture Separation of Concerns*) [156] são exemplos desta categoria.

- Processos baseados em busca: caracterizam-se pela definição de representações sistematicamente estruturadas do espaço de problema (*problem space*) e do espaço de solução (*solution space*), associada à especificação de (meta)heurísticas para descoberta de arquiteturas candidatas com alto potencial de atendimento dos atributos de qualidade. Dentre os representantes desta categoria pode-se citar os trabalhos de Räihä et al. [254], Simons & Parmee [280], Koziolek [173] e a abordagem proposta nesta tese.

Vale ressaltar que processos de projeto arquitetural são apenas uma das fontes de obtenção de conhecimento de projeto descritas por Falessi et al. em [95]. Outras fontes incluem reutilização de soluções arquiteturais passadas e uso de experiência refinada e intuição.

2.3.1. Gestão de Conhecimento de Projeto Arquitetural

O alto grau de sofisticação de algumas arquiteturas e o reconhecimento da importância de tais artefatos para o sucesso de um projeto de *software* motivam a realização de pesquisas para prospecção de mecanismos para gestão de conhecimento de projeto arquitetural [27]. Arquiteturas de *software* são consequência de um conjunto de decisões de projeto.

Entretanto, a informação geralmente documentada em modelos arquiteturais é somente o resultado da tomada de tais decisões. As razões, intenções e ponderações realizadas pelo arquiteto (*rationale*) não são explicitamente capturadas, gerando questionamentos e retrabalhos desnecessários nas atividades de evolução e prejudicando a transmissão de conhecimento de projeto. Os principais desafios incluem esclarecimentos sobre o que é, como estruturar, armazenar e utilizar conhecimento de projeto de arquiteturas de *software*.

Não é objetivo deste trabalho discutir aspectos sobre conhecimento e gestão de conhecimento nos seus sentidos mais amplos; referências clássicas sobre o assunto podem ser encontradas na literatura [72, 83, 88]. A gestão de conhecimento tem como meta a transformação de conhecimento tácito (presente nas pessoas de forma implícita e não-estruturada) em conhecimento documentado (presente em algum artefato) ou, preferencialmente, conhecimento formalizado (documentado e organizado de maneira sistemática) [222].

Escola	**Subescola**	**Foco**	**Objetivo**	**Unidade de Conhecimento**
Tecnocrática	Sistemas	Tecnologia	Bases	Domínio
	Cartográfica	Mapas	Diretórios	Empresa
	Engenharia	Processos	Fluxos	Atividade
Econômica	Comercial	Renda	Ativos	*Know-how*
Comportamental	Organizacional	Redes	Agrupamentos	Comunidades
	Espacial	Espaço	Compartilhamento	Lugares de trabalho
	Estratégica	Mentalidade	Capacidades	Negócios

Tabela 2.1.: Escolas de gestão de conhecimento propostas por Earl [87].

Em um trabalho publicado em 2001 [87], Earl classifica as estratégias de gestão do conhecimento em três escolas principais, apresentadas na Tabela 2.1. As subescolas tecnocráticas são: *i)* escola de sistemas, que foca no uso de tecnologias para compartilhamento de bases de conhecimento; *ii)* escola cartográfica, baseada na criação de mapas e diretórios de conhecimento; e *iii)* escola de engenharia, preocupada com a criação de processos e fluxos de conhecimento dentro de empresas.

A subescola econômica-comercial busca investigar como ativos de conhecimento se relacionam com a renda de uma determinada corporação. Já a escola comportamental se divide em: *i)*

escola organizacional, focada na criação de redes de compartilhamento de conhecimento; *ii)* escola espacial, que investiga como espaços indutores de compartilhamento de conhecimento podem ser criados; e *iii)* escola estratégica, que estuda formas de uso do conhecimento como essência da estratégia de mercado da empresa.

Embora não constituam soluções completas para gestão de conhecimento em engenharia de *software*, as contribuições apresentadas neste trabalho seguem as escolas tecnocráticas de sistemas e de engenharia, visto que disponibilizam novas tecnologias e processos para projeto de arquiteturas de *software* de domínio específico.

Em particular, tomando como base os trabalhos realizados por Davenport [72] e Kruchten [180], pode-se derivar uma definição para **gestão de conhecimento de projeto arquitetural**.

Definição 2.10: Gestão de Conhecimento de Projeto Arquitetural

Gestão de conhecimento de projeto arquitetural é a utilização de processos que simplificam o compartilhamento, distribuição, criação, captura e compreensão do conhecimento utilizado no projeto de arquiteturas de *software*.

Gestão de conhecimento em engenharia de *software* tem sido o foco de muitas pesquisas recentes, incluindo revisões de literatura [35, 260], ontologias e representações estruturadas de decisões de projeto [180, 301], ferramentas [197] e concepção de espaços de projeto (*design spaces*) [20, 173, 187, 268, 276].

2.4. Modelagem e Metamodelagem

A utilização de modelos na engenharia de *software* [47, 287] é motivada por fatores diversos e adotada para servir a diferentes propósitos. Todo *software* é formado por contructos de código-fonte que realizam as intenções do sistema de forma mais ou menos nebulosa, fragmentada em elementos com nível de abstração fortemente dependente da habilidade do desenvolvedor.

Reconhecer os paradigmas e intenções gerais de construção do *software* diretamente a partir do código-fonte é uma atividade extremamente árdua. Modelos de *software* viabilizam a discussão dos artefatos em diferentes níveis de abstração, dependendo das partes interessadas envolvidas, estágio atual de desenvolvimento e objetivos da discussão (propósito descritivo). Modelos também podem assumir papéis centrais no desenvolvimento de sistemas computacionais, ao serem considerados não somente documentação mas abstrações (semi-)formais essenciais que guiam a execução das atividades do processo (propósito prescritivo).

Um dos objetivos da utilização de modelos é a derivação de representações semanticamente ricas, com constructos mais próximos do domínio de aplicação em questão e que viabilizem o reuso de *frameworks* e componentes pré-disponibilizados para alavancar a produtividade do processo e a qualidade do produto. Uma série de abordagens, publicadas sob diferentes acrônimos, foram propostas nos últimos anos [47].

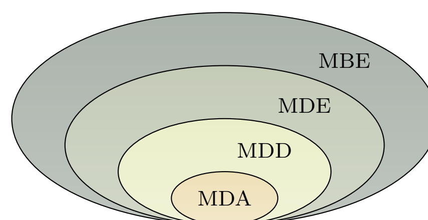


Figura 2.3.: Acrônimos para diferentes abordagens de uso de modelos de *software* (adaptada de [47]).

Conforme ilustrado na Figura 2.3, *Model-Driven (Software) Development* (MDD) é um paradigma genérico de desenvolvimento de *software* baseado na utilização de modelos como artefato-guia para a geração dos elementos que irão implementar o sistema. A *Model-Driven Architecture* (MDA) [170] é uma forma particular de MDD, proposta pelo *Object Management Group* (OMG) e que baseia-se na utilização de notações padronizadas pelo mesmo grupo.

A *Model-Driven (Software) Engineering* (MDE), por outro lado, estende a utilização de modelos para outras atividades além da implementação, tais como testes e evolução. No MDD, MDA e MDE, os modelos exercem o papel de artefato central, responsável por guiar a execução das atividades do processo. Abordagens mais relaxadas, onde a presença de modelos de *software* é importante porém não-central, são classificadas como parte da *Model-Based (Software) Engineering* (MBE).

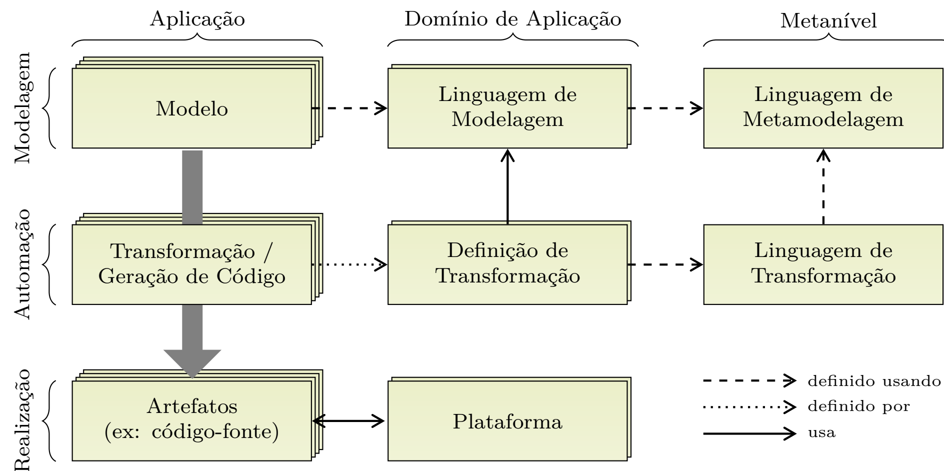


Figura 2.4.: Visão geral da MDE (adaptada de [47]).

Conforme apresentado na Figura 2.4, a MDE trabalha com artefatos e operações definidos em duas dimensões ortogonais: implementação (linhas na Figura 2.4) e conceitualização (colunas na Figura 2.4). A dimensão de implementação lida com o mapeamento/instanciação de modelos em representações operacionais (ex: artefatos de implementação), enquanto a dimensão de conceitualização busca a definição de modelos conceituais (metamodelos) que representem a realidade sendo descrita.

A dimensão de implementação lida com artefatos em três níveis: *i)* modelagem, *ii)* realização e *iii)* automação. Artefatos de modelagem são modelos abstratos e compactos que descrevem a estrutura e comportamento do sistema. Artefatos de realização, por sua vez, são as soluções (códigos-fonte, arquivos de dados, etc) que efetivamente implementam a aplicação. Os artefatos de automação são responsáveis por executar os mapeamentos entre artefatos de modelagem e artefatos de realização.

A dimensão de conceitualização utiliza artefatos de três categorias: *i)* aplicação, *ii)* domínio de aplicação e *iii)* metanível. Artefatos de aplicação são representações específicas de um sistema pertencente ao domínio em questão, tais como os modelos, execuções de transformações e implementações deste sistema. Artefatos de domínio de aplicação são responsáveis pela especificação de linguagens (metamodelos), transformações e plataformas/ambientes de execução; todos voltados para um determinado domínio de aplicação. Já os artefatos de metanível definem as linguagens utilizadas para a especificação dos artefatos deste domínio de aplicação (meta-metamodelos).

Uma **linguagem de modelagem** define os constructos disponíveis para a criação de modelos e é formada por três elementos básicos:

- Sintaxe abstrata: descreve os constructos básicos, suas propriedades e as possibilidades de combinação de constructos para a formação de estruturas maiores.
- Sintaxe concreta: define representações específicas – textuais ou gráficas – que viabilizam a codificação e a melhor compreensão da linguagem. Se a sintaxe concreta é gráfica então o resultado do processo de modelagem é a geração de um ou mais diagramas. Sintaxes concretas definem uma "interface" de utilização da linguagem pelo modelador.
- Semântica: define o significado dos constructos básicos da linguagem e das possíveis combinações de constructos.

Quando os constructos definidos na linguagem representam abstrações semanticamente ricas e próximas de um determinado domínio de aplicação, classifica-se a linguagem como uma Linguagem de Domínio Específico (DSL: *Domain-Specific Language*). Se os constructos permitem a modelagem de sistemas de diversos domínios de aplicação, diz-se que a linguagem é uma Linguagem de Propósito Geral (GPL: *General-Purpose Language*).

Vale ressaltar que, em algumas situações, a classificação de uma linguagem como DSL ou GPL não é uma atividade determinística e bem definida, devido a diferentes perspectivas sobre quão amplo ou restrito é o domínio de referência sendo utilizado. Por exemplo, a UML (*Unified Modeling Language*) [232] pode ser considerada como GPL, em função da sua adequação à modelagem de sistemas de *software* em diversos domínios. Por outro lado, considerar como restrição o direcionamento da UML para a modelagem de sistemas de *software* (e, eventualmente, processos de negócio) implica na sua categorização como DSL.

A adoção de DSLs bem projetadas é fator primordial para alavancar os benefícios da utilização de modelos em sistemas de um domínio específico. A adequação direta aos requisitos do domínio, associada a uma maior expressividade dos constructos e riqueza semântica das estruturas, gera melhorias na produtividade e na qualidade do *software* gerado.

Projetar DSLs efetivas, entretanto, não é uma tarefa fácil [167]. A escolha de abstrações intuitivas e ao mesmo tempo expressivas, a definição de mecanismos para extensão da linguagem e a disponibilização de ferramentas de suporte demandam um amplo conhecimento do domínio em questão, bem como experiência prévia em atividades de modelagem e metamodelagem. Adicionalmente, decisões devem ser tomadas em relação ao grau de especialização da linguagem (horizontal vs vertical), estilo de especificação (imperativo vs declarativo), notação (textual vs gráfica) e modo de execução (interpretação vs geração de código).

O projeto de novas linguagens de modelagem é geralmente realizado através da construção de **metamodelos**, em uma atividade denominada **metamodelagem** [47, 287].

Definição 2.11: Metamodelo (Stahl et al., 2006, pág. 85)

Um metamodelo é um modelo que define os constructos de uma linguagem de modelagem (sintaxe abstrata), os possíveis relacionamentos entre estes constructos, suas restrições de formação e regras de modelagem associados (semântica estática).

Vale ressaltar que metamodelos não definem sintaxes concretas, visto que estas são elementos conceitualmente irrelevantes. Todas as operações de verificação e transformação de modelos são feitas com base na sintaxe abstrata. Em contrapartida, tal separação permite a definição de diferentes sintaxes concretas para um mesmo metamodelo, com diferentes graus de legibilidade e facilidade de uso.

Todo modelo válido é uma instância de algum metamodelo, que define os constructos (linguagem) utilizados na descrição deste modelo. Metamodelos são também descritos em alguma linguagem de metamodelagem (meta-metamodelo), geralmente mais abstrata e menos expressiva que a linguagem definida pelo metamodelo em questão. Este caráter "recursivo" continua até que a

linguagem de metamodelagem seja abstrata o suficiente para ser descrita nela mesma (linguagem auto-reflexiva), não requerendo, portanto, mais níveis de metamodelagem.

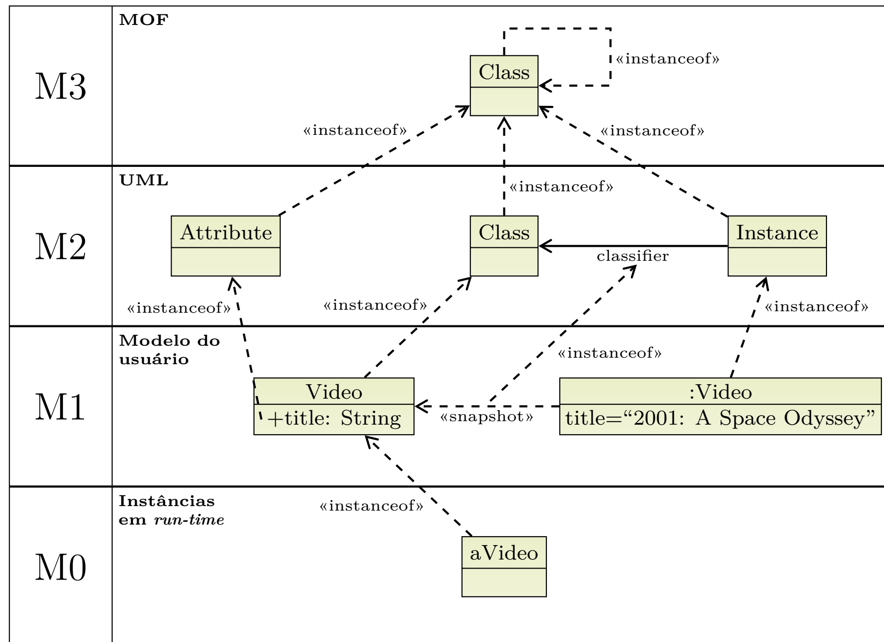


Figura 2.5.: Hierarquia de metamodelagem definida na MDA (adaptada de [232]).

Conforme citado anteriormente, a MDA (*Model-Driven Architecture*) [170] é uma forma específica de MDD padronizada pelo *Object Management Group*, que define suas próprias linguagens de modelagem, níveis de metamodelagem e mecanismos para armazenamento, verificação e transformação de modelos. A Figura 2.5 apresenta a hierarquia de metamodelagem definida na MDA.

O nível M0 é formado por instâncias reais (ex: `aVideo`) dos elementos que compõem o modelo do sistema (ex: `Video`), implementadas em alguma plataforma computacional específica. O nível M1 contém os modelos de sistema criados pelo usuário, com o objetivo de representar a estrutura e comportamento da aplicação. Tais modelos são descritos através da instanciação de constructos do nível M2, onde é definida a linguagem/metamodelo utilizada na criação dos modelos do nível M1. Este metamodelo pode ser mais ou menos expressivo, dependendo de quão amplo (horizontal) é o domínio modelado pela linguagem e quão próximos deste domínio estão os constructos definidos pelo metamodelo.

O (meta)modelo do nível M2, por sua vez, é descrito através da instanciação de constructos do nível M3, onde é definida o (meta-)metamodelo utilizado na criação dos (meta)modelos do nível M2. Por fim, a linguagem definida no nível M3 é abstrata e compacta o suficiente para ser auto-reflexiva e, portanto, definida através da instanciação de constructos do mesmo nível de metamodelagem.

A linguagem auto-reflexiva utilizada na MDA para a especificação de linguagens (metamodelos) é a MOF (*Meta Object Facility*) [231]. A MOF é uma linguagem compacta, formada por 53 metaclasses (apenas 37 delas são concretas) que definem constructos para modelagem de alguns classificadores (apenas classes e *data types*), elementos estruturais e comportamentais simples (ex: propriedades e operações) e relacionamentos básicos tais como associações e herança. A MOF tem como objetivo a definição de um núcleo semântico abstrato, compacto e auto-reflexivo, que atua como linguagem de metamodelagem na construção de linguagens mais expressivas e mais próximas de um determinado domínio de aplicação.

No contexto da MDA, a MOF é uma tecnologia importante para generalizar os mecanismos de

verificação e transformação de modelos para outras linguagens além da UML, tais como o CWM (*Common Warehouse Metamodel*) [118], SPEM (*Software Process Engineering Metamodel*) [230] e ODM (*Ontology Definition Metamodel*) [225]. O fato de terem seus metamodelos descritos em MOF viabiliza a utilização direta dos mecanismos da MDA para armazenar, verificar e transformar modelos descritos nestas linguagens.

A UML é uma linguagem utilizada para análise, projeto e implementação de sistemas de *software* e processos de negócio. Seu metamodelo é complexo (242 metaclasses, onde 192 destas são concretas) e define constructos para representação de estrutura e comportamento de aplicações, bem como máquinas de estado, atividades e interações. Adicionalmente, a UML é frequentemente caracterizada como uma MDL (*Many-Domains Language*), em função dos mecanismos de extensão (*profiles* e *stereotypes*) que permitem a sua adaptação a domínios específicos. Sob esta perspectiva, a UML pode ser vista também como uma plataforma flexível para definição de DSLs.

Decidir se uma nova DSL deve ser construída com base na especialização da UML ou como um novo metamodelo diretamente descrito em MOF requer a consideração de diversos fatores. Se a intersecção entre os constructos da nova DSL e da UML possuir alta cardinalidade, então projetar a DSL com base nas extensões disponibilizadas pela UML ajuda a reduzir custos e prazos. Por outro lado, os *profiles* e *stereotypes* da UML não constituem mecanismos de extensão de primeira classe pois se limitam à adição de novas restrições e propriedades às metaclasses já existentes. Se o projeto da DSL envolver a remoção de restrições/metaclasses/relacionamentos ou a definição de novas hierarquias de metaclasses então um novo metamodelo, preferencialmente baseado na MOF, deve ser criado.

A MDA define ainda tecnologias específicas para definição de restrições (OCL: *Object Constraint Language* [226]) e para transformação de modelos (QVT: *Query / View / Transformation* [229]). A OCL é uma ferramenta importante para a definição de regras de boa formação em atividades de metamodelagem. Embora a UML seja frequentemente criticada devido à sua complexidade, especificação confusa e dificuldade de interoperabilidade entre ferramentas, ela ainda representa a linguagem de maior adoção na indústria, trazendo reais benefícios quando bem utilizada [31, 32] e adaptada às reais necessidades do projeto.

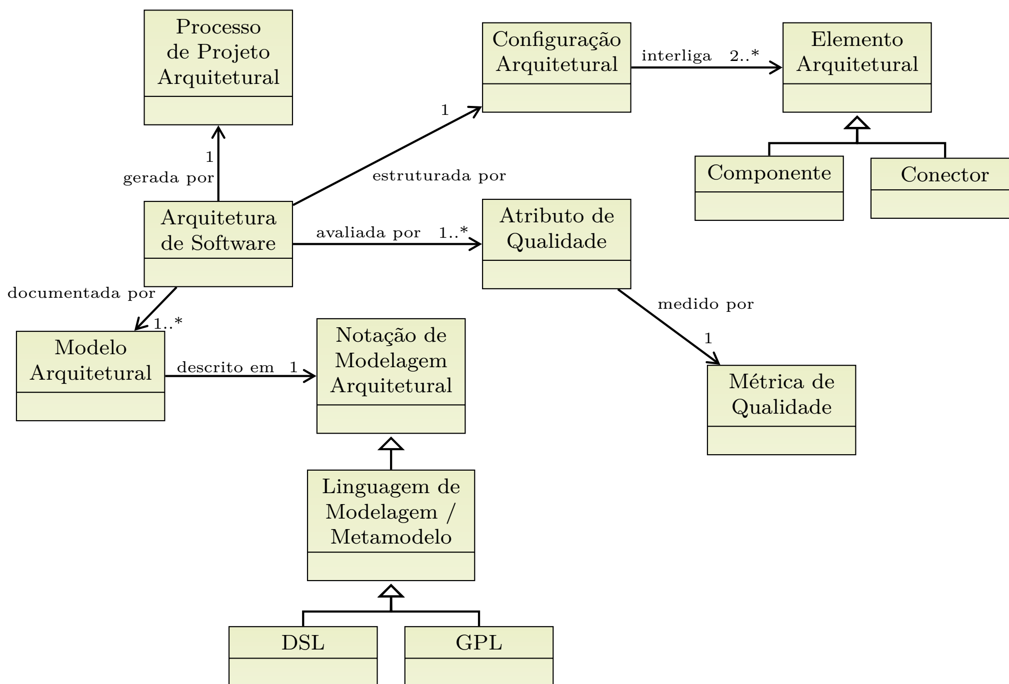


Figura 2.6.: Modelo conceitual das definições apresentadas no capítulo 2.

Não é intenção deste trabalho apresentar os principais constructos da UML e os mecanismos disponibilizados na OCL e QVT; boas referências sobre estes assuntos podem ser encontradas na literatura [24, 102, 314]. A Figura 2.6 apresenta um resumo dos conceitos vistos neste capítulo.

Capítulo 3

Sistemas Self-Adaptive

If you're not confused, you're not paying attention.
Tom Peters (Thriving on Chaos: Handbook for a Management Revolution)

Em 1987, Frederick Brooks – no artigo "*No Silver Bullet - Essence and Accident in Software Engineering*" [49] – afirma que nenhum avanço tecnológico ou gerencial será capaz de trazer, em um prazo de 10 anos, melhorias de uma ordem de magnitude na produtividade, confiabilidade e simplicidade das tarefas de desenvolvimento de *software*. Brooks afirma que os principais avanços da engenharia de *software*, naquela época, foram caracterizados pela disponibilização de mecanismos para redução da complexidade acidental – aquela decorrente da dificuldade de mapeamento das entidades abstratas que caracterizam o *software* em artefatos de implementação. Como exemplos de tais avanços pode-se citar as linguagens de alto nível, a Orientação a Objetos, as técnicas para geração automática de código e os ambientes/ferramentas de desenvolvimento.

Brooks afirma ainda que reduzir a complexidade acidental a zero não trará nenhum avanço significativo, visto que o maior obstáculo será sempre a complexidade essencial – decorrente da especificação, projeto e teste das entidades conceituais abstratas (conjuntos de dados, algoritmos e colaborações) que inerentemente caracterizam o sistema. Esta complexidade essencial é considerada um fator irredutível, visto que sua simplificação implica numa descaracterização da real natureza do *software*. Algumas tecnologias para gerenciamento da complexidade essencial são apresentadas por Brooks, tais como a substituição do desenvolvimento pela utilização de algum produto COTS (*Commercial Off-the-Shelf*); a execução incremental de levantamento de requisitos e implementação; e o incentivo à identificação, documentação e disseminação de projetos de qualidade excepcionalmente alta.

De 1987 até os dias atuais, uma série de novas demandas [141, 224, 267] contribuíram para aumentar ainda mais a complexidade essencial de sistemas de *software*. A disponibilização de computadores com melhor desempenho e menor custo, redes de comunicação mais velozes e dispositivos com maior capacidade de armazenamento de dados traz, a cada dia, novas oportunidades e desafios ao uso de *software* como elemento estratégico central de negócio. Sistemas altamente distribuídos, heterogêneos e com demandas rigorosas por escalabilidade (*ultra-large scale*), baixo consumo de energia, *dependability*, facilidade de integração, flexibilidade e autogerenciamento na presença de ambientes incertos têm sido o foco de muitas pesquisas atuais. O objetivo é propor soluções de engenharia de *software* que gerenciem a complexidade essencial e reduzam a complexidade acidental em tais cenários.

Mesmo utilizando tecnologias poderosas para gerência de complexidade, acredita-se que a capacidade humana de compreensão e manipulação de artefatos de *software* implicará, em poucos

anos, em limites na escalabilidade de aplicação destas tecnologias [145]. A complexidade essencial continuará crescendo até o ponto em que os profissionais mais habilidosos e as tecnologias mais efetivas não serão suficientes para o desenvolvimento de produtos de qualidade. Uma possível solução – investigada por uma série de pesquisas recentes – é a transferência de certas atividades de processo para o próprio *software*, dotando-o com alguma capacidade de autogerenciamento ou auto-adaptação[1] [145, 168, 266].

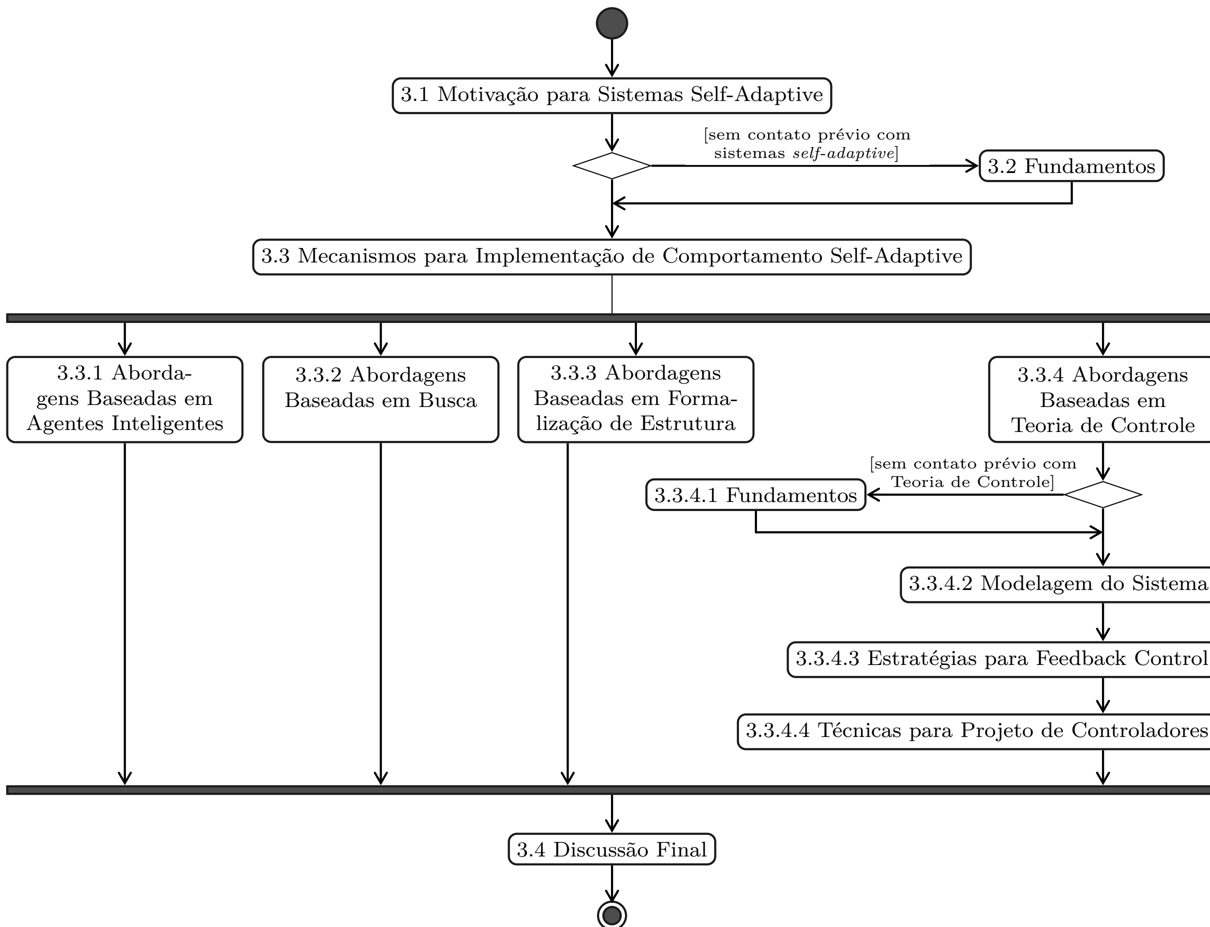


Figura 3.1.: Roteiro do capítulo 3.

Este capítulo apresenta os sistemas *self-adaptive* [266] como abordagem promissora para o gerenciamento, em alta escala, da complexidade essencial de sistemas de *software*. Inicialmente, os motivos para a construção de aplicações com possibilidades de autogerenciamento são apresentados, seguidos dos fundamentos sobre o que significa ser *self-adaptive* e das principais dimensões (graus de liberdade) presentes no espaço de problema (*problem space*) e espaço de solução (*solution space*) deste domínio de aplicação. A Figura 3.1 apresenta o roteiro deste capítulo.

3.1. Motivação para Sistemas Self-Adaptive

Embora a construção de sistemas com capacidades de auto-adaptação seja uma estratégia utilizada para diferentes propósitos, todos são caracterizados pela necessidade de que certas decisões – anteriormente tomadas *off-line* pelo programador/projetista – passem a ser tomadas em tempo de execução pela própria aplicação. Possíveis inflexibilidades (e consequente ineficiências) da solução são causadas pelo comprometimento antecipado com uma decisão que não é satisfatória

[1] Autogerenciamento e auto-adaptação são utilizados como sinônimos nesta tese para representar o comportamento que caracteriza os sistemas *self-adaptive*.

em situações não previstas pelo arquiteto. A prevalência de tais situações não previstas depende diretamente do dinamismo do ambiente de execução e dos casos de uso da aplicação. Dessa forma, assumir menos premissas sobre a homogeneidade das plataformas constituintes, disponibilidade dos serviços e possibilidades de evolução das arquiteturas parece uma demanda imperativa. Encontrar um bom equilíbrio entre "decidir tudo *off-line*" e a visão utópica de "*software* auto-construtivo" é um ponto central no projeto de sistemas *self-adaptive*.

É este o caso, por exemplo, dos algoritmos adaptativos, utilizados quando a adoção de um algoritmo não-adaptativo específico somente garante eficiência (de tempo e/ou espaço) para um conjunto particular de dados de entrada. Ao adiar a derivação de valores de parâmetros para o tempo de execução – realizada de forma automática – os algoritmos adaptativos viabilizam a definição de mais graus de liberdade, trazendo um caráter multialgorítmico que estende a sua eficiência para uma faixa mais ampla de situações.

Um outro exemplo refere-se às operações de sintonia (*tuning*) de bancos de dados. O comprometimento antecipado com planos de execução específicos e políticas particulares para *caching* e paralelismo, por exemplo, pode fazer com que a eficiência seja degradada caso os padrões de acesso e/ou natureza dos dados armazenados mudem. Mesmo em situações menos dinâmicas, um alto número de parâmetros de sintonia com interações complicadas pode inviabilizar a obtenção de configurações ótimas. Definir mecanismos para que o próprio banco de dados monitore as condições atuais e replaneje/reconfigure o seu comportamento é uma estratégia que ataca os dois problemas de comprometimento antecipado acima descritos.

De um modo geral, a construção de aplicações *self-adaptive* é motivada pela presença de um ou mais dos seguintes fatores:

- Alta complexidade do espaço de problema e espaço de solução: cenários que só admitem soluções com complexidade exponencial ou onde a dificuldade das regras de negócio (espaço de problema) ou implementação e configuração das soluções (espaço de solução) já atingiram os limites humanos de gerência. Como exemplos, pode-se citar a utilização de aprendizado por reforço (*reinforcement learning*) para otimizar a alocação de recursos em *data centers*, alocação de processos de tempo-real e roteamento de requisições em serviços de balanceamento de carga [298]. Outros exemplos incluem a derivação de esquemas complexos de implantação e mobilidade de arquiteturas [207] e o uso de técnicas de otimização para alocação de aplicações em ambientes de *cloud computing* [236].

- Dinamismo nos dados: caracterizado por mudanças constantes na natureza dos dados sendo processados, com a consequente degradação do serviço caso este tenha sido projetado para um tipo particular de perfil. Como exemplo, o desempenho do processamento de *big data* via arquiteturas *MapReduce* [76] no *Hadoop* [320] é diretamente influenciado por 190 parâmetros de configuração e por mais de 10 técnicas diferentes de execução de *joins*. Configurações ótimas de parâmetros e a seleção da técnica de *join* mais eficiente dependem fortemente da natureza dos dados, do processamento sendo realizado e das características do *cluster*. A utilização de abordagens adaptativas, nestes cenários, tem trazido resultados promissores [139, 140].

- Dinamismo nas demandas por serviço: situações onde o provisionamento via análise de desempenho considerando demandas estáticas frequentemente implica em altos custos ou em (sub/super)dimensionamento dos recursos necessários. É o caso, por exemplo, das aplicações de comércio eletrônico – onde picos sazonais de acesso geralmente degradam a disponibilidade do sistema – e dos serviços de hospedagem multi-inquilino (*hosting multi-tenant*), onde diferentes níveis de QoS (*Quality of Service*) devem ser garantidos para clientes com diferentes perfis. A implementação de mecanismos adaptativos (elásticos) para armazenamento e processamento em ambientes de *cloud computing* é uma das formas de amenizar tais problemas [195, 196, 305].

- Dinamismo no ambiente de execução: situações onde, por exemplo, requisitos intencionalmente parciais ou incertos, falhas de *hardware*, sobrecargas na rede de comunicação, mobilidade e alta heterogeneidade podem inviabilizar uma decisão de projeto tomada *off-line*. Em tais ambientes dinâmicos, a utilização de uma infraestrutura em tempo de execução para a monitoração, planejamento e reconfiguração contínuos da aplicação é de fundamental importância para a obtenção dos atributos de qualidade desejados. Como exemplos, pode-se citar as redes de sensores [281, 303], a utilização de modelos para obtenção de *dependability* [107], a descoberta e seleção dinâmica de elementos estruturais em arquiteturas orientadas a serviços [82], adaptação em sistemas ubíquos e pervasivos [160] e os mecanismos para otimização de consumo de energia em *data centers* [137].
- *Self-adaptiveness* inerente: cenários onde ou o próprio problema sendo resolvido ou a estrutura funcional do *software* já possuem natureza adaptativa. Nestes casos, mecanismos *self-adaptive* são utilizados não como meios para atendimento de requisitos não-funcionais, mas sim como soluções semanticamente adequadas à implementação das regras de negócio do sistema. O principal representante deste grupo são os sistemas que exibem algum comportamento emergente em função de uma cooperação autônoma e descentralizada de agentes de *software* (sistemas *self-organizing*). Como exemplos, pode-se citar os sistemas multiagentes [97], os *grids* geograficamente dispersos e sob diferentes fronteiras administrativas [205], abordagens bioinspiradas para redes de computadores [57], redes auto-organizáveis de sensores [198] e exploração espacial através de micronaves auto-organizáveis [300].

As motivações acima apresentadas são atualmente discutidas e investigadas, sob diferentes perspectivas, em comunidades de pesquisa dedicadas a tolerância a falhas, computação distribuída, computação bioinspirada, inteligência artificial distribuída, robótica e sistemas de controle, dentre outras. Influências de outras áreas, como a biologia, química e sociologia são também frequentemente encontradas em pesquisas relacionadas a sistemas *self-adaptive*. As motivações, abordagens existentes e consequências são amplas, caracterizando um domínio de aplicação relativamente indisciplinado e que demanda a definição de estratégias sistemáticas gerais para a aplicação deste conhecimento em projetos de sistemas *self-adaptive*.

3.2. Fundamentos

Diferentes definições de sistemas de *self-adaptive*, associadas a taxonomias baseadas em diferentes critérios, podem ser encontradas na literatura [18, 62, 145, 193, 266]. Definir as fronteiras com outras abordagens, identificar as principais dimensões de modelagem e apresentar diretrizes para um levantamento efetivo de requisitos de adaptação não é uma tarefa fácil. Uma definição amplamente utilizada de **sistema self-adaptive** é aquela proposta pela DARPA (*Defense Advanced Research Projects Agency*), em 1997 [22]:

Definição 3.1: Sistema Self-Adaptive (DARPA, 1997)

Um sistema *self-adaptive* é aquele que avalia o seu próprio comportamento e o modifica quando a avaliação indica que: *i)* o seu propósito principal não está sendo efetivamente cumprido; ou *ii)* uma melhor funcionalidade e/ou desempenho pode ser alcançado.

Embora a definição acima compreenda, de certa forma, as motivações apresentadas na Seção 3.1, uma análise mais detalhada dos mecanismos viabilizadores de *self-adaptation* faz-se necessária. Para isso, Salehie & Tahvildari [266] utilizam a abordagem dos "*six honest serving men*"[2] para estender as questões de levantamento de requisitos de *self-adaptation* inicialmente propostas por Laddaga [184]:

[2]Seis questões (*what*, *where*, *who*, *when*, *why* e *how*) presentes no poema "*Six Honest Men*", de R. Kipling.

- Onde (está a causa da adaptação) ? Conjunto de questões relacionadas à localização da parte do sistema (ou do ambiente) que causou o problema que, por sua vez, será resolvido através de mecanismos *self-adaptive*. Geralmente envolve a identificação dos artefatos causadores, a(s) camada(s) na(s) qual(is) eles residem e seus níveis de granularidade. Para isso, é necessário a coleta contínua de certos atributos estruturais e operacionais do sistema, eventualmente em diferentes níveis de abstração.
- Quando (adaptar) ? Questões que investigam os momentos onde a adaptação é necessária e viável. Sistemas altamente dinâmicos geralmente possuem estados transientes, onde a aplicação de mais adaptações pode trazer instabilidades indesejadas. Nestes casos, é necessário a utilização de mecanismos para detecção de estados quiescentes [174], onde adaptações podem ser realizadas de forma segura. *Overheads* causados por adaptações muito frequentes e *trade-offs* (soluções de compromisso) entre abordagens reativas e proativas são também tratados nestas questões.
- O quê (adaptar) ? Conjunto de questões que identificam os atributos e/ou artefatos a serem modificados pelas ações de adaptação. Os elementos modificados podem variar desde novos valores de parâmetros até a substituição/adição de componentes, modificação de estruturas arquiteturais e provisionamento de novos recursos. Geralmente requer que conhecimento de projeto e de análise de desempenho estejam representados e disponibilizados em tempo de execução.
- Por quê (motivação para adaptação) ? Questões relacionadas com as metas cujo alcance é facilitado através do uso de mecanismos *self-adaptive*. Embora algumas motivações já tenham sido apresentadas na Seção 3.1, um tópico comumente presente nas pesquisas atuais é a prospecção de mecanismos para mapeamento de metas operacionais (ex: tempos de resposta ou utilização de CPU) – geralmente numerosas e difíceis de derivar – em objetivos de mais alto nível, mais fáceis de serem afirmados e gerenciados [304, 293].
- Quem (realizará a adaptação) ? Questões que abordam o grau de automação e envolvimento humano que haverá nas adaptações realizadas no sistema. Embora, em grande parte dos casos, a meta seja o projeto de sistemas autogerenciáveis com o mínimo de intervenção humana, alguns trabalhos justificam a importância e propõem soluções para a inserção do usuário nas ações de adaptação [93].
- Como (realizar a adaptação) ? Conjunto importante de questões que investigam os meios pelos quais os artefatos serão modificados em tempo de execução. Geralmente envolvem a definição de ações elementares de adaptação, ordem de execução destas ações e os custos e efeitos colaterais da execução destas modificações.

Desenvolvimentos dos seis aspectos apresentados acima podem ser encontrados nos trabalhos de Andersson et al. [18], Oreizy et al. [234, 235], Salehie & Tahvildari [266] e Brun et al. [50]. Embora estes trabalhos contribuam para um melhor conhecimento das características do domínio e possibilidades de projeto de sistemas *self-adaptive*, a não distinção entre dimensões relacionadas a diferentes espaços torna mais difícil o gerenciamento do conhecimento de projeto arquitetural de tais sistemas.

Para isto, esta tese classifica as dimensões de projeto de sistemas *self-adaptive* em três diferentes espaços:

- **Espaço de problema** (*problem space*): contém dimensões relacionadas aos requisitos de adaptação a serem atendidos pelo projeto. Geralmente captura respostas para as questões “Onde”, “O quê” e “Por quê”.
- **Espaço de solução** (*solution space*): contém dimensões relacionadas às estratégias de projeto utilizadas para atendimento dos requisitos levantados no espaço de problema. Estas dimensões são independentes das tecnologias que serão utilizadas na sua realização. Geralmente envolve a captura de respostas para as questões “Quando”, “Quem” e “Como”.

- **Espaço de realização** (*realization space*): contém dimensões que definem as tecnologias, soluções de *middleware* e algoritmos a serem utilizados na implementação dos mecanismos descritos no espaço de solução. Envolve respostas para as mesmas questões do espaço de solução, porém considerando aspectos de implementação.

As seções a seguir apresentam detalhes sobre possíveis dimensões do espaço de problema e espaço de solução envolvidos no projeto de sistemas *self-adaptive*. Visto que o foco deste trabalho é a definição de mecanismos para captura e utilização de conhecimento de projeto, aspectos do espaço de realização não serão aqui discutidos. As dimensões apresentadas a seguir foram derivadas a partir da análise, comparação e classificação das informações presentes em [18, 50, 234, 235, 266] e em um amplo conjunto de trabalhos primários deste domínio.

Ao mesmo tempo em que não tem-se a intenção de apresentar uma lista completa das dimensões do domínio de sistemas *self-adaptive*, acredita-se que os espaços apresentados a seguir contribuem para uma melhor compreensão da problemática e das possibilidades de engenharia de sistemas *self-adaptive*. A riqueza (e consequente complexidade) deste domínio de aplicação é caso legítimo para justificar a necessidade de prospecção de mecanismos para representação e aplicação sistemática de conhecimento de projeto arquitetural de domínio específico.

3.2.1. Espaço de Problema

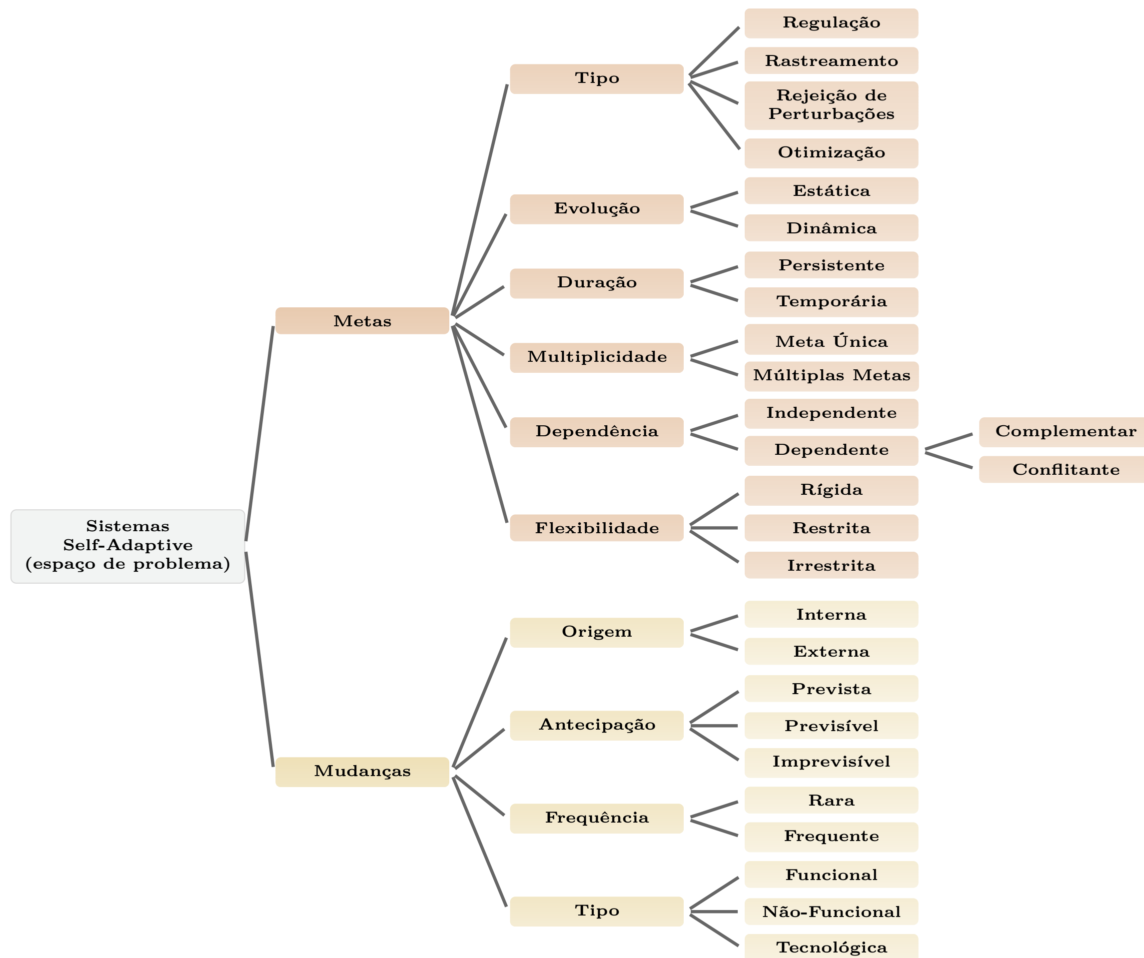


Figura 3.2.: Dimensões do espaço de problema de sistemas *self-adaptive*.

Conforme mencionado anteriormente, o espaço de problema captura decisões relacionadas aos requisitos de adaptação que o sistema deve atender, de modo a ter suas metas de qualidade

atingidas e mantidas ao longo da sua operação. A figura 3.2 apresenta as dimensões mais importantes deste espaço. Os requisitos são categorizados em dois grandes grupos: metas e mudanças.

As dimensões do grupo de metas definem a forma de caracterização dos objetivos a serem alcançados pelo sistema, utilizando, para este fim, a implementação de comportamento *self-adaptive*. As metas podem representar quatro tipos diferentes de objetivos: regulação, rastreamento, rejeição de perturbações e otimização. As metas de regulação têm como objetivo garantir que uma determinada saída (*output*) medida (ex: utilização de CPU) seja igual (ou suficientemente próximo) a um valor de referência (ex: 66% para acomodar a falha de um entre três servidores). Este valor de referência pode mudar ao longo do tempo, daí o nome de meta de regulação. Se este valor de referência muda com muita frequência, caracteriza-se então a meta como de rastreamento. Outro objetivo das metas pode ser não mais regular o valor de referência, mas garantir que perturbações atuando no sistema (ex: tarefas esporádicas ou variações na demanda) não afetem a saída medida (rejeição de perturbações). Por fim, a necessidade de obtenção contínua de valores ótimos para uma determinada saída (ex: minimizar o tempo de resposta) pode ser definida através das metas de otimização.

As metas podem ser também classificadas como estáticas (fixas em tempo de projeto) ou dinâmicas (passíveis de modificação, assistida ou automática, em tempo de execução). Podem ser almejadas durante toda a execução da aplicação (persistentes) ou somente em estados particulares predefinidos (temporárias). Um sistema *self-adaptive* pode trabalhar com uma única meta ou com múltiplas metas. Neste último caso, cada duas metas podem ser classificadas como independentes, mutualmente indutoras (dependentes complementares) ou conflitantes. As metas podem ainda ser classificadas, quanto à criticidade do seu alcance, em: rígidas (obrigatórias), restritas (obrigatórias desde que certas condições estejam presentes) ou irrestritas (opcionais).

As dimensões do grupo de mudanças definem informações sobre as possíveis perturbações presentes no sistema e/ou no ambiente, que fazem com que a(s) meta(s) deixe(m) de ser alcançada(s) ao longo da execução. Tais perturbações podem ser internas (com origem no próprio sistema, por exemplo, falhas de componentes) ou externas (com origem no ambiente, por exemplo, modificações de demandas por serviço). Quanto à antecipação, podem ser classificadas em: previstas (detectáveis por tratamento particular definido em tempo de projeto), previsíveis (detectáveis por mecanismo genérico em tempo de execução) ou imprevisíveis (não-detectáveis). Podem ser classificas como raras ou frequentes; e ainda como funcionais (mudança no propósito do sistema), não-funcionais (demandas de melhoria, por exemplo, no desempenho ou confiabilidade) ou tecnológicas (por exemplo, atualização da plataforma de *middleware* utilizada).

3.2.2. Espaço de Solução

Embora diferentes mecanismos viabilizadores de comportamento *self-adaptive* tenham sido propostos nos últimos anos, a maioria deles baseia-se na utilização de um ou mais **loops de adaptação**. Estes *loops* são caracterizados pela execução subsequente das quatro atividades básicas apresentadas na Figura 3.3: **monitoramento**, **análise**, **planejamento** e **atuação**. Os componentes que executam tais atividades constituem o **sistema gerenciador** (*managing element*), responsável pela adição de capacidades de autogerenciamento ao sistema que implementa as regras de negócio – o **sistema gerenciado** (*managed element*). Os *loops* de adaptação podem ser implementados de forma externa ao sistema gerenciado (conforme apresentado na figura) ou fortemente acoplado (interno), como parte integrante do sistema que sofrerá as adaptações.

A atividade de monitoramento consiste na obtenção de informações referentes ao estado atual do sistema e do ambiente de execução. Tais informações – geralmente operacionais, de alta cardinalidade e efêmeras – devem subsidiar as operações subsequentes de análise e planejamento,

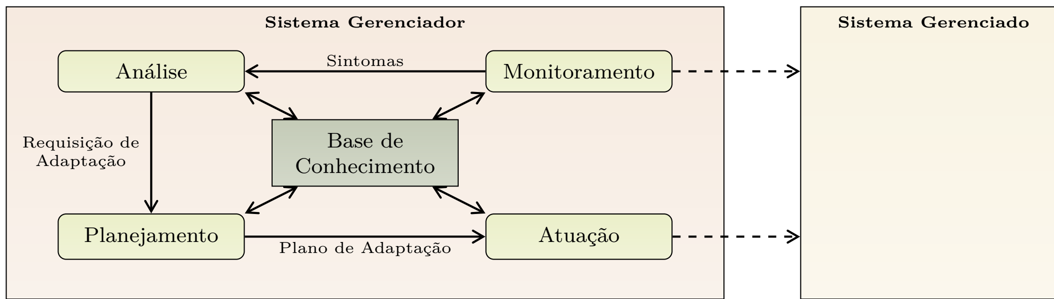


Figura 3.3.: Estrutura básica de um *loop* externo de adaptação (adaptada de [62] e [168]).

com o objetivo de garantir que as metas estabelecidas continuem sendo cumpridas. Como exemplos, pode-se citar a obtenção de taxas de utilização de CPUs, tempo médio de resposta de serviços, taxas de perda de pacotes na rede e perfis de comportamento do usuário. Envolve problemas tais como a decisão sobre a técnica de sensoriamento utilizada, taxas de amostragem e tratamentos de incertezas e informações parciais.

A atividade de análise tem como objetivo investigar as informações produzidas pela atividade de monitoramento e decidir se há necessidade de alguma adaptação e o local onde esta deverá ocorrer. Geralmente envolve a utilização de modelos representativos do sistema e das metas, armazenamento de dados históricos e mecanismos para decisão. Uma série de técnicas podem ser utilizadas para este fim e serão discutidas a seguir.

A atividade de planejamento é responsável pela derivação de um ou mais conjuntos de ações de adaptação que, após executadas, irão supostamente trazer o sistema de volta a um estado de atendimento da(s) meta(s) definida(s). Diferentes cursos de adaptação podem ser obtidos e um deles deverá ser selecionado para execução na próxima atividade. Novamente, diferentes técnicas – apresentadas a seguir, podem ser utilizadas para este propósito.

Finalmente, a atividade de atuação tem como objetivo a efetiva execução – no sistema real – das ações de adaptação definidas na etapa de planejamento. Envolve a seleção dos atuadores a serem utilizados e a detecção de momentos seguros de execução das adaptações, dentre outros aspectos. Diferentes técnicas para execução destas ações serão também discutidas a seguir.

Variações desta estrutura básica de *loops* de adaptação podem ser encontradas na literatura. Em [168], Kephart & Chess definem o MAPE-K, estrutura formada pelas atividades de *Monitor* (M), *Analyze* (A), *Plan* (P) e *Execute* (E), executadas com o auxílio de uma base de conhecimento (K: *Knowledge Base*). Dobson et al. [84] apresentam *loops* autonômicos de controle, com atividades similares, voltados para a área de redes adaptativas de comunicação. Oreizy et al. [234] denominam este *loop* "Gerenciamento de Adaptação", em contraste ao "Gerenciamento de Evolução", executado *off-line*. Alguns autores denominam "Adaptação Estática" a execução *off-line* e assistida destas atividades de adaptação.

Sistemas *self-adaptive* complexos geralmente requerem a utilização de mais de um *loop* de adaptação, preferencialmente separados no espaço (aplicados em diferentes módulos/subsistemas) ou no tempo (presentes no mesmo módulo/subsistema de forma hierárquica, porém com frequências de execução diferentes) [9, 318, 257, 309]. Decidir sobre os possíveis arranjos dos múltiplos *loops* de adaptação presentes e as técnicas particulares utilizadas nas atividades de cada *loop* é uma atividade difícil porém crucial para o projeto de sistemas *self-adaptive* efetivos.

A partir desta perspectiva, pode-se derivar um espaço de solução para sistemas *self-adaptive* a partir da classificação das dimensões de projeto em cinco grupos principais, descritos na Figura 3.4: *loops* de adaptação, monitoramento, análise, planejamento e atuação. As Figuras 3.5, 3.6, 3.7, 3.8 e 3.9 apresentam possíveis dimensões para estes grupos, respectivamente.

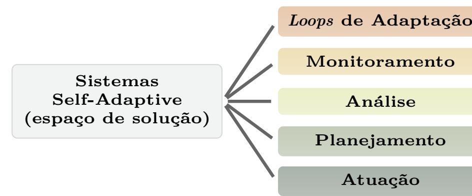


Figura 3.4.: Dimensões do espaço de solução de sistemas *self-adaptive* (1º nível).

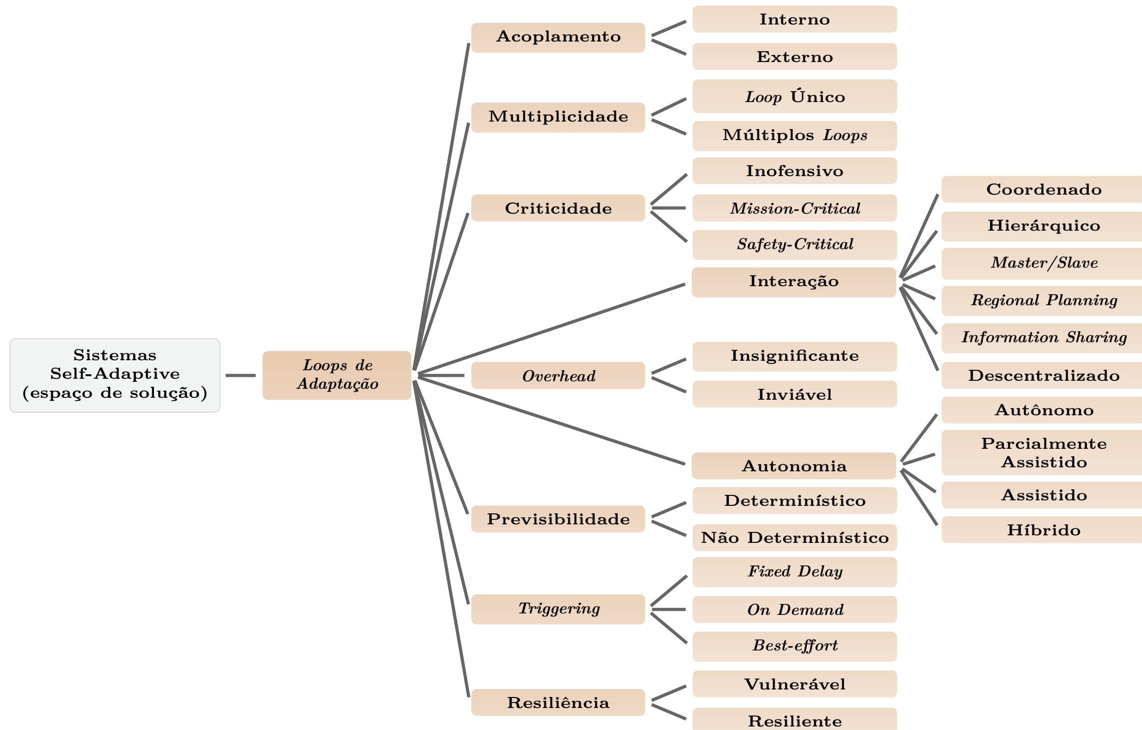


Figura 3.5.: Dimensões do espaço de solução de sistemas *self-adaptive* (*loops* de adaptação).

Loops de adaptação. *Loops* de adaptação são artefatos centrais no projeto de sistemas *self-adaptive*. Diferentes tipos e quantidades de *loops*, atuando em diferentes granularidades, podem ser necessários para atingir o grau de adaptação requerido para satisfação das metas. Decidir as características de cada *loop* que atua no sistema é uma tarefa importante, pois pode impactar diretamente os atributos de qualidade apresentados pelo sistema ou gerar efeitos colaterais não previsíveis. A Figura 3.5 apresenta as principais dimensões de projeto envolvidas na especificação de *loops* de adaptação.

Os *loops* de adaptação podem ser implementados de forma interna (fortemente acoplados ao sistema sendo adaptado) ou externa (como um subsistema independente do sistema sendo adaptado). Abordagens externas viabilizam soluções mais reutilizáveis e mais adequadas a cenários distribuídos. Por outro lado, *loops* internos podem se beneficiar de melhores desempenhos nas atividades de monitoramento e atuação. Conforme mencionado anteriormente, um sistema *self-adaptive* pode adotar apenas um *loop* de adaptação ou, em cenários mais complexos, múltiplos *loops*. A existência de múltiplos *loops* requer a análise criteriosa sobre o local de atuação de cada *loop* e sobre eventuais impactos e interações entre diversos *loops*. Múltiplos *loops* podem ser projetados em diferentes arranjos de interação: coordenado, hierárquico, mestre/escravo (*master/slave*), planejamento regional (*regional planning*), compartilhamento de informação (*information sharing*) e descentralizado. Tais arranjos serão discutidos de forma detalhada no Capítulo 7.

Um *loop* de adaptação pode ser classificado quanto à sua criticidade em: inofensivo (quando

a adaptação por ele realizada é um requisito opcional), *mission-critical* (quando a adaptação por ele realizada é indispensável para o atendimento dos atributos de qualidade desejados) e *safety-critical* (quando a adaptação por ele realizada é mecanismo-chave para a implementação de *dependability*). Outro aspecto importante é o *overhead* introduzido, pela atuação do *loop*, no sistema computacional e rede de comunicação. Quanto a este aspecto, um *loop* pode ter seu *overhead* classificado como insignificante ou inviável (inviabilizador da operação fundamental do *software*). Um *loop* pode ainda operar de forma determinística (produz sempre a mesma atuação para o mesmo cenário de desvio das metas) ou não-determinística (caso contrário).

Embora o projeto de sistemas *self-adaptive* que atuam sem intervenção humana seja o foco de muitas pesquisas recentes, formas mais fracas de autonomia são também consideradas. Um *loop* de adaptação pode operar de forma totalmente autônoma (sem intervenção humana em nenhum momento), parcialmente assistida (intervenção humana somente em algumas situações previstas), assistida (sempre dependente de intervenção humana) ou híbrida (seleção em tempo de execução de uma das três formas anteriores).

A operação do *loop* de adaptação pode ser classificada como: *fixed-delay*, *on demand* ou *best-effort*. Em *loops* com operação *fixed-delay* (*time-triggered*) os ciclos de atividades de monitoramento, análise, planejamento e atuação são continuamente executados, com um intervalo fixo de tempo entre duas execuções consecutivas de ciclo. Já em *loops* com operação *on-demand* (*event-triggered*) a execução de um ciclo é disparada somente quando algum evento específico é percebido pelo sistema gerenciador. Os *loops* com operação *best-effort* (híbrido) são caracterizados pela seleção automática, em tempo de execução, entre *fixed-delay* (em momentos onde há disponibilidade de recursos para esta forma de operação) ou *on-demand* (em situações de sobrecarga do sistema).

Por fim, um *loop* pode ainda ser caracterizado como resiliente ou vulnerável. *Loops* resilientes conseguem cumprir suas metas de adaptação mesmo quando o sistema e/ou ambiente passam a operar em estados diferentes daqueles para os quais o *loop* foi projetado. Tal característica é importante para implementação de comportamento *self-adaptive* em ambientes bastante dinâmicos ou para suportar metas não-previstas. *Loops* vulneráveis são aqueles que degradam seu desempenho no alcance das metas nestes estados não previstos.

Monitoramento. Obter informações sobre o sistema e seu ambiente de execução é fundamental para verificar se as metas estão sendo alcançadas e para subsidiar as operações subsequentes de análise, planejamento e atuação. Este monitoramento pode capturar diferentes tipos de dados e ser executado de formas mais ou menos invasivas. A Figura 3.6 apresenta as principais dimensões de projeto a serem consideradas na especificação de elementos de monitoramento para sistemas *self-adaptive*.

Os elementos de monitoramento podem ser inicialmente classificados quanto ao tipo de informação obtida do sistema/ambiente. O monitoramento é considerado paramétrico quando a informação extraída é um valor escalar que representa alguma variável a ser controlada. Como exemplo, pode-se citar o monitoramento de tempo médio de resposta de um serviço, utilização de CPU, vazão ou taxa média de perda de pacotes na rede. Estes valores podem ser medidos de forma direta, inferidos a partir de outros dados ou agregados a partir de informações mais elementares.

Por outro lado, monitoramentos estruturais têm como objetivo a obtenção de informações sobre a estrutura atual do sistema e/ou do ambiente. Por exemplo, certas adaptações podem requerer a obtenção de informações sobre a presença/ausência de determinados componentes/conectores em arquiteturas dinâmicas, verificar a alocação atual de componentes a nós em sistemas baseados em agentes móveis, ou verificar os pares (*peers*) disponíveis para comunicação em redes de sensores.

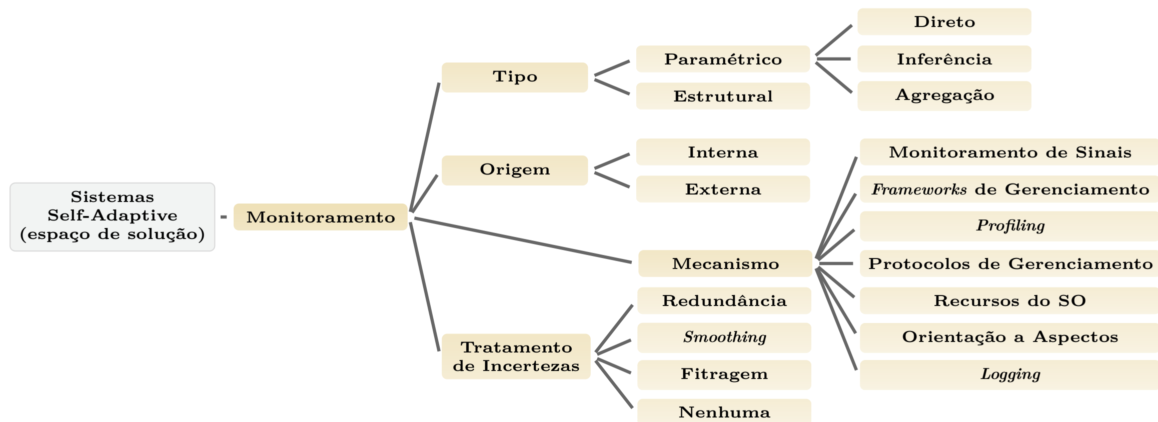


Figura 3.6.: Dimensões do espaço de solução de sistemas *self-adaptive* (monitoramento).

Nestes casos, o dado obtido pelo monitoramento não é um valor escalar, mas uma representação estruturada, por exemplo, sob a forma de um grafo.

Monitoramentos internos têm como objetivo obter informações sobre o sistema, ao passo que os externos realizam observação do ambiente de execução. Uma série de mecanismos, mais ou menos invasivos, podem ser utilizados para implementar o monitoramento. Dentre estes, destacam-se o monitoramento de sinais [290], uso de *frameworks* de gerenciamento (ex: JMX – *Java Management eXtension* [175]), *profiling* (ex: JVMTI – *JVM Tool Interface* [1]) protocolos de gerenciamento [58], recursos do sistema operacional (ex: utilitários *top*, *vmstat* ou acesso ao */proc/* em sistemas *unix-like* [45]), orientação a aspectos [147, 159] e análise de arquivos de *log* [146].

A forte presença de componentes estocásticos nos sinais medidos em sistemas computacionais demanda a utilização de mecanismos para tratamento de ruídos e incertezas [138]. Tais mecanismos incluem, por exemplo, o uso de filtros de média móvel, suavização (*smoothing*) via amostragem ou sensores redundantes.

Análise. A atividade de análise é responsável pela investigação dos dados de monitoramento para decidir se há necessidade de execução de alguma adaptação. Para isso, é geralmente necessário estimar como o sistema e o ambiente irão se comportar na nova situação pronunciada pelos dados de monitoramento. A Figura 3.7 apresenta as principais dimensões de projeto de elementos de análise em sistemas *self-adaptive*.

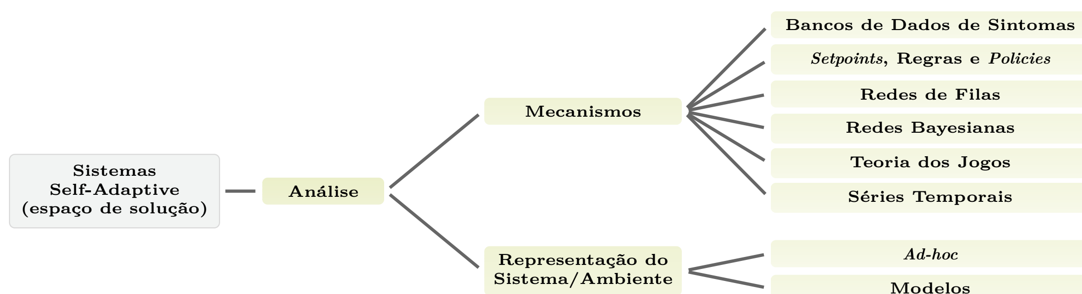


Figura 3.7.: Dimensões do espaço de solução de sistemas *self-adaptive* (análise).

O sistema e o ambiente podem ser representados através de modelos – mantidos consistentes durante o tempo de execução – ou diretamente instrumentados (*ad-hoc*). A detecção de necessidade de adaptação pode ser implementada através de mecanismos diversos. Dentre eles, destacam-se o uso de bancos de dados de sintomas [122, 241]; valores de referência (*setpoints*),

regras e *policies*[3] [2, 169]; redes de filas [71, 313]; redes Bayesianas [7, 265], teoria dos jogos [183, 262] e séries temporais [89, 90]. Vale ressaltar que com a utilização de mecanismos mais simples, como por exemplo *policies* baseadas em regras/*actions*, o elemento de análise geralmente passa a estar fortemente acoplado ao elemento de planejamento. Isto ocorre pelo fato de *policies* serem uma estratégia que resolve tanto o problema de análise quanto o de planejamento. Outras abordagens, entretanto, são utilizadas apenas para detectar se o sistema deixou de atender as metas estabelecidas.

Planejamento. A atividade de planejamento[4] é uma das mais importantes de um *loop* de adaptação. O seu objetivo é a derivação de um plano de adaptação que, quando executado, supostamente trará o sistema de volta ao atendimento das metas predefinidas. Considera-se o sucesso da adaptação como "suposto" em função de possíveis perturbações, geralmente imprevisíveis, que afetam a efetividade das ações de adaptação. Diferentes mecanismos de planejamento assumem diferentes premissas sobre o grau de perturbação presente no sistema, variando desde nenhuma consideração de perturbação (planejamento vulnerável) até mecanismos adaptativos para geração de planos. A Figura 3.8 apresenta as principais dimensões de projeto para os elementos de planejamento.

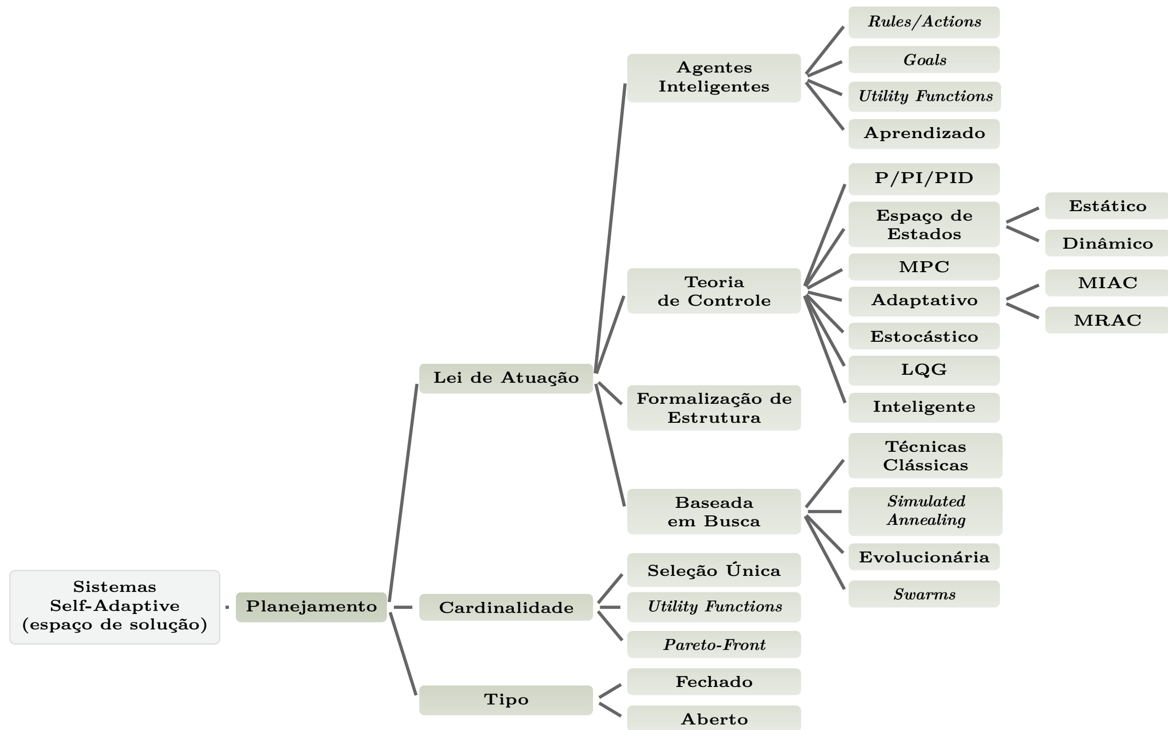


Figura 3.8.: Dimensões do espaço de solução de sistemas *self-adaptive* (planejamento).

O elemento de planejamento pode ser classificado quanto à previsibilidade das mudanças como fechado ou aberto. Planejamento fechado envolve a definição, em tempo de projeto, de conjuntos predefinidos de ações de adaptação e das situações nas quais cada um destes conjuntos deverá ser executado. É uma solução mais simples mas que produz comportamentos *self-adaptive* limitados a situações antecipadamente detectadas. No planejamento aberto, as ações de adaptação são derivadas em tempo de execução, em função do estado atualmente percebido no sistema e no

[3] Regras simples de adaptação, geralmente seguindo o estilo ECA (*Event-Condition-Action*). Ex: a cada 5s (*event*), se a utilização de CPU estiver acima de 75% (*condition*) aumente o tamanho do *buffer* em 50% (*action*).

[4] Termo aqui utilizado no sentido mais geral, sem nenhum relacionamento com *planning* enquanto técnica para implementação de agentes em ambientes com representações fatoradas ou estruturadas [261].

ambiente. Tais soluções geralmente envolvem a utilização de técnicas para tomada de decisão e/ou aprendizado de máquina.

Quando à cardinalidade, os elementos de planejamento podem ser classificados como: solução única, *utility functions* ou *Pareto-front*. No planejamento com solução única, o resultado desta atividade é a geração de um único plano de adaptação, posteriormente executado na atividade de atuação. Entretanto, sistemas *self-adaptive* mais complexos podem suportar a geração de múltiplos planos de adaptação para a mesma meta (apresentando diferentes desempenhos de adaptação). Adicionalmente, sistemas com metas múltiplas e conflitantes geralmente requerem decisão entre múltiplos planos de adaptação, cada um com diferentes graus de atendimento das metas envolvidas.

No planejamento baseado em *utility functions* [169, 261], um único plano é escolhido dentre os disponíveis, baseado no "grau de utilidade" que cada plano apresenta em relação ao alcance das metas. *Utility functions* representam um meio de articulação *a priori* de preferências (redução de muitas soluções para somente uma), sua determinação nem sempre é um trabalho fácil e sua efetividade é altamente afetada por perturbações.

Uma outra opção é a utilização do planejamento com *Pareto-front* [80, 128]. Esta abordagem, considerada como meio de articulação *a posteriori* de preferências, consiste na entrega de múltiplos planos de adaptação. Estes planos são todos eficientes no alcance das metas, porém diferem em relação a qual meta particular (dentre as múltiplas envolvidas) é favorecida na solução. É responsabilidade do usuário escolher qual plano deve ser executado no sistema e, portanto, implica em *loops* com autonomia assistida ou parcialmente assistida.

A derivação do plano de adaptação é efetuada através da aplicação de alguma lei de atuação. Esta dimensão fundamenta a classificação das técnicas de planejamento em quatro grandes escolas: planejamento baseado em agentes inteligentes (adaptação paramétrica ou estrutural), planejamento baseado em busca (adaptação paramétrica ou estrutural), planejamento baseado em formalização de estrutura (geralmente adaptação estrutural) e planejamento baseado em Teoria de Controle (geralmente adaptação paramétrica). Uma série de técnicas diferentes são utilizadas em cada uma destas escolas. Dada a importância dos mecanismos de planejamento para a compreensão deste trabalho, uma apresentação e discussão mais aprofundada destes aspectos são apresentados na Seção 3.3. Em particular, a Seção 3.3.4 apresenta, com mais detalhes, as abordagens baseadas em Teoria de Controle, visto ser esta a escola adotada para investigação nesta tese.

Atuação. A atividade de atuação é responsável pela execução, no sistema real, do plano de adaptação gerado pela atividade de planejamento. Esta atuação pode ser implementada através de diversos mecanismos, mais ou menos invasivos e de maior ou menor impacto no desempenho do sistema. A Figura 3.9 apresenta as dimensões de projeto para a especificação de atuadores em sistemas *self-adaptive*.

Os elementos de atuação podem ser classificados, quanto ao escopo, em locais (quando artefatos são modificados em partes pontuais do sistema) ou globais (quando envolvem a reconfiguração das estruturas gerais da aplicação). Podem ser do tipo paramétrico (quando atribuem novos valores a parâmetros predefinidos de componentes) ou estrutural (quando realizam a remoção, inclusão ou substituição de componentes). Podem ainda ser de fraco impacto (baixo efeito colateral) ou forte impacto (alto efeito colateral). Quando o tempo necessário para a execução da atuação é conhecido e limitado, diz-se que o atuador possui *deadline* garantido. Caso contrário, é classificado como *best-effort*. É informalmente classificado como de duração curta, média ou longa, dependendo do tempo necessário para execução da mudança.

Diferentes mecanismos são utilizados atualmente para a implementação de atuadores. Alguns são mais adequados à implementação de *loops* externos, enquanto outros só podem ser utilizados

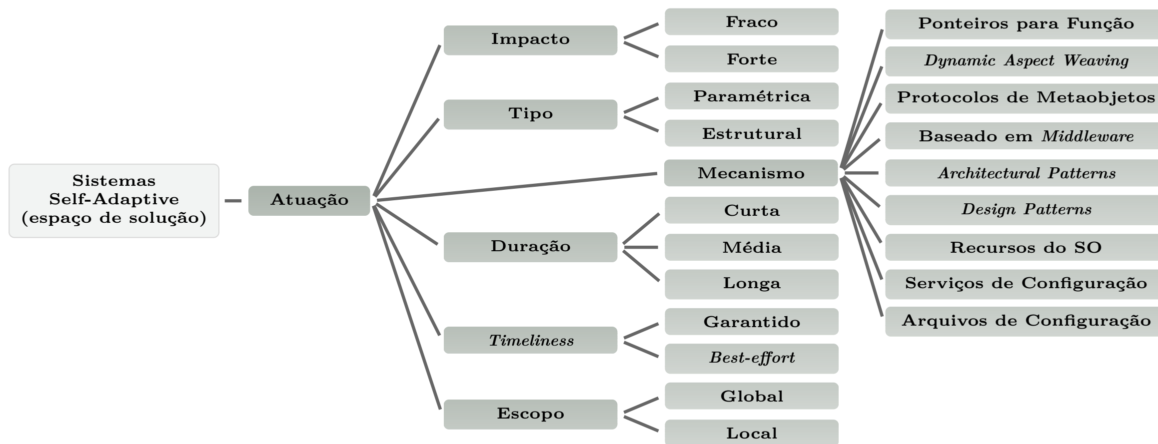


Figura 3.9.: Dimensões do espaço de solução de sistemas *self-adaptive* (atuação).

na implementação de *loops* internos. Como exemplos de mecanismos internos, pode-se citar a substituição de componentes através de ponteiros para função, uso de padrões de projeto (*design patterns*) ou padrões arquiteturais (*architectural patterns*) [297], orientação a aspectos [240] ou protocolos de metaobjetos [264]. Exemplos de mecanismos externos incluem atuação baseada em *middleware* [244, 172], recursos do sistema operacional (ex: alteração de prioridades e afinidades de CPUs), serviços de configuração (acessíveis, por exemplo, via *remote procedure call*) e modificação de arquivos de configuração.

Embora o espaço de solução apresentado acima não esgote as possibilidades de tecnologias e mecanismos utilizados no projeto e implementação de sistemas *self-adaptive*, constitui evidência da riqueza e complexidade deste domínio de aplicação. O projeto de sistemas *self-adaptive* efetivos é geralmente fruto da aplicação de experiência refinada e conhecimento tácito de projeto, geralmente desestruturados ou estruturados de forma não sistemática, o que dificulta a sua disseminação, a sua reutilização no projeto de novas aplicações e a análise de arquiteturas já existentes.

3.3. Mecanismos para Implementação de Comportamento Self-Adaptive

A escolha das abordagens utilizadas na implementação dos *loops* de adaptação é influenciada por fatores tais como: comunidade científica que realiza a pesquisa (ex: redes de computadores, sistemas distribuídos, robótica ou inteligência artificial), tipo de adaptação desejada (paramétrica, estrutural ou híbrida), grau de dinamismo a ser acomodado (totalmente previsto, parcialmente previsto ou totalmente imprevisto), escalabilidade de controle necessária, dentre outros. Sem a intenção de apresentar uma enumeração completa das abordagens utilizadas, as subseções a seguir discutem os mecanismos utilizados nas quatro maiores escolas de projeto de sistemas *self-adaptive*.

3.3.1. Abordagens Baseadas em Agentes Inteligentes

Esta abordagem de implementação de comportamento *self-adaptive* baseia-se na utilização de agentes inteligentes que decidem sobre a necessidade de realização de modificações na estrutura e configuração do sistema. As atividades básicas de um agente se assemelham àquelas apresentadas na Figura 3.3: monitoração do ambiente através de sensores, tomada de decisão de adaptação e execução dessas adaptações no sistema real.

Abordagens mais flexíveis e poderosas modelam o problema como um conjunto de estados de adaptação, representando diferentes graus de atendimento às metas estabelecidas. Este conjunto de estados define o ambiente de operação (*task environment*) no qual os agentes trabalham, com o objetivo de levar o sistema de volta a estados de satisfação das metas desejadas. As ações de adaptação são definidas nas transições de um estado de adaptação para outro[5]. A tomada de decisão de adaptação pode ser implementada através de mecanismos mais simples (e inflexíveis), tais como *policies*, ou mais elaborados, como aprendizado de máquina.

A escolha da técnica utilizada para implementação do agente depende fortemente da flexibilidade e robustez desejada para ele, bem como das características do ambiente de operação. O ambiente de operação é classificado sob diferentes aspectos, apresentados na Tabela 3.1.

Dimensão	**Grau de Liberdade**	**Descrição**
Observability	Totalmente Observável	Os sensores disponibilizam todas as informações do ambiente que são relevantes para a tomada de decisão.
	Parcialmente Observável	Nem todas as informações necessárias estão disponíveis. É necessário utilizar um modelo para derivação das informações não-observáveis.
	Não-observável	Nenhuma informação é disponibilizada sobre o ambiente.
Cardinalidade	Agente Único	Um único agente tenta alcançar a(s) sua(s) meta(s).
	Múltiplos Agentes	Múltiplos agentes tentam alcançar sua(s) meta(s). Podem ser cooperativos ou competitivos.
Transição	Determinístico	O próximo estado é determinado pela estado atual e ação tomada.
	Estocástico	Incertezas sobre as transições são quantificadas através de probabilidades.
	Não-Determinístico	Incertezas sobre as transições são afirmadas, mas sem atribuição de probabilidades.
Dependência	Episódico	Uma transição não depende de nenhuma transição anterior.
	Sequencial	Cada transição afeta todas as transições futuras.
Temporalidade	Estático	O ambiente não muda com o tempo.
	Dinâmico	O ambiente muda com o tempo.
	Semi-dinâmico	O ambiente não muda com o tempo, mas o desempenho do agente muda.
Estados	Atômicos	*Black-box* sem estrutura interna (valor escalar).
	Fatorados	Vetor de atributos.
	Estruturados	Grafo de objetos.
Ambiente	Conhecido	As regras de funcionamento e navegação no ambiente são conhecidas.
	Não-conhecido	O ambiente e seu funcionamento são desconhecidos.

Tabela 3.1.: Dimensões de caracterização de ambientes de operação de agentes inteligentes (derivada de [261]).

Diferentes arquiteturas de agentes produzem soluções com diferentes graus de flexibilidade, reuso em outros problemas de adaptação e efetividade de funcionamento em ambientes de operação estocásticos e/ou não-conhecidos. Escolher a arquitetura mais adequada para o problema em questão e avaliar os benefícios e consequências desta escolha é uma atividade importante no projeto de agentes inteligentes. A Tabela 3.2 apresenta as arquiteturas mais utilizadas na implementação de agentes inteligentes.

Não é objetivo deste trabalho apresentar mais detalhes sobre arquiteturas de agentes inteligentes, visto que a abordagem aqui proposta baseia-se na utilização de Teoria de Controle. Uma visão extensa e geral sobre o assunto pode ser encontrada em [261], enquanto aplicações de agentes inteligentes na implementação de sistemas *self-adaptive* podem ser encontradas em [2, 169, 219].

[5]Não confundir o ambiente formado pelos estados de adaptação com aquele formado por estados de negócio, geralmente utilizado para modelar o comportamento funcional (em contraponto com o comportamento *self-adaptive*) do sistema.

Tipo	Descrição	Vantagens	Desvantagens
Reativo Simples (*rule/action policies*)	Escolhe uma ação a partir da avaliação de sentenças ECA (*Event-Condition-Action*).	Simplicidade.	Requer que o ambiente seja totalmente observável.
Reativo Baseado em Modelos	Mantém um estado interno (histórico de percepções) que reflete algum aspecto não-observável do estado atual.	Ideal quando o ambiente é parcialmente observável.	Não leva metas em consideração. Mudanças nos objetivos implicam na reescrita das sentenças ECA.
Reativo Baseado em Metas (*goal policies*)	O conhecimento que suporta as decisões é explicitamente descrito em uma base de conhecimento (em contraponto a sentenças ECA acopladas ao agente).	Flexibilidade: metas e suporte à decisão podem ser facilmente modificados.	Não implementa recursos para tratamento de metas conflitantes: cada estado é classificado como *happy/unhappy* no atendimento de cada meta.
Reativo Baseado em *Utility Functions*	Viabiliza mecanismo para indicação gradual de atendimento de metas, em contraponto ao modelo tudo-ou-nada das *goal policies*.	Definição de mecanimos *a priori* para articulação de preferências entre metas.	A derivação das *utility functions* pode ser complicada. Requer que o ambiente seja conhecido.
Baseado em Aprendizado	Caracterizado pela utilização de uma máquina genérica de aprendizado.	Alta flexibilidade e possibilidade de operação em ambientes inicialmente não-conhecidos.	Maior complexidade e eventuais ineficiências em certas situações.

Tabela 3.2.: Diferentes tipos (arquiteturas) de agentes inteligentes (derivada de [261]).

3.3.2. Abordagens Baseadas em Busca

O projeto e implementação de sistemas *self-adaptive* através de abordagens baseadas em busca consiste na execução *on-line* (em tempo de execução) de mecanismos até então utilizados *off-line* (em qualquer momento pré-execução) na área conhecida como Engenharia de Software Baseada em Busca (SBSE – *Search-Based Software Engineering*).

A SBSE [132] baseia-se na premissa de que as principais atividades da engenharia de *software* (levantamento de requisitos, projeto, testes, refatoração, etc) são, em sua essência, problemas de otimização. Ao argumentar que a natureza essencial do *software* o faz adequado às abordagens de busca baseadas em otimização [126], a SBSE passa a ser atrativa ao apresentar requisitos mínimos ao seu funcionamento. Tudo o que é necessário é: *i)* definir uma forma de representação do problema como um espaço de busca; e *ii)* especificar a(s) função(ões) de *fitness* utilizada(s) na otimização, frequentemente já presente(s) quando adotado algum tipo de gerência de qualidade [129].

A SBSE tem gerado resultados promissores [73] desde a publicação do primeiro trabalho de Harman & Jones, em 2001 [131]. Revisões abrangentes de literatura podem ser encontradas sobre aplicação da SBSE em problemas de análise de requisitos [326], modelagem [3, 125], propriedades não-funcionais [4], compreensão de programa [123], projeto [253] e testes [10, 124, 212]. Dentre as técnicas mais utilizadas, destacam-se a programação genética, *hill climbing*, *simulated annealing*, otimização multiobjetivo e algoritmos baseados em *swarms* [53, 80].

Uma visão mais detalhada sobre SBSE (e, em particular, projeto arquitetural baseado em busca) é apresentada no Capítulo 8. Esta tese é fortemente baseada na utilização de técnicas da SBSE, não como mecanismo *on-line* de implementação de comportamento *self-adaptive*, mas como estratégia para projeto *off-line* de arquiteturas de sistemas gerenciadores baseados em Teoria de Controle. Tal distinção é importante para a compreensão das contribuições aqui apresentadas.

Quando utilizada, entretanto, como mecanismo estruturante para execução *on-line* de atividades tais como projeto, testes ou implantação, estas buscas baseadas em otimização constituem o que é atualmente conhecida como *Dynamic Adaptive Search-Based Software Engineering* (DASBSE) [127, 128, 130]. O objetivo é implantar, junto com o *software*, um subsistema que continuamente executa a otimização (*self-adaptation*).

Alguns resultados interessantes já estão disponíveis, referentes à otimização contínua de desempenho através da modelagem de parâmetros impactantes sob a forma de espaços de busca [67, 177], uso de programação genética para correção automática de *bugs* [23, 315], migração automática para novas plataformas/linguagens [189] e otimização de propriedades não-funcionais [319].

Em uma visão mais ambiciosa, um sistema poderia automaticamente redesenvolver parte do seu código-fonte com o objetivo de acomodar mudanças no ambiente ou novas plataformas/contextos, mantendo ainda as mesmas funcionalidades já presentes. Mesmo com críticas em relação à potencial baixa legibilidade e manutenibilidade de códigos gerados automaticamente, a ideia da transferência das atividades de programação para um nível mais alto de abstração – provavelmente focado em afirmações declarativas de funcionalidades – parece promissora. Todo o trabalho de "compilação" destas especificações declarativas em códigos-fonte com eficiência não-funcional ficaria a cargo do módulo realizador da otimização.

Outros trabalhos na área envolvem a utilização de hiper-heurísticas. As hiper-heurísticas definem um metaespaço de busca, onde o resultado da otimização é uma heurística que será, por sua vez, utilizada no processo principal de otimização. O objetivo é definir mecanismos mais genéricos, que atendam de forma unificada todas as atividades do processo de desenvolvimento e não demande, dos engenheiros de *software*, formação especializada em otimização para modelar novos problemas e selecionar as melhores heurísticas para o caso em questão. Uma visão geral sobre hiper-heurísticas pode ser encontrada em [52], enquanto aplicações de hiper-heurísticas em problemas da engenharia de *software* podem ser encontradas em [227, 238, 292].

Novamente, não é objetivo deste trabalho apresentar mais detalhes sobre DASBSE enquanto técnica para implementação de comportamento *self-adaptive*. Maiores informações sobre SBSE enquanto técnica para automação *off-line* de projeto arquitetural serão apresentadas no Capítulo 8.

3.3.3. Abordagens Baseadas em Formalização de Estrutura

As abordagens baseadas em formalização de estrutura [46] implementam comportamento *self-adaptive* através do uso de especificações formais que descrevem a estrutura de arquiteturas dinâmicas de *software*. Estas especificações formais permitem a representação de estilos arquiteturais, de arquiteturas particulares como instâncias de um determinado estilo arquitetural e de modificações que, quando realizadas em tempo de execução, viabilizam as adaptações estruturais desejadas. Dentre os formalismos mais utilizados nas abordagens desta escola, destacam-se: gramáticas de grafos e regras de reescrita de grafos, álgebra de processos e lógica.

Nas abordagens baseadas em gramáticas de grafos [217], a representação da arquitetura do sistema através de nós e arestas permite a definição de regras que manipulam o grafo em tempo de execução, implementando a adaptação desejada. Algumas abordagens utilizam nós para representar componentes e arestas para representar conectores binários. Outras soluções, mais sofisticadas, utilizam hipergrafos para representar conectores de mais alta cardinalidade.

As gramáticas de grafos são formadas por produções que definem como um grafo pode ser transformado em outro. Um grafo válido, gerado pela aplicação das produções de uma gramática, é portanto a representação de uma arquitetura de *software* em conformidade com o estilo arquitetural definido pela gramática. Adicionalmente, as adaptações podem ser especificadas por

regras adicionais de reescrita, que indicam as possíveis modificações estruturais que a arquitetura pode sofrer em tempo de execução. Classes específicas de gramáticas têm sido utilizadas para representação de arquiteturas de *software* nos últimos anos, tais como gramáticas com atributos [295], gramáticas em camadas [256] e gramáticas reservadas [325].

As abordagens baseadas em álgebra de processos utilizam formalismos tais como *Calculus of Communicating Systems* (CCS), *Communicating Sequential Processes* (CSP) ou *π-calculus* para definir as adições e remoções de componentes/conectores que caracterizam as adaptações sofridas pela arquitetura em tempo de execução. Dentre as abordagens desta categoria, destacam-se o *Dynamic Wright* [11], Darwin [206], LEDA [56] e PiLar [70].

As abordagens baseadas em lógica especificam arquiteturas dinâmicas de *software* utilizando, por exemplo, lógica de primeira ordem, lógica temporal ou a linguagem de especificação Z. Como exemplos de soluções desta categoria, pode-se citar o *Generic Reconfiguration Language* [91], a abordagem de Aguirre-Maibaum [5], o ZCL [75] e o FORMS [317].

3.3.4. Abordagens Baseadas em Teoria de Controle

As abordagens de *self-adaptation* baseadas em Teoria de Controle [138] consistem na utilização de técnicas sistemáticas para projeto de sistemas gerenciadores (controladores) que regulam a qualidade do serviço através de adaptações paramétricas, levando ainda em consideração as dinâmicas envolvidas nestas adaptações. Dinâmicas complexas ou não-conhecidas são frequentemente a causa de degradações de serviço, falhas ou provisionamento de recursos acima ou abaixo do necessário.

Como exemplo motivador, considere um sistema *self-adaptive* para comércio eletrônico onde o número de réplicas dos servidores *web* é automaticamente alterado de modo a garantir um tempo de resposta desejado. O *loop* de adaptação implementado deve continuamente monitorar os tempos médios de resposta e decidir quando e quantos servidores precisarão ser ligados ou desligados de modo a atender as metas de desempenho predefinidas, sem entretanto arcar com o custo de réplicas funcionando sem necessidade.

Premissas incorretas sobre variações nas demandas por serviço (*workloads*) ou sobre o tempo necessário para que novas réplicas acomodem as requisições podem gerar controladores com oscilações (máquinas ligando e desligando a todo momento) ou com adaptações que demoram muito para trazer o tempo de resposta de volta ao valor desejado. Tal situação pode aumentar os custos ou comprometer o atendimento de *Service Level Agreements* (SLAs), inviabilizando o negócio. Infelizmente, é esta a consequência do projeto *ad-hoc* de *loops* de adaptação.

A dinâmica de um sistema é modelada através da descrição de como um ou mais parâmetros de entrada – denominados entradas de controle (*control inputs*) – afetam, ao longo do tempo, um ou mais atributos de saída – denominados saídas medidas (*measured outputs*). Esta descrição geralmente envolve aspectos de regime transiente e aspectos de regime estacionário. O aspecto transiente é a resposta temporariamente percebida na saída medida após a manipulação da entrada de controle e que tende a cessar com o tempo. Após o período transiente, o sistema passa a apresentar a resposta em regime estacionário (estável ou não) até que uma nova atuação seja efetuada na entrada de controle.

Embora análises via teoria das filas e planejamento de capacidade [154] sejam frequentemente utilizadas em cenários como o do exemplo acima, tais teorias geralmente são aplicadas *off-line* e consideram apenas respostas em regime estacionário. A prática atual tende a evidenciar que os aspectos transientes são igualmente importantes durante o projeto de *loops* de adaptação.

3.3.4.1. Fundamentos

Um **sistema de controle** [85, 221, 228] tem como objetivo fazer com que um determinado parâmetro de desempenho de um sistema (saída medida: *measured output*) seja mantido em um valor de referência predefinido (entrada de referência: *reference input*), através da manipulação de um valor de entrada (entrada de controle: *control input*) que interfere no desempenho em questão. Os primeiros sistemas de controle datam do ano 300 a.C., quando os gregos criaram os primeiros mecanismos para regular níveis de água em tanques ou de combustíveis para alimentação de lâmpadas. No século XVIII os primeiros sistemas mecânicos para controle de velocidade em motores a vapor foram criados.

Após o desenvolvimento dos fundamentos matemáticos para estabilidade de sistemas no final do século XIX e das técnicas de análise e projeto no século XX, os sistemas de controle passaram a estar presentes em aplicações de diversas áreas. Atualmente, são utilizados para navegação de mísseis, aeronaves e navios; controle de níveis, temperaturas e concentrações em processos químicos; automação de plantas industriais e robôs; posicionamento de antenas; leitura de CDs; dentre inúmeras outras aplicações.

A Teoria de Controle disponibiliza todo o fundamento matemático e sistematização dos processos de projeto e análise de sistemas de controle. O objetivo é garantir que certas propriedades (a serem discutidas a seguir) sejam mantidas no sistema controlado. Tais propriedades implicam, por exemplo, no conforto e segurança dos passageiros de uma aeronave, na qualidade apresentada pelo produto final de um processo químico, ou na garantia da qualidade de serviço em um sistema computacional. Dado o papel central dos sistemas de controle nesta tese, uma visão mais detalhada sobre seus fundamentos será aqui apresentada.

A compreensão da Teoria de Controle requer conhecimento prévio sobre equações diferenciais [36, 144], teoria de sinais (transformadas de Fourier, Laplace e Z) e sistemas (estabilidade, linearidades e variâncias no tempo) [61, 284], números complexos, dentre outros assuntos. Este trabalho resume-se à apresentação dos aspectos da teoria que fundamentam a análise e projeto de controladores de acordo com atributos de qualidade desejados. O leitor interessado em uma discussão mais aprofundada pode consultar as referências acima apresentadas.

Esta tese adota a definição de **sinal** proposta por Proakis & Manolakis [246]:

Definição 3.2: Sinal (Proakis & Manolakis, 2006, pág. 2)

Um sinal é qualquer quantidade mensurável que varia em função do tempo, espaço ou qualquer outra variável ou variáveis independentes.

Embora os sinais mais comuns representem valores reais variando em função do tempo, o valor mensurado (variável dependente) pode ser um número complexo ou um vetor (sinal multicanal), definido em função de qualquer número de variáveis independentes. Os sinais podem ser classificados em relação a diversos critérios. De acordo com os tipos das variáveis envolvidas, o sinal pode ser classificado em: *i)* analógico – todas as variáveis são contínuas; *ii)* de "tempo" discreto – a(s) variável(eis) independente(s) é(são) discreta(s), mas a variável dependente é contínua; e *iii)* digital – todas as variáveis são discretas.

Os sinais podem ainda ser classificados como randômicos ou determinísticos, periódicos ou aperiódicos, pares ou ímpares, dentre outros critérios. Sinais podem ainda sofrer uma série de operações e serem representados em diferentes formatos. Não é objetivo desta tese apresentar tais fundamentos. O leitor interessado pode consultar referências tais como [61, 233, 284]. Sinais analógicos em função do tempo são geralmente denotados por $x(t)$, enquanto sinais digitais são representados por $x[k]$.

Definição 3.3: Sistema (Oppenheim et al., 1996, pág. 38 [233])

Um sistema é um processo através do qual sinais de entrada são transformados pelo sistema ou fazem com que o sistema responda de uma forma que produza outros sinais como saída.

Conforme apresentado na Figura 3.10(a), um **sistema** produz um sinal de saída (*output*) $y(t)$ a partir de um sinal de entrada (*input*) $u(t)$ e seu comportamento é geralmente influenciado por um sinal de perturbação (*disturbance input*) $d(t)$. Quando um valor de saída $y(t_0)$ depende somente do valor de entrada $u(t_0)$, e não de valores passados ou futuros de $u(t)$, diz-se que o sistema é **estático** ou *memoryless*. Os **sistemas dinâmicos**, por outro lado, retêm algum estado interno, influenciado por valores passados (ou futuros, em sistemas não-causais) do sinal de entrada $u(t)$. Este estado interno irá influenciar a geração do sinal de saída $y(t)$. Um sistema pode ser classificado como contínuo ou discreto a depender dos tipos dos sinais por ele manipulado; e ainda como **SISO** (*Single-Input Single-Output*) ou **MIMO** (*Multiple-Input Multiple-Output*), dependendo da quantidade de sinais de entrada e saída.

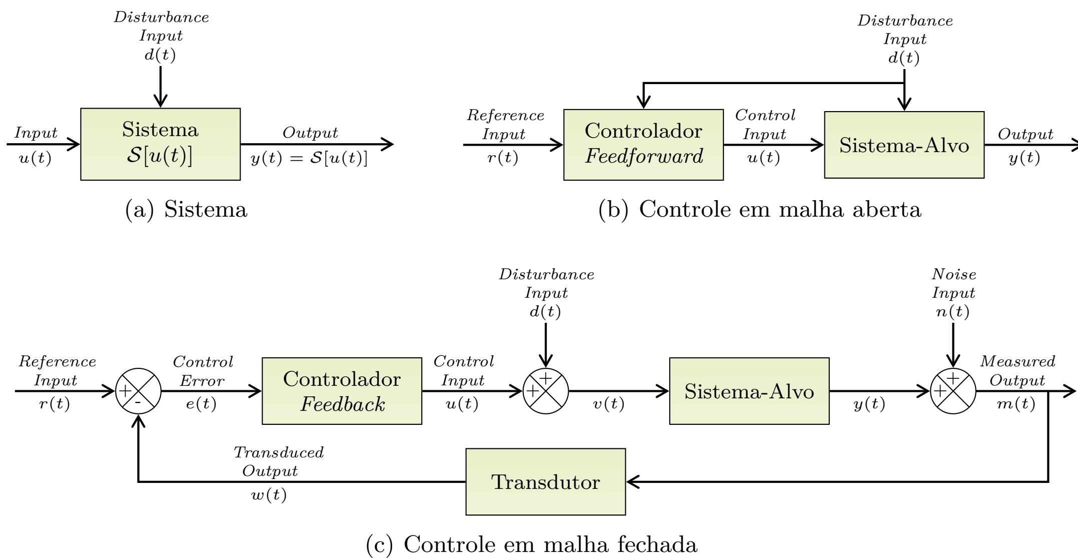


Figura 3.10.: Sistema dinâmico e estruturas de controle em malha aberta e malha fechada (adaptada de [138]).

Um controlador pode ser adicionado a um sistema com o objetivo de manter a saída em algum valor de referência (*reference input*) $r(t)$ desejado, mesmo na presença de um sinal de perturbação $d(t)$. No controle em **malha aberta** (*feedforward control*) – apresentado na Figura 3.10(b) – o controlador decide o valor da entrada de controle (*control input*) $u(t)$ que irá produzir o valor de saída desejado (faz com que $y(t)$ se iguale a $r(t)$), com base nas dinâmicas do sistema e nas características atuais das perturbações. Esta estratégia requer a modelagem precisa tanto do sistema quanto das perturbações, de modo que a modificação do valor de entrada produza o valor de saída desejado, mesmo na presença de perturbações.

Uma outra alternativa, apresentada na Figura 3.10(c), é a utilização de controle em **malha fechada** (*feedback control*). Nesta abordagem, perturbações $d(t)$ externas ao sistema não são diretamente consideradas. Para isso, a saída medida (*measured output*) $m(t)$ é constantemente monitorada por um transdutor/sensor, que a envia de volta ao controlador (eventualmente com transformações, passando a constituir o *transduced output* $w(t)$). O erro de controle (*control error*) $e(t)$ é a diferença entre o valor de referência e o valor atual do *transduced output*, e serve como base para que o controlador decida o novo valor de entrada de controle a ser aplicado no sistema. A presença de perturbações faz com que a entrada efetiva de controle $v(t)$ seja definida como $u(t) + d(t)$. De forma análoga, a presença de ruídos de sensoriamento $n(t)$ faz com que a saída medida $m(t)$ seja definida como $y(t) + n(t)$.

Controladores em malha fechada definem mecanismos reativos, tornando desnecessária a difícil atividade de modelagem explícita de perturbações. Controladores em malha fechada bem projetados detectam rapidamente a presença de perturbações e fazem com que a saída medida retorne para o valor de referência o mais rápido possível, preferencialmente sem oscilações. Controladores mal projetados, por outro lado, podem introduzir tempos baixos de convergência, oscilações ou, no pior dos casos, gerar sistemas instáveis (não-convergentes). Estabilidade e outras propriedades de controladores serão discutidas a seguir.

Decidir entre controle em malha aberta ou fechada requer análise criteriosa das variações das perturbações, complexidade do sistema e custos de projeto do controlador. Eventualmente, abordagens híbridas (em malha aberta e fechada) podem ser adotadas, beneficiando-se tanto dos aspectos proativos da malha aberta quanto da adaptação a perturbações da malha fechada. A Tabela 3.3 apresenta uma comparação entres essas duas abordagens de controle.

	Malha Aberta	**Malha Fechada**
Requer monitoração do *output* ?	Não	Sim
Pode tornar instável um sistema que, quando não controlado, é estável ?	Não	Sim, com controladores mal projetados
Requer modelagem precisa do sistema e das perturbações ?	Sim	Não
É capaz de adaptar-se a perturbações não conhecidas ?	Não	Sim

Tabela 3.3.: Comparação entre controle em malha aberta e malha fechada (adaptada de [138]).

Propriedades de sistemas de controle. Sistemas dinâmicos podem variar amplamente em relação à forma como convertem sinais de entrada em sinais de saída. Definir classes mais restritas de sistemas permite a derivação de técnicas gerais de análise e projeto, aplicáveis a qualquer representante da categoria em questão. Os **sistemas lineares** representam uma classe de sistemas amplamente utilizada e fortemente suportada por um conjunto poderoso de ferramentas para análise e projeto.

Definição 3.4: Sistema Linear (Chaparro, 2010, pág. 120 [61])

Um sistema representado por $\mathcal{S}$ é dito linear se para quaisquer entradas $u(t)$ e $v(t)$, e quaisquer constantes α e β, o princípio da superposição é mantido, ou seja:

$$\begin{aligned} \mathcal{S}[\alpha u(t) + \beta v(t)] &= \mathcal{S}[\alpha u(t)] + \mathcal{S}[\alpha v(t)] \\ &= \alpha\mathcal{S}[u(t)] + \beta\mathcal{S}[v(t)] \end{aligned} \tag{3.1}$$

Em sistemas dinâmicos, a resposta a uma entrada pontual em algum tempo t_0 é um sinal que se estende além de t_0 e é descrito pelos componentes em regime transiente e em regime estacionário. A Figura 3.11(a) apresenta a resposta $y(t)$ de um sistema a um sinal de entrada $u(t)$ modelado como um passo (*step*) unitário (variação abrupta, no tempo 0, do valor 0 para o valor 1). A resposta transiente é a reação do sistema à sua "inércia" natural, verificada imediatamente após a mudança de patamar no sinal de entrada. Já a resposta em regime estacionário é o sinal apresentado pelo sistema após o término da fase transiente. Sistemas *memoryless* não apresentam resposta transiente.

Sistemas lineares apresentam propriedades interessantes, tais como linearidade estática e fidelidade senoidal. A linearidade estática afirma que se o sinal de entrada for mantido constante, a resposta em regime estacionário do sistema será o sinal de entrada multiplicado por uma determinada constante. A fidelidade senoidal, por sua vez, afirma que se o sinal de entrada for uma

senoide, a resposta em regime estacionário será também uma senoide, com a mesma frequência do senoide de entrada (embora com amplitudes e fases eventualmente diferentes). Uma outra propriedade apresentada por alguns sistemas é a **invariância no tempo**.

Definição 3.5: Sistema Invariante no Tempo (Chaparro, 2010, pág. 125 [61])

Um sistema representado por $\mathcal{S}$ é dito invariante no tempo se para qualquer entrada $u(t)$ e saída $\mathcal{S}[u(t)]$ correspondente, a saída de uma entrada deslocada no tempo $u(t \pm \tau)$ (antecipada ou atrasada) é a saída original deslocada pelo mesmo fator: $\mathcal{S}[u(t \pm \tau)]$, ou seja:

$$\begin{aligned} u(t) &\Rightarrow y(t) = \mathcal{S}[u(t)] \\ u(t \pm \tau) &\Rightarrow y(t \pm \tau) = \mathcal{S}[u(t \pm \tau)] \end{aligned} \tag{3.2}$$

Sistemas lineares e invariantes no tempo (LIT) podem ser analisados através da decomposição dos sinais envolvidos em elementos mais simples e mais fáceis de serem investigados. De forma análoga, sistemas complexos podem ser sintetizados a partir de componentes mais elementares. A linearidade e invariância no tempo garantem que estes componentes podem ser combinados, escalados e deslocados no tempo, constituindo o sistema final desejado.

As decomposições mais comuns envolvem a descrição de como o sistema reage quando a entrada é um impulso unitário (convolução), passo unitário, senoides em diferentes frequências (Transformadas de Fourier) ou senoides em diferentes frequências em conjunto com exponenciais com diferentes amortecimentos (Transformada de Laplace / Transformada Z) [61, 284]. Conhecer a resposta do sistema a qualquer um destes casos é suficiente para caracterizar a resposta a qualquer sinal de entrada arbitrário. Novamente, não é objetivo deste trabalho apresentar os fundamentos sobre Transformada de Laplace e Transformada Z, o leitor pode utilizar as referências acima para um aprofundamento no assunto.

Um controlador é adicionado a um sistema dinâmico com o objetivo de modificar suas propriedades e/ou garantir que perturbações não desviem a saída medida do valor de referência estipulado. Diferentes controladores irão produzir sistemas que apresentam diferentes dinâmicas, geralmente descrita pelas propriedades conhecidas como SASO (*Stability*, *Accuracy*, *Settling time* e *Overshoot*).

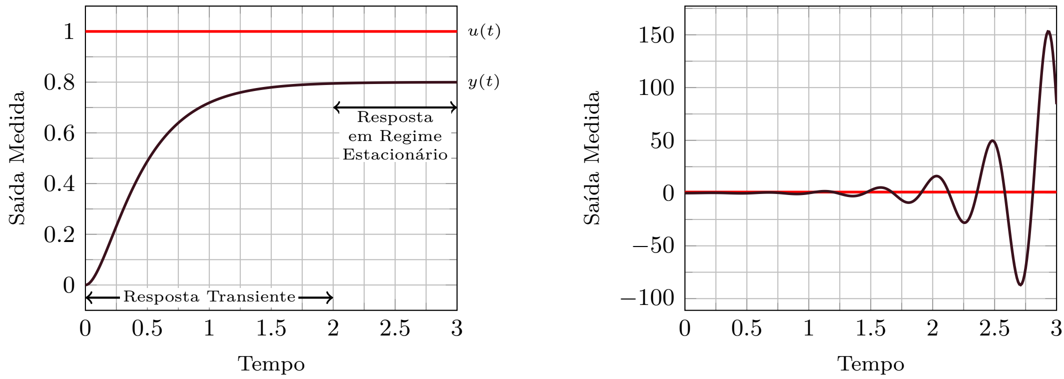
Dentre as diferentes definições e critérios de estabilidade existentes [283], a estabilidade BIBO (*Bounded Input Bounded Output*) é geralmente a utilizada em sistemas de controle. O sistema apresentado na Figura 3.11(a) é **BIBO-estável**. Já o sistema da Figura 3.11(b) não é BIBO-estável pois não produz um sinal delimitado quando o passo unitário (sinal delimitado) é utilizado como entrada.

Definição 3.6: Sistema BIBO-estável (Hayes, 1998, pág. 10 [133])

Um sistema é BIBO-estável se para todo sinal delimitado (*bounded*) de entrada, $|u(t)| \leq A \leq \infty$, é produzido um sinal de saída também delimitado $|y(t)| \leq B \leq \infty$. Um sistema LIT é BIBO-estável se sua resposta a um impulso unitário $h(t)$ é absolutamente somável:

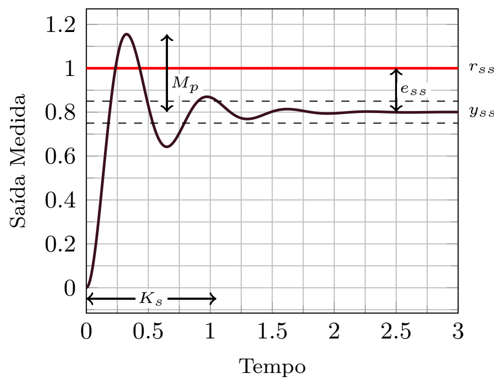
$$\sum_{n=-\infty}^{\infty} |h(t)| < \infty$$

Estabilidade é sempre a meta mínima no projeto de sistemas de controle, visto que sistemas instáveis geralmente implicam em falhas ou em consumo excessivo de recursos. Se o sistema é estável, outras propriedades podem então ser consideradas. Conforme apresentado na Figura 3.11(c), um sistema é **preciso** (*accurate*) se, em regime estacionário, a saída medida (y_{ss}) é igual ao (ou se torna suficientemente próximo do) valor de referência (r_{ss}). O erro em regime

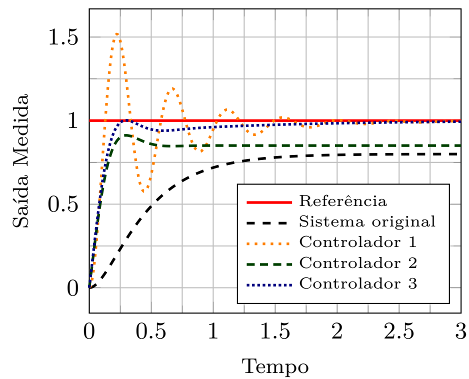


(a) Resposta ao passo unitário em um sistema estável não-controlado

(b) Resposta ao passo unitário em um sistema instável não-controlado



(c) Propriedades de um sistema de controle



(d) Respostas de sistemas com diferentes controladores

Figura 3.11.: Propriedades de sistemas de controle (a, b e c) e ações realizadas por diferentes controladores (d).

estacionário de um sistema (e_{ss}) é a diferença, em regime estacionário, entre o valor de referência e a saída medida. Sistemas de controle precisos são essenciais para garantir que as metas sejam satisfatoriamente obtidas, mesmo na presença de perturbações.

O **tempo de estabilização** (*settling time*) K_s é o tempo necessário para que, após uma modificação no sinal de entrada ou presença de uma perturbação, a saída medida limite os seus valores a uma faixa pré-estabelecida de variação ao redor de y_{ss} (retas horizontais tracejadas na Figura 3.11(c)). Sistemas com tempo de estabilização baixo são importantes para convergir para um novo valor (ou rejeitar uma perturbação) antes que uma nova perturbação ou mudança de referência apareçam.

O **sobressinal** (*overshoot*) M_p de um sistema é o valor máximo (normalizado) pelo qual a saída medida excede o seu valor em regime estacionário (y_{ss}), durante a resposta transiente do sistema. A motivação para construção de sistemas de controle com baixo (ou nenhum) sobressinal pode ser consequência de limitações físicas em sistemas mecânicos (ex: pouca oscilação em sistemas de controle de aviões), de requisitos para qualidade do produto em processos químicos ou da necessidade de baixo *overhead* de controle em sistemas computacionais.

Conforme apresentado na Figura 3.11(d), diferentes controladores – projetados a partir de técnicas variadas – podem gerar sistemas com diferentes respostas transiente e em regime estacionário. As propriedades acima descritas geralmente são conflitantes. Por exemplo, diminuir o tempo de estabilização do sistema final, obtido após a inserção do controlador, geralmente implica em um aumento do sobressinal. A Teoria de Controle disponibiliza todo um arcabouço matemático para

analisar e projetar controladores que sejam estáveis e que atendam a critérios específicos de precisão, tempo de estabilização e sobressinal. Este arcabouço requer que os elementos envolvidos (sistema original e controlador) sejam sistemas LIT.

Uma outra propriedade frequentemente considerada em sistemas de controle é robustez. Um sistema de controle é **robusto** se ele continua apresentando o mesmo desempenho de controle quando operando em regiões (valores de atuação) diferentes daquela para a qual ele foi projetado ou quando mudanças significativas ocorrem no sistema e/ou ambiente. Embora sistemas LIT sejam raramente encontrados na prática, grande parte dos sistemas reais pode ter uma determinada região de operação linearizada. Isto faz com que o desempenho de controle seja satisfatório nessa região mas se degrade rapidamente quando o sistema passa a operar em condições diferentes. Uma ampla classe de controladores adaptativos ou auto-reguláveis pode ser utilizada para amenizar este problema.

3.3.4.2. Modelagem do Sistema

Conhecer o sistema original a ser controlado é o primeiro passo para o projeto sistemático de controladores. O objetivo é construir um sistema controlado final que apresente características específicas para as propriedades acima descritas. Um amplo conjunto de técnicas para projeto de controladores pode ser encontrado atualmente na literatura [25, 33, 221]. A escolha da técnica mais adequada depende de diversos fatores, dentre eles a forma com a qual os requisitos de controle foram especificados (no domínio do tempo ou no domínio da frequência – [291], pág. 231), do objetivo do controle (regulação, rastreamento, rejeição de perturbações ou otimização – [138], pág. 6), da estratégia de controle (descritas a seguir) e do quanto se conhece sobre o sistema (forma de obtenção do modelo).

De um modo geral, o modelo do sistema pode ser obtido por técnicas pertencentes a dois grupos principais:

- Modelagem por princípios fundamentais (*first-principles*): consiste na utilização de leis físicas que regem o processo que caracteriza o sistema a ser controlado. É bastante utilizado no controle de sistemas mecânicos e processos químicos, em função da disponibilidade de um arcabouço teórico bastante expressivo para descrição das dinâmicas de tais sistemas. Em sistemas computacionais, entretanto, estes princípios fundamentais se resumem à teoria das filas, mais utilizadas para modelagem de respostas em regime estacionário. A alta presença de componentes estocásticos em sistemas computacionais é um outro fator complicador na derivação de princípios fundamentais.
- Modelagem por experimentação: utiliza técnicas estatísticas ou de aproximação para derivar – a partir de dados coletados do sistema a ser controlado – um modelo que satisfatoriamente capture os relacionamentos entre o sinal de entrada e o sinal de saída [203]. É apropriado quando princípios fundamentais não estão disponíveis e experimentação com o sistema real é viável. Dentre as técnicas mais aplicadas, destacam-se o uso de modelos ARX (*AutoRegressive eXogeneous*) ([138], pág. 39) associados a técnicas de identificação de sistema (*system identification*) [86] e aproximações a partir de parâmetros obtidos através de uma experimentação denominada ensaio ao degrau (*bump test*)[6] [25, 312].

Os modelos são geralmente descritos por equações diferenciais ou equações-diferença (em sistemas contínuos e discretos, respectivamente); por funções de transferência no plano $\mathcal{S}$ ou no plano $\mathcal{Z}$ (em sistemas contínuos e discretos, respectivamente); por modelos de espaço de estados (*state-space* - geralmente utilizados em sistemas MIMO); ou por modelos paramétricos de aproximação. Devido ao foco deste trabalho na aplicação de controle em sistemas computacionais, uma breve descrição das versões discretas das representações acima será apresentada a seguir.

[6] Investigação empírica que coleta parâmetros referentes à resposta do sistema a um determinado sinal, geralmente um passo ou impulso unitários.

Equações-diferença. Uma **equação-diferença** define uma relação de recorrência onde o valor atual da saída é definido por valores passados de entrada e de saída. Uma classe de equações-diferença largamente utilizada em sistemas de controle são os **modelos ARX**:

$$y[k] = a_1 y[k-1] + \ldots + a_n y[k-n] + b_1 u[k-1] + \ldots + b_m u[k-m] \tag{3.3}$$

É comum se referir aos valores de n e m como a "estrutura do modelo", enquanto os coeficientes a_k e b_k são chamados de "parâmetros do modelo". A ordem do modelo/sistema é definida por n e indica quantas saídas passadas são utilizadas para o cálculo da saída atual. Modelos de primeira ordem são interessantes pois facilitam as tarefas de análise e projeto de controladores, mas podem constituir representações imprecisas do sistema real. Neste caso, modelos de mais alta ordem devem ser derivados, com o objetivo de melhor capturar as dinâmicas apresentadas, ao custo de uma maior dificuldade de análise do sistema e projeto de controladores.

Um outro problema da utilização de modelos de alta ordem é o *overfitting* ([138]: pág. 45), situação onde um baixo número de amostras coletadas associado a valores altos de n e m gera modelos com baixa generalidade em cenários diferentes daquele utilizado na experimentação. A precisão do modelo em relação à generalização da dinâmica de um sistema pode ser avaliada através dos indicadores apresentados na Tabela 3.4, aplicados ou nos dados do experimento que originou o modelo ou em dados obtidos em uma nova investigação.

Modelos efetivos geralmente apresentam valores baixos para o RMSE, quando comparado com a faixa de valores de saída apresentada. A variabilidade (R^2) é um número entre 0 a 1, onde 0 significa que o modelo se reduz à obtenção da média de y (baixa captura da variabilidade) e 1 sugere (mas não garante) captura total da variabilidade. Bons modelos geralmente apresentam $R^2 \geq 0.8$. Valores pequenos para o coeficiente de correlação (CC) indicam que a maior parte da informação foi capturada no modelo e pouco pode ser feito para melhorá-lo. Vale ressaltar que estes indicadores estão sujeitos a casos excepcionais, onde bons valores são obtidos em modelos ruins ou vice-versa. Portanto, é sempre recomendado a utilização de gráficos de análise de resíduos ([138]: pág. 55) para confirmar os resultados.

Root-Mean-Square Error	**Variabilidade**	**Coeficiente de Correlação**
$RMSE = \sqrt{\frac{1}{N}\sum_{k=1}^{N}(y[k+1]-\hat{y}[k+1])^2}$	$R^2 = 1 - \frac{var(y-\hat{y})}{var(y)}$	$CC = \frac{\sum_k e[k]u[k]}{\sqrt{var(e[k])var(u[k])}}$
$y[k]$=valor obtido experimentalmente; $var(y)$=variância de $y[k]$;	$\hat{y}[k]$=valor previsto pelo modelo; $e[k] = y[k] - \hat{y}[k]$;	

Tabela 3.4.: Indicadores para avaliação de modelos obtidos via identificação de sistema.

Funções de transferência. O modelo ARX apresentado na Equação 3.3 pode ser descrito no domínio da frequência como:

$$Y(z) = \frac{(b_1 z^{m-1} + \ldots + b_m) z^{n-m}}{z^n - a_1 z^{n-1} - \ldots - a_n} \tag{3.4}$$

A função $Y(z)$ é denominada **função de transferência** e permite a análise facilitada de diversas propriedades do sistema, não tão evidentes quando utilizando equações-diferença (domínio do tempo discreto). O denominador da função $Y(z)$ é denominado **polinômio característico** e suas raízes são os **polos** da função de transferência. As raízes do numerador da função $Y(z)$ são denominadas **zeros** da função de transferência.

Os polos são importantes pois exercem influência direta em propriedades como estabilidade e

tempo de estabilização. A Tabela 3.5 apresenta como estas propriedades podem ser estimadas a partir da função de transferência do sistema. Sabe-se que o sistema descrito por $Y(z)$ é BIBO-estável se todos os polos tiverem magnitude < 1 (estiverem dentro do círculo unitário definido no plano $\mathcal{Z}$). Sabe-se também que o ganho em regime estacionário y_{ss} é dado por $Y(1)$ e que o tempo de estabilização será menor quanto mais próximo o polo dominante (de maior magnitude) estiver de zero. Equações de transferência com polo dominante real $p_1 \geq 0$ não apresentam sobressinal pois contribuem com um fator p_1^k não-oscilante no domínio do tempo. Já sistemas com polo real < 0 ou polos complexos apresentam sobressinal e seu valor pode ser estimado conforme apresentado na Tabela 3.5 [138].

Critério para Estabilidade BIBO	**Tempo de Estabilização**	**Sobressinal Máximo**	**Ganho em Regime Estacionário**
Todos os polos da função de transferência devem ter magnitude < 1	$K_s \approx \dfrac{-4}{\log r}$	$M_p \approx \begin{cases} 0 & \text{polo dominante real } p_1 \geq 0 \\ \lvert p_1 \rvert & \text{polo dominante real } p_1 < 0 \\ r^{\pi/\lvert\theta\rvert} & \text{polos dominantes } p_1, p_2 = re^{\pm j\theta} \end{cases}$	$y_{ss} = Y(1)$
$r = \max_i \lvert p_i \rvert$=magnitude do maior polo em *loop* fechado.			

Tabela 3.5.: Análise de propriedades do sistema a partir da função de transferência.

O número de polos (incluindo os repetidos) é sempre igual à ordem do sistema. Sistemas de segunda ou mais alta ordem podem apresentar polos complexos, que implicam em respostas oscilatórias a sinais comuns como o impulso ou passo unitários. Embora sistemas de mais alta ordem dificultem a análise de propriedades através dos polos, uma aproximação satisfatória pode ser obtida ao substituir o sistema por um outro de primeira ordem contendo somente o **polo dominante** p' [138]:

$$Y'(z) = \frac{Y(1)(1-p')}{z-p'}; \qquad \text{se } p' \text{for real} \tag{3.5}$$

Nem todos os sistemas, entretanto, possuem polo dominante. Observações empíricas indicam que, geralmente, o polo precisa ter magnitude duas vezes maior que a dos outros para conseguir dominar as propriedades.

Os zeros da função de transferência também influenciam as propriedades, embora de forma não tão importante quantos os polos. Esta influência é proporcional à distância do zero até algum polo. Quando o zero é igual a algum polo ocorre o que é denominado de cancelamento polo-zero e o sistema passa a se comportar como um sistema de uma ordem a menos. Sistemas com algum zero fora do círculo unitário apresentam uma resposta transiente onde a saída medida inicialmente se move na direção contrária ao valor de referência para, após algum tempo, começar a convergir para o valor em regime estacionário (sistemas com fase não-mínima).

Modelos de Espaço de Estados. Sistemas de controle mais sofisticados podem trabalhar com múltiplos sinais de entrada e saída, com possibilidades de interação entre entradas e influência de um único sinal de entrada em diversas saídas. Tais sistemas, denominados MIMO, requerem a manipulação de diversas equações de transferência. Os **modelos de espaço de estados (state-space)** ([85]: Capítulo 3) constituem uma forma escalável de representação de sistemas com múltiplas entradas e saídas, pois passam a representar sinais e sistemas como vetores e matrizes, respectivamente. Felizmente, grande parte das técnicas de análise utilizadas em sistemas SISO são também aplicadas, de forma direta, em modelos de espaço de estados.

Em modelos de espaço de estados, um sistema passa a ser representado por uma ou mais variáveis

de estado, não necessariamente iguais às variáveis que constituem os sinais de saída. As variáveis de estado atendem a dois propósitos principais: *i)* descrever a dinâmica do sistema, informando como o estado varia em função do estado anterior e das entradas; e *ii)* informar como as saídas podem ser derivadas a partir do estado. A forma geral de um modelo de espaço de estados com n variáveis de estado, m_I sinais de entrada e m_O sinais de saída é:

$$\mathbf{x}[k+1] = \mathbf{A}\mathbf{x}[k] + \mathbf{B}\mathbf{u}[k] \tag{3.6}$$

$$\mathbf{y}[k] = \mathbf{C}\mathbf{x}[k] \tag{3.7}$$

onde $\mathbf{x}[k]$ (note o uso do negrito para diferenciar de sinais escalares) é um vetor $n \times 1$ representando o estado do sistema, $\mathbf{u}[k]$ é um vetor $m_I \times 1$ representando as entradas e $\mathbf{y}[k]$ é um vetor $m_O \times 1$ representando as saídas. $\mathbf{A}$ é uma matriz $n \times n$ indicando, em cada linha, como cada variável de estado é influenciada por valores passados de cada variável de estado. $\mathbf{B}$ é uma matriz $n \times m_I$ indicando, em cada linha, como cada variável de estado é influenciada por valores passados de cada sinal de entrada. Por fim, $\mathbf{C}$ é uma matriz $m_O \times n$ que informa como cada sinal de saída pode ser calculado em função das variáveis de estado. Modelos de espaço de estados disponibilizam uma forma concisa de representação de sistemas MIMO pois capturam $m_I \cdot m_O$ funções de transferência em apenas duas equações (3.6 e 3.7 apresentadas acima).

Os polos de um modelo de espaço de estados são as soluções da equação $det(z\mathbf{I} - \mathbf{A}) = 0$ (autovalores de $\mathbf{A}$) e os mesmos critérios apresentado na Tabela 3.5 são utilizados para verificar estabilidade BIBO, tempo de estabilização, precisão e sobressinal. Modelos descritos em equações-diferença ou funções de transferência podem ser transformados em modelos de espaço de estados, caso sejam mais fáceis de serem manipulados deste modo. Modelos de espaço de estados podem também ser construídos via identificação de sistema, desde que o sistema seja observável, ou seja, o estado possa ser inferido a partir da saída medida.

Embora facilitem a análise e projeto de sistemas MIMO, modelos de espaço de estados requerem considerações especiais em relação à escolha das variáveis de estado, de modo que o sistema seja controlável e observável (mais detalhes em [138]: Seção 7.6). Em outros casos, nem todas as variáveis de entrada têm influência direta em todas as variáveis de saída e o sistema pode ser transformado em um controle MIMO com menos variáveis, associado a um ou mais controles SISO. Técnicas tais como *Relative Gain Array* ([33]: Seção 13.4 e [25]: pág. 312) podem ser utilizadas para obter emparelhamentos eficientes entre variáveis de entrada e saída.

Modelos paramétricos de aproximação. Quando princípios fundamentais não estão disponíveis e experimentação com sinais de entrada de alta variação é inviável, uma opção frequentemente considerada para obtenção de modelos aproximados do sistema é a realização de ensaios ao degrau (*bump tests*). Nestes cenários, verifica-se a resposta apresentada pelo sistema quando o sinal de entrada é um passo unitário, com o objetivo de coletar parâmetros que serão utilizados em algum modelo "genérico" previamente definido. Um modelo de três parâmetros amplamente utilizado é o *First-Order Plus Dead Time* (FOPDT) [312]:

$$G(s) = \frac{R}{1 + sT} \cdot e^{-sL} \tag{3.8}$$

O FOPDT é caracterizado por três parâmetros, obtidos via ensaio ao degrau conforme indicado na Figura 3.12. O parâmetro R é denominado ganho estático e corresponde à saída medida, em regime estacionário. O parâmetro L, denominado *dead time* ou *time lag*, é obtido através da intersecção do eixo do tempo com a reta tangente de maior inclinação durante a resposta transiente do sistema. Duas formas de obtenção do parâmetro T, denominado *time constant*, estão presentes na literatura. Em uma delas, T é definido como o tempo necessário, após o *dead time*, para que o sistema atinja 63% $(1 - e^{-1})$ da sua saída medida, em regime estacionário

(indicado como A na Figura 3.12). Em outras referências, define-se T como o tempo necessário, após o *dead time*, para que o sistema atinja a saída medida, em regime estacionário (indicado como B na Figura 3.12). Ambos os métodos produzem valores semelhantes, desde que a dinâmica do sistema seja satisfatoriamente aproximada pela Equação 3.8.

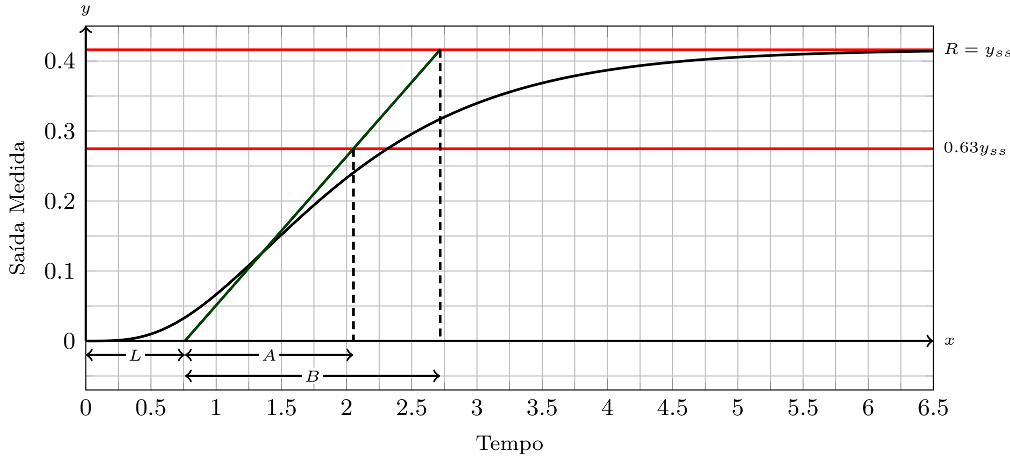


Figura 3.12.: Parâmetros coletados via ensaio ao degrau (*bump test*) (adaptada de [25], pág. 17).

Modelos paramétricos aproximados são interessantes porque diversas técnicas empíricas para projeto de controladores fazem uso direto dos parâmetros R (ou $a = RL/T$), L e T para construir sistemas controlados com propriedades específicas. Essas técnicas empíricas, apresentadas na Seção 3.3.4.4, são mais simples de serem utilizadas do que estratégias analíticas que requerem modelos mais sofisticados do sistema e/ou posicionamento de polos.

Outros modelos paramétricos, mais simples ou mais complexos, também podem ser utilizados. Por exemplo, o *Integrator Plus Dead Time* (IPDT) requer somente dois parâmetros e é descrito por $G(s) = Re^{-Ls}/s$. É papel do engenheiro de controle escolher um modelo paramétrico que melhor aproxime a dinâmica do sistema e selecionar uma técnica de projeto compatível com os parâmetros utilizados no modelo. Mais informações podem ser encontradas em [25] e um amplo catálogo de modelos paramétricos e técnicas de projeto correspondentes está disponível em [312].

3.3.4.3. Estratégias para Feedback Control

Decidir a lei de atuação e os parâmetros de configuração utilizados em um controlador é uma das tarefas mais importantes e desafiadoras do projeto de sistemas *self-adaptive* baseados em *feedback control*. Uma vasta coleção de estratégias e técnicas para projeto e sintonia (*tuning*) de controladores está disponível na literatura [25, 26, 312]. De um modo geral, pode-se classificar as estratégias em relação ao grau de robustez apresentado pelo controlador por elas projetado. Conforme apresentado na Figura 3.13, um controlador pode trabalhar com parâmetros constantes (ganho fixo) ou variáveis.

Sistemas cuja atuação é limitada a uma única região de operação (faixa de variação da entrada de controle), linear ou satisfatoriamente linearizável, podem ser efetivamente controlados por estratégias de ganho fixo (parâmetros constantes). Estes parâmetros são sistematicamente definidos *off-line* e não sofrem modificações durante a operação. Sistemas com maior dinamismo e/ou regiões de operação mais amplas, entretanto, geralmente sofrem degradação (perda de robustez) ou quando as perturbações mudam drasticamente ou quando passam a atuar em regiões de operação diferentes daquela para a qual foram sintonizados.

Quando a região de operação é ampla demais (não-linear) para se obter um desempenho de controle satisfatório com um único conjunto de valores de parâmetros, pode-se utilizar o **es-**

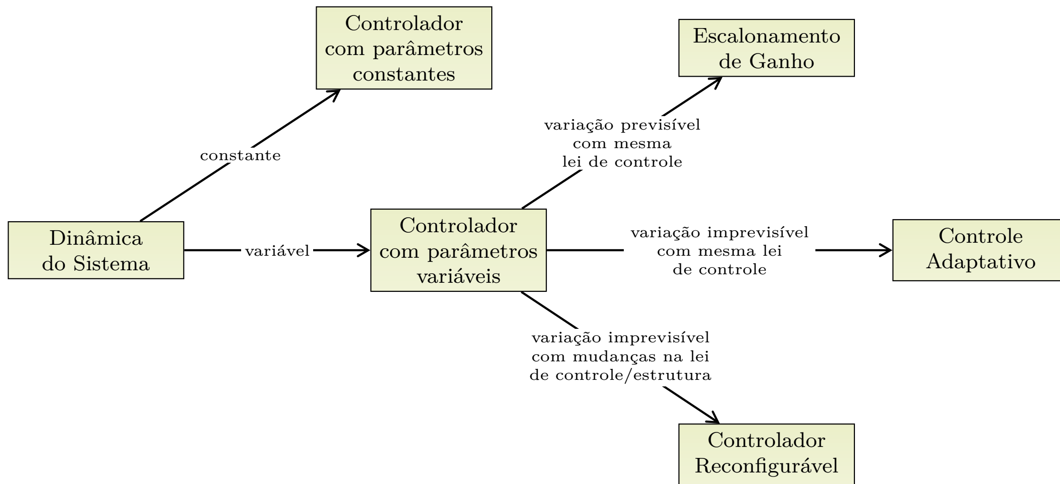


Figura 3.13.: Diferentes graus de robustez apresentado por controladores (extensão de [26], pág. 30).

calonamento de ganho[7] ([26], Capítulo 9). Esta estratégia consiste em dividir a região de operação em duas ou mais sub-regiões, cada uma contendo valores específicos para os parâmetros de controle. O controlador deve ser capaz de detectar em qual sub-região ele está atuando no momento e ressintonizar seus parâmetros (escalonar o ganho) com os valores daquela sub-região. Embora consiga um melhor desempenho em diversas regiões de operação, o escalonamento de ganho requer o projeto de vários controladores (maior custo) e pode trazer instabilidades caso o trânsito entre diferentes regiões ocorra com muita frequência.

Quando as sub-regiões e não-linearidades não são conhecidas, entretanto, as estratégias de sintonia (escolha dos valores dos parâmetros) – antes realizadas *off-line* – podem passar a ser continuamente executadas *on-line*, caracterizando os chamados **controladores adaptativos**. O objetivo é introduzir um segundo *loop* de adaptação, reconfigurando continuamente o controlador.

Em situações com dinâmicas e perturbações ainda mais imprevisíveis, sintonizar continuamente em tempo de execução os parâmetros de um determinada lei de controle pode não ser suficiente para se obter um bom desempenho em todas as regiões de operação. Nestes casos, é necessário não somente reconfigurar continuamente os parâmetros da lei de controle mas, eventualmente, substituir a lei de controle ou estrutura do controlador por outra totalmente diferente. Denomina-se de **controlador reconfigurável** aquele que apresenta tal capacidade. Mais detalhes sobre controladores reconfiguráveis podem ser encontrados em [171, 273].

Estratégias SISO com ganho fixo. Dentre as principais leis de controle utilizadas em sistemas com única entrada e única saída (SISO), o PID (*Proportional-Integral-Derivative*) e suas variações têm tido ampla aceitação na indústria. O PID decide a atuação a ser realizada no sistema a partir de três ações distintas: a primeira é proporcional ao erro atual (valor de referência − saída medida), a segunda é proporcional à integral do erro e a terceira, proporcional à taxa de mudança (derivada) do erro. Um controlador PID é descrito, no domínio do tempo, por:

$$u[k] = K_P e[k] + K_I \sum_{i=1}^{k} e[i] + K_D(e[k] - e[k-1]) \tag{3.9}$$

[7]Também chamado de *Open-Loop Adaptive Control*.

ou, no domínio da frequência, por:

$$G(z) = K_P + K_I \cdot \frac{z}{z-1} + K_D \cdot \frac{z-1}{z} \quad (3.10)$$

Os componentes proporcional, integral e derivativo podem ser anulados ajustando-se os parâmetros K_P, K_I ou K_D para zero, respectivamente, viabilizando a utilização de variações tais como controle P, I, PI e PD. Controladores P representam uma forma simples de compensação de perturbações, mas são conhecidos pela sua impossibilidade de redução do erro em regime estacionário a zero. Se o erro $e[k]$ for zero a atuação $u[k]$ também será zero, levando à existência de erro em regime estacionário sempre que o valor de referência ou perturbação forem não-nulos. Esta situação pode ser remediada usando precompensação (somente para imprecisão devido ao valor de referência) ou utilizando a ação integral de forma conjunta à proporcional.

Controladores I atuam de forma proporcional à integral (histórico acumulado) do erro, ou seja, continuam ampliando a atuação mesmo quando o erro se mantém relativamente constante. Em função deste comportamento, são capazes de reduzir o erro em regime estacionário a zero. Por outro lado, este benefício é geralmente acompanhado de tempos de estabilização mais longos. A razão para isso é que o integrador adiciona um polo em $z = 1$ que, quando combinado com os polos do sistema original, resulta em um polo adicional mais distante da origem que os já existentes. Controladores PI buscam combinar a precisão dos controladores I com os tempos de estabilização mais baixos dos controladores P. Esta combinação é a mais prevalente na indústria atualmente.

O componente derivativo indica que a atuação deve ser mais forte quanto mais drástica for a mudança no erro percebido. Embora isso ajude a projetar controladores com reação mais rápida, este comportamento mais "agressivo e precipitado" pode aumentar o sobressinal se mal utilizado ou quando a saída medida possui fator estocástico forte. O componente derivativo nunca é utilizado sozinho, pois um erro constante implica em derivada nula e, consequentemente, atuação nula. Controladores PD podem ser utilizados para reduzir o sobressinal em sistemas com muitas oscilações quando controlados somente via controlador P. Finalmente, controladores PID podem ser utilizados quando deseja-se sistematicamente combinar as influências dos três fatores.

Vale ressaltar ainda que os componentes integral e derivativo acrescentam um novo polo, cada um, ao sistema final formado pelo sistema-alvo + controlador. Isto significa que sistemas de primeira ordem passarão a ser de segunda ordem quando controlados via PI ou PD; ou de terceira ordem quando controlados via PID. Sistemas originais de mais alta ordem, portanto, dificultam a operação de análise e são geralmente aproximados por um sistema de ordem menor. Controladores P não adicionam polos ao sistema original a ser controlado.

Uma série de técnicas de projeto de controladores PID definem diferentes métodos de obtenção dos parâmetros K_P, K_I e K_D de modo a obter as propriedades dinâmicas desejadas. As mais utilizadas são apresentadas na Seção 3.3.4.4.

Estratégias MIMO com ganho fixo. Dentre os controladores MIMO mais utilizados em sistemas descritos via modelos de espaço de estados, destacam-se: *Static State Feedback Control* (também chamado de *Full-State Feedback Control*), *Static State Feedback Control* com precompensação e *Dynamic State Feedback Control*. O objetivo do *Static State Feedback Control* é manter a saída em um valor fixo, correspondente ao estado no centro da região de operação do sistema ($\mathbf{x}[k] = 0$). Na presença de perturbações, o controlador faz com que o estado do sistema volte ao valor $\mathbf{x}[k] = 0$, de forma proporcional ao desvio observado. A lei de atuação do *Static State Feedback Control* é:

$$\mathbf{u}[k] = -\mathbf{K}\mathbf{x}[k] \quad (3.11)$$

onde a matriz $\mathbf{K}$ contém os ganhos que farão com que a atuação $-\mathbf{Kx}[k]$ leve $\mathbf{x}[k]$ de volta a 0. O comportamento final do sistema, após a inserção do controlador *Static State Feedback* é obtido através da substituição da Equação 3.11 na Equação 3.6, apresentada anteriormente:

$$\mathbf{x}[k+1] = \mathbf{Ax}[k] + \mathbf{Bu}[k] = \mathbf{Ax}[k] - \mathbf{BKx}[k] = (\mathbf{A} - \mathbf{BK})\mathbf{x}[k] \tag{3.12}$$

Note que o valor que é retroalimentado no sistema é o estado $\mathbf{x}[k]$ em vez da saída $\mathbf{y}[k]$. Os polos do sistema final são as soluções de $det\{z\mathbf{I} - (\mathbf{A} - \mathbf{BK})\} = 0$ e podem ser utilizados para analisar as propriedades de controle conforme apresentado na Tabela 3.5. Embora simples, o *Static State Feedback Control* não pode ser utilizado para rastrear um valor de referência qualquer e apresenta limitações na acomodação de perturbações.

O *Static State Feedback Control* com precompensação permite o rastreamento de um valor de referência específico $\mathbf{r}$. Seja $\mathbf{e}[k] = \mathbf{r} - \mathbf{y}[k]$ o erro de controle, $\mathbf{x}_{ss}$ o valor em regime estacionário do estado quando $\mathbf{e}[k] = 0$ e $\mathbf{u}_{ss}$ a entrada de controle associada a $\mathbf{x}_{ss}$. O objetivo é modificar a lei de controle do *Static State Feedback* de modo a obter o valor desejado da saída medida. Isto é obtido através do uso de um *offset* no vetor que representa o estado:

$$\mathbf{u}[k] = -\mathbf{K}(\mathbf{x}[k] - \mathbf{x}_{ss}) + \mathbf{u}_{ss} \tag{3.13}$$

Na prática, é realizada uma modificação do ponto central de operação para aquele que produz uma saída medida que é igual ao valor de referência. A lei de atuação do *Static State Feedback Control* com precompensação é:

$$\mathbf{u}[k] = -\mathbf{Kx}[k] + \overline{\mathbf{N}}\mathbf{r}\,; \qquad \text{onde } \overline{\mathbf{N}} = \begin{bmatrix} \mathbf{K} & 1 \end{bmatrix} \begin{bmatrix} \mathbf{A}-\mathbf{I} & \mathbf{B} \\ \mathbf{C} & 0 \end{bmatrix}^{-1} \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{3.14}$$

Static State Feedback Control com precompensação permite o rastreamento de um valor de referência específico, porém é ineficaz na rejeição de perturbações devido à ausência de um fator integrador. O *Static State Feedback Control* com precompensação é similar a um controlador P para sistemas SISO. Uma melhor acomodação de perturbações pode ser obtida com o *Dynamic State Feedback Control*, um controlador MIMO semelhante ao controlador PI para sistemas SISO.

O *Dynamic State Feedback Control* estende o vetor de estado $\mathbf{x}[k]$ de modo a incluir a integral do erro de controle $\mathbf{e}[k] = \mathbf{r} - \mathbf{y}[k]$. A integral é definida por $\mathbf{x_I}[k+1] = \mathbf{x_I}[k] + \mathbf{e}[k]$ e a lei de controle passa a ser:

$$\mathbf{u}[k] = -\begin{bmatrix} \mathbf{K_P} & \mathbf{K_I} \end{bmatrix} \begin{bmatrix} \mathbf{x}[k] \\ \mathbf{x_I}[k] \end{bmatrix} \tag{3.15}$$

O polinômio característico de um sistema controlado via *Dynamic State Feedback Control* é dado por:

$$det\left\{z\mathbf{I} - \left(\begin{bmatrix} \mathbf{A} & 0 \\ -\mathbf{C} & 1 \end{bmatrix} - \begin{bmatrix} \mathbf{B} \\ 0 \end{bmatrix} \begin{bmatrix} \mathbf{K_P} & \mathbf{K_I} \end{bmatrix}\right)\right\} \tag{3.16}$$

A escolha criteriosa de $\mathbf{K_P}$ e $\mathbf{K_I}$ determina as propriedades de controle desejadas para o sistema. Novamente, técnicas específicas para escolha dos parâmetros $\mathbf{K}$ (no *Static State Feedback Control* com e sem precompensação) e $\mathbf{K_P}$ e $\mathbf{K_I}$ (no *Dynamic State Feedback Control*) são apresentadas na Seção 3.3.4.4.

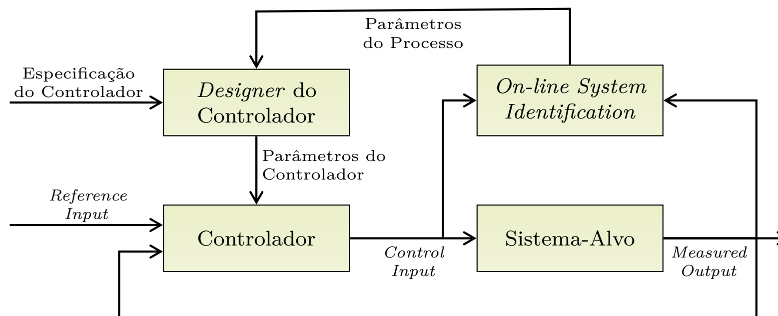
Estratégias adaptativas. Conforme mencionado anteriormente, os controladores adaptativos são utilizados quando o alto dinamismo do sistema ou das perturbações inviabilizam a utilização

de um único conjunto de valores para os parâmetros de controle. Dentre as diversas abordagens para controle adaptativo disponíveis atualmente [26, 151, 186], duas delas são comumente utilizadas na indústria.

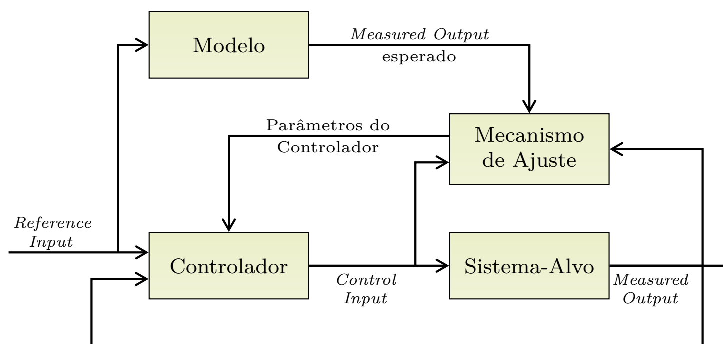
O *Model Identification Adaptive Control* (MIAC)[8], apresentado na Figura 3.14(a), é caracterizado pela transferência das atividades de identificação de sistema para o tempo de execução, onde o modelo do sistema é continuamente identificado e os parâmetros do controlador, consequentemente, modificados. O MIAC requer a derivação de fórmulas para obtenção dos parâmetros do controlador a partir dos polos desejados e dos parâmetros do sistema, sendo estes geralmente obtidos via mínimos quadrados recursivo [26] (versão *on-line* dos mínimos quadrados tradicional).

O MIAC se torna atrativo ao não requerer nenhum tipo de projeto prévio de controle visto que o ajuste de parâmetros é realizado *on-line* pelo próprio controlador. Por outro lado, o projeto manual e *off-line* de controladores abre margem para otimizações que podem gerar controladores mais eficientes do que aqueles obtidos com o MIAC. Além disso, realizar o cálculo de parâmetros a cada unidade de amostragem pode aumentar consideravelmente o *overhead* de controle e distinguir variações estocásticas de mudanças reais no sistema pode ser uma tarefa difícil.

O *Model Reference Adaptive Control* (MRAC), apresentado na Figura 3.14(b), utiliza uma abordagem diferente para realização de controle adaptativo. Em vez de tentar inferir continuamente modelos mais precisos do sistema, o MRAC utiliza um modelo de referência previamente definido, que apresenta as características que espera-se encontrar no sistema final (sistema-alvo + controlador). O MRAC verifica continuamente quão distante a saída apresentada pelo sistema real está da saída apresentada pelo modelo de referência e sintoniza o controlador de modo que o sistema final se comporte como o modelo de referência fornecido. Derivar um modelo de referência efetivo, entretanto, nem sempre é uma tarefa fácil.



(a) *Model Identification Adaptive Control* (MIAC)



(b) *Model Reference Adaptive Control* (MRAC)

Figura 3.14.: Esquemas MIAC (c) e MRAC (b) para controle adaptativo (adaptada de [26], págs. 20-21).

[8]Também conhecido como *Self-Tuning Regulator*.

Mais informações sobre o MIAC, MRAC e outras abordagens para controle adaptativo estão disponíveis em [26, 151, 186]. Estratégias mais sofisticadas de controle, não descritas neste trabalho, incluem controle estocástico (ex: controle com variância mínima), controle *fuzzy* ([138]: Capítulo 11) e *Model Predictive Control* (MPC) [55].

3.3.4.4. Técnicas para Projeto de Controladores

Uma vez escolhida a estratégia a ser utilizada na atuação de controle, é necessário especificar valores para os parâmetros que definem o controlador. Dentre as técnicas mais utilizadas destacam-se: posicionamento de polos (*pole placement*), lugar das raízes (*root locus*), LQR (*Linear-Quadratic Regulator*) e os métodos empíricos.

Posicionamento de polos. A técnica de **posicionamento de polos** parte da escolha de polos que apresentem as propriedades de controle desejadas (ex: tempo de estabilização, sobressinal, etc). A partir daí, a função de transferência é construída em função dos parâmetros do controlador, o que permite a determinação analítica dos valores de parâmetros que irão apresentar as propriedades desejadas.

Em controladores P aplicados a sistemas de primeira ordem, há apenas um parâmetro a ser identificado (K_P) e um polo a ser posicionado. Por exemplo, considere um sistema de primeira ordem associado a um controlador P cuja função de transferência em malha fechada final é:

$$F_R(z) = \frac{0.47K_P}{z - 0.43 + 0.47K_P} \tag{3.17}$$

O sistema possui um único polo $p_1 = 0.43 - 0.47K_P$. Para que o sistema final seja estável, este polo deve estar dentro do círculo unitário do plano $\mathcal{Z}$: $|p_1| = |0.43 - 0.47K_P| < 1$, ou seja, $-1.21 < K_P < 3.0$. Com base nas técnicas de análise apresentadas na Tabela 3.5, caso seja desejado um erro em regime estacionário $e_{ss} < 0.1$, tem-se que $e_{ss} = 1 - F_R(1) < 0.1$, ou seja, $K_P > 10.9$. Pode-se perceber, desde já, que o erro em regime estacionário desejado não pode ser obtido com esta configuração de controle, pois implica em um valor de K_P que está fora da faixa de estabilidade detectada.

Para obter, por exemplo, um tempo de estabilização < 10, tem-se que:

$$k_s \approx \frac{-4}{\log|0.43 - 0.47K_P|} < 10 \tag{3.18}$$

e, portanto, $-0.51 < K_P < 2.3$. Finalmente, um sobressinal < 0.1 implica que $M_P = |0.43 - 0.47K_P| < 0.1$, ou seja, $K_P < 1.1$. O sistema final possui um único polo. Este polo é, portanto, um polo real e só haverá sobressinal se este polo for negativo. Considerando as faixas de valores acima apresentadas e descartando o requisito inviável de erro em regime estacionário, verifica-se que as propriedades são atendidas quando $-0.51 < K_P < 1.1$.

Em sistemas de primeira ordem regulados via controle PI, dois polos precisam ser posicionados. Assume-se que estes polos são complexos conjugados $re^{\pm j\theta}$. A partir do tempo de estabilização desejado k_s^* e do sobressinal máximo desejado M_P^* os polos são obtidos via:

$$r = e^{-4/k_s^*} \qquad \text{e} \qquad \theta = \pi \cdot \frac{\log r}{\log M_P^*} \tag{3.19}$$

Encontrar os valores de K_P e K_I, entretanto, requer que o polinômio característico desejado $(z - re^{j\theta})(z - re^{-j\theta}) = z^2 - 2r\cos\theta z + r^2$ seja igualado ao polinômio característico modelado para o sistema. No caso de um sistema de primeira ordem controlado via PI, este polinômio é

$z-1+((K_P+K_I)z-K_P)\cdot G(z)$, onde $G(z)$ é a função de transferência do sistema não-controlado original. Ao igualar as potências de z dos dois polinômios acima obtém-se um sistema linear com duas equações e duas variáveis (K_P e K_I). Dessa forma, os valores obtidos de K_P e K_I podem então ser utilizados para verificar se o sistema final é estável, caso as demais propriedades sejam atingidas.

Em sistemas de primeira ordem regulados via controlador PID, três polos necessitam ser posicionados. A prática geral é posicionar dois deles conforme apresentado para controladores PI e localizar o terceiro de modo a não ser um polo dominante (menor que os outros dois). A derivação dos parâmetros segue de forma semelhante, caracterizado agora por um sistema linear com mais equações. Sistemas originais de mais alta ordem implicam em um número maior de polos, mas somente três deles poderão ser estrategicamente posicionados através dos valores de K_P, K_I e K_D. Nestes casos, a técnica de posicionamento de polos pode não produzir controladores aceitáveis.

A técnica de posicionamento de polos pode também ser utilizada para encontrar os parâmetros de controladores *Static State Feedback* e *Dynamic State Feedback*. Nestes casos, entretanto, a função de transferência terá n polos, onde n é o número de variáveis de estado utilizadas. Dois destes polos são estrategicamente posicionados conforme apresentado e os outros $n-2$ devem ter magnitude suficientemente pequena para serem dominados (geralmente $0.25r$).

Lugar das raízes. A técnica de posicionamento de polos permite a obtenção dos valores de parâmetros que atendem a requisitos de controle previamente definidos. Entretanto, é geralmente difícil estimar os melhores requisitos possíveis e, portanto, controladores ainda mais eficientes poderiam ser utilizados. O **lugar das raízes** é uma técnica que permite visualizar diretamente como um parâmetro influencia os polos, viabilizando a escolha daqueles valores que produzem um controlador com o melhor desempenho possível.

O lugar das raízes é um gráfico que apresenta os caminhos percorridos pelos polos à medida em que um determinado parâmetro (por exemplo, K_P) varia de zero até infinito. São apresentados n caminhos no lugar das raízes, um para cada polo do sistema. Cada caminho se inicia em um polo e termina em um zero (finito ou infinito).

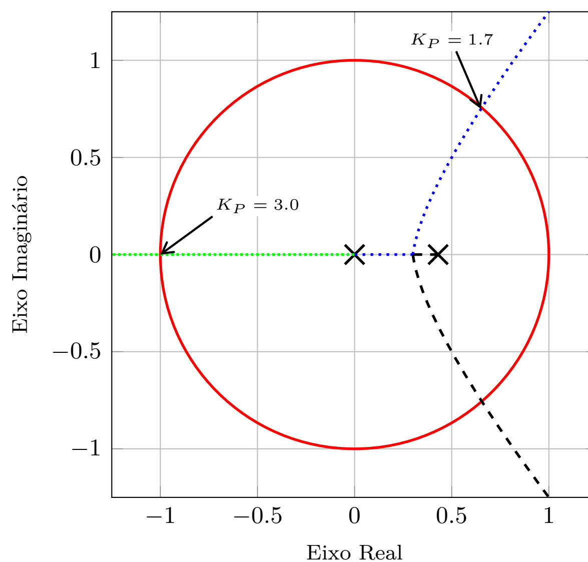


Figura 3.15.: Lugar das raízes em função do parâmetro K_P (adaptada de [138], pág. 258).

A Figura 3.15 apresenta o lugar das raízes do sistema descrito pela função de transferência:

$$G(z) = \frac{0.47}{z - 0.43} \cdot \frac{1}{z^2} \tag{3.20}$$

O sistema possui dois polos na origem do plano $\mathcal{Z}$ e um polo em 0.43. O primeiro caminho no gráfico, descrito em verde (reta com pontilhado denso), indica como um dos polos na origem varia à medida em que o valor de K_P é aumentado a partir de 0. O segundo polo, inicialmente na origem, varia conforme apresentado no caminho em azul (curva com pontilhado esparso). O terceiro polo, inicialmente na posição 0.43, segue o caminho descrito em preto (curva tracejada) conforme K_P aumenta de valor.

Note que, a partir de um determinado valor de K_P, os polos nos caminhos azul e preto passam a ser polos complexos conjugados e, portanto, seguem caminhos simétricos em relação ao eixo real. A Figura 3.15 apresenta ainda os valores de K_P nos quais o polo passa a estar exatamente na fronteira do círculo unitário. No exemplo acima, $K_P \geq 1.7$ implica em um par de polos complexos fora do círculo unitário e, portanto, o sistema passa a ser instável.

LQR. Um objetivo comum nos métodos para projeto de controladores é encontrar os valores mais altos dos ganhos, desde que o sistema final ainda mantenha a estabilidade. Valores maiores para os parâmetros do controlador geralmente produzem um menor erro em regime estacionário. Por outro lado, uma maior sensibilidade a ruídos e reações precipitadas de controle são também obtidos. Um *trade-off* está geralmente presente entre esforço de controle (quantificado pelos quadrados de $u[k]$) e erro de controle (quantificado, em modelos de espaço de estados, pelos quadrados das variáveis de estado). Reduzir o erro de controle implica em maior esforço de controle e ao reduzir o esforço de controle, geralmente obtém-se maiores erros.

A técnica **LQR** modela o problema de projeto do controlador como a minimização de uma função que envolve o uso de duas matrizes de pesos: **Q** e **R**. **Q** quantifica o custo das variáveis de estado (e combinações delas) divergirem do seu ponto inicial de operação, enquanto **R** especifica o custo do esforço de controle. A função a ser minimizada no LQR é:

$$J = \frac{1}{2}\sum_{k=0}^{\infty}(\mathbf{x}^{\mathsf{T}}[k]\mathbf{Q}\mathbf{x}[k] + \mathbf{u}^{\mathsf{T}}[k]\mathbf{R}\mathbf{u}[k]) \tag{3.21}$$

Para que J seja maior ou igual a 0 é necessário que as matrizes **Q** e **R** sejam escolhidas de forma que sejam, respectivamente, positiva semidefinida (com autovalores não-negativos) e positiva definida (com autovalores positivos). O cálculo dos ganhos **K** é geralmente realizado com o auxílio de ferramentais tais como o comando `lqr` no Scilab [270] ou `dlqr` no MATLAB [210].

Métodos empíricos. A representação de sistemas através dos modelos paramétricos de aproximação descritos na Seção 3.3.4.2 viabiliza a derivação de técnicas de projeto de controladores que utilizam diretamente os parâmetros R (ou $a = RL/T$), L e T para encontrar os parâmetros K_P, K_I e K_D do controlador. Um conjunto extenso de diferentes métodos empíricos para projeto de controladores está disponível na literatura [312, 324]. Cada método produz controladores com diferentes comportamentos ou mais adequados, por exemplo, a rastreamento de referência ou rejeição de perturbações.

Dentre os métodos empíricos mais utilizados, destacam-se o Ziegler-Nichols, CHR (*Chien-Hrones-Reswick*) e CC (*Cohen-Coon*). O método de Ziegler-Nichols, apresentado na Tabela 3.6, é utilizado para projetar controladores estáveis e também precisos nos casos PI e PID. Entretanto, oscilações ainda estão presentes mesmo no controlador PID. O método de Ziegler-Nichols é também caracterizado por gerar controladores que apresentam uma taxa de decaimento (razão entre as amplitudes do primeiro e segundo sobressinal) de 1/4.

Tipo do Controlador	K_P	$T_i\,(K_P/K_I)$	$T_d\,(K_D/K_P)$
P	$1/a$		
PI	$0.9/a$	$3L$	
PID	$1.2/a$	$2L$	$L/2$

Tabela 3.6.: Método empírico de Ziegler-Nichols (adaptada de [324]).

META: rastreamento de valor de referência						
	com 0% de sobressinal			**com 20% de sobressinal**		
Tipo do Controlador	K_P	T_i	T_d	K_P	T_i	T_d
P	$0.3/a$			$0.7/a$		
PI	$0.35/a$	$1.2T$		$0.6/a$	T	
PID	$0.6/a$	T	$0.5L$	$0.95/a$	$1.4T$	$0.47L$

META: rejeição de perturbações						
	com 0% de sobressinal			**com 20% de sobressinal**		
Tipo do Controlador	K_P	T_i	T_d	K_P	T_i	T_d
P	$0.3/a$			$0.7/a$		
PI	$0.6/a$	$4L$		$0.7/a$	$2.3L$	
PID	$0.95/a$	$2.4L$	$0.42L$	$1.2/a$	$2L$	$0.42L$

Tabela 3.7.: Método empírico de Chien-Hrones-Reswick (adaptada de [324]).

O método CHR, apresentado na Tabela 3.7, possibilita o projeto de controladores com dois objetivos diferentes e com duas possibilidades de *trade-off* entre tempo de estabilização e sobressinal. Controladores podem ser criados com o objetivo de rastreamento de valor de referência ou rejeição de perturbações (supondo que o valor de referência é mantido em um valor fixo). Em cada uma dessas duas categorias, pode-se ainda decidir entre 0% ou 20% de sobressinal.

Tipo do Controlador	K_P	T_i	T_d
P	$\frac{1.00}{a}\left(1+\frac{0.35\tau}{1-\tau}\right)$		
PI	$\frac{0.90}{a}\left(1+\frac{0.92\tau}{1-\tau}\right)$	$\frac{3.3-3\tau}{1+1.2\tau}\cdot L$	
PD	$\frac{1.24}{a}\left(1+\frac{0.13\tau}{1-\tau}\right)$		$\frac{0.27-0.36\tau}{1-0.87\tau}\cdot L$
PID	$\frac{1.35}{a}\left(1+\frac{0.18\tau}{1-\tau}\right)$	$\frac{2.5-2\tau}{1-0.39\tau}\cdot L$	$\frac{0.37-0.37\tau}{1-0.81\tau}\cdot L$

Tabela 3.8.: Método empírico de Cohen-Coon (adaptada de [324]).

Finalmente, o método CC, apresentado na Tabela 3.8, possibilita o projeto de controladores P, PI, PD e PID. O método define os valores do controlador em função do parâmetro L e de um novo parâmetro $\tau = L/(L+1)$. O método CC é mais adequado para controle de processos com *dead time* expressivo (altos valores de L) e frequentemente gera respostas mais rápidas que os demais.

3.4. Discussão Final

Este capítulo apresentou as principais motivações para o projeto de sistemas computacionais que apresentem capacidades de *self-adaptation*; os fundamentos e principais dimensões do espaço de problema e espaço de solução neste domínio; e as quatro principais escolas prospectoras de abordagens para projeto e implementação de sistemas *self-adaptive*. Dentre estas escolas, as abordagens baseadas em Teoria de Controle foram discutidas em maior profundidade devido ao papel central que desempenham neste trabalho.

Sem o objetivo de esgotar completamente o assunto ou servir de referência básica em tais tópicos, este capítulo tem, entretanto, a tarefa de evidenciar ao leitor a ampla riqueza (e consequente complexidade) do domínio de sistemas *self-adaptive*. Mesmo ao limitar-se a uma determinada escola, tal complexidade é ainda presente. A ausência de mecanismos mais disciplinados e sistematizados para organização de conhecimento de projeto e para geração e avaliação facilitadas de alternativas de projeto é fator impeditivo para uma maior adoção e completa realização dos sistemas *self-adaptive* na indústria de *software*.

Conhecer as diferentes problemáticas, metas, estratégias e consequências da adoção de uma determinada abordagem para implementação de comportamento *self-adaptive* requer, atualmente, uma formação especializada que é demorada, custosa e que demanda um amplo conjunto de pré-requisitos. Ao mesmo tempo em que parte desta dificuldade é característica de um domínio inerentemente complexo, acredita-se que outra parcela é decorrente da falta de organização sistemática de conhecimento especializado que, se disponível, as barreiras conceituais e técnicas entre o engenheiro de *software* e o engenheiro de controle poderiam ser minimizadas. Este cenário é o ponto central das motivações deste trabalho.

Capítulo 4

Otimização Multiobjetivo

Better a diamond with a flaw than a pebble without.
Confucius (Analects)

A utilização de tecnologias de busca e otimização tem sido fator determinante para o sucesso dos sistemas de apoio a decisão em áreas tais como escalonamento de veículos de transporte, bioinformática, geração de escalas de trabalho, apoio a decisão médica, dentre outros. Esta área de pesquisa é formada por contribuições de diferentes comunidades, tais como Pesquisa Operacional, Inteligência Artificial e Ciência do Gerenciamento[1] (*Management Science*).

Um amplo conjunto de técnicas de otimização está disponível atualmente [53], variando de acordo com o número de objetivos (único ou múltiplos), tempo de convergência, susceptibilidade a máximos locais, premissas assumidas sobre o problema, suporte ou não a restrições, execução determinística ou não-determinística, uso ou não de populações, dentre outros critérios. Escolher a técnica mais adequada para o problema em questão é fundamental para a obtenção de boas soluções.

Este capítulo apresenta os fundamentos sobre otimização multiobjetivo [80] e a sua importância como infraestrutura básica para automação de processos de engenharia de *software*. Inicialmente, os conceitos fundamentais para definição de problemas e soluções multiobjetivo são apresentados, seguidos das principais técnicas para resolução de tais problemas. Em seguida, as motivações para utilização de abordagens evolutivas e elitistas são apresentadas. Em particular, os fundamentos sobre algoritmos genéticos e o algoritmo evolucionário NSGA-II são apresentados. Ao final do capítulo, as principais métricas de desempenho de algoritmos de otimização multiobjetivo são discutidas. A Figura 4.1 apresenta o roteiro deste capítulo.

4.1. Otimização

Um **problema de otimização** é informalmente definido por Kalyanmoy Deb [80] como:

Definição 4.1: Otimização (Deb, 2001, pág. 1)

Otimização é o processo de descoberta de uma ou mais soluções viáveis (*feasible*), que correspondem aos valores extremos de um ou mais objetivos.

[1] A ciência dos processos operacionais, da tomada de decisão e do gerenciamento.

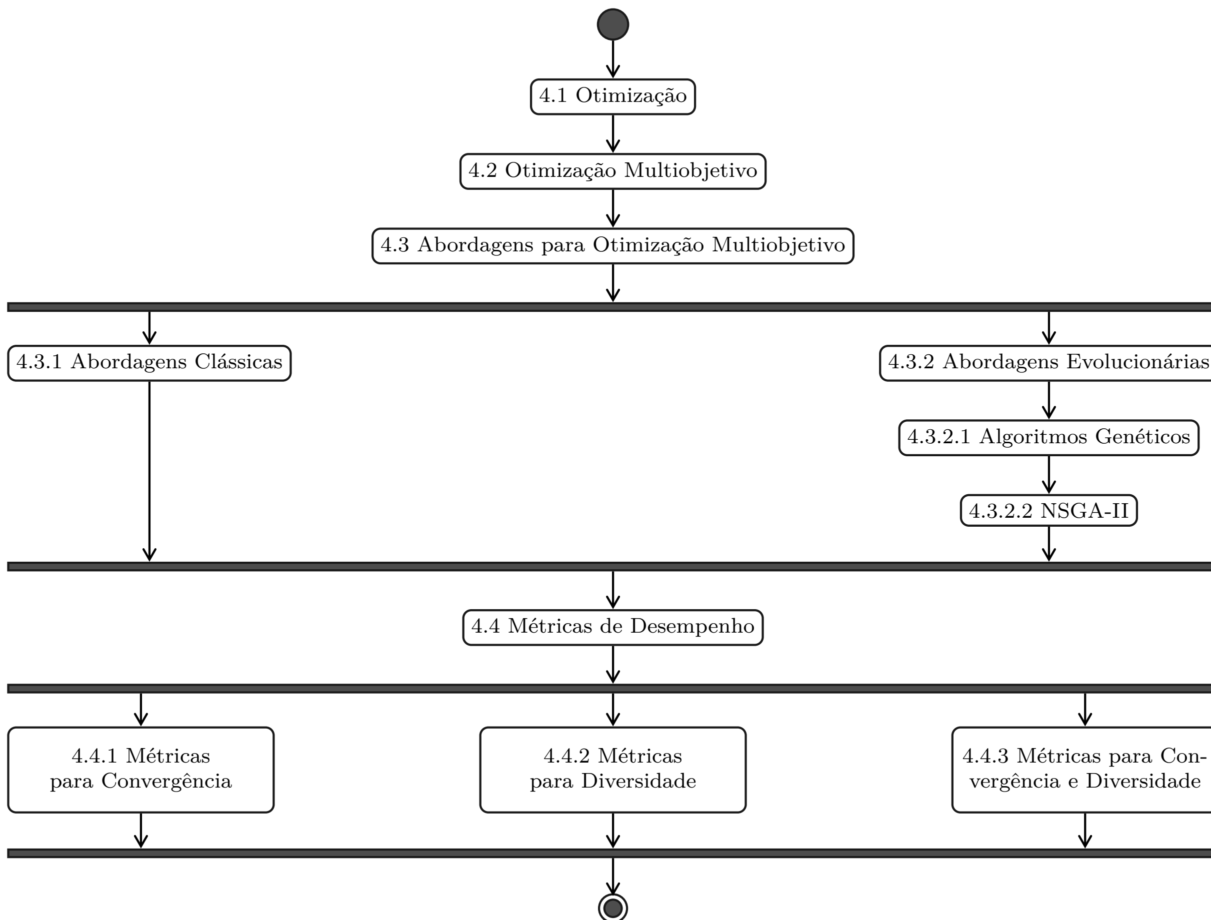


Figura 4.1.: Roteiro do capítulo 4.

Ao longo das últimas décadas, tais problemas vêm sendo classificados de acordo com as premissas assumidas sobre sua estrutura e técnicas adequadas a cada classe de problemas vêm sendo propostas. Uma classe bastante ampla de problemas de otimização são os **problemas de programação não-linear**:

Definição 4.2: Problema de Programação Não-Linear (extensão de Papadimitriou & Steiglitz [237], 1998, pág. 1)

$$\begin{aligned} \text{Minimizar} \quad & f(x); & \\ \text{Sujeito a} \quad & g_j(x) \geq 0, & j = 1, 2, \ldots, J; \\ & h_k(x) = 0, & k = 1, 2, \ldots, K; \\ & x_i^{(L)} \leq x_i \leq x_i^{(U)}, & i = 1, 2, \ldots, n. \end{aligned}$$

A função $f(x)$ é denominada **função-objetivo** e uma solução $\mathbf{x}$ é um vetor de n variáveis de decisão: $x = (x_1, x_2, \ldots, x_n)^\intercal$. Os termos $g_j(x)$ e $h_k(x)$ são denominados **funções de restrição** (restrições de desigualdade e igualdade, respectivamente), enquanto os valores $x_i^{(L)}$ e $x_i^{(U)}$ especificam os limites inferior e superior, respectivamente, de cada variável de decisão x_i. Tais limites definem o **espaço de decisão** (*decision space*) $\mathcal{D}$. Uma solução x que não satisfaz as $J + K$ restrições e os $2n$ limites de variáveis de decisão é denominada **solução inviável** (*infeasible solution*), caso contrário é uma **solução viável** (*feasible solution*). O conjunto de todas as soluções viáveis em $\mathcal{D}$ define o que é chamado de **região viável** (*feasible region*) ou **espaço de busca** (*search space*) $\mathcal{F}$.

A definição de solução viável pode ser compreendida, por exemplo, como uma solução para o problema de posicionamento de um número de figuras geométricas em um retângulo maior, de modo que não haja sobreposição das figuras e que todas elas estejam completamente contidas no interior do retângulo (utilizado, por exemplo, para realizar cortes de tecidos ou de peças de metal). Caso haja sobreposição ou o retângulo não comporte todas as figuras então a solução é inviável.

Quando a função-objetivo e as funções de restrição são todas lineares, tem-se um **problema de programação linear**. Tais problemas apresentam certas propriedades teóricas que viabilizam, por exemplo, a elaboração de provas de convergência dos algoritmos de otimização propostos. Visto que muitos problemas reais são não-lineares, entretanto, técnicas genéricas – aplicáveis também a problemas não-lineares – são frequentemente propostas.

Muitos algoritmos de otimização são desenvolvidos particularmente para resolver problemas de minimização, conforme definido acima. Quando deseja-se maximizar o objetivo, entretanto, pode-se transformar este problema em um problema de minimização através da multiplicação da função objetivo por -1 (princípio da dualidade [77]) e então aplica-se o algoritmo original.

Problemas simples de otimização, caracterizados por um número pequeno de possíveis soluções, podem ser facilmente solucionados por busca exaustiva: análise de todas as soluções possíveis de modo a retornar a melhor solução. A maioria dos problemas reais, entretanto, são caracterizados por espaços de busca enormes, motivando a utilização de técnicas não-determinísticas e/ou baseadas em heurísticas.

Uma **heurística** é definida por Reeves [255] como:

Definição 4.3: Heurística (Heeves, 1993, pág. 5)

Uma heurística é uma estratégia que guia o processo de busca por boas (*near-optimal*) soluções a um custo computacional razoável, sem, entretanto, garantir que as soluções encontradas são globalmente ótimas ou viáveis. Infelizmente, pode ainda não ser possível afirmar quão próxima à solução globalmente ótima a solução encontrada pela heurística está.

As heurísticas são, em geral, projetadas para um problema em particular e frequentemente sofrem de susceptibilidade a mínimos locais. As **metaheurísticas** [109], por outro lado, são técnicas independentes de problema – e, portanto, passíveis de serem utilizadas como mecanismos genéricos de otimização – que combinam procedimentos de melhoria local com estratégias de mais alto nível, de modo a evitar os mínimos locais. Uma série de métodos heurísticos/metaheurísticos para otimização está atualmente disponível na literatura [53, 109], tais como *Simulated Annealing*, Algoritmos Genéticos, Programação Genética, Otimização por Colônias de Formigas, *Scatter Search* e *Tabu Search*.

Um grau maior de sofisticação e flexibilidade pode ainda ser obtido através do uso de **hiper-heurísticas** [51]. As hiper-heurísticas, diferente das heurísticas e metaheurísticas, não constituem estratégias de busca propriamente ditas. O termo é utilizado para informar que o espaço de decisão utilizado na busca é composto por heurísticas. O problema de otimização passa então a ser a obtenção de uma heurística (localmente) ótima, que será subsequentemente utilizada no problema principal de otimização sendo resolvido. Mais informações sobre hiper-heurísticas podem ser encontradas em [53] (Capítulo 11).

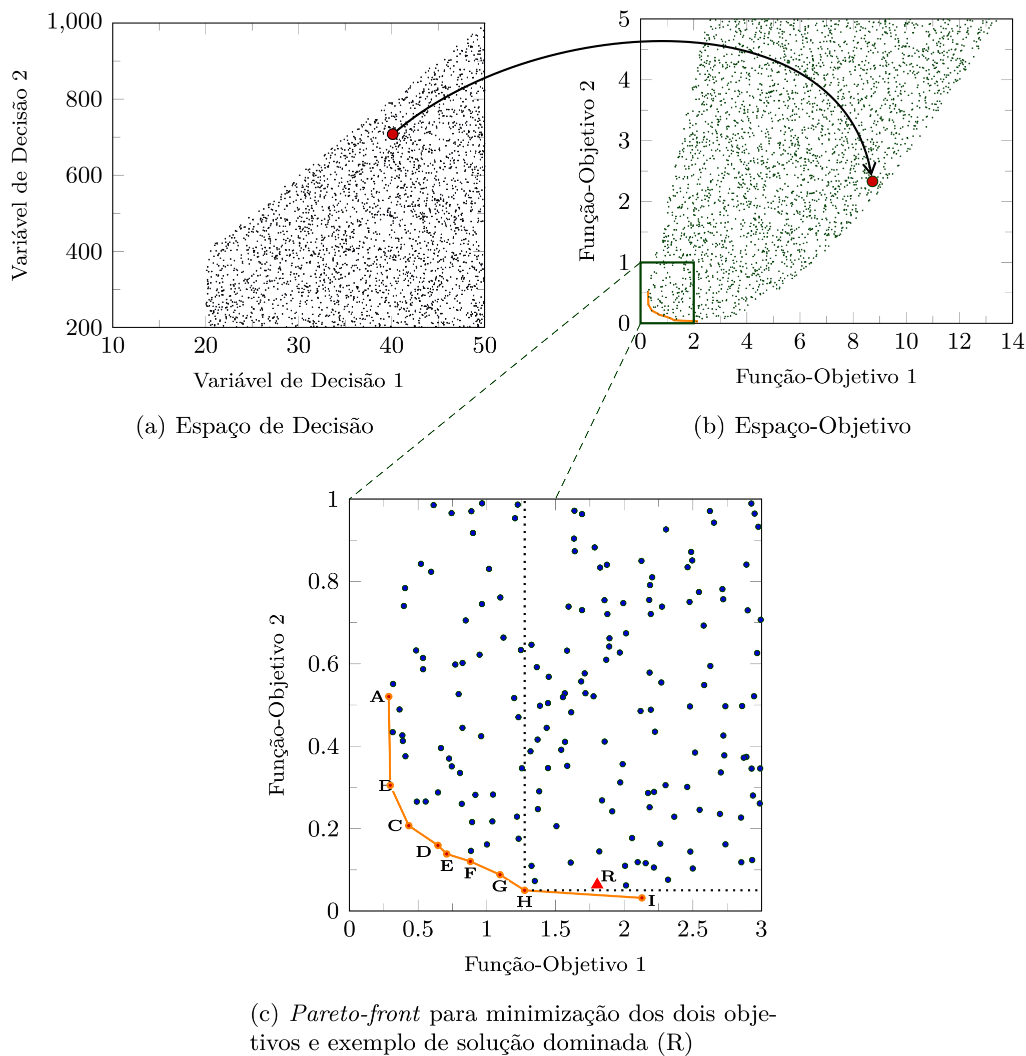
4.2. Otimização Multiobjetivo

Um **problema de otimização multiobjetivo** é definido por Deb [80] como:

Definição 4.4: Problema de Otimização Multiobjetivo (Deb, 2001, pág. 13)

$$\begin{array}{rlr} \text{Minimizar/Maximizar} & f_m(x) & m = 1, 2, \ldots, M; \\ \text{Sujeito a} & g_j(x) \geq 0, & j = 1, 2, \ldots, J; \\ & h_k(x) = 0, & k = 1, 2, \ldots, K; \\ & x_i^{(L)} \leq x_i \leq x_i^{(U)}, & i = 1, 2, \ldots, n. \end{array}$$

Neste caso, tem-se M funções-objetivo $f(x) = (f_1(x), f_2(x), \ldots, f_M(x))^\intercal$ e cada uma delas pode ser maximizada ou minimizada. O princípio da dualidade é mais uma vez aplicado para facilitar a manipulação de tipos combinados de objetivos (maximização e minimização). Uma característica importante dos problemas de otimização multiobjetivo é que – além da existência do espaço de decisão $\mathcal{D}$ (Figura 4.2(a)) – a presença de múltiplas funções-objetivo dá origem ao que é chamado de **espaço-objetivo** (*objective space*) $\mathcal{O}$ (Figura 4.2(b)).



(a) Espaço de Decisão

(b) Espaço-Objetivo

(c) *Pareto-front* para minimização dos dois objetivos e exemplo de solução dominada (R)

Figura 4.2.: Espaço de decisão viável (a), espaço-objetivo viável (b) e exemplos de *Pareto-front* e de solução dominada (c).

Conforme apresentado na Figura 4.2, para cada solução x no espaço de decisão existe um ponto correspondente no espaço-objetivo, denotado por $f(x) = z = (z_1, z_2, \ldots, z_M)^\intercal$. Ou seja, o problema de otimização define um mapeamento entre um vetor-solução n-dimensional e um vetor-objetivo M-dimensional. Otimização multiobjetivo é comumente chamada de otimização

de vetor devido ao uso de um vetor de objetivos em vez de um único valor.

Uma outra classe bastante popular de problemas de otimização são os **problemas convexos de otimização multiobjetivo**:

Definição 4.5: Problema Convexo de Otimização Multiobjetivo (Deb, 2001, pág. 16)

Um problema de otimização multiobjetivo é convexo se todas as funções-objetivo são convexas e a região viável é também convexa (ou seja, todas as restrições de desigualdade são não-convexas e todas as restrições de igualdade são lineares).

Uma função $f\colon \mathbb{R}^n \to \mathbb{R}$ é convexa se, para qualquer par de soluções $x^{(1)}, x^{(2)} \in \mathbb{R}^n$, a seguinte condição é verdadeira:

$$f(\lambda x^{(1)} + (1-\lambda)x^{(2)}) \leq \lambda f(x^{(1)}) + (1-\lambda) f(x^{(2)}); \quad \text{para todo } 0 \leq \lambda \leq 1.$$

A convexidade de um problema de otimização multiobjetivo é uma característica importante visto que muitos algoritmos são bastante eficientes quando aplicados em problemas convexos. Vale ressaltar que os dois espaços, de decisão e objetivo, devem ter suas convexidades analisadas, embora problemas com espaços de decisão não-convexos podem ainda apresentar espaços-objetivo convexos.

Problemas de otimização multiobjetivo são caracterizados pela natureza frequentemente conflitante das suas funções-objetivo. Por exemplo, otimizar a compra de um automóvel de modo a minimizar o custo e maximizar o conforto; ou a compra de uma passagem aérea minimizando o custo e também minimizando o número de escalas, apresentam funções-objetivo conflitantes.

Nestes casos, é impossível melhorar o atendimento de uma função-objetivo sem prejudicar o atendimento de outra. Consequentemente, não existe uma única solução que seja ótima em relação a todas as funções-objetivo. A resolução de tais problemas envolve a descoberta de um conjunto de soluções – todas ótimas – que descrevem um *trade-off* (solução de compromisso) no atendimento dos diversos objetivos.

A inexistência de uma solução ótima para todos os objetivos requer que algum **mecanismo de articulação de preferência** (entre objetivos) seja aplicado. As abordagens atuais utilizam dois mecanismos diferentes para este propósito [80] (pág. 4):

- Articulação *a priori* de preferência (*preference-based multi-objective optimization procedure*): utilizada quando um **vetor de fatores de preferência** relativos entre os objetivos é conhecido para o problema em questão. Deriva-se, então, uma função-objetivo única, composta por uma soma ponderada das diversas funções-objetivo originais. O peso utilizado para cada função-objetivo original é proporcional ao fator de preferência atribuído àquela função-objetivo, reduzindo o problema a um problema de otimização com objetivo único. A repetição do problema de otimização com diferentes vetores de fatores produz soluções referentes ao *trade-off* em questão.
- Articulação *a posteriori* de preferência (*ideal multi-objective optimization procedure*): baseia-se na obtenção de múltiplas soluções representativas do *trade-off*, considerando todos os objetivos como igualmente importantes. Após a obtenção deste conjunto – e melhor subsidiado com informações trazidas pelas soluções encontradas – o usuário utiliza, então, critérios qualitativos de mais alto nível para escolher a solução ótima a ser utilizada.

Estratégias com articulação *a priori* de preferência dependem da escolha correta dos fatores; uma atividade que é altamente subjetiva, não-técnica, orientada a experiências passadas e dependente do problema em questão. Adicionalmente, não é qualquer vetor de fatores que é capaz de produzir uma solução que é ótima em relação ao *trade-off* (solução não-dominada, conforme explicado a seguir).

Nas abordagens com articulação *a posteriori*, este conhecimento subjetivo e não-técnico é também utilizado, porém na escolha de uma única solução dentre um conjunto de soluções já reconhecidamente ótimas (em contraste à escolha de uma solução em todo o espaço de busca). É uma distinção sutil, porém importante.

Por outro lado, embora as estratégias com articulação *a posteriori* representem soluções mais metódicas, mais práticas e menos subjetivas, abordagens com articulação *a priori* podem ser interessantes quando o problema é suficientemente conhecido a ponto de ser possível a derivação de um vetor confiável de fatores de preferência. Em função da utilização, neste trabalho, de técnicas de otimização multiobjetivo em problemas genéricos de projeto de arquiteturas de *software* – onde nenhuma premissa particular sobre preferências é assumida – tratar-se-á a partir daqui somente as abordagens com articulação *a posteriori* de preferências.

4.2.1. Dominância e Pareto-Optimality

A obtenção das soluções ótimas que descrevem o *trade-off* de atendimento das funções-objetivo baseia-se no conceito de dominância entre soluções. Seja $i \lhd j$ a notação utilizada para informar que a solução i é melhor que a solução j, para um objetivo particular (produz valores menores quando a meta é minimizar o objetivo, ou valores maiores caso contrário). Assumindo a existência de M funções-objetivo, pode-se então apresentar a definição de **dominância** entre soluções:

Definição 4.6: Dominância (Deb, 2001, pág. 28)

Uma solução $x^{(1)}$ domina uma outra solução $x^{(2)}$ ($x^{(1)} \preceq x^{(2)}$) se ambas as condições 1 e 2 forem satisfeitas:

1. A solução $x^{(1)}$ é não-pior que $x^{(2)}$ em todos os objetivos, ou seja, $f_j(x^{(1)}) \ntriangleright f_j(x^{(2)})$; para todo $j \in \{1, 2, \ldots, M\}$.
2. A solução $x^{(1)}$ é estritamente melhor que $x^{(2)}$ em ao menos um objetivo, ou seja, $f_j(x^{(1)}) \lhd f_j(x^{(2)})$; para ao menos um $j \in \{1, 2, \ldots, M\}$.

Caso alguma das duas condições acima seja violada, então $x^{(1)}$ não domina $x^{(2)}$. Isto não quer dizer, entretanto, que $x^{(2)}$ domina $x^{(1)}$ pois ambas podem ser não-dominadas entre si. Por exemplo, na Figura 4.2(c), a solução H domina a solução R pois H é não-pior (na verdade, melhor) que R em todos os objetivos e é também estritamente melhor que R em ao menos um objetivo (na verdade, em ambos).

Considerando agora as soluções C e D, não pode-se afirmar que C domina D, pois C não é não-pior que D em todos os objetivos (é pior que D em relação ao objetivo 2). Tampouco pode-se afirmar que D domina C, pois D é pior que C em relação ao objetivo 1. Neste caso, diz-se que as soluções C e D são **não-dominadas** entre si.

Definição 4.7: Conjunto de soluções não-dominadas (Deb, 2001, pág. 31)

Em um conjunto de soluções P, o conjunto de soluções não-dominadas P' é formado por aquelas soluções de P que não são dominadas por qualquer solução presente em P.

Visto que o conceito de dominância estabelece uma relação que não é reflexiva e é assimétrica e transitiva, uma relação de ordem parcial estrita é por ela determinada. Consequentemente, um conjunto de soluções P pode ser particionado em dois subconjuntos: P' – formado por todas

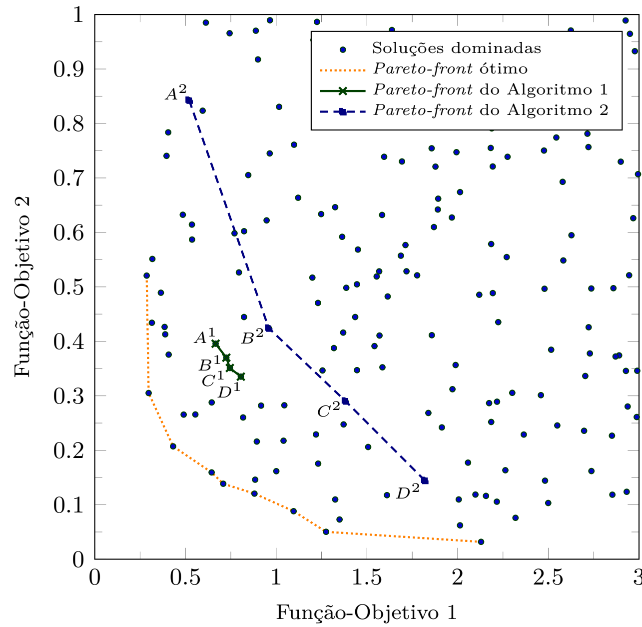


Figura 4.3.: Soluções com diferentes graus de convergência e diversidade.

as soluções mutualmente não-dominadas em P; e P'' – formado pelas soluções de P que são dominadas por pelo menos um elemento de P'.

Na Figura 4.2(c), as soluções de A a I são mutualmente não-dominadas e compõem o conjunto P'. Considerando ambos os objetivos como importantes, nada pode-se afirmar sobre qual solução é melhor dentre duas quaisquer selecionadas de P'. Todas as outras soluções apresentadas na figura são dominadas por pelo menos uma das soluções de A a I.

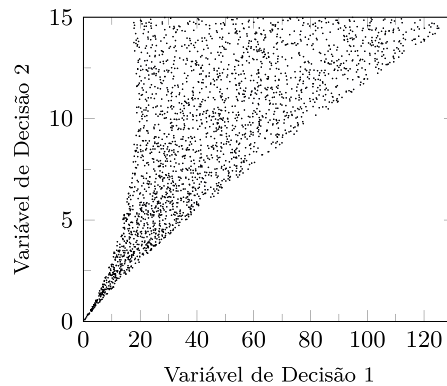


Figura 4.4.: Exemplo de problema de otimização multiobjetivo com objetivos não-conflitantes.

Quando o conjunto de soluções P é todo o espaço viável de busca ($P = \mathcal{F}$), o conjunto de soluções não-dominadas P' é denominado **conjunto Pareto-ótimo**. A curva formada pela ligação das soluções em P' é denominada **Pareto-optimal front** (ou simplesmente **Pareto-front**).

Adicionalmente, um conjunto Pareto-ótimo pode ser **global** ou **local**:

Definição 4.8: Conjunto Pareto-Ótimo Global (Deb, 2001, pág. 31)

O conjunto formado pelas soluções não-dominadas de todo o espaço viável de busca $\mathcal{F}$ é denominado conjunto Pareto-ótimo global.

Definição 4.9: Conjunto Pareto-Ótimo Local (Deb, 2001, pág. 31)

Se, para cada membro x em um conjunto $\underline{P}$, não existe nenhuma solução y (na vizinhança de x, tal que $\|y - x\|_\infty \leq \epsilon$, onde ϵ é um número positivo pequeno) dominando qualquer solução em $\underline{P}$, então $\underline{P}$ é um conjunto Pareto-ótimo local.

Problemas que apresentam conjuntos Pareto-ótimo locais podem trazer dificuldades aos algoritmos de otimização multiobjetivo, pois mecanismos para o escape de ótimos locais devem ser utilizados. Note também que todo conjunto Pareto-ótimo global, por definição, também é um conjunto Pareto-ótimo local.

Diferente de um problema de otimização com objetivo único, um bom algoritmo de otimização multiobjetivo apresenta duas características fundamentais: *i)* obtenção de um conjunto de soluções que esteja o mais próximo possível do conjunto Pareto-ótimo global (alto grau de **convergência**); e *ii)* maximização do espalhamento das soluções não-dominadas encontradas (alta **diversidade**).

Conforme apresentado na Figura 4.3, um determinado algoritmo 1 pode obter um conjunto de soluções não-dominadas que é satisfatoriamente próximo do *Pareto-front* global, porém apresenta baixa diversidade pois não apresenta soluções igualmente espaçadas ao longo do *Pareto-front*. Por outro lado, o algoritmo 2 apresenta uma melhor diversidade de soluções do que o algoritmo 1, porém a convergência para o *Pareto-front* global é mais baixa. Soluções com alta diversidade são interessantes para melhor elucidar as características do *trade-off* envolvido na otimização multiobjetivo.

Algoritmo 4.1: Algoritmo simples para obtenção das soluções não-dominadas.

```
 1: procedure naiveNDSet(P)                 ▷ P = conjunto original de soluções
 2:     P' ← ∅                              ▷ P' = conjunto de soluções não-dominadas
 3:     for all k ∈ P do
 4:         kIsDominated ← false
 5:         for all l ∈ P | l ≠ k do
 6:             if l ≼ k then                ▷ Se l domina k
 7:                 kIsDominated ← true
 8:                 break
 9:         if !kIsDominated then
10:             P' ← P' ∪ {k}
11:     return P'
```

Embora a obtenção de conjuntos Pareto-ótimo globais de maior cardinalidade e com soluções bem espalhadas seja o objetivo da maioria dos problemas de otimização, alguns deles apresentam objetivos não-conflitantes, o que implica na existência de apenas uma solução não-dominada, conforme ilustrado na Figura 4.4.

Diversos algoritmos para o particionamento de um conjunto de soluções P em subconjuntos P' (soluções não-dominadas) e P'' (soluções dominadas) estão disponíveis na literatura. A abordagem mais simples e ineficiente, apresentada no Algoritmo 4.1, consiste em verificar se cada solução $P^{(k)}$, presente no conjunto, é dominada por alguma outra solução $P^{(l)}$ do mesmo conjunto. Caso alguma solução domine a solução $P^{(k)}$, então existe alguma solução em P que é melhor que $P^{(k)}$ em todos os objetivos e, portanto, $P^{(k)}$ não fará parte do conjunto de soluções não-dominadas. Caso contrário, a solução $P^{(k)}$ é incluída no conjunto de soluções não-dominadas. Para um conjunto P de cardinalidade N e M funções-objetivo sendo consideradas, é evidente a constatação que este algoritmo apresenta complexidade $O(MN^2)$, no pior caso.

Uma abordagem mais eficiente, apresentada no Algoritmo 4.2, consiste na comparação de cada solução de P com um conjunto P' parcialmente preenchido e continuamente atualizado. Inici-

almente, inclui-se a primeira solução de P ($P^{(1)}$) no conjunto P'. Cada outra solução $P^{(k)}$ de P – da segunda até a última – é então comparada com cada solução $P^{(l)}$ em P'. Toda solução $P^{(l)}$ que é dominada por $P^{(k)}$ é removida de P'. Se $P^{(k)}$ é dominada por alguma solução $P^{(l)}$, parte-se então para a próxima solução $P^{(k)}$. Caso $P^{(k)}$ não seja dominada por nenhuma solução $P^{(l)}$, $P^{(k)}$ é então incluída em P'. Repete-se o processo para as outras soluções $P^{(k)}$ de P. Ao final, P' irá conter as soluções não-dominadas de todo o conjunto P.

Algoritmo 4.2: Algoritmo para obtenção das soluções não-dominadas com atualização contínua.

```
 1: procedure continuousUpdateNDSet(P)
 2:     P' ← {P^(1)}
 3:     for k ← P^(2), ..., P^(N) do                    ▷ N=|P|
 4:         kIsDominated ← false
 5:         for all l ∈ P' do
 6:             if k ⪯ l then                           ▷ Se k domina l
 7:                 P' ← P' \ l                         ▷ Remova l de P'
 8:             if l ⪯ k then                           ▷ Se l domina k
 9:                 kIsDominated ← true
10:                 break
11:         if !kIsDominated then
12:             P' ← P' ∪ {k}
13:     return P'
```

Note que o segundo elemento de P é comparado com uma única solução em P', o terceiro é comparado com no máximo duas soluções em P' e assim sucessivamente. No pior caso (todas as soluções de P não-dominadas), serão necessárias $1+2+\ldots+(N-1) = N(N-1)/2$ verificações de dominância. Embora esta complexidade também seja $O(MN^2)$, o número real de operações é cerca de metade do número de operações realizadas no Algoritmo 4.1. Visto que a cardinalidade de P' pode diminuir frequentemente e que uma solução $P^{(k)}$ pode ser dominada pelos primeiros elementos de P', a complexidade prática real é geralmente menor que a apresentada acima.

Na abordagem proposta por Kung et al. [182], apresentada no Algoritmo 4.3, o conjunto inicial de soluções P é ordenado de forma descendente em relação ao atendimento da primeira função-objetivo. A partir daí, o conjunto é recursivamente particionado nas metades superior (T: *top*) e inferior (B: *bottom*), até que cada partição contenha apenas um elemento.

A recursão conclui verificando se cada elemento de B é não-dominado em relação a todos os elementos de T. Em caso afirmativo, inclui-se o elemento de B em um conjunto G que é o *merge* de T com as soluções não-dominadas de B. Kung et al. demonstraram que a complexidade desta abordagem é $O(N \log N)$ quando o número de funções-objetivo M é 2 ou 3. Para $M \geq 4$, a complexidade passa a ser $O(N(\log N)^{M-2})$.

4.3. Principais Abordagens para Otimização Multiobjetivo

A Seção 4.2.1 apresentou o conceito de dominância e os principais algoritmos para obtenção das soluções não-dominadas de um conjunto. Em função dos enormes espaços de busca que frequentemente caracterizam os problemas reais de otimização multiobjetivo, a aplicação dos algoritmos acima apresentados em soluções obtidas via busca exaustiva é obviamente inviável. Esta seção discute as principais abordagens para obtenção de *Pareto-fronts*, preferencialmente aqueles com alto grau de convergência ao *Pareto-front* global e formados por soluções que apresentem alta diversidade.

Algoritmo 4.3: Algoritmo de Kung et al. para obtenção das soluções não-dominadas.

1: **procedure** KUNGNDSET(P, f)
2: $PS \leftarrow sort(P, <_{f(1)})$ ▷ Ordena em relação à primeira função-objetivo
3: **return** FRONT(PS)

4: **procedure** FRONT(PS)
5: **if** $|PS| = 1$ **then**
6: **return** PS
7: **else**
8: $T =$ FRONT($\{PS^{(1)}, PS^{(2)}, \ldots, PS^{(|P|/2)}\}$) ▷ Partição *Top*
9: $B =$ FRONT($\{PS^{(|P|/2+1)}, PS^{(|P|/2+2)}, \ldots, PS^{(|P|)}\}$) ▷ Partição *Bottom*
10: $G \leftarrow T$ ▷ G=*Merge* de T com as soluções não-dominadas de B
11: **for all** $b \in B$ **do**
12: $bIsDominated \leftarrow false$
13: **for all** $t \in T$ **do**
14: **if** $t \preceq b$ **then** ▷ Se t domina b
15: $bIsDominated \leftarrow true$
16: **break**
17: **if** $!bIsDominated$ **then**
18: $G \leftarrow G \cup \{b\}$
19: **return** G

Uma das formas de classificação dos algoritmos para otimização multiobjetivo é em relação ao número de soluções encontradas por cada execução da solução. Segundo este critério, as abordagens são classificadas como clássicas ou evolucionárias.

Nas abordagens clássicas, o problema é reduzido a um problema de objetivo único através de uma parametrização das funções-objetivo. Cada execução – geralmente determinística e caracterizada por valores específicos de parâmetros – possui como objetivo encontrar uma solução particular do *Pareto-front*. Dessa forma, diferentes execuções – realizadas com valores distintos de parâmetros – dão origem às soluções que compõem o *Pareto-front* do problema. As abordagens evolucionárias, por outro lado, utilizam populações de soluções, viabilizando a descoberta de múltiplos componentes do *Pareto-front* em uma única execução. Adicionalmente, o uso de múltiplas soluções viabiliza o uso de algoritmos intencionalmente projetados para obtenção de *Pareto-fronts* com alta diversidade.

4.3.1. Abordagens Clássicas

Dentre as principais abordagens clássicas para otimização multiobjetivo destacam-se [80] (Capítulo 3): *Weighted Sum*, *ϵ-Constraint*, *Weighted Metric*, Método de Benson e *Value Function*.

O método *Weighted Sum* é caracterizado pela utilização de um vetor de fatores de preferência para transformar as múltiplas funções-objetivo em uma única função $F(x) = \sum_{m=1}^{M} \omega_m \cdot f_m(x)$. Utiliza-se então algum algoritmo de otimização com objetivo único para encontrar uma das soluções que compõem o *Pareto-front*. Embora seja uma abordagem simples e que garante a descoberta de todas as soluções de um *Pareto-front* convexo, valores uniformemente distribuídos de ω_m podem não implicar em soluções uniformemente espaçadas no *Pareto-front* de problemas não-lineares. Adicionalmente, valores diferentes de ω_m podem conduzir à mesma solução e soluções de regiões convexas do espaço-objetivo não são encontradas. Em resumo, a capacidade de descoberta de boas soluções no *Pareto-front* convexo depende da habilidade do usuário na escolha dos valores de ω_m.

A impossibilidade de descoberta de soluções em regiões convexas do *Weighted Sum* motivou a criação do ϵ-*Constraint*. Nesta abordagem, apenas uma das funções-objetivo (μ) é mantida e as outras são parametrizadas de modo a modificar a região viável do problema. O problema passa a ser a minimização de $f_\mu(x)$, assumindo $f_m(x) \leq \epsilon_m;\ m = 1, 2, \ldots, M\ e\ m \neq \mu$ como restrições adicionais. Embora viabilize a descoberta de soluções em regiões convexas, o ϵ-*Constraint* também depende da escolha de bons valores para os parâmetros ϵ_m.

O método *Weighted Metric* baseia-se na minimização de uma única função-objetivo, desta vez formada pela ponderação de uma determinada métrica de distância, aplicada a cada função-objetivo original. Uma métrica frequentemente utilizada, neste caso, é a *norma*-L^p (norma de Lebesgue). A função a ser minimizada passa então a ser $l_p(x) = (\sum_{m=1}^{M} \omega_m \cdot |f_m(x) - z_m^*|^{\,1/p})$; onde z_m^* é a solução ideal[2] do problema. Diferentes valores de ω_m permitem a obtenção de diferentes soluções do *Pareto-front*, todas equidistantes (e isto depende do valor de p) da solução ideal z_m^*. Com $p = 1$, o algoritmo comporta-se de forma equivalente ao *Weighted Sum*. Com $p = 2$, a distância Euclidiana é utilizada. Valores maiores de p permitem a obtenção de soluções presentes em regiões convexas do espaço-objetivo. Este método requer a normalização dos objetivos e o cálculo da solução ideal z_m^*.

O Método de Benson utiliza uma abordagem semelhante ao *Weighted Metric*, exceto que uma solução de referência z^0 viável, porém não-Pareto-ótima, é utilizada no lugar da solução ideal z_m^*. Dessa forma, o problema é reduzido à maximização da função $F(x) = \sum_{m=1}^{M} \max(0, (z_m^0 - f_m(x)))$, com $f_m(x) \leq z_m^0$ como restrição adicional. As diferenças podem ser ponderadas, antes do somatório, para obtenção de diversas soluções do *Pareto-front*. O método pode ser utilizado para encontrar soluções em regiões convexas caso z^0 seja apropriadamente escolhido.

O método *Value Function* (também chamado *Utility Function*) consiste na maximização de uma função $\mathcal{U} : \mathbb{R}^M \rightarrow \mathbb{R}$, disponibilizada pelo usuário, que relaciona todos os M objetivos em um valor geral de utilidade. A função $\mathcal{U}$ deve ser válida em todo o espaço de busca e fortemente decrescente, ou seja, a preferência de uma solução deve aumentar caso um dos valores das funções-objetivo diminua, enquanto todos os outros são mantidos nos mesmos valores. Embora simples, o método depende de boas escolhas para a função $\mathcal{U}$, apresentando problemas quando os critérios acima não são satisfeitos.

4.3.2. Abordagens Evolucionárias

A maioria dos algoritmos para otimização com objetivo único utiliza um procedimento determinístico para descoberta da solução ótima. Tais algoritmos partem de uma solução viável aleatoriamente escolhida e, aplicando uma regra de transição previamente definida, seguem uma direção de busca, encontrada pelo algoritmo com base em alguma informação local ao ponto atual do espaço-objetivo/espaço de decisão. Uma solução melhor que a anterior é então encontrada e o processo se repete até a obtenção da solução (localmente) ótima. A maior parte destes algoritmos sofre das seguintes dificuldades:

- A convergência a uma solução globalmente ótima depende da solução aleatória inicialmente utilizada;
- A maioria dos algoritmos são susceptíveis a soluções localmente ótimas;
- O algoritmo pode ser eficiente somente em uma classe particular de problemas de otimização;
- O algoritmo pode não ser eficiente ao lidar com espaços discretos de busca; e

[2] Solução única, encontrada caso os objetivos não fossem conflitantes. Cada componente é a minimização individual de uma determinada função-objetivo.

- Os algoritmos não se beneficiam de execuções *multi-threaded* visto que múltiplas soluções são encontradas por execuções independentes.

As abordagens evolucionárias de otimização tentam remediar os problemas acima através do uso de uma população de soluções (em contraponto a uma única solução) a cada iteração do algoritmo. Dentre as abordagens evolucionárias mais utilizadas destacam-se: algoritmos genéticos, estratégias evolucionárias, programação evolucionária, programação genética e otimização multimodal de funções [80]. Estas abordagens podem ser utilizadas para otimização com objetivo único, mas são frequentemente adaptadas e adotadas como estratégia básica em algoritmos para otimização multiobjetivo. Dado o papel fundamental dos algoritmos genéticos na solução proposta neste trabalho, as demais abordagens não serão aqui apresentadas. O leitor interessado pode obter mais informações em [77] (Capítulo 4).

4.3.2.1. Algoritmos Genéticos

Ao longo dos últimos anos, os Algoritmos Genéticos (AGs) [112] têm sido largamente utilizados como ferramentas de busca e otimização em uma ampla faixa de domínios de aplicação. Os AGs utilizam mecanismos baseados nos princípios de genética natural e seleção natural para fazer com que uma população inicial de soluções evolua para uma geração seguinte que melhor atenda aos objetivos da otimização. A Figura 4.5 apresenta as atividades principais de um AG.

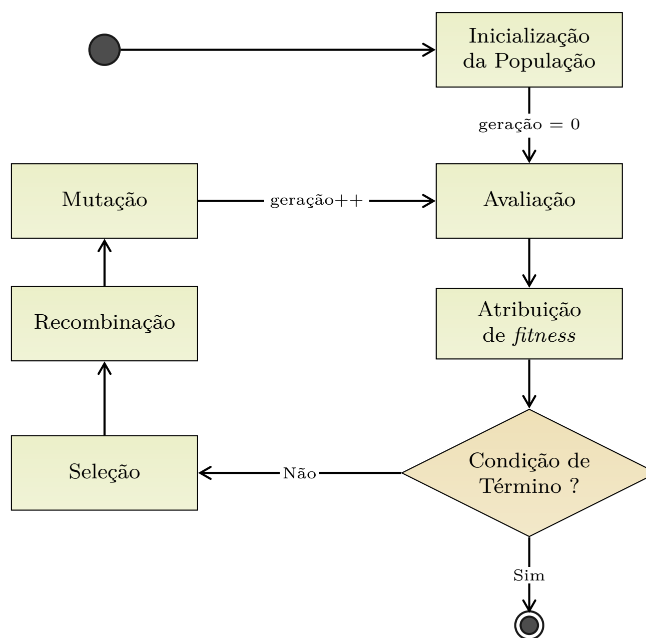


Figura 4.5.: Atividades gerais de um algoritmo genético (adaptada de [80]).

Conforme mencionado anteriormente, a meta de um bom algoritmo de otimização multiobjetivo é encontrar um *Pareto-front* com alta convergência e alta diversidade. Os AGs – bem como qualquer outra abordagem evolucionária – operam sempre com uma **população** de soluções a cada iteração. Esta população é geralmente inicializada de forma aleatória e passa por diversos ciclos de evolução, denominados **gerações**. Cada indivíduo da população é avaliado em relação às funções-objetivo em questão e um valor de **aptidão** (*fitness*) é a ele atribuído.

Cada iteração de evolução é geralmente composta pelas seguintes fases: **seleção** (*selection*), **recombinação** (*crossover*) e **mutação** (*mutation*). O objetivo principal da fase de seleção é promover a duplicação de indivíduos com boa aptidão e a eliminação daqueles com aptidão ruim, ao mesmo tempo em que mantém constante o tamanho da população.

Esta nova população é então submetida à fase de recombinação, onde pares de indivíduos combinam sua informação genética e geram um novo par de soluções, denominadas soluções *offspring*. De modo a preservar boas soluções identificadas na fase de seleção, nem todos os indivíduos são utilizados na fase de recombinação. Geralmente um valor de probabilidade de recombinação p_c é utilizado, indicando que $100p_c\%$ dos indivíduos selecionados serão recombinados e $100(1-p_c)\%$ deles serão diretamente copiados para a fase de mutação.

A fase de mutação tem como objetivo a manutenção da diversidade na população. Uma mutação consiste na alteração de uma determinada informação genética com base em uma probabilidade de mutação p_m. A seleção, recombinação e mutação podem ocorrer de diferentes formas, dependendo do tipo particular de **operador** adotado para cada uma dessas fases.

A forma com que um operador atua depende diretamente do modo de codificação das soluções. Um AG pode trabalhar com soluções codificadas em modo binário ou em modo real. Na codificação em modo binário, cada solução é representada pela concatenação de *strings* representando o valor em binário de cada variável de decisão x_i. Por exemplo, em um problema de otimização de volumes de latas de refrigerante usando a altura h e diâmetro d como variáveis de decisão, as possíveis soluções poderiam ser representadas como:

$$\underbrace{01000}_{d}\ \underbrace{01010}_{h}$$

A codificação acima permite que tanto a altura h e diâmetro d assumam valores inteiros entre 0 e 31. A codificação binária não está restrita à representação de variáveis de decisão inteiras, entretanto. Qualquer faixa de valores reais com diferentes graus de precisão pode representada sob a forma:

$$x_i = x_i^{min} + \frac{x_i^{max} - x_i^{min}}{2^{l_i} - 1} \cdot DV(s_i) \tag{4.1}$$

Na representação acima, s_i é o valor codificado pela *substring* correspondente à representação da variável de decisão x_i e l_i é o número de *bits* utilizado na *substring* s_i. x_i^{max} e x_i^{min} representam os limites superior e inferior, respectivamente, assumidos pela variável de decisão x_i. Esta faixa de valores é particionada em 2^{l_i} segmentos que representam a discretização da variável de decisão x_i. Uma maior precisão pode ser obtida simplesmente utilizando uma *string* com um tamanho l_i maior. A codificação binária é interessante por viabilizar uma representação cromossômica de soluções, onde operadores de seleção, recombinação e mutação atuam diretamente nos *bits* (genes) que representam um indivíduo da população.

AGs com codificação binária sofrem de uma série de problemas, entretanto. O primeiro deles é conhecido como *hamming clifs* (ou *hamming walls*) – situação onde a transição para uma solução vizinha requer a mudança de muitos *bits*, por exemplo, a transição de 01111 para 10000. Tal situação implica em uma probabilidade menor de transição entre tais soluções. Outro problema é a necessidade de uso de *strings* maiores quando deseja-se aumentar a precisão das soluções. Estudos [112] indicam que o uso de *strings* maiores requer também a utilização de populações de maior tamanho, aumentando consideravelmente o tempo de execução e consumo de memória do AG. Adicionalmente, os limites superior e inferior das variáveis de decisão devem ser conhecidos *a priori*, informação nem sempre disponível em todos os problemas de otimização.

Os AGs com codificação em modo real podem ser utilizados para mitigar tais problemas. Nestes casos, os operadores de seleção, recombinação e mutação são aplicados diretamente às representações reais das variáveis de decisão. Visto que os operadores de seleção dependem somente do valor de aptidão do indivíduo sendo analisado, qualquer operador de seleção para codificação binária pode ser utilizado em AGs com codificação real. Os operadores de recombinação e mutação para codificação binária, entretanto, atuam diretamente nos *bits* que representam

a solução e não podem ser diretamente utilizados em AGs com codificação real. Dessa forma, novos operadores de recombinação e mutação são propostos, com o objetivo de derivar um novo par de soluções *offspring* a partir de um par de vetores de variáveis reais e realizar perturbações significativas em um vetor de variáveis reais, respectivamente. São apresentados a seguir os principais operadores de seleção, recombinação e mutação utilizados em AGs com codificação binária e real.

Operadores de seleção. Os operadores de seleção, de um modo geral, realizam as seguintes atividades: *i)* identificação de boas soluções na população (geralmente acima da média); *ii)* geração de múltiplas cópias destas boas soluções; e *iii)* eliminação de soluções ruins de modo a manter o tamanho da população constante. Tais atividades podem ser realizadas de formas diferentes, dando origem aos diversos operadores de seleção utilizados em AGs. Dentre os operadores de seleção mais utilizados destacam-se: seleção por torneio (*tournament selection*), seleção proporcionada (*proportionate selection*) e seleção por *ranking* (*ranking selection*).

Na seleção por torneio, cada solução presente na população participa de dois torneios com duas outras soluções. Para cada torneio, a solução com melhor aptidão é selecionada para a próxima fase (recombinação). Visto que cada solução participa de dois torneios, a melhor solução da população irá vencer os dois torneios e duas cópias dela serão selecionadas para a próxima fase. De forma semelhante, a pior solução da população perderá ambos os torneios e não estará presente na fase de recombinação. Soluções medianas estarão presentes somente uma vez na fase de recombinação.

A seleção proporcionada, por sua vez, é caracterizada pela criação de cópias das soluções, de forma proporcional ao valor de aptidão de cada uma. Supondo que a aptidão média de todos os N indivíduos da população é f_{avg}, o número de exemplares de uma solução i com aptidão f_i será f_i/f_{avg}. A partir daí, escolhe-se aleatoriamente N indivíduos da população formada por todas as cópias criadas. A seleção proporcionada pode ser compreendida como a realização de N execuções de uma roleta-russa formada por N partições, onde o tamanho de cada partição é proporcional ao valor de aptidão de cada indivíduo. Visto que o cálculo da aptidão média requer a avaliação de aptidão de cada indivíduo da população, o operador de seleção proporcionada possui desempenho pior que a seleção por torneio. Outro problema da seleção proporcionada é que indivíduos com aptidão muito acima da média podem dominar a fase de seleção, diminuindo a diversidade durante a recombinação.

O operador de seleção por *ranking* pode ser utilizado para minimizar o problema de dominação por soluções com aptidão acima da média. O operador consiste na ordenação das soluções de acordo com suas aptidões, da pior (*rank* 1) para a melhor (*rank* N). O valor de aptidão de cada solução é então substituído pelo seu valor de *rank* e a seleção proporcionada é aplicada.

Operadores de recombinação. Um operador de recombinação é aplicado a um par de soluções identificadas na fase de seleção, com o objetivo de gerar dois novos indivíduos na população. Os operadores de recombinação para codificação binária selecionam aleatoriamente duas soluções (soluções pais) da população resultante da fase de seleção e realizam a troca de uma ou mais partes das *strings* que representam estas soluções, gerando dois novos indivíduos (soluções *offspring*). Os operadores de recombinação para codificação binária mais utilizados são: recombinação com ponto único (*single-point crossover*), recombinação com n pontos (*n-point crossover*) e recombinação uniforme (*uniform crossover*).

Na recombinação com ponto único, um local de recombinação (*crossing site*) é aleatoriamente escolhido ao longo das *strings* que representam as soluções sendo recombinadas. A partir daí, os conteúdos do lado direito do local de recombinação são trocados, gerando dois novos indivíduos.

Por exemplo, considerando o local de recombinação entre o terceiro e quarto *bit* a recombinação com ponto único é:

$$\begin{array}{ccc} \text{soluções pais} & & \text{soluções } \textit{offspring} \\ \overbrace{010|0001010} & \rightarrow & \overbrace{010|1000110} \\ 011|1000110 & & 011|0001010 \end{array}$$

Embora não exista a garantia de que as soluções *offspring* criadas tenham aptidão melhor que a de seus pais, é esperado que isto aconteça com uma frequência maior que o simples acaso visto que os pais sendo recombinados representam soluções com alta aptidão, resultantes do processo de seleção anteriormente realizado. Caso soluções com baixa aptidão sejam geradas, elas serão automaticamente descartadas na fase de seleção da próxima geração.

A ideia de recombinação com ponto único pode ser estendida para uso de n locais de recombinação. Por exemplo, na recombinação com dois pontos, escolhe-se dois locais de recombinação aleatoriamente e realiza-se a troca da *substring* central de cada solução pai. Por exemplo, considerando como locais de recombinação as posições entre o segundo e terceiro *bit* e entre o sexto e sétimo *bit*, a recombinação é:

$$\begin{array}{ccc} \text{soluções pais} & & \text{soluções } \textit{offspring} \\ \overbrace{01|0000|1010} & \rightarrow & \overbrace{01|1100|1010} \\ 01|1100|0110 & & 01|0000|0110 \end{array}$$

A ideia pode ser estendida para a execução de recombinação com n pontos, realizando a troca de informação a cada *substring* em posição par. O extremo desse conceito é a recombinação uniforme onde, para cada par de *bits* compondo as *strings* das soluções pais, realiza-se a troca com alguma probabilidade p (geralmente 0.5). Embora o uso de mais locais de recombinação ajude a aumentar o número de soluções *offspring* diferentes possíveis, tem-se como desvantagem a redução da preservação da estrutura das soluções pais, o que pode aumentar a probabilidade de geração de soluções com aptidão menor.

Operações de recombinação para AGs com codificação real são mais difíceis de serem realizadas pois são aplicadas diretamente em representações reais das variáveis de decisão, em vez de *strings* com codificação binária de tais valores. Dentre os principais operadores de recombinação para codificação real, destacam-se: recombinação linear (*linear crossover*), *blend crossover*, recombinação binária simulada (*simulated binary crossover*), recombinação *fuzzy* e recombinação simplex. Visto que AGs com codificação real não são utilizados neste trabalho, maiores detalhes sobre o funcionamento destes operadores não serão aqui apresentados. Mais informações podem ser encontradas em [80] (Capítulo 4) e uma discussão mais aprofundada sobre as implicações e propriedades dos operadores de recombinação pode ser encontrada em [286].

Operadores de mutação. Os operadores de mutação são utilizados para manter a diversidade na população, após a fase de recombinação. O operador de mutação para codificação binária mais utilizado é a mutação *bit-a-bit*, onde cada *bit* é modificado de 1 para 0 (ou vice-versa) com uma probabilidade p_m. Novamente, não há como garantir que a solução após a mutação será melhor que a anterior, mas espera-se que eventuais soluções ruins sejam descartadas na fase de seleção da próxima geração. A mutação *bit-a-bit* requer a geração de um número randômico para cada *bit* da *string*. O operador relógio de mutação (*mutation clock*) reduz esta complexidade através do uso de uma função exponencial com média $\mu = 1/p_m$ para indicar a posição do próximo *bit* a ser modificado. O próximo *bit* a ser modificado é aquele que está $\eta = -p_m \cdot \ln(1-r)$ posições à frente do *bit* atual, onde r é um número randômico entre 0 e 1.

Dentre os principais operadores de mutação para codificação real, destacam-se: mutação randô-

mica (*random mutation*), mutação não-uniforme (*non-uniform mutation*), mutação normalmente distribuída (*normally distributed mutation*) e mutação polinomial (*polinomial mutation*). Novamente, maiores detalhes sobre estes operadores não serão aqui apresentados. O leitor interessado pode encontrar tais informações em [80].

4.3.2.2. O NSGA-II

O NSGA-II (*Non-dominated Sorting Genetic Algorithm II*) [78] é uma abordagem evolucionária para otimização multiobjetivo baseada na utilização de operadores de **preservação de elite** e de **preservação de diversidade**. Em abordagens sem preservação de elite, a próxima geração é formada somente pelas soluções *offspring* Q_t. Nenhuma das soluções pais P_t que originaram as soluções *offspring* estará presente nesta próxima geração. Consequentemente, caso nenhuma solução *offspring* tenha aptidão igual ou melhor que a melhor dentre as soluções pais, a geração seguinte será caracterizada por indivíduos com pior aptidão que seus pais.

Abordagens para otimização multiobjetivo com preservação de elite garantem que a próxima geração não será pior que a atual, ao permitir que as soluções pais também participem da próxima geração. Para isso, as populações Q_t e P_t são combinadas de modo a formar uma população R_t de tamanho $2N$. Algum critério é então aplicado de modo a selecionar os N melhores indivíduos para a próxima geração (operação de **redução**).

O NSGA-II utiliza a **ordenação por não-dominância** (*non-dominated sorting*) e **ordenação por aglomeração** (*crowding sorting*) para realizar a redução da população R_t (de tamanho $2N$) para a população P_{t+1} (de tamanho N). A Seção 4.2.1 apresentou três algoritmos diferentes para obtenção do subconjunto P' das soluções não-dominadas de um conjunto P de soluções. O NSGA-II requer, entretanto, a obtenção de diversos conjuntos de soluções não-dominadas, representando diferentes **níveis (ou ranks) de não-dominância (NND)** que englobam todas as soluções presentes em P.

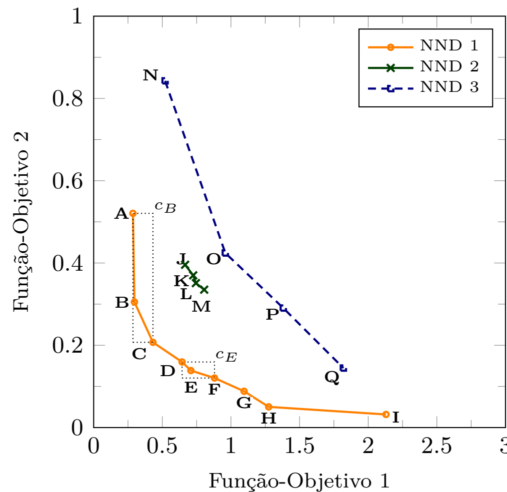


Figura 4.6.: Exemplos dos níveis de não-dominância (NND) e cuboides da distância de aglomeração (DA) utilizados no NSGA-II (adaptada de [80]).

Esta operação é denominada ordenação por não-dominância e é ilustrada na Figura 4.6. As soluções de A a I representam o subconjunto P' das soluções não-dominadas do conjunto P de todas as possíveis soluções (NND 1). Pode-se verificar, entretanto, que as soluções de J a M formam um segundo NND, obtido após a remoção, de P, dos elementos do NND 1. O processo se repete, até que cada solução de P esteja alocada a algum NND.

Uma implementação simples da ordenação por não-dominância pode ser obtida através da aplicação de um dos algoritmos para obtenção de soluções não-dominadas apresentados na Seção

4.2.1, seguida da exclusão destas soluções do conjunto P original e da re-execução do algoritmo, de modo a obter-se o segundo NND. O processo se repete com a obtenção dos demais NNDs até que o conjunto P esteja vazio. Esta implementação requer $O(MN^2)$ computações por NND, no pior caso.

Algoritmo 4.4: Algoritmo rápido para ordenação por não-dominância.

1: **procedure** FASTNDSORT(P) ▷ P = conjunto original de índices de soluções
2: **for all** $p \in P$ **do**
3: $S_p \leftarrow \emptyset$ ▷ Conjunto de soluções dominadas por $P^{(p)}$
4: $n_p \leftarrow 0$ ▷ Número de soluções que dominam $P^{(p)}$
5: **for all** $q \in P \mid q \neq p$ **do**
6: **if** $P^{(p)} \preceq P^{(q)}$ **then** ▷ Se $P^{(p)}$ domina $P^{(q)}$
7: $S_p \leftarrow S_p \cup \{q\}$
8: **if** $P^{(q)} \preceq P^{(p)}$ **then** ▷ Se $P^{(q)}$ domina $P^{(p)}$
9: $n_p \leftarrow n_p + 1$
10: **if** $n_p = 0$ **then** ▷ $P^{(p)}$ pertence ao primeiro NND
11: $p_{rank} \leftarrow 1$
12: $\mathcal{F}_1 \leftarrow \mathcal{F}_1 \cup \{p\}$
13: $i \leftarrow 1$
14: **while** $\mathcal{F}_i \neq \emptyset$ **do**
15: $Q \leftarrow \emptyset$ ▷ Membros do próximo NND
16: **for all** $p \in \mathcal{F}_i$ **do**
17: **for all** $q \in S_p$ **do**
18: $n_q \leftarrow n_q - 1$
19: **if** $n_q = 0$ **then** ▷ $P^{(q)}$ pertence ao próximo NND
20: $q_{rank} \leftarrow i + 1$
21: $Q \leftarrow Q \cup \{q\}$
22: $i \leftarrow i + 1$
23: $\mathcal{F}_i \leftarrow Q$
24: **return** $\mathcal{F}$ ▷ Retorna os *fronts* ordenados

Uma implementação mais eficiente da ordenação por não-dominância é apresentada no Algoritmo 4.4, proposto por Deb em [80]. Esta implementação requer $O(MN^2)$ computações para encontrar todos os NNDs. O algoritmo calcula, para cada solução i em P, duas variáveis: o contador de dominância n_i (número de soluções que dominam a solução i) e o conjunto S_i das soluções dominadas por i.

Ao final do cálculo destas variáveis, as soluções do NND 1 são aquelas com n_i igual a 0. Em seguida, para cada uma das soluções f no NND 1, visita-se cada membro j presente no conjunto S_f, reduzindo o contador n_j de uma unidade. Caso este contador passe a ser 0, coloca-se o membro j em um conjunto temporário Q, que representa as soluções que constituem o próximo NND. Após a realização deste procedimento para cada solução f do NND 1, as soluções presentes em Q serão aquelas que compõem o NND 2. O processo se repete até que todas as soluções sejam alocadas a algum NND.

A ordenação por não-dominância define uma ordem parcial que classifica as soluções em relação ao grau de convergência para um *Pareto-front* ótimo. Entretanto, dadas duas soluções pertencentes ao mesmo NND, não é possível classificar uma como melhor que a outra. De modo a obter-se um alto grau de diversidade no *Pareto-front* encontrado, o NSGA-II utiliza a ordenação por aglomeração para classificar soluções de um mesmo NND. Esta operação é realizada utilizando a **distância de aglomeração (DA)** (*crowding distance*) como operador de ordenação. O Algoritmo 4.5 descreve como a DA é atribuída a cada solução de um NND.

Algoritmo 4.5: Algoritmo para cálculo da distância de aglomeração (DA).

1: **procedure** CROWDINGDISTANCE($\mathcal{F}, f$)
2: ▷ $\mathcal{F}$ = NND a ser ordenado
3: ▷ f = conjunto de funções-objetivo
4: $l \leftarrow |\mathcal{F}|$
5: **for all** $i \in \mathcal{F}$ **do**
6: $d_i \leftarrow 0$
7: **for all** $m \in f$ **do**
8: $I^m \leftarrow sort(\mathcal{F}, <_{f_m})$ ▷ Índices ordenados pela m-ésima função-objetivo
9: $d_{I_1^m} \leftarrow \infty$ ▷ Atribui-se DA infinita à primeira solução de I^m
10: $d_{I_l^m} \leftarrow \infty$ ▷ Atribui-se DA infinita à última solução de I^m
11: **for** $i \leftarrow 2, \ldots, l-1$ **do**
12: $d_{I_i^m} \leftarrow d_{I_i^m} + (f_m^{(I_{i+1}^m)} - f_m^{(I_{i-1}^m)})/(f_m^{max} - f_m^{min})$
13: **return** d ▷ Retorna a distância de aglomeração das soluções em $\mathcal{F}$

Inicialmente, a DA de todas as soluções do NND sendo analisado é inicializada com 0. Em seguida, para cada função-objetivo m, ordena-se o conjunto de soluções em relação ao valor apresentado por cada solução para a função-objetivo m e realiza-se o cálculo da contribuição parcial desta função-objetivo à DA. A DA tem como meta atribuir valores maiores àquelas soluções presentes em uma região menos populosa do NND sendo analisado. Para isso, a distância calcula um valor estimado do perímetro do cuboide formado pela solução sendo analisada e seus dois vizinhos em relação à função-objetivo em questão.

A Figura 4.6, apresentada anteriormente, ilustra esta operação. Note como o cuboide formado pela solução B e seus dois vizinhos (A e C) possui perímetro maior que o cuboide referente à solução E e seus vizinhos. Dessa forma, o valor da DA para B será maior que o de E, indicando que, dentre estas duas soluções, B é uma solução que melhor preserva a diversidade do *Pareto-front* que E. O cálculo da contribuição de cada função-objetivo à DA é apresentado na linha 12 do Algoritmo 4.5. A ordenação por aglomeração classifica as soluções de um NND em ordem decrescente de DA. Vale ressaltar que as duas soluções extremas serão sempre as com maior DA (infinito), de modo a preservar todo o espectro de soluções encontradas.

Algoritmo 4.6: Algoritmo NSGA-II para otimização multiobjetivo.

1: **procedure** NSGAII($P_0, maxGen, f$)
2: ▷ P_0 = população inicial de soluções
3: ▷ $maxGen$ = número máximo de gerações
4: ▷ f = conjunto de funções-objetivo
5: $t \leftarrow 1$
6: **while** $t <= maxGen$ **do**
7: $Q_t \leftarrow createOffspring(P_{t-1}, \textbf{CTSOper}, crossoverOper, mutationOper)$
8: $R_t \leftarrow P_{t-1} \cup Q_t$
9: $\mathcal{F} \leftarrow$ FASTNDSORT(R_t) ▷ $\mathcal{F}$ = conjunto ordenado de NND
10: $P_t \leftarrow \emptyset$
11: $i \leftarrow 0$
12: **while** $|P_t| + |\mathcal{F}_i| < N$ **do** ▷ Enquanto puder adicionar um NND inteiro
13: $P_t \leftarrow P_t \cup \mathcal{F}_i$
14: $i \leftarrow i + 1$
15: $\mathcal{F}_i \leftarrow sort(\mathcal{F}_i, <_{\text{CROWDINGDISTANCE}(\mathcal{F}_i, f)})$ ▷ Ordene $\mathcal{F}_i$ em relação à DA
16: $P_t \leftarrow P_t \cup \mathcal{F}_i[1 : (N - |P_t|)]$ ▷ Adicione os melhores elementos de $\mathcal{F}_i$
17: $t \leftarrow t + 1$

Dadas as definições de ordenação por não-dominância e ordenação por aglomeração, a estrutura principal do NSGA-II pode então ser apresentada (Algoritmo 4.6). O processo começa com a criação da primeira população de soluções *offspring* Q_1 (linha 7). Para isso, são utilizados dois operadores quaisquer de recombinação e mutação, associados a um operador de seleção particularmente adotado pelo NSGA-II: o **operador de seleção por torneio aglomerado** (*crowded tournament selection operator*), indicado como **CTSOper** na linha 7 do Algoritmo 4.6. Este operador define que uma solução i vence um torneio com outra solução j quando:

1. A solução i está em um NND com índice menor que o NND da solução j: $i_{rank} < j_{rank}$; ou
2. Caso i e j façam parte do mesmo NND, a solução i possui DA maior que o da solução j: $i_{rank} = j_{rank}$ e $d_i > d_j$.

Dessa forma, o processo de seleção favorece a criação de cópias de soluções presentes em NNDs com menor índice e, dentre estas, daquelas que residem em regiões menos aglomeradas (soluções com maior DA) do NND. A partir daí, as populações de soluções pais (P_0) e *offspring* (Q_1) são combinadas ($R_1 \leftarrow P_0 \cup Q_1$) e a nova população R_1 é passada para o processo de ordenação por não-dominância descrito no Algoritmo 4.4 (linhas 8 e 9). A saída $\mathcal{F}$ é o conjunto de NNDs encontrados na população R_1, ordenados do menor para o maior nível. Inicia-se então a fase de redução – formação da população das soluções pais da próxima geração (P_1). Para isso, adiciona-se a P_1 – em ordem crescente de nível de não-dominância i – todos os NNDs $\mathcal{F}_i$ que caibam por completo (sem exceder o tamanho N da população) em P_1 (linhas 12 a 14). As soluções restantes que irão completar a população P_1 são aquelas com maiores valores de DA (presentes em regiões menos aglomeradas do NND) no próximo NND $\mathcal{F}_i$ (linhas 15 e 16). Todo o processo se repete até que o número máximo de gerações *maxGen* seja alcançado.

Os mecanismos de preservação de elite e preservação de diversidade introduzidos pelo NSGA-II fazem dele um dos algoritmos com maior velocidade de convergência disponíveis na literatura. Por outro lado, o NSGA-II apresenta desempenho aceitável somente em problemas de otimização com no máximo quatro funções-objetivo [79].

O uso de um número maior de funções-objetivo traz uma série de problemas para os algoritmos de otimização multiobjetivo. Primeiro, a proporção de soluções não-dominadas em uma população aumenta consideravelmente, fazendo com que a preservação de elite tenha mais dificuldade em acomodar um número adequado de novas soluções e, consequentemente, torne o processo de convergência mais lento. Segundo, o cálculo do operador de preservação de diversidade (distância de aglomeração) se torna muito mais custoso computacionalmente. Terceiro, a avaliação de métricas de desempenho (apresentadas na Seção 4.4) também passa a demandar maior poder computacional. Embora uma versão melhorada do NSGA-II – atacando os problemas acima levantados – já esteja disponível (NSGA-III [79, 153]), tal mecanismo não foi utilizado neste trabalho.

4.4. Métricas de Desempenho

À medida em que novos algoritmos de otimização multiobjetivo eram propostos, fez-se necessário comparar os seus desempenhos com base em um conjunto padronizado de problemas-teste (*benchmarks*) e métricas de desempenho. Uma ampla coleção de problemas-teste está atualmente disponível na literatura [81], variando em relação à natureza e extensão das dificuldades encontradas, bem como em relação à localização e formato do *Pareto-front* a ser identificado.

O desempenho de um algoritmo de otimização multiobjetivo é medido por dois fatores distintos: a qualidade do *Pareto-front* encontrado (convergência e diversidade) e o número de gerações

necessárias para obter este *Pareto-front* [328]. A qualidade do *Pareto-front* também é uma medida indireta da qualidade da modelagem realizada para o problema de otimização em questão, visto que *Pareto-fronts* com maior diversidade são um indício do caráter conflitante das funções-objetivo utilizadas. Vale ressaltar que os algoritmos de otimização evolucionária são processos estocásticos – cada execução pode gerar um *Pareto-front* diferente. A saída de tais algoritmos é uma variável aleatória e, portanto, múltiplas execuções são necessárias para se afirmar, com significância estatística, a superioridade de uma determinada abordagem em relação a outra.

Visto que esta tese não propõe novos algoritmos para otimização multiobjetivo, serão aqui apresentadas somente as métricas para avaliação da qualidade do *Pareto-front* obtido. Tais métricas são classificadas em: métricas para avaliação de convergência, métricas para avaliação de diversidade e métricas para avaliação conjunta de convergência e diversidade [80].

4.4.1. Métricas para Avaliação de Convergência

Métricas deste grupo calculam a proximidade de um conjunto B em relação a outro conjunto A conhecido. Dentre as métricas mais utilizadas para este fim destacam-se: *error ratio* (ER), *set coverage* ($\mathcal{C}$), *generational distance* (GD) e *maximum pareto-optimal front error* (MFE) [80].

A métrica *error ratio* (ER) conta o número de soluções i de B que não são membros de A:

$$ER(B, A) = \frac{\sum_{i=1}^{|B|} e_i}{|B|} \tag{4.2}$$

Na equação acima, $e_i = 1$ caso $i \notin A$; e $e_i = 0$ caso contrário. A métrica ER assume um valor entre 0 (todas as soluções de B estão em A) e 1 (nenhuma solução de B está em A), sendo que valores menores indicam uma melhor convergência para A. Esta métrica, entretanto, apresenta duas desvantagens. Primeiro, mesmo que uma solução de B seja Pareto-ótima, se esta solução não existir em A haverá uma penalização no valor de ER. Para minimizar este problema, é importante que A seja um *Pareto-front* de alta cardinalidade. Segundo, caso nenhuma solução de B esteja em A, a métrica ER não informa quão próximo B está de A. Tais limitações fazem com que esta métrica seja pouco utilizada na prática.

A métrica *set coverage* ($\mathcal{C}$) calcula a proporção de soluções em B que são dominadas por A:

$$\mathcal{C}(B, A) = \frac{|\{b \in B \mid \exists a \in A \colon a \preceq b\}|}{|B|} \tag{4.3}$$

Um valor de $\mathcal{C}(B, A) = 1$ indica que todos os membros de B são dominados por A, enquanto $\mathcal{C}(B, A) = 0$ indica que nenhum membro de B é dominado por A. Dessa forma, valores menores de $\mathcal{C}(B, A)$ indicam uma melhor convergência. Visto que a operação de dominância não é simétrica, tem-se que $\mathcal{C}(B, A)$ não necessariamente é igual a $1 - \mathcal{C}(A, B)$. Note também que a cardinalidade dos conjuntos A e B não precisam ser iguais.

A métrica *generational distance* (GD) encontra a distância média das soluções de B em relação a A:

$$GD(B, A) = \frac{(\sum_{i=1}^{|B|} d_i^p)^{1/p}}{|B|} \tag{4.4}$$

Qualquer *norma-*L^p pode ser utilizada na métrica *generational distance*. Para $p = 2$, o parâmetro d_i passa a ser a distância Euclidiana entre a solução $i \in B$ e o membro de A mais próximo:

$$d_i = \min_{k=1}^{|A|} \sqrt{\sum_{m=1}^{\mathcal{M}} (f_m^{(i)} - f{*}_m^{(k)})^2} \tag{4.5}$$

Na equação acima, $f*_m^{(k)}$ é o valor da m-ésima função-objetivo do k-ésimo membro de A e $f_m^{(i)}$ é o valor da m-ésima função-objetivo da solução de B sendo avaliada. Valores menores de GD indicam uma maior proximidade do *Pareto-front* B em relação a A. A *generational distance* requer um conjunto A de alta cardinalidade e a normalização dos valores das funções-objetivo. Adicionalmente, o cálculo da variância de GD pode ser necessário caso haja muita flutuação entre os valores de distância.

A métrica *maximum Pareto-optimal front error* (MFE) calcula a pior distância d_i entre todos os membros de B. MFE é uma medida conservadora de convergência e pode disponibilizar informações incorretas sobre a convergência da solução B.

4.4.2. Métricas para Avaliação de Diversidade

Tais métricas calculam a diversidade das soluções pertencentes ao *Pareto-front* encontrado. A medida de diversidade é influenciada por dois fatores distintos: extensão (distância entre as duas soluções mais extremas do *Pareto-front*) e distribuição (variação das distâncias relativas entre as soluções do *Pareto-front*). As métricas mais utilizadas neste grupo são: *spacing* (S), *spread* (Δ) e *maximum spread* ($\overline{\mathcal{D}}$).

A métrica *spacing* (S) calcula a distância relativa entre soluções consecutivas do *Pareto-front* B sendo investigado:

$$S(B) = \sqrt{\frac{1}{|B|} \cdot \sum_{i=1}^{|B|} (d_i - \overline{d})^2}\,; \qquad \text{onde} \tag{4.6}$$

$$d_i = \min_{k \in B | k \neq i} \sum_{m=1}^{\mathcal{M}} \left| f_m^{(i)} - f_m^{(k)} \right|\,; \qquad \text{e} \qquad \overline{d} = \sum_{i=1}^{|B|} d_i / \left| B \right| \tag{4.7}$$

Na equação acima, d_i é o valor mínimo da soma das diferenças absolutas dos valores de funções-objetivo entre a i-ésima solução de B e qualquer outra solução também presente neste conjunto. A métrica *spacing* mede o desvio padrão dos valores de d_i e apresenta valores menores quando as soluções de B estão uniformemente distribuídas. Esta métrica também requer a normalização dos valores das funções-objetivo e não disponibiliza nenhuma informação sobre a extensão do *Pareto-front*.

A métrica *spread* (Δ) avalia a diversidade do *Pareto-front* B em função tanto da extensão quanto da distribuição das soluções:

$$\triangle(B, A) = \frac{\sum_{m=1}^{\mathcal{M}} d_m^e + \sum_{i=1}^{|B|} \left| d_i - \overline{d} \right|}{\sum_{m=1}^{\mathcal{M}} d_m^e + |Q| \cdot \overline{d}} \tag{4.8}$$

Na equação acima, d_m^e é a distância entre as soluções extremas de B e as soluções extremas de um *Pareto-front* de referência A, em relação à m-ésima função-objetivo. O conjunto A é utilizado para avaliar o quanto a extensão das soluções em B se aproxima da extensão apresentada no *Pareto-front* de referência A. O valor d_i pode ser qualquer medida de distância entre soluções vizinhas em B e $\overline{d}$ é a média destas distâncias. A métrica assume valor 0 na situação ideal onde os extremos de B são iguais aos de A (alta extensão) e todos os valores de d_i são iguais à sua média (distribuição uniforme). Valores maiores são obtidos nesta métrica quando os extremos de B se afastam dos de A ou (não-exclusivo) quando as soluções deixam de estar uniformemente distribuídas.

A métrica *maximum spread* ($\overline{\mathcal{D}}$) mede o comprimento da diagonal do hipercubo formado pelos valores extremos de funções-objetivo das soluções de B:

$$\overline{\mathcal{D}}(B,A) = \sqrt{\frac{1}{\mathcal{M}} \cdot \sum_{m=1}^{\mathcal{M}} \left(\frac{\max_{i=0}^{|B|} f_m^{(i)} - \min_{i=0}^{|B|} f_m^{(i)}}{f_m^{max} - f_m^{min}}\right)^2} \quad (4.9)$$

Na equação acima, f_m^{max} e f_m^{min} são os valores máximo e mínimo dos valores da m-ésima função-objetivo, dentre todas as soluções de um *Pareto-front* de referência A. Dessa forma, um valor de $\overline{\mathcal{D}} = 1$ indica que uma solução com alta extensão foi encontrada. A métrica não avalia, entretanto, a distribuição das soluções de B.

4.4.3. Métricas para Avaliação Conjunta de Convergência e Diversidade

Métricas deste grupo disponibilizam uma medida tanto da convergência quanto da diversidade do *Pareto-front* encontrado. Dentre as métricas deste grupo destacam-se: *hypervolume* (HV), *weighted metric* ($\mathcal{W}$) e *non-dominated evaluation metric*.

A métrica *hypervolume* (HV) calcula o volume (no espaço-objetivo) coberto por todos os membros de B para problemas de minimização de todas as funções-objetivo. Para cada solução $i \in B$, constrói-se um hipercubo v_i com diagonal formada pela solução i e um ponto de referência fixo $\mathcal{W}$ – geralmente o vetor formado pelos piores valores de funções-objetivo. A métrica *hypervolume* calcula o hipervolume da união de todos os hipercubos v_i construídos:

$$HV(B) = \text{volume}(\cup_{i=1}^{|B|} v_i) \quad (4.10)$$

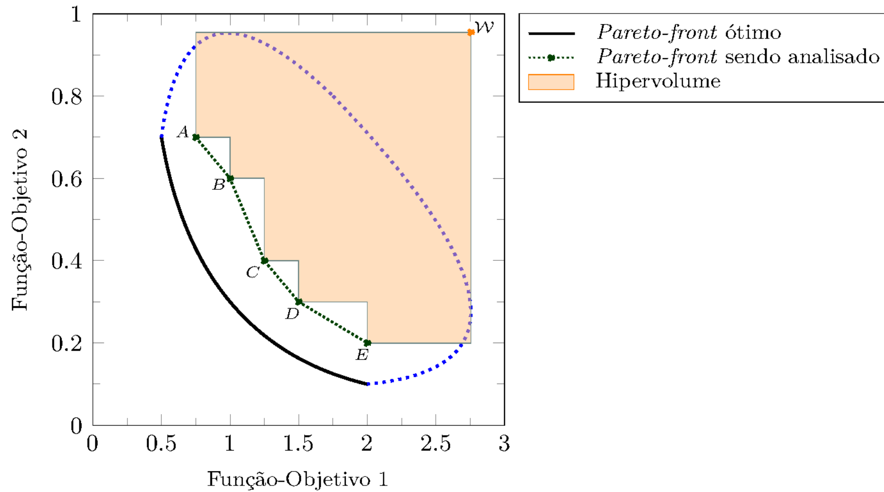


Figura 4.7.: Hipervolume gerado pelas soluções do *Pareto-front* e o ponto de referência $\mathcal{W}$ (adaptada de [80]).

A Figura 4.7 apresenta um exemplo de hipervolume para as soluções de um *Pareto-front*. Soluções que apresentam maiores valores para HV caracterizam *Pareto-fronts* com maior convergência e maior diversidade. A métrica HV, entretanto, é fortemente influenciada pelas diferenças de escala entre as funções-objetivo. Para minimizar este problema, pode-se realizar uma normalização prévia dos valores das funções-objetivo ou calcular o hipervolume relativo (HVR) da solução B em relação a alguma solução de referência A:

$$HVR(B,A) = \frac{HV(B)}{HV(A)} \quad (4.11)$$

Para um problema de minimização de todas as funções-objetivo, o melhor valor de HVR é 1 ($B = A$).

A *weighted metric* ($\mathcal{W}$) é um procedimento simples para avaliação conjunta de convergência e diversidade, utilizando a combinação ponderada de uma métrica de convergência e uma métrica de diversidade. Por exemplo, pode-se utilizar a *generational distance* (GD) em conjunto com a *spread* (Δ) da seguinte forma:

$$\mathcal{W}(B, A) = w_1 \cdot GD(B, A) + w_2 \cdot \Delta(B, A) \tag{4.12}$$

Onde, $w_1 + w_2 = 1$ na equação acima. Visto que ambas as métricas GD e Δ atribuem valores menores a soluções melhores, soluções com menor valor de $\mathcal{W}$ também apresentam boa convergência e diversidade. Aconselha-se, entretanto, que um par normalizado de métricas seja utilizado.

Por fim, a *non-dominated evaluation metric* analisa duas soluções A e B através da formulação da comparação de soluções como um problema de otimização com dois objetivos (convergência e diversidade). A relação de dominância é utilizada para verificar se: *i)* A é melhor que (domina) B; *ii)* B é melhor que (domina) A; ou *iii)* A e B são soluções não-dominadas e nada pode-se afirmar sobre a diferença de qualidade das soluções.

Parte II

Projeto Arquitetural Automatizado de Sistemas Self-Adaptive

Capítulo 5

Projeto Arquitetural Automatizado de Sistemas Self-Adaptive

É nas quedas que o rio cria energia.
Hermógenes

O Capítulo 3 apresentou as motivações, fundamentos e principais mecanismos utilizados no projeto e implementação de capacidades de autogerenciamento em sistemas computacionais. Os requisitos amplos e complexos que caracterizam o espaço de problema, a amplitude do espaço de solução e a presença de *trade-offs* frequentemente não conhecidos ou difíceis de avaliar fazem com que o projeto de arquiteturas efetivas para sistemas *self-adaptive* seja uma tarefa desafiadora. O sucesso desta atividade depende fortemente da aplicação criteriosa de técnicas gerais de projeto de arquiteturas de *software*, mas principalmente do uso oportuno de conhecimento de domínio específico.

Conforme apresentado anteriormente, um sistema *self-adaptive* com adaptação externa é constituído por dois módulos principais: o sistema gerenciado e o sistema gerenciador. O sistema gerenciado implementa as regras de negócio e disponibiliza interfaces para monitoramento e realização de adaptações. O sistema gerenciador, por sua vez, é responsável pela implementação dos *feedback control loops* que mantêm os níveis de qualidade do serviço oferecido pelo sistema gerenciado – mesmo na presença de perturbações (variações nas demandas por serviço, falhas, etc).

A escolha da arquitetura mais efetiva para o sistema gerenciador envolve a consideração de aspectos tais como: número de saídas medidas (*measured outputs*) e entradas de controle (*control inputs*), necessidade de uso de transdutores e filtros, abordagem para autogerenciamento adotada, estratégia de controle (lei de atuação) a ser aplicada, propriedades de controle desejadas, necessidade de uso de múltiplos *loops* e análise de eventuais interações, grau de robustez e adaptação de controle necessários, análise dos *trade-offs* envolvidos, dentre outros. Realizar boas decisões em relação aos aspectos acima citados requer experiência e formação específica no domínio.

Este capítulo apresenta o processo automatizado de projeto arquitetural para sistemas *self-adaptive* proposto nesta tese. Serão apresentados as motivações e os objetivos da abordagem, bem como a visão geral do processo e as premissas e limitações envolvidas. A Figura 5.1 apresenta o roteiro deste capítulo.

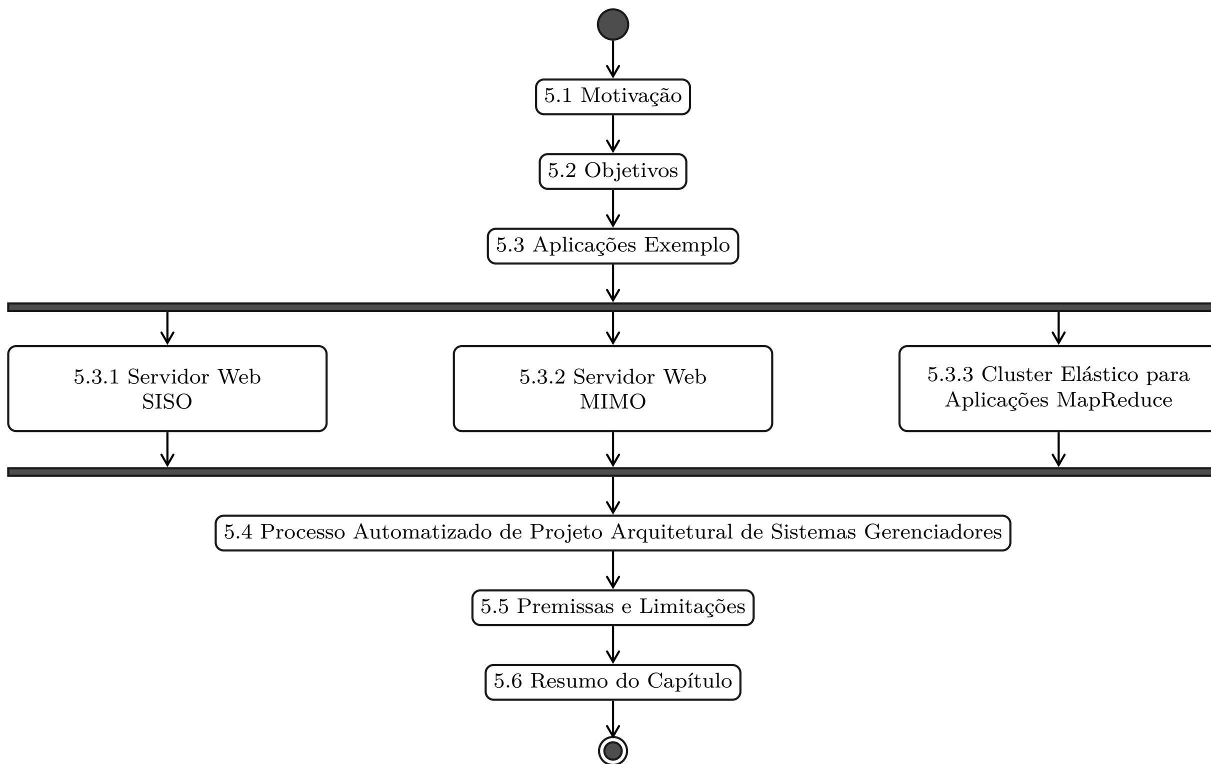


Figura 5.1.: Roteiro do capítulo 5.

5.1. Motivação

O projeto de arquiteturas de *software* que efetivamente capturem os requisitos funcionais envolvidos e que apresentem um atendimento satisfatório dos múltiplos atributos de qualidade em questão não é uma tarefa fácil. Conforme apresentado no Capítulo 2 (Seção 2.3), tais atributos são frequentemente conflitantes e resultam tanto de aspectos técnicos quanto de interesses divergentes oriundos das diversas partes envolvidas no projeto. Uma boa arquitetura deve evidenciar a indução daqueles atributos de qualidade mais prioritários, balancear o atendimento dos atributos secundários, alavancar qualidades através do uso de conhecimento específico de domínio e ser passível de realização com o custo e tempo previstos.

Conforme ilustrado na Figura 2.2, um processo de projeto arquitetural envolve a execução, geralmente iterativa, das fases de análise dos requisitos arquiteturalmente relevantes, tomada de decisão em relação às táticas arquiteturais adotadas e análise das vantagens e desvantagens da arquitetura obtida. Diversas propostas de projeto arquitetural especificam, para cada uma das fases acima, os atores envolvidos, as atividades realizadas e os produtos gerados. Cada uma destas fases trazem desafios (discutidos a seguir) à adoção de um processo efetivo de projeto arquitetural.

5.1.1. Levantamento de Requisitos

Esta fase requer a identificação criteriosa daqueles requisitos com alta relevância em âmbito arquitetural. Requisitos não-funcionais têm impacto direto na realização da fase seguinte (tomada de decisão). Por exemplo, demandas por alta portabilidade podem requerem amplo uso de arquiteturas em camadas, alto desempenho pode sugerir o uso de soluções baseadas em eventos, alta disponibilidade e escalabilidade são comumente obtidas com arquiteturas descentralizadas, etc.

Embora com relevância menor, requisitos funcionais também influenciam aspectos tais como o particionamento do sistema em módulos principais, os fluxos de interação e troca de dados entre componentes e as necessidades de integração com outros sistemas. Em particular, requisitos que impactam atributos de qualidade de domínio específico (vide Seção 2.2) são difíceis de serem identificados por arquitetos sem experiência no domínio em questão. Uma documentação mais sistemática dos espaços de problema e solução envolvidos – conforme apresentada na Seção 3.2 para o caso particular de sistemas *self-adaptive* – pode trazer benefícios em relação a este aspecto.

5.1.2. Tomada de Decisão

Nesta fase, conhecimento refinado de projeto arquitetural é utilizado para selecionar os componentes, conectores, interfaces e configurações que irão constituir uma arquitetura que atenda satisfatoriamente os requisitos identificados na fase anterior. Alguns desafios podem ser identificados na realização desta atividade, descritos a seguir.

Representação sistemática e estruturada de conhecimento de projeto. Esta representação envolve a captura das táticas comumente aplicadas no projeto de boas arquiteturas. O objetivo é documentar o conhecimento refinado – frequentemente obtido com anos de prática – que subsidia a identificação de uma ou mais arquiteturas potencialmente efetivas para o problema em questão.

Tais táticas incluem, por exemplo: *i)* configurações arquiteturais, elementos principais e restrições (documentadas sob a forma de estilos arquiteturais); *ii)* adoção estratégica de tipos particulares de conectores (ex: barramentos de eventos, balanceadores de carga, escalonadores, etc); *iii)* aspectos referentes a implantação (ex: co-localização de componentes com interações frequentes e minimização de pontos únicos de falha); e *iv)* uso de arquiteturas dinâmicas (ex: adaptação estrutural e reconfiguração em tempo de execução, mobilidade de componentes, etc).

A prática comum envolve o uso de arquiteturas de referência e catálogos de estilos e padrões arquiteturais [105, 101, 259, 278]. Embora tais abordagens já tragam benefícios para uma formação mais abreviada de bons arquitetos, a falta de suporte para manipulação direta por ferramentas e a dificuldade de associação e comparação de arquiteturas candidatas limita os benefícios destas abordagens no uso rotineiro pela indústria.

Mapeamento entre táticas e indução de atributos de qualidade. Arquiteturas de qualidade são frutos de táticas intencionalmente aplicadas, resultantes de um conhecimento prévio de que uma tática em particular potencialmente induz o atendimento de algum atributo de qualidade desejado. Sistematizar este mapeamento não é uma tarefa fácil, porém é fundamental para a consolidação de métodos mais disciplinados para projeto de arquiteturas de *software*.

Atributos conflitantes tornam o processo ainda mais difícil pois implicam em efeitos complexos e difíceis de serem previstos. Embora alguns trabalhos atuais [297, 323] já apresentem diretrizes para associação de táticas/estilos arquiteturais a atributos de qualidade impactados, tais recomendações são ainda abstratas e requerem experiência para serem efetivamente utilizadas na prática.

Identificação de arquiteturas alternativas em relação a um determinado trade-off. O uso criterioso das táticas envolve, primeiro, a identificação dos atributos conflitantes e, depois, a seleção de múltiplas arquiteturas que evidenciam o *trade-off* em questão de forma ótima (candidatas pertencentes a um *Pareto-front* preferencialmente global – vide Definição 4.8, Seção 4.2.1,

pág. 71). É necessário ressaltar, neste caso, a existência de múltiplas arquiteturas efetivas. Ainda assim, as táticas são utilizadas para obter-se não múltiplas arquiteturas quaisquer, mas aquelas que caracterizam o *trade-off* de forma ótima e apresentam um bom número de soluções alternativas (*Pareto-front* com alta convergência e alta diversidade – vide Seção 4.2.1, pág. 70).

5.1.3. Análise Arquitetural

A fase de análise tem como objetivo a realização de uma avaliação antecipada da arquitetura em relação a diversos aspectos, dentre eles o potencial de indução – na aplicação final – dos atributos de qualidade que guiaram a fase de tomada de decisão. Alguns desafios presentes neste estágio são descritos a seguir.

Utilização de notações expressivas para modelagem arquitetural. O grau de formalidade e rigor da notação utilizada tem impacto direto nos tipos de operações de análise possíveis de serem realizadas. Análises mais simples incluem a verificação de regras de boa formação de arquiteturas e de compatibilidade de interfaces entre componentes e conectores. Notações mais rigorosas, entretanto, viabilizam a realização de análises mais sofisticadas tais como as comportamentais, as baseadas em simulação ou aquelas para predição de propriedades.

Encontrar um bom equilíbrio, entretanto, entre a curva de aprendizado da notação e os benefícios da análise realizada é fundamental nesta fase. As notações variam desde aquelas que suportam apenas avaliações manuais (ex: linguagem natural e gráficos informais), passando por outras com suporte limitado a análise automatizada (ex: UML), até aquelas com alto rigor formal (ex: ADLs) e, portanto, passíveis de uso em análises mais sofisticadas.

Derivação de modelos precisos para avaliação de atributos. Um atributo de qualidade é quantificado – geralmente de forma indireta – por uma determinada métrica de qualidade. Dessa forma, a validade da predição é ameaçada por fatores tais como: *i)* grau de representatividade do atributo pela métrica; *ii)* corretude na extração e análise dos valores obtidos para a métrica; *iii)* e grau de fidelidade da arquitetura implementada (descritiva) em relação à arquitetura projetada (prescrita).

Diversos modelos têm sido utilizados para quantificar diferentes atributos de qualidade de *software* [162]. Por exemplo, Redes de Filas em Camadas [103] podem ser utilizadas para avaliação de desempenho, modelos POFOD (*Probability Of Failure On Demand*) [96] para quantificar confiabilidade e modelos COCOMO (*COnstructive COst MOdel*) [38] para avaliar custos de implementação de arquiteturas. Um aspecto desafiador, neste cenário, é a realização de bons mapeamentos entre modelos arquiteturais e modelos de predição de atributos de qualidade, de modo a obter-se as vantagens de múltiplas notações. Por exemplo, pode-se utilizar modelos UML anotados (com *profiles*, por exemplo) para representar aspectos arquiteturais estruturais e transferir os valores de tais anotações para modelos mais adequados à predição de propriedades.

Retroalimentação da fase de tomada de decisão. Visto que as fases de tomada de decisão e avaliação são geralmente executadas de forma iterativa, é interessante que a fase de avaliação gere esclarecimentos que conduzam a uma melhoria da arquitetura produzida na próxima iteração. Para isso, é necessário saber não somente os valores das métricas mas os motivos (táticas) que levaram à avaliação obtida.

Comparação de arquiteturas em relação a trade-offs. Investigar se uma arquitetura é melhor que (dominada por) outra ou se ambas evidenciam um *trade-off* (arquiteturas não-dominadas) é

fundamental para a atividade de avaliação arquitetural. Adicionalmente, é interessante verificar quão próximas estas arquiteturas estão de um *Pareto-front* global conhecido (vide Definição 4.8, Seção 4.2.1, pág. 71). A não utilização de mecanismos rigorosos para análise de tais aspectos pode implicar em investimento em arquiteturas inferiores ou no favorecimento não planejado de um atributo de qualidade em detrimento de outros.

Embora os desafios acima discutidos estejam diretamente relacionados à viabilização de processos de projeto arquitetural automatizados, não tem-se a intenção, nesta tese, de propor um processo que elimine por completo análises manuais e investigações mais subjetivas das arquiteturas produzidas. Acredita-se, entretanto, que um maior suporte à representação sistemática de conhecimento de projeto arquitetural e ao projeto e análise automatizados de arquiteturas de domínio específico representa um passo importante para melhorar a efetividade de tais práticas. O benefício é que avaliações manuais subsequentes serão realizadas em arquiteturais já potencialmente efetivas, obtidas como resultado da abordagem aqui proposta.

5.2. Objetivos

Esta tese apresenta o projeto, implementação e avaliação de uma abordagem para projeto arquitetural automatizado de sistemas *self-adaptive*, baseada na representação sistemática de conhecimento refinado de projeto e na aplicação de técnicas de otimização multiobjetivo. A meta foi viabilizar uma exploração mais efetiva do espaço de projeto envolvido na construção de sistemas *self-adaptive* e suportar, de forma mais rigorosa, as decisões referentes aos *trade-offs* presentes.

O objetivo geral foi identificar até que ponto o conhecimento refinado de projeto arquitetural para domínios específicos pode ser sistematicamente estruturado para uso rotineiro e suportado por ferramentas. Dentre os objetivos particulares, destacam-se a investigação de *trade-offs* relacionados aos atributos de qualidade avaliados pelas métricas relacionadas a sistemas *self-adaptive* e a identificação de arquiteturas que atendam a estes atributos de forma ótima. Acredita-se que a sistematização de tais espaços de projeto ofereça suporte valioso a processos de projeto arquitetural que avaliem espaços de solução de uma forma mais completa e disponibilizem subsídios mais rigorosos para identificação de *trade-offs* entre arquiteturas candidatas.

A solução aqui apresentada constitui, tanto quanto sabemos, a primeira abordagem baseada em busca para projeto arquitetural de sistemas *self-adaptive*. Embora apenas uma parte do espaço de solução do domínio seja aqui considerada (Teoria de Controle), acredita-se que os benefícios evidenciados durante a avaliação da abordagem continuarão sendo apresentados em espaços de projeto mais completos, desde que as premissas aqui descritas sejam mantidas. A abordagem de otimização baseada em metaheurísticas adotada é fator fundamental para viabilizar a escalabilidade do método quando aplicado a espaços de projeto de maior cardinalidade.

Para isso, uma infraestrutura genérica para projeto arquitetural automatizado foi proposta – permitindo a exploração de espaços de projeto e otimização multiobjetivo de arquiteturas não só para sistemas *self-adaptive* mas em qualquer domínio de aplicação. A proposta aqui apresentada é fundamentada no uso conjunto de técnicas de três áreas do conhecimento: metamodelagem, otimização multiobjetivo e Teoria de Controle. Tais técnicas são resumidas a seguir.

Metamodelagem. A abordagem apresentada nesta tese tem como objetivo a transformação de um modelo de entrada através da adição de extensões que caracterizam comportamentos de um determinado domínio de aplicação. No caso particular de sistemas *self-adaptive*, o modelo de entrada representa o sistema gerenciado e o modelo de saída consiste no sistema gerenciado estendido com alguma arquitetura específica de sistema gerenciador. O modelo de entrada é anotado com especificações mínimas que viabilizam: *i)* a exploração automática do espaço de

projeto em questão; e *ii)* o uso de otimização multiobjetivo para obtenção das arquiteturas candidatas.

De modo a não limitar a abordagem ao caso particular de sistemas *self-adaptive*, uma infraestrutura genérica para especificação de espaços de projeto (*design spaces*) de domínio específico foi projetada e implementada. Tal infraestrutura demandou a definição de uma nova linguagem de modelagem – denominada **DuSE** – dedicada à especificação de espaços de projeto que sistematicamente capturam, para um determinado domínio de aplicação: *i)* as principais dimensões de projeto; *ii)* os pontos de variação (possíveis soluções) em cada dimensão; *iii)* as extensões arquiteturais requeridas para implementar cada ponto de variação; e *iv)* as métricas de qualidade para avaliação de arquiteturas candidatas.

Todo espaço de projeto (criado para um domínio de aplicação em particular) deve estar associado a um *profile* que define as anotações que suportam o projeto automatizado das arquiteturas finais. A linguagem DuSE foi utilizada para a especificação do **SA:DuSE** – espaço de projeto que captura os principais aspectos de projeto arquitetural de sistemas *self-adaptive* baseados em *feedback control loops*.

Otimização multiobjetivo. Os constructos disponibilizados pela linguagem DuSE permitem representar o problema de projeto arquitetural como um espaço de busca, onde arquiteturas podem ser automaticamente projetadas e métricas de qualidade podem avaliar o *fitness* de cada solução candidata. Tal infraestrutura não só viabiliza a exploração manual do espaço de projeto mas também permite o uso de técnicas de otimização multiobjetivo (vide Seção 4.3.2, pág. 75) para a descoberta automática de soluções pertencentes ao *Pareto-front*.

Embora o uso de metaheurísticas não garanta a obtenção de soluções pertencentes ao *Pareto-front* global, acredita-se que a utilização de algoritmos com alta convergência e alta diversidade produza soluções ainda assim efetivas e que evidenciam com expressividade o *trade-off* em questão. A infraestrutura de metamodelagem utilizada nesta tese permite a reutilização deste mecanismo de otimização de arquiteturas em qualquer domínio para o qual tenha sido criado um espaço de projeto. Embora este trabalho utilize o NSGA-II como *backend* de otimização multiobjetivo, outros algoritmos podem ser facilmente acomodados na solução.

Teoria de controle. Conforme apresentado no Capítulo 3 (Seção 3.3.4), *feedback control loops* baseados em Teoria de Controle representam uma abordagem promissora para o projeto de sistemas *self-adaptive* devido à disponibilidade de uma rigorosa fundamentação matemática para projeto de controladores e análise de propriedades tais como estabilidade, precisão e sobressinal. O espaço de projeto SA:DuSE – uma das contribuições desta tese – captura as principais alternativas de projeto de controladores apresentadas na Seção 3.3.4.3, definindo, para cada possibilidade de controle, as extensões arquiteturais a serem incluídas no modelo inicial que representa o sistema a ser gerenciado.

O SA:DuSE captura as dimensões de projeto mais prevalentes (ex: lei de atuação, técnica de sintonia de controladores, grau de adaptação de controle e aspectos de implantação de *loops*) do domínio e define métricas de qualidade (ex: tempo de estabilização, precisão, sobressinal, *overhead* de controle, etc) utilizadas para avaliação das propriedades de controle.

5.3. Aplicações Exemplo

Para melhor ilustrar os objetivos discutidos na seção anterior, são aqui apresentadas três aplicações exemplo utilizadas ao longo desta tese: duas relacionadas a servidores *web self-adaptive* e a

terceira envolvendo um *cluster* elástico para execução de aplicações (*jobs*) *MapReduce*. Tais aplicações representam cenários com demandas por comportamento de autogerenciamento amplamente discutidos na literatura e que permitem a adoção de sistemas gerenciadores com diferentes arquiteturas.

5.3.1. Servidor Web SISO

A Figura 5.2(a) apresenta um modelo UML de um servidor *web* SISO (*Single-Input Single-Output*) ainda sem capacidades de autogerenciamento (sistema gerenciado somente). Este modelo apresenta um único componente de processo (`WebServer`), anotado com o estereótipo `<<TFProcessComponent>>` de modo a identificá-lo como um sistema dinâmico representado por uma função de transferência (TF – *Transfer Function*). Tal estereótipo define duas propriedades para representação do numerador (*tfNum*) e denominador (*tfDen*) da função de transferência que caracteriza a dinâmica do servidor. Portanto, o servidor a ser gerenciado é um sistema dinâmico de primeira ordem representado pela função de transferência (vide Seção 3.3.4.2, pág. 51):

$$Y(z) = \frac{-0.014}{z - 0.6} \tag{5.1}$$

Note que o sistema gerenciado, sem a atuação de nenhum controlador, é estável pois o seu único polo (0.6) está contido no círculo unitário definido no plano $\mathcal{Z}$ (vide Tabela 3.5, pág. 52). O modelo apresenta ainda duas portas (*ports*) que **disponibilizam** as interfaces `IKATimeout` e `ICpuUtilization`. Conforme apresentado na Figura 5.2(b), a interface `IKATimeout` é anotada com o estereótipo `<<ControllableInterface>>`, indicando que trata-se de uma **interface controlável**, ou seja, um serviço que permite a definição do valor da entrada de controle. Neste caso em particular, a entrada de controle é o ajuste do parâmetro `KeepAliveTimeout`, responsável por especificar quanto tempo a conexão do servidor *web* é mantida mesmo sem haver novas requisições do cliente.

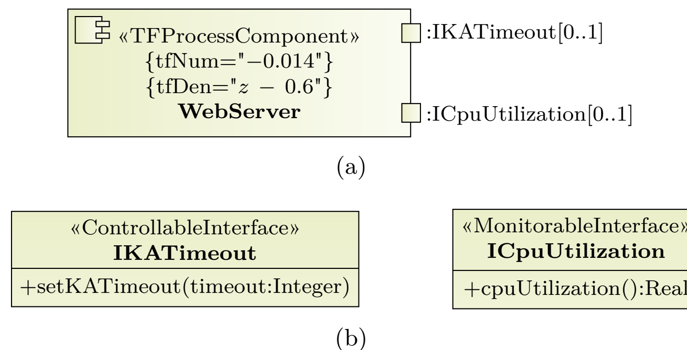


Figura 5.2.: Exemplo de modelo para servidor *web* SISO (a), apresentando suas interfaces monitorável e controlável (b).

Valores baixos de `KeepAliveTimeout` resultam em contínuas execuções das operações de fechamento e reabertura de conexões, aumentando o *overhead* e implicando em um tempo de resposta maior. Valores altos de `KeepAliveTimeout`, por outro lado, resultam em consumo desnecessário de recursos para aqueles clientes que não executarão requisições em um futuro próximo. Encontrar um valor ideal de `KeepAliveTimeout` é uma tarefa difícil pois depende de fatores tais como frequência das demandas e tempo de serviço das requisições – parâmetros que variam frequentemente com o tempo. A solução ideal é dotar o servidor *web* com a capacidade de autogerenciamento contínuo do valor de `KeepAliveTimeout` de modo a manter o tempo de resposta sempre igual a um valor de referência previamente definido, a despeito de perturbações

experimentadas no servidor.

A interface `ICpuUtilization`, por sua vez, atua como uma **interface monitorável** visto que é anotada com o estereótipo `<<MonitorableInterface>>`, ou seja, disponibiliza um serviço que permite a obtenção da saída medida. Neste caso particular, a saída medida é a utilização média da CPU do servidor, variável diretamente impactada pelo valor atribuído ao parâmetro `KeepAliveTimeout` e também por perturbações presentes no sistema. O objetivo de controle, neste exemplo, é rejeitar perturbações ou regular a utilização da CPU através da manipulação do parâmetro `KeepAliveTimeout`.

A interface `IKATimeout` realiza a configuração do valor do parâmetro `KeepAliveTimeout` através da modificação de um arquivo de configuração do servidor *web* (ex: arquivo `httpd-default.conf` do servidor *web* `httpd` da Apache). A interface `ICpuUtilization` obtém o valor médio de utilização da CPU também acessando um arquivo de estatísticas, disponibilizado pelo sistema operacional (ex: arquivo `/proc/stat` em sistemas GNU/Linux).

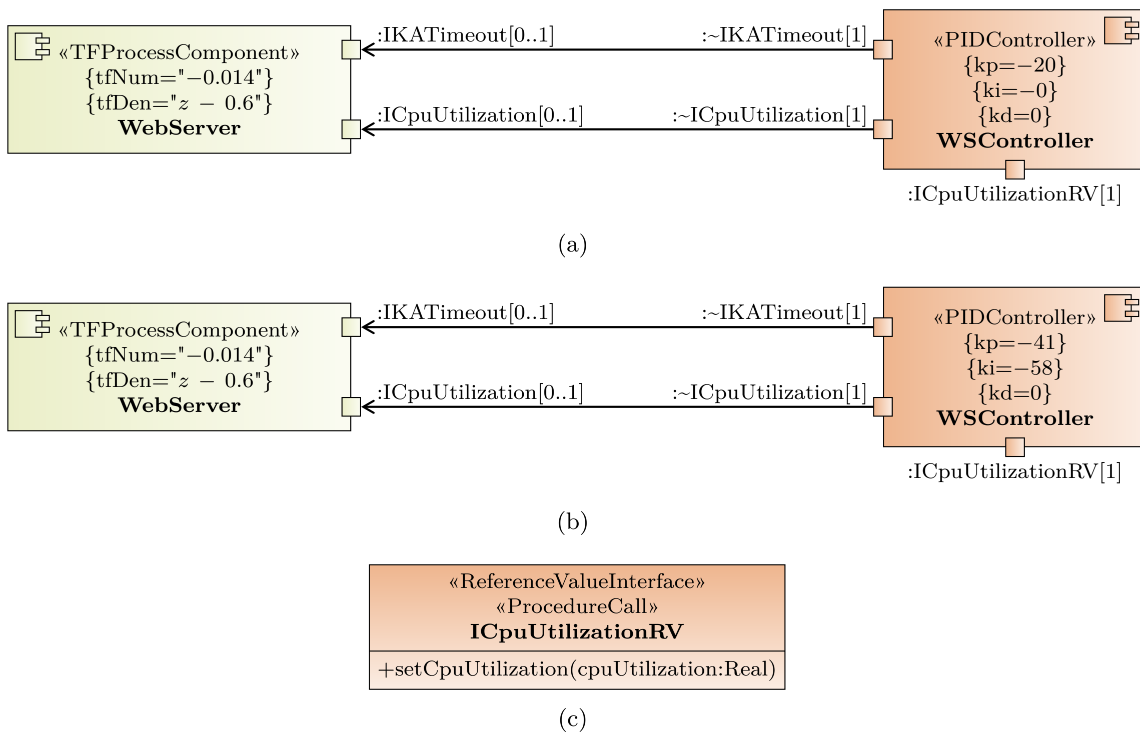


Figura 5.3.: Duas alternativas de controle do servidor *web* SISO: controle P (a) e controle PI (b). Interface para ajuste do valor de referência utilizado pelo controlador (c).

As Figuras 5.3(a) e 5.3(b) apresentam duas possíveis arquiteturas para o sistema *self-adaptive* final, formado pelo sistema gerenciado apresentado na Figura 5.2(a) (modelo de entrada para a abordagem aqui proposta) estendido por uma arquitetura candidata para o sistema gerenciador. Tais arquiteturas ilustram a geração automática de sistemas gerenciadores proposta nesta tese. Na Figura 5.3(a), o servidor *web* é gerenciado por um controlador P (note que $ki = kd = 0$) sem adaptação de controle. A Figura 5.3(b) apresenta o mesmo servidor sendo gerenciado por um controlador PI ($kd = 0$). Os controladores apresentam duas portas que **requerem** (note que as portas são conjugadas – notação "~") as interfaces `IKATimeout` e `ICpuUtilization`. Embora as arquiteturas candidatas sejam estruturalmente idênticas, o componente que representa o controlador é configurado de forma diferente. A Figura 5.3(c) apresenta a interface para ajuste do valor de referência do controlador (`ICpuUtilizationRV`).

Os modelos finais apresentam também as associações que representam os tipos de conectores utilizados para ligar o sistema gerenciador ao sistema gerenciado. Tais tipos são automaticamente inferidos das anotações presentes nas interfaces correspondentes. Outras possibilidades de controle, neste exemplo, incluem: controle PID associado aos diferentes métodos de sintonia apresentados na Seção 3.3.4.4 e às variações de controle adaptativo apresentadas na Seção 3.3.4.3. Diferentes arquiteturas de controle apresentam diferentes valores para os atributos de qualidade envolvidos. A identificação e representação destes *trade-offs*, bem como o suporte à tomada de decisão nestas situações, são alguns dos objetivos principais da abordagem apresentada nesta tese.

5.3.2. Servidor Web MIMO

Um exemplo de modelo para servidor *web* com múltiplas entradas de controle e múltiplas saídas medidas é apresentado na Figura 5.4(a). Neste caso, o componente que representa o servidor é anotado com o estereótipo `<<SSProcessComponent>>`, indicando que a dinâmica do sistema é descrita por um modelo de espaço de estados (SS – *State-Space*). As propriedades A, B e C, definidas pelo estereótipo, armazenam os valores das matrizes do modelo de espaço de estados. Dessa forma, a dinâmica do servidor é dada por:

$$\begin{bmatrix} y_1(k+1) \\ y_2(k+1) \end{bmatrix} = \begin{bmatrix} 0.54 & -0.11 \\ -0.026 & 0.63 \end{bmatrix} \cdot \begin{bmatrix} y_1(k) \\ y_2(k) \end{bmatrix} + \begin{bmatrix} -0.0085 & 0.00044 \\ -0.00025 & 0.00028 \end{bmatrix} \cdot \begin{bmatrix} u_1(k) \\ u_2(k) \end{bmatrix} \tag{5.2}$$

Note que a matriz C é igual à identidade e, portanto, indica que o sinal de saída $y[k]$ é exatamente igual ao vetor de estado $\mathbf{x}[k]$. O modelo apresenta quatro portas que **disponibilizam** as interfaces descritas na Figura 5.2(b) (`IKATimeout`, `ICpuUtilization`, `IMaxRW` e `IMemUtilization`).

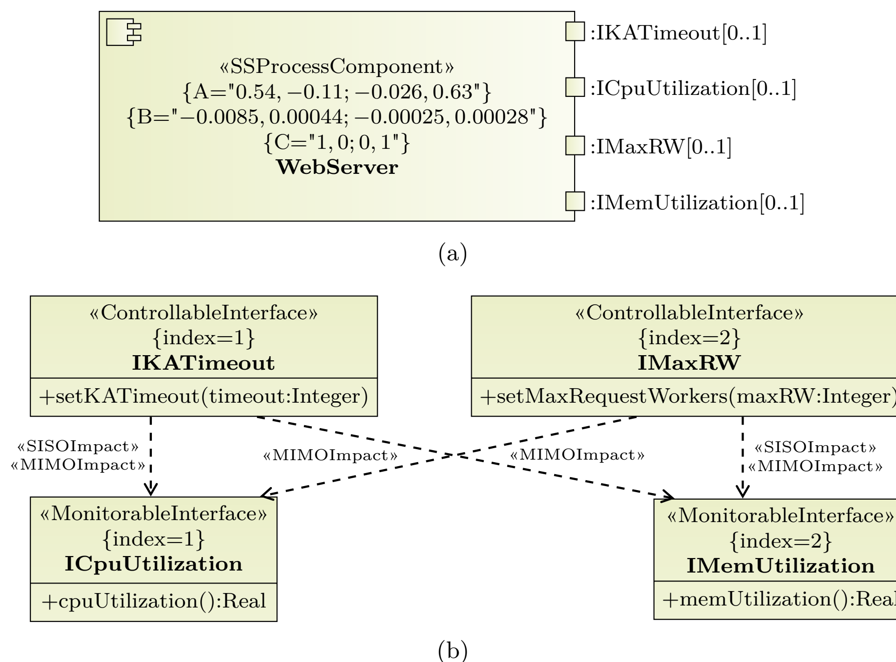


Figura 5.4.: Exemplo de modelo para servidor *web* MIMO (a) com suas interfaces monitoráveis e controláveis (b).

As interfaces `IKATimeout` e `ICpuUtilization` se comportam exatamente como descritas no exemplo anterior. A interface `IMaxRW`, identificada como uma interface controlável, ajusta o parâmetro `MaxRequestWorkers` – referente ao número de *threads* (*request workers* – RW) que atendem requisições de clientes. Já a interface monitorável `IMemUtilization` disponibiliza um

serviço para verificar o consumo de memória do servidor. Evidentemente, ajustes tanto em `KeepAliveTimeout` quanto em `MaxRequestWorkers` impactam diretamente tanto a utilização da CPU quanto o consumo de memória, constituindo um cenário típico para controle MIMO. Entretanto, a alternativa de uso de dois controladores SISO – cada um regulando uma das entradas de controle e a saída medida mais impactada pela entrada utilizada – é comumente considerada na prática.

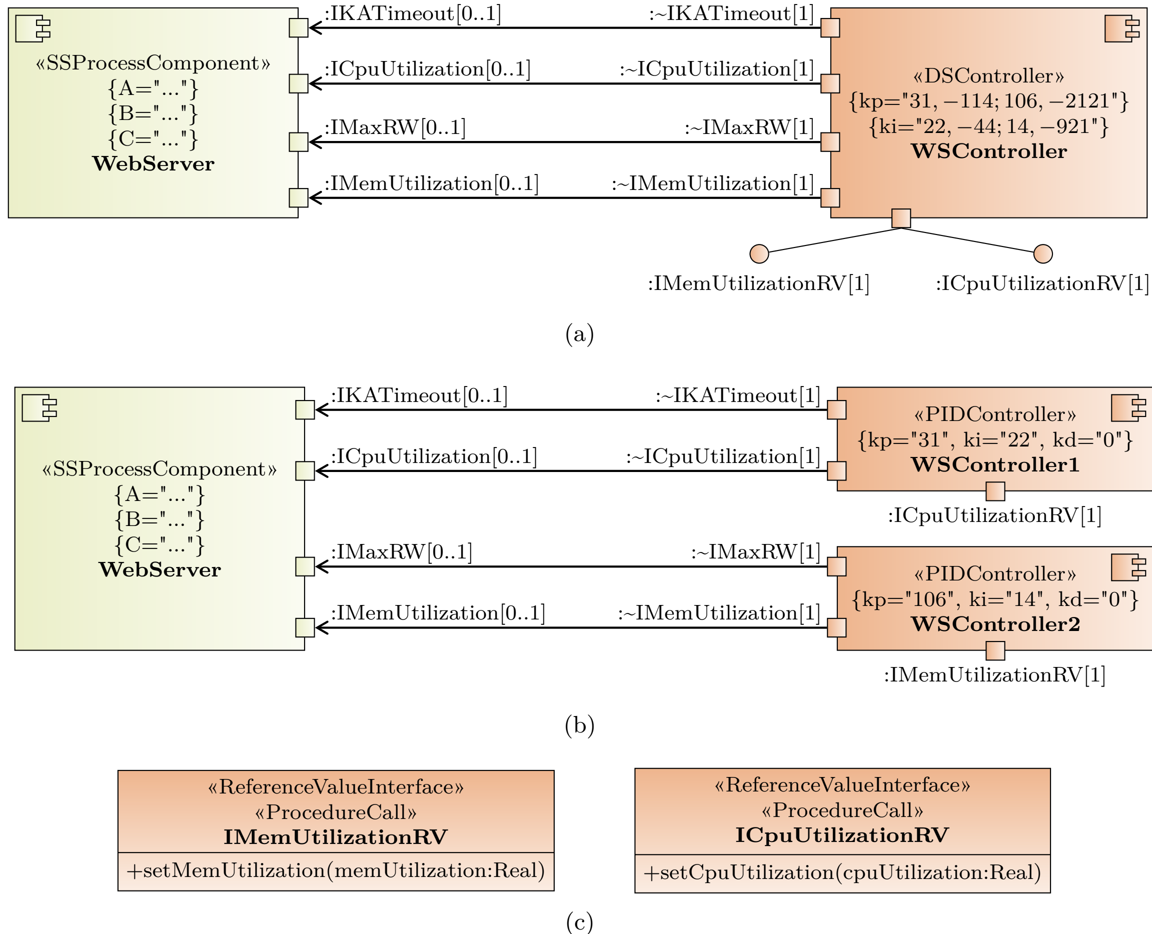


Figura 5.5.: Duas alternativas de controle do servidor *web* MIMO: controle MIMO *dynamic state* (a) e múltiplos controles SISO via PI (b). Interfaces para ajuste dos valores de referência utilizados pelos controladores (c).

Visto que múltiplas entradas e saídas estão envolvidas, é necessário identificar quais interfaces controláveis impactam quais interfaces monitoráveis, tanto nos cenários SISO quanto no cenário MIMO. Para isso, realiza-se a identificação explícita dos impactos entre interfaces através do uso de relacionamentos de dependência anotados. Conforme apresentado na Figura 5.4(b), ao considerar-se controle MIMO sabe-se que a interface `IKATimeout` influencia tanto `ICpuUtilization` quanto `IMemUtilization`. A mesma situação acontece com a interface `IMaxRW` e o estereótipo `<<MIMOImpact>>` é utilizado para este fim. O cenário SISO, por sua vez, é caracterizado pelo impacto mais expressivo de `IKATimeout` em `ICpuUtilization` e de `IMaxRW` em `IMemUtilization`[1]. O estereótipo `<<SISOImpact>>` é utilizado para descrever tais

[1] Embora `IKATimeout` também impacte `IMemUtilization` e `IMaxRW` também impacte `ICpuUtilization`, tais influências são menos expressivas. A técnica *Relative Gain Array* ([33]: Seção 13.4; [25]: pág. 312) pode ser utilizada para verificar os pares [entrada de controle, saída medida] mais importantes.

dependências. Adicionalmente, visto que o metamodelo UML armazena as portas como atributos não ordenados, a propriedade `index` é utilizada para associar `KeepAliveTimeout` a u_1 e `MaxRequestWorkers` a u_2, bem como a utilização da CPU a y_1 e o consumo de memória a y_2.

As Figuras 5.5(a) e 5.5(b) apresentam duas possíveis arquiteturas para o sistema *self-adaptive* final deste exemplo. Na Figura 5.5(a), o servidor *web* é gerenciado por um controlador com *Dynamic State Feedback Control* (*<<DSController>>*). A porta de ajuste dos valores de referência do controlador `WSController` deixa de ser uma porta simples (com uma única interface provida ou requerida) e, portanto, passa a exibir explicitamente as duas interfaces providas.

A Figura 5.5(b) apresenta o mesmo servidor *web* sendo gerenciado por múltiplos controladores PI. Note que, neste caso, as arquiteturas candidatas são estruturalmente diferentes. A Figura 5.5(c) apresenta as interfaces para ajuste dos valores de referência do controlador (`ICpuUtilizationRV` e `IMemUtilizationRV`). Outras possibilidades de controle incluem: *Static State Feedback Control* e múltiplos controladores P ou PID associados aos diferentes métodos de sintonia apresentados na Seção 3.3.4.4 e às variações de controle adaptativo apresentadas na Seção 3.3.4.3. Novamente, diferentes arquiteturas de controle apresentam diferentes valores para os atributos de qualidade envolvidos.

5.3.3. Cluster Elástico para Aplicações MapReduce

Os exemplos apresentados nas seções anteriores ilustram algumas possibilidades de controle de um sistema gerenciado formado por um único componente de processo e, portanto, apenas um *loop* esteve presente no sistema gerenciador. Embora tal cenário já traga desafios para o arquiteto, sistemas *self-adaptive* reais são geralmente caracterizados pela presença de múltiplos componentes de processo, demandando o uso de vários *loops*. Nestes casos, análises mais rigorosas devem ser realizadas de modo a identificar interações não esperadas. Adicionalmente, o projeto do sistema gerenciador passa a considerar um número muito maior de arquiteturas candidatas.

A Figura 5.6(a) apresenta um exemplo de modelo para um *cluster* de computadores utilizado como infraestrutura para execução de *jobs MapReduce*. O *MapReduce* [76] é um estilo arquitetural e um modelo para computação distribuída amplamente utilizado no processamento de grandes bases de dados, com suporte nativo a tratamento de falhas e escalabilidade horizontal[2] do *cluster*. O *MapReduce* assume que os dados a serem processados estão armazenados de forma distribuída (e replicada, para suportar falhas) no *cluster*. Cada nó do *cluster* contém zero ou mais blocos da base de dados. Para cada bloco, uma tarefa de *map* é executada no nó que contém aquele bloco. Um nó específico é automaticamente escolhido para executar a tarefa de *reduce*, que tem como objetivo receber as saídas de cada tarefa de *map* e consolidar o resultado final. Em algumas situações, utiliza-se mais de uma tarefa de *reduce* para melhorar o desempenho. O tempo de resposta do *job* é aquele decorrido desde a solicitação da sua execução até o momento em que a tarefa de *reduce* termina a consolidação das saídas de todas as tarefas de *map*.

Dentre os diversos fatores que impactam o tempo de resposta do *job*, dois são aqui considerados. Primeiro, o número de máquinas no *cluster* afeta diretamente o número de blocos que são processados simultaneamente. Quanto mais máquinas estiverem presentes, blocos com tamanhos menores poderão ser utilizados e, consequentemente, maior será o paralelismo durante o processamento. Embora o ajuste do número de máquinas do *cluster* tenha influência considerável no tempo de resposta, o custo de inclusão de novos nós motiva a investigação de outros parâmetros de impacto.

[2] Aquela caracterizada pelo aumento do poder computacional do *cluster* através da inclusão facilitada (sem impacto nos *jobs* e na infraestrutura de processamento distribuído) de novas máquinas, em contraponto ao *upgrade* das máquinas existentes (escalabilidade vertical).

O segundo fator aqui considerado, portanto, é o número máximo de tarefas simultâneas de *map* executando em cada nó do *cluster*. O autogerenciamento deste parâmetro é interessante em cenários onde o *cluster* é formado por um conjunto heterogêneo de máquinas, com um número variado de núcleos de diferentes velocidades, ou quando perturbações nos nós são esperadas. Visto que um nó pode hospedar mais de um bloco da mesma base de dados sendo processada, a execução simultânea de várias tarefas de *map* reduz o tempo de resposta do *job*, desde que o nó tenha recursos disponíveis (ex: folga na utilização de CPU) para isso.

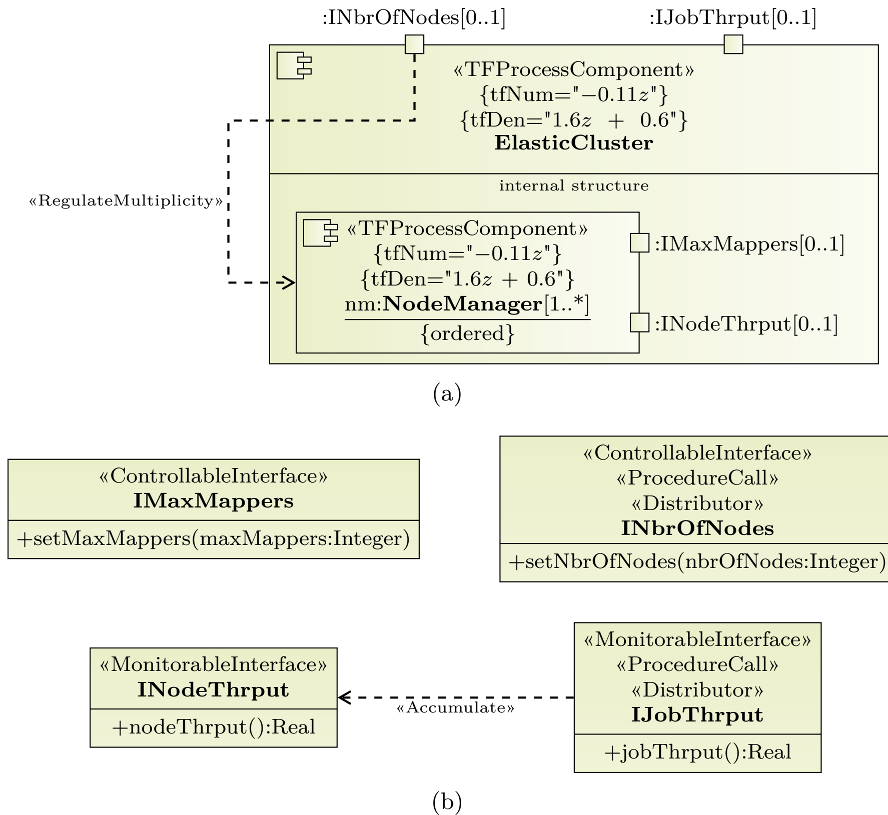
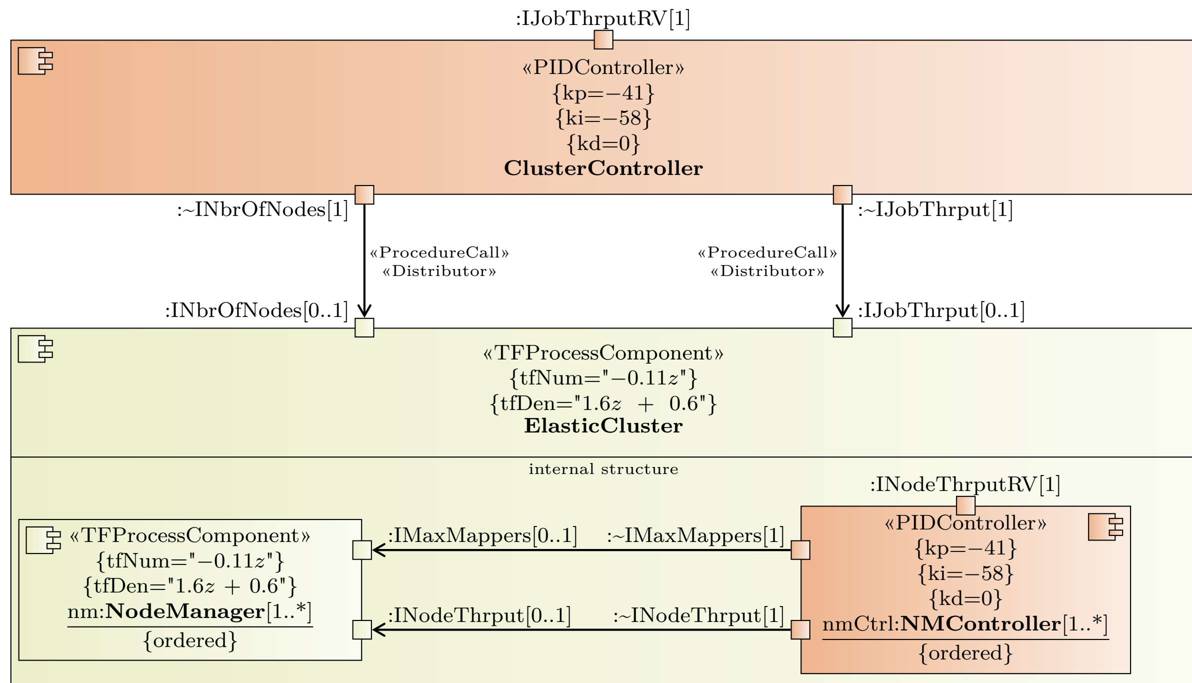


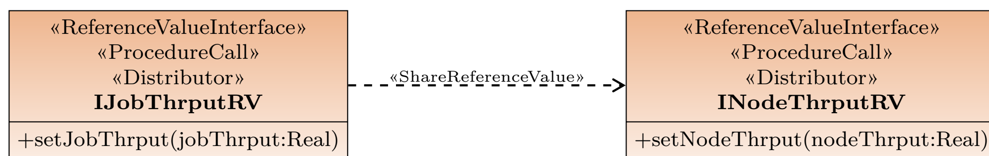
Figura 5.6.: Exemplo de modelo para um *cluster* de computadores utilizado como infraestrutura para execução de aplicações *MapReduce* (a) e suas interfaces monitoráveis e controláveis (b).

A Figura 5.6(a) modela o cenário descrito. O componente `ElasticCluster` representa todo o *cluster* e é estruturado como uma coleção `nm` de uma ou mais (1..*) instâncias do componente `NodeManager`, utilizado para representar cada nó presente no *cluster*. O tempo de resposta do *job* é indiretamente controlado mantendo-se a vazão de processamento do *cluster* em uma taxa fixa desejada (valor de referência). Note que tanto o componente `ElasticCluster` quanto as instâncias de `NodeManager` que representam os nós são anotados como componentes de processo (estereótipo `<<TFProcessComponent>>`).

A Figura 5.6(a) define um cenário onde múltiplos *loops* atuam em dois níveis: *i)* diversos *loops* locais a cada nó de processamento, regulando a vazão local (via porta do tipo `INodeThrput`) ao controlar o número de tarefas simultâneas de *map* (via porta do tipo `IMaxMappers`); e *ii)* um único *loop* atuando *cluster-wide*, regulando a vazão total do *job* (via porta do tipo `IJobThrput`) ao controlar o número total de nós do *cluster* (via porta do tipo `INbrOfNodes`). O estereótipo `<<RegulateMultiplicity>>` é utilizado para indicar que o serviço disponibilizado por `INbrOfNodes` controla o número de instâncias de `NodeManager` que constituem o *cluster*. A Figura 5.6(b) apresenta as interfaces do modelo. O estereótipo `<<Accumulate>>` é utilizado para indicar que o serviço disponibilizado por `IJobThrput` acumula as vazões locais disponibilizadas, por todos os nós, através das portas do tipo `INodeThrput`.



(a)



(b)

= sistema gerenciado (modelo de entrada)

= sistema gerenciador (automaticamente incluído pela abordagem aqui proposta)

Figura 5.7.: Controle do *cluster* elástico através da utilização de controladores PID aninhados.

A Figura 5.7(a) apresenta uma arquitetura candidata de sistema gerenciador para o *cluster* elástico. Nesta solução, controladores PI foram utilizados tanto localmente em cada nó quanto na gerência do número de nós do *cluster*. Para cada `NodeManager` `nm` presente no *cluster* um controlador `nmCtrl` é instanciado. Note que ambas as propriedades `nm` e `nmCtrl` são ordenadas (`{ordered}`) de modo a ter-se os `NodeManagers` e seus respectivos controladores sempre associados.

A Figura 5.7(b) apresenta as interfaces para ajuste dos valores de referência dos controladores. O uso do estereótipo <<ShareReferenceValue>> indica que todos os nós serão responsáveis por prover a mesma fração da vazão total desejada para o *job* (valor dado por $jobThrput/nbrOfNodes$). Evidentemente, esta forma de controle não é interessante para *clusters* formados por máquinas com diferentes capacidades de processamento. Uma arquitetura alternativa consiste em compartilhar o erro de controle entre os nós (utilizando o estereótipo <<ShareControlError>>) a um custo de maior *overhead* de controle na rede. Tais *trade-offs* são descritos em mais detalhes no Capítulo 7. Outras formas de controle incluem o uso de outras técnicas de sintonia ou alguma forma de controle adaptativo tanto nos *loops* locais a cada nó quanto no *loop* geral do *cluster*.

5.4. Processo Automatizado de Projeto Arquitetural de Sistemas Gerenciadores

Esta seção descreve o processo automatizado de projeto arquitetural proposto nesta tese. Este processo suporta o projeto e análise automáticos das diversas arquiteturas candidatas de sistemas gerenciadores, como os apresentados nos exemplos da seção anterior. Adicionalmente, mecanismos para otimização multiobjetivo de arquiteturas viabilizam a descoberta daquelas candidatas pertencentes a um *Pareto-front*, preferencialmente com boa aproximação ao *Pareto-front* global. Embora a infraestrutura que suporta o processo aqui proposto tenha sido utilizada para o projeto automatizado de sistemas *self-adaptive*, a forma com que o processo foi concebido permite a sua aplicação na automatização do projeto de aplicações de qualquer domínio, conforme descrito a seguir.

Um ponto central da abordagem é a utilização de **espaços de projeto** (*design spaces*) estruturados de forma a suportar o projeto automatizado. Um espaço de projeto captura sistematicamente os seguintes aspectos de um domínio de aplicação: *i)* as principais dimensões de projeto (aspectos arquiteturais a serem considerados); *ii)* os pontos de variação (possíveis soluções) de cada dimensão de projeto; *iii)* as modificações arquiteturais necessárias para estender um modelo de entrada de modo a implementar um determinado ponto de variação; e *iv)* um conjunto de métricas para avaliação de atributos de qualidade em arquiteturas candidatas.

De modo a permitir a aplicação da abordagem em diversos domínios de aplicação, uma linguagem de modelagem para especificação de espaços de projeto foi projetada e implementada nesta tese. Tal linguagem – denominada DuSE – é descrita por um metamodelo que define os constructos necessários para a: *i)* especificação de espaços de projeto para domínios específicos; *ii)* exploração automática destes espaços de projeto (geração de arquiteturas candidatas); e *iii)* utilização de algoritmos de otimização para obtenção de soluções pertencentes ao *Pareto-front*.

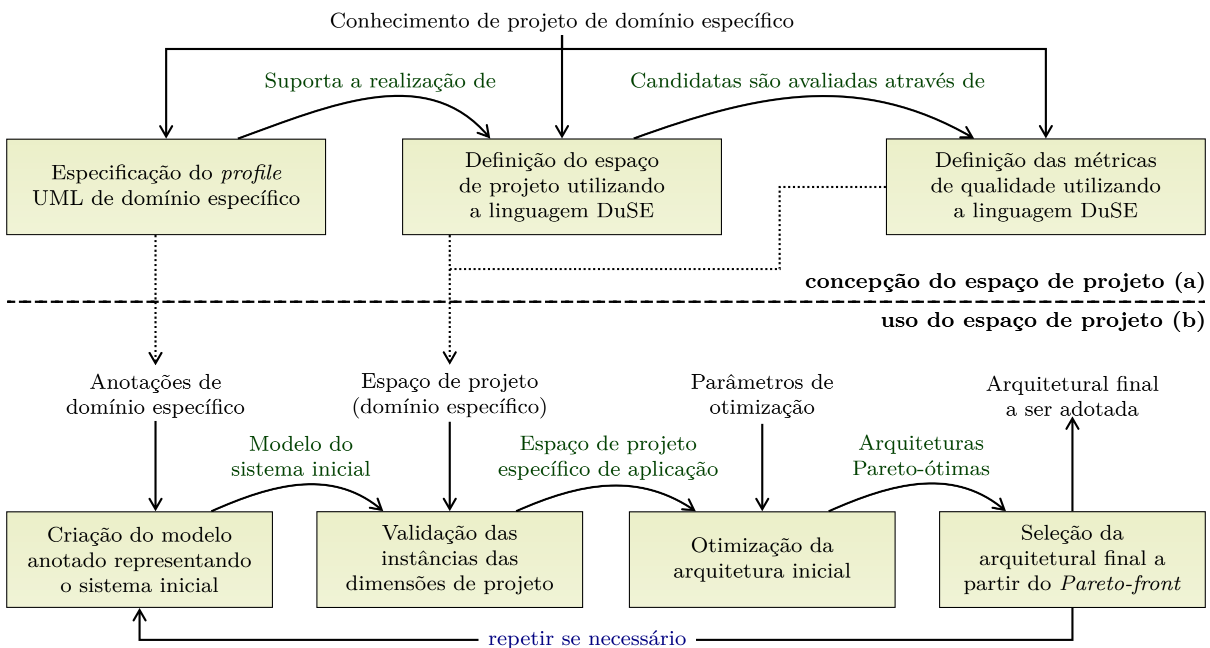


Figura 5.8.: Processo automatizado de projeto arquitetural proposto nesta tese.

A Figura 5.8 apresenta as atividades e produtos previstos no processo proposto. Duas grandes fases são definidas: concepção do espaço de projeto (executada uma vez por domínio de aplicação) e uso do espaço de projeto (executada uma vez para cada sistema do domínio de aplicação em questão).

Na fase de concepção do espaço de projeto (parte *a* da Figura 5.8), um arquiteto com experiência no domínio de aplicação em questão executa três tarefas principais. A primeira tarefa é a especificação do *profile* UML que suportará tanto o projeto automatizado quanto a avaliação das métricas que representam atributos de qualidade de interesse. No caso dos sistemas *self-adaptive* apresentados na seção anterior, este *profile* contém os estereótipos (ex: `<<TFProcessComponent>>`, `<<PIDController>>` etc) utilizados para anotar o modelo inicial que representa o sistema a ser controlado. A segunda atividade é a definição do espaço de projeto. Neste momento, especifica-se as dimensões de projeto mais prevalentes no domínio, os pontos de variação associados e as modificações arquiteturais correspondentes. Finalmente, na última tarefa, define-se como as métricas serão calculadas a partir de um determinado modelo resultante do processo automatizado de projeto. A linguagem DuSE oferece todos os constructos necessários à representação do espaço de projeto e das métricas associadas.

Uma vez criado o espaço de projeto, o projeto automatizado de sistemas do domínio em questão pode ser realizado a partir das quatro tarefas descritas na fase de uso do espaço de projeto (parte *b* da Figura 5.8). A primeira tarefa é a modelagem do sistema de entrada. Vale ressaltar que o modelo de entrada não está limitado à utilização somente de componentes. Esta abordagem é utilizada no projeto automatizado de sistemas *self-adaptive* porque o objetivo é estender a arquitetura de um sistema já existente (sistema gerenciado) com novos componentes implementando os *loops* de controle (sistema gerenciador). Espaços de projeto para outros domínios de aplicação podem, por exemplo, receber diagramas de casos de uso ou diagramas de sequência como entrada e produzir, como saída, diagramas de componentes representando possíveis arquiteturas. Qualquer constructo da UML é suportado pela abordagem aqui proposta.

A segunda tarefa da fase de uso do espaço de projeto é a derivação do espaço de projeto específico de aplicação. Conforme descrito anteriormente, cada dimensão de projeto representa um aspecto arquitetural que demanda uma tomada de decisão (escolha de algum ponto de variação desta dimensão de projeto). Por exemplo, no caso particular de sistemas *self-adaptive*, uma dimensão de projeto é a lei de atuação a ser utilizada no controlador e possíveis pontos de variação desta dimensão são: P, PI, PD, PID, *Static State Feedback Control* e *Dynamic State Feedback Control*. Note que uma mesma dimensão de projeto pode precisar ser avaliada múltiplas vezes para o mesmo modelo de entrada. Por exemplo, o modelo do *cluster* elástico apresentado na Seção 5.3.3 apresenta dois componentes de processo (`ElasticCluster` e as propriedades do tipo `NodeManager`) e, portanto, a dimensão referente à lei de atuação precisa ser avaliada duas vezes. Para isso, duas instâncias dessa dimensão de projeto são criadas para o modelo de entrada do *cluster* elástico. O conjunto formado por todas as instâncias de dimensões de projeto, para um modelo de entrada particular, é denominado **espaço de projeto específico de aplicação**.

A terceira tarefa é a fase de otimização da arquitetura inicial. Neste momento, um determinado algoritmo de otimização multiobjetivo é executado de modo a obter-se as arquiteturas candidatas pertencentes ao *Pareto-front*. A abordagem aqui descrita não está limitada a um algoritmo de otimização em particular, embora o NSGA-II seja atualmente utilizado para este fim. Nesta tarefa, os parâmetros do algoritmo de otimização sendo utilizado deverão ser ajustados. Para o NSGA-II, tais parâmetros incluem o número máximo de gerações, bem como os tipos de operadores e as probabilidades de recombinação e mutação a serem utilizados. Finalmente, na última tarefa, o arquiteto adota algum critério – não definido nesta tese – para escolha de uma das arquiteturas presentes no *Pareto-front* gerado na tarefa de otimização.

Os capítulos seguintes desta tese detalham a infraestrutura necessária à realização das tarefas acima citadas. Uma descrição mais completa e rigorosa da representação sistemática de espaços de projeto é realizada no Capítulo 6, enquanto o espaço de projeto para o domínio particular de sistemas *self-adaptive* é apresentado no Capítulo 7. O Capítulo 8 detalha como as técnicas de otimização multiobjetivo foram utilizadas para automatizar a exploração dos espaços de projeto e identificar arquiteturas pertencentes ao *Pareto-front*.

5.5. Premissas e Limitações

A efetividade do processo de projeto arquitetural automatizado apresentado na seção anterior depende do atendimento das seguintes premissas:

- **P1** – Um modelo inicial representando o sistema a ser projetado está disponível. O modelo inicial pode variar desde simples casos de uso até arquiteturas complexas a serem modificadas. Em um cenário ideal, o espaço de projeto sendo utilizado deve requerer somente as anotações minimamente necessárias para subsidiar a derivação do espaço de projeto específico de aplicação e a execução da otimização. Com isso, diminui-se a complexidade do modelo utilizado como entrada do processo. Acredita-se, entretanto, que – para espaços de projeto bem formulados – o benefício da identificação automática das melhores arquiteturas candidatas justifica o custo de aprendizado do *profile* UML do domínio e da criação dos modelos de entrada necessários.

- **P2** – Os atributos de qualidade de interesse podem ser quantitativamente avaliados. As métricas utilizadas para avaliar os atributos de qualidade são fundamentais para guiar a busca por soluções efetivas. Alguns atributos de qualidade (ex: segurança e usabilidade) são difíceis de serem avaliados quantitativamente e, portanto, improváveis de serem utilizados na abordagem aqui apresentada. Entretanto, qualquer modelo para avaliação quantitativa de atributos de qualidade pode ser aplicado no processo aqui descrito, desde expressões simples para investigação de anotações até modelos mais complexos, tais como Redes de Filas em Camadas ou simulações.

- **P3** – As dimensões de projeto do domínio podem ser sistematicamente documentadas. A criação de novos espaços de projeto é uma tarefa que requer experiência no domínio em questão e habilidades para encontrar dimensões de projeto com boa ortogonalidade e seus respectivos pontos de variação. Alguns domínios de aplicação são caracterizados por dimensões difíceis de serem separadas ou por pontos de variação cuja implementação (mudanças arquiteturais necessárias) não pode ser descrita de forma genérica. Nestes casos, a utilização da abordagem aqui proposta fica prejudicada.

- **P4** – Algum critério para escolha da arquitetura final a partir do *Pareto-front* está disponível. O Capítulo 4 apresentou as desvantagens dos métodos para otimização com articulação *a priori* de preferências, motivando as técnicas com articulação *a posteriori* aqui adotadas. Entretanto, o objetivo final continua sendo a seleção de uma única arquitetura a ser efetivamente implementada. O processo aqui proposto produz como resultado um conjunto de arquiteturas candidatas igualmente ótimas em relação aos atributos de qualidade do espaço de projeto em uso (soluções do *Pareto-front* encontrado). Uma arquitetura particular deve ser selecionada deste conjunto, através de técnicas tais como *utility functions* (aplicadas ao *Pareto-front*, em vez de todo o espaço de busca) ou processos manuais e/ou qualitativos.

Algumas limitações podem ser identificadas para o processo automatizado de projeto aqui proposto. Primeiro, embora a abordagem possa ser adaptada para outros meta-metamodelos, a linguagem DuSE e a infraestrutura de suporte por ferramenta desenvolvidas nesta tese são fortemente baseadas na linguagem MOF (*Meta Object Facility*) [231]. Entretanto, a MOF (bem como a UML) é uma linguagem com ampla aceitação na indústria e frequentemente utilizada como base para especificação de novas linguagens de modelagem.

Segundo, o espaço de projeto para o domínio de sistemas *self-adaptive* aqui proposto se limita ao uso de Teoria de Controle como mecanismo viabilizador de autogerenciamento. Outros mecanismos podem ser investigados no futuro, bastando para isso a criação de novas dimensões de projeto. A abordagem evolucionária para otimização aqui adotada é particularmente adequada à

exploração de grandes espaços de busca. Espera-se que novas dimensões, portanto, não tenham impacto expressivo na qualidade das arquiteturas encontradas.

Finalmente, o uso de técnicas de otimização multiobjetivo limita a abordagem de projeto arquitetural àquelas situações onde o problema pode ser modelado como um espaço de busca associado a um conjunto de funções de *fitness* (métricas de qualidade). Abordagens mais qualitativas, mesmo que parcialmente automatizadas, podem contribuir para uma melhor efetividade do processo.

5.6. Resumo do Capítulo

Este capítulo apresentou as motivações, os objetivos e a visão geral do processo automatizado de projeto arquitetural proposto nesta tese, bem como exemplos de aplicações com demandas de autogerenciamento. Foram identificados os principais desafios presentes nas fases de levantamento de requisitos, projeto arquitetural (tomada de decisão) e avaliação arquitetural. Tais desafios constituem parte da motivação para construção de um processo de projeto arquitetural mais quantitativo, suportado por ferramentas e que viabilize a representação sistematizada de conhecimento refinado de domínio específico. As aplicações exemplo apresentadas ilustram algumas das diversas possibilidades de arquiteturas para sistemas gerenciadores, evidenciando a necessidade de um suporte mais rigoroso para identificação e análise de *trade-offs*.

O abordagem aqui proposta disponibiliza a infraestrutura necessária para representação sistemática de conhecimento refinado de projeto arquitetural para aplicações de um determinado domínio. Tal representação é realizada através de espaços de projeto que identificam as principais dimensões de projeto (decisões arquiteturais a serem tomadas), os pontos de variação (possíveis soluções) em cada dimensão e as modificações arquiteturais necessárias à implementação de cada ponto de variação. Adicionalmente, métricas para avaliação quantitativa de atributos de qualidade são também especificadas como parte do espaço de projeto. A construção do espaço de projeto é requerida uma vez por domínio de aplicação e deve ser realizada por um arquiteto com experiência neste domínio.

Uma vez criado o espaço de projeto, a construção de arquiteturas para aplicações daquele domínio passa a ser melhor suportada, uma vez que tem-se à disposição um conjunto valioso de possibilidades de solução que – quando combinadas de forma não tão óbvia para arquitetos iniciantes – passam a evidenciar arquiteturas bastante efetivas. Um espaço de projeto constitui uma representação de conhecimento de projeto arquitetural de domínio específico altamente refinada e passível de ser consumida por ferramentas de suporte ao processo automatizado. Um espaço de projeto é um artefato a ser constantemente corrigido e evoluído, de modo a capturar novas dimensões de projeto, disponibilizar arquiteturas candidatas mais eficientes e adotar métricas de qualidade mais efetivas e precisas.

O Capítulo 6 apresenta a formalização da infraestrutura genérica para espaços de projeto aqui proposta. O suporte matemático para identificação de *trade-offs* e arquiteturas ótimas apresentado no Capítulo 8 contribui para uma melhor fundamentação das tomadas de decisão, ampliação das alternativas consideradas e minimização de tendências (*bias*) por arquiteturas inferiores anteriormente utilizadas. Tais benefícios são difíceis de serem obtidos pelas técnicas comumente adotadas para representação de conhecimento de projeto tais como os catálogos de estilos arquiteturais ou arquiteturas de referência.

Capítulo 6

Infraestrutura Independente de Domínio para Espaços de Projeto

Everything has been composed, just not yet written down.
Wolfgang Amadeus Mozart

Este capítulo apresenta a infraestrutura independente de domínio para representação de espaços de projeto proposta nesta tese. Conforme descrito no Capítulo 5, esta infraestrutura permite a captura sistemática das principais dimensões de projeto (graus de liberdade) de um determinado domínio de aplicação, as soluções alternativas (pontos de variação) de cada dimensão de projeto, as modificações arquiteturais necessárias à implementação de cada ponto de variação e as métricas de qualidade a serem utilizadas para avaliar cada arquitetura candidata. Adicionalmente, a formalização de espaços de projeto apresentada neste capítulo subsidia a derivação do espaço de projeto para sistemas *self-adaptive* apresentado no Capítulo 7 e a adoção dos mecanismos para otimização multiobjetivo de arquiteturas apresentados no Capítulo 8.

Os principais requisitos para projeto arquitetural automatizado são discutidos na Seção 6.1, enquanto a formalização do espaço de projeto é apresentada na Seção 6.2. A Seção 6.3 apresenta os algoritmos para projeto arquitetural automatizado propostos nesta tese. Finalmente, a Seção 6.4 discute as premissas e limitações da proposta, enquanto um resumo do capítulo é apresentado na Seção 6.5. A Figura 6.1 apresenta o roteiro deste capítulo.

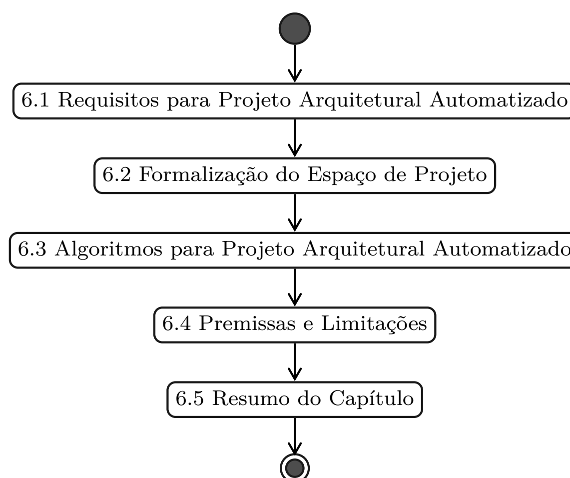


Figura 6.1.: Roteiro do capítulo 6.

6.1. Requisitos para Projeto Arquitetural Automatizado

A realização das atividades de projeto arquitetural automatizado descritas nesta tese requer a existência de uma infraestrutura que capture sistematicamente as alternativas de projeto e viabilize as modificações arquiteturais necessárias à implementação de cada uma dessas alternativas. Para isso, um conjunto de cinco requisitos para projeto arquitetural automatizado são identificados nesta seção. Tais requisitos são importantes para garantir que a abordagem aqui proposta seja passível de uso em outros domínios de aplicação (além dos sistemas *self-adaptive* aqui considerados) e em modelos descritos por outros metamodelos (além dos metamodelos MOF e UML aqui utilizados). Adicionalmente, requisitos para tratamento de mudanças conflitantes viabilizam o uso da abordagem em domínios caracterizados por uma alta dependência entre decisões de projeto. Estes requisitos são descritos em mais detalhes a seguir.

R1 – expressividade de mudanças. As mudanças necessárias à implementação de alguma tática arquitetural em sistemas de um determinado domínio podem variar desde simples modificações de parâmetros até operações mais complexas, tais como substituição de componentes, criação ou remoção de componentes/conectores e modificações na topologia da arquitetura. Para suportar o projeto automatizado em sistemas de qualquer domínio de aplicação, deve-se ser capaz de representar mudanças nos diferentes níveis de abstração acima exemplificados. De um modo geral, alterações em propriedades de um elemento do modelo representam as modificações mais elementares (com menor granularidade possível) e alterações mais sofisticadas são descritas como um conjunto de modificações elementares.

R2 – independência de modelo e dependência de metamodelo. A abordagem aqui apresentada tem como objetivo a realização de modificações arquiteturais em qualquer modelo que represente um sistema do domínio de aplicação em questão. Uma infraestrutura para projeto arquitetural automatizado deve permitir a especificação de espaços de projeto que capturem modificações arquiteturais para qualquer aplicação do domínio em questão (independência de modelo). Tais modificações devem ser descritas com base em algum metamodelo (vide Seção 2.4, pág. 20) particular, o mesmo utilizado para descrever os diversos modelos de entrada que representam as aplicações do domínio. Embora um espaço de projeto em particular esteja restrito à manipulação de modelos descritos no metamodelo considerado (dependência de metamodelo), uma infraestrutura para projeto arquitetural automatizado deve permitir a criação de espaços de projeto associados a qualquer metamodelo. Este requisito permite a adequação da abordagem aqui proposta para o projeto automatizado de modelos descritos em linguagens outras além da MOF e da UML.

R3 – ortogonalidade de decisões arquiteturais. Arquiteturas efetivas surgem frequentemente da combinação bem orquestrada de decisões referentes a diversos aspectos do sistema em questão. Uma parte importante da captura sistemática de conhecimento de projeto para domínios específicos é a fatoração e agrupamento destas decisões em relação a aspectos funcionais e não-funcionais do domínio (dimensões de projeto). É uma atividade que requer experiência no projeto de arquiteturas em geral e, em particular, de soluções para o domínio sendo considerado. A identificação explícita de tais dimensões de projeto já traz benefícios substanciais, visto que evita que decisões referentes a certas dimensões menos óbvias sejam consequências não propositais de outras decisões. Uma solução (arquitetura candidata) particular requer a escolha de uma solução para cada dimensão de variação identificada para o domínio em questão. A união de todas as soluções parciais de cada dimensão define a arquitetura candidata final.

R4 – tratamento de mudanças conflitantes. Uma consequência do requisito R3 é que, eventualmente, nem todas as combinações das soluções escolhidas para as diversas dimensões de projeto envolvidas representarão arquiteturas válidas. Embora recomende-se a definição de dimensões de projeto com maior ortogonalidade possível, mudanças decorrentes de soluções de dimensões de projeto diferentes podem ser conflitantes e produzir arquiteturas inválidas. Uma infraestrutura para projeto arquitetural automatizado deve viabilizar a detecção de tais conflitos e a identificação das arquiteturas resultantes de tais combinações como inválidas. Um espaço de projeto deve tentar minimizar a ocorrência de arquiteturas inválidas através da derivação de dimensões de projeto com maior ortogonalidade possível, embora tal situação seja difícil em alguns domínios de aplicação.

R5 – conformidade com metamodelo. Em adição à detecção de mudanças conflitantes descrita no requisito R4, modelos resultantes do projeto automatizado devem estar em conformidade com o metamodelo utilizado para descrever o modelo inicial e o espaço de projeto sendo utilizado. Note que o requisito R5 garante conformidade com o metamodelo em uso porém não garante validade dos aspectos de domínio específico envolvidos (este problema é coberto no requisito R4). Adicionalmente, o *profile* UML definido para o espaço de projeto em questão pode definir novas restrições que devem ser atendidas por qualquer modelo resultante do processo automatizado.

6.2. Formalização do Espaço de Projeto

Esta seção apresenta os conceitos e formalizações necessários à representação de espaços de projeto de forma independente de domínio de aplicação. Inicialmente, as definições de metamodelo arquitetural e modelo arquitetural são apresentadas, seguidas das formalizações que suportam a definição de modificações arquiteturais, pontos de variação, dimensões de projeto, métricas de qualidade e espaços de projeto.

Definição 6.1: Metamodelo Arquitetural

Um **metamodelo arquitetural** MM é um conjunto de tipos de elementos arquiteturais. Cada tipo t presente em MM define um conjunto de propriedades, denotado por $props(t)$. Cada propriedade p de t, por sua vez, possui um tipo, uma forma de agregação e uma multiplicidade, definidos como:

$$\forall t \in MM, \forall p \in props(t)\colon type(p) = pt \mid pt \in MM \vee pt = Any \tag{6.1}$$

$$\forall t \in MM, \forall p \in props(t)\colon aggr(p) = a \mid a \in \{shared, composite\} \tag{6.2}$$

$$\forall t \in MM, \forall p \in props(t)\colon mult(p) = m \mid m \in \{single, multiple\} \tag{6.3}$$

Um elemento de um modelo arquitetural (definido a seguir) é sempre uma instância de algum tipo presente no metamodelo arquitetural correspondente. A definição acima permite a especificação de modelos arquiteturais como estruturas hierárquicas, onde um tipo representa uma metaclasse do metamodelo arquitetural e define propriedades cujos valores devem ser instâncias de um tipo particular, também presente no metamodelo arquitetural. Assume-se que o tipo *Any* está presente em qualquer metamodelo arquitetural e é utilizado para aquelas propriedades que não possuem restrições de tipo (propriedades não-tipadas). A condição 6.1 acima define esta situação. Um metamodelo arquitetural será, daqui para frente, denominado simplesmente metamodelo.

Propriedades podem ter forma de agregação *shared* ou *composite* (condição 6.2). Propriedades *composite* definem uma relação de parentesco entre instâncias dos tipos do metamodelo, facilitando a navegação entre elementos e a manipulação de modelos. A remoção de um elemento

particular e implica na remoção de todos os elementos que são valores de propriedades *composite* de e. Propriedades *shared*, por outro lado, atuam como simples referências para elementos do modelo e não implicam em nenhuma relação de parentesco.

Finalmente, a multiplicidade de cada propriedade pode ser única (*single*) ou múltipla (*multiple*) (condição 6.3). Propriedades *single* podem armazenar zero ou uma instância do tipo para ela definido. Propriedades *multiple*, por outro lado, podem armazenar qualquer número de instâncias do tipo para ela definido.

Definição 6.2: Modelo Arquitetural

Seja M um conjunto de elementos arquiteturais e MM um metamodelo. Cada elemento arquitetural e presente em M possui um tipo, denotado por $type(e)$ e armazena, para cada propriedade p presente em $props(type(e))$ de MM, um conjunto de valores, denotado por $vals(p, e)$.

Seja $parent(e)$ o conjunto formado pelas tuplas $\langle u, p\rangle$ dos elementos u de M que contêm e como valor de alguma propriedade *composite* p:

$$parent(e) = \{ \langle u, p\rangle \in \left\langle M, \bigcup_{i\in M} props(i) \right\rangle \mid (\\ p \in props(type(u)) \wedge aggr(p) = composite \wedge e \in vals(p, u))\} \tag{6.4}$$

Seja $children(e)$ o conjunto dos valores das propriedades *composite* de e:

$$children(e) = \{c \in M \mid (\exists p \in props(type(e)) \mid (\\ aggr(p) = composite \wedge c \in vals(p, e)))\} \tag{6.5}$$

M é um **modelo arquitetural** em conformidade com um metamodelo MM (situação denotada por $M \blacktriangleleft MM$) se as seguintes condições são satisfeitas:

$$\forall e \in M : type(e) \in MM \tag{6.6}$$

$$\forall e \in M, \forall p \in props(type(e)), \forall v \in vals(p, e): \\ v \in M \wedge (type(p) = type(v) \vee type(p) = Any) \tag{6.7}$$

$$\forall e \in M, \forall p \in props(type(e)) : (mult(p) = single) \implies |vals(p, e)| \leq 1 \tag{6.8}$$

$$\forall e \in M : \ |parent(e)| \leq 1 \tag{6.9}$$

$$\forall e \in M : e \notin \pi_u(parent(e)) \tag{6.10}$$

$$|\{e \in M \mid parent(e) = \emptyset\}| = 1 \tag{6.11}$$

O elemento raiz do modelo M é definido por $root(M) = r \in M \mid parent(r) = \emptyset$.

Para simplificação da notação, $e.p$, $e.parent$ e $e.children$ serão utilizados, daqui para frente, para denotar $values(p, e)$, $parent(e)$ e $children(e)$, respectivamente. Adicionalmente, $e.parent.u$ e $e.parent.p$ serão utilizados para denotar as projeções $\pi_u(e.parent)$ e $\pi_p(e.parent)$, respectivamente (o mesmo vale para as demais tuplas daqui para frente). Um modelo arquitetural será denominado, daqui para frente, simplesmente modelo.

A condição 6.6 define que todo elemento do modelo M deve ser uma instância de algum tipo presente no metamodelo MM associado. Um elemento e armazena valores para cada propriedade p definida em $type(e)$. A condição 6.7 garante que todos os valores, em e, de uma propriedade p sejam instâncias do tipo definido, no metamodelo, para esta propriedade (exceto quando a propriedade é do tipo *Any*).

A condição 6.8 constata se todas as propriedades *single* possuem no máximo um elemento em seu conjunto de valores. A condição 6.9 especifica que cada elemento do modelo deve possuir

zero ou um elemento pai. A condição 6.10 informa que um elemento não pode ser pai dele mesmo, enquanto a condição 6.11 informa que somente um elemento deve atuar como elemento raiz do modelo.

Para exemplificar as definições acima apresentadas, considere o modelo cuja representação gráfica (sintaxe concreta) é apresentada na Figura 6.2 (versão simplificada do modelo para o servidor *web* SISO, apresentado no Capítulo 5, pág. 97). Este modelo é composto por um elemento `WebServer` (instância de `TFProcessComponent`), que possui duas portas (instâncias de `Port`) cujas funcionalidades providas são definidas por `IKATimeout` (instância de `ControllableInterface`) e `ICpuUtilization` (instância de `MonitorableInterface`). Adicionalmente, uma dependência (instância de `SISOImpact` – vide Seção 3.3.4.3, pág. 54) é definida entre `IKATimeout` e `ICpuUtilization`. Todos os elementos acima descritos são especificados como parte do pacote `GlobalPackage` (instância de `Package`).

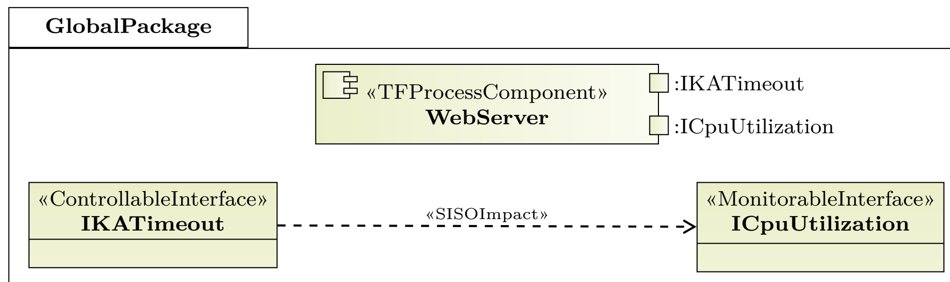


Figura 6.2.: Representação gráfica (sintaxe concreta) do modelo arquitetural utilizado para exemplificar a formalização do espaço de projeto.

A Tabela 6.1 apresenta o metamodelo que suporta o modelo apresentado na Figura 6.2. Oito diferentes tipos de elementos são definidos (apresentados na coluna "$t \in MM$"). O tipo `Package` define uma única propriedade, *ownedElements*, que armazena – com forma de agregação *composite* – múltiplas instâncias de qualquer tipo (*Any*). Os tipos `ControllableInterface` e `MonitorableInterface`[1] (vide Seção 5.3.1, pág. 97) não definem propriedades. O tipo `SISOImpact` define duas propriedades – *source* e *destination* – que armazenam, respectivamente, as instâncias de `ControllableInterface` e de `MonitorableInterface` envolvidas no impacto SISO. Tanto `TFProcessComponent` quanto `PIDController` definem uma propriedade denominada *ports*, com forma de agregação *composite*, multiplicidade *multiple* e tipo `Port`.

O tipo `Port`, por sua vez, define a propriedade *type*[2], que indica o elemento do modelo que especifica os serviços oferecidos pela porta. O tipo `Connector` define duas propriedades – *source* e *destination* – ambas do tipo `Port`. Embora não sejam utilizados no modelo apresentado na Figura 6.2, instâncias dos tipos `PIDController` e `Connector` serão utilizadas – pela abordagem automatizada de projeto – para suportar, respectivamente, a criação de controladores e a conexão destes ao `WebServer`.

A Tabela 6.2 apresenta a sintaxe abstrata (instâncias de metaclasses) do Modelo cuja representação gráfica é ilustrada na Figura 6.2. A primeira coluna ($e \in M$) apresenta os elementos e suas relações de parentesco. As propriedades do tipo *composite* são apresentadas de forma sublinhada. O elemento `GlobalPackage` atua como elemento pai de `IKATimeout`, `ICpuUtilization`, `TimeoutCpuImpact` e `WebServer`. `WebServer`, por sua vez, atua como elemento pai das portas `pKATimeout` (cujo valor da propriedade *type* é `IKATimeout`) e `pCpuUtilization` (cujo valor da propriedade *type* é `ICpuUtilization`).

[1] Estereótipos são aqui considerados como metaclasses para simplificar a ilustração da formalização.

[2] Não confundir a propriedade *type* de `Port`, que associa um elemento do modelo a outro elemento do modelo, com a função *type(e)* – restrição 6.1 da Definição 6.1 – que realiza um cruzamento de metaníveis ao associar um elemento *e* do modelo a um elemento *type(e)* do metamodelo.

$t \in MM$	$p \in props(t)$	$type(p)$	$aggr(p)$	$mult(p)$
Package	ownedElements	*Any*	composite	multiple
ControllableInterface				
MonitorableInterface				
SISOImpact	source	ControllableInterface	shared	single
	destination	MonitorableInterface	shared	single
TFProcessComponent	ports	Port	composite	multiple
PIDController	ports	Port	composite	multiple
Port	type	*Any*	shared	single
Connector	source	Port	shared	single
	destination	Port	shared	single

Tabela 6.1.: Metamodelo do modelo apresentado na Figura 6.2.

$e \in M$	$type(e)$	$p \in props(type(e))$	$vals(p, e)$
GlobalPackage	Package	ownedElements	{IKATimeout, ICpuUtilization, TimeoutCpuImpact, WebServer}
├ IKATimeout	ControllableInterface		
├ ICpuUtilization	MonitorableInterface		
├ TimeoutCpuImpact	SISOImpact	source	IKATimeout
		destination	ICpuUtilization
└ WebServer	TFProcessComponent	ports	{pKATimeout, pCpuUtilization}
├ pKATimeout	Port	type	IKATimeout
└ pCpuUtilization	Port	type	ICpuUtilization

Tabela 6.2.: Modelo (sintaxe abstrata) do sistema apresentado na Figura 6.2.

`IKATimeout` e `ICpuUtilization` são definidas, respectivamente, como interfaces anotadas com o tipo `ControllableInterface` e `MonitorableInterface`. Dessa forma, podem atuar como valores para as propriedades *source* e *destination*, respectivamente, de `TimeoutCpuImpact` (instância de `SISOImpact`), conforme definido no metamodelo.

Definição 6.3: Modificação Arquitetural

Uma **modificação arquitetural** c é uma operação indivisível que, quando aplicada a um modelo $M \blacktriangleleft MM$, produz um modelo $M' \blacktriangleleft MM$ tal que $M' \neq M$. Cada modificação arquitetural c possui um tipo, definido por:

$$type(c) = t \mid t \in \{elementsAddition, elementsRemoval, propertyChange\} \tag{6.12}$$

Uma modificação arquitetural (daqui para frente, simplesmente modificação) do tipo *elementsAddition* representa a adição de um ou mais novos elementos ao modelo, todos do mesmo tipo. O tipo dos elementos adicionados (pertencente a MM) é definido em $addedType(c)$. A quantidade de elementos criados é dada pelo número de elementos retornados por $guideElementsExp(c)$ – quando ausente, assume-se que um único elemento será criado. Os novos elementos criados podem ser acessados, em outras modificações, através da expressão $addedElements(c)$.

De forma semelhante, uma modificação do tipo *elementsRemoval* realiza a remoção de um ou mais elementos do modelo. Denota-se por $removedElementsExp(c)$ a expressão que define os elementos a serem removidos. Assume-se que os elementos removidos são automaticamente excluídos de todas as propriedades (*composite* ou não) que os referenciam.

Uma modificação do tipo *propertyChange* realiza a alteração de uma propriedade de um ou mais elementos do modelo. Denota-se por $changedPropertyExp(c)$ e $propertyValuesExp(c)$ as expressões que definem a propriedade a ser modificada e o novo valor (ou valores caso possua multiplicidade *multiple*) a ser atribuído à propriedade em questão, respectivamente.

Adicionalmente, a aplicação de qualquer modificação c está condicionada à avaliação verdadeira da pré-condição definida em $preCondExp(c)$. Sejam $removedElements(c)$, $changedProperty(c)$, $propertyValues(c)$ e $preCond(c)$ os resultados das avaliações de $removedElementsExp(c)$, $changedPropertyExp(c)$, $propertyValuesExp(c)$ e $preCondExp(c)$ em M, respectivamente. Uma modificação c é valida para aplicação em M, produzindo M' (situação denotada por $M \xrightarrow{c} M'$), se as seguintes condições forem verdadeiras:

$$M' \neq M \tag{6.13}$$
$$M \blacktriangleleft MM \implies M' \blacktriangleleft MM \tag{6.14}$$
$$type(c) = elementAddition \implies addedType(c) \in MM \tag{6.15}$$
$$type(c) = propertyChange \implies \forall v \in propertyValues(c): \tag{6.16}$$
$$type(changedProperty(c)) = type(v) \vee type(changedProperty(c)) = Any$$
$$preCond(c) = true \tag{6.17}$$

As condições 6.13 e 6.14 indicam que a modificação deve produzir um modelo M' diferente do modelo M original e em conformidade com o metamodelo MM que suportou a construção de M. A condição 6.15 define que, para modificações de adição de novos elementos, tais elementos devem ser instâncias de tipos definidos no metamodelo MM. Na condição 6.16, verifica-se se os valores resultantes da avaliação de $propertyValues(c)$ são do tipo especificado, no metamodelo, para a propriedade em questão (exceto quando a propriedade é do tipo *Any*). Finalmente, a condição 6.17 verifica se a pré-condição da modificação é satisfeita.

Definição 6.4: Ponto de Variação

Um **ponto de variação** é uma tupla $vp = \langle C, postCondExp \rangle$ onde C é um conjunto não-vazio e ordenado de modificações e $postCondExp$ é um predicado de pós-condição. Este predicado é avaliado após a aplicação ordenada, em um modelo M, de todas as modificações em C. Denota-se por $M \xrightarrow{c_1 \circ \cdots \circ c_i} M_{i+1}$ a aplicação consecutiva e válida, em M, das modificações de C:

$$M \xrightarrow{c_1 \circ \cdots \circ c_i} M_{i+1} \implies M \xrightarrow{c_1} M_2 \wedge M_2 \xrightarrow{c_2} M_3 \wedge \ldots \wedge M_i \xrightarrow{c_i} M_{i+1} \tag{6.18}$$

Seja $postCond(vp)$ o resultado da avaliação de $postCondExp(vp)$ em M_{i+1}. Um ponto de variação vp é válido para aplicação em um modelo M produzindo M' (situação denotada por $M \xrightarrow{vp} M'$) se as seguintes condições são verdadeiras:

$$M \xrightarrow{\circ_{c_i \in vp}} M_{i+1} \tag{6.19}$$
$$postCond(vp) = true \tag{6.20}$$

Um ponto de variação agrega as modificações arquiteturais necessárias à implementação de uma determinada solução de uma dimensão de projeto. Para que um determinado ponto de variação seja válido para aplicação em um modelo M, todas as modificações constituintes devem ser sequencialmente válidas para aplicação nos modelos intermediários (gerados após a aplicação de cada modificação em particular). Esta situação é descrita na condição 6.19. O uso das pré-condições definidas pelas modificações, em conjunto com a pós-condição definida para o ponto de variação, permite a verificação do modelo resultante em cada etapa intermediária. Tal recurso permite uma detecção mais precisa de combinações inválidas de pontos de variação.

Modificações de Inclusão de Elementos		
#c	Metapropriedade	Especificação OCL
1	$addedType(c)$	PIDController
	$guideElements(c)$	ausente
2	$addedType(c)$	Port
	$guideElements(c)$	target.ports
3	$addedType(c)$	Connector
	$guideElements(c)$	target.ports

Tabela 6.3.: Exemplo de modificações de inclusão de elementos na especificação de ponto de variação para inclusão de controlador.

Modificações de Alteração de Propriedades - VP21		
#c	Metapropriedade	Especificação OCL
4	$changedPropertyExp(c)$	addedElements(c2).type
	$propertyValuesExp(c)$	guideElements(c2)[index].type
5	$changedPropertyExp(c)$	addedElements(c1).ports
	$propertyValuesExp(c)$	addedElements(c2)
6	$changedPropertyExp(c)$	addedElements(c3).source
	$propertyValuesExp(c)$	addedElements(c2)[index]
7	$changedPropertyExp(c)$	addedElements(c3).destination
	$propertyValuesExp(c)$	guideElements(c2)[index]
8	$changedPropertyExp(c)$	target.parent.ownedElements
	$propertyValuesExp(c)$	target.parent.ownedElements ∪ addedElements(c1) ∪ addedElements(c3)

Tabela 6.4.: Exemplo de modificações de alteração de propriedades na especificação de ponto de variação para inclusão de controlador.

As Tabelas 6.3 e 6.4 apresentam um exemplo de especificação de ponto de variação para adição de um controlador ao servidor *web* apresentado na Figura 6.2. O ponto de variação inclui três modificações do tipo *elementsAddition* (Tabela 6.3) e cinco modificações do tipo *propertyChange* (Tabela 6.4). A primeira modificação realizada é a inclusão de uma nova instância do tipo `PIDController`, definido no metamodelo apresentado na Tabela 6.1. Visto que a expressão *guideElementsExp* não está definida, apenas uma instância do tipo `PIDController` é criada.

Em seguida, na modificação 2, são criadas tantas novas instâncias de `Port` quantas forem as portas presentes no elemento-alvo (identificado por *target*) considerado no momento. O conceito de elemento-alvo será apresentado a seguir, durante a definição de dimensão de projeto. Seu papel é viabilizar a aplicação de um ponto de variação em qualquer local do modelo de entrada que demande a avaliação do aspecto arquitetural em questão. Para o exemplo sendo aqui

investigado, o elemento-alvo é o `WebServer`, o que faz com que o controlador possua o mesmo número de portas que o servidor *web* possui (duas, no exemplo aqui utilizado). A modificação 3 cria uma instância de `Connector` para cada porta presente no servidor *web*. Tais instâncias ligarão as portas do servidor *web* às portas correspondentes do controlador.

As próximas modificações do ponto de variação são do tipo *propertyChange*. A modificação 4 ajusta a propriedade *type* de cada porta criada na modificação 2 para o mesmo valor desta propriedade na porta que serviu como elemento-guia para a criação dos elementos da modificação 2. A expressão *propertyValuesExp* é avaliada uma vez para cada propriedade definida em *changedPropertyExp*. Em cada avaliação, o elemento-guia que originou a propriedade pode ser acessado através do identificador *guideElements(c2)[index]*. Implementações da infraestrutura aqui descrita devem garantir que tais conjuntos sejam mantidos ordenados.

A modificação 5 ajusta a propriedade *ports* do controlador recém-criado (*controller*) para o conjunto formado pelas duas portas recém-criadas – disponíveis em *addedElements(c2)*. Note que a propriedade *ports* é do tipo *composite* e, portanto, *controller* passa a atuar como elemento pai das portas recém-criadas. As modificações 6 e 7 conectam as portas do controlador às portas correspondentes do servidor *web*. Para isso, ajusta-se a propriedade *source* do conector para a porta adicionada correspondente (acessível via identificador *addedElements(c2)[index]*). Adicionalmente, ajusta-se a propriedade *destination*, do mesmo conector, para a porta que atuou como elemento-guia da modificação de criação do elemento (acessível via identificador *guideElements(c2)[index]*). Finalmente, a modificação 8 indica que o controlador e conectores recém-criados fazem parte do pacote `GlobalPackage`, que passa a atuar como elemento pai daqueles. A Figura 6.3 apresenta a representação gráfica do modelo arquitetural resultante da aplicação do ponto de variação descrito nas Tabelas 6.3 e 6.4.

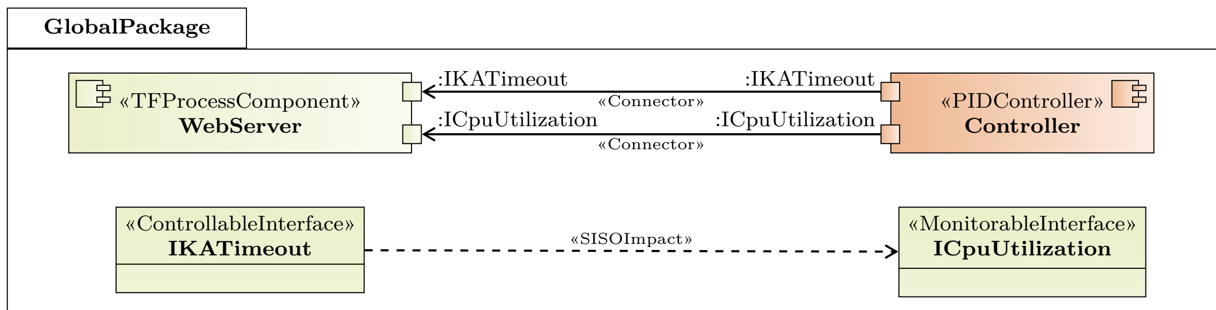


Figura 6.3.: Representação gráfica (sintaxe concreta) do modelo arquitetural resultante da aplicação do ponto de variação apresentado nas Tabelas 6.3 e 6.4.

Definição 6.5: Dimensão de Projeto

Uma **dimensão de projeto** é uma tupla $dd = \langle VP, targetElementsExp \rangle$, onde VP é um conjunto não-vazio e ordenado de pontos de variação e $targetElementsExp$ é uma expressão que, quando avaliada em um modelo M, retorna o conjunto formado pelos elementos de M que demandam uma decisão referente a dd (escolha de um ponto de variação em VP). Tais elementos são denominados **elementos-alvo** da dimensão de projeto dd.

Uma dimensão de projeto representa um determinado aspecto arquitetural a ser avaliado durante o projeto, em um local específico da arquitetura representada pelo modelo inicial. Esta avaliação envolve a escolha de uma solução (ponto de variação) dentre um conjunto de alternativas. Dessa forma, para cada dimensão de projeto dd, o conjunto de pontos de variação que representam soluções alternativas para o aspecto capturado na dimensão de projeto é representado em VP. Visto que múltiplos elementos arquiteturais do modelo de entrada podem demandar a resolução da mesma dimensão de projeto, a expressão que realiza a detecção deste elementos demandantes

é especificada em $targetElementsExp$. A obtenção de uma arquitetura candidata válida requer a escolha de um ponto de variação válido da dimensão de projeto em questão para cada elemento-alvo resultante da avaliação de $targetElementsExp$.

Por exemplo, uma arquitetura inicial apresentando dois elementos controláveis (duas instâncias de `TFProcessComponent`) requer a escolha de um tipo de controlador para cada um destes elementos. Estes dois elementos atuarão como elementos-alvo da mesma dimensão de projeto responsável pela escolha de um controlador. Será criada, para cada elemento-alvo de uma dimensão de projeto, uma instância de dimensão de projeto correspondente (conceito apresentado mais à frente).

Definição 6.6: Espaço de Projeto

Um **espaço de projeto** é uma tupla $ds = \langle DD, QM, P \rangle$, onde DD é um conjunto não-vazio e ordenado de dimensões de projeto; QM é um conjunto não-vazio e ordenado de métricas de qualidade; e P é um *profile* que define as anotações a serem utilizadas nos modelos arquiteturais iniciais e nas extensões arquiteturais criadas automaticamente pelo espaço de projeto.

Um espaço de projeto captura, para um determinado domínio de aplicação: *i)* um conjunto de aspectos arquiteturais que demandam uma decisão (dimensões de projeto); *ii)* um conjunto de métricas de qualidade (conceito apresentado mais à frente) para avaliação das arquiteturas candidatas automaticamente construídas durante a exploração do espaço de projeto; e *iii)* um *profile* contendo as anotações disponíveis para o domínio de aplicação em questão. Note que duas dimensões de projeto diferentes podem selecionar um mesmo elemento arquitetural como elemento-alvo. Tal situação é esperada e indica que decisões referentes a dois aspectos arquiteturais (preferencialmente) ortogonais deverão ser tomadas sobre este mesmo elemento.

Por exemplo, cada elemento controlável (instância de `TFProcessComponent`) do modelo arquitetural de entrada deve demandar a escolha de uma lei de controle (vide Seção 3.3.4.3, pág. 54) e também de uma técnica de sintonia do controlador (vide Seção 3.3.4.4, pág. 59). Portanto, duas dimensões de projeto (lei de controle e técnica de sintonia) deverão prover extensões e configurações arquiteturais a um mesmo elemento de entrada.

As escolhas de uma lei particular de controle e de uma técnica particular de sintonia são representadas pela seleção de um ponto de variação em cada dimensão de projeto correspondente. Algumas combinações poderão ser inválidas e são detectadas pelos predicados de pré- e pós-condição das modificações e pontos de variação envolvidos. Um espaço de projeto é criado uma única vez por domínio de aplicação, preferencialmente por um arquiteto experiente em aplicações deste domínio.

Um espaço de projeto documenta, de forma sistemática, os aspectos arquiteturais a serem avaliados nas aplicações do domínio em questão, as métricas de qualidade a serem utilizadas e – para cada aspecto arquitetural (dimensão de projeto) definido – as expressões que selecionam, no modelo arquitetural inicial, os elementos que demandam a avaliação daquela dimensão de projeto. A necessidade de avaliação (escolha de ponto de variação) de uma dimensão de projeto dd em um elemento-alvo te é denotada pela criação de uma **instância de dimensão de projeto**.

Definição 6.7: Instância de Dimensão de Projeto

Uma **instância de dimensão de projeto** é uma tupla $ddi = \langle M, dd, te \rangle$, onde M é um modelo arquitetural, dd é uma dimensão de projeto e te é um elemento de M que demanda uma decisão referente a dd (escolha de um ponto de variação em $dd.VP$). Seja $dd.targetElements(M)$ o resultado da avaliação de $dd.targetElementsExp$ em M. A seguinte condição deve ser verdadeira para qualquer instância de dimensão de projeto:

$$te \in dd.targetElements(M) \tag{6.21}$$

O elemento te pode ser acessado em expressões de qualquer modificação de qualquer ponto de variação de dd através do identificador "*target*".

Note que o conceito de instância de dimensão de projeto só pode ser considerado ao aplicar um espaço de projeto dd a um modelo arquitetural inicial M. Toda instância de dimensão de projeto representa a necessidade de avaliação de alguma dimensão de projeto dd em algum elemento arquitetural te presente no modelo inicial (condição 6.21). Uma instância de dimensão de projeto é criada para cada elemento-alvo retornado pela avaliação da expressão $targetElementsExp$ da dimensão de projeto em questão.

Vale ressaltar que uma instância de dimensão de projeto atua no mesmo nível de metamodelagem que uma dimensão de projeto, embora a nomenclatura utilizada (instância) possa sugerir o contrário. Enquanto as dimensões de projeto são criadas uma vez por domínio de aplicação, as instâncias de dimensão de projeto são automaticamente criadas – pela infraestrutura aqui proposta – para cada modelo arquitetural inicial sendo utilizado. O conjunto formado por todas as instâncias de dimensão de projeto criadas para um modelo inicial M com base em um espaço de projeto ds define um **espaço de projeto específico de aplicação**.

Definição 6.8: Espaço de Projeto Específico de Aplicação

Um **espaço de projeto específico de aplicação** é uma tupla $asds = \langle M, ds, DDI \rangle$, onde M é um modelo arquitetural inicial, ds é um espaço de projeto e DDI é um conjunto parcialmente ordenado de instâncias de dimensão de projeto, definido como:

$$DDI = \bigcup^{dd \in ds.DD} \left(\bigcup^{te \in dd.targetElements(M)} ddi = \langle M, dd, te \rangle \right) \tag{6.22}$$

Um espaço de projeto específico de aplicação identifica, para cada dimensão de projeto do espaço de projeto em questão, os elementos do modelo inicial que demandam avaliação daquela dimensão de projeto (elementos-alvo). Uma instância de dimensão de projeto é criada em DDI para cada par ⟨dimensão de projeto, elemento-alvo⟩ encontrado. Consequentemente, uma dimensão de projeto pode apresentar zero ou mais instâncias de dimensão de projeto em DDI, dependendo do modelo arquitetural inicial e das expressões $targetElementsExp$ de cada dimensão de projeto.

Frequentemente, $targetElementsExp$ realiza a identificação de elementos-alvo através da procura por anotações (ex: estereótipos UML) particulares. Modelos iniciais sem anotações ou com anotações incorretas podem inviabilizar a identificação dos elementos-alvo e, portanto, produzir conjuntos DDI vazios. A hipótese de que modelos iniciais minimamente anotados estão disponíveis é uma das premissas assumidas neste trabalho. Entretanto, recomenda-se a criação de espaços de projeto que demandem não mais que o minimamente necessário em relação às anotações requeridas para a instanciação das dimensões de projeto.

Por outro lado, modelos iniciais que apresentem muitos elementos-alvo para cada dimensão de projeto implicam na geração de um conjunto DDI de alta cardinalidade. Consequentemente,

o número de arquiteturas candidatas possíveis pode rapidamente se tornar muito grande. Note que o número de instâncias de dimensão de projeto criadas para um modelo inicial M e um espaço de projeto ds é dado por:

$$|DDI| = \sum_{dd \in ds.DD} |dd.targetElements(M)| \tag{6.23}$$

Definição 6.9: Espaço de Decisão Arquitetural e Vetor Candidato

---

O **espaço de decisão arquitetural** $\mathcal{D}_{asds}$ para um espaço de projeto específico de aplicação $asds$ é o produto cartesiano dos índices dos pontos de variação das dimensões de projeto associadas a cada instância de dimensão de projeto presente em $asds.DDI$:

$$\begin{aligned} \mathcal{D}_{asds} \quad = \quad & \{1, 2, \ldots, |asds.DDI_1.dd.VP|\} \times \\ & \{1, 2, \ldots, |asds.DDI_2.dd.VP|\} \times \\ & \ldots \\ & \{1, 2, \ldots, \left|asds.DDI_{|DDI|}.dd.VP\right|\} \end{aligned} \tag{6.24}$$

Um **vetor candidato** é um vetor $\mathbf{x} \in \mathcal{D}_{asds}$. As seguintes condições devem ser verdadeiras para qualquer vetor candidato:

$$dim(x) = |asds.DDI| \tag{6.25}$$

$$\forall x_t \text{ em } \mathbf{x}\colon x_t \in \{1, \ldots, |asds.ddi_t.dd.VP|\} \tag{6.26}$$

Um vetor candidato representa uma possível escolha de pontos de variação para cada instância de dimensão de projeto presente no espaço de projeto específico de aplicação $asds$. A condição 6.25 indica que um vetor candidato deve apresentar uma seleção de ponto de variação para cada instância de dimensão de projeto presente em $asds.DDI$. A condição 6.26 indica que cada índice x_t presente em $\mathbf{x}$ deve ser um índice de ponto de variação válido. O número total de vetores candidatos diferentes em $\mathcal{D}_{asds}$ é dado por:

$$\prod_{dd \in ds.DD} |dd.VP|^{|dd.targetElements(M)|} \tag{6.27}$$

Definição 6.10: Arquitetura Candidata

Uma **arquitetura candidata** é o modelo arquitetural resultante da aplicação válida, em um modelo arquitetural inicial M, dos pontos de variação identificados por um vetor candidato $\mathbf{x} \in \mathcal{D}_{asds=\langle M, ds, DDI\rangle}$.

Seja $asds.ddi_t$ a t-ésima instância de dimensão de projeto do conjunto $asds.DDI$. Seja $asds.ddi_t.vp_{x_t}$ o ponto de variação do conjunto $ddi_t.VP$ identificado pelo t-ésimo índice presente no vetor candidato $\mathbf{x}$ (x_t). Uma arquitetura candidata M_c é válida se a aplicação consecutiva de todos os pontos de variação $asds.ddi_t.dd.vp_{x_t}$ for possível em $asds.M$ (assume-se $M_1 = asds.M$):

$$\forall x_t \in x\colon M_t \xrightarrow{asds.ddi_t.dd.vp_{x_t}} M_{t+1} \tag{6.28}$$

Note que cada ponto de variação acima é aplicado ao *target* correspondente, definido em $asds.ddi_t.te$.

Denota-se por $M \xrightarrow{vp_1 \circ \cdots \circ vp_t} M_{t+1}$ a aplicação consecutiva e válida, em M, de um conjunto VP de pontos de variação:

$$M \xrightarrow{vp_1 \circ \cdots \circ vp_t} M_{t+1} \implies M \xrightarrow{vp_1} M_2 \wedge M_2 \xrightarrow{vp_2} M_3 \wedge \ldots \wedge M_t \xrightarrow{vp_t} M_{t+1} \tag{6.29}$$

Finalmente, a arquitetura candidata M_c é aquela tal que:

$$asds.M \xrightarrow{\circ_{x_t \in x}(asds.DDI_t.dd.vp_{x_t})} M_c \tag{6.30}$$

Note que a definição acima de arquitetura candidata inclui somente aquelas arquiteturas válidas, resultantes de aplicações de pontos de variação (e suas respectivas modificações) cujas pré- e pós-condições foram avaliadas como verdadeiras. As arquiteturas que não atendem a estas condições são denominadas **arquiteturas inválidas**. Um vetor candidato $\mathbf{x}$, portanto, pode estar relacionado tanto a uma arquitetura candidata quanto a uma arquitetura inválida.

Definição 6.11: Espaço Arquitetural Viável

O **espaço arquitetural viável** $\mathcal{F}_{asds}$ para um espaço de projeto específico de aplicação $asds$ é o subconjunto de $\mathcal{D}_{asds}$ que contém apenas vetores candidatos associados a arquiteturas candidatas:

$$\mathcal{F}_{asds} = \{f \mid f \in \mathcal{D}_{asds} \wedge (\exists M_c \mid asds.M \xrightarrow{\circ_{f_t \in f}(asds.DDI_t.dd.vp_{f_t})} M_c)$$

O espaço arquitetural viável é formado somente pelas arquiteturas candidatas (válidas) de $\mathcal{D}_{asds}$. Espaços de projeto caracterizados por dimensões de projeto com alta ortogonalidade apresentam um número reduzido de arquiteturas inválidas. Entretanto, conflitos entre pontos de variação são inevitáveis em alguns domínios e, portanto, admite-se a existência de $\mathcal{F}_{asds}$ como um espaço com dimensão menor que aquela de $\mathcal{D}_{asds}$.

Definição 6.12: Métrica de Qualidade (adaptada de [39], cap. 2, pág. 7)

Uma **métrica de qualidade** é uma uma tupla $qm = \langle \Phi, g \rangle$. Φ é uma função $\Phi\colon \mathcal{F}_{asds} \rightarrow \mathcal{V}$, onde $\mathcal{F}_{asds}$ é um espaço arquitetural viável e $\mathcal{V}$ é um conjunto com escala no mínimo intervalar. g assume os valores 1 e -1 para indicar se a métrica é melhorada com valores maiores (1: a meta é maximizá-la) ou menores (-1: a meta é minimizá-la).

A função Φ é definida por uma expressão $qualityMetricExp(\Phi, M_c)$ que avalia o valor da métrica qm na arquitetura candidata M_c.

Note que a função Φ da métrica de qualidade (daqui para frente, simplesmente métrica) não precisa estar definida para arquiteturas inválidas. Para cada arquitetura candidata, a função Φ deve associar um único valor do conjunto $\mathcal{V}$. Embora $\mathcal{V}$ possa apresentar escalas do tipo racional [322], assume-se a escala intervalar (com ordem total e métrica definida) como requisito mínimo para comparação de qualidade entre arquiteturas candidatas.

A formalização aqui apresentada não assume nenhuma premissa em relação à forma com que a expressão $qualityMetricExp(\Phi, M_c)$ é computada. A avaliação de uma métrica pode incluir desde simples expressões baseadas nas informações presentes na arquitetura candidata até o uso de modelos secundários baseados, por exemplo, em simulação ou Redes de Filas.

6.3. Algoritmos para Projeto Arquitetural Automatizado

As definições acima apresentadas viabilizam tanto a captura sistemática das diferentes alternativas arquiteturais para aplicações de um determinado domínio quanto a derivação de um mecanismo para exploração automática de espaços de projeto. Este mecanismo de exploração permite a construção automática de arquiteturas representadas por qualquer vetor candidato de um espaço de projeto específico de aplicação.

Esta seção apresenta os algoritmos propostos nesta tese para: *i)* derivação do espaço de projeto específico de aplicação a partir de um espaço de projeto particular e um modelo inicial; e *ii)* projeto automático da arquitetura (candidata ou inválida) correspondente a um determinado vetor candidato.

6.3.1. Algoritmo de Derivação do Espaço de Projeto Específico de Aplicação

Algoritmo 6.1: Algoritmo para derivação do espaço de projeto específico de aplicação.

1: **procedure** GENERATEAPPSPECIFICDS(M, ds): *asds*
2: ▷ M = Modelo Arquitetural inicial
3: ▷ ds = Espaço de Projeto
4: ▷ asds = Espaço de Projeto Específico de Aplicação resultante
5: $DDI \leftarrow \emptyset$
6: **for all** $dd \in ds.DD$ **do**
7: **for all** $te \in eval(dd.targetElementsExp, M)$ **do**
8: $DDI \leftarrow DDI \cup \{\langle M, dd, te \rangle\}$
9: **return** asds=$\langle M, ds, DDI \rangle$

O Algoritmo 6.1 apresenta os passos necessários para a derivação de um espaço de projeto específico de aplicação *asds* a partir de um modelo arquitetural inicial M e um espaço de projeto *ds*. Inicialmente, na linha 5, inicializa-se DDI como um conjunto vazio de instâncias de dimensão de projeto. Em seguida, para cada dimensão de projeto *dd* presente no espaço de projeto *ds* (linha 6), avalia-se a expressão *targetElementsExp* desta dimensão no modelo inicial M de modo a identificar os elementos-alvo da dimensão. Para cada elemento-alvo *te* encontrado para a dimensão *dd* (linha 7), cria-se uma instância de dimensão de projeto $\langle M, dd, te \rangle$ e armazena-se esta instância no conjunto DDI (linha 8). Ao final, retorna-se o espaço de projeto específico de aplicação definido por $\langle M, ds, DDI \rangle$.

Assume-se que o conjunto $ds.DD$ é ordenado e avaliado nesta ordem na linha 6. Adicionalmente, o conjunto DDI é mantido parcialmente ordenado, contendo: todas as instâncias da dimensão de projeto dd_1, todas as instâncias da dimensão de projeto dd_2 e assim sucessivamente. Entretanto, a ordem entre as instâncias de uma dimensão de projeto dd_i é arbitrária e depende da ordem dos elementos retornados pela expressão *targetElementsExp* de dd_i.

6.3.2. Algoritmo de Projeto Automatizado de Arquiteturas Candidatas

O Algoritmo 6.2 apresenta os passos necessários para a geração da arquitetura que corresponde à modificação de um modelo inicial M a partir dos pontos de variação definidos por um vetor candidato $\mathbf{x}$, pertencente a um espaço de projeto específico de aplicação *asds*.

O primeiro passo consiste na inicialização do modelo resultante M_r, fazendo-o igual ao modelo arquitetural inicial $asds.M$ (linha 4). Em seguida, na linha 5, um objeto *ctx* do tipo `EvaluationContext` é criado para armazenar os elementos criados pelas modificações, bem

Algoritmo 6.2: Algoritmo para geração de arquiteturas.

1: **procedure** GENERATEARCHITECTURE($asds, x$): M_r
2: ▷ $asds$ = Espaço de Projeto Específico de Aplicação
3: ▷ x = Vetor Candidato; M_r = arquitetura resultante
4: $M_r \leftarrow asds.M$
5: EvaluationContext ctx
6: $t \leftarrow 1$
7: **for all** $x_t \in x$ **do**
8: $vp_t \leftarrow asds.DDI[t].dd.VP[x_t]$
9: $ctx.setProperty(\text{"}target\text{"}, asds.DDI[t].te)$
10: **for all** $c \in vp_t.C$ **do**
11: **if** $!eval(preCondExp(c), M_r, ctx)$ **then**
12: **return** $M_r = invalidArchitecture$
13: $addedElem \leftarrow \emptyset$
14: $guideElem \leftarrow \emptyset$
15: **switch** $(type(c))$
16: **case** $elementsAddition$**:**
17: $guideElem \leftarrow eval(guideElementsExp(c), M_r, ctx)$
18: **for all** $element \in guideElem$ **do**
19: $addedElem \leftarrow addedElem \cup \{newInstance(addedType(c))\}$
20: $M_r \leftarrow M_r \cup \{addedElem\}$
21: $ctx.setProperty(\text{"}addedElements\text{-}c\text{"} + t, addedElem)$
22: $ctx.setProperty(\text{"}guideElements\text{-}c\text{"} + t, guideElem)$
23: **end case**
24: **case** $elementsRemoval$**:**
25: $M_r \leftarrow M_r \setminus eval(removedElementsExp(c), M_r, ctx)$
26: **end case**
27: **case** $propertyChange$**:**
28: $u \leftarrow 1$
29: **for all** $property \in eval(changedPropertyExp(c), M_r, ctx)$ **do**
30: $ctx.setProperty(\text{"}index\text{"}, u)$
31: $property \leftarrow eval(propertyValuesExp(c), M_r, ctx)$
32: $u \leftarrow u + 1$
33: **end case**
34: **end switch**
35: **if** $!eval(vp_t.postCondExp, M_r, ctx)$ **then**
36: **return** $M_r = invalidArchitecture$
37: $t \leftarrow t + 1$
38: **if** $!validateConstr(asds.ds.P, M_r)$ **or** $!validateConstr(MM, M_r)$ **then**
39: **return** $M_r = invalidArchitecture$
40: **return** M_r

como o elemento-alvo (representado pelo *id* "*target*") da instância de dimensão de projeto sendo avaliada em cada iteração. Este objeto é sempre passado como parâmetro durante a avaliação de expressões, de modo que todos os *ids* contidos em *ctx* estejam disponíveis para uso pelas expressões sendo avaliadas.

Logo após, para cada índice x_t presente no vetor candidato $\mathbf{x}$ (linha 7), armazena-se em vp_t o ponto de variação correspondente à decisão indicada por x_t. Esta decisão é o x_t-ésimo ponto de variação da dimensão de projeto da qual a t-ésima instância de dimensão de projeto em $asds.DDI$ foi criada (linha 8). Adicionalmente, ajusta-se o elemento *target* de *ctx* para o

elemento-alvo da instância de dimensão de projeto sendo considerada (linha 9).

As linhas 10 a 34 implementam a aplicação de cada modificação c presente em $vp_t.C$. Inicialmente, verifica-se se a pré-condição da modificação é atendida (linha 11). Caso esta pré-condição não seja satisfeita, uma arquitetura inválida é retornada. Caso contrário, aplica-se a modificação c, de acordo com o seu tipo.

Para modificações do tipo $elementsAddition$, obtém-se os elementos-guia da criação (linha 17), cria-se as novas instâncias do tipo definido em $addedType(c)$ (linhas 18 e 19), adiciona-se estes novos elementos na arquitetura resultante M_r (linha 20) e, finalmente, registra-se os novos elementos e os elementos-guia em ctx com o id adequado (linhas 21 e 22, respectivamente). Para modificações do tipo $elementsRemoval$, remove-se de M_r aqueles elementos resultantes da avaliação da expressão $removedElementsExp(c)$ (linha 25). Para modificações do tipo $propertyChange$, os seguintes passos são realizados: para cada i-ésima propriedade resultante da avaliação da expressão $changedPropertyExp(c)$ (linha 29), registra-se o índice da propriedade sendo atualmente alterada (linha 30), obtém-se o novo conjunto de valores da propriedade (através da avaliação da expressão $propertyValuesExp(c)$) e atribui-se este conjunto à propriedade em questão (linha 31).

Finalmente, após a aplicação de todas as modificações c do ponto de variação vp_t, verifica-se se a pós-condição do ponto de variação é satisfeita (linha 35). Caso não seja satisfeita, uma arquitetura inválida é retornada. As linhas 38 e 39 verificam se as restrições do *profile* e do metamodelo associados ao espaço de projeto são satisfeitas. Todo o processo é então novamente executado para o próximo índice x_t presente no vetor candidato $\mathbf{x}$. Após o processamento de todos os índices, com o atendimento de todas as pré- e pós-condições envolvidas, M_r contém a arquitetura candidata gerada pelo vetor candidato $\mathbf{x}$.

6.4. Premissas e Limitações

A abordagem para especificação de espaços de projeto e geração automática de arquiteturas candidatas proposta nesta tese assume que as premissas descritas a seguir são satisfeitas.

P1 – As decisões de projeto do domínio de aplicação em questão podem ser ortogonalmente separadas. Conforme apresentado anteriormente, o conjunto total das modificações aplicadas a um determinado elemento-alvo é aquele formado pela união de todas as contribuições fornecidas por cada dimensão de projeto que identifica o elemento-alvo. Dessa forma, assume-se que o domínio de aplicação para o qual está sendo criado um espaço de projeto é caracterizado por decisões arquiteturais referentes a aspectos com algum grau de ortogonalidade.

P2 – Um modelo inicial suficientemente anotado pode ser criado para aplicações do domínio. Conforme apresentado no Algoritmo 6.1, a criação das instâncias de dimensão de projeto requer a identificação dos elementos-alvo associados a cada dimensão de projeto. Frequentemente, as expressões $targetElementsExp$ de cada dimensão de projeto procuram por elementos do modelo inicial que apresentem anotações de interesse. Por exemplo, no domínio de sistemas *self-adaptive*, pode-se identificar todos os elementos do modelo inicial que estejam anotados com o estereótipo `TFProcessComponent`. A utilização de modelos iniciais com anotações insuficientes pode inviabilizar a geração das instâncias de dimensão de projeto, produzindo espaços de projeto nulos ou de dimensão reduzida.

Visto que as demandas de anotação de um espaço de projeto particular têm impacto direto no custo de produção dos modelos iniciais, recomenda-se que – durante a concepção de um novo espaço de projeto – defina-se um conjunto mínimo de anotações utilizadas na identificação de

elementos-alvo e aplicação das modificações. É necessário, entretanto, uma análise minuciosa do *trade-off* entre custo de produção dos modelos iniciais versus expressividade/completude do espaço de projeto envolvido.

Os constructos e mecanismos para projeto arquitetural automatizado aqui apresentados procuram disponibilizar uma infraestrutura passível de uso em uma série de domínios de aplicação. Entretanto, visto que tal infraestrutura foi primariamente criada para viabilizar o projeto arquitetural automatizado de sistemas *self-adaptive*, outros domínios podem requerer funcionalidades não atualmente suportadas. Tais limitações são descritas a seguir.

L1 – Instâncias de uma mesma dimensão de projeto são avaliadas em ordem arbitrária. Conforme apresentado no Algoritmo 6.2, a pré-condição da modificação c é avaliada em M_r e não em M. Adicionalmente, cada modificação c é aplicada também em M_r, viabilizando uma construção incremental do modelo arquitetural resultante. Tal situação traz algumas implicações para a abordagem aqui proposta.

Primeiro, visto que $asds.ds.DD$ é um conjunto ordenado de dimensões de projeto, a pré-condição de uma modificação c associada a uma instância de dimensão de projeto ddi pode considerar, durante sua avaliação, resultados de modificações realizadas por qualquer modificação c' associada a uma instância de dimensão de projeto ddi' tal que $ddi'.DD < ddi.DD$. Note que o conjunto $asds.DDI$ é parcialmente ordenado: garante-se, para cada elemento-alvo, a aplicação ordenada das modificações em relação às dimensões de projeto às quais estão associadas. Modificações em elementos-alvo diferentes são arbitrariamente aplicadas. Acredita-se, entretanto, ser rara a situação onde a avaliação de modificações em um determinado elemento-alvo dependa do resultado da avaliação de modificações em outro elemento-alvo. Suportar tal requisito traria uma maior complexidade à formalização aqui apresentada, sem benefícios consideráveis.

Segundo, durante a fase de concepção de um novo espaço de projeto, é necessário que o arquiteto especifique as dimensões de projeto na ordem correta. Por exemplo, no domínio dos sistemas *self-adaptive*, a dimensão de projeto relacionada à lei de controle deve preceder, em $ds.DD$, aquela relacionada à técnica de sintonia utilizada. Tal situação é necessária, pois uma determinada técnica de sintonia pode requerer a verificação – como expressão de pré-condição – da presença de um tipo particular de controlador. Caso a pré-condição não seja satisfeita a arquitetura é marcada como inválida. De forma semelhante, as modificações associadas a um ponto de variação devem também ser especificadas na ordem correta. Como regra geral, modificações do tipo $elementsAddition$ geralmente precedem aquelas do tipo $propertyChange$, visto que os *ids* dos elementos criados são frequentemente utilizados na expressão $propertyValuesExp$. Embora demandem cuidado por parte do arquiteto responsável pela concepção de um espaço de projeto, tal atividade ocorre somente uma vez por domínio de aplicação conforme explicado no Capítulo 5.

L2 – Instâncias de diferentes dimensões de projeto são avaliadas em ordem fixa. Embora garanta-se a avaliação das instâncias de dimensão de projeto na ordem com a qual as dimensões de projeto correspondentes foram definidas no conjunto $ds.DD$, tal ordem é fixa e não varia, por exemplo, de acordo com os pontos de variação selecionados por um determinado vetor candidato. Assume-se que, independente de quais pontos de variação pv_i e pv_j são escolhidos, respectivamente, para as dimensões de projeto dd_i e dd_j, a ordem de avaliação de tais dimensões é sempre fixa e igual àquela especificada durante a concepção do espaço de projeto.

Suportar tal requisito implicaria na especificação, por ponto de variação, daquelas dimensões de projeto a serem previamente avaliadas. O algoritmo para geração de arquiteturas teria de encontrar, para cada vetor candidato $\mathbf{x}$, uma ordem topológica de aplicação dos pontos de variação e detectar a presença de ciclos de dependência. Novamente, acredita-se que tais

demandas sejam raras e, portanto, decidiu-se pela não introdução desta complexidade na solução aqui proposta.

L3 – O número de pontos de variação em cada dimensão de projeto não varia em função do elemento-alvo. A abordagem aqui apresentada não suporta a variação, em tempo de execução, do número de pontos de variação apresentados para uma determinada dimensão de projeto. Em alguns domínios de aplicação, as possibilidades de solução (pontos de variação) para um determinado aspecto arquitetural (dimensão de projeto) podem variar em função do modelo inicial utilizado. Por exemplo, uma dimensão de projeto que captura o nó (máquina) no qual um determinado componente será implantado terá tantos pontos de variação quanto o número de nós apresentados no modelo inicial (possíveis locais de implantação). Em situações ainda mais dinâmicas, o número de pontos de variação pode variar em função de aspectos do elemento-alvo em questão, produzindo instâncias de dimensão de projeto com um número diferente de pontos de variação, mesmo tendo estas sido criadas a partir de uma mesma dimensão de projeto.

Mesmo sendo esta situação mais frequente que aquelas identificadas nas limitações **L1** e **L2**, decidiu-se por não suportar tal requisito, pelas razões descritas a seguir. Primeiro, novas expressões teriam de ser especificadas para cada dimensão de projeto, com o objetivo de derivar o número de pontos de variação em função do modelo inicial e elemento-alvo em questão. Segundo, as modificações de cada ponto de variação teriam de considerar não somente o elemento-alvo em questão mas também alguma outra informação que caracterizaria o ponto de variação associado (ex: o nome do nó de implantação disponível). Finalmente, uma maior complexidade seria introduzida nos mecanismos para otimização multiobjetivo de arquiteturas apresentados no Capítulo 8, visto que o número de *bits* utilizados para representar cada indivíduo da população iria variar em função do modelo de entrada.

Como tal requisito não foi demandado no domínio de aplicação foco deste trabalho (sistemas *self-adaptive*), decidiu-se por não suportá-lo. Espaços de projeto para outros domínios de aplicação – futuramente criados – podem requerer tal aspecto e, portanto, a infraestrutura aqui apresentada deve ser evoluída.

L4 – O número de instâncias de dimensão de projeto não varia em função dos pontos de variação selecionados. Alguns domínios de aplicação podem apresentar situações onde a existência de uma instância de dimensão de projeto está condicionada à escolha por um determinado ponto de variação de uma outra instância de dimensão de projeto. Por exemplo, escolher uma forma para comunicação remota entre dois componentes só faz sentido caso estes dois componentes sejam implantados em nós diferentes.

Suportar tal aspecto requer o uso de vetores candidatos com um número variável de dimensões. Adicionalmente, este número deve ser dinamicamente calculado, em função dos índices de pontos de variação atribuídos a cada elemento do vetor candidato $\mathbf{x}$. A escolha por um determinado valor de x_i impacta na quantidade e na semântica (instância de dimensão de projeto correspondente) de cada $x_{j>i}$, trazendo complexidades adicionais ao Algoritmo 6.2. Adicionalmente, o Algoritmo 6.1 – atualmente executado uma vez para cada modelo inicial considerado – precisaria ser avaliado uma vez para cada arquitetura sendo gerada.

Pelas razões apresentadas acima, decidiu-se não suportar tal requisito. Entretanto, pode-se contornar tais problemas através da definição de dimensões de projeto mais amplas e do uso de modificações com pré-condições que identificam as situações acima como arquiteturas inválidas. No exemplo citado, adotaria-se uma dimensão de projeto mais genérica ("forma de comunicação" em vez de "forma de comunicação remota"), incluindo tanto técnicas para comunicação local quanto aquelas para comunicação remota entre componentes. Para cada ponto de variação im-

plementando um tipo de comunicação remota, a expressão de pré-condição de uma determinada modificação verificaria se os componentes envolvidos estão implantados em nós diferentes.

L5 – Espaços de projeto são dependentes de metamodelo. Finalmente, os espaços de projeto criados com a abordagem aqui apresentada são dependentes do metamodelo utilizado na sua concepção, no sentido em que somente modelos iniciais descritos neste metamodelo poderão ser automaticamente manipulados. Dessa forma, um novo espaço de projeto precisa ser criado para cada combinação de domínio de aplicação e metamodelo.

Decidiu-se não suportar espaços de projeto independentes de metamodelo por três razões: *i)* um nível adicional de independência teria de ser introduzido em toda a formalização, aumentando sua complexidade; *ii)* considera-se rara a situação onde diferentes metamodelos são utilizados para modelar um mesmo aspecto de um sistema; e *iii)* linguagens com alta expressividade – como a UML – permitem a modelagem de uma ampla faixa de aspectos de um sistema com a utilização de um único metamodelo.

6.5. Resumo do Capítulo

Este capítulo apresentou a formalização da infraestrutura para especificação de espaços de projeto que suporta a geração automática de arquiteturas candidatas de um domínio de aplicação em particular. Adicionalmente, apresentou-se o algoritmo para derivação das instâncias de dimensão de projeto para uma combinação particular de espaço de projeto e modelo arquitetural inicial, bem como o algoritmo para geração automática de arquiteturas correspondentes a um determinado vetor candidato, especificado em um espaço de projeto específico de aplicação.

Um **metamodelo arquitetural** define os tipos de elementos (constructos) disponíveis para uso em modelos e as propriedades presentes em cada tipo. Cada propriedade representa referências a instâncias de outros tipos definidos no mesmo metamodelo. A multiplicidade de uma propriedade define se ela se refere a no máximo uma instância (propriedades *single*) ou a um número ilimitado de instâncias (propriedades *multiple*). Adicionalmente, uma propriedade pode definir uma relação de parentesco com as instâncias referidas (propriedades *composite*) – assumindo certas responsabilidades tais como gerência do ciclo de vida de tais instâncias – ou representar simples referências (propriedades *shared*).

Um **modelo arquitetural** é um conjunto de instâncias de tipos definidos no metamodelo associado. Adicionalmente, cada elemento mantém um conjunto particular de valores para todas as propriedades definidas, no metamodelo, para o tipo do qual foi criado. As relações de parentesco definidas pelas propriedades *composite* viabiliza a representação hierárquica de modelos, facilitando navegação entre elementos e a gerência do ciclo de vida destes elementos.

Uma **modificação arquitetural** é uma operação indivisível que realiza a transformação de um modelo M em um modelo $M' \neq M$. Modificações podem ser de três tipos: *i)* aquelas que realizam a inclusão de novos elementos no modelo (tipo $elementAddition$); *ii)* aquelas que realizam a remoção de elementos do modelo (tipo $elementsRemoval$); e *iii)* aquelas que realizam modificações em propriedades de elementos já existentes (tipo $propertyChange$). Adicionalmente, uma modificação c define uma expressão de pré-condição $preCondExp(c)$ que deve ser satisfeita, em um modelo M, para que c seja passível de aplicação (válida) em M.

Um **ponto de variação** mantém um conjunto não-vazio e ordenado de modificações arquiteturais. Um ponto de variação é o constructo, aqui definido, para representar todas as modificações arquiteturais necessárias à implementação de uma determinada solução para um determinado aspecto de projeto arquitetural do domínio de aplicação em questão. Adicionalmente, um ponto de variação pv define uma expressão de pós-condição $postCondExp(pv)$ que deve ser satisfeita,

após a aplicação de todas as modificações arquiteturais de vp em um modelo M, para que vp seja passível de aplicação (válido) em M.

Uma **dimensão de projeto** mantém um conjunto não-vazio e ordenado de pontos de variação. Uma dimensão de projeto representa um determinado aspecto de projeto arquitetural, do domínio em questão, que demanda a escolha por uma solução particular (seleção de algum ponto de variação desta dimensão de projeto). Visto que mais de um elemento do modelo arquitetural inicial pode demandar a avaliação de um mesmo aspecto, cada dimensão de projeto dd define uma expressão $targetElementsExp(dd)$ – responsável por identificar os elementos do modelo inicial que demandam a avaliação daquela dimensão de projeto (elementos-alvo de dd).

Um **espaço de projeto** mantém um conjunto não-vazio e ordenado de dimensões de projeto e um conjunto não-vazio e ordenado de métricas de qualidade. Um espaço de projeto é o elemento central para captura sistemática de conhecimento refinado de projeto arquitetural proposto nesta tese, viabilizando os mecanismos para geração automática de arquiteturas candidatas. Este mecanismo, associado às métricas de qualidade definidas para o espaço de projeto tornam possível a adoção das técnicas para otimização multiobjetivo descritas no Capítulo 8.

Uma **instância de dimensão de projeto** indica que um determinado elemento do modelo arquitetural inicial (denominado elemento-alvo) demanda uma tomada de decisão relacionada ao aspecto de projeto arquitetural capturado por uma determinada dimensão de projeto. Visto que múltiplos elementos-alvo podem demandar tomadas de decisão relacionadas a uma mesma dimensão de projeto, o conceito de instância de dimensão de projeto viabiliza a seleção independente dos pontos de variação da dimensão de projeto em questão, porém aplicadas a diferentes elementos-alvo.

O conjunto formado por todas as instâncias de dimensão de projeto relacionadas aos elementos-alvo identificados, em um modelo arquitetural inicial M, pelas expressões $targetElementsExp$ de cada dimensão de projeto de um espaço de projeto ds é denominado **espaço de projeto específico de aplicação** de ds em M. Um espaço de projeto específico de aplicação $asds$ viabiliza o projeto automatizado de arquiteturas referentes a todas as combinações possíveis de pontos de variação selecionados para cada uma das instâncias de dimensão de projeto presentes em $asds$.

O produto cartesiano dos possíveis índices de pontos de variação selecionados define um **espaço de decisão arquitetural**. Uma combinação particular de índices de pontos de variação selecionados é denominada **vetor candidato** e representa uma possível solução de projeto arquitetural no domínio em questão. Denomina-se **arquitetura candidata** aquelas soluções geradas por vetores candidatos que selecionam apenas pontos de variação válidos para aplicação no modelo arquitetural inicial M. Caso contrário, diz-se que a solução gerada pelo vetor candidato é uma **arquitetura inválida**. O conjunto formado por todas as arquiteturas candidatas do espaço de decisão arquitetural é denominado **espaço arquitetural viável**.

Uma **métrica de qualidade** é uma tupla $qm = \langle \Phi, g \rangle$. Φ é uma função $\Phi\colon \mathcal{F}_{asds} \to \mathcal{V}$, onde $\mathcal{F}_{asds}$ é um espaço arquitetural viável e $\mathcal{V}$ é um conjunto com escala no mínimo intervalar. Cada métrica de qualidade qm define uma expressão $qualityMetricExp(\Phi, M_c)$ que avalia o valor da função Φ na arquitetura candidata M_c. Nenhuma restrição é aqui assumida em relação ao modo de interpretação da expressão $qualityMetricExp$. A avaliação de uma métrica de qualidade pode incluir desde simples expressões baseadas nas informações presentes na arquitetura candidata até o uso de modelos secundários baseados, por exemplo, em simulação ou Redes de Filas. Adicionalmente, é informado em g se a métrica deve ser maximizada (valor 1) ou minimizada (valor -1).

A Seção 6.3 apresentou os algoritmos para derivação de um espaço de projeto específico de aplicação e para geração automática da arquitetura que corresponde a um determinado vetor candidato $\mathbf{x}$. Duas premissas e cinco limitações foram discutidas na Seção 6.4, bem como suas

justificativas, possíveis impactos e soluções alternativas. A Figura 6.4 apresenta um resumo dos conceitos vistos neste capítulo.

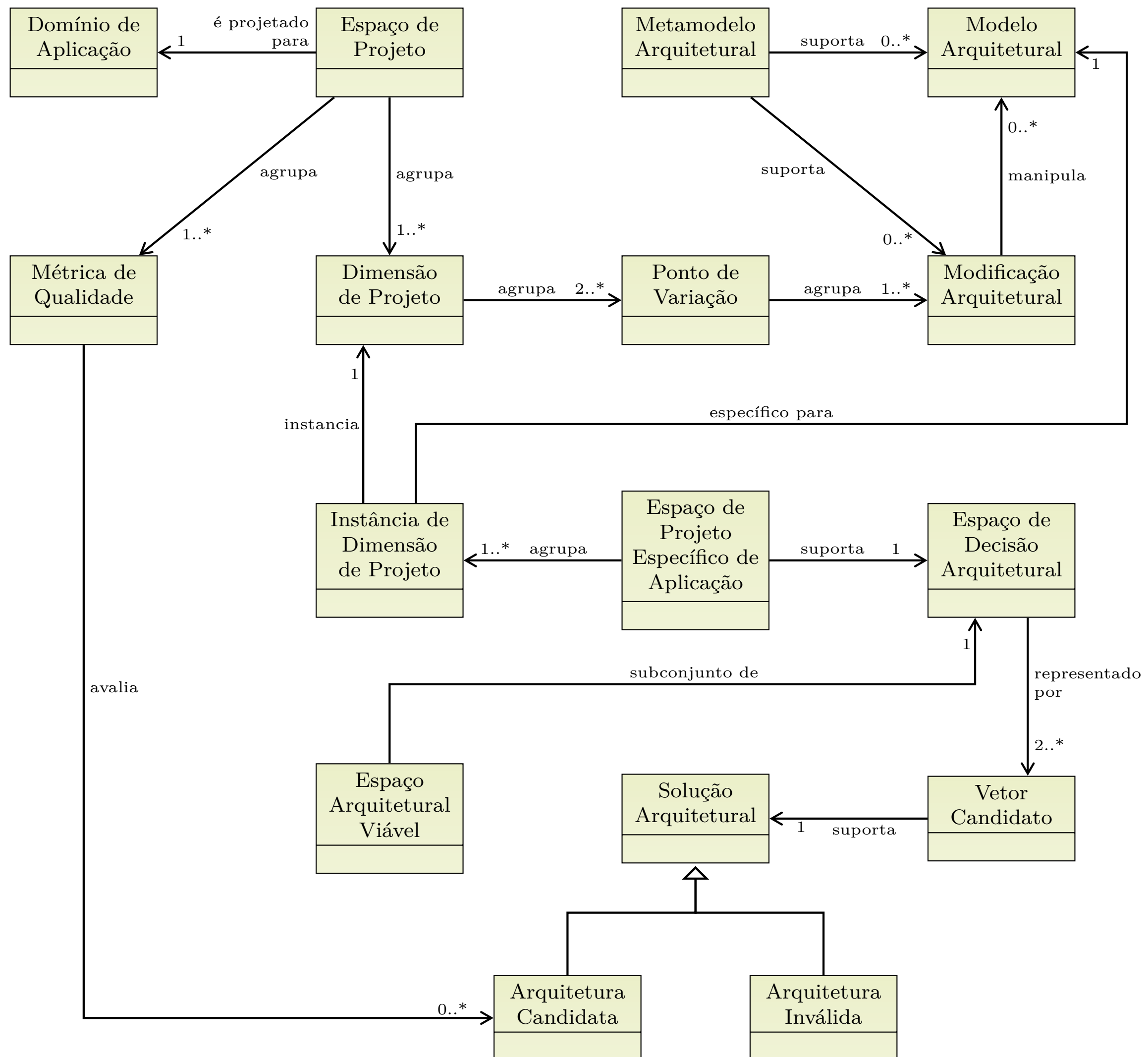


Figura 6.4.: Modelo conceitual das definições apresentadas no capítulo 6.

Capítulo 7

Espaço de Projeto Arquitetural para Sistemas Self-Adaptive

A wealth of information creates a poverty of attention.
Herbert Simon

Este capítulo apresenta o espaço de projeto para sistemas *self-adaptive* proposto nesta tese. Este espaço de projeto é especificado com base na formalização apresentada no Capítulo 6 e viabiliza o projeto automático de sistemas gerenciadores (*feedback control loops*) para sistemas gerenciados utilizados como entrada para o processo descrito na Seção 5.4, pág. 104. Conforme mencionado anteriormente, serão aqui apresentadas aquelas dimensões de projeto relacionadas à construção de arquiteturas para sistemas *self-adaptive* que utilizam *feedback control* como mecanismo viabilizador do autogerenciamento. Outros mecanismos – tais como agentes inteligentes, busca ou formalização de estrutura (vide Seção 3.3, pág. 40) – podem ser incluídos no futuro, de modo a promover uma captura mais completa do espaço de solução que caracteriza o domínio de aplicação.

Conforme descrito na Seção 5.4 (pág. 104) e apresentado formalmente na Definição 6.5 (pág. 117), cada dimensão de projeto deve identificar, no modelo arquitetural de entrada (inicial), aqueles elementos que demandam uma tomada de decisão referente ao aspecto arquitetural capturado pela dimensão de projeto. Tais elementos são os elementos-alvo da dimensão de projeto e uma instância de dimensão de projeto é criada, para cada elemento-alvo, de modo a viabilizar a escolha por um determinado ponto de variação da dimensão de projeto em questão. A identificação de tais elementos requer que o modelo arquitetural inicial seja anotado com informações particulares do domínio de aplicação do espaço de projeto sendo utilizado.

As dimensões de projeto mais prevalentes no domínio de aplicação dos sistemas *self-adaptive* são apresentadas na Seção 7.2. Para cada dimensão de projeto, são apresentados os seus pontos de variação e respectivas modificações arquiteturais. A descrição de cada dimensão de projeto segue o esquema apresentado na Seção 7.1. As diferentes arquiteturas candidatas resultantes da escolha de um ponto de variação particular em cada instância de dimensão de projeto apresentam diferentes atributos de qualidade. Tais atributos são quantificados através das métricas de qualidade discutidas na Seção 7.3. Finalmente, a Seção 7.4 apresenta o resumo deste capítulo. A Figura 7.1 apresenta o roteiro deste capítulo.

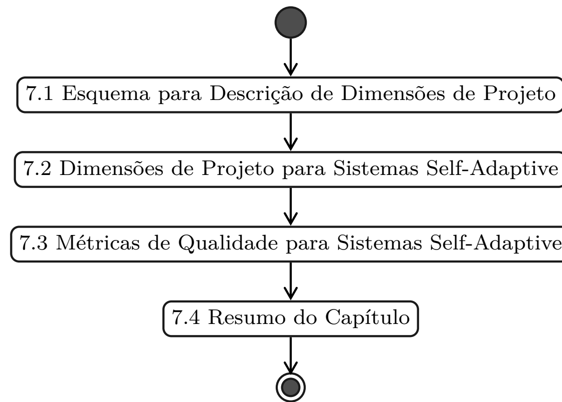


Figura 7.1.: Roteiro do capítulo 7.

7.1. Esquema para Descrição das Dimensões de Projeto

As dimensões de projeto para sistemas *self-adaptive* apresentadas na Seção 7.2 serão descritas de acordo com o esquema a seguir. Para cada dimensão de projeto serão informados:

- **Resumo:** uma breve descrição do aspecto arquitetural capturado pela dimensão de projeto.
- **Rationale:** as razões pelas quais a dimensão de projeto merece ser considerada no projeto automatizado de sistemas *self-adaptive*, bem como as eventuais premissas e limitações presentes.
- **Descrição:** uma explanação mais completa sobre o aspecto arquitetural capturado pela dimensão de projeto.
- **Impactos em Atributos de Qualidade:** indicação dos atributos de qualidade impactados pela escolha por um determinado ponto de variação da dimensão de projeto em questão. Deve indicar as razões pelas quais o impacto existe.
- **Elementos-Alvo:** apresentação da motivação para a escolha dos elementos-alvo e da expressão que os identifica no modelo arquitetural inicial. Esta expressão frequentemente busca por anotações, criadas com base no *profile* UML definido em conjunto com o espaço de projeto em questão. O *profile* UML construído para suportar o espaço de projeto para sistemas *self-adaptive* aqui proposto é descrito no Apêndice A.
- **Pontos de Variação:** apresentação dos pontos de variação da dimensão de projeto em questão, bem como suas respectivas modificações arquiteturais. São também apresentadas as expressões de pré-condição de cada modificação e a expressão de pós-condição de cada ponto de variação. Tais expressões são também definidas com base no *profile* UML associado.
- **Exemplo:** apresentação de um ou mais exemplos de aplicação dos pontos de variação definidos.

Vale ressaltar que a definição dos pontos de variação e modificações arquiteturais são dependentes de metamodelo. O espaço de projeto para sistemas *self-adaptive* proposto na Seção 7.2 assume que arquiteturas são descritas em UML. Adicionalmente, as expressões para identificação dos elementos-alvo e para pré- e pós-condições são descritas em OCL (*Object Constraint Language*) [226].

7.2. Dimensões de Projeto do SA:DuSE

O espaço de projeto para sistemas *self-adaptive* proposto nesta tese – SA:DuSE – realiza a captura sistemática de conhecimento de projeto levando em consideração cinco diferentes aspectos arquiteturais (dimensões de projeto) prevalentes no domínio em questão: cardinalidade de controle (DD2), lei de controle (DD3), técnica de sintonia (DD4), grau de adaptabilidade do controlador (DD5) e forma de cooperação entre múltiplos *loops* (DD6). Tais dimensões de projeto, seus pontos de variação e respectivas modificações arquiteturais são apresentadas a seguir.

Adicionalmente, uma dimensão de projeto relevante (DD1: modelagem da dinâmica do sistema gerenciado) – porém não automatizada – é discutida a seguir. As justificativas da decisão pela não-automação, seguidas das instruções sobre como disponibilizar as informações relevantes no modelo inicial, são também apresentadas a seguir. As modificações arquiteturais especificadas para cada ponto de variação de cada dimensão de projeto, bem como as métricas para avaliação de arquiteturas candidatas, fazem uso de um *profile* UML particularmente projetado para suportar o espaço de projeto SA:DuSE. A especificação completa deste *profile* UML pode ser encontrada no Apêndice A.

7.2.1. DD1) Modelagem da Dinâmica do Sistema Gerenciado

Resumo. Esta dimensão de projeto captura a estrutura (vide Seção 3.3.4.2, pág. 50) utilizada para representar o modelo da dinâmica do sistema gerenciado e a técnica (vide Seção 3.3.4.4, pág. 59) aplicada na construção deste modelo.

Rationale. A modelagem da dinâmica do sistema-alvo (sistema gerenciado) é uma das primeiras atividades do processo de projeto de controladores. A estrutura utilizada na representação deste modelo tem relação direta com as técnicas de projeto de controladores passíveis de serem utilizadas. A precisão da modelagem tem impacto direto na qualidade de controle observada. A análise de propriedades de controle tais como estabilidade, tempo de estabilização e sobressinal (aqui denominadas métricas de qualidade de domínio específico), depende diretamente da obtenção do modelo da dinâmica do sistema final (sistema-alvo + sistema gerenciador).

Descrição. Conforme apresentado na Seção 3.3.4.2, conhecer a dinâmica do sistema-alvo é fundamental para sistematizar o projeto de controladores, fazendo com que o sistema final apresente as propriedades desejadas. Um modelo representativo e preciso viabiliza a construção de controladores estáveis e que apresentem tempo de estabilização e sobressinal mínimos. Modelos lineares e de baixa ordem facilitam as atividades de análise e projeto. Entretanto, muitos sistemas dinâmicos (incluindo os computacionais) são caracterizados por comportamentos não-lineares e/ou variantes no tempo (vide Seção 3.3.4.1, pág. 45). Mesmo assim, linearizações e aproximações em regiões de operação estreitas fazem com que modelos lineares sejam suficientemente precisos no controle de sistemas não-lineares. O uso de múltiplos modelos – um para cada região de operação – é também frequentemente utilizado em técnicas tais como o escalonamento de ganho (vide Seção 3.3.4.3, pág. 54).

Impactos em Atributos de Qualidade. Visto que controladores são projetados com base no modelo construído para a dinâmica do sistema-alvo, propriedades de controle podem não ser completamente atendidas quando modelos pouco precisos são utilizados. Erros em regime estacionário (vide Seção 3.3.4.1, pág. 48) podem ser percebidos mesmo quando a análise informa que

o controle possui alta precisão. Os tempos de estabilização ou sobressinal observados podem ultrapassar aqueles previstos nas atividades de análise. Em sistemas computacionais, tal situação pode implicar em violações de SLAs, uso ineficiente de recursos ou maior *overhead* de controle devido a oscilações não desejadas. Em casos mais críticos, instabilidades podem também ser observadas.

Elementos-Alvo. A construção do modelo deve ser realizada para cada elemento controlável presente no modelo arquitetural inicial. A depender da técnica de modelagem utilizada (ponto de variação escolhido para esta dimensão de projeto), atividades matemáticas analíticas ou experimentação com o sistema gerenciado são necessários. Técnicas analíticas de modelagem são difíceis de serem automatizadas, enquanto experimentação com o sistema gerenciado (requerida, por exemplo, nas técnicas de identificação de sistema e ensaio ao degrau apresentadas na Seção 3.3.4.2, pág. 50) requer a implementação de conectores com o sistema-alvo real e implica em automatizações com maior tempo de execução. Por estes motivos, decidiu-se pela não automatização do aspecto arquitetural capturado nesta dimensão de projeto. Informações sobre a dinâmica do sistema gerenciado devem estar presentes no modelo inicial, sob a forma de anotações. Embora isto demande um maior trabalho na criação destes modelos arquiteturais iniciais, modelagens via ensaio ao degrau são relativamente fáceis de serem realizadas e, portanto, primariamente recomendadas neste trabalho. Trabalhos futuros podem incluir a automatização deste aspecto.

Pontos de Variação. As técnicas mais comuns para modelagem da dinâmica de sistemas são descritas abaixo. Visto que esta dimensão de projeto não será alvo de automação, as modificações arquiteturais correspondentes não são apresentadas.

VP11) Princípios Fundamentais (*first-principles*): utiliza as leis naturais que regem o processo que caracteriza o sistema a ser controlado. É bastante utilizado no controle de sistemas mecânicos e processos químicos, em função da disponibilidade de um arcabouço teórico bastante expressivo para descrição da dinâmica de tais sistemas. É pouco utilizado, entretanto, na modelagem da dinâmica de sistemas computacionais, em função da alta ocorrência de componentes estocásticos. Pode ser utilizado para modelar tanto sistemas SISO quanto MIMO.

VP12) Black-Box (identificação de sistema): requer que o sistema gerenciado já esteja construído e disponível para experimentação (condição nem sempre atendida). Consiste na obtenção de um modelo que correlaciona o efeito da(s) entrada(s) de controle na(s) saída(s) medida(s). Modelos ARX (*AutoRegressive eXogeneous*) e regressão por mínimos quadrados (vide Seção 3.3.4.2, pág. 50) são amplamente utilizados nesta técnica de modelagem. A estrutura do modelo (número de entradas e saídas passadas que impactam a saída atual) deve ser criteriosamente definida. Modelos de baixa ordem podem ser imprecisos, ao passo que modelos de mais alta ordem podem sofrer *overfitting* (baixa generalização para situações diferentes daquela onde os dados foram coletados). A Tabela 3.4 (pág. 51) apresentou alguns indicadores frequentemente utilizados na avaliação dos modelos obtidos via *system identification*. Pode ser utilizado para modelar tanto sistemas SISO quanto MIMO.

VP13) Ensaio ao Degrau (*bump test*): também requer a existência do sistema gerenciado e sua disponibilidade para experimentação. O impacto da experimentação, entretanto, é mínimo quando comparado àquele presente nas técnicas de *system identification*. Consiste na variação da entrada de controle sob a forma de um passo (sinal degrau) e na medição de certos atributos do comportamento observado na saída medida (vide Seção 3.3.4.2, pág. 53). Tais atributos

são diretamente aplicados em modelos paramétricos de aproximação tais como o *First-Order Plus Dead Time* (FOPDT), apresentado na pág. 53. Embora o FOPDT possa apresentar baixa precisão quando utilizado para aproximar dinâmicas de mais alta ordem, modelos paramétricos mais expressivos estão disponíveis na literatura e a simplicidade de realização do ensaio ao degrau o torna bastante atrativo para modeladores iniciantes. Visto que a dimensão de projeto aqui descrita não é automatizada na abordagem proposta neste trabalho, sugere-se o ensaio ao degrau como primeira opção de modelagem da dinâmica do sistema gerenciado, de modo a facilitar a criação dos modelos arquiteturais iniciais. Frequentemente utilizado para modelar sistemas SISO.

Exemplos. As Figuras 7.2(a), 7.2(b) e 7.2(c) apresentam, respectivamente, sistemas cujas dinâmicas foram modeladas via função de transferência (*Transfer Function*), FOPDT e espaço de estados (*State Space*). Os estereótipos `<<TFProcessComponent>>`, `<<FOPDTProcessComponent>>` e `<<SSProcessComponent>>` – parte integrante do *profile* UML apresentado no Apêndice 269 – definem os *tagged values* necessários ao armazenamento das informações referentes ao modelo da dinâmica do sistema. Tais anotações permitem a identificação dos elementos-alvo das dimensões de projeto apresentadas a seguir e a avaliação das métricas de qualidade apresentadas na Seção 7.3.

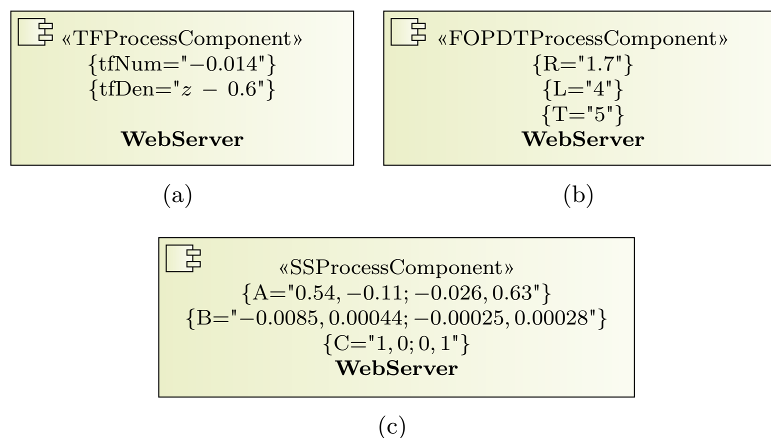


Figura 7.2.: Exemplo de modelagem de sistemas via função de transferência (a), FOPDT (b) e espaço de estados (c).

7.2.2. DD2) Cardinalidade de Controle

Resumo. Esta dimensão de projeto captura a quantidade e o tipo dos controladores atuando em cada componente controlável do modelo arquitetural inicial.

Rationale. Cada componente controlável do modelo arquitetural inicial pode apresentar qualquer número de interfaces controláveis e interfaces monitoráveis (vide seções 5.3.1 e 5.3.2, págs. 97 e 99 , respectivamente). Um controlador pode trabalhar com única entrada e única saída (SISO: *Single-Input Single-Output*) ou múltiplas entradas e múltiplas saídas (MIMO: *Multiple-Input Multiple-Output*) (vide Seção 3.3.4.1, pág. 45). Decisões referentes a este aspecto impactam tanto a qualidade do controle quanto a complexidade de projeto dos controladores.

Descrição. Em sistemas computacionais é comum a situação onde mais de uma entrada de controle impacta uma ou mais saídas medidas. Por exemplo, a utilização de CPU e consumo de

memória de um servidor *web* podem ser impactados tanto pelo número de *threads* que atendem requisições de clientes quanto pelo tempo de manutenção da conexão do servidor *web* após uma requisição do cliente. Em abordagens SISO, identifica-se aquela entrada de controle que mais impacta a saída medida mais importante e implementa-se um *feedback control loop* que atua em tais variáveis. Em situações onde mais de uma entrada de controle possui impacto considerável nas saídas medidas, o controle SISO pode apresentar desempenho insatisfatório. O controle MIMO, embora apresente maior complexidade de projeto, é uma alternativa a ser considerada. Mesmo quando múltiplas entradas de controle e múltiplas saídas medidas estão presentes, múltiplos controladores SISO podem ser aplicados caso altos impactos entre entrada de controle e saída medida possam ser emparelhados (impactos separáveis). É importante, neste caso, que cada emparelhamento tenha influência mínima em cada outro emparelhamento.

Impactos em Atributos de Qualidade. A decisão pelo uso de um único controlador SISO, um único controlador MIMO ou múltiplos controladores SISO impacta diretamente alguns atributos de qualidade importantes. Controladores SISO são relativamente fáceis de serem projetados pois envolvem somente uma entrada de controle e uma saída medida. Entretanto, podem apresentar baixo desempenho de controle quando outras variáveis estão presentes. Nestas situações, é comum a presença de oscilações ou altos tempos de estabilização – frequentemente diferentes daqueles valores previstos nas atividades de análise. Utilizar múltiplos controladores SISO (quando os impactos forem separáveis) ou um único controlador MIMO pode trazer melhorias neste cenário. Por outro lado, controle MIMO demanda o uso de técnicas mais avançadas de projeto como o LQR (vide Seção 3.3.4.4, pág. 61) e implica em um maior *overhead* de instrumentação, visto que múltiplas saídas devem ser medidas e múltiplas entradas de controle devem ser aplicadas ao sistema-alvo.

Elementos-Alvo. Esta dimensão de projeto deve ser avaliada para cada componente de processo presente no modelo arquitetural inicial. Tais elementos são identificados pela seguinte expressão OCL:

```
1: rootElement.allOwnedElements()->select(
2:    oclIsTypeOf(Component)
3: ).oclAsType(Component)->select(
4:    extension_ProcessComponent <> null
5: )
```

Para cada componente de processo encontrado deve-se decidir se o controle será realizado por múltiplos controladores SISO ou por um único controlador MIMO. Tal decisão é capturada pelos pontos de variação descritos a seguir.

Pontos de Variação. Esta dimensão de projeto viabiliza a utilização de múltiplos controladores SISO (ponto de variação VP21) ou um único controlador MIMO (ponto de variação VP22) no gerenciamento das portas do componente em questão.

VP21) SISO: indica que múltiplos controladores SISO serão utilizados para implementar os *feedback control loops* entre as portas controláveis e suas respectivas portas monitoráveis utilizadas como alvo (*supplier*) da dependência SISO associada. A Tabela 7.1 apresenta as modificações de inclusão de novos elementos e de alteração de propriedades deste ponto de variação. Nas descrições de pontos de variação a seguir são apresentados: o *id* da modificação (coluna #*c* – lembre-se que modificações são aplicadas nesta ordem), a metapropriedade da mudança e a especificação OCL para esta metapropriedade. Tais constructos para especificação de espaços de

projeto foram apresentados na Seção 6.2, pág. 114. Assume-se que, quando não explicitamente apresentadas, as expressões $GuideElementsExp(c)$ e $preCondExp(c)$ não estão definidas para a modificação arquitetural em questão.

	Modificações de Inclusão de Elementos - VP21	
#c	Metapropriedade	Especificação OCL
01	$addedType(c)$	SADuSE::PIDController
	$guideElementsExp(c)$	target.extension_ProcessComponent.controllablePort
02	$addedType(c)$	Uml::Port
	$guideElementsExp(c)$	target.extension_ProcessComponent.controllablePort. type
03	$addedType(c)$	Uml::Port
	$guideElementsExp(c)$	target.extension_ProcessComponent.controllablePort.type. clientDependency->**select**(extension_SISOImpact <> **null**). suppliers.**oclAsType**(Type)->**asOrderedSet**()
04	$addedType(c)$	Uml::Association
	$guideElementsExp(c)$	target.extension_ProcessComponent.controllablePort.type
05	$addedType(c)$	Uml::Association
	$guideElementsExp(c)$	target.extension_ProcessComponent.controllablePort.type. clientDependency->**select**(extension_SISOImpact <> **null**). suppliers.**oclAsType**(Type)->**asOrderedSet**()
	Modificações de Alteração de Propriedades - VP21	
#c	Metapropriedade	Especificação OCL
06	$changedPropertyExp(c)$	addedElements(c2).type
	$propertyValuesExp(c)$	guideElement(c2)[index]
07	$changedPropertyExp(c)$	addedElements(c2).isConjugated
	$propertyValuesExp(c)$	**true**
08	$changedPropertyExp(c)$	addedElements(c3).type
	$propertyValuesExp(c)$	guideElement(c3)[index]
09	$changedPropertyExp(c)$	addedElements(c3).isConjugated
	$propertyValuesExp(c)$	**true**
10	$changedPropertyExp(c)$	addedElements(c1).ownedPort
	$propertyValuesExp(c)$	addedElements(c2)∪ addedElements(c3)
11	$changedPropertyExp(c)$	addedElements(c4).memberEnd

	$propertyValuesExp(c)$	**OrderedSet**{
		addedElements(c2)[index],
		target.extension_ProcessComponent.controllablePort->
		select(type = guideElements(c2)[index])
		}

12	$changedPropertyExp(c)$	addedElements(c5).memberEnd

	$propertyValuesExp(c)$	**OrderedSet**{
		addedElements(c3)[index],
		target.extension_ProcessComponent.monitorablePort->
		select(type = guideElements(c3)[index])
		}

13	$changedPropertyExp(c)$	target.package.ownedElement
	$propertyValuesExp(c)$	target.package.ownedElement ∪ addedElements(c1) ∪ addedElements(c4) ∪ addedElements(c5)

Tabela 7.1.: Modificações do ponto de variação VP21.

As modificações de 1 a 5 realizam a criação de novos elementos do modelo. A modificação 1 cria uma nova instância de `PIDController` para cada porta controlável encontrada no componente

que representa o elemento-alvo. A modificação 2 cria uma nova instância de `Port` para cada tipo de porta controlável encontrada no elemento-alvo. Tais portas serão utilizadas pelos controladores recém-criados. Note que o resultado da avaliação da expressão *guideElementsExp(c2)* são os tipos das portas controláveis. Tais tipos ficam disponíveis para uso, em outras modificações, através do identificador *guideElements(c2)*.

A modificação 3 cria uma nova instância de `Port` para cada tipo que atua como alvo (*supplier*) de uma dependência do tipo `SISOImpact`. Tais portas também serão utilizadas pelos controladores recém-criados. Os tipos das portas monitoráveis ficam disponíveis em *guideElements(c3)*. Note que os conjuntos *guideElements(c2)* e *guideElements(c3)* são mantidos ordenados e, portanto, armazenam conjuntamente os tipos dos pares de tipos de portas controlável e monitorável, relacionadas por dependências do tipo `SISOImpact`.

Finalmente, as modificações 4 e 5 criam uma nova instância de `Association` para cada tipo de porta (controlável e monitorável, respectivamente) do elemento-alvo. Estas instâncias serão utilizadas para ligar as portas dos controladores recém-criados às portas correspondentes do componente atuando como elemento-alvo.

As modificações de 6 a 13 realizam a alteração de propriedades dos elementos do modelo. A modificação 6 altera a propriedade `type` de cada *i*-ésimo elemento criado na modificação 2 (futuras portas controláveis dos controladores) para o *i*-ésimo tipo de porta utilizado como elemento-guia da modificação 2. A modificação 7 faz com que todas as portas criadas na modificação 2 sejam conjugadas, ou seja, representem interfaces requeridas. As modificações 8 e 9 realizam os mesmos procedimentos para as futuras portas monitoráveis dos controladores.

A modificação 10 faz com que cada par de portas criadas nas modificações 2 e 3 passem a ser de propriedade de cada controlador criado na modificação 1 (note que a propriedade `ownerPort` é *composite* e, portanto, grafada com o sublinhado). Note que as primeiras portas monitorável e controlável pertencerão ao controlador 1, as segundas ao controlador 2, e assim sucessivamente.

A modificação 11 ajusta os elementos das associações criadas na modificação 4. Note como a primeira associação relaciona a porta controlável do primeiro controlador à primeira porta controlável do elemento-alvo, a segunda associação relaciona a porta controlável do segundo controlador à segunda porta controlável do elemento-alvo, e assim sucessivamente. A modificação 12 realiza os mesmo procedimentos para as portas monitoráveis. Finalmente, a modificação 13 indica que os controladores e associações recém-criados farão parte do mesmo pacote que contém o componente que atua como elemento-alvo.

VP22) MIMO: indica que um único controlador MIMO será utilizado para implementar um *feedback control loop* entre portas controláveis do mesmo componente de processo e portas monitoráveis utilizadas como alvos (*suppliers*) da dependência MIMO associada. A Tabela 7.2 apresenta as modificações de inclusão de novos elementos e alteração de propriedades deste ponto de variação.

As modificações de 1 a 5 realizam a criação de novos elementos do modelo. Tais modificações são semelhantes àquelas apresentadas no ponto de variação VP21, exceto que somente uma instância de `SSController` é criada na modificação 1 (expressão *guideElementsExp* não definida). Note que, neste ponto de variação, um único controlador MIMO será utilizado para gerenciar todas as portas controláveis e monitoráveis do componente que atua como elemento-alvo.

As modificações de 6 a 13 são também semelhantes às apresentadas no ponto de variação anterior, com exceção da modificação 10. Visto que um único controlador MIMO será utilizado, todas as portas controláveis e monitoráveis criadas nas modificações 2 e 3, respectivamente, farão parte do mesmo controlador criado na modificação 1. Tal situação é descrita em *propertyValuesExp(c10)*.

Modificações de Inclusão de Elementos - VP22		
#*c*	Metapropriedade	Especificação OCL
01	*addedType*(*c*)	SADuSE::SSController
02	*addedType*(*c*)	Uml::Port
	guideElementsExp(*c*)	target.extension_ProcessComponent.controllablePort.type
03	*addedType*(*c*)	Uml::Port
	guideElementsExp(*c*)	target.extension_ProcessComponent.controllablePort.type. clientDependency->**select**(extension_MIMOImpact <> **null**). suppliers.**oclAsType**(Type)->**asOrderedSet**()
04	*addedType*(*c*)	Uml::Association
	guideElementsExp(*c*)	target.extension_ProcessComponent.controllablePort.type
05	*addedType*(*c*)	Uml::Association
	guideElementsExp(*c*)	target.extension_ProcessComponent.controllablePort.type. clientDependency->**select**(extension_MIMOImpact <> **null**). suppliers.**oclAsType**(Type)->**asOrderedSet**()
Modificações de Alteração de Propriedades - VP22		
#*c*	Metapropriedade	Especificação OCL
06	*changedPropertyExp*(*c*)	addedElements(c2).type
	propertyValuesExp(*c*)	guideElement(c2)[index]
07	*changedPropertyExp*(*c*)	addedElements(c2).isConjugated
	propertyValuesExp(*c*)	**true**
08	*changedPropertyExp*(*c*)	addedElements(c3).type
	propertyValuesExp(*c*)	guideElement(c3)[index]
09	*changedPropertyExp*(*c*)	addedElements(c3).isConjugated
	propertyValuesExp(*c*)	**true**
10	*changedPropertyExp*(*c*)	addedElements(c1).ownedPort
	propertyValuesExp(*c*)	addedElements(c2)∪ addedElements(c3)
11	*changedPropertyExp*(*c*)	addedElements(c4).memberEnd

	propertyValuesExp(*c*)	**OrderedSet**{
		addedElements(c2)[index],
		target.extension_ProcessComponent.controllablePort->
		select(type = guideElements(c2)[index])
		}

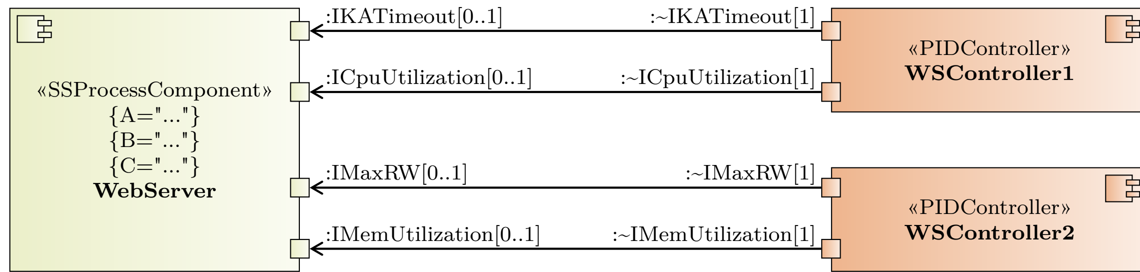
12	*changedPropertyExp*(*c*)	addedElements(c5).memberEnd

	propertyValuesExp(*c*)	**OrderedSet**{
		addedElements(c3)[index],
		target.extension_ProcessComponent.monitorablePort->
		select(type = guideElements(c3)[index])
		}

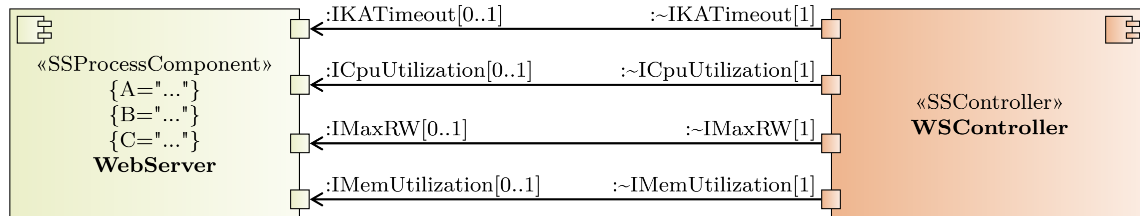
13	*changedPropertyExp*(*c*)	target.package.ownedElement
	propertyValuesExp(*c*)	target.package.ownedElement ∪ addedElements(c1) ∪ addedElements(c4) ∪ addedElements(c5)

Tabela 7.2.: Modificações do ponto de variação VP22.

Exemplos. As Figuras 7.3(a) e 7.3(b) ilustram a aplicação dos pontos de variação VP21 e VP22, respectivamente, a um modelo de um servidor *web* MIMO. Para este modelo arquitetural inicial, o conjunto de elementos-alvo retornado pela expressão OCL acima apresentada é formado apenas pelo componente `WebServer` (note que `SSProcessComponent` é um estereótipo filho de `ProcessComponent` - vide Apêndice A).



(a)



(b)

= sistema gerenciado (modelo de entrada)
= sistema gerenciador (automaticamente incluído pela abordagem aqui proposta)

Figura 7.3.: Exemplos de arquiteturas candidatas geradas para o ponto de variação VP21 (a) e VP22 (b).

7.2.3. DD3) Lei de Controle

Resumo. Esta dimensão de projeto captura a lei de controle a ser utilizada no sistema gerenciador. A lei de controle indica como a entrada de controle é calculada em função do erro obtido para a saída medida (diferença entre o valor de referência e a saída medida).

Rationale. A lei de controle indica como as atividades de análise e planejamento (vide Seção 3.2.2, pág. 33) do *loop* de adaptação serão realizadas. Tais atividades são fundamentais para a obtenção de um bom desempenho de controle e obtenção das metas de adaptação. Um amplo conjunto de leis de controle está atualmente disponível na literatura (vide Seção 3.3, pág. 40) e apresentam diferentes características em relação à consideração ou não de respostas transientes, graus de adaptação, suporte a incertezas sobre o sistema e o ambiente, bem como o uso ou não de bases de conhecimento.

Descrição. Esta dimensão de projeto captura as principais leis de controle originadas na área de Teoria de Controle e atualmente aplicadas no gerenciamento de sistemas computacionais adaptativos. São apresentadas cinco leis de controle para sistemas SISO e duas leis de controle para sistemas MIMO. Note, portanto, que a seleção de um ponto de variação representando uma lei de controle SISO nesta dimensão de projeto (DD3), associado a um ponto de variação que representa cardinalidade de controle MIMO na dimensão de projeto anterior (DD2) resulta em uma arquitetura inválida. Esta dimensão apresenta pontos de variação para a implementação das seguintes leis de controle SISO: Proporcional (P), Integral (I), Proporcional-Integral (PI), Proporcional-Derivativo (PD) e Proporcional-Integral-Derivativo (PID) – vide Seção 3.3.4.3, pág. 55. Dois pontos de variação para leis de controle MIMO são apresentados: *Static State Feedback Control* e *Dynamic State Feedback Control* – vide Seção 3.3.4.3, pág. 56.

Impactos em Atributos de Qualidade. Conforme apresentado no Capítulo 3, Seção 3.3.4.3 (pág. 54), diferentes leis de controle apresentam diferentes valores para as propriedades SASO (estabilidade, precisão, tempo de estabilização e sobressinal) e caracterizam *trade-offs* no atendimento de tais atributos de qualidade. Tornar estes *trade-offs* explícitos e disponibilizar ao arquiteto diferentes possibilidades de atendimento das propriedades é fator importante para viabilizar um projeto mais sistemático de sistemas *self-adaptive* e garantir a consideração de diversas alternativas arquiteturais.

Elementos-Alvo. Esta dimensão de projeto deve ser avaliada para cada porta controlável de cada componente de processo presente no modelo arquitetural inicial. Tais elementos são identificados pela seguinte expressão OCL:

```
1: rootElement.allOwnedElements()->select(
2:    oclIsTypeOf(Port)
3: ).oclAsType(Port)->select(
4:    type.oclIsTypeOf(Interface) and
5:    type.oclAsType(Interface).
6:       extension_ControllableInterface <> null
7: )
```

Deve-se decidir, para cada porta controlável, a lei de controle a ser utilizada no controlador associado à porta em questão. Tal decisão é capturada pelos pontos de variação descritos a seguir.

Pontos de Variação. Conforme apresentado anteriormente, esta dimensão de projeto captura a implementação de cinco leis de controle relacionadas a sistemas SISO e duas leis de controle relacionadas a sistemas MIMO, descritas a seguir.

VP31) Proporcional: indica que o controlador associado ao elemento-alvo em questão (porta controlável) utilizará controle Proporcional (vide Seção 3.3.4.3, pág. 55) na gerência do componente de processo que contém a porta. A Tabela 7.3 apresenta as modificações associadas a este ponto de variação.

Modificações de Alteração de Propriedades - VP31		
#c	Metapropriedade	Especificação OCL
1	*changedPropertyExp*(*c*)	target.opposite.owner.**oclAsType**(Component).extension_Controller. **oclAsType**(SADuSE::PIDController).kp
	propertyValuesExp(*c*)	-1
	preCondExp(*c*)	target.opposite.owner.**oclAsType**(Component).extension_Controller. **oclIsTypeOf**(SADuSE::PIDController)
2	*changedPropertyExp*(*c*)	target.opposite.owner.**oclAsType**(Component).extension_Controller. **oclAsType**(SADuSE::PIDController).ki
	propertyValuesExp(*c*)	0
3	*changedPropertyExp*(*c*)	target.opposite.owner.**oclAsType**(Component).extension_Controller. **oclAsType**(SADuSE::PIDController).kd
	propertyValuesExp(*c*)	0

Tabela 7.3.: Modificações do ponto de variação VP31.

As três modificações deste ponto de variação indicam quais parâmetros do controlador PID (K_P, K_I e K_D) deverão ser ajustados pela técnica de sintonia escolhida na dimensão de projeto DD4 (apresentada a seguir). Atribui-se o valor 0 aos parâmetros que não deverão ser ajustados pela técnica de sintonia e o valor 1 para aqueles a serem ajustados.

Note que o elemento-alvo da dimensão de projeto (representado por *target*) é uma porta controlável de um componente de processo. Como consequência da escolha por um ponto de variação da dimensão de projeto DD2, cada porta controlável participa de uma associação binária onde o outro participante é a porta correspondente do controlador que gerencia o componente de processo que contém a porta em questão. O atributo *opposite* do metamodelo da UML dá acesso à propriedade oposta (porta do controlador) de uma associação. O atributo *owner* permite a obtenção do componente (controlador) que contém a porta que representa a propriedade oposta. Finalmente, *extension_Controller* dá acesso ao estereótipo que armazena os parâmetros K_P, K_I e K_D do controlador.

Note que – em função do controle Proporcional ser uma lei de controle SISO – vetores candidatos contendo este ponto de variação só produzirão arquiteturas válidas se a porta controlável que atua como elemento-alvo estiver associada a um controlador também SISO (do tipo `PIDController`). Tal condição é capturada por $preCondExp(c1)$. Vale ressaltar que a dimensão de projeto DD2 é avaliada antes da dimensão de projeto DD3 (vide algoritmo 6.2, pág. 123) e, portanto, esta informação já está disponível no modelo arquitetural intermediário. Qualquer tentativa de uso de uma lei de controle SISO em uma porta controlável associada a um controlador MIMO produzirá uma arquitetural inválida.

VP32) Integral: indica que o controlador associado ao elemento-alvo em questão (porta controlável) utilizará controle Integral (vide Seção 3.3.4.3, pág. 55) na gerência do componente de processo que contém a porta. As modificações deste ponto de variação são análogas àquelas apresentadas no ponto de variação VP31, atribuindo – desta vez – o valor 0 a $propertyValuesExp(c1)$ e $propertyValuesExp(c3)$, enquanto $propertyValuesExp(c2)$ utiliza o valor -1. Dessa forma, os parâmetros K_P e K_D são ajustados para zero, indicando que somente o fator integral será utilizado pela lei de controle (e, consequentemente, ajustado pela técnica de sintonia selecionada na dimensão de projeto DD4). A mesma pré-condição para detecção de uso de um controlador SISO – $preCondExp(c1)$ – é utilizada neste ponto de variação.

VP33) Proporcional-Integral: indica que o controlador associado ao elemento-alvo em questão (porta controlável) utilizará controle Proporcional-Integral (vide Seção 3.3.4.3, pág. 55) na gerência do componente de processo que contém a porta. As modificações deste ponto de variação são análogas àquelas apresentadas no ponto de variação VP31, atribuindo – desta vez – o valor 0 apenas a $propertyValuesExp(c3)$, enquanto $propertyValuesExp(c1)$ e $propertyValuesExp(c2)$ utilizam o valor -1. Dessa forma, apenas o parâmetro K_D é ajustado para zero, indicando que os fatores proporcional e integral serão utilizados pela lei de controle. A mesma pré-condição para detecção de uso de um controlador SISO – $preCondExp(c1)$ – é utilizada neste ponto de variação.

VP34) Proporcional-Derivativo: indica que o controlador associado ao elemento-alvo em questão (porta controlável) utilizará controle Proporcional-Derivativo (vide Seção 3.3.4.3, pág. 55) na gerência do componente de processo que contém a porta. As modificações deste ponto de variação são análogas àquelas apresentadas no ponto de variação VP31, atribuindo – desta vez – o valor 0 apenas a $propertyValuesExp(c2)$, enquanto $propertyValuesExp(c1)$ e $propertyValuesExp(c3)$ utilizam o valor -1. Dessa forma, apenas o parâmetro K_I é ajustado para zero, indicando que os fatores proporcional e derivativo serão utilizados pela lei de controle. A mesma pré-condição

para detecção de uso de um controlador SISO – $preCondExp(c1)$ – é utilizada neste ponto de variação.

VP35) Proporcional-Integral-Derivativo: indica que o controlador associado ao elemento-alvo em questão (porta controlável) utilizará controle Proporcional-Integral-Derivativo (vide Seção 3.3.4.3, pág. 55) na gerência do componente de processo que contém a porta. As modificações deste ponto de variação são análogas àquelas apresentadas no ponto de variação VP31, atribuindo – desta vez – o valor -1 a $propertyValuesExp(c1)$, $propertyValuesExp(c2)$ e $propertyValuesExp(c3)$. Dessa forma, nenhum parâmetro é ajustado para zero, indicando que os fatores proporcional, integral e derivativo serão utilizados pela lei de controle. A mesma pré-condição para detecção de uso de um controlador SISO – $preCondExp(c1)$ – é utilizada neste ponto de variação.

VP36) Static State Feedback: indica que o controlador associado ao elemento-alvo em questão (porta controlável) utilizará controle *Static State Feedback* (vide Seção 3.3.4.3, pág. 56) na gerência do componente de processo que contém a porta. A Tabela 7.4 apresenta as modificações associadas a este ponto de variação.

Modificações de Alteração de Propriedades - VP36		
#c	Metapropriedade	Especificação OCL
1	$changedPropertyExp(c)$	target.opposite.owner.**oclAsType**(Component).extension_Controller. **oclAsType**(SADuSE::SSController).kp
	$propertyValuesExp(c)$	**Sequence**{}
	$preCondExp(c)$	target.opposite.owner.**oclAsType**(Component).extension_Controller. **oclIsTypeOf**(SADuSE::SSController)
2	$changedPropertyExp(c)$	target.opposite.owner.**oclAsType**(Component).extension_Controller. **oclAsType**(SADuSE::SSController).ki
	$propertyValuesExp(c)$	**Sequence**{0}

Tabela 7.4.: Modificações do ponto de variação VP36.

Visto que o *Static State Feedback Control* é lei de controle MIMO, a expressão de pré-condição definida em $preCondExp(c1)$ verifica se a porta controlável que atua como elemento-alvo está associada a um controlador MIMO (do tipo `SSController`). Qualquer tentativa de uso deste ponto de variação em uma porta controlável associada a um controlador SISO produzirá uma arquitetural inválida. Note que os parâmetros K_P e K_I do controlador `SSController` associado são matrizes (representadas por `Sequences` na UML). O parâmetro K_I é ajustado para a sequência formada por um único elemento (0) para indicar que a técnica de sintonia selecionada na dimensão de projeto DD4 não deverá ajustar este valor e, portanto, a lei de controle *Static State Feedback* será adotada (vide Seção 3.3.4.3, pág. 56). A sequência que representa K_P é mantida vazia.

VP37) Dynamic State Feedback: indica que o controlador associado ao elemento-alvo em questão (porta controlável) utilizará controle *Dynamic State Feedback* (vide Seção 3.3.4.3, pág. 56) na gerência do componente de processo que contém a porta. As modificações deste ponto de variação são análogas àquelas apresentadas no ponto de variação VP36, atribuindo – desta vez – o sequência vazia (`Sequence {}`) a $propertyValuesExp(c1)$ e $propertyValuesExp(c2)$. Dessa forma, nenhum parâmetro do controlador é ajustado para a sequência contendo o zero, indicando que todos os fatores (proporcional e integral) serão ajustados pela técnica de sintonia

selecionada na dimensão de projeto DD4 e, portanto, a lei de controle *Dynamic State Feedback Control* será adotada (vide Seção 3.3.4.3, pág. 56). A mesma pré-condição para detecção de uso de um controlador MIMO – $preCondExp(c1)$ do ponto de variação VP36 – é utilizada neste ponto de variação.

Exemplos. Ver exemplificação conjunta com a dimensão de projeto DD4.

7.2.4. DD4) Técnica de Sintonia

Resumo. Esta dimensão de projeto captura a técnica de sintonia a ser utilizada em cada *feedback control loop* presente no sistema. A técnica de sintonia indica como os parâmetros do controlador serão ajustados de modo a produzir sistemas estáveis e que apresentem bons valores para as demais propriedades (precisão, tempo de estabilização e sobressinal).

Rationale. Após a modelagem do sistema-alvo e escolha da lei de controle a ser utilizada, o ajuste dos parâmetros do controle passa a ser fundamental para obtenção de boas propriedades de controle. Tais parâmetros indicam, por exemplo, quão "agressivo" o controlador será na presença de perturbações, se o sistema final formado pelo sistema-alvo + controlador será estável, bem como quanto tempo será necessário para que o sistema atinja o valor de referência desejado. Um conjunto de técnicas de sintonia foi apresentado na Seção 3.3.4.4, pág. 59. Esta dimensão de projeto captura aquelas técnicas passíveis de serem realizadas de forma automática, tais como os métodos empíricos (Seção 3.3.4.4, pág. 61) e o LQR (Seção 3.3.4.4, pág. 61).

Descrição. A adição de um controlador a um sistema-alvo produz um sistema final com uma nova dinâmica, preferencialmente apresentando propriedades intencionalmente planejadas pelo arquiteto. Embora a escolha da lei de controle influencie fortemente a habilidade do sistema final em rejeitar perturbações e rastrear valores de referência, controladores mal sintonizados podem produzir instabilidades e desempenho sub-ótimo das adaptações.

Esta dimensão de projeto captura seis técnicas empíricas para sintonia de controladores SISO e uma técnica para sintonia de controladores MIMO. As técnicas SISO aqui consideradas são: CHR (Chien-Hrones-Reswick – nas suas quatro variações – vide Seção 3.3.4.4, Tabela 3.7, pág. 62), Ziegler-Nichols (vide Seção 3.3.4.4, Tabela 3.6, pág. 62) e Cohen-Coon (vide Seção 3.3.4.4, Tabela 3.8, pág. 62). A técnica MIMO aqui descrita é a LQR (*Linear-Quadratic Regulator*), apresentada na Seção 3.3.4.4, pág. 61.

As técnicas empíricas têm como objetivo a obtenção dos parâmetros K_P, K_I e K_D a partir dos valores de R, L e T presentes em um modelo obtido a partir de um experimento de ensaio ao degrau (vide Seção 3.3.4.2, pág. 53). O LQR, por outro lado, tem como objetivo a obtenção dos vetores K_P e K_I que produzem desempenho ótimo (bom *trade-off* entre tempo de estabilização e sobressinal) em controladores que trabalham com espaços de estado (controle MIMO).

Impactos em Atributos de Qualidade. Os valores atribuídos aos parâmetros do controlador impactam diretamente a precisão, tempo de estabilização e sobressinal observados no sistema final. Embora algumas técnicas empíricas, tais como o CHR, apresentem variantes que explicitamente favorecem uma ou outra propriedade de controle, uma avaliação criteriosa destes impactos requer operações de análise geralmente custosas (vide Tabela 3.5, pág. 52) e difíceis de serem realizadas por arquitetos sem formação na área de Teoria de Controle. Favorecer certa propriedade de controle (ex: tempo de estabilização) em detrimento de outra (ex: sobressinal) deve ser uma decisão intencionalmente tomada pelo arquiteto. Acredita-se que a sistematização

das alternativas arquiteturais e a apresentação fundamentada dos *trade-offs* existentes – tarefas realizadas neste trabalho – são fundamentais para uma avaliação criteriosa dos impactos da sintonia no desempenho de controle.

Elementos-Alvo. Esta dimensão de projeto deve ser avaliada para cada porta controlável de cada componente de processo presente no modelo arquitetural inicial. Tais elementos são identificados pela seguinte expressão OCL:

```
1: rootElement.allOwnedElements()->select(
2:    oclIsTypeOf(Port)
3: ).oclAsType(Port)->select(
4:    type.oclIsTypeOf(Interface) and
5:    type.oclAsType(Interface).
6:       extension_ControllableInterface <> null
7: )
```

Pontos de Variação. Conforme apresentado anteriormente, esta dimensão de projeto captura a implementação de seis técnicas de sintonia de controladores SISO e uma técnica de sintonia de controladores MIMO, descritas a seguir.

VP41) CHR-0OS-DR: indica que o controlador associado ao elemento-alvo em questão (porta controlável) será sintonizado utilizando a técnica empírica de Chien-Hrones-Reswick (CHR), com 0% de sobressinal e tendo como meta a rejeição de perturbações (vide Tabela 3.7, pág. 62). A Tabela 7.5 apresenta as modificações associadas a este ponto de variação.

As três modificações que constituem este ponto de variação têm como objetivo o ajuste dos parâmetros K_P, K_I e K_D do controlador (respeitada a lei de controle selecionada na dimensão de projeto anterior – DD3). De forma semelhante ao apresentado na dimensão de projeto anterior, a primeira modificação deste ponto de variação define uma expressão – $preCondExp(c1)$ – que exige a presença de um componente de processo do tipo `FOPDTComponentProcess` e um controlador SISO (do tipo `PIDController`). Qualquer tentativa de uso desta técnica de sintonia em portas associadas a controladores MIMO irá produzir uma arquitetura inválida.

As expressões apresentadas em $propertyValuesExp$ definem, de acordo com a técnica de sintonia em questão e lei de controle escolhida, os valores a serem atribuídos a K_P, K_I e K_D. Note que estas expressões são avaliadas no contexto do elemento arquitetural resultante da avaliação da expressão $changedPropertyExp$ – cada propriedade que representa cada parâmetro de sintonia do controlador.

A modificação 1, inicialmente, armazena em `ext` o estereótipo que contém os parâmetros de controle a serem ajustados e em `pc` o estereótipo que contém os valores que representam a dinâmica do sistema-alvo. Em seguida, caso a lei de controle escolhida na dimensão de projeto DD3 considere o fator Proporcional (condição $ext.kp \neq 0$), atribui-se a K_P o valor correspondente, conforme descrito a seguir.

Caso somente controle Proporcional esteja sendo utilizado (detectado pela expressão "$ext.ki = 0$ and $ext.kd = 0$"), atribui-se a K_P o valor $^{0.3*pc.T}/_{pc.R*pc.L}$. Os valores de ajuste para a técnica de sintonia CHR com 0% de sobressinal e tendo como meta a rejeição de perturbações estão disponíveis na Tabela 3.7, pág. 62 (lembre-se que $a = RL/T$). Caso controle Proporcional-Integral ou Proporcional-Derivativo esteja sendo utilizado (detectado pela expressão "$ext.ki = 0$ xor $ext.kd = 0$"), atribui-se a K_P o valor $^{0.6*pc.T}/_{pc.R*pc.L}$. Finalmente, caso a lei de controle Proporcional-Integral-Derivativo esteja sendo utilizada, o valor atribuído a K_P é

		Modificações de Alteração de Propriedades - VP41
#c	Metapropriedade	Especificação OCL
1	*changedProperty Exp(c)*	target.opposite.owner.**oclAsType**(Component).extension_Controller. **oclAsType**(SADuSE::PIDController).kp

	propertyValues Exp(c)	**let** ext = owner.extension_Controller.**oclAsType**(SADuSE::PIDController) **in**
		let pc = target.**oclAsType**(SADuSE::FOPDTProcessComponent) **in**
		if (ext.kp <> 0) **then**
		if (ext.ki = 0 **and** ext.kd = 0) **then** (0.3*pc.T)/(pc.R*pc.L)
		else
		if (ext.ki = 0 **xor** ext.kd = 0) **then** (0.6*pc.T)/(pc.R*pc.L)
		else (0.95*pc.T)/(pc.R*pc.L) **endif**
		endif
		else 0 **endif**

	preCondExp(c)	target.**oclIsTypeOf**(SADuSE::FOPDTProcessComponent) **and** target.opposite.owner.**oclAsType**(Component).extension_Controller. **oclIsTypeOf**(SADuSE::PIDController)
2	*changedProperty Exp(c)*	target.opposite.owner.**oclAsType**(Component).extension_Controller. **oclAsType**(SADuSE::PIDController).ki
	propertyValues Exp(c)	**let** ext = owner.extension_Controller.**oclAsType**(SADuSE::PIDController) **in** **let** pc = target.**oclAsType**(SADuSE::FOPDTProcessComponent) **in** **if** (ext.ki <> 0) **then** **if** (ext.kd = 0) **then** (ext.kp)/(4.0*pc.L) **else** (ext.kp)/(2.4*pc.L) **endif** **else** 0 **endif**
3	*changedProperty Exp(c)*	target.opposite.owner.**oclAsType**(Component).extension_Controller. **oclAsType**(SADuSE::PIDController).kd
	propertyValues Exp(c)	**let** ext = owner.extension_Controller.**oclAsType**(SADuSE::PIDController) **in** **let** pc = target.**oclAsType**(SADuSE::FOPDTProcessComponent) **in** **if** (ext.kd <> 0) **then** (ext_kp)*(0.42*pc.L) **else** 0 **endif**

Tabela 7.5.: Modificações do ponto de variação VP41.

${}^{0.95*pc.T}/_{pc.R*pc.L}$. As modificações 2 e 3 realizam operações semelhantes, de acordo com a lei de controle em questão e os valores disponibilizados na Tabela 3.7. Os valores de parâmetros desta técnica de sintonia são descritos, nas modificações aqui apresentadas, com notação sublinhado duplo.

VP42) CHR-0OS-RT: indica que o controlador associado ao elemento-alvo em questão (porta controlável) será sintonizado utilizando a técnica empírica de Chien-Hrones-Reswick (CHR), com 0% de sobressinal e tendo como meta o rastreamento de valores de referência (vide Tabela 3.7, pág. 62). As modificações deste ponto de variação atuam de forma semelhante àquelas apresentadas no ponto de variação VP41, utilizando – desta vez – os valores da Tabela 3.7 referentes a 0% de sobressinal e tendo como meta o rastreamento de valores de referência. A mesma pré-condição utilizada no ponto de variação VP41 é aqui aplicada.

VP43) CHR-20OS-DR: indica que o controlador associado ao elemento-alvo em questão (porta controlável) será sintonizado utilizando a técnica empírica de Chien-Hrones-Reswick (CHR), com 20% de sobressinal e tendo como meta a rejeição de perturbações (vide Tabela 3.7, pág. 62). As modificações deste ponto de variação atuam de forma semelhante àquelas apresentadas no ponto de variação VP41, utilizando – desta vez – os valores da Tabela 3.7 referentes a 20% de sobressinal e tendo como meta a rejeição de perturbações. A mesma pré-condição utilizada no ponto de variação VP41 é aqui aplicada.

VP44) CHR-20OS-RT: indica que o controlador associado ao elemento-alvo em questão (porta controlável) será sintonizado utilizando a técnica empírica de Chien-Hrones-Reswick (CHR), com 20% de sobressinal e tendo como meta o rastreamento de valores de referência (vide Tabela 3.7, pág. 62). As modificações deste ponto de variação atuam de forma semelhante àquelas apresentadas no ponto de variação VP41, utilizando – desta vez – os valores da Tabela 3.7 referentes a 20% de sobressinal e tendo como meta o rastreamento de valores de referência. A mesma pré-condição utilizada no ponto de variação VP41 é aqui aplicada.

VP45) Ziegler-Nichols: indica que o controlador associado ao elemento-alvo em questão (porta controlável) será sintonizado utilizando a técnica empírica de Ziegler-Nichols (vide Tabela 3.6, pág. 62). As modificações deste ponto de variação atuam de forma semelhante àquelas apresentadas no ponto de variação VP41, utilizando – desta vez – os valores da Tabela 3.6. A mesma pré-condição utilizada no ponto de variação VP41 é aqui aplicada.

VP46) Cohen-Coon: indica que o controlador associado ao elemento-alvo em questão (porta controlável) será sintonizado utilizando a técnica empírica de Cohen-Coon (vide Tabela 3.8, pág. 62). A Tabela 7.6 apresenta as modificações associadas a este ponto de variação.

As modificações deste ponto de variação atuam de forma semelhante àquelas apresentadas no ponto de variação VP41, utilizando – desta vez – os valores da Tabela 3.8 e novas formas de detecção da lei de controle escolhida na dimensão de projeto DD3. A mesma pré-condição utilizada no ponto de variação VP41 é aqui aplicada.

VP47) LQR: indica que o controlador associado ao elemento-alvo em questão (porta controlável) será sintonizado utilizando a técnica LQR (vide Tabela 3.7, pág. 62). A Tabela 7.7 apresenta as modificações associadas a este ponto de variação.

Exemplos. As Figuras 7.4(a) e 7.4(b) apresentam dois exemplos de combinações de pontos de variação das dimensões DD3 e DD4. A Figura 7.4(a) ilustra a arquitetura resultante da seleção dos pontos de variação VP33 (lei de controle Proporcional-Integral) e VP46 (sintonia de Cohen-Coon). Note que o parâmetro K_D do controlador é mantido em 0. Os demais valores (K_P e K_I) são ajustados de acordo com o apresentado na Tabela 62 e descritos em OCL na Tabela 148. A Figura 7.4(b) apresenta a arquitetura produzida pela combinação dos pontos de variação VP35 (controle Proporcional-Integral-Derivativo) e VP42 (sintonia de Chien-Hrones-Reswick com 0% de sobressinal e rastreamento de valores de referência).

7.2.5. DD5) Grau de Adaptabilidade do Controlador

Resumo. Esta dimensão de projeto captura o mecanismo utilizado para adaptação de cada controlador adicionado ao sistema-alvo inicial. O mecanismo de adaptação informa quão robusto o controle será em situações (dinâmicas) diferentes daquela prevista durante a escolha da técnica de sintonia adotada.

Rationale. Conforme apresentado na Seção 3.3.4.3, pág. 57, diversas técnicas de controle adaptativo podem ser utilizadas naquelas situações onde o alto dinamismo e imprevisibilidade do sistema, do ambiente e das perturbações inviabilizam a obtenção de bom desempenho de controle através da sintonia *off-line* do controlador. Neste casos, é necessário a realização contínua de operações de ressintonia do controlador, executadas em *run-time*. Tais esquemas de controle adaptativo melhoram a robustez do controlador e apresentam melhor desempenho em ambientes dinâmicos e incertos.

Modificações de Alteração de Propriedades - VP46		
#*c*	Metapropriedade	Especificação OCL
1	*changedProperty Exp*(*c*)	target.opposite.owner.**oclAsType**(Component).extension_Controller. **oclAsType**(SADuSE::PIDController).kp

	propertyValues Exp(*c*)	**let** ext = owner.extension_Controller.**oclAsType**(SADuSE::PIDController) **in**
		let pc = target.**oclAsType**(SADuSE::FOPDTProcessComponent) **in**
		let tau=pc.L/(pc.L+1) **in**
		if (ext.kp <> 0) **then**
		if (ext.ki = 0 **and** ext.kd = 0) **then** (pc.T)/(pc.R*pc.L)*(1+(0.35*tau)/(1-tau))
		else
		if (ext.kd = 0) **then** (0.9*pc.T)/(pc.R*pc.L)*(1+(0.92*tau)/(1-tau))
		else
		if (ext.ki = 0) **then** (1.24*pc.T)/(pc.R*pc.L)*(1+(0.13*tau)/(1-tau))
		else (1.35*pc.T)/(pc.R*pc.L)*(1+(0.18*tau)/(1-tau)) **endif**
		endif
		endif
		else 0 **endif**

	preCondExp(*c*)	target.**oclIsTypeOf**(SADuSE::FOPDTProcessComponent) **and** target.opposite.owner.**oclAsType**(Component).extension_Controller. **oclIsTypeOf**(SADuSE::PIDController)
2	*changedProperty Exp*(*c*)	target.opposite.owner.**oclAsType**(Component).extension_Controller. **oclAsType**(SADuSE::PIDController).ki
	propertyValues Exp(*c*)	**let** ext = owner.extension_Controller.**oclAsType**(SADuSE::PIDController) **in** **let** pc = target.**oclAsType**(SADuSE::FOPDTProcessComponent) **in** **let** tau=pc.L/(pc.L+1) **in** **if** (ext.ki <> 0) **then** **if** (ext.kd = 0) **then** (ext.kp)/(((3.3-3*tau)/(1+1.2*tau))*pc.L) **else** (ext.kp)/(((2.5-2*tau)/(1-0.39*tau))*pc.L) **else** 0 **endif**
3	*changedProperty Exp*(*c*)	target.opposite.owner.**oclAsType**(Component).extension_Controller. **oclAsType**(SADuSE::PIDController).kd
	propertyValues Exp(*c*)	**let** ext = owner.extension_Controller. **oclAsType**(SADuSE::PIDController) **in** **let** pc = target.**oclAsType**(SADuSE::FOPDTProcessComponent) **in** **let** tau=pc.L/(pc.L+1) **in** **if** (ext.kd <> 0) **then** **if** (ext.ki = 0) **then** (ext.kp)/(((0.27-0.36*tau)/(1-0.87*tau))*pc.L) **else** (ext.kp)/(((0.37-0.37*tau)/(1-0.81*tau))*pc.L) **endif** **else** 0 **endif**

Tabela 7.6.: Modificações do ponto de variação VP46.

Descrição. Quatro mecanismos para controle adaptativo foram apresentados na Seção 3.3.4.3: escalonamento de ganho, *Model Identification Adaptive Control* (MIAC), *Model Reference Adaptive Control* (MRAC) e controle reconfigurável. Cada um destes mecanismos apresentam adaptações mais ou menos drásticas (e, consequentemente, com maior ou menor *overhead*) na forma com que o controle é realizado.

O escalonamento de ganho identifica diferentes regiões de operação de controle (intervalos da saída medida) e define valores dos parâmetros do controlador para cada região de operação. Ao detectar que o sistema passou de uma região para outra, o controlador é automaticamente ressintonizado com os valores da região em questão. O escalonamento de ganho é eficaz para manter bom desempenho de controle em ambientes com alto dinamismo, mas com regiões de operação conhecidas.

Em situações mais imprevisíveis, entretanto, as regiões de operação não são previamente conhecidas. Nestes casos, a sintonia deve ser totalmente realizada em tempo de execução. O MIAC e MRAC são duas técnicas comumente utilizadas para este fim. O MIAC implementa versões *on-line* de algoritmos de identificação de sistema (ex: mínimos quadrados recursivo [26]), de modo a obter sempre valores atualizados para os parâmetros do controlador. Já o MRAC utiliza

	Modificações de Alteração de Propriedades - VP47	
$\#c$	Metapropriedade	Especificação OCL
1	*changedProperty Exp*(c)	target.opposite.owner.**oclAsType**(Component).extension_Controller. **oclAsType**(SADuSE::SSController).kp

	propertyValues Exp(c)	**let** ext = owner.extension_Controller.**oclAsType**(SADuSE::SSController) **in**
		let pc = target.**oclAsType**(SADuSE::SSProcessComponent) **in**
		if (ext.kp->**isEmpty**()) **then**
		lqr(pc.A, pc.B, pc.C)->iterate(
		row: Sequence(Real)
		kp: Sequence(Sequence(Real)) = Sequence{} |
		kp->including(row.subSequence(1, pc.mo))
		)
		else
		Sequence{}
		endif

	preCondExp(c)	target.**oclIsTypeOf**(SADuSE::SSProcessComponent) **and** target.opposite.owner.**oclAsType**(Component).extension_Controller. **oclIsTypeOf**(SADuSE::SSController)
2	*changedProperty Exp*(c)	target.opposite.owner.**oclAsType**(Component).extension_Controller. **oclAsType**(SADuSE::SSController).ki

	propertyValues Exp(c)	**let** ext = owner.extension_Controller.**oclAsType**(SADuSE::SSController) **in**
		let pc = target.**oclAsType**(SADuSE::SSProcessComponent) **in**
		if (ext.ki->**isEmpty**()) **then**
		lqr(pc.A, pc.B, pc.C)->iterate(
		row: Sequence(Real)
		ki: Sequence(Sequence(Real)) = Sequence{} |
		ki->including(row.subSequence(pc.mo+1, 2*pc.mo))
		)
		else
		Sequence{}
		endif

Tabela 7.7.: Modificações do ponto de variação VP47.

um modelo de referência previamente definido e encontra os valores de parâmetros que fazem com que o sistema final se comporte como o modelo de referência utilizado. O MRAC é mais utilizado em sistemas críticos e apresenta *overhead* maior que o MIAC.

Em ambientes ainda mais imprevisíveis e dinâmicos, a ressintonia contínua dos parâmetros do controlador pode não ser suficiente para a obtenção do desempenho de controle desejado. Nestes casos, adaptações mais drásticas devem ser realizadas, tais como a troca do controlador por outro que utilize uma lei de controle totalmente diferente. Note que as adaptações deixam de ser paramétricas (modificação de valores de atributos de componentes) e passam a ser estruturais. Consequentemente, o *overhead* de adaptação do controle é consideravelmente maior e as alterações estruturais devem ser realizadas em momentos adequados.

Impactos em Atributos de Qualidade. Conforme descrito anteriormente, esta dimensão de projeto tem impacto direto na avaliação do *trade-off* entre robustez de controle e *overhead* de adaptação. Em ambiente mais conhecidos, com pouca incerteza e perturbações previsíveis, controle não-adaptativo (com ganho fixo) pode ser suficiente para a obtenção das propriedades desejadas. Em outras situações, como aquelas apresentadas em sistemas robóticos autônomos ou *data centers* atendendo demandas com alta variabilidade, o uso de mecanismos para controle adaptativo é indispensável para o atendimento das metas de controle. O ponto central na tomada de boas decisões nesta dimensão é a identificação do patamar mínimo de robustez necessário e do patamar máximo de *overhead* aceitável. A partir daí, deve-se verificar se existe uma solução que atenda a estes dois critérios. A utilização de técnicas para identificação de arquiteturas

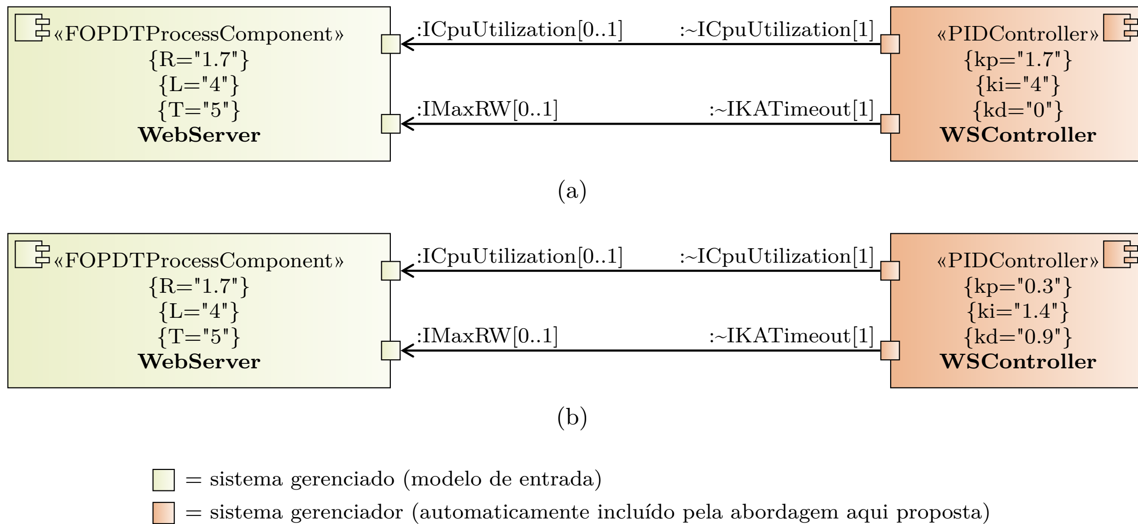


Figura 7.4.: (a) Arquitetura produzida pela combinação dos pontos de variação VP33 (controle Proporcional-Integral) e VP46 (sintonia de Cohen-Coon). (b) Arquitetura produzida pela combinação dos pontos de variação VP35 (controle Proporcional-Integral-Derivativo) e VP42 (sintonia de Chien-Hrones-Reswick com 0% de sobressinal e rastreamento de valores de referência).

candidatas que representam o *Pareto-front* deste *trade-off* é fundamental para a seleção de boas soluções para este problema.

Elementos-Alvo. Esta dimensão de projeto deve ser avaliada para cada porta controlável de cada componente de processo presente no modelo arquitetural inicial. Tais elementos são identificados pela seguinte expressão OCL:

```
1: rootElement.allOwnedElements()->select(
2:    oclIsTypeOf(Port)
3: ).oclAsType(Port)->select(
4:    type.oclIsTypeOf(Interface) and
5:    type.oclAsType(Interface).
6:       extension_ControllableInterface <> null
7: )
```

Pontos de Variação. Esta dimensão de projeto captura a implementação das quatro técnicas de adaptação de controladores apresentadas anteriormente. Adicionalmente, um ponto de variação representando controle não-adaptativo é também definido.

VP51) Ganho Fixo: indica que nenhum mecanismo para adaptação de controle será aplicado ao controlador que atua na porta em questão (elemento-alvo). Visto que, para a formação de uma arquitetura candidata, toda instância de dimensão de projeto deve ter um ponto de variação selecionado, este ponto de variação é utilizado para indicar que nenhuma ressintonia será realizada, em *run-time*, no controlador. Esta solução é adequada para o controle em ambientes com dinamismo controlado ou quando deseja-se abrir mão de alta robustez em função de um *overhead* mínimo de controle. Nenhuma modificação arquitetural é realizada, por este ponto de variação, no modelo arquitetural sendo construído.

VP52) Escalonamento de Ganho: indica que o controlador associado ao elemento-alvo em questão (porta controlável) será adaptado em *run-time* via escalonamento de ganho (vide Seção 3.3.4.3, pág. 54). A Tabela 7.8 apresenta as modificações associadas a este ponto de variação.

	Modificações de Inclusão de Elementos - VP52	
#*c*	Metapropriedade	Especificação OCL
01	*addedType*(*c*)	SADuSE::ITuneInterface
02	*addedType*(*c*)	Uml::Port
03	*addedType*(*c*)	SADuSE::GainScheduler
04	*addedType*(*c*)	Uml::Port
	guideElementsExp(*c*)	**Sequence** { addedElements(c1), target.extension_ProcessComponent.controllablePort.type. clientDependency->**select**(extension_SISOImpact <> **null**). suppliers.**oclAsType**(Type)->**first**() }
05	*addedType*(*c*)	Uml::Association
	guideElementsExp(*c*)	**Sequence** {addedElements(c2)->first(), target.opposite}
	Modificações de Alteração de Propriedades - VP52	
#*c*	Metapropriedade	Especificação OCL
06	*changedPropertyExp*(*c*)	addedElements(c2).type
	propertyValuesExp(*c*)	addedElements(c1)[index]
07	*changedPropertyExp*(*c*)	target.opposite.owner.**oclAsType**(Component).ownedPort
	propertyValuesExp(*c*)	target.opposite.owner.**oclAsType**(Component).ownedPort ∪ addedElements(c2)
08	*changedPropertyExp*(*c*)	addedElements(c4).type
	propertyValuesExp(*c*)	guideElements(c4)[index]
09	*changedPropertyExp*(*c*)	addedElements(c4).isConjugated
	propertyValuesExp(*c*)	**true**
10	*changedPropertyExp*(*c*)	addedElements(c3).ownedPort
	propertyValuesExp(*c*)	addedElements(c4)
11	*changedPropertyExp*(*c*)	addedElements(c5).memberEnd
	propertyValuesExp(*c*)	**OrderedSet**{addedElements(c4)[index], guideElements(c5)[index]}
12	*changedPropertyExp*(*c*)	target.package.ownedElement
	propertyValuesExp(*c*)	target.package.ownedElement ∪ addedElements(c1) ∪ addedElements(c3) ∪ addedElements(c5)

Tabela 7.8.: Modificações do ponto de variação VP52.

As modificações 1 a 5 especificam a criação de novos elementos. A modificação 1 cria uma nova instância de `SADuSE::ITuneInterface`, responsável pela modificação, em *run-time*, dos parâmetros K_P, K_I e K_D de um controlador PID. A modificação 2 cria a porta que disponibilizará, no controlador em questão, os serviços de ressintonia em *run-time* definidos em `SADuSE::ITuneInterface`. A modificação 3 cria uma nova instância de `SADuSE::GainScheduler`, componente responsável pelo monitoramento do sistema-alvo, identificação da região de operação em uso no momento e ajuste dos parâmetros do controlador para os valores correspondentes à região em questão.

A modificação 4 cria duas novas portas, ambas definidas nas modificações a seguir como de propriedade do elemento `SADuSE::GainScheduler`. A primeira porta será responsável pela interação com o controlador, com o objetivo de ajustar os valores de parâmetros. A segunda porta implementará a comunicação com o sistema-alvo, de modo a identificar a região de operação

em que o sistema atualmente se encontra. Note que os dois elementos-guia desta modificação correspondem exatamente aos tipos das portas com as quais as novas portas sendo criadas irão se comunicar. A modificação 5 cria duas novas associações UML, uma para porta criada na modificação 4. Note que os dois elementos-guia desta modificação correspondem às portas que atuarão como alvo das associações criadas.

As modificações 6 a 12 representam alterações de propriedades dos elementos. A modificação 6 indica que a porta criada na modificação 2 possui como tipo a interface criada na modificação 1. A modificação 7 inclui a porta criada na modificação 2 no conjunto de portas mantidas pelo controlador em questão (acessível via `target.opposite.owner`). A modificação 8 ajusta os tipos das portas criadas na modificação 4 para, respectivamente, `SADuSE::ITuneInterface` e o tipo da porta monitorável do componente de processo em questão (lembre-se que ambos os tipos são os elementos-guia da modificação 4). A modificação 9 indica que as duas portas criadas na modificação 4 são conjugadas, ou seja, definem interfaces requeridas. A modificação 10 indica que as duas portas criadas na modificação 4 são de propriedade do componente criado na modificação 3 (`SADuSE::GainScheduler`).

A modificação 11 realiza a ligação das portas do componente `SADuSE::GainScheduler` às portas correspondentes do controlador e do sistema-alvo em questão (as origens das associações são as portas criadas na modificação 4 e os alvos são os elementos-guia da modificação 5). Finalmente, a modificação 12 inclui – no mesmo pacote que contém a porta atuando como elemento-alvo – as duas associações recém-criadas, a interface `SADuSE::ITuneInterface` e o componente `SADuSE::GainScheduler`.

VP53) MIAC: indica que o controlador associado ao elemento-alvo em questão (porta controlável) será adaptado em *run-time* utilizando *Model Identification Adaptive Control* (vide Seção 3.3.4.3, pág. 57). As modificações deste ponto de variação são semelhantes àquelas apresentadas no ponto de variação VP52, exceto que uma instância de `SADuSE::MIAC` é criada no lugar de `SADuSE::GainScheduler` e uma nova associação – e suas portas correspondentes – são criadas para monitorar a entrada de controle (em adição à saída medida). Para fins de simplificação deste texto, tais modificações não serão apresentadas.

VP54) MRAC: indica que o controlador associado ao elemento-alvo em questão (porta controlável) será adaptado em *run-time* utilizando *Model Reference Adaptive Control* (vide Seção 3.3.4.3, pág. 57). As modificações deste ponto de variação são semelhantes àquelas apresentadas no ponto de variação VP52, exceto que uma instância de `SADuSE::MRAC` é criada no lugar de `SADuSE::GainScheduler` e uma nova associação – e suas portas correpondentes – são criadas para monitorar a entrada de controle (em adição à saída medida).

VP55) Controle Reconfigurável: indica que o controlador associado ao elemento-alvo em questão (porta controlável) será adaptado em *run-time* utilizando controle reconfigurável (vide Seção 3.3.4.3, pág. 57). As modificações deste ponto de variação são semelhantes àquelas apresentadas no ponto de variação VP52, exceto que uma instância de `SADuSE::ReconfigurationManager` é criada em vez de `SADuSE::GainScheduler`. Este trabalho não aborda os aspectos referentes a como uma nova configuração de controle será escolhida em *run-time*. Para os objetivos da proposta aqui apresentada, é suficiente saber apenas o tipo de adaptação à qual o controlador em questão está sujeito.

Exemplos. As Figuras 7.5(a) e 7.5(b) apresentam dois exemplos de uso dos pontos de variação da dimensão DD5. A Figura 7.5(a) apresenta um modelo arquitetural inicial estendido com controle PID e adaptação de controle via escalonamento de ganho. A Figura 7.5(b) ilustra a

mesma situação, só que utilizando adaptação de controle via MIAC (note o uso tanto da porta controlada quanto da porta monitorada do componente de processo em questão). Conforme apresentado anteriormente, conhecer o mecanismo de adaptação de controle utilizado (nenhum, escalonamento de ganho, MIAC, MRAC ou controle reconfigurável) é suficiente para as metas de análise arquitetural propostas nesta tese. O uso destes modelos para outros fins – por exemplo suporte à implementação – pode requerer uma modelagem mais detalhada da estrutura interna dos mecanismos de adaptação.

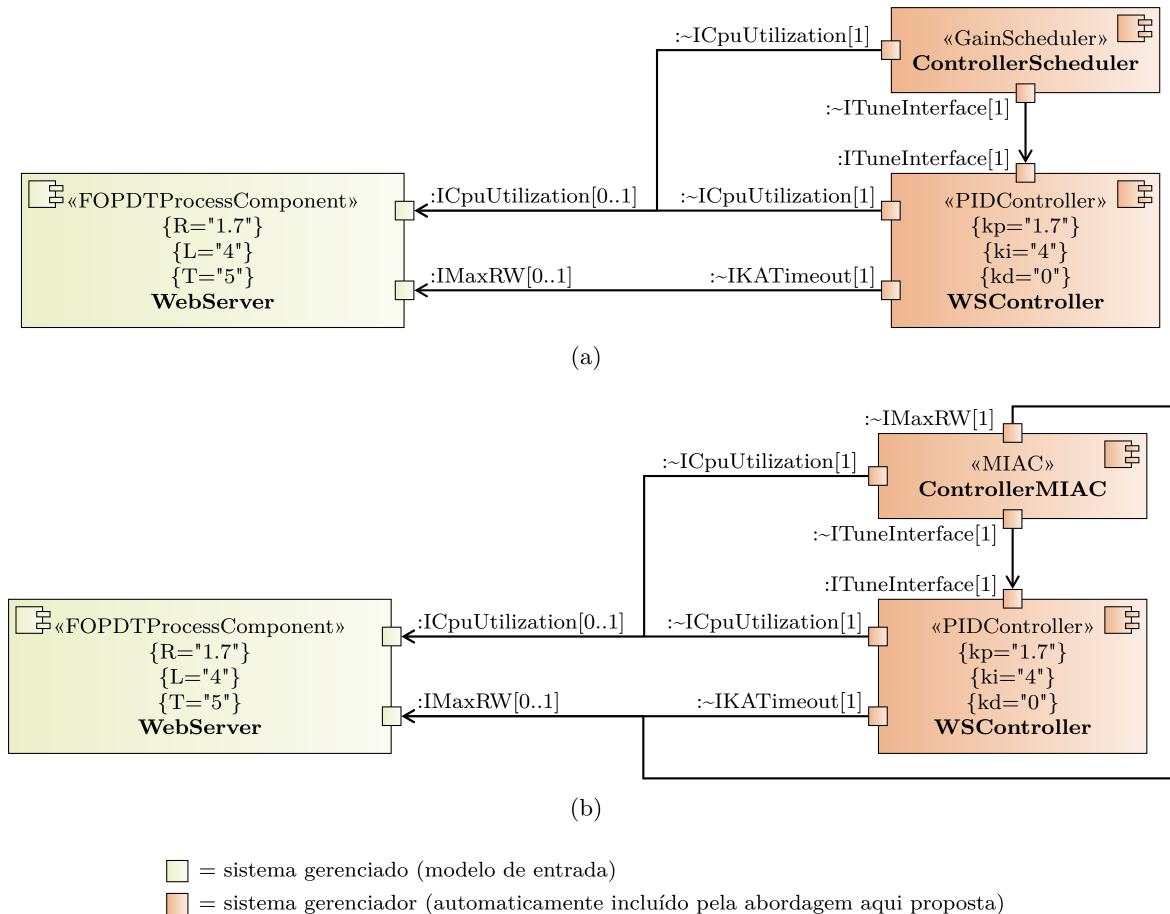


Figura 7.5.: (a) Arquitetura produzida pela combinação dos pontos de variação VP33, VP46 e VP52 (adaptação de controle via escalonamento de ganho). (b) Arquitetura produzida pela combinação dos pontos de variação VP35, VP42 e VP53 (adaptação via MIAC).

7.2.6. DD6) Forma de Cooperação entre Múltiplos Loops

Resumo. Esta dimensão de projeto captura a forma de cooperação entre os múltiplos *loops* eventualmente adicionados ao sistema-alvo inicial. A forma de cooperação define diferentes graus de descentralização do controle e, consequentemente, diferentes níveis de *overhead* de comunicação.

Rationale. Em sistemas distribuídos modernos – tais como as plataformas para *cloud computing* – a utilização de um único *loop* de adaptação pode ser inviável, devido à própria natureza distribuída do sistema gerenciado, às demandas rigorosas por alta escalabilidade e à presença de requisitos complexos de autogerenciamento. Nestes casos, o uso de múltiplos *loops* viabiliza a separação dos diferentes aspectos de controle, a distribuição dos custos das adaptações entre

múltiplos nós do sistema e a realização de políticas mais complexas de autogerenciamento. Por outro lado, problemas tais como interações conflitantes e custos de comunicação entre *loops* – não presentes em sistemas com um único *loop* – devem ser cuidadosamente analisados.

Descrição. De um modo geral, múltiplos *loops* podem estar presentes em sistemas *self-adaptive* de duas formas: múltiplos *loops* em um mesmo nível de abstração ou *loops* organizados de forma hierárquica. Por exemplo, os diversos nós que constituem uma plataforma de *cloud computing* podem executar, cada um, um *loop* de adaptação local. Estes *loops* possuem uma mesma meta de adaptação (ex: garantir uma certa vazão de processamento) e trabalham no mesmo nível de abstração. Comunicação entre um ou mais elementos MAPE-K (vide Seção 3.2.2, pág. 33) de tais *loops* pode ser necessária, por exemplo, para que o tempo de resposta total do *job* seja controlado. Esta dimensão de projeto captura duas diferentes arquiteturas para comunicação entre *loops* que trabalham no mesmo nível de abstração. Tais soluções apresentam diferentes graus de escalabilidade e *overhead* de controle e foram baseadas no trabalho realizado por Weyns et al. [318].

Loops organizados de forma hierárquica, por outro lado, são utilizados para separar os objetivos de autogerenciamento em diferentes níveis de abstração (camadas), onde níveis mais altos representam metas mais globais (ex: consumo geral de energia do *cluster*). Nestas situações, o sistema *self-adaptive* de mais baixo nível (sistema gerenciado + sistema gerenciador) atua como o sistema gerenciado do *loop* de nível mais alto. *Loops* hierárquicos requerem que análises comportamentais verifiquem se o período de execução do *loop* de mais alto nível é maior ou igual ao tempo de estabilização do sistema atuando no nível mais baixo. Caso contrário, instabilidades ou baixo desempenho de controle podem acontecer. Esta dimensão de projeto captura três diferentes arquiteturas para comunicação em arranjos hierárquicos de *loops*.

Impactos em Atributos de Qualidade. A forma de cooperação entre múltiplos *loops* em um sistema *self-adaptive* tem impacto direto em atributos tais como escalabilidade e disponibilidade e produz *trade-offs* importantes entre desempenho de controle e *overhead* de comunicação. Em sistemas distribuídos com múltiplos *loops* no mesmo nível de abstração, a forma mais simples de controle é a utilização de um único *loop* central que monitora todos os nós remotamente, executa as atividades de análise e planejamento, e finalmente solicita a execução das adaptações necessárias em cada nó.

Embora metas globais sejam mais facilmente atendidas com esta abordagem – em função da presença de informações globais do sistema no *loop* central – a escalabilidade e disponibilidade de controle são obviamente prejudicadas. Tal abordagem é inviável nos cenários atuais de *clusters* formados por milhares de máquinas e com requisitos rigorosos de disponibilidade. Uma solução é descentralizar o controle, melhorando a disponibilidade e reduzindo o *overhead* de comunicação entre *loops*, eventualmente em detrimento do atendimento ótimo das metas globais de adaptação. Diferentes graus de descentralização, com diferentes impactos nos atributos acima descritos, são apresentados como pontos de variação da dimensão de projeto aqui tratada.

Elementos-Alvo. Esta dimensão de projeto deve ser avaliada para cada componente de processo, presente no modelo arquitetural inicial, que atue como parte de outro componente (seja uma propriedade UML com atributo `aggregation` igual a `composite`) e que possua multiplicidade superior maior que 1 ou igual a *. Tais elementos são identificados pela seguinte expressão OCL:

```
1: rootElement.allOwnedElements()->select(
2:    oclIsTypeOf(Property)
3: ).oclAsType(Property)->select(
4:    owner.oclIsTypeOf(Component) and
5:    type.oclIsTypeOf(Component) and
6:    type.oclAsType(Component).
7:       extension_ProcessComponent <> null and
8:    isMultiValued()
9: )
```

Pontos de Variação. Esta dimensão de projeto captura a implementação de cinco formas de cooperação entre múltiplos *loops* em sistemas *self-adaptive*. Dois pontos de variação (VP62 e VP63) implementam cooperação em um mesmo nível de abstração, enquanto outros três (VP64, VP65 e VP66) implementam cooperação hierárquica entre *loops*. Adicionalmente, um ponto de variação representando ausência de cooperação é também definido. As dimensões são apresentadas em ordem crescente de centralização (da mais descentralizada para a mais centralizada) e, consequentemente, em ordem crescente de *overhead* de comunicação entre *loops* e em ordem descrescente de escalabilidade de controle.

VP61) Loops Independentes (sem cooperação): indica que nenhum mecanismo de cooperação será utilizado entre os múltiplos *loops*. Visto que, para a formação de uma arquitetura candidata, toda instância de dimensão de projeto deve ter um ponto de variação selecionado, este ponto de variação é utilizado para indicar que os múltiplos *loops* atuarão de forma independente. Esta solução é adequada para o controle em ambientes onde os múltiplos *loops* implementam metas de adaptação independentes, que não apresentam interações e que não possuem impacto em metas globais. Consequentemente, não há *overhead* de comunicação entre *loops* e a escalabilidade do sistema não é influenciada por aspectos de controle. Nenhuma modificação arquitetural é realizada, por este ponto de variação, no modelo arquitetural sendo construído.

Como exemplo, suponha um ambiente de *cloud computing* executando um *job* cuja vazão total V desejada (valor de referência) é igualmente dividido entre os n nós do *cluster*. Cada nó executaria um *loop* independente, responsável por oferecer vazão V/n. Em *clusters* formados por máquinas heterogêneas (com diferentes capacidades de processamento) tal situação é indesejada, pois impede que *loops* menos favorecidos entreguem uma vazão menor que V/n desde que a vazão total seja maior ou igual a V. Como não há coordenação entre os *loops* é impossível obter informações globais sobre o sistema.

VP62) Descentralização Coordenada por Monitoramento: indica que apenas o elemento M (*Monitor*) de cada *loop* interage com elementos M de *loops* presentes em uma vizinhança localmente conhecida. Este ponto de variação representa a forma mais fraca de acoplamento entre *loops*, permitindo que a obtenção de informação de monitoramento de *loops* vizinhos viabilize uma melhor coordenação para o atendimento da meta. A cardinalidade da vizinhança conhecida tem impacto direto no *overhead* de comunicação, na escalabilidade e no desempenho de controle (atendimento de metas globais).

Vizinhanças de maior cardinalidade disponibilizam mais informações sobre o estado global do sistema (viabilizando adaptações mais eficientes), ao custo de um maior *overhead* de comunicação e menor escalabilidade. Por outro lado, vizinhanças pequenas viabilizam alta escalabilidade em função do baixo *overhead* de comunicação, porém a eficiência das adaptações pode ficar comprometida em função da ausência de informações globais. A decisão sobre o tamanho da vizinhança conhecida não será abordada neste trabalho. As métricas apresentadas na Seção

7.3 comparam os diferentes pontos de variação aqui apresentados supondo que todas trabalham com vizinhanças de mesma cardinalidade. Note que, em função da ausência de interação entre os elementos A (*Analyze*), P (*Plan*) e E (*Execute*) dos *loops*, o uso de técnicas mais complexas de planejamento e a execução coordenada de adaptações nos diversos nós são inviabilizados por este ponto de variação.

Considerando o exemplo discutido anteriormente, este ponto de variação permite a comunicação entre *loops* da vizinhança, com o objetivo de encontrar a vazão total entregue V_V. Dessa forma, é possível atribuir, aos nós individuais, uma fração do erro ($V/n \cdot \#vizinhança - V_V$) em vez de uma fração do valor de referência. Tal abordagem apresenta um melhor desempenho de controle em *clusters* formados por máquinas heterogêneas.

A Tabela 7.9 apresenta as modificações deste ponto de variação. As modificações 1 a 4 realizam a inclusão de novos elementos no modelo inicial. A modificação 1 cria uma nova instância de `SADuSE::EventBus`, componente responsável pelo compartilhamento de informação entre os elementos M de cada *loop* em uma vizinhança. A modificação 2 realiza a criação de duas novas portas, uma utilizada pela instância de `SADuSE::EventBus` recém criada e outra utilizada pelo elemento que representa os múltiplos controladores que deverão cooperar (criados na dimensão de projeto DD2). O tipo de ambas as portas será `SADuSE::IEventComm`, uma interface que define serviços para troca de mensagens de forma desacoplada. A modificação 4 cria uma nova instância de `SADuSE::MSharingAssociation`, uma associação que indica que haverá coordenação (compartilhamento de dados) apenas entre os elementos M dos controladores. Esta associação irá ligar as portas criadas na modificação 2.

As modificações 5 a 9 realizam alteração de propriedades. A modificação 5 indica que a primeira porta criada na modificação 2 fará parte do barramento de eventos criado na modificação 2. A modificação 6 inclui a segunda porta criada na modificação 2 no conjunto de portas do elemento que representa os múltiplos controladores que deverão cooperar. A modificação 7 indica que as portas criadas na modificação 2 serão os elementos envolvidos na associação criada na modificação 4. A modificação 8 ajusta os tipos das portas criadas na modificação 2 para a interface criada na modificação 3. Finalmente, a modificação 9 inclui os elementos criados nas modificações 1, 3 e 4 no conjunto de elementos presentes no mesmo pacote que contém o elemento-alvo em questão.

VP63) Descentralização Totalmente Coordenada: indica que cada elemento M (*Monitor*), A (*Analyze*), P (*Plan*) e E (*Execute*) de cada *loop* interage com os elementos correspondentes de *loops* presentes em uma vizinhança localmente conhecida. Este ponto de variação implica em um *overhead* de comunicação maior que aquele apresentado pelo ponto de variação VP62, mas ainda apresenta boa escalabilidade quando vizinhanças de baixa cardinalidade são utilizadas. Visto que todos os elementos MAPE interagem com os elementos correspondentes dos *loops* na vizinhança, técnicas de planejamento distribuído e execução coordenada de adaptações podem ser implementadas em arquiteturas geradas por este ponto de variação.

No exemplo relacionado a plataformas de *cloud computing*, arquiteturas geradas por este ponto de variação permitiriam a coordenação da execução das adaptações nos múltiplos nós. Um exemplo de tal coordenação é a operação de rebalanceamento dos dados armazenados no *cluster*, onde deve-se aguardar a finalização das adaptações de cada nó antes de executar uma nova iteração do *feedback control loop*. As modificações deste ponto de variação são semelhantes àquelas apresentadas no ponto de variação VP62, exceto que – na modificação 1 – uma associação do tipo `SADuSE::MAPESharingAssociation` é criada, de modo a conectar o barramento de evento aos múltiplos controladores. Esta associação indica que haverá coordenação (compartilhamento de dados) entre todos os elementos MAPE dos controladores.

Modificações de Inclusão de Elementos - VP62		
#c	Metapropriedade	Especificação OCL
01	$addedType(c)$	SADuSE::EventBus
02	$addedType(c)$	Uml::Port

	$guideElementsExp(c)$	**Sequence** {
		addedElements(c1)[index],
		target.extension__ProcessComponent.controllablePort.opposite.owner
		}

03	$addedType(c)$	SADuSE::IEventComm
04	$addedType(c)$	SADuSE::MSharingAssociation
Modificações de Alteração de Propriedades - VP62		
#c	Metapropriedade	Especificação OCL
05	$changedPropertyExp(c)$	addedElements(c1).ownedPort
	$propertyValuesExp(c)$	addedElements(c2)->at(0)
06	$changedPropertyExp(c)$	guideElements(c2)->at(1).**oclAsType**(Component).ownedPort
	$propertyValuesExp(c)$	guideElements(c2)->at(1).**oclAsType**(Component).ownedPort ∪ addedElements(c2)->at(1)
07	$changedPropertyExp(c)$	addedElements(c4).memberEnd
	$propertyValuesExp(c)$	addedElements(c2)
08	$changedPropertyExp(c)$	addedElements(c2).type
	$propertyValuesExp(c)$	addedElements(c3)
09	$changedPropertyExp(c)$	target.package.ownedElement
	$propertyValuesExp(c)$	target.package.ownedElement ∪ addedElements(c1) ∪ addedElements(c3) ∪ addedElements(c4)

Tabela 7.9.: Modificações do ponto de variação VP62.

VP64) Planejamento Regional: indica que um novo controlador, executando somente o elemento P (*Plan*), é criado para centralizar o planejamento de adaptações de um grupo de controladores locais (região). Este novo coordenador é denominado "planejador regional" e múltiplas regiões – cada uma coordenada por um planejador regional particular – podem estar presentes. Coordenação entre planejadores regionais viabiliza a implementação de metas globais de adaptação. Este ponto de variação implementa um estilo híbrido, com cooperação centralizada entre controladores de uma mesma região (forte acoplamento intra-região) e cooperação descentralizada entre regiões diferentes (baixo acoplamento inter-região). A solução apresenta um bom equilíbrio entre escalabilidade e *overhead* de comunicação, além de permitir a implementação de mecanismos complexos para planejamento global.

A Tabela 7.10 apresenta as modificações deste ponto de variação. Note que os pontos de variação VP62 e VP63 não realizam a criação de novos controladores pois a cooperação ocorre entre *loops* em um mesmo nível de abstração. O único componente criado em tais situações é o barramento de eventos, responsável por viabilizar a comunicação desacoplada entre *loops* de cada vizinhança. Já o ponto de variação 64, aqui descrito, realiza a criação de dois componentes adicionais: um novo controlador – o planejador regional criado na modificação 2 – e o barramento de eventos criado na modificação 1 – utilizado para promover a cooperação entre planejadores regionais (através de associações do tipo `SADuSE::PSharingAssociation`). As modificações de alteração de propriedades são semelhantes àquelas apresentadas nos pontos de variação anteriores.

VP65) Controle Centralizado (master/slave): indica que um novo controlador, executando os elementos A (*Analyze*) e P (*Plan*), é criado para centralizar a análise e planejamento das adap-

	Modificações de Inclusão de Elementos - VP64	
#*c*	Metapropriedade	Especificação OCL
01	*addedType*(*c*)	SADuSE::EventBus
02	*addedType*(*c*)	SADuSE::GenericController
03	*addedType*(*c*)	Uml::Port

	guideElementsExp(*c*)	**Sequence** {
		addedElements(c1)->first(),
		addedElements(c2)->first(),
		}

04	*addedType*(*c*)	Uml::Port

	guideElementsExp(*c*)	**Sequence** {
		addedElements(c2)->first(),
		target.extension_ProcessComponent.controllablePort.opposite.owner
		}

05	*addedType*(*c*)	SADuSE::IEventComm
06	*addedType*(*c*)	SADuSE::PSharingAssociation

	guideElementsExp(*c*)	**Sequence** {
		addedElements(c1)->first(),
		addedElements(c2)->first()
		}

	Modificações de Alteração de Propriedades - VP64	
#*c*	Metapropriedade	Especificação OCL
07	*changedPropertyExp*(*c*)	addedElements(c1).ownedPort
	propertyValuesExp(*c*)	addedElements(c3)->at(0)
08	*changedPropertyExp*(*c*)	addedElements(c2).ownedPort
	propertyValuesExp(*c*)	addedElements(c3)->at(1)$\cup$ addedElements(c4)->at(0)
09	*changedPropertyExp*(*c*)	guideElements(c3)->at(1).**oclAsType**(Component).ownedPort
	propertyValuesExp(*c*)	guideElements(c2)->at(1).**oclAsType**(Component).ownedPort $\cup$ addedElements(c4)->at(1)
10	*changedPropertyExp*(*c*)	addedElements(c6).memberEnd
	propertyValuesExp(*c*)	addedElements(c3)[index]$\cup$ addedElements(c4)[index]
11	*changedPropertyExp*(*c*)	addedElements(c3).type
	propertyValuesExp(*c*)	addedElements(c5)
12	*changedPropertyExp*(*c*)	addedElements(c4).type
	propertyValuesExp(*c*)	addedElements(c5)
13	*changedPropertyExp*(*c*)	target.package.ownedElement
	propertyValuesExp(*c*)	target.package.ownedElement $\cup$ addedElements(c1) $\cup$ addedElements(c2)$\cup$ addedElements(c5)$\cup$ addedElements(c6)

Tabela 7.10.: Modificações do ponto de variação VP64.

tações de todos os controladores locais do sistema. Os controladores locais executam apenas os elementos M (*Monitor*) e E (*Execute*). Este ponto de variação implementa uma cooperação com maior centralização, apresentando excelente desempenho de controle (em função da disponibilidade de dados globais do sistema) ao custo de limitações severas na escalabilidade e alto *overhead* de comunicação entre *loops*. Não há coordenação entre os elementos de mesmo tipo nos *loops* envolvidos.

A Tabela 7.11 apresenta as modificações deste ponto de variação. Note que barramentos de eventos não são utilizados e um único controlador adicional (o *master*) é criado na modificação 1. Controladores *slave* são ligados ao controlador *master* através de associações do tipo `SADuSE::MasterSlaveAssociation`, caracterizando a natureza centralizada da arquitetura de

#c	Metapropriedade	Especificação OCL
Modificações de Inclusão de Elementos - VP65		
01	*addedType*(*c*)	SADuSE::GenericController
02	*addedType*(*c*)	Uml::Port
	guideElementsExp(*c*)	**Sequence** { addedElements(c1)[index], target.extension_ProcessComponent.controllablePort.opposite.owner }
03	*addedType*(*c*)	SADuSE::IMasterSlave
04	*addedType*(*c*)	SADuSE::MasterSlaveAssociation
Modificações de Alteração de Propriedades - VP65		
#c	Metapropriedade	Especificação OCL
05	*changedPropertyExp*(*c*)	addedElements(c1).ownedPort
	propertyValuesExp(*c*)	addedElements(c2)->at(0)
06	*changedPropertyExp*(*c*)	guideElements(c2)->at(1).**oclAsType**(Component).ownedPort
	propertyValuesExp(*c*)	guideElements(c2)->at(1).**oclAsType**(Component).ownedPort ∪ addedElements(c2)->at(1)
07	*changedPropertyExp*(*c*)	addedElements(c4).memberEnd
	propertyValuesExp(*c*)	addedElements(c2)
08	*changedPropertyExp*(*c*)	addedElements(c2).type
	propertyValuesExp(*c*)	addedElements(c3)
09	*changedPropertyExp*(*c*)	target.package.ownedElement
	propertyValuesExp(*c*)	target.package.ownedElement ∪ addedElements(c1) ∪ addedElements(c4)

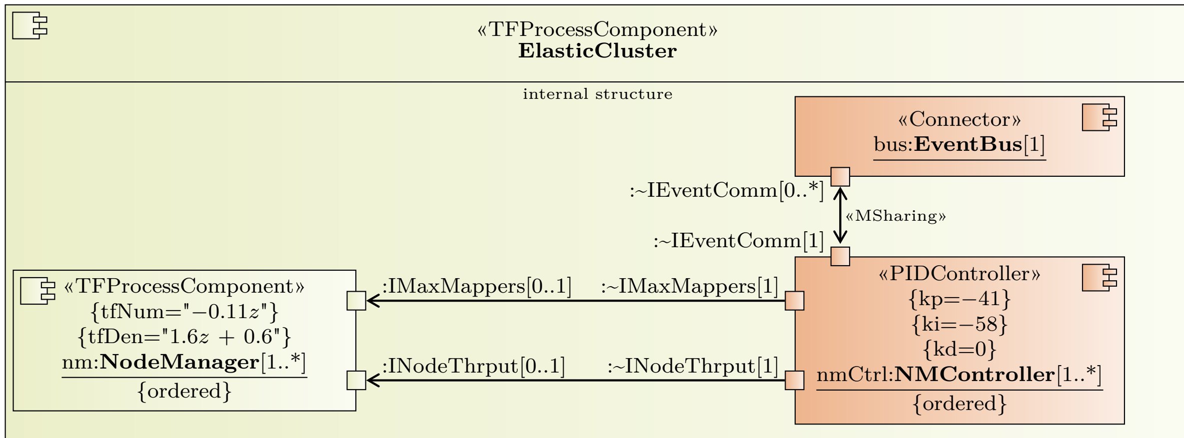
Tabela 7.11.: Modificações do ponto de variação VP65.

controle. As modificações de alteração de propriedades são semelhantes àquelas apresentadas nos pontos de variação anteriores.

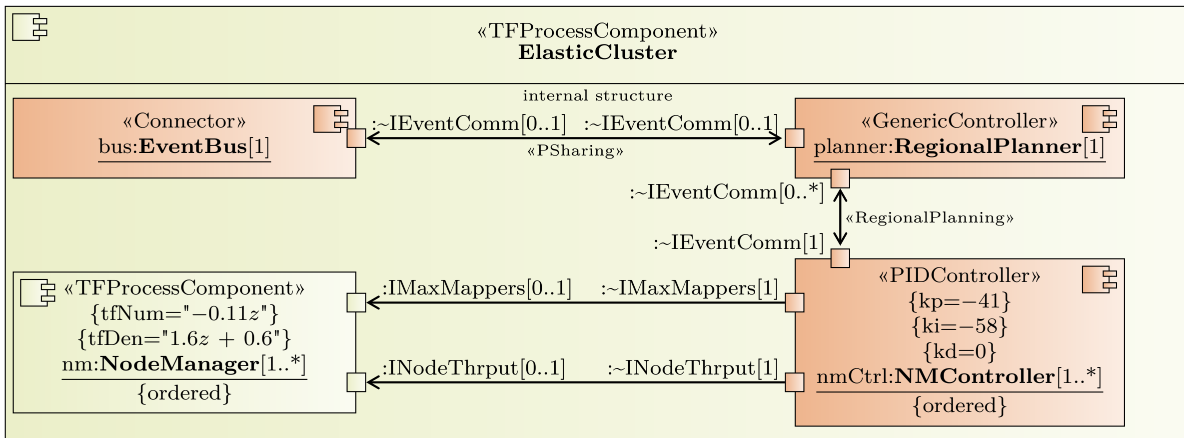
VP66) Controle Hierárquico (múltiplos níveis): indica que o sistema *self-adaptive* de nível mais baixo (sistema gerenciado + sistema gerenciador) irá atuar como um novo sistema gerenciado, adaptado por um *feedback control loop* de mais alto nível. Todos os elementos MAPE estão presentes em cada controlador. Não há coordenação entre os elementos de mesmo tipo nos *loops* envolvidos. De forma similar ao controle centralizado, este ponto de variação produz arquiteturas com escalabilidade limitada e alto *overhead* de comunicação.

As modificações deste ponto de variação são semelhantes àquelas apresentadas no ponto de variação VP65, exceto que `SADuSE::HierarchicalAssociation` é utilizado como tipo da associação criada – na modificação 4 – para conectar os controladores do nível mais baixo ao controlador do nível mais alto. Adicionalmente, o tipo de porta – criado na modificação 3 – passa ser `SADuSE::IHierarchical`. Note que o número final de níveis na hierarquia de controle depende do número de componentes aninhados (atuando como propriedades de multiplicidade superior maior que 1 ou igual a *) presentes no modelo arquitetural inicial. As modificações de alteração de propriedades são semelhantes àquelas apresentadas nos pontos de variação anteriores.

Exemplos. As Figuras 7.6(a) e 7.6(b) apresentam, respectivamente, as arquiteturas resultantes da aplicação dos pontos de variação VP62 (descentralização coordenada por monitoramento) e VP64 (planejamento regional).



(a)



(b)

= sistema gerenciado (modelo de entrada)

= sistema gerenciador (automaticamente incluído pela abordagem aqui proposta)

Figura 7.6.: Arquiteturas resultantes da aplicação dos pontos de variação VP62 (a) e VP64 (b).

As dimensões de projeto apresentadas nesta seção de forma alguma representam uma captura exaustiva de todos os aspectos arquiteturais importantes no projeto de sistemas *self-adaptive*. Outras dimensões, igualmente relevantes, devem ser investigadas e podem ser futuramente incluídas no espaço de projeto para sistemas *self-adaptive* aqui proposto. Dentre estas dimensões adicionais, destacam-se: mecanismo de filtragem utilizado no monitoramento (exs: *moving-average* ou *sampling* [138]), forma de sensoriamento e de atuação (exs: não-intrusiva via *logs*, IPC/RPC, intrusiva) e grau de acoplamento com o sistema gerenciado (exs: *in-thread*, *in-process*, *out-process* ou *remote*). Tais dimensões podem ter impactos relevantes nos atributos de qualidade aqui considerados e nas métricas propostas na Seção 7.3. Acredita-se que a infraestrutura para otimização multiobjetivo de arquiteturas apresentadas no Capítulo 8 viabilize um bom desempenho de busca (tanto referente ao tempo de execução das otimizações quanto à qualidade das arquiteturas encontradas) em espaços de projeto maiores. Tal aspecto, entretanto, não foi avaliado neste trabalho.

7.3. Métricas de Qualidade do SA:DuSE

A definição de um espaço de projeto para domínio específico envolve a especificação não somente das dimensões de projeto e seus pontos de variação com suas respectivas modificações arquiteturais, mas também das métricas que avaliam cada arquitetura gerada por vetores candidatos (vide Seção 6.2, pág. 120) em relação a atributos de qualidade relevantes do domínio.

Esta seção apresenta as quatro métricas de qualidade, definidas neste trabalho, para a avaliação das diversas arquiteturas para sistemas gerenciadores geradas pelas dimensões de projeto descritas na Seção 7.2. As duas primeiras – tempo médio de estabilização e sobressinal médio – têm sua origem na Teoria de Controle. Estas métricas foram conceitualmente apresentadas na Seção 3.3.4.1, pág. 47, e as fórmulas para obtenção dos seus valores apresentadas na Tabela 3.5, pág. 52. As duas últimas – robustez de controle e *overhead* de controle – foram definidas como parte deste trabalho e definem grandezas adimensionais, viabilizando a comparação de duas ou mais arquiteturas em relação a estes dois atributos de qualidade.

Conforme apresentado no Capítulo 6, uma métrica de qualidade é uma uma tupla $qm = \langle \Phi, g \rangle$. Φ é uma função $\Phi \colon \mathcal{F}_{asds} \rightarrow \mathcal{V}$, onde $\mathcal{F}_{asds}$ é um espaço arquitetural viável e $\mathcal{V}$ é um conjunto com escala no mínimo intervalar. g assume os valores 1 e -1 para indicar se a métrica é melhorada com valores maiores (1: a meta é maximizá-la) ou menores (-1: a meta é minimizá-la). Para cada uma das métricas apresentadas abaixo, serão apresentados: o conjunto $\mathcal{V}$ adotado, a forma de avaliação de métrica a partir de uma arquitetura candidata (função Φ) e a meta de otimização da métrica (valor de g igual a 1 ou -1).

7.3.1. Tempo Médio de Estabilização

Esta métrica calcula o tempo médio de estabilização $\overline{K}_s$ de todos os *loops* presentes na arquitetura candidata. Adaptações com tempos de estabilização baixos são geralmente desejados, pois minimizam a possibilidade de interações conflitantes em hierarquias de *loops* e viabilizam o atendimento de SLAs (*Service Level Agreements*) mais rigorosos. Portanto, valores menores para esta métrica representam arquiteturas de melhor qualidade ($g = -1$). A forma de avaliação da métrica (função Φ), apresentada a seguir, mapeia uma arquitetura candidata em um valor $v \in \mathbb{R}_+$ representando o tempo médio de estabilização.

```
1: let controllers = rootElement.allOwnedElements()->select(
2:    oclIsTypeOf(Component) and extension_Controller <> null
3: ).oclAsType(Component).extension_Controller in
4: controllers->collect(
5:    -4 / log(abs(largestPole(controllablePort.opposite.owner)))
6: )->sum() / controllers->size()
```

Os controladores presentes na arquitetura candidata sendo avaliada são obtidos com as sentenças presentes nas linhas 1 a 3 e armazenados na variável `controllers`. Em seguida, para cada controlador, estima-se o tempo médio de estabilização utilizando o método apresentado na Tabela 3.5, pág. 52. A função `largestPole`, parte do estereótipo `SADuSE::Controller` e descrita no Apêndice A, retorna o polo de maior magnitude no sistema – em malha fechada (vide Seção 3.3.4.1, pág. 46) – formado pelo controlador em questão e o sistema por ele gerenciado (obtido via `controllablePort.opposite.owner`).

A função `largestPole` é redefinida, no *profile* UML do SA:DuSE, em cada controlador representando subtipos particulares de `SADuSE::Controller`. Dessa forma, é possível a obtenção do polo de maior magnitude a partir de modelos descritos nos diferentes formatos apresentados para a dimensão de projeto DD1 (funções de transferência, espaço de estados e FOPDT). Mais detalhes podem ser encontrados no Apêndice A.

Note que a condição para estabilidade do sistema final é que todos os polos tenha magnitude $<$ 1. Supondo que o modelo representando o sistema inicial (não controlado) é estável, associado ao fato que todas as técnicas de sintonia apresentadas na dimensão de projeto DD4 produzem sistemas finais estáveis, conclui-se que o polo r retornado pela função `largestPole` possui magnitude < 1 ($|r| < 1$). Consequentemente, o valor de $\log(r)$ é negativo, o que faz com que o tempo de estabilização seja positivo, conforme esperado.

A métrica aqui apresentada tem caráter preditivo (espera-se encontrar valores semelhantes em protótipos reais implementando a arquitetura candidata em questão) e a unidade de medida é a mesma utilizada na descrição da dinâmica do sistema representado pelo modelo arquitetural inicial. Tais aspectos são discutidos nas atividades de avaliação apresentadas no Capítulo 10.

7.3.2. Sobressinal Médio

Esta métrica calcula o sobressinal médio $\overline{M}_p$ de todos os *loops* presentes na arquitetura candidata. Adaptações sem sobressinal não apresentam oscilações durante a resposta transiente do sistema e são desejáveis devido a um uso mais comedido de recursos durante as adaptações. Portanto, valores menores para esta métrica representam arquiteturas de melhor qualidade ($g = -1$). Entretanto, a avaliação do *trade-off* comumente percebido entre sobressinal e tempo de estabilização é um aspecto importante no projeto de controladores efetivos.

Controladores que produzem tempos de estabilização mais baixos utilizam adaptações (entradas de controle) mais drásticas, frequentemente tornando inevitável a presença do sobressinal. Da mesma forma, a ausência de sobressinal requer o uso de adaptações mais moderadas, de modo que a saída medida não ultrapasse o valor de referência. A forma de avaliação da métrica (função Φ), apresentada a seguir, mapeia uma arquitetura candidata em um valor $v \in \mathbb{R}_+$ representando o percentual máximo médio pelo qual a saída medida ultrapassa o valor de referência.

```
 1: let controllers = rootElement.allOwnedElements()->select(
 2:    oclIsTypeOf(Component) and extension_Controller <> null
 3: ).oclAsType(Component).extension_Controller in
 4: controllers->collect(
 5:    let domPole = largestPole(controllablePort.opposite.owner) in
 6:    if domPole.oclIsTypeOf(Real) then
 7:       if domPole >= 0 then 0 else abs(domPole) endif
 8:    else
 9:       domPole.realPart()^(PI/domPole.imaginaryPart())
10:    endif
11: )->sum() / controllers->size()
```

A métrica acima apresentada é calculada de acordo com o apresentado na Tabela 3.5, pág. 52. Sistemas finais caracterizados por um polo dominante real e positivo não irão produzir oscilações e, portanto, apresentam sobressinal zero (linha 7). Sistemas com polos reais porém negativos apresentam oscilação maior quanto mais o polo se aproxima de 1. Lembre-se que convergência requer que o polo dominante seja menor que 1. Sistemas com polo dominante igual a 1 são estáveis, porém não convergem (sobressinal de 100%). Sistemas com polo dominante maior que 1 são instáveis. Finalmente, sistemas com polo dominante complexo (na verdade, um par de polos dominantes) apresentam oscilações na ordem de $r^{\pi/|\theta|}$, onde r é a parte real e $|\theta|$ é o valor absoluto da parte imaginária dos polos (linha 9).

Novamente, a métrica aqui apresentada tem caráter preditivo e representa o percentual de ultrapassagem, pela saída medida, do valor de referência em uso. Tais aspectos são também discutidos nas atividades de avaliação apresentadas no Capítulo 10.

7.3.3. Efetividade de Controle

Esta métrica estima quão efetivo é o controle geral obtido pela atuação dos *loops* que compõem o sistema final. A efetividade de controle de um sistema *self-adaptive* é influenciada por dois aspectos principais: *i)* a capacidade dos controladores continuarem apresentando bom desempenho quando as dinâmicas do sistema e do ambiente variam de forma não prevista (robustez); e *ii)* o grau de cooperação entre múltiplos *loops*, diretamente impactado por quanto de informação global do sistema é conhecido em cada *loop*. A parcela de efetividade de controle dada pela robustez é definida, neste trabalho, como:

$$E_r(c) = \begin{cases} \dfrac{x_t - 1}{|asds.ds.dd_5.VP| - 1} & ;\text{ se } \alpha_r^E = 0 \\ \\ \dfrac{e^{\alpha_r^E \cdot \frac{x_t - 1}{|asds.ds.dd_5.VP|-1}} - 1}{e^{\alpha_r^E} - 1} & ;\text{ se } \alpha_r^E \neq 0 \end{cases} \tag{7.1}$$

$$;\text{ onde } t \in \{1, \ldots |DDI|\} \mid ddi_t.dd = dd_5 \wedge ddi_t.te = c.controllablePort.opposite.owner$$

O modelo acima atribui valores no intervalo $[0, 1]$ de acordo com o ponto de variação escolhido na dimensão DD5 (grau de adaptabilidade do controlador). Visto que os pontos de variação desta dimensão foram especificados em ordem crescente de adaptabilidade de controle, o valor de E_r será 0 (pior efetividade) quando o ponto de variação VP51 (ganho fixo) for selecionado e 1 (melhor efetividade) quando o ponto de variação VP55 (controle reconfigurável) for selecionado.

O valor de E_r para os demais pontos de variação (VP52 a VP54) depende do valor atribuído a α_r^E, um parâmetro a ser definido para a avaliação da métrica. Quando $\alpha_r^E = 0$, a efetividade de controle cresce linearmente em função do ponto de variação selecionado (as contribuições dos pontos de variação VP51 a VP55 seriam, respectivamente: 0, 0.25, 0.50, 0.75 e 1). Valores positivos de α_r^E indicam que um ganho considerável de efetividade só é percebida nos pontos de variação que representam adaptações mais agressivas de controle. Por exemplo, para $\alpha_r^E = 2$, as contribuições dos pontos de variação VP51 a VP55 seriam, respectivamente: 0, 0.10, 0.26, 0.54 e 1.

De forma similar, valores negativos de α_r^E indicam que há um ganho considerável de efetividade já na utilização dos pontos de variação que representam adaptações menos agressivas. Por exemplo, para $\alpha_r^E = -2$, as contribuições dos pontos de variação VP51 a VP55 seriam, respectivamente: 0, 0.45, 0.73, 0.89 e 1. O valor de α_r^E deve ser escolhido, pelo arquiteto, durante as avaliações das arquiteturas candidatas produzidas pelo espaço de projeto SA:DuSE. A Figura 7.7 ilustra como o valor de α_r^E influencia a atribuição dos valores de E_r.

A parcela de efetividade de controle dada pelo grau de cooperação entre os múltiplos *loops* – E_m – é definida de forma semelhante, utilizando desta vez um novo parâmetro α_m^E e a dimensão DD6 no lugar da dimensão DD5. Note também que o domínio $[0, 1]$ é, desta vez, dividido em cinco partes iguais visto que a dimensão DD6 possui seis pontos de variação. Finalmente, a efetividade final de um controlador c é dada por:

$$E(c) = \beta^E \cdot E_r(c) + (1 - \beta^E) \cdot E_m(c) \qquad ;\text{ onde } 0 \leq \beta^E \leq 1 \tag{7.2}$$

O parâmetro β^E permite variar a importância dada a cada uma das parcelas (robustez e cooperação) no cálculo final da efetividade do controlador c. A efetividade de controle final da arquitetura candidata é a média aritmética das efetividades de cada controlador presente no modelo. A estimativa da métrica requer, portanto, a escolha dos valores de α_r^E, α_m^E e β^E. Valores

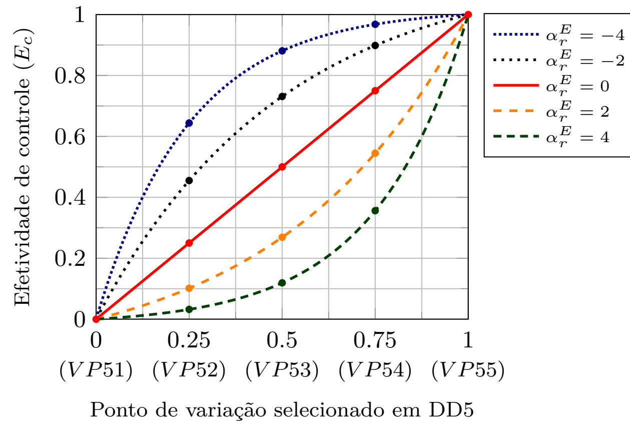


Figura 7.7.: Influência de α_r^E na efetividade de controle gerada pelos pontos de variação da dimensão DD5.

maiores para esta métrica representam arquiteturas de melhor qualidade ($g = 1$). Vale ressaltar que a métrica aqui apresentada produz um valor adimensional, sem caráter preditivo e utilizado somente como critério de comparação de duas ou mais arquiteturas candidatas em relação a este atributo de qualidade.

7.3.4. Overhead de Controle

Esta métrica estima o *overhead* de comunicação e processamento introduzido pelos *loops* que compõem o sistema final. Um *overhead* de controle alto demais pode inviabilizar o uso da arquitetura projetada para o sistema gerenciador ou pode modificar a dinâmica do sistema gerenciado quando os *loops* são implantados na mesma máquina do sistema-alvo.

Duas dimensões de projeto apresentadas na Seção 7.2 têm impacto direto no *overhead* de controle: grau de adaptabilidade do controlador (DD5) e forma de cooperação entre múltiplos *loops* (DD6). Conforme mencionado anteriormente, os pontos de variação das dimensões de projeto DD5 e DD6 foram especificados em ordem crescente de *overhead*. Dessa forma, um modelo similar ao utilizado para avaliar efetividade de controle é aqui aplicado na estimativa do *overhead*. O *overhead* final de um controlador c é dada por:

$$O(c) = \beta^O \cdot O_r(c) + (1 - \beta^O) \cdot O_m(c) \qquad ; \text{onde } 0 \leq \beta^O \leq 1 \tag{7.3}$$

De forma similar ao apresentado anteriormente, o cálculo de $O_r(c)$ e $O_m(c)$ requerem a definição, respectivamente, dos parâmetros α_r^O e α_m^O. O parâmetro β^O indica a importância dada a cada uma das parcelas (robustez e cooperação) no cálculo final do *overhead* do controlador c. O *overhead* final da arquitetura candidata é a média aritmética dos *overheads* apresentados por cada controlador presente no modelo. A estimativa da métrica requer, portanto, a escolha dos valores de α_r^O, α_m^O e β^O. Valores menores para esta métrica representam arquiteturas de melhor qualidade ($g = -1$). Novamente, a métrica aqui apresentada não possui caráter preditivo e é utilizada somente para comparar duas ou mais arquiteturas candidatas em relação a este atributo de qualidade.

7.4. Resumo do Capítulo

Este capítulo apresentou o SA:DuSE – um espaço de projeto arquitetural para sistemas *self-adaptive* – descrito como uma instância da formalização para espaços de projeto genéricos apresentada no Capítulo 6. O SA:DuSE realiza a captura sistemática de soluções para cinco dimen-

sões de projeto importantes do domínio e especifica quatro métricas para avaliação de atributos de qualidade relevantes no projeto de arquiteturas para sistemas *self-adaptive*.

A dimensão de projeto DD1 – modelagem do sistema – é avaliada manualmente pelo arquiteto e sua solução anotada diretamente no modelo inicial submetido ao processo de otimização. Tal etapa é necessária para gerar as anotações mínimas que irão subsidiar a automação das decisões arquiteturais referentes às demais dimensões de projeto. O *profile* UML associado ao SA:DuSE permite a definição das dinâmicas do sistema gerenciado através de modelos de função de transferência, espaços de estado e FOPDT (*First-Order Plus Dead Time*).

A dimensão de projeto DD2 captura as soluções relacionadas ao uso de um ou múltiplos *loops* quando o sistema-alvo apresenta mais de uma porta controlável. Dois pontos de variação foram definidos para esta dimensão de projeto: SISO (VP21) e MIMO (VP22). O ponto de variação VP21 adiciona um controlador SISO para cada porta controlável do componente de processo, ao passo que o ponto de variação VP22 utiliza um único controlador MIMO para gerenciar todo o sistema gerenciado.

A dimensão de projeto DD3 lida com a escolha da lei de controle a ser utilizada em cada controlador. Um conjunto de sete diferentes leis (pontos de variação) foram capturadas nesta dimensão: Proporcional, Integral, Proporcional-Integral, Proporcional-Derivativo, Proporcional-Integral-Derivativo, *Static State Feedback Control* e *Dynamic State Feedback Control*.

A dimensão de projeto DD4 captura as técnicas utilizadas para sintonizar os parâmetros dos controladores criados na dimensão de projeto DD2, respeitada a lei de controle selecionada na dimensão de projeto DD3. Sete técnicas foram capturadas nesta dimensão de projeto: CHR (Chien-Hrones-Reswick) – nas suas quatro variações –, Ziegler-Nichols, Cohen-Coon e LQR (*Linear-Quadratic Regulator*).

A dimensão de projeto DD5 define diferentes soluções para adaptação do sistema gerenciador em *run-time*, permitindo uma melhor robustez de controle naquelas situações onde o sistema/ambiente se desvia das condições apresentadas durante a fase de sintonia. Cinco possibilidades (pontos de variação) – representando diferentes graus de adaptabilidade do controlador – foram apresentadas: ganho fixo (sem adaptação de controle), escalonamento de ganho, MIAC (*Model Identification Adaptive Control*), MRAC (*Model Reference Adaptive Control*) e controle reconfigurável.

Finalmente, a dimensão DD6 captura diferentes formas de cooperação, utilizadas quando múltiplos *loops* estão presentes na arquitetura final do sistema *self-adaptive*. Seis possibilidades – com diferentes graus de efetividade de controle e *overhead* de comunicação – foram apresentadas: *loops* independentes (sem cooperação), descentralização coordenada por monitoramento, descentralização totalmente coordenada, planejamento regional, controle centralizado e controle hierárquico.

Para cada um dos pontos de variação acima descritos, foram apresentadas as modificações de inclusão de novos elementos e de alteração de propriedades que viabilizam o projeto automático das arquiteturas dos sistemas gerenciadores. Adicionalmente, as expressões para a identificação dos elementos-alvo de cada dimensão foram apresentados, definindo os elementos do modelo arquitetural inicial que irão demandar a criação das instâncias de dimensão de projeto.

Quatro métricas para avaliação de arquiteturas candidatas foram apresentadas. A primeira – tempo médio de estabilização – calcula quanto tempo, em média, o sistema precisa para atingir um novo estado estacionário após a ocorrência de perturbação ou modificação do valor de referência. A segunda – sobressinal médio – avalia a arquitetura candidata em relação ao grau de oscilação apresentado durante a resposta transiente. Estas duas métricas têm origem na Teoria de Controle e são utilizadas para fins preditivos.

A terceira métrica – efetividade de controle – atribui a cada arquitetura candidata um número adimensional que viabiliza a comparação de arquiteturas em relação à robustez e grau de cooperação dos *loops* presentes. Finalmente, a quarta métrica – *overhead* de controle – estima o custo computacional e de comunicação gerado pela inclusão dos *loops* responsáveis pelas adaptações.

Capítulo 8

Otimização no Projeto Arquitetural de Sistemas Self-Adaptive

It is meaningless to speak in the name of – or against – Reason, Truth, or Knowledge.
Michel Foucault

Este capítulo apresenta o mecanismo para otimização multiobjetivo de arquiteturas proposto neste trabalho. A solução utiliza metaheurísticas para a realização de busca por arquiteturas ótimas em espaços de projeto. Este mecanismo pode ser aplicado não só ao espaço de projeto SA:DuSE – apresentado no Capítulo 7 – mas a qualquer outro espaço de projeto descrito como uma instância dos constructos apresentados no Capítulo 6.

A Seção 8.1 apresenta a definição do problema de otimização de arquiteturas caracterizado pela busca por boas soluções no espaço de projeto formalizado no Capítulo 6. A Seção 8.2 apresenta os detalhes sobre a forma de representação de arquiteturas como indivíduos de uma população e sobre os operadores utilizados na seleção, recombinação e mutação de soluções. A Seção 8.3 discute as premissas e limitações do mecanismo aqui apresentado, enquanto a Seção 8.4 apresenta o resumo deste capítulo. A Figura 8.1 apresenta o roteiro deste capítulo.

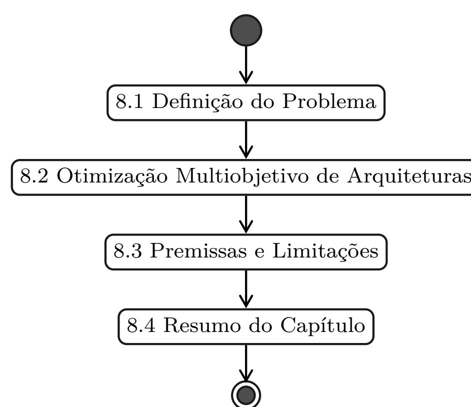


Figura 8.1.: Roteiro do capítulo 8.

8.1. Definição do Problema de Otimização

As formalizações apresentadas no Capítulo 6 permitem a definição de espaços de projeto para domínios específicos, formados por diferentes extensões aplicadas a um modelo arquitetural inicial.

O problema de otimização aqui apresentado representa a busca automática por arquiteturas participantes de um *Pareto-front*, preferencialmente global, que explicitamente evidencie o *trade-off* de atendimento das métricas de qualidade definidas para o espaço de projeto em questão.

Conforme descrito na Seção 6.2, pág. 120, o espaço de decisão arquitetural $\mathcal{D}_{asds}$ (denominado espaço de projeto específico de aplicação) – gerado a partir de um espaço de projeto dd e um modelo arquitetural inicial M – é o produto cartesiano dos índices dos pontos de variação das dimensões de projeto associadas a cada instância de dimensão de projeto criada. O Algoritmo 6.1 (pág. 122) apresentou os passos necessários à construção de $\mathcal{D}_{asds}$ a partir de dd e M.

Um vetor candidato $\mathbf{x} \in \mathcal{D}_{asds}$ descreve uma localização particular de $\mathcal{D}_{asds}$ e, portanto, armazena os índices dos pontos de variação cujas extensões (modificações arquiteturais) devem ser aplicadas ao modelo inicial de modo a derivar a arquitetura representada por $\mathbf{x}$. As condições para detecção de arquiteturas inválidas (combinações incompatíveis de pontos de variação) foram apresentadas no Capítulo 6 e o Algoritmo 6.2 (pág. 123) descreveu como obter uma arquitetura candidata M_r (ou uma arquitetura inválida, se for o caso) a partir de um vetor candidato $\mathbf{x}$.

Seja $\mathcal{F}_{asds}$ o espaço arquitetural viável derivado de $\mathcal{D}_{asds}$, $\mathbf{x}' \in \mathcal{F}_{asds}$ um vetor candidato representando uma arquitetura válida e $\mathcal{T}\colon \mathbf{x}' \to M$ a função que obtém a arquitetura M representada pelo vetor candidato $\mathbf{x}'$ (Algoritmo 6.2). Seja $QM = \{qm_1, qm_2, \ldots, qm_n\}$ o conjunto de métricas de qualidade definidas para o espaço de projeto em uso. O espaço objetivo $\mathcal{O}$ é definido, dessa forma, como o produto cartesiano $\mathcal{V}_1 \times \mathcal{V}_2 \times \ldots \times \mathcal{V}_n$, onde $\mathcal{V}_i$ é a imagem da função Φ_i de avaliação da i-ésima métrica. Seja $\Phi_{QM}(M_r)$ a função que avalia todas as métricas QM em um modelo arquitetural válido M_r:

$$\begin{aligned} &\Phi_{QM}\colon \mathcal{F}_{asds} \to \mathcal{O} \\ &\Phi_{QM}(M_r) \mapsto (-g_1 \cdot \Phi_1(M_r), -g_2 \cdot \Phi_2(M_r), \ldots, -g_n \cdot \Phi_n(M_r)) \end{aligned} \tag{8.1}$$

Conforme mencionado na Seção 6.2, pág. 121, cada métrica de qualidade define um valor $g \in \{-1, 1\}$ que indica se a métrica deve ser minimizada (-1) ou maximizada (1). Ao multiplicar cada avaliação de métrica Φ_i por $-g_i$, viabiliza-se o tratamento uniforme da otimização como sendo um problema de minimização, mesmo quando certas métricas desejam ser maximizadas. Com estas definições, pode-se finalmente apresentar o problema de otimização de arquiteturas aqui proposto como:

$$\overset{\prec}{\min_{\mathbf{x}' \in \mathcal{F}_{asds}}} \; \Phi_{QM}(\mathcal{T}(\mathbf{x}')) \tag{8.2}$$

; onde $\overset{\prec}{\min}$ é o operador de minimização por dominância (vide Seção 4.2.1, pág. 70).

A escolha da técnica de otimização mais adequada para a busca por boas arquiteturas depende de certas propriedades apresentadas pelo problema acima formulado. Tais propriedades são descritas a seguir.

O problema de otimização é multiobjetivo. Espaços de projeto, construídos com base na formalização anteriormente apresentada, provavelmente incluirão mais de uma métrica para avaliação da qualidade das arquiteturas candidatas. Adicionalmente, métricas conflitantes são comuns na avaliação de atributos de qualidade em sistemas computacionais. No caso particular de sistemas *self-adaptive*, é comum a presença de *trade-offs* entre tempo de estabilização e sobressinal, bem como entre *overhead* de controle e efetividade das adaptações no atendimento de metas globais. Por este motivo, técnicas de otimização com articulação *a posteriori* de preferência (vide Seção 4.2, pág. 67) são preferíveis na busca por arquiteturas efetivas.

O problema possui variáveis de decisão discretas. Visto que arquiteturas candidatas são formadas a partir de contribuições arquiteturais definidas pelos índices de pontos de variação selecionados em cada instância de dimensão de projeto, vetores candidatos apresentam coordenadas sempre com valores inteiros e positivos. Tal condição favorece o uso de representações binárias de indivíduos durante a otimização (genes como sequência de *bits*). Por outro lado, as variáveis de decisão assumem valores finitos (definidos pelo número de pontos de variação da dimensão em questão) influenciando a forma com que os operadores de recombinação e mutação atuam.

O espaço de busca é finito, porém potencialmente grande. Conforme apresentado no Capítulo 6, o número de vetores candidatos diferentes gerados por um espaço de projeto ds quando aplicado a um modelo arquitetural inicial M é:

$$\prod_{dd \in ds.DD} |dd.VP|^{|dd.targetElements(M)|} \tag{8.3}$$

Para o espaço de projeto SA:DuSE, apresentado no Capítulo 7, o número total de arquiteturas capturadas (incluindo as inválidas) é:

$$(2 \times 7 \times 7 \times 5 \times 6)^{\#portas\text{-}control\acute{a}veis} \tag{8.4}$$

Para um modelo de entrada simples, com um único componente de processo apresentando duas portas controláveis, o número total de possibilidades de arquiteturas é 8.643.600. Com dois componentes aninhados apresentando duas portas cada um, este número sobe para 7.4711821e13. O uso de algoritmos de otimização que obtenham *Pareto-fronts* com alta convergência e alta diversidade é, portanto, fundamental no problema aqui apresentado.

O problema não adota premissas sobre as funções-objetivo. Embora o uso de métricas que apresentem valores representados em uma escala no mínimo intervalar seja requerido neste trabalho, nenhuma premissa adicional – tal como linearidade, continuidade ou diferenciabilidade – é assumida. Este aspecto é importante para viabilizar o uso, da abordagem aqui proposta, em domínios cujos atributos de qualidade são avaliados por métricas de qualquer natureza (ex: simulação). O uso de metaheurísticas é um dos mecanismos atuais para otimização multiobjetivo sem a adoção de premissas sobre as funções-objetivo.

Motivado pelas propriedades acima apresentadas, a infraestrutura para otimização de arquiteturas proposta neste trabalho faz uso de uma classe particular de técnicas de otimização multiobjetivo: os algoritmos evolucionários baseados em metaheurísticas (vide Seção 4.3.2, pág. 75). Atualmente, o algoritmo NSGA-II (apresentado na Seção 4.3.2.2, pág. 80) é utilizado como *backend* de otimização, mas outros algoritmos da mesma classe podem ser facilmente utilizados (ver Capítulo 9, pág. 175).

8.2. Otimização Multiobjetivo de Arquiteturas

Esta seção apresenta detalhes sobre como abordagens evolucionárias de otimização multiobjetivo foram utilizadas, neste trabalho, para automatizar a busca por arquiteturas efetivas dentre aquelas representadas em um espaço de projeto. A Seção 8.2.1 descreve como os indivíduos de uma população (possíveis arquiteturas candidatas) foram representados utilizando cromossomos (conjunto de genes) com codificação binária. A Seção 8.2.2 apresenta os operadores de seleção, recombinação e mutação utilizados, bem como os aspectos necessários à identificação de arquiteturas inválidas.

8.2.1. Representação das Soluções

A concepção de espaços de projeto como um conjunto de dimensões ortogonais de projeto, formalizada no Capítulo 6, viabiliza a utilização de vetores candidatos como representantes diretos das possíveis arquiteturas resultantes do processo. Conforme mencionado anteriormente, um vetor candidato $\mathbf{x} = (x_1, x_2, \ldots, x_{|asds.DDI|})$ armazena os índices dos pontos de variação selecionados para cada instância de dimensão de projeto em $asds.DDI$ (conjunto das instâncias de dimensão de projeto criadas quando o espaço de projeto $asds.ds$ é aplicado no modelo arquitetural inicial $asds.M$). Dessa forma, cada coordenada x_i de $\mathbf{x}$ é número inteiro entre 1 e $|asds.ddi_i.dd.VP|$. Cada vetor candidato pode, portanto, ser mapeado em um cromossomo com representação binária na forma:

$$\overbrace{010101011\ldots0}^{\substack{\text{Gene 1 de tamanho}\\ \lceil\log_2(|asds.ddi_1.dd.VP|)\rceil}}\underbrace{100111001\ldots0}_{\substack{\text{Gene 2 de tamanho}\\ \lceil\log_2(|asds.ddi_2.dd.VP|)\rceil}}\quad\ldots\quad\overbrace{001101000\ldots0}^{\substack{\text{Gene } n \text{ de tamanho}\\ \lceil\log_2(|asds.ddi_n.dd.VP|)\rceil}} \tag{8.5}$$

O número de *bits* necessários à representação de um vetor candidato é, portanto:

$$\sum_{i=1}^{|asds.DDI|} \lceil\log_2(|asds.ddi_i.ds.VP|)\rceil \tag{8.6}$$

Note que o número de *bits* depende tanto do número de dimensões de projeto (e respectivos pontos de variação) do espaço de projeto em uso, quanto do número de elementos-alvo identificados no modelo arquitetural inicial em questão (número de instâncias de dimensão de projeto criadas). Visto que os índices dos pontos de variação variam de 1 a $|asds.ddi_i.ds.VP|$, a codificação armazenada no gene é sempre igual ao índice do ponto de variação -1.

Note também que o uso da função teto ($\lceil\,\rceil$) faz com que nem todo gene represente um índice válido de ponto de variação: uma dimensão com cinco pontos de variação usará um gene com três *bits* representando, portanto, valores de 0 a 7. A implementação do mecanismo de otimização deverá garantir que somente genes com valores de 0 a 4 sejam considerados como indivíduos válidos da população. Este procedimento deve ser realizado para cada gene que compõe o cromossomo. Adicionalmente, genes representando índices válidos de pontos de variação (daqui para frente, denominados genes válidos) podem ainda produzir arquiteturas inválidas, conforme descrito na Seção 6.2, pág. 111. Tais arquiteturas devem ser identificadas e descartadas da população, conforme descrito a seguir.

8.2.2. Operadores de Seleção, Recombinação e Mutação

Os passos realizados durante uma iteração de otimização de arquiteturas são semelhantes àqueles descritos na Figura 4.5, pág. 76. Esta seção descreve os requisitos para os operadores de seleção, recombinação e mutação utilizados na abordagem aqui proposta.

O operador de seleção tem como objetivo promover a duplicação de indivíduos com boa aptidão e a eliminação daqueles com aptidão ruim, ao mesmo tempo em que mantém constante o tamanho da população. Visto que novos indivíduos não são criados por este operador, nenhuma verificação de genes inválidos e arquiteturas inválidas precisa ser realizada. A implementação atual da abordagem aqui proposta – descrita no Capítulo 191 – utiliza o NSGA-II como mecanismo de otimização e, portanto, o operador de seleção por torneio aglomerado (vide Seção 4.3.2.2, pág. 80) é utilizado. O NSGA-II é apresentado no Algoritmo 4.6, pág. 82.

Os operadores de recombinação e mutação, por outro lado, criam novos indivíduos e, portanto, devem desconsiderar qualquer novo cromossomo que represente vetores candidatos associados a arquiteturas inválidas ou contendo genes que apresentem índices inválidos de pontos de variação. Para isso, novos operadores de recombinação e mutação foram desenvolvidos neste trabalho, descritos a seguir.

Algoritmo 8.1: Operador de recombinação com detecção de indivíduos *offspring* inválidos.

```
1:  procedure DuSECrossoverOper(basicCrossoverOper, p1, p2, asds): ⟨o1, o2⟩
2:                        ▷ basicCrossoverOper = operador básico de recombinação
3:                               ▷ p1 = primeira solução pai da recombinação
4:                                ▷ p2 = segunda solução pai da recombinação
5:                          ▷ asds = espaço de projeto específico de aplicação
6:                                  ▷ ⟨o1, o2⟩ = soluções offspring retornadas
7:      while true do
8:          ⟨o1, o2⟩ = basicCrossoverOper(p1, p2)
9:          i ← 0
10:         for all ddi ∈ asds.DDI do
11:             if o1.gene(i) ≥ |ddi.ds.VP| or o2.gene(i) ≥ |ddi.ds.VP| then
12:                 break
13:             i ← i + 1
14:         if i < |asds.DDI| then                 ▷ Algum gene inválido foi encontrado
15:             continue
16:         M_r1 ← generateArchitecture(asds, o1.chromosome())
17:         M_r2 ← generateArchitecture(asds, o2.chromosome())
18:         if M_r1 = invalidArchitecture or M_r2 = invalidArchitecture then
19:             continue
20:         evaluateIndividual(M_r1, asds.ds.M, o1)
21:         evaluateIndividual(M_r2, asds.ds.M, o2)
22:         return ⟨o1, o2⟩                        ▷ Retorna as soluções offspring
```

O novo operador de recombinação – denominado `DuSECrossoverOper` – encapsula qualquer operador básico de recombinação, tais como aqueles descritos na Seção 4.3.2.1, pág. 78. O novo operador executa continuamente o operador básico em uso até que os dois indivíduos *offspring* criados representem arquiteturas válidas do espaço de projeto em questão. Este procedimento é descrito no Algoritmo 8.1.

O operador `DuSECrossoverOper` recebe como parâmetros: o operador básico de recombinação encapsulado, as duas soluções pai ($p1$ e $p2$) e o espaço de projeto específico de aplicação sendo utilizado ($asds$). O operador aqui apresentado realiza sucessivas chamadas ao operador básico até que as duas soluções *offspring* obtidas contenham genes com índices válidos de pontos de variação e representem vetores candidatos que produzam arquiteturas válidas do espaço de projeto em questão. Para isso, as sentenças descritas nas linhas 10 a 13 verificam se cada gene i armazena um valor entre 0 e $|asds.ddi_{i+1}.ds.VP| - 1$. Caso algum índice inválido de ponto de variação seja encontrado o operador básico de recombinação é novamente executado e o procedimento de verificação é reiniciado (linhas 14 e 15).

Caso todos os genes contenham índices válidos de pontos de variação, executa-se o procedimento `generateArchitecture` – apresentado no Algoritmo 6.2, pág. 123 – para obtenção das arquiteturas associadas às duas soluções *offspring* geradas (linhas 16 e 17). Caso alguma das arquiteturas obtidas seja inválida, executa-se novamente o operador básico de recombinação e o procedimento de verificação é reiniciado (linhas 18 e 19).

Finalmente, caso as arquiteturas associadas às duas soluções *offspring* sejam válidas, realiza-se a avaliação das métricas de qualidade definidas em $asds.ds.M$ em cada uma das arquiteturas

resultantes, M_{r1} e M_{r2}, e armazena-se os valores obtidos nos objetos que representam os indivíduos – respectivamente, $o1$ e $o2$ (linhas 20 e 21). Após a avaliação, as arquiteturas M_{r1} e M_{r2} podem ser destruídas. Este procedimento é importante para reduzir o consumo de memória durante a otimização. Finalmente, o par de soluções *offspring* é retornado na linha 22.

O novo operador de mutação – denominado `DuSEMutationOper` – realiza uma abordagem similar ao apresentado para o novo operador de recombinação, encapsulando, desta vez, um operador básico de mutação, como os descritos na Seção 4.3.2.1, pág. 79. Note que os dois novos operadores aqui apresentados são utilizados na invocação do procedimento `createOffspring` (linha 7 do Algoritmo 4.6).

Os valores de avaliação das métricas, armazenados diretamente em cada indivíduo da população, são utilizados no procedimento `fastNDSort` (linha 9 do Algoritmo 4.6) e na ordenação via distância de aglomeração (linha 15 do Algoritmo 4.6), viabilizando a construção da próxima geração da população. Detalhes sobre os aspectos de implementação dos novos operadores, bem como as configurações necessárias para que o NSGA-II os utilize, são apresentados no Capítulo 9, pág. 175.

8.3. Premissas e Limitações

O mecanismo para otimização multiobjetivo de arquiteturas apresentado na seção anterior assume que as premissas descritas a seguir são satisfeitas.

P1 – O espaço de projeto específico de aplicação sendo utilizado possui alta cardinalidade. A adoção do mecanismo de otimização de arquiteturas, aqui apresentado, na automatização do processo de exploração de um espaço de projeto viabiliza a identificação de soluções (próximas de) ótimas mesmo na presença de um alto número de arquiteturas candidatas. Considerando o espaço de projeto para sistemas *self-adaptive* apresentado no Capítulo 7, mesmo modelos arquiteturais iniciais pequenos produzem espaços de projeto específicos de aplicação com centenas de milhares de soluções. Este número cresce exponencialmente de acordo com a quantidade de elementos-alvo detectados no modelo inicial. Dessa forma, espaços de projeto com baixa cardinalidade não demandam as técnicas aqui adotadas nem usufruem dos benefícios por elas introduzidos. Em tais situações, abordagens evolucionárias de otimização podem ter dificuldades em gerenciar o tamanho da população e garantir a preservação da diversidade.

P2 – As métricas do espaço de projeto são conflitantes. Embora a presença de métricas não conflitantes não inviabilize o uso da abordagem aqui proposta, todas as decisões relacionadas às técnicas de otimização adotadas foram baseadas na articulação *a posteriori* de preferência e, portanto, espera-se que o resultado da otimização seja um conjunto de arquiteturas ótimas (elementos do *Pareto-front* obtido). Assume-se que métricas conflitantes seja um cenário corriqueiro em um amplo conjunto de domínios de aplicação e, portanto, a infraestrutura de projeto arquitetural automatizado aqui apresentada seria diretamente aplicável nestes novos casos.

P3 – As dimensões de projeto possuem alta cardinalidade. O mecanismo de otimização baseado em algoritmos evolucionários aqui adotado mantém, a cada geração, uma população formada por um subconjunto das inúmeras arquiteturas que compõem o espaço de projeto. A formalização dos espaços de projeto e os algoritmos para projeto automatizado de arquiteturas apresentados no Capítulo 6 definem mecanismos para identificação de arquiteturas inválidas, consequências de combinações conflitantes de pontos de variação. Assume-se que, embora o número de arquiteturas inválidas seja alto em alguns domínios de aplicação, o número total de

arquiteturas válidas ainda justifica o uso de uma abordagem baseada em busca. Adicionalmente, identificar uma arquitetura como inválida requer a aplicação, no modelo inicial, das transformações identificadas pelo vetor candidato. Dessa forma, é inviável a identificação *a priori* do subconjunto formado apenas pelas arquiteturas válidas, de modo a realizar a busca apenas nesta versão reduzida do espaço de projeto.

P4 – As métricas definem valores em uma escala no mínimo intervalar. Conforme descrito anteriormente, embora a abordagem aqui proposta não apresente restrições quanto à técnica utilizada na avaliação da qualidade de uma arquitetura candidata, os valores por ela disponibilizados devem ser passíveis de ordem total e da estimativa de distância entre valores. Esta premissa é fundamental para a realização – pelo NSGA-II – da ordenação por dominância e do cálculo da distância de aglomeração.

As limitações identificadas para a abordagem de otimização de arquiteturas aqui apresentadas são descritas a seguir.

L1 – Não há garantia de obtenção de um Pareto-front global. Conforme mencionado no Capítulo 4, embora o uso de metaheurísticas não garanta a obtenção de soluções pertencentes ao *Pareto-front* global, acredita-se que a utilização de algoritmos com alta convergência e alta diversidade produza soluções ainda assim efetivas e que evidenciam com expressividade o *trade-off* em questão. Embora apresente limitações quanto ao número de funções-objetivo utilizadas (vide Seção 4.3.2.2, pág. 80), o NSGA-II é um algoritmo com desempenho de convergência amplamente reconhecido na literatura [329].

L2 – Ausência do uso de táticas de domínio específico. A natureza "independente de domínio" apresentada pelas abordagens de otimização baseadas em metaheurísticas é fundamental para o uso facilitado de tais mecanismos em uma ampla faixa de problemas. Entretanto, associar metaheurísticas ao uso de táticas de domínio específico pode aumentar consideravelmente o tempo de convergência da otimização. Tais táticas utilizam conhecimento específico do domínio para guiar as operações de seleção, recombinação e mutação, de modo a produzir – com um número menor de gerações – indivíduos com melhores características. Estes aspectos não foram investigados no trabalho aqui apresentado.

8.4. Resumo do Capítulo

Este capítulo discutiu como os fundamentos de otimização multiobjetivo apresentados no Capítulo 4 foram utilizados para automatizar a exploração de um espaço de projeto específico de aplicação. A busca por arquiteturas efetivas em tais espaços de projeto foi formalmente apresentada como um problema de otimização multiobjetivo, com base nos constructos propostos no Capítulo 6. Argumentou-se que o problema de otimização de arquiteturas apresenta quatro propriedades: *i)* é um problema de otimização multiobjetivo; *ii)* possui variáveis de decisão discretas; *iii)* o espaço de busca é finito, porém potencialmente grande; e *iv)* não adota premissas sobre as funções-objetivo. Tais características justificam a decisão pelo uso de técnicas evolucionárias para otimização multiobjetivo.

Em seguida, discutiu-se como os vetores candidatos – representantes diretos das inúmeras arquiteturas capturadas em um espaço de projeto – foram codificados como uma sequência binária de genes (cromossomo). O cálculo do número de *bits* necessários à representação de um vetor candidato – e as implicações nos operadores de recombinação e mutação – foram também

apresentados. A detecção de genes inválidos e cromossomos representando arquiteturas inválidas demandou a criação de novos operadores de recombinação e mutação. Tais operadores, encapsulam operadores convencionais de recombinação e mutação, porém realizam o descarte de qualquer indivíduo com gene inválido ou cromossomo associado a uma arquitetura inválida.

Capítulo 9

Suporte por Ferramenta

Skill in the digital age is confused with mastery of digital tools ...
John Maeda

Este capítulo apresenta os aspectos de projeto e implementação do DuSE-MT – ferramenta de apoio ao processo de projeto arquitetural automatizado proposto neste trabalho. A Seção 9.1 discute os principais requisitos funcionais e não-funcionais da ferramenta, bem como as principais diretrizes que guiaram a fase de projeto arquitetural da aplicação. A Seção 9.2 apresenta os módulos para manipulação de modelos e otimização multiobjetivo implementados neste trabalho, bem como uma discussão sobre como a arquitetura projetada para a ferramenta viabiliza o uso futuro de novos metamodelos, novos espaços de projeto e novos algoritmos de otimização multiobjetivo. A Figura 9.1 apresenta o roteiro deste capítulo.

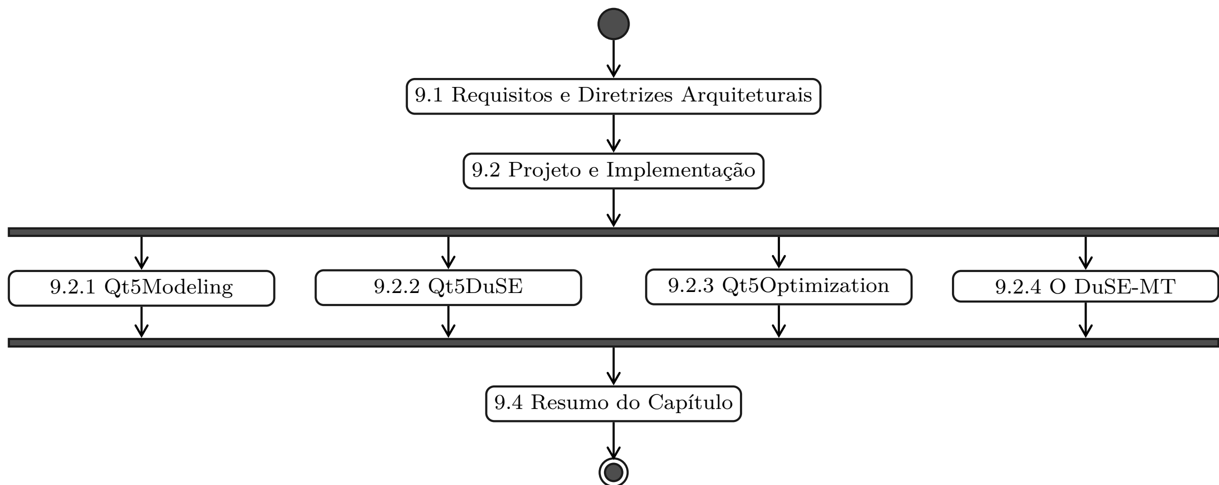


Figura 9.1.: Roteiro do capítulo 9.

9.1. Requisitos e Diretrizes Arquiteturais

A ferramenta DuSE-MT, implementada como parte deste trabalho, suporta as principais atividades do processo automatizado de projeto arquitetural aqui apresentado. Os seguintes requisitos funcionais foram definidos para a ferramenta:

- **RF1:** a ferramenta deve permitir a criação de novos espaços de projeto para domínios de aplicação específicos, descritos na linguagem de modelagem DuSE (proposta neste trabalho e descrita na Seção 9.2.1).
- **RF2:** a ferramenta deve implementar a derivação do espaço de projeto específico de aplicação (descrito no Algoritmo 6.1, pág. 122) a partir de um modelo arquitetural inicial e um espaço de projeto descrito em DuSE, ambos fornecidos como entrada para o processo automatizado de projeto arquitetural.
- **RF3:** a ferramenta deve permitir a navegação manual no espaço de projeto específico de aplicação, através da seleção dos pontos de variação a serem adotados em cada instância de dimensão de projeto (definição manual do vetor candidato). A arquitetura resultante (ou uma indicação de arquitetura inválida) deve ser apresentada ao usuário. Os valores das métricas, apresentados pela arquitetura resultante, também devem ser informadas ao usuário.
- **RF4:** a ferramenta deve permitir a exploração automática do espaço de projeto específico de aplicação, através do uso de algum algoritmo evolucionário para otimização multiobjetivo. O usuário deve ser capaz de ajustar os parâmetros convencionais da otimização (tamanho da população, operadores utilizados, probabilidade de recombinação, probabilidade de mutação e número máximo de gerações). O *Pareto-front* resultante deve ser apresentado ao usuário, permitindo a seleção de uma arquitetura particular do conjunto e a visualização dos valores de métricas associados à arquitetura selecionada.

Os seguintes requisitos não-funcionais foram definidos para a ferramenta e serviram de guia para as decisões tomadas em relação à sua arquitetura e implementação:

- **RNF1:** a ferramenta deve ser multiplataforma, podendo ser executada nos principais sistemas operacionais em uso atualmente.
- **RNF2:** a ferramenta deve ser independente de metamodelo. Todas as operações de criação e manipulação de modelos realizadas na ferramenta devem poder suportar, futuramente, novos metamodelos. Nenhuma modificação deverá precisar ser realizada, no núcleo da ferramenta, de modo a suportar uma nova linguagem de modelagem.
- **RNF3:** a ferramenta deve apresentar bom desempenho na manipulação de modelos grandes e na utilização simultânea de diversos modelos. O atendimento deste requisito é fundamental para o uso de técnicas evolucionárias de otimização visto que múltiplos modelos serão manipulados simultaneamente.
- **RNF4:** a ferramenta deve estar em conformidade com as especificações dos metamodelos MOF e UML propostos pelo *Object Management Group* (OMG). Espera-se que profissionais com conhecimento prévio de uso destes metamodelos tenham acesso direto e uniforme às metaclasses e propriedades disponibilizadas pela implementação da MOF e da UML aqui realizadas.
- **RNF5:** o mecanismo de otimização de arquiteturas implementado na ferramenta deve ser independente do algoritmo de otimização utilizado. Novas abordagens de otimização devem poder ser facilmente incluídas no futuro, sem requerer modificações no núcleo da ferramenta. Assume-se, entretanto, que a API projetada contemple somente a classe de algoritmos de otimização baseados em técnicas evolucionárias.
- **RNF6:** a evolução da ferramenta deve ser facilitada, de modo que novas funcionalidades sejam encapsuladas em unidades individuais de compilação e implantação. Inversão de controle é desejável e novos recursos devem poder ser incluídos com base no conhecimento de uma API mínima de extensão a ser implementada.

Um conjunto de estratégias arquiteturais e tecnologias, consideradas adequadas para o atendimento dos requisitos acima descritos, foram adotadas no projeto e implementação do DuSE-MT.

Tais estratégias e tecnologias são rapidamente apresentadas a seguir e discutidas em detalhes nas seções seguintes.

Primeiro, bom desempenho e execução em múltiplas plataformas foram viabilizados pelo uso da linguagem C++ e do *toolkit* Qt [66] como tecnologias de implementação. O Qt é um projeto de *software* livre desenvolvido desde 1995 e amplamente utilizado em aplicações *desktop*, aplicativos de dispositivos móveis e sistemas embarcados em geral. O *toolkit* disponibiliza um extenso conjunto de bibliotecas e *application frameworks* para a implementação de funcionalidades relacionadas a interfaces gráficas, banco de dados, OpenGL, comunicação entre processos, sistemas distribuídos e multimídia, somente para citar algumas. O Qt foi originalmente concebido para ser uma solução multiplataforma e atualmente suporta o desenvolvimento de aplicações para Windows, Linux/X11, Mac OS X e para um conjunto de plataformas embarcadas e de dispositivos móveis, tais como o Android e iOS.

Segundo, a independência de metamodelo foi alcançada através de uso intenso dos mecanismos de reflexão computacional disponibilizados pelo Qt e do projeto de uma arquitetura que suporte fábricas (*factories*) de instâncias de metaclasses como *plugins*. Esta decisão permite a extensão facilitada da ferramenta, ao suportar o uso de novas linguagens de modelagem sem requerer nenhuma alteração no núcleo arquitetural do DuSE-MT. A extensibilidade da ferramenta foi potencializada através do uso de uma arquitetura baseada em *microkernel* [54] (Capítulo 9, pág. 194), inversão de controle e *plugins*.

Terceiro, conformidade com as especificações do OMG foi obtida através da geração automática da maior parte do código C++/Qt que implementa os metamodelos MOF e UML. Os arquivos XMI (*XML Metadata Interchange* [119]) normativos dos metamodelos MOF e UML, disponibilizados no *website* do OMG, foram utilizados como entrada do processo de geração automática de código.

Finalmente, mas não menos importante, um *application framework* para resolução de problemas de otimização multiobjetivo foi desenvolvido como parte desta tese. O uso de inversão de controle, associado à adoção de padrões de projeto como o *Strategy* e *Bridge* [104], permitiu a configuração – em tempo de execução – do algoritmo de otimização e operadores de seleção, recombinação e mutação utilizados. A seção a seguir detalha as decisões e tecnologias apresentadas nesta seção.

9.2. Projeto e Implementação

A Figura 9.2 apresenta a visão arquitetural originada do estilo de uso de módulos (*module uses style*) – conforme descrito em [105], Capítulo 2, pág. 64 – para os artefatos de programação desenvolvidos neste trabalho. Os artefatos apresentados na figura podem ser classificados em cinco grupos:

- Módulos pré-disponibilizados pelo Qt: `Qt5Core`, `Qt5Gui` e `Qt5Widgets` são três dos quinze módulos essenciais (de uso mais frequente) oferecidos pelo Qt. O módulo `Qt5Core` disponibiliza funcionalidades básicas, não relacionadas a interfaces gráficas de usuário e utilizadas pelos demais módulos. Tais funcionalidades incluem: mecanismo para comunicação desacoplada entre objetos (via sinais e *slots*) [252], suporte à avaliação dinâmica de propriedades de objetos [251], suporte a hierarquia de objetos (padrão de projeto *Composite*) [249], um amplo conjunto de *smart pointers* [250] e um poderoso mecanismo para reflexão computacional [247]. Tais recursos foram fundamentais para o atendimento de alguns dos requisitos acima apresentados. Mais informações podem ser encontradas nas referências acima citadas.

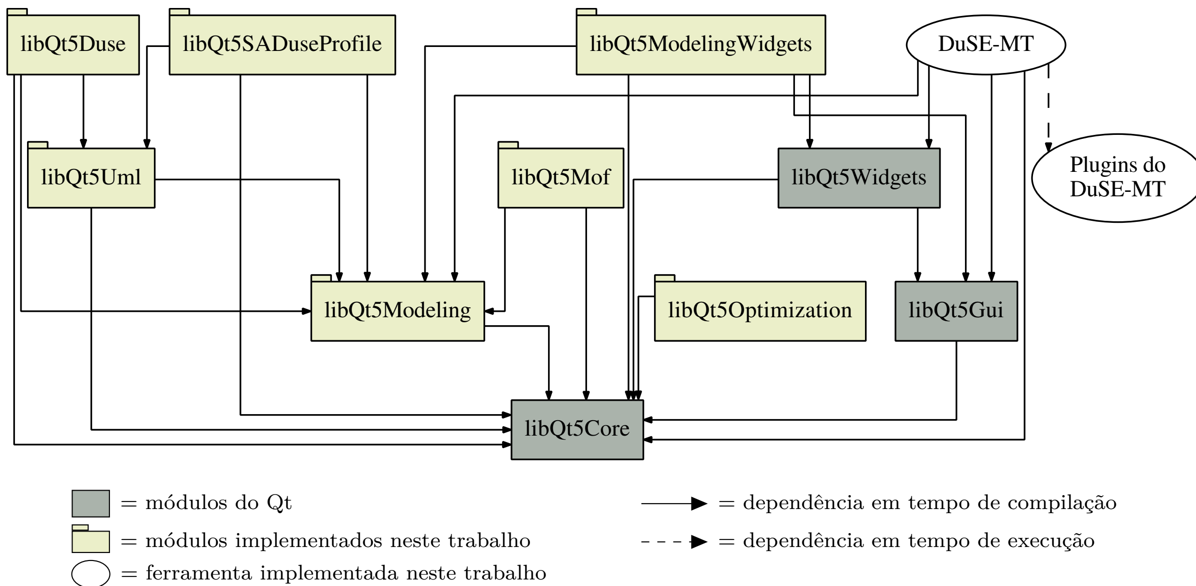


Figura 9.2.: Visão originada do estilo de uso de módulos (*module uses style*) para os artefatos utilizados neste trabalho.

- Módulos de implementação da infraestrutura de metamodelagem: a infraestrutura para manipulação de modelos de forma independente de linguagem de modelagem é implementada nos módulos `Qt5Modeling` e `Qt5ModelingWidgets`. Os módulos `Qt5Mof` e `Qt5Uml` implementam, respectivamente, as linguagens de modelagem MOF e UML, conforme especificadas pelo OMG. Os módulos deste grupo são descritos na Seção 9.2.1.

- Módulos de implementação da infraestrutura de espaços de projeto: o módulo `Qt5Duse` implementa a linguagem de modelagem para especificação de espaços de projeto proposta neste trabalho de modo a suportar a formalização apresentada no Capítulo 6. Com isso, o SA:DuSE – espaço de projeto para sistemas *self-adaptive* aqui proposto – pode ser diretamente descrito como uma instância do metamodelo implementado no módulo `Qt5Duse`. Espaços de projeto para outros domínios de aplicação podem futuramente ser criados de forma semelhante. O módulo `Qt5SADuseProfile` implementa o *profile* UML de suporte ao SA:DuSE. Os módulos deste grupo são descritos na Seção 9.2.2.

- Módulos de implementação da infraestrutura de otimização: o `Qt5Optimization` é um módulo que implementa um *application framework* para resolução de problemas de otimização multiobjetivo. O *framework* não está restrito à otimização de arquiteturas de *software* e seu projeto favorece a inclusão facilitada de novos operadores de seleção, recombinação e mutação, bem como novos algoritmos de otimização. O módulo `Qt5Optimization` é descrito na Seção 9.2.3.

- A ferramenta DuSE-MT e seu conjunto de *plugins*: a arquitetura construída para o DuSE-MT viabiliza que, no futuro, novas funcionalidades relacionadas ao uso de modelos de *software* sejam facilmente incluídas. Um grupo de *plugins*, desenvolvido como parte deste trabalho, integra as funcionalidades providas pelos módulos anteriores, de modo a suportar o processo de projeto arquitetural proposto nesta tese. Tais aspectos são discutidos na Seção 9.2.4.

Todos os módulos acima, com exceção daqueles pré-disponibilizados pelo Qt, foram desenvolvidos como parte do trabalho aqui apresentado.

9.2.1. O Qt5Modeling e o Qt5ModelingWidgets

Os módulos `Qt5Modeling` e `Qt5ModelingWidgets` têm como objetivo suportar – de forma independente de metamodelo – as operações fundamentais de definição e uso de linguagens de modelagem. O `Qt5Modeling` amplia os recursos de reflexão computacional da classe `QObject` (disponibilizada pelo módulo `Qt5Core`) para suportar funcionalidades não oferecidas pelo Qt, tais como: propriedades opostas, propriedades do tipo união derivada e herança múltipla envolvendo a classe `QObject`.

Adicionalmente, a infraestrutura para uso de metamodelos como *plugins* e a serialização (leitura e gravação em disco) de modelos no formato XMI são também implementados no módulo `Qt5Modeling`. A Figura 9.3 apresenta as principais classes do `Qt5Modeling` e seus relacionamentos com as classes do `Qt5Core`. As classes `QModelingElement` e `QModelingObject` têm papel fundamental na implementação dos mecanismos de metamodelagem utilizados neste trabalho.

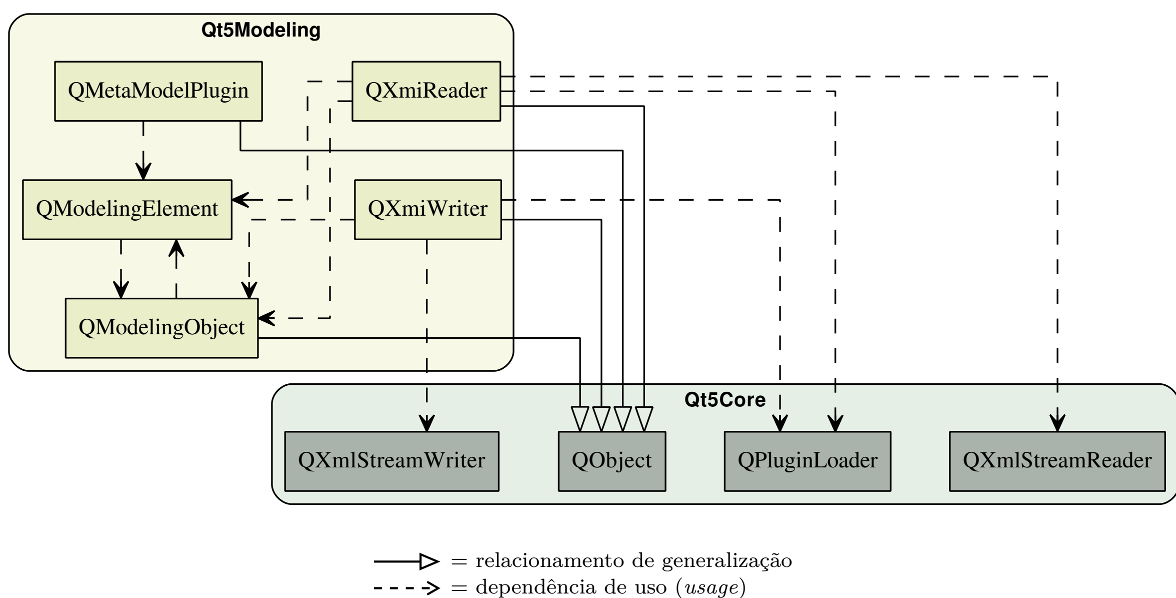


Figura 9.3.: Visão originada do estilo de decomposição (*module decomposition style*) para o módulo `Qt5Modeling`.

`QModelingElement` é a classe-base de todas as metaclasses implementadas para um metamodelo, pois disponibiliza o mecanismo de clonagem de elementos (fundamental para duplicar a arquitetura inicial de modo a aplicar as extensões providas pelos pontos de variação). Para que uma determinada classe faça uso dos recursos de reflexão computacional do Qt, tal classe deve ser derivada (direta ou indiretamente) da classe `QObject`, disponibilizada pelo módulo `Qt5Core`. Entretanto, classes derivadas de `QObject` não podem estar envolvidas em heranças múltiplas nem fazer uso de herança virtual para resolver o chamado “problema do diamante” [208] (amplamente presente nas especificações do OMG).

Para solucionar o problema acima, a classe `QModelingObject`, desenvolvida neste trabalho, encapsula um `QModelingElement` e atua como um *wrapper* que provê os recursos de reflexão computacional necessários ao atendimento do requisito de independência de metamodelo apresentado anteriormente (RNF2). Operações para obtenção de um `QModelingObject` a partir de um `QModelingElement` e vice-versa foram implementados.

As classes `QXmiWriter` e `QXmiReader` realizam, respectivamente, a gravação e leitura de modelos no formato XMI. As classes `QXmlStreamWriter` e `QXmlStreamReader`, presentes no módulo `Qt5Core`, foram utilizadas para este propósito. Estas operações de serialização de modelos são realizadas de forma independente de metamodelo, utilizando uma arquitetura de *plugins*

conforme descrito a seguir. O exemplo abaixo apresenta o arquivo XMI gerado para um modelo simples, descrito em UML. Todo modelo é representado como uma hierarquia de instâncias de metaclasses (instâncias de subtipos específicos de `QModelingObject`).

```
01. <?xml version="1.0" encoding="UTF-8"?>
02. <xmi:XMI xmlns:xmi="http://www.omg.org/spec/XMI/20110701"
03.              xmlns:uml="http://www.omg.org/spec/UML/20110701">
04.   <uml:Package name="MyRootPackage" xmi:id="MyRootPackage">
05.     <packagedElement xmi:type="uml:Class" name="Professor" xmi:id="Professor">
06.       <ownedAttribute xmi:type="uml:Property" name="name" xmi:id="name">
07.         <type href=":/metamodels/PrimitiveTypes.xmi#String"/>
08.       </ownedAttribute>
09.     </packagedElement>
10.   </uml:Package>
11. </xmi:XMI>
```

A classe `QMetaModelPlugin` define a API das fábricas de instâncias de metaclasses. Este recurso é fundamental para a implementação das demais operações de forma independente do metamodelo (linguagem de modelagem) utilizado. Um *plugin* de metamodelo implementa uma API simples, formada por um único *factory method*:

```
virtual QModelingElement *createModelingElement(QString type) = 0;
```

Este *factory method* é responsável por criar e retornar uma nova instância cujo tipo é descrito no parâmetro *type*. Todas as metaclasses concretas do metamodelo devem poder ser instanciadas pelo *plugin*. Um ponteiro nulo é retornado caso o metamodelo não conheça o tipo informado. Adicionalmente, cada *plugin* de metamodelo define metadados que auxiliam o uso correto do *plugin*. Por exemplo, para o *plugin* do metamodelo UML tais metadados são:

```
{
  “Version”: “2.4.1”,
  “Vendor”: “Qt Project”,
  “MetaModelNamespaceUri”: “http://www.omg.org/spec/UML/20110701”,
  ”MetaModelPrefix”: “QUml”,
  “MetaModelTopLevelTypes”: [ “QUmlModel”, “QUmlPackage”, “QUmlProfile” ]
}
```

Dos metadados acima, `MetaModelNamespaceUri` é utilizado para identificar qual *plugin* de metamodelo deve ser utilizado para instanciar um elemento encontrado em um arquivo XMI. A classe `QXmiReader` – um dos clientes das fábricas de instâncias de metamodelos – mantém uma tabela *hash* contendo, como chave, o valor do metadado `MetaModelNamespaceUri` e, como valor, um ponteiro para o *plugin* em questão. Todos os *plugins* de metamodelos instalados no sistema são mantidos nesta tabela *hash*. Por exemplo, se os *plugins* dos metamodelos MOF e UML estiverem instalados os dados presentes na tabela *hash* são:

Chave	**Valor**
http://www.omg.org/spec/MOF/20110701	<ponteiro para o *plugin* do metamodelo MOF>
http://www.omg.org/spec/UML/20110701	<ponteiro para o *plugin* do metamodelo UML>

Dessa forma, para instanciar o elemento `uml:Package` apresentado na linha 4 do XMI acima, os seguintes passos são realizados: *i)* obtém-se o *namespace* XML utilizado (neste caso ’`uml`’); *ii)* encontra-se o *namespace* URI associado ao *namespace* XML (presente na linha 3 do XMI, neste

caso 'http://www.omg.org/spec/UML/20110701'); *iii)* consulta-se a tabela *hash* para obter o *plugin* associado ao *namespace* URI encontrado; e *iv)* utiliza-se o *factory method* do *plugin* para instanciar o elemento em questão. A *string* informada no parâmetro *type* do *factory method* é a concatenação do metadado MetaModelPrefix com o elemento presente no XMI, neste caso "QUml"+"Package". Esta infraestrutura permite a adição futura de novos metamodelos sem demandar nenhuma modificação nas classes do Qt5Modeling.

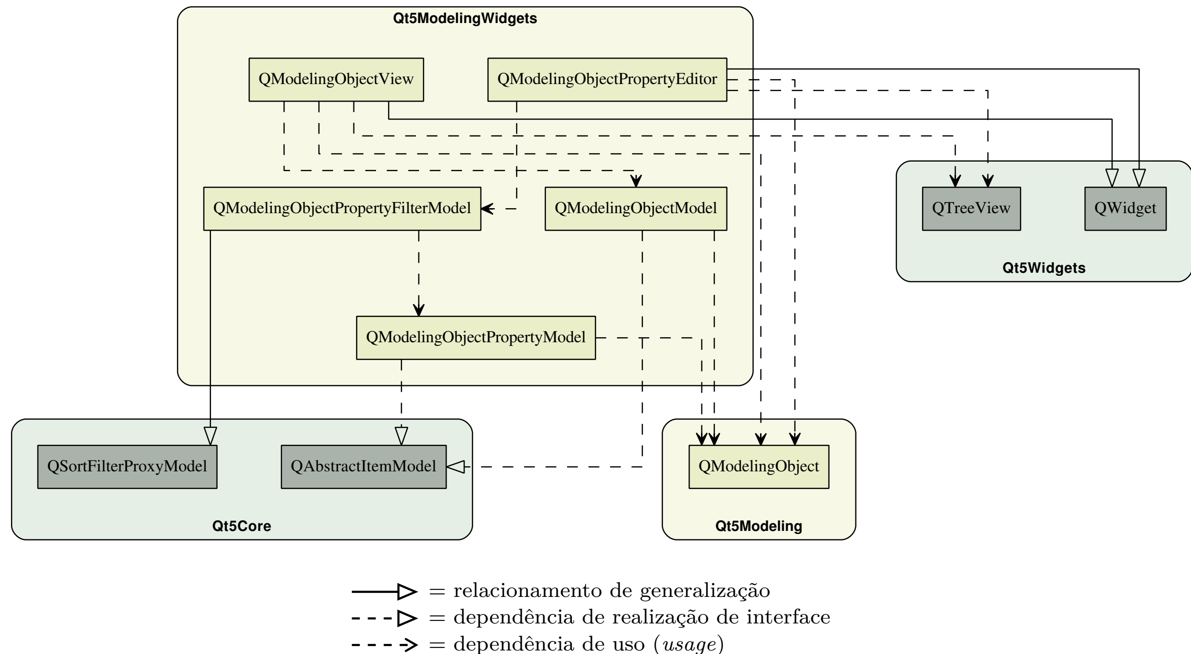


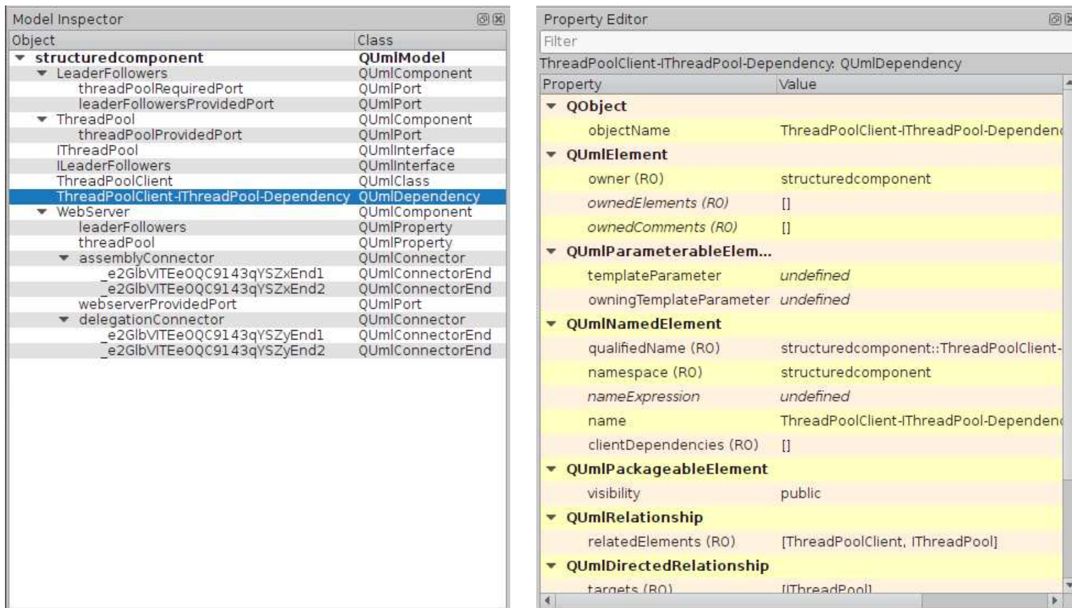
Figura 9.4.: Visão originada do estilo de decomposição (*module decomposition style*) para o módulo Qt5ModelingWidgets.

As classes presentes no módulo Qt5Modeling não apresentam recursos para interfaces gráficas de usuário e, portanto, podem ser utilizadas na construção de aplicações console de manipulação de modelos. Recursos para manipulação de modelos via interfaces gráficas de usuário são disponibilizados pelo módulo Qt5ModelingWidgets, detalhado na Figura 9.4. Esta separação é importante para minimizar o número de dependências entre módulos e facilitar a evolução independente dos artefatos.

O Qt5ModelingWidgets disponibiliza classes de interface gráfica de usuário que atuam como novos modelos e visões[1], integrados ao *framework* de *model-view* [248] do Qt. As classes QModelingObjectModel e QModelingObjectPropertyModel implementam novos modelos que representam, respectivamente, o conteúdo de um arquivo XMI (hierarquia de QModelingObjects) e as propriedades de um QModelingObject em particular. Um mecanismo para filtragem de propriedades é implementado na classe QModelingObjectPropertyFilterModel.

As classes QModelingObjectView e QModelingObjectPropertyEditor implementam as visões que fazem uso dos modelos acima citados. Conforme apresentado na Figura 9.4, os modelos e visões do Qt5ModelingWidgets não possuem dependência com nenhuma implementação de um metamodelo em particular. Todas as operações foram implementadas utilizando apenas a classe QModelingObject, as fábricas disponibilizadas pelos *plugins* e os mecanismos de reflexão computacional do Qt. A Figura 9.5 ilustra os *widgets* disponibilizados pelo módulo Qt5ModelingWidgets.

[1] Não confundir com modelos arquiteturais e visões arquiteturais. Os conceitos de modelos e visões implementados nestas classes são aqueles descritos no padrão arquitetural *Model-View-Controller* [101] (Capítulo 14, pág. 330).



(a) `QModelingObjectView` utilizando o `QModelingObjectModel` para visualizar o conteúdo de um arquivo XMI.

(b) `QModelingObjectPropertyEditor` utilizando o `QModelingObjectPropertyModel` para visualizar as propriedades do `QModelingObject` selecionado na figura ao lado.

Figura 9.5.: *Widgets* implementados no módulo `Qt5ModelingWidgets`.

A classe `QModelingObjectView` visualiza o conteúdo de um `QModelingObjectModel` que, por sua vez, é configurado com o `QModelingObject` que representa o elemento-raiz da hierarquia a ser visualizada. Sempre que um `QModelingObject` particular é selecionado na hierarquia exibida no `QModelingObjectView`, um sinal é emitido para que o `QModelingObjectPropertyEditor` apresente as propriedades referentes a este objeto. Novamente, modelos descritos em qualquer metamodelo podem ser visualizados e ter propriedades de seus elementos visualizadas e alteradas nos *widgets* disponibilizados pelo módulo `Qt5ModelingWidgets`.

Três metamodelos foram implementados como parte deste trabalho: MOF (módulo `Qt5Mof`), UML (módulo `Qt5Uml`) e DuSE (módulo `Qt5Duse`, apresentado na Seção 9.2.2). Embora a UML, em particular, apresente um metamodelo consideravelmente grande (239 metaclasses, sendo destas 193 metaclasses concretas), 91.83% do código-fonte que implementa o metamodelo pôde ser automaticamente gerado. Os outros 8.17% do código-fonte correspondem a propriedades derivadas, operações e restrições, requerendo implementação manual.

Um conjunto de *scripts* – desenvolvidos neste trabalho – utilizam sentenças *XPath* [163] e o *Perl Template Toolkit* [60] para realizar a leitura dos metamodelos (arquivos XMI disponibilizados pelo OMG) e a geração subsequente do código C++/Qt que os implementa. As partes que requerem implementação manual não estão completamente codificados. Esta atividade foi realizada sob demanda, conforme necessário para a realização da abordagem de projeto arquitetural aqui proposta.

O `Qt5Modeling` é desenvolvido de forma "*upstream*": toda a infraestrutura de repositórios, revisões pré-commit de código, integração contínua e gerenciamento de *releases* da comunidade Qt é utilizada. O objetivo é disponibilizar o `Qt5Modeling` como um dos módulos oficiais do Qt, assim que atingir a maturidade adequada. O código-fonte do projeto está disponível em `http://wiki.ifba.edu.br/qtmodeling`.

9.2.2. O Qt5Duse e o Qt5SADuseProfile

A DuSE é uma linguagem de modelagem de domínio específico (DSL) – criada como parte deste trabalho – que viabiliza a criação de modelos de espaços de projeto. Tais modelos capturam as soluções arquiteturais mais prevalentes em um determinado domínio de aplicação. A linguagem DuSE disponibiliza os constructos apresentados no Capítulo 6 (dimensões de projeto, pontos de variação, modificações arquiteturais, etc) e permite que o mecanismo de otimização de arquiteturas apresentado no Capítulo 8 possa ser utilizado em qualquer espaço de projeto descrito na linguagem DuSE.

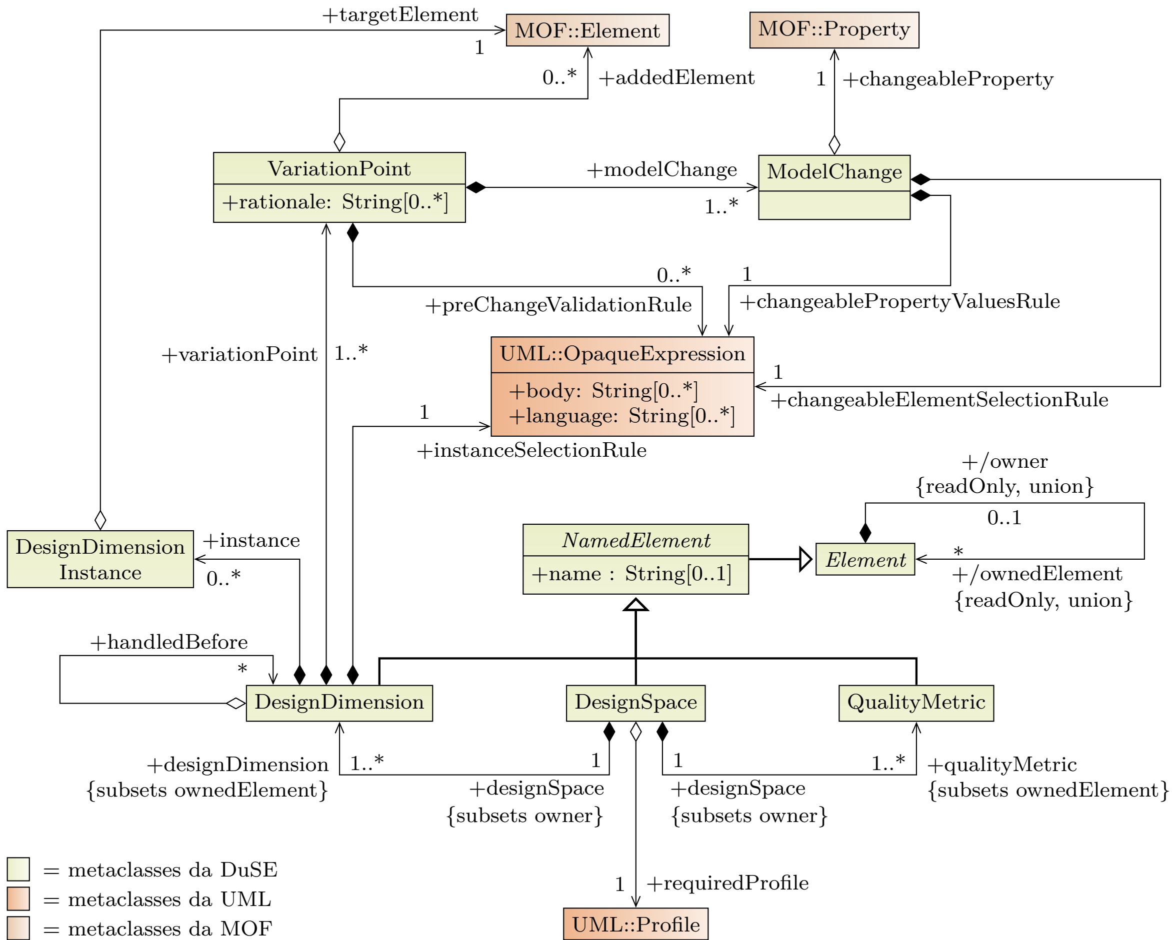


Figura 9.6.: Metamodelo da linguagem DuSE para especificação de espaços de projeto para domínios específicos.

A Figura 9.6 apresenta o metamodelo da linguagem DuSE. As metaclasses apresentadas na figura implementam os constructos apresentados no Capítulo 6 e foram codificadas em um arquivo XMI, de forma semelhante ao realizado, pelo OMG, para as linguagens MOF e UML. O mesmo mecanismo de geração automática de código, apresentado na Seção 9.2.1, foi utilizado para produzir o código-fonte C++/Qt que implementa a linguagem DuSE (módulo `Qt5Duse`). Dessa forma, qualquer espaço de projeto de domínio específico (como o espaço de projeto para sistemas *self-adaptive* – SA:DuSE – apresentado no Capítulo 7) pode ser descrito em um arquivo XMI, utilizando como metamodelo a implementação presente no módulo `Qt5Duse`.

Em função da independência de metamodelo proporcionada pelos módulos `Qt5Modeling` e `Qt5ModelingWidgets`, modelos representando espaços de projeto de domínios específicos podem ser diretamente criados, visualizados e modificados com a utilização das classes apresentadas na Seção 9.2.1. Conforme mencionado anteriormente, o espaço de projeto SA:DuSE é descrito em um arquivo XMI. Não há necessidade de geração de código-fonte para o SA:DuSE pois ele não

atua como metamodelo de nenhum modelo.

O *profile* UML que acompanha o espaço de projeto SA:DuSE (apresentado no Apêndice A) foi descrito em um arquivo XMI e os mesmos mecanismos apresentados anteriormente para geração de código foram utilizados para implementar o *profile* (módulo `Qt5SADuseProfile`). O código-fonte do `Qt5Duse` e `Qt5SADuseProfile` é mantido no mesmo repositório do `Qt5Modeling` (`http://wiki.ifba.edu.br/qtmodeling`).

9.2.3. O Qt5Optimization

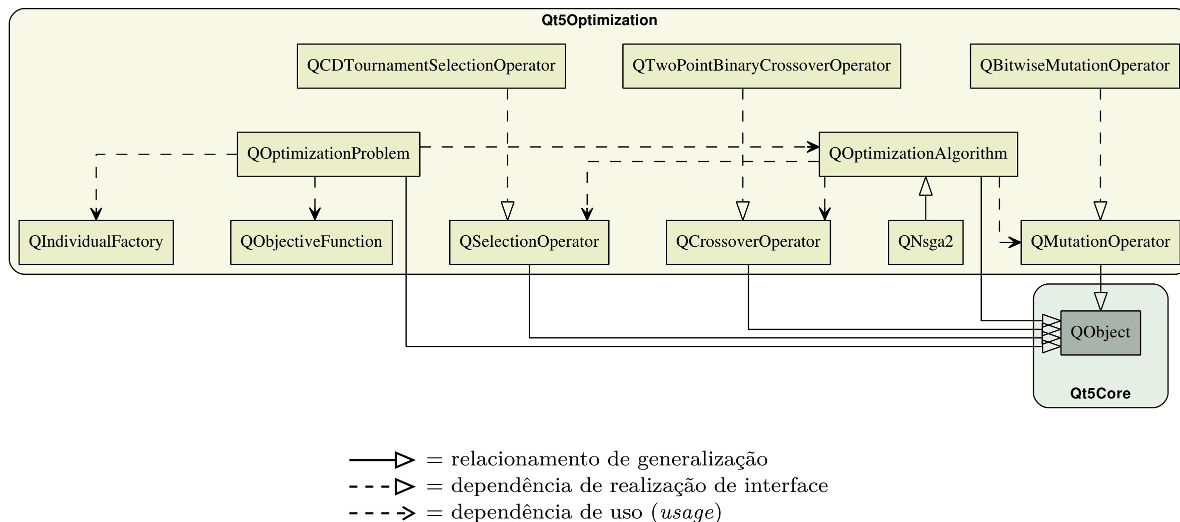


Figura 9.7.: Visão originada do estilo de decomposição (*module decomposition style*) para o módulo `Qt5Optimization`.

O módulo `Qt5Optimization`, desenvolvido neste trabalho, disponibiliza um *application framework* para resolução de problemas de otimização multiobjetivo. A Figura 9.7 apresenta as principais classes do `Qt5Optimization`. O elemento central é a classe `QOptimizationProblem`, responsável pela realização da inversão de controle e definição dos *hot-spots* (pontos de variabilidade) que permitem a especialização do *framework* a um problema de otimização particular. Uma instância de `QOptimizationProblem` é configurada com a fábrica de indivíduos a ser utilizada (instância de algum subtipo de `QIndividualFactory`), um conjunto de funções-objetivo (instâncias de subtipos de `QObjectiveFunction`) e um algoritmo (*backend*) particular de otimização (instância de algum subtipo de `QOptimizationAlgorithm`).

O algoritmo de otimização, por sua vez, é configurado com os operadores de seleção, recombinação e mutação desejados. Tais operadores são instâncias de subtipos de, respectivamente, `QSelectionOperator`, `QCrossoverOperator` e `QMutationOperator`. Atualmente, o *framework* disponibiliza implementações do algoritmo NSGA-II (classe `QNsga2`), e dos operadores de mutação bit-a-bit (classe `QBitwiseMutationOperator`), recombinação binária com dois pontos (classe `QTwoPointBinaryCrossoverOperator`) e seleção por torneio aglomerado (classe `QCDTournamentSelectionOperator`), dentre outros não apresentados na Figura 9.7.

O módulo Qt5Optimization não disponibiliza classes para problemas particulares de otimização multiobjetivo. Os mecanismos para busca por arquiteturas ótimas em espaços de projeto (apresentados no Capítulo 8) foram implementados como *plugins* da ferramenta DuSE-MT, conforme discutido na Seção 9.2.4. O código-fonte do `Qt5Optimization` está disponível em `http://wiki.ifba.edu.br/qtoptimization`.

9.2.4. A Ferramenta DuSE-MT

Os módulos apresentados nas seções anteriores implementam as funcionalidades de metamodelagem, espaços de projeto e otimização multiobjetivo que fundamentam a abordagem apresentada nesta tese. O DuSE-MT – ferramenta que suporta o projeto automatizado de projeto arquitetural aqui proposto – integra as funcionalidades anteriormente apresentadas. A Figura 9.8 apresenta a arquitetura da ferramenta e seu relacionamento com os módulos anteriormente apresentados. Uma visão geral da interface gráfica de usuário do DuSE-MT é apresentada na Figura 9.9.

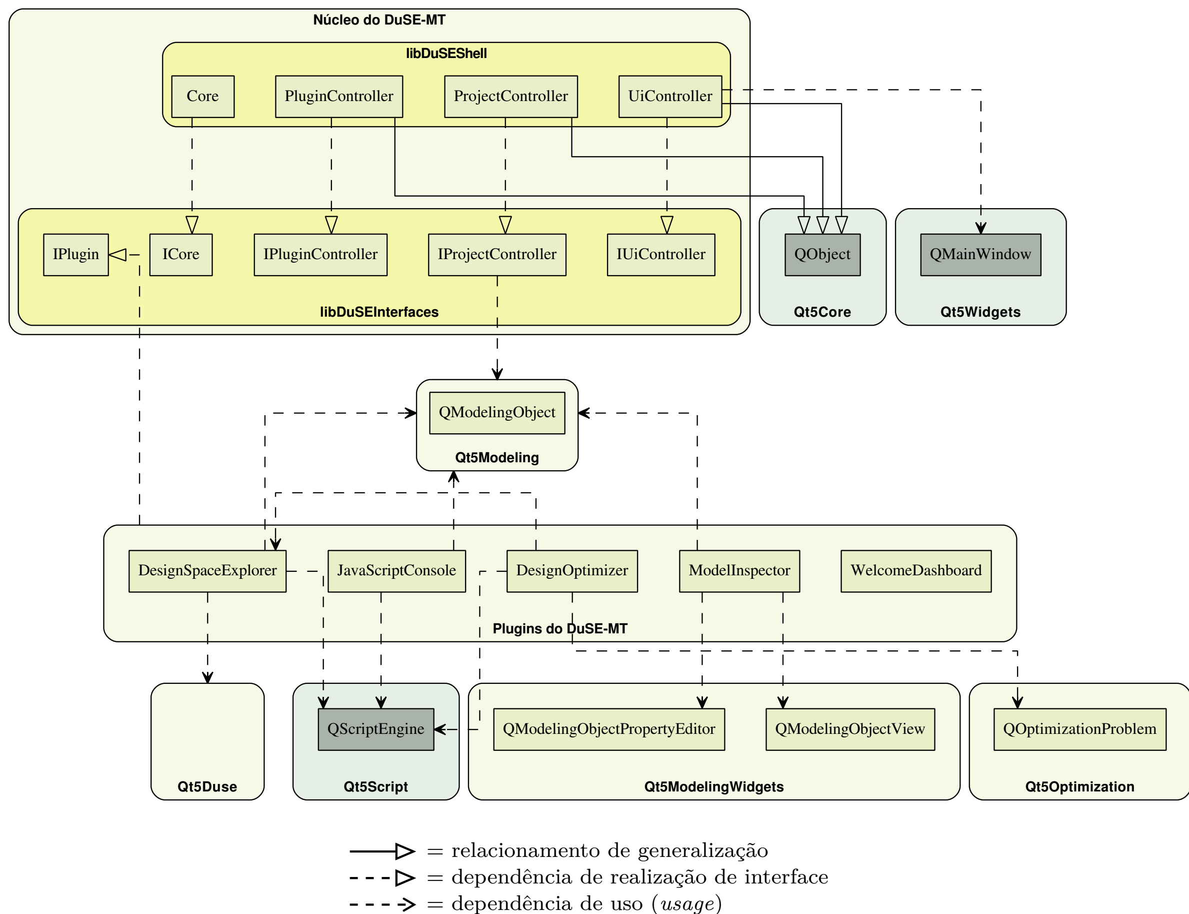


Figura 9.8.: Visão originada do estilo de decomposição (*module decomposition style*) para a ferramenta DuSE-MT.

O DuSE-MT foi projetado de modo a permitir, no futuro, a inclusão de outras funcionalidades além daquelas implementadas como parte deste trabalho. Para isso, um *framework* com arquitetura baseada em *microkernel* foi projetado. No padrão arquitetural *microkernel*, o núcleo da aplicação oferece as funcionalidades mínimas para a sua execução, deixando a cargo de *plugins* a implementação dos demais recursos. Com isso, potencializa-se o reuso do núcleo em outras aplicações de metamodelagem e facilita-se a evolução da ferramenta.

Conforme apresentado na Figura 9.8, o núcleo do DuSE-MT é composto por duas bibliotecas: `libDuSEInterfaces` e `libDuSEShell`. A biblioteca `libDuSEInterfaces` define a API a ser utilizada pelos *plugins*, viabilizando a evolução e compilação independente de tais extensões. Todo *plugin* do DuSE-MT implementa a interface `IPlugin`. Com isso, os *plugins* passam a ter acesso à instância de alguma implementação de `ICore` – um *singleton* responsável pela agregação dos *controllers* da aplicação. Tais *controllers* definem as APIs utilizadas pelos *plugins* para realizar as contribuições de funcionalidades desejadas.

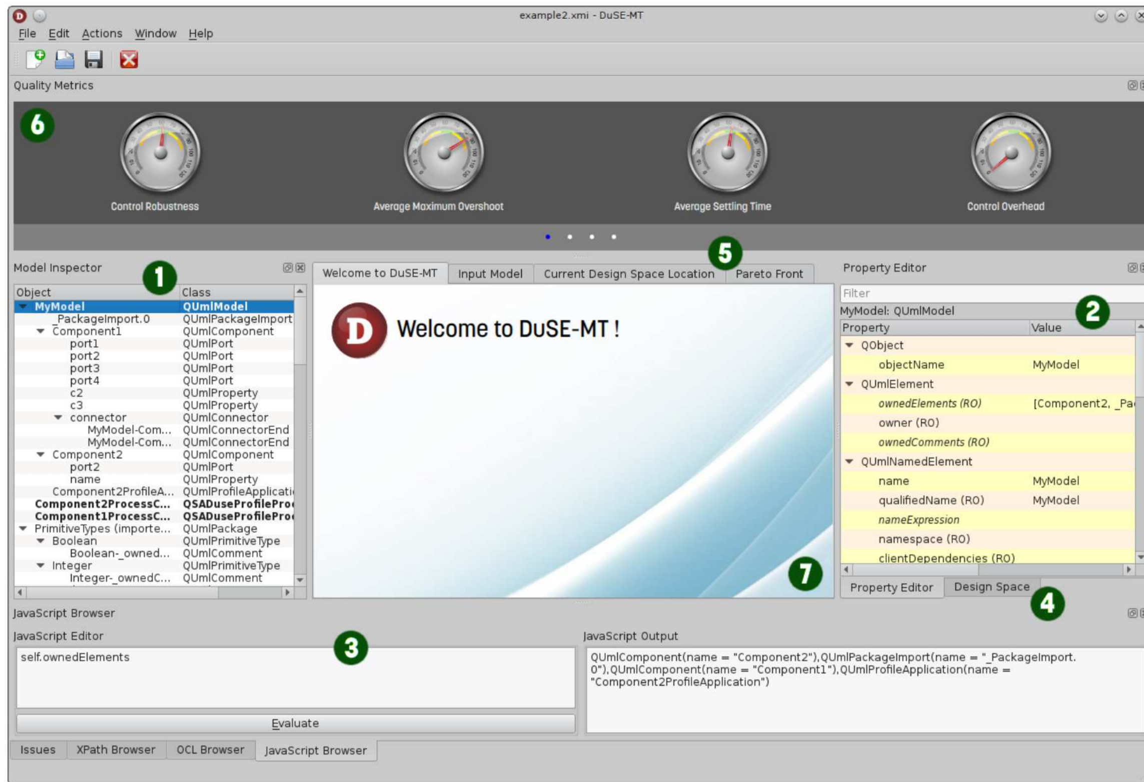


Figura 9.9.: Visão geral da interface gráfica de usuário do DuSE-MT.

A interface `IUiController` disponibiliza os recursos para inclusão de novos elementos visuais no DuSE-MT: itens de menu, botões de *toolbar* e as *toolviews* apresentadas com números na Figura 9.9 (explicadas a seguir). A interface `IProjectController` define a API que realiza as operações de leitura e gravação de arquivos XMI, bem como de inicialização de um processo automatizado de projeto (escolha do espaço de projeto e modelo arquitetural inicial a serem utilizados). Finalmente, a interface `IPluginController` disponibiliza os recursos para descoberta e inicialização dos *plugins*, bem como a verificação de dependências entre *plugins*. As quatro interfaces presentes na `libDuSEInterfaces` são implementadas na biblioteca `libDuSEShell`. Este separação é importante para que o núcleo da ferramenta evolua sem ter impacto nos *plugins*.

As funcionalidades para realização do processo automatizado de projeto arquitetural, aqui apresentado, foram implementadas como um grupo de *plugins* do DuSE-MT (apresentados na Figura 9.8). Os seguintes *plugins* foram implementados:

- `ModelInspector`: responsável pelas funcionalidades básicas de leitura, gravação, visualização e manipulação de modelos de *software*, descritos em alguma linguagem de metamodelagem representada por um *plugin* do `Qt5Modeling`. O `ModelInspector` adiciona as *toolviews* ① e ②, apresentadas na Figura 9.9.
- `JavaScriptConsole`: responsável pela consulta e manipulação de modelos via *JavaScript*. O módulo `Qt5Script` do Qt disponibiliza recursos para acesso a propriedades e invocação de métodos de objetos C++ via *JavaScript*. Em função dos recursos de reflexão computacional da classe `QObject`, o interpretador *JavaScript* do Qt (implementado na classe `QScriptEngine`) interaje com instâncias de `QObject` sem requerer mudanças consideráveis nas classes que originaram estas instâncias. Em função da alta produtividade promovida por esta abordagem e da dificuldade em implementar um *parser* OCL, todas as expressões OCL que representam seleção de elementos-alvo e pré-condição de modificações – apresentadas no Capítulo 7 – foram convertidas para *JavaScript*. O `JavaScriptConsole` adiciona a *toolview* ③ apresentada na Figura 9.9 e disponibiliza uma API para avaliação de *scripts*,

utilizada por outros *plugins*.

- `DesignSpaceExplorer`: responsável pela implementação das funcionalidades de geração do espaço de projeto específico de aplicação (Algoritmo 6.1, pág. 122) e geração de arquitetura candidata (ou identificação de arquitetura inválida) a partir de um vetor candidato (Algoritmo 6.2, pág. 123). Este *plugin* usa a implementação do metamodelo DuSE (módulo `Qt5Duse`) e os recursos de *scripting* do Qt para a realização de tais funcionalidades. O `DesignSpaceExplorer` adiciona os recursos para visualização: *i)* das instâncias de dimensão de projeto e seleção manual de pontos de variação (*toolview* ④ – aba "*Design Space*"); *ii)* da arquitetura candidata (ou inválida) associada ao vetor candidato (pontos de variação) atualmente selecionado (*toolview* ⑤ – aba "*Current Design Space Location*"); e *iii)* dos valores de métricas apresentados pela arquitetura candidata (*toolview* ⑥).
- `DesignOptimizer`: responsável pela inclusão das funcionalidades de otimização aos recursos de navegação manual de espaços de projeto implementados no `DesignSpaceExplorer`. O `DesignOptimizer` utiliza o módulo `Qt5Optimization` e disponibiliza especializações das classes `QOptimizationProblem`, `QIndividual`, `QCrossoverOperator`, `QMutationOperator` e `QObjectiveFunction` de modo a suportar as funcionalidades de otimização de arquiteturas apresentadas no Capítulo 167. Este *plugin* adiciona os recursos para configuração e execução da otimização (presentes na aba "*Design Space*") e visualização do *Pareto-front* encontrado (*toolview* ⑤ – aba "*Pareto Front*").
- `WelcomeDashboard`: responsável pela inclusão do *dashboard* apresentado na *toolview* ⑦. No futuro, informações gerais sobre a ferramenta, espaços de projeto e otimizações pode ser apresentadas nesta área.

O código-fonte do DuSE-MT está disponível em `http://wiki.ifba.edu.br/duse-mt`. Informações adicionais estão disponíveis em `http://duse.sf.net`. As atividades de desenvolvimento de *software*, realizadas como parte desta tese, produziram um total de 118 classes, implementadas em 15.920 linhas de código (excluíndo a geração automática da implementação dos metamodelos MOF e UML). 705 *commits* (submissões de código ao sistema de controle de versão) foram produzidos ao longo de um ano de trabalho.

9.3. Resumo do Capítulo

Este capítulo discutiu os requisitos, aspectos arquiteturais e detalhes de implementação do DuSE-MT – ferramenta de suporte ao processo automatizado de projeto arquitetural proposto neste trabalho. A ferramenta adota uma arquitetura com estilo *microkernel*, onde um núcleo de aplicação oferece as funcionalidades mínimas para a sua execução e *plugins* são responsáveis pela implementação dos demais recursos. A ferramenta foi concebida de modo a ser independente de metamodelo, apresentar bom desempenho, ter alta conformidade com as especificações de metamodelagem do *Object Management Group* e ser independente do algoritmo de otimização adotado.

Para isso, três grupos de módulos foram implementados. O primeiro grupo contém os módulos que implementam as funcionalidades básicas para criação, manipulação e serialização de modelos (`Qt5Modeling`), os metamodelos das linguagens MOF e UML (`Qt5Mof` e `Qt5Uml`, respectivamente) e os elementos visuais para visualização e edição de modelos (`Qt5ModelingWidgets`). O segundo grupo é formado pela implementação do metamodelo da linguagem DuSE (módulo `Qt5Duse`) e do *profile* UML de suporte ao SA:DuSE (módulo `Qt5SADuseProfile`). O terceiro grupo contém a implementação dos mecanismos de otimização multiobjetivo evolucionária (módulo `Qt5Optimization`). A ferramenta DuSE-MT integra as funcionalidades disponibilizadas pelos módulos anteriores através de um conjunto de *plugins*, disponibilizando o ambiente visual para realização do processo automatizado de projeto arquitetural aqui proposto.

Parte III

Avaliação e Conclusão

Capítulo 10

Avaliação

The aim of argument, or of discussion, should not be victory but progress.
Karl Popper

Este capítulo descreve as atividades, realizadas neste trabalho, para a avaliação dos mecanismos apresentados na Parte II. Os estudos descritos neste capítulo investigam a hipótese de que a abordagem para projeto arquitetural automatizado de sistemas *self-adaptive* proposta nesta tese captura corretamente os principais *trade-offs* presentes no projeto de *loops* de adaptação. Adicionalmente, deseja-se saber se as propriedades modeladas pelas métricas de qualidade descritas no Capítulo 7 são observadas, com acurácia satisfatória, em protótipos reais de sistemas *self-adaptive* implementando as arquiteturas produzidas pela abordagem aqui apresentada. Finalmente, um estudo para investigação da hipótese de que a abordagem aqui proposta auxilia arquitetos no projeto de arquiteturas mais efetivas para sistemas *self-adaptive* é também apresentado.

A Seção 10.1 descreve as diferentes metas de avaliação e as hipóteses (questões de avaliação) investigadas em cada meta. A Seção 10.2 apresenta os três estudos de avaliação realizados nesta tese, direcionados à investigação das hipóteses estabelecidas na Seção 10.1. Finalmente, a Seção 10.3 apresenta o resumo deste capítulo. A Figura 10.1 apresenta o roteiro deste capítulo.

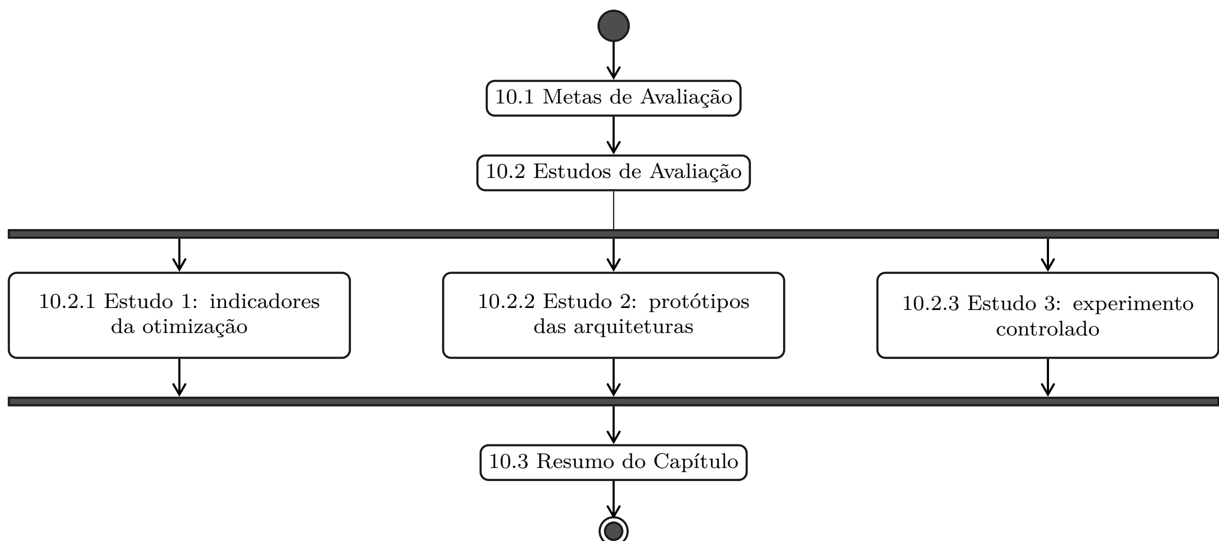


Figura 10.1.: Roteiro do capítulo 10.

10.1. Metas de Avaliação

Conforme descrito nos capítulos anteriores, a abordagem para projeto arquitetural automatizado de sistemas *self-adaptive*, proposta nesta tese, assume um conjunto de premissas e estabelece um grupo de hipóteses sobre os benefícios do uso de abordagens baseadas em busca no projeto de arquiteturas de *loops* de adaptação. Esta seção apresenta as diferentes dimensões (metas) de avaliação consideradas nesta tese – com suas respectivas questões de avaliação e instrumentos de investigação utilizados – e identifica as premissas e hipóteses não contempladas e, portanto, selecionadas para avaliação em trabalhos futuros.

O principal objetivo deste trabalho é a proposta de métodos mais efetivos para captura de conhecimento refinado de projeto, tornando-os mais sistematicamente organizados e oferecendo um suporte mais fundamentado à tomada de decisão referente aos *trade-offs* apresentados pelas possíveis arquiteturas alternativas. Em particular, o domínio de sistemas *self-adaptive* é caracterizado por um espaço de problema bastante intricado e por um espaço de solução amplo o suficiente para tornar frequente a adoção de arquiteturas inferiores. Esta situação ocorre, por exemplo, devido a viés por arquiteturas anteriormente utilizadas, alto custo de investigação de soluções alternativas, ou conhecimento limitado das soluções disponíveis. Supõe-se, neste trabalho, que a captura sistemática de conhecimento de projeto em um domínio específico, aliada à adoção de mecanismos para busca automática de arquiteturas efetivas, contribui significativamente para a redução dos problemas acima identificados. Alguns aspectos desta suposição são investigados neste capítulo.

A primeira avaliação realizada neste trabalho investiga se, dadas as dimensões de projeto e pontos de variação definidos no SA:DuSE, os *trade-offs* envolvidos no projeto de sistemas *self-adaptive* são efetivamente evidenciados. Características de convergência e espalhamento (vide Seção 4.2.1, pág. 70) do *Pareto-front* encontrado pelas otimizações são investigadas no Estudo 1 (apresentado na Seção 10.2.1). O objetivo é a análise das seguintes questões de validação:

QA1.1: as dimensões de projeto capturadas no espaço de projeto SA:DuSE refletem o caráter multiobjetivo do problema de projeto de arquiteturas para sistemas *self-adaptive* ?

QA1.2: as soluções apresentadas pelo mecanismo de otimização de arquiteturas aqui proposto apresentam boa convergência e bom espalhamento ?

Note que, embora o NSGA-II seja reconhecidamente eficiente na obtenção de *Pareto-fronts* com boa convergência e espalhamento, a modelagem do problema e as métricas de avaliação de indivíduos adotadas têm influência direta nestes aspectos. Espaços de projeto de maior complexidade podem implicar em espaços-objetivo irregulares e com maior susceptibilidade a máximos locais. O Estudo 1 investiga até que ponto o mecanismo de otimização aqui adotado consegue obter bom desempenho na busca por soluções efetivas no espaço-objetivo produzido pelo SA:DuSE. A obtenção de *Pareto-fronts* de alta cardinalidade evidencia o sucesso do SA:DuSE na captura dos *trade-off* envolvidos.

As demais avaliações realizadas neste trabalho são baseadas nos diferentes níveis de validação de abordagens de predição baseadas em modelos, propostos por Böhme & Reussner [40]. Os três níveis de validação são apresentados a seguir.

Nível 1 (validação da métrica): refere-se à comparação das predições disponibilizadas pela métrica com medições realizadas em protótipos reais. No caso particular de métricas de qualidade em arquiteturas de *software*, deseja-se saber se os valores disponibilizados pelas métricas são de fato observados em implementações reais da arquitetura em questão. Este aspecto é fundamental para a definição de métodos efetivos de análise arquitetural.

O espaço de projeto para sistemas *self-adaptive*, apresentado no Capítulo 7, viabiliza a busca por arquiteturas ótimas em relação a quatro atributos de qualidade: tempo médio de estabilização, sobressinal médio, efetividade de controle e *overhead* de controle. Métricas para estimativa destes atributos foram apresentadas em detalhes na Seção 7.3, pág. 161. Dentre estas métricas, duas possuem caráter preditivo: tempo médio de estabilização e sobressinal médio.

Embora estas métricas sejam originadas na área de Teoria de Controle e possuam suporte matemático rigoroso, é importante verificar com qual acurácia estes valores são observados em implementações reais das arquiteturas produzidas pelo espaço de projeto SA:DuSE. O objetivo é investigar se as contribuições arquiteturais, introduzidas pelos diversos pontos de variação apresentados no Capítulo 7, preservam as interações das dinâmicas do sistema-alvo e do controlador adotado, conforme esperado pelas análise realizadas via Teoria de Controle. Desse modo, a seguinte questão de validação é definida para o nível 1:

QA2: as modificações arquiteturais definidas pelos pontos de variação do SA:DuSE produzem arquiteturas de *loops* de adaptação que preservam as predições de tempo médio de estabilização e sobressinal médio oferecidas pela Teoria de Controle ?

Esta meta de avaliação é investigada no Estudo 2 (apresentado na Seção 10.2.2). Protótipos reais dos *loops* de adaptação produzidos pelo SA:DuSE para o modelo do servidor *web* MIMO e do *cluster* elástico para aplicações *MapReduce* apresentado na Seção 5.3.3, pág. 101, são utilizados para verificar este aspecto.

Nível 2 (validação da aplicabilidade): investiga diversos aspectos da viabilidade de aplicação prática do modelo de predição em questão. Tais aspectos estão relacionados à facilidade/dificuldade de criação dos modelos, execução das predições e interpretação dos resultados obtidos. Uma das premissas adotadas neste trabalho é que um modelo arquitetural inicial – anotado com as dinâmicas do sistema-alvo original – pode ser criado. O *profile* UML que acompanha o SA:DuSE foi criado de modo a demandar o mínimo necessário de anotações. As modificações arquiteturais definidas pelos pontos de variação se tornam mais complexas à medida em que um número menor de anotações é exigido no modelo arquitetural inicial.

Arquitetos construindo espaços de projeto para novos domínios de aplicação devem encontrar um bom equilíbrio em relação a esta questão. Visto que o espaço de projeto é definido uma única vez por domínio de aplicação – e utilizado inúmeras vezes para projeto de aplicações deste domínio – considera-se desejável adotar o mínimo necessário nas demandas por anotações. No caso particular do SA:DuSE, tais anotações se resumem à descrição das dinâmicas do sistema-alvo. Estes dados podem ser facilmente obtidos com a realização de testes de ensaio ao degrau, conforme apresentado na Seção 3.3.4.2, pág. 53. Obviamente, a realização do ensaio ao degrau requer que o sistema-alvo já exista e possa ser experimentado. Acredita-se, entretanto, ser frequente a situação onde o sistema-alvo já existe e a abordagem proposta neste trabalho é então utilizada para identificar as arquiteturas de controle mais adequadas para o gerenciamento do sistema existente.

Outros aspectos referentes à aplicabilidade da abordagem aqui proposta incluem a facilidade de execução das otimizações e a capacidade de interpretação dos resultados obtidos. Visto que, uma vez criado o modelo arquitetural inicial, o processo de busca por arquiteturas efetivas acontece de forma automatizada, acredita-se que mesmo arquitetos iniciantes podem se beneficiar dos produtos deste trabalho. A interpretação do *Pareto-front* obtido, por outro lado, requer conhecimento dos atributos de qualidade avaliados pelas métricas adotadas e das soluções implementadas pelas arquiteturas candidatas. Interpretações corretas são importantes para estimar o custo de implementação da solução finalmente escolhida. Diante do exposto acima, e em razão do foco adotado neste trabalho às atividades de projeto e análise de arquiteturas (em

contraponto à implementação das soluções obtidas), decidiu-se pela não avaliação dos aspectos do Nível 2 nesta tese.

Nível 3 (validação do benefício): analisa os benefícios da abordagem de predição quando comparada a abordagens alternativas. O objetivo é investigar se a abordagem proposta traz melhorias à prática corrente. O trabalho aqui apresentado propõe um novo processo de projeto de arquiteturas de *software*, caracterizado pelo uso de representações sistemáticas do conhecimento de projeto e de técnicas de otimização multiobjetivo. Dessa forma, a questão a ser avaliada neste nível é se a abordagem aqui proposta traz benefícios em algum aspecto relevante, quando comparada a técnicas tradicionais de projeto arquitetural como as descritas na Seção 2.3, pág. 17. As seguintes questões de avaliação foram definidas para investigar aspectos do Nível 3:

QA3.1: o uso de técnicas de projeto arquitetural baseadas em busca faz com que as arquiteturas de sistemas *self-adaptive* produzidas sejam mais efetivas em relação ao atendimento das propriedades de controle ?

QA3.2: o uso de técnicas de projeto arquitetural baseadas em busca faz com que as arquiteturas de sistemas *self-adaptive* produzidas sejam menos complexas ?

QA3.3: o uso de técnicas de projeto arquitetural baseadas em busca facilita a aquisição, por arquitetos iniciantes, de conhecimento refinado de projeto na área de sistemas *self-adaptive* ?

Para avaliar estas questões, um experimento controlado com 24 estudantes de pós-graduação foi conduzido, com o objetivo de comparar os produtos da abordagem aqui proposta em relação àqueles produzidos por técnicas convencionais de projeto arquitetural. Detalhes sobre o experimento são apresentados no Estudo 3 (Seção 10.2.3).

10.2. Estudos de Avaliação

Esta seção apresenta os estudos realizados, nesta tese, para investigar as questões de validação descritas na Seção 10.1. Todos os estudos utilizaram dois modelos arquiteturais iniciais: o servidor *web* MIMO e o *cluster* elástico para aplicações *MapReduce* (apresentados nas seções 5.3.2 e 5.3.3, respectivamente).

10.2.1. Estudo 1: Efetividade da Captura de Trade-offs no SA:DuSE

Este estudo teve como objetivo analisar a efetividade do espaço de projeto SA:DuSE em relação à captura dos *trade-offs* arquiteturais envolvidos no projeto de sistemas *self-adaptive* (questão de avaliação QA1.1). Adicionalmente, avaliou-se a qualidade do *Pareto-front* encontrado em uma única execução de otimização em relação à sua proximidade a um (suposto) *Pareto-front* global e ao espalhamento das soluções nele presentes (questão de avaliação QA1.2).

Para responder a questão de avaliação QA1.1, 31 execuções de buscas no espaço de projeto SA:DuSE foram realizadas para cada modelo inicial. O *Pareto-front* do conjunto formado por todas as 31 saídas de cada execução – denominado *Pareto-front* de referência P^* – foi calculado para cada modelo e analisou-se as características deste *Pareto-front* e das soluções nele presentes. Um *Pareto-front* com alta cardinalidade e alto espalhamento é um indício da efetividade do SA:DuSE na captura dos *trade-offs* envolvidos. Um número de execuções de buscas maior que 31 não apresentou, para ambos os modelos, diferença na cardinalidade do *Pareto-front* de referência obtido. Este procedimento foi realizado para ambos os modelos iniciais (servidor *web* MIMO e

cluster elástico para aplicações *MapReduce*), produzindo, respectivamente, os *Pareto-fronts* de referência P^*_{SW} e P^*_C.

Para responder a questão de avaliação QA1.2, comparou-se – para cada modelo inicial – o *Pareto-front* obtido em uma única execução da busca com o *Pareto-front* de referência P^*, obtido anteriormente. Visto que a obtenção do *Pareto-front* global por força bruta é inviável devido à alta cardinalidade do espaço de projeto, assume-se que o *Pareto-front* de referência P^* é uma boa aproximação do *Pareto-front* global. Analisou-se então quão próximo o *Pareto-front* obtido em uma única execução está do *Pareto-front* de referência P^*. Adicionalmente, investigou-se a velocidade de convergência da métrica *hypervolume* (apresentada na Seção 4.4.3, pág. 86) ao longo das iterações da otimização.

Os detalhes deste estudo utilizando o modelo do servidor *web* MIMO são apresentados na Seção 10.2.1.1. A Seção 10.2.1.2 apresenta os resultados obtidos com o modelo do *cluster* elástico para aplicações *MapReduce*. Uma discussão geral do Estudo 1 é apresentada na Seção 10.2.1.3.

10.2.1.1. Servidor Web MIMO

O modelo do servidor *web* MIMO – apresentado na Seção 5.3.2, pág. 99 – representa um cenário mais simples (porém não menos desafiador) de autogerenciamento pois não apresenta demandas para cooperação entre múltiplos *loops*. Ainda assim, o número total de arquiteturas presentes no espaço de projeto específico de aplicação produzido pelo SA:DuSE neste caso é:

$$(2 \times 7 \times 7 \times 5 \times 6)^{\#portas\text{-}control\acute{a}veis=2} = 8643600 \quad (10.1)$$

Supondo que a construção e avaliação de cada arquitetura candidata leve cerca de 200ms (tempo médio observado nas experimentações), a análise de todo o espaço de projeto para o modelo do servidor *web* MIMO demoraria cerca de 20 dias[1]. Para o modelo do *cluster* elástico para aplicações *MapReduce*, discutido na Seção 10.2.1.2, o número de arquiteturas sobe para 2.5412184e10 e a avaliação do espaço de projeto por força bruta demoraria cerca de 161 anos. Frente a este cenário, assumiu-se como representante significativo do *Pareto-front* global o *Pareto-front* do conjunto formado pelas saídas de 31 execuções de otimização (*Pareto-front* de Referência P^*_{SW}), servindo como base para a análise do espalhamento das soluções encontradas e do desempenho de execuções individuais das buscas.

Configuração da Otimização. Para a busca de arquiteturas de controle para o servidor *web* MIMO, configurou-se o DuSE-MT para executar 300 iterações de otimização mantendo uma população de 20 arquiteturas por iteração. Uma população inicial de 20 arquiteturas, aleatoriamente selecionadas no espaço de projeto, foi utilizada. O operador `DuSECrossoverOper` (vide Seção 8.2.2, pág. 170) foi configurado para encapsular o operador convencional de recombinação com dois pontos (vide Seção 4.3.2.1, pág. 78) e utilizar uma probabilidade de recombinação de 0.9. O operador `DuSEMutationOper` encapsulou o operador convencional de mutação bit-a-bit com probabilidade de mutação igual a 1. Tais valores se mostraram adequados para garantir a diversidade dos indivíduos e minimizar a ocorrência de arquiteturas inválidas. O operador convencional de seleção do NSGA-II (seleção por torneio aglomerado) foi utilizado.

A métrica de efetividade de controle do SA:DuSE (vide Seção 7.3.3, pág. 163) foi configurada com $\alpha^E_r = -2$ (atribuição maior de efetividade já com o uso da técnica de escalonamento de ganho). Visto que o modelo do servidor *web* MIMO não apresenta cooperação entre múltiplos *loops*, o valor do parâmetro β^E foi configurado para 1 de modo a não considerar a parcela de

[1] Considerando uma única máquina, uma execução de otimização por vez e uma implementação não-concorrente do algoritmo evolucionário de otimização multiobjetivo.

efetividade de controle dada pelo grau de cooperação entre *loops*. O valor de α_m^E é irrelevante (foi mantido em 0) visto que ele não impacta o valor de efetividade obtido. De forma semelhante, a métrica de *overhead* de controle do SA:DuSE (vide Seção 7.3.4, pág. 164) utilizou um valor de $\alpha_r^O = 4$ (aumento pequeno de *overhead* ao utilizar escalonamento de ganho e significativamente maior com as outras alternativas de adaptação de controle). Pelos mesmos motivos apresentados acima, o valor do parâmetro β^O foi configurado para 1.

As otimizações foram executadas numa máquina Intel Core i7, com 4 núcleos reais e 8GB de memória RAM, utilizando a ferramenta DuSE-MT no sistema operacional GNU/Linux, *kernel* 3.17. Cada uma das 31 execuções analisou cerca de 5400 arquiteturas e durou, em média, 18 minutos. Embora o módulo `Qt5Optimization` ainda não implemente o algoritmo NSGA-II de forma concorrente (aproveitando os múltiplos núcleos presentes na máquina em questão), 4 instâncias do DuSE-MT foram executadas em paralelo, fazendo com que as 31 execuções durassem cerca de 2.3 horas.

Resultados Obtidos. No total, cerca de 167400 das 8643600 arquiteturas do espaço de projeto foram analisadas durante as 31 execuções (provavelmente incluindo arquiteturas repetidas) e o *Pareto-front* de referência P_{SW}^* apresentou 13 arquiteturas candidatas. A Figura 10.2 apresenta as soluções presentes em P_{SW}^* e as soluções dominadas de uma das populações finais encontradas na otimização do modelo do servidor *web* MIMO. Os resultados são apresentados em uma matriz de gráficos de dispersão (*scatter plot matrix*), onde as quatro funções-objetivo são apresentadas em pares.

Cada célula na diagonal principal da matriz exibe, em relação à métrica descrita na linha (ou coluna) em questão, o histograma de todas as arquiteturas da população (barras sem hachuras) e das arquiteturas presentes no *Pareto-front* (barras com hachuras). As demais células apresentam, no eixo das abscissas, os valores da métrica indicada na parte superior ou inferior da coluna em questão. No eixo das ordenadas, são apresentados os valores da métrica indicada no lado esquerdo ou direito da linha em questão. Por exemplo, o gráfico de dispersão na segunda linha e primeira coluna apresenta o tempo médio de estabilização no eixo das abscissas e o sobressinal médio no eixo das ordenadas.

Arquiteturas do *Pareto-front* são apresentadas como losangos enquanto as demais (soluções dominadas) são representadas por círculos. Os *Pareto-fronts* parciais – calculados utilizando apenas duas das quatro funções-objetivos – são também apresentados, em cada célula, por linhas contínuas ligando as soluções não-dominadas.

Conforme apresentado na legenda da Figura 10.2, a cor de um círculo ou losango indica a técnica de sintonia (dimensão DD4 – vide Seção 7.2.4, pág. 144) escolhida para a porta `IKATimeout` do servidor *web* MIMO. Já o tamanho de cada círculo ou losango indica o grau de adaptabilidade do controlador utilizado (dimensão DD5 – vide Seção 7.2.5, pág. 147) – quanto maior o tamanho mais robusto é o controlador. Note que os pontos de variação da dimensão DD5 foram definidos, no Capítulo 7, em ordem crescente de adaptabilidade de controle (ganho fixo, escalonamento de ganho, MIAC, MRAC e controle reconfigurável).

A Figura 10.3 apresenta como a métrica *hypervolume* se comporta ao longo das iterações. Deseja-se saber quão próximo o *hypervolume*, de uma única execução, consegue chegar daquele formado pelo *Pareto-front* de referência $HV(P_{SW}^*)$ (descrito na figura em linha vermelha sem marcadores). Visto que o *hypervolume* é uma variável aleatória, 10 replicações da otimização foram realizadas, produzindo as populações finais $T_{r=1,2,\ldots,10}$. A Figura 10.3 apresenta os valores médio, máximo, mínimo e desvio padrão do *hypervolume* gerado, em cada replicação r, pelo *Pareto-front* da população presente em cada iteração i (T_r^i).

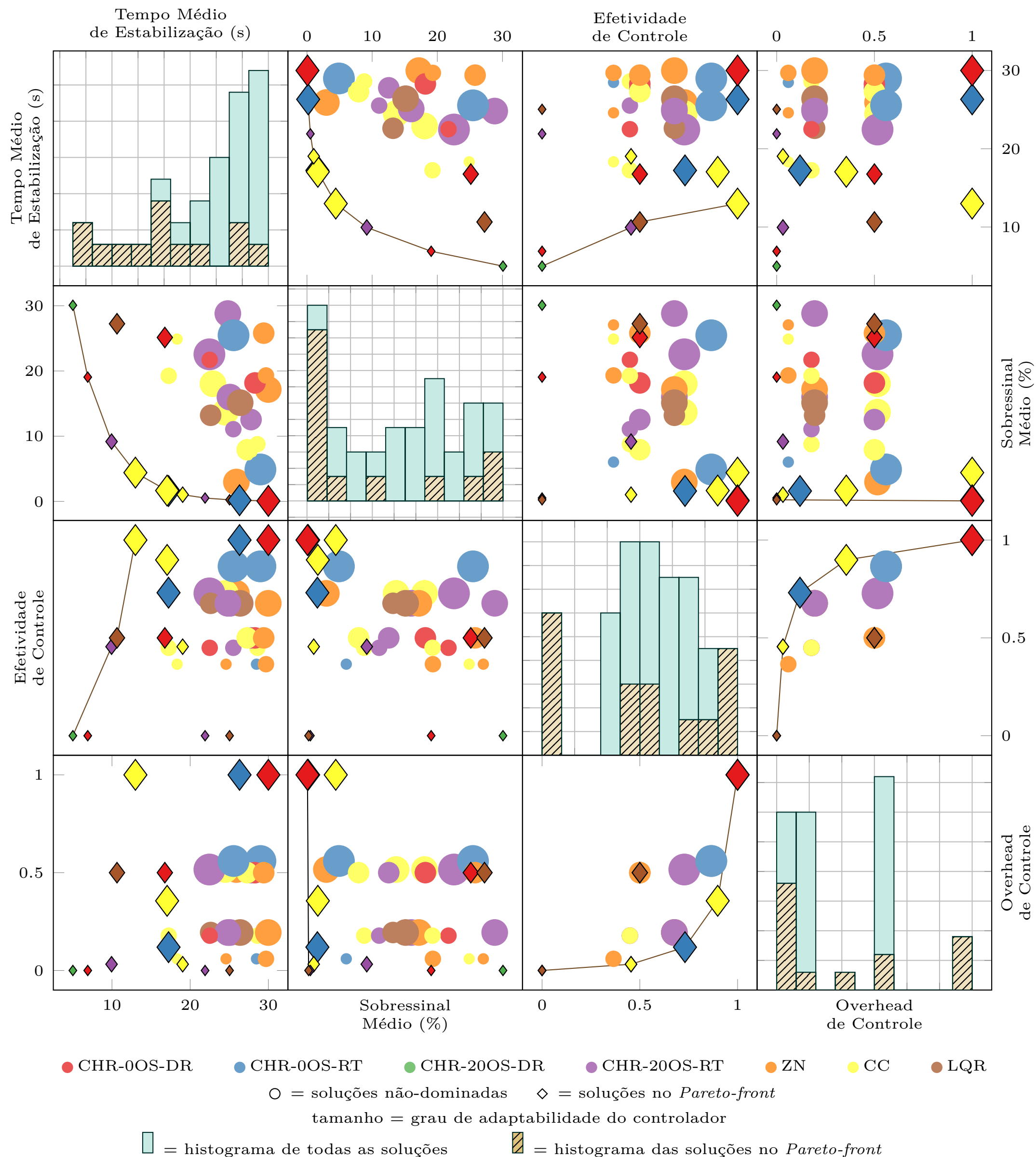


Figura 10.2.: Matriz de gráficos de dispersão do *Pareto-front* de referência P^{*}_{SW} obtido para o modelo do servidor *web* MIMO.

Análise dos Dados. Algumas observações podem ser realizadas sobre os dados apresentados na Figura 10.2. Primeiro, nota-se que as arquiteturas mais frequentes na população final são aquelas com tempo de estabilização entre 25s e 30s. As arquiteturas com tempo de estabilização menor apresentam a sétima maior frequência na população. Arquiteturas com efetividade mediana e baixo *overhead* de controle (não necessariamente as mesmas arquiteturas) são também bastante frequentes. Este cenário corrobora o benefício de uso de abordagens baseadas em busca no projeto de arquiteturas para sistemas *self-adaptive*, visto que explorações manuais ou buscas aleatórias dificilmente encontrariam arquiteturas com baixo tempo de estabilização.

Segundo, observa-se a existência do *trade-off* característico entre tempo de estabilização e sobressinal, bem como entre efetividade de controle e *overhead* de controle. O gráfico de dispersão para estas variáveis apresenta um *Pareto-front* de maior cardinalidade e maior espalhamento, in-

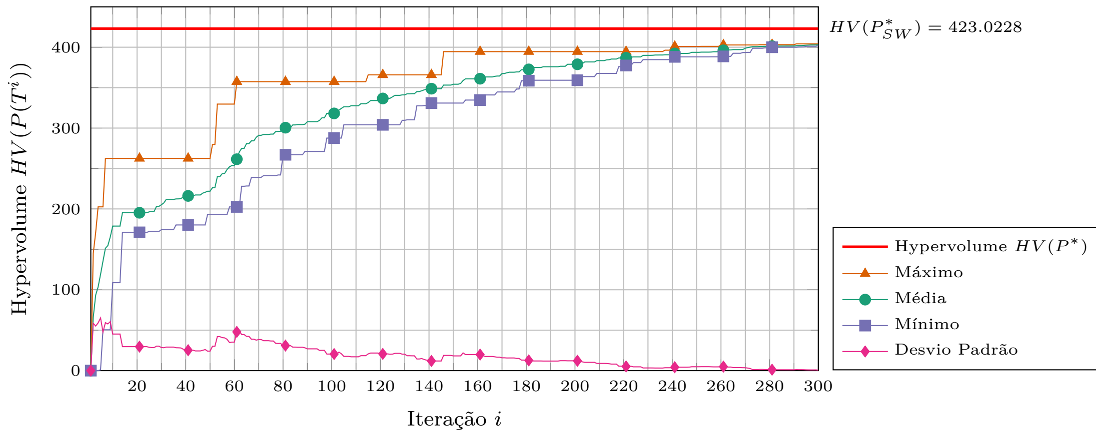


Figura 10.3.: Gráfico de convergência da métrica *hypervolume* durante as iterações de otimização do modelo do servidor *web* MIMO.

dicando que o problema de otimização destas variáveis é de fato multiobjetivo. Tanto o *trade-off* entre tempo de estabilização e efetividade de controle quanto aquele entre sobressinal e *overhead* de controle apresentam *Pareto-fronts* menos expressivos (de menor cardinalidade, conforme apresentado na figura). O *Pareto-front* entre tempo de estabilização e *overhead* de controle apresenta uma única solução, indicando que tais objetivos não são conflitantes. O mesmo é observado para o *trade-off* entre sobressinal e efetividade de controle.

Terceiro, a técnica de sintonia utilizada na arquitetura com menor tempo médio de estabilização (5s) foi a CHR-20OS-DR (vide Seção 3.3.4.4, pág. 61). Arquiteturas com tempo de estabilização um pouco maior, porém com valores menores de sobressinal, foram construídas com as técnicas de Ziegler-Nichols e Cohen-Coon. Apenas uma arquitetura sintonizada com a técnica LQR esteve presente no *Pareto-front* obtido. A arquitetura com 0% de sobressinal, porém com maior tempo de estabilização (30s), foi produzida com a técnica CHR-0OS-DR.

Finalmente, embora caracterizado por um *Pareto-front* de menor cardinalidade, o *trade-off* entre efetividade de controle e *overhead* de controle reflete bem a influência do grau de adaptabilidade do controlador nestas variáveis. Note como o tamanho da solução (grau de adaptabilidade do controlador) cresce à medida em que sua posição varia da coordenada (0,0) – menor efetividade e menor *overhead* – para a coordenada (1,1) – máxima efetividade e máximo *overhead*.

Em relação à Figura 10.3, nota-se que o *hypervolume* $HV(P(T^i))$ de uma única execução de otimização converge ao longo das iterações, em média, para $\overline{HV}(P(T^{300})) = 403.1808$, cerca de 95.3% do *hypervolume* apresentado pelo *Pareto-front de referência* P^*_{SW}. Conclui-se, dessa forma, que execuções adicionais de otimização têm pouca probabilidade de contribuir com novas soluções do *Pareto-front*. Adicionalmente, para este caso particular de otimização, 95% do valor final do *hypervolume* médio ($0.95 \cdot 403.1808 = 383.02176$) é atingido na iteração de número 209. Daí para frente, a descoberta de soluções mais efetivas passa a ser mais difícil.

Para melhor avaliar a convergência do *hypervolume* de uma única execução, investigou-se – em cada iteração i – a hipótese nula de que a média do *hypervolume* é menor ou igual a 95% do valor final do *hypervolume*: $H_0: \overline{HV}(P(T^i)) \leq 0.95 \cdot \overline{HV}(P(T^{300}))$. O objetivo foi encontrar, com um determinado nível de significância α, a menor iteração onde a hipótese H_0 é rejeitada (o *hypervolume* ultrapassa 95% do seu valor final).

O primeiro passo realizado foi verificar se as premissas usuais para utilização de testes paramétricos de hipótese eram atendidas. Testes paramétricos são preferíveis pois apresentam maior potência estatística (habilidade do teste na detecção de um efeito, dado que o efeito realmente existe). As premissas comumente verificadas são: *i)* dados são obtidos a partir de uma escala intervalar ou racional (atendida para o caso do *hypervolume*); *ii)* observações são independen-

tes (uma execução de otimização não interfere na outra); *iii)* valores medidos são normalmente distribuídos nas populações; e *iv)* as variâncias das populações são iguais entre grupos (homoscedasticidade).

O teste de Anderson-Darling [68] foi utilizado para avaliar – em cada iteração i – se os valores de *hypervolume* $HV(P(T^i))$ eram normalmente distribuídos. Já a hipótese nula de homoscedasticidade entre os valores de *hypervolume* $HV(P(T^i))$ e os valores de *hypervolume* da iteração final $HV(P(T^{300})$ foi investigada com o teste de Levene [115]. Utilizou-se nível de significância $\alpha = 0.05$ em ambos os testes.

Iterações com *p-value* de Anderson-Darling ≥ 0.05 apresentam valores de *hypervolume* normalmente distribuídos. Iterações com *p-value* de Levene ≥ 0.05 apresentam homoscedasticidade nos valores de *hypervolume*. Se estas duas condições forem satisfeitas, a hipótese H_0 deve ser investigada utilizando métodos paramétricos. Caso contrário, métodos não-paramétricos devem ser aplicados. Utilizou-se o *t-test* [322] como método paramétrico e o *Wilcoxon Signed-Rank* (SR) [322, 68] como método não-paramétrico, ambos unicaudais (a hipótese H_0 é uma desigualdade) com duas amostras: $HV(P(T^i))$ e $HV(P(T^{300})$.

Iteração i	$\overline{HV}(P(T^i))$	***p-value* de Anderson-Darling**	***p-value* de Levene**	**Teste Utilizado**	***p-value* (*t-test* / *Wilcoxon SR*)**
...					
214	384.557789	0.126933	0.017576	*Wilcoxon SR*	0.071252
215	385.380244	0.063485	0.033153	*Wilcoxon SR*	**0.011840**
216	385.380244	0.063485	0.033153	*Wilcoxon SR*	0.011840
...					
227	388.018550	0.481500	0.004645	*Wilcoxon SR*	0.011840
228	389.914992	0.086326	0.012296	*Wilcoxon SR*	**0.000023**
229	389.944924	0.073570	0.012241	*Wilcoxon SR*	0.000023
...					

Tabela 10.1.: Resultados dos testes de hipótese para verificação de convergência do *hypervolume* na otimização do modelo do servidor *web* MIMO.

A Tabela 10.1 apresenta os resultados obtidos para as iterações 214 a 216 e 227 a 229. Embora todas estas iterações tenham apresentado *p-value* de Anderson-Darling ≥ 0.05 (indicando que os *hypervolumes* são normalmente distribuídos), nenhuma iteração apresentou *p-value* de Levene ≥ 0.05 (apresentam heteroscedasticidade nos valores de *hypervolume*). Como ambas as condições devem ser satisfeitas para o uso do método paramétrico, o teste de *Wilcoxon Signed-Rank* foi adotado em todas estas iterações.

Com um nível de significância (α) de 0.05, pode-se afirmar que o *hypervolume* de uma única execução atinge 95% do seu valor final na iteração 215 (primeiro *p-value* com valor menor que 0.05). Ou seja, de cada 100 execuções de otimização, em 95 delas o *hypervolume* terá atingindo 95% do seu valor final na 215ª iteração. Com um nível de significância de 0.01 (99% das execuções) o limite para atingir os 95% passa a ser a iteração 228 (primeiro *p-value* com valor menor que 0.01).

O cálculo do *hypervolume* foi realizado na ferramenta R [37], utilizando o pacote `emoa`[2] de algoritmos de otimização multiobjetivo evolucionária. Os testes de Anderson-Darling, *t-test* e *Wilcoxon Signed-Rank* foram executados na ferramenta Scilab [114], com o auxílio do módulo CASCILIB (*CAstagliola's SCIlab LIBrary*)[3].

[2] `http://cran.r-project.org/web/packages/emoa/`.
[3] `https://atoms.scilab.org/toolboxes/casci`.

10.2.1.2. Cluster Elástico para Aplicações MapReduce

O segundo modelo utilizado no Estudo 1 – o *cluster* elástico para aplicações *MapReduce* apresentado na Seção 5.3.3, pág. 101 – apresenta mais desafios de autogerenciamento pois introduz demandas para cooperação entre múltiplos *loops*. Neste cenário, o número total de arquiteturas presentes no espaço de projeto específico de aplicação produzido pelo SA:DuSE é:

$$(2 \times 7 \times 7 \times 5 \times 6)^{\#portas\text{-}control\acute{a}veis=3} = 2.5412184e10 \quad (10.2)$$

A mesma estratégia de comparação utilizada no modelo do servidor *web* MIMO foi aqui aplicada: o *Pareto-front* do conjunto formado pelas saídas de 31 execuções de otimização (P^*_C) foi utilizado como representante significativo do *Pareto-front* global.

Configuração da Otimização. As buscas executadas para este modelo realizaram 300 iterações de otimização. Em função da maior cardinalidade do espaço de busca neste caso – quando comparado ao modelo do servidor *web* MIMO – uma população um pouco maior (40 arquiteturas) foi mantida em cada iteração. Uma conjunto inicial de 40 arquiteturas, aleatoriamente selecionadas no espaço de projeto, foi utilizada. Os operadores `DuSECrossoverOper`, `DuSEMutationOper` e o operador convencional de seleção do NSGA-II utilizaram as mesmas configurações aplicadas no modelo do servidor *web* MIMO.

A métrica de efetividade de controle do SA:DuSE (vide Seção 7.3.3, pág. 163) foi configurada com $\alpha^E_r = -2$ (atribuição maior de efetividade já com o uso da técnica de escalonamento de ganho). Visto que, neste caso, decisões sobre formas de cooperação entre múltiplos *loops* foram avaliadas, o valor do parâmetro β^E foi configurado para 0.5, de modo a considerar de forma igualitária as parcelas de efetividade de controle dadas pela robustez de controle e pelo grau de cooperação entre *loops*. Utilizou-se $\alpha^E_m = 2$ para indicar que uma maior penalidade de *overhead* só é obtida ao selecionar aqueles pontos de variação da dimensão DD6 que trabalham de forma mais centralizada. De forma similar, a métrica de *overhead* de controle do SA:DuSE (vide Seção 7.3.4, pág. 164) utilizou um valor de $\alpha^O_r = 4$ (aumento pequeno de *overhead* ao utilizar escalonamento de ganho e significativamente maior com as outras alternativas de adaptação de controle). O valor do parâmetro β^O foi configurado para 0.25, indicando que 25% do *overhead* é influenciado pela técnica de adaptação de controle utilizada e 75% pela forma de cooperação entre múltiplos *loops* (pois envolve troca de mensagens via rede de comunicação).

Cada uma das 31 execuções analisou cerca de 10800 arquiteturas e durou, em média, 36 minutos. 4 instâncias do DuSE-MT foram executadas em paralelo, fazendo com que as 31 execuções durassem cerca de 4.65 horas.

Resultados Obtidos. No total, cerca de 334800 arquiteturas do espaço de projeto foram analisadas durante as 31 execuções (provavelmente incluíndo arquiteturas repetidas) e o *Pareto-front* de referência P^*_C apresentou 18 arquiteturas candidatas. A Figura 10.4 apresenta as soluções presentes em P^*_C e as soluções dominadas de uma das populações finais encontradas na otimização do modelo do *cluster* elástico para aplicações *MapReduce*.

Conforme apresentado na legenda da Figura 10.4, a cor de um círculo ou losango indica a forma de cooperação entre múltiplos *loops* (dimensão DD6 – vide Seção 7.2.6, pág. 153) escolhida para a porta `IMaxMappers` dos serviços `NodeManager` do *cluster*. O tamanho de cada círculo ou losango continua indicando, como no exemplo do servidor *web* MIMO, o grau de adaptabilidade do controlador utilizado.

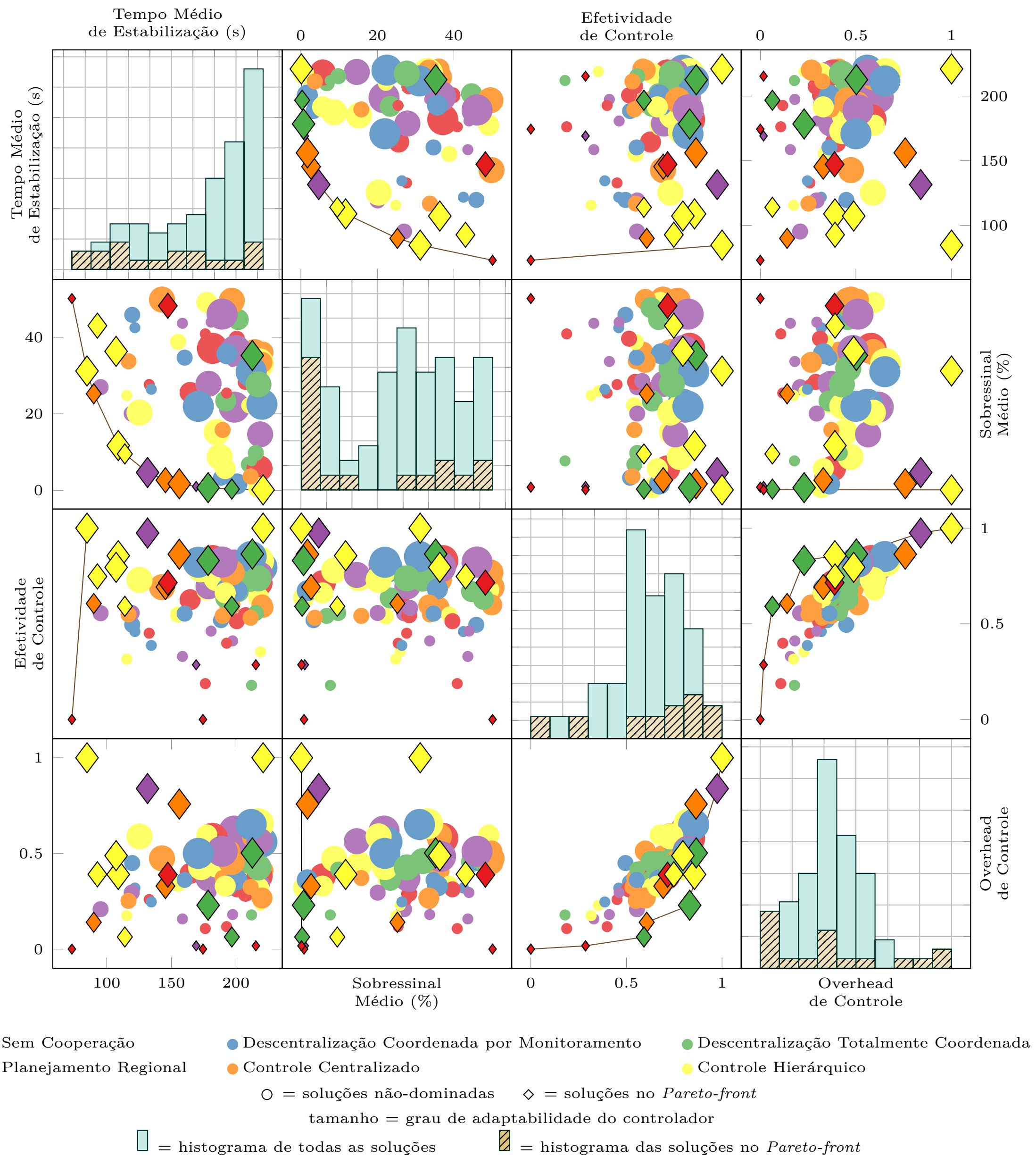


Figura 10.4.: Matriz de gráficos de dispersão do *Pareto-front* de referência P_C^* obtido para o modelo do *cluster* elástico para aplicações *MapReduce*.

A Figura 10.5 apresenta como a métrica *hypervolume* se comporta ao longo das iterações. De forma semelhante ao modelo do servidor *web* MIMO, 10 replicações da otimização foram realizadas, produzindo as populações finais $T_{r=1,2,...,10}$. Os valores médio, máximo, mínimo e desvio padrão do *hypervolume* gerado, em cada replicação r, pelo *Pareto-front* da população presente em cada iteração i (T_r^i) são apresentados na Figura 10.3.

Análise dos Dados. Em relação aos dados apresentados na Figura 10.4, nota-se que as arquiteturas mais frequentes na população final são aquelas com alto tempo de estabilização. As arquiteturas com baixo tempo de estabilização são raras, dificultando sua descoberta via exploração manual do espaço de projeto. Arquiteturas com efetividade e *overhead* de controle medianos são também bastante frequentes. Intui-se que, à medida em que o tamanho do espaço de projeto

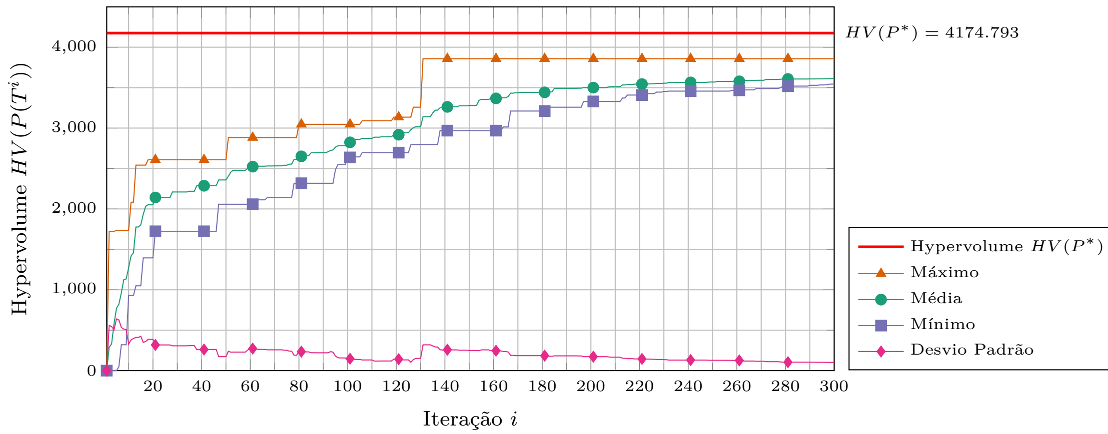


Figura 10.5.: Gráfico de convergência da métrica *hypervolume* durante as iterações de otimização do modelo do *cluster* elástico para aplicações *MapReduce*.

aumenta, menos frequentes são as arquiteturas com tempo de estabilização baixo e maior é o benefício de uso da abordagem de otimização aqui apresentada, quando comparada a explorações manuais. Estudos mais aprofundados, não realizados nesta tese, podem comprovar/refutar tal hipótese.

O *trade-off* característico entre tempo de estabilização e sobressinal, bem como entre efetividade de controle e *overhead* de controle, são também evidentes embora com espalhamento menos uniforme que aquele apresentado para o modelo do servidor *web* MIMO. Tal situação é também consequência do tamanho maior do espaço de projeto, aumentando o número de soluções não-dominadas e tornando mais difícil sua identificação. Um *Pareto-front* com um espalhamento mais uniforme pode ser produzido utilizando um número maior de iterações de otimização, pois a distância de aglomeração passa a ter atuação mais decisiva. Os demais *trade-offs* continuaram apresentando expressividade mínima ou nula.

O *trade-off* entre efetividade de controle e *overhead* de controle reflete a influência da forma de cooperação entre múltiplos *loops* nestas variáveis. Note como a cor da solução (neste caso, forma de cooperação entre múltiplos *loops*) varia de vermelho (sem cooperação) a amarelo (controle hierárquico) à medida em que sua posição varia da coordenada (0,0) – menor efetividade e menor *overhead* – para a coordenada (1,1) – máxima efetividade e máximo *overhead*. Vale relembrar que os pontos de variação da dimensão DD6 foram definidos, no Capítulo 7, em ordem crescente de efetividade e *overhead* de controle (sem cooperação, descentralização coordenada por monitoramento, descentralização totalmente coordenada, planejamento regional, controle centralizado e controle hierárquico).

Em relação à Figura 10.5, nota-se que o *hypervolume* $HV(P(T^i))$ de uma única execução de otimização converge ao longo das iterações i, em média, para $\overline{HV}(P(T^{300})) = 3612.8311$, cerca de 85.53% do *hypervolume* apresentado pelo *Pareto-front de referência* originado pelas 31 execuções. Intui-se, dessa forma, que o uso de modelos iniciais maiores faz com que a convergência para um suposto *Pareto-front* se torne mais difícil, considerando apenas uma execução de otimização. Novamente, a avaliação de como o desempenho da otimização varia, em função do tamanho do modelo inicial utilizado, não foi realizado nesta tese e é objeto de futuras investigações. Para este caso particular de otimização, 95% do valor final médio do *hypervolume* $(0.95 \cdot 3612.8311 = 3432.1895)$ é atingido na iteração de número 169. Daí para frente, a descoberta de soluções mais efetivas passa a ser mais difícil.

De forma semelhante ao estudo com o servidor *web* MIMO, investigou-se – em cada iteração i – a hipótese nula de que a média do *hypervolume* é menor ou igual a 95% do valor final do

Iteração i	$\overline{HV}(P(T^i))$	*p-value* de Anderson-Darling	*p-value* de Levene	Teste Utilizado	*p-value* (*t-test* / *Wilcoxon SR*)
...					
212	3512.200411	0.113852	0.274383	*t-test*	0.101512
213	3533.574236	0.152420	0.260664	*t-test*	**0.046263**
214	3533.574236	0.152420	0.260664	*t-test*	0.046263
215	3539.272402	0.047992	0.320262	*Wilcoxon SR*	**0.009520**
216	3539.272402	0.047992	0.320262	*Wilcoxon SR*	0.009520
...					

Tabela 10.2.: Resultados dos testes de hipótese para verificação de convergência do *hypervolume* na otimização do modelo do *cluster* elástico para aplicações *MapReduce*.

hypervolume: $H_0: \overline{HV}(P(T^i)) \leq 0.95 \cdot \overline{HV}(P(T^{300}))$. A Tabela 10.2 apresenta os resultados obtidos para as iterações 212 a 216. Os *hypervolumes* das iterações 212, 213 e 214 apresentaram *p-value* de Anderson-Darling ≥ 0.05 e *p-value* de Levene ≥ 0.05, portanto o *t-test* foi utilizado nestes casos. Para as demais iterações, o teste de *Wilcoxon Signed-Rank* foi adotado. Com um nível de significância (α) de 0.05, pode-se afirmar que o *hypervolume* de uma única execução atinge 95% do seu valor final na iteração 213 (primeiro *p-value* com valor menor que 0.05). Ou seja, de cada 100 execuções de otimização, em 95 delas o *hypervolume* terá atingindo 95% do seu valor final na 213ª iteração. Com um nível de significância de 0.01 (99% das execuções) o limite para atingir os 95% passa a ser a iteração 215 (primeiro *p-value* com valor menor que 0.01).

10.2.1.3. Discussão

O Estudo 1 teve como objetivo a análise de dois diferentes aspectos da abordagem aqui apresentada: *i)* a capacidade do espaço de projeto SA:DuSE evidenciar os *trade-offs* arquiteturais comumente envolvidos no projeto de sistemas *self-adaptive* (questão de avaliação QA1.1); e *ii)* a qualidade do *Pareto-front* encontrado em uma única execução de otimização, quando comparado a um *Pareto-front* de referência P^* (questão de avaliação QA1.2).

Em relação à questão de avaliação QA1.1, a cardinalidade dos *Pareto-fronts* de referência P^*_{SW} e P^*_C obtidos (respectivamente, 13 e 18 soluções) é uma evidência da efetividade do SA:DuSE na captura dos *trade-offs* envolvidos no projeto de sistemas *self-adaptive*. Visto que o *Pareto-front* global real não é conhecido, as cardinalidades de P^*_{SW} e P^*_C devem ser interpretadas como valores mínimos. Outras soluções não-dominadas provavelmente existem e não foram encontradas por dificuldades de convergência e/ou consequência da aleatoriedade inerente da técnica evolucionária (foco da questão de avaliação QA1.2).

A despeito disto, acredita-se que a disponibilização de 13 diferentes arquiteturas candidatas – com diferentes atendimentos de atributos de qualidade – já constitua contribuição importante. O mesmo pode ser argumentado em relação às 18 arquiteturas candidatas encontradas para o modelo do *cluster* elástico para aplicações *MapReduce*. Obviamente, mais estudos devem ser realizados para verificar a validade das conclusões aqui apresentadas quando utilizando modelos arquiteturais iniciais diferentes dos aqui adotados.

Em relação à questão de avaliação QA1.2, observa-se que uma única execução de otimização apresenta desempenho (em termos de *hypervolume*) consideravelmente satisfatório: 95.3% de aproximação ao *Pareto-front* de referência no caso do servidor *web* MIMO e 85.53% no caso do *cluster* elástico para aplicações *MapReduce*. Adicionalmente, chega-se suficientemente próximo do *hypervolume* final da otimização antes do final das 300 iterações (mais precisamente, na iteração 228 para o servidor *web* MIMO e 215 para o *cluster* elástico para aplicações *MapReduce*).

Estes dados contribuem para aumentar a confiança em relação à qualidade das arquiteturas candidatas (*Pareto-front*) entregues por uma única execução da otimização. Novamente, outros estudos devem investigar os resultados obtidos para modelos arquiteturais iniciais diferentes dos aqui adotados.

10.2.2. Estudo 2: Acurácia das Métricas de Qualidade do SA:DuSE

Este estudo teve como objetivo investigar com qual acurácia os valores de atributos de qualidade obtidos pelas métricas preditivas do SA:DuSE (tempo médio de estabilização e sobressinal médio) são de fato observados em implementações reais das arquiteturas candidatas projetadas pelas buscas (questão de avaliação QA2). Para isso, duas arquiteturas candidatas do *Pareto-front* de referência $P^*(c_1$ e $c_2)$ e uma arquitetura candidata dominada (c_d) – de desempenho pior que c_1 e c_2 em ambas as métricas – foram implementadas em protótipos reais.

O objetivo foi comparar os valores de atributos de qualidade previstos pelo SA:DuSE com aqueles efetivamente apresentados pelos protótipos reais. Espera-se que a implementação das arquiteturas c_1 e c_2 apresentem o *trade-off* entre tempo médio de estabilização e sobressinal médio, com valores próximos daqueles apresentados na otimização. Adicionalmente, a implementação de c_d deve apresentar tempo de estabilização menor e sobressinal maior que aqueles apresentados por c_1 e c_2.

Os detalhes deste estudo utilizando arquiteturas do servidor *web* MIMO são apresentados na Seção 10.2.2.1. A Seção 10.2.2.2 apresenta os resultados obtidos com arquiteturas do *cluster* elástico para aplicações *MapReduce*. Uma discussão geral do Estudo 2 é apresentada na Seção 10.2.2.3.

10.2.2.1. Servidor Web MIMO

A avaliação da acurácia das métricas de qualidade do SA:DuSE, para o modelo do servidor *web* MIMO, foi realizada através de implementações de *loops* de controle para o *Apache HTTP Server* (`httpd`) [100]. O `httpd` é uma implementação *open source* de um servidor *web*, desenvolvido há 17 anos pela *Apache Foundation*[4] e amplamente utilizado tanto na academia quanto na indústria.

A arquitetura do `httpd` foi projetada de modo a atender requisitos de portabilidade, desempenho e extensibilidade. Um amplo conjunto de parâmetros, configurados em arquivos texto, permite a seleção de implementações particulares de modelos de concorrência (ex: uma *thread* por requisição ou *thread pool*), gerência de I/O (ex: síncrono ou assíncrono) e protocolos de comunicação suportados (ex: HTTP, HTTPS), dentre muitas outras características. Dessa forma, administradores podem configurar o `httpd` de acordo com as características de carga (*workload*) e natureza dos dados do cenário em questão.

O `httpd` é caracterizado por uma arquitetura modular, onde funcionalidades mínimas – disponibilizadas pelo núcleo do servidor – são ampliadas através da instalação de módulos. Módulos podem suportar novos protocolos de comunicação, acrescentar novos parâmetros de configuração e implementar novos modelos de concorrência, dentre outros aspectos. Dessa forma, o `httpd` define um *application framework* que viabiliza a implementação produtiva de qualquer serviço de rede, não somente servidores *web*. Parâmetros de configuração podem ser modificados e módulos podem ser inseridos/removidos sem requerer a interrupção do serviço. Uma documentação completa das decisões arquiteturais e aspectos de implementação do `httpd` pode ser encontrada em [99].

[4] `http://www.apache.org`.

Configuração da Medição. Para a realização do Estudo 2 com o modelo do servidor *web* MIMO, o `httpd` 2.4 foi instalado em uma máquina Intel Core i5, com 8GB de memória RAM, executando o sistema operacional GNU/Linux, *kernel* 3.17. O parâmetro `KeepAliveTimeout` é definido no arquivo `conf/extra/httpd-default.conf` e apresenta valor *default* de 5s. O parâmetro `MaxRequestWorkers`, por sua vez, apresenta valor *default* de 400 *threads* e é definido no arquivo `conf/extra/httpd-mpm.conf`. O modelo de concorrência utilizado foi o *prefork*, caracterizado pela utilização de um processo por requisição (até o limite máximo definido por `MaxRequestWorkers`) e pela criação antecipada de um número fixo de processos durante a inicialização do servidor (definido pelo parâmetro `StartServers`).

Para a geração das requisições ao servidor *web*, a ferramenta `httperf` [220] versão 0.9 foi instalada em uma outra máquina de configuração igual à do servidor. O `httperf` permite simular sessões de requisições HTTP a uma determinada frequência. Por exemplo, o comando "`httperf --server 192.168.1.2 --wsess=720,50,2 --burst-len 5 --rate 40`" executa 720 sessões HTTP a uma taxa de 40 sessões/s, produzindo um teste de 18s de duração. Em cada sessão, 50 requisições HTTP são realizadas em rajadas (*bursts*) de 5 requisições cada uma. As 10 rajadas são espaçadas por um intervalo de 2s. Com isso, é possível simular o cenário comum onde um usuário realiza diversas solicitações a páginas *web* (com as rajadas representando requisições de outros conteúdos associados à página como, por exemplo, imagens), espaçadas por um determinado intervalo de tempo (*thinking time*). O documento acessado pelo `httperf` é definido no parâmetro `--uri`, cujo valor *default* é o documento raiz do servidor (`index.html` ou `index.php`).

Parâmetro do `httperf`	Distribuição Utilizada	Parâmetros da Distribuição
Tempo entre sessões (s)	Exponencial	$\mu = 0.025$
Requisições por sessão	LogNormal	$\mu = 50$, $\sigma = 3$
Requisições por rajadas	Normal	$\mu = 5$, $\sigma = 0.5$
Tempo entre rajadas (s)	LogNormal	$\mu = 2$, $\sigma = 0.5$

Tabela 10.3.: Parâmetros do `httperf` utilizados na experimentação com o servidor *web* `httpd`.

Para este experimento, uma página PHP que realiza a ordenação de um vetor de 210 posições foi utilizada, de modo a gerar alguma utilização de CPU por requisição. Os parâmetros do `httperf` (apresentados na Tabela 10.3) foram configurados de acordo com o modelo WAGON, proposto por Liu et al. em [202]. O tamanho do vetor sendo ordenado e os valores apresentados na Tabela 10.3 foram escolhidos experimentalmente, de modo a produzir uma utilização média da CPU do servidor de 75%, número médio de *threads worker* igual a 1024 e consumo médio de memória do servidor de 33%. Tais valores foram utilizados como valores máximos destas variáveis. Nestas configurações o servidor trabalha sem saturação, apresentando vazão igual à taxa de requisição.

A Figura 10.6 apresenta a dinâmica do servidor *web* (em malha aberta – sem controladores atuando) para uma execução do `httperf` com os parâmetros acima descritos, em um teste com duração de 180s. As Figuras 10.6(a) e 10.6(b) apresentam como a utilização de CPU, consumo de memória e número de *threads worker* variam em função – respectivamente – dos valores atribuídos aos parâmetros `MaxRequestWorkers` e `KATimeout`. Nas medições apresentadas na Figura 10.6(a) o parâmetro `KATimeout` foi mantido em seu valor *default* (5s), enquanto naquelas ilustradas na Figura 10.6(b), `MaxRequestWorkers` assumiu o valor máximo aqui adotado (1024). As amostras de consumo de memória e número de *threads worker* foram obtidas a cada segundo. Para a utilização de CPU, adotou-se a média de cinco valores coletados em intervalos de 5s, de modo a filtrar o forte componente estocástico presente.

Note que o parâmetro `MaxRequestWorkers` impõe um limite ao número de *threads worker* apresentados nos gráficos da terceira coluna da Figura 10.6 e possui forte impacto na utilização de

CPU e consumo de memória. Um valor de `MaxRequestWorkers` igual a 1024 implica, em regime estacionário, em uma utilização média de CPU de cerca de 68% e em um consumo médio de memória de 33%. Com `MaxRequestWorkers` igual a 512 e 256, a utilização de CPU cai para cerca de 45% e 25% e o consumo de memória para 27% e 22%, respectivamente.

Obviamente, qualquer valor de `MaxRequestWorkers` menor que 1024 implica na diminuição do tempo de resposta experimentado pelo cliente. Vale ressaltar que este experimento não teve como objetivo controlar o tempo de resposta e sim a utilização de CPU e consumo de memória percebidos no servidor. Conforme apresentado na Figura 10.6(b), o parâmetro `KATimeout` exerce uma menor influência na utilização de CPU e consumo de memória.

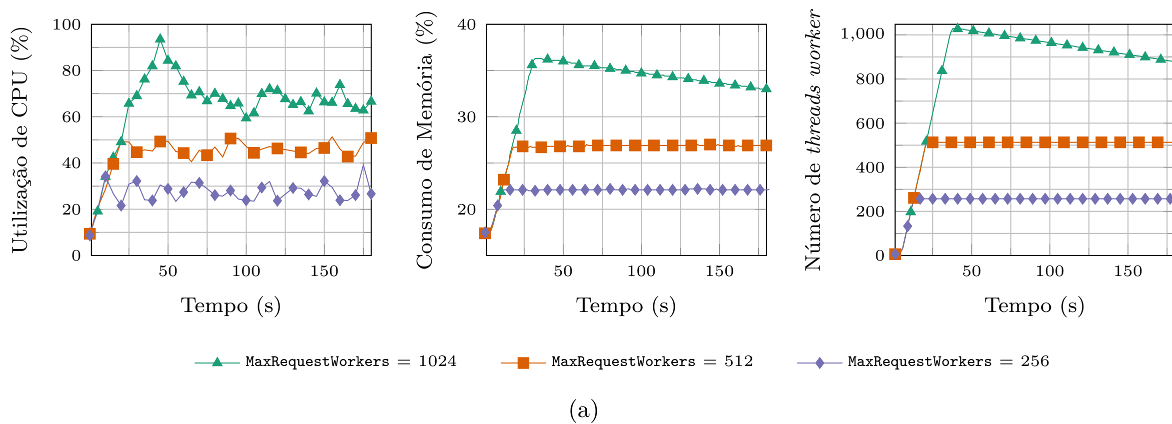
Os sensores de utilização de CPU e consumo de memória foram implementados em *Shell Script*, com base nos comandos `top` e `ps` do sistema operacional Linux:

Utilização de CPU:

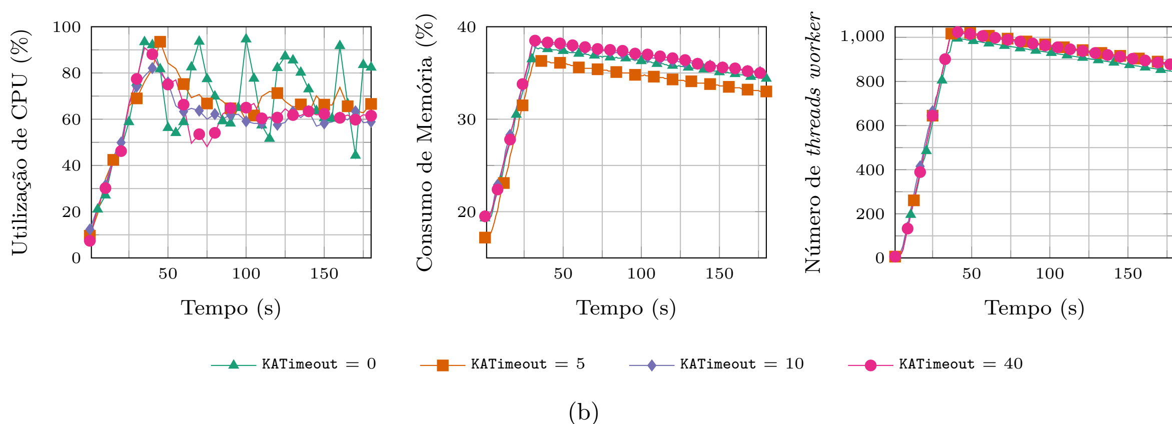
```
top -b -n5 -d1 | grep "Cpu" | tail -n1 | awk '{print $2}' | awk -F"/" '{print $1+$2}'
```

Consumo de memória:

```
top -b -n1 -d1 | grep "GiB Mem" | tail -n1 | cut -d":" -f2 | cut -d"/" -f1
```



(a)



(b)

Figura 10.6.: Dinâmica do `httpd`, em malha aberta, para a utilização de CPU, número de *threads worker* e consumo de memória, em função dos parâmetros `MaxRequestWorkers` (a) e `KATimeout` (b).

Os atuadores para manipulação dos parâmetros `MaxRequestWorkers` e `KATimeout` foram implementados também em *Shell Script*, utilizando *templates* de arquivos de configuração:

Trecho do *template* do arquivo de configuração `conf/extra/http-mpm.tpl`:

```
<IfModule mpm_prefork_module>
    MinSpareServers 5
    MaxSpareServers 10
    MaxRequestWorkers {% MaxRW %}
</IfModule>
```

Trecho do *template* do arquivo de configuração `conf/extra/http-default.tpl`:

```
Timeout 60
KeepAlive On
MaxKeepAliveRequests 100
KeepAliveTimeout {% KA %}
```

***Script* de atuação ($1 - novo valor de `MaxRequestWorkers`; $2 - novo valor de `KATimeout`):**

```
cat http-mpm.tpl | sed "s/{% MaxRW %}/$1/g" > /etc/httpd/conf/extra/http-mpm.conf
cat http-default.tpl | sed "s/{% KA %}/$2/g" > /etc/httpd/conf/extra/http-default.conf
/usr/bin/apachectl -k graceful
```

Concluídas as instrumentações necessárias ao monitoramento e controle do `httpd`, os procedimentos de ensaio ao degrau (vide Seção 3.3.4.2, pág. 53) foram realizados com um período de amostragem de 5s. Com isso, obteve-se os parâmetros que descrevem a dinâmica do sistema e criou-se o modelo arquitetural inicial devidamente anotado. Uma execução de otimização utilizando os mesmos valores de parâmetros descritos no Estudo 1 para o servidor *web* MIMO foi realizada. Após as 300 interações, duas arquiteturas do *Pareto-front* (c_1 e c_2) e uma arquitetura dominada (c_d) foram selecionadas. Estas soluções são apresentadas na Tabela 10.4.

c_i	Dimensão de Projeto	Ponto de Variação `IKATimeout`	Ponto de Variação `IMaxRW`	$\hat{\overline{K_s^i}}$	$\hat{\overline{M_p^i}}$
c_1	DD2 (SISO x MIMO) DD3 (Lei de Controle) DD4 (Técnica de Sintonia) DD5 (Robustez de Controle) DD6 (Forma de Cooperação)	VP21 (SISO) VP33 (PI) VP41 (CHR-0OS-DR) VP51 (Ganho Fixo) VP61 (Sem Cooperação)	VP21 (SISO) VP35 (PID) VP46 (CC) VP51 (Ganho Fixo) VP61 (Sem Cooperação)	86.3	11.89
c_2	DD2 (SISO x MIMO) DD3 (Lei de Controle) DD4 (Técnica de Sintonia) DD5 (Robustez de Controle) DD6 (Forma de Cooperação)	VP21 (SISO) VP35 (PID) VP43 (CHR-20OS-DR) VP51 (Ganho Fixo) VP61 (Sem Cooperação)	VP21 (SISO) VP33 (PI) VP46 (CC) VP51 (Ganho Fixo) VP61 (Sem Cooperação)	55.0	19.31
c_d	DD2 (SISO x MIMO) DD3 (Lei de Controle) DD4 (Técnica de Sintonia) DD5 (Robustez de Controle) DD6 (Forma de Cooperação)	VP22 (MIMO) VP37 (DS Feedback) VP47 (LQR) VP51 (Ganho Fixo) VP61 (Sem Cooperação)	VP22 (MIMO) VP37 (DS Feedback) VP47 (LQR) VP51 (Ganho Fixo) VP61 (Sem Cooperação)	111.0	29.76

Tabela 10.4.: Arquiteturas implementadas nos protótipos reais de sistemas gerenciadores para o `httpd`.

A arquitetura c_1 utiliza um controlador SISO em cada porta controlável `IKATimeout` e `IMaxRW`. O controlador da porta `IKATimeout` utiliza a lei de controle PI, sintonizado com a técnica CHR-0OS-DR. Já o controlador da porta `IMaxRW` adota a lei de controle PID, sintonizado com a técnica CC. Tais decisões produzem *feedback control loops* com tempo médio de estabilização previsto ($\hat{\overline{K_s^1}}$) de 86.3s e sobressinal médio previsto ($\hat{\overline{M_p^1}}$) de 11.89%.

A arquitetura c_2 também utiliza dois controladores SISO, porém adotando lei de controle PID com sintonia CHR-20OS-DR para a porta `IKATimeout` e lei de controle PI com sintonia CC para a porta `IMaxRW`. Esta arquitetura é tão efetiva quando c_1 pois representa uma alternativa onde obtém-se um tempo de estabilização menor ($\hat{\overline{K_s^2}} = 55.0 < \hat{\overline{K_s^1}}$) às custas de um maior sobressinal ($\hat{\overline{M_p^2}} = 19.31 > \hat{\overline{M_p^1}}$).

Finalmente, arquitetura c_3 utiliza um único controlador MIMO para a gerência das duas portas controláveis. A lei de controle adotada é *Dynamic State Feedback Control* com sintonia realizada pela técnica LQR. Esta solução é dominada por ambas as arquiteturas c_1 e c_2, pois apresenta o maior tempo de estabilização ($\hat{\overline{K_s^3}} = 111.0 > \hat{\overline{K_s^1}} > \hat{\overline{K_s^2}}$) e também o maior sobressinal ($\hat{\overline{M_p^3}} = 29.76 > \hat{\overline{M_p^2}} > \hat{\overline{M_p^2}}$). A solução c_3, portanto, não representa uma arquitetura efetiva de controle e foi aqui utilizada somente para verificar se tais propriedades são empiricamente constatadas em protótipos reais.

Os controladores PI, PID e *Dynamic State Feedback* foram implementados em *Shell Script* e executados na mesma frequência com a qual o ensaio ao degrau foi realizado (5s). Cada um dos três sistemas finais resultantes (servidor *web* + $c_{1,2,d}$) foi executado e monitorado por 360s. Nos primeiros 120s, os valores de referência para utilização de CPU e consumo de memória foram, respectivamente, 25% e 22%. No 121º segundo, tais valores foram modificados para 68% e 33%, respectivamente.

Resultados Obtidos. As Figuras 10.7(a), 10.7(b) e 10.7(c) apresentam os resultados de 31 replicações deste experimento utilizando, respectivamente, os protótipos das arquiteturas c_1, c_2 e c_d.

Cada figura apresenta, à esquerda, a utilização de CPU ao longo do tempo e, à direita, o consumo de memória ao longo do tempo. O sinal de referência, tanto para utilização de CPU quanto para consumo de memória, é representado pela linha tracejada sem marcadores (em preto). Os valores máximos, médios e mínimos de utilização de CPU e consumo de memória obtidos nas 31 replicações são também apresentados (ver legenda da figura). A dinâmica prevista na modelagem realizada pelo SA:DuSE é descrita pela curva contínua sem marcadores (em vermelho).

São apresentados ainda, em cada gráfico, os valores previstos pelo SA:DuSE para o tempo médio de estabilização ($\hat{\overline{K_s}}$) e sobressinal médio ($\hat{\overline{M_p}}$), bem como a média obtida para estas variáveis, nas 31 replicações ($\overline{K_s}$ e $\overline{M_p}$). Os valores previstos e medidos são apresentados para cada variável controlada de cada arquitetura candidata. Por exemplo, $\hat{\overline{K_{s_c}^2}}$ representa o tempo médio de estabilização da utilização de CPU, previsto para a arquitetura candidata c_2. $\overline{M_{p_m}^d}$ representa o sobressinal médio do consumo de memória, medido na implementação da arquitetura candidata c_d. O valor $\hat{\overline{K_s^i}}$ apresentado na Tabela 10.4 é a média aritmética de $\hat{\overline{K_{s_c}^i}}$ e $\hat{\overline{K_{s_m}^i}}$. O mesmo vale para $\hat{\overline{M_p^i}}$.

Análise dos Dados. Algumas observações podem ser realizadas sobre os dados apresentados na Figura 10.7. Primeiro, nota-se que o *trade-off* característico entre tempo de estabilização e sobressinal foi de fato observado nos protótipos. Note como o controlador utilizado na arquitetura c_2 produz um tempo de estabilização menor que aquele gerado pela arquitetura c_1, às custas de um maior sobressinal. A escolha por c_1 ou c_2 depende de qual destes dois atributos de qualidade deseja-se priorizar.

Segundo, visto que todos os controladores adotados pelas três arquiteturas (PI, PID e *Dynamic State Feedback Control*) utilizam fator integral, o erro em regime estacionário (e_{ss}) observado em todas as medições foi zero (obviamente descartando alguma flutuação estocástica). Terceiro, todos os sistemas finais (servidor `httpd` + controlador) apresentaram comportamento BIBO-estável, conforme garantido pelas técnicas de sintonia capturadas no SA:DuSE.

Quarto, a efetividade relativa apresentada pelos protótipos das arquiteturas c_1, c_2 e c_d estão compatíveis com as informações previstas pela otimização. De fato, o protótipo da arquitetura c_1 apresentou sobressinal menor que aquele apresentado pelo protótipo da arquitetura c_2. Por

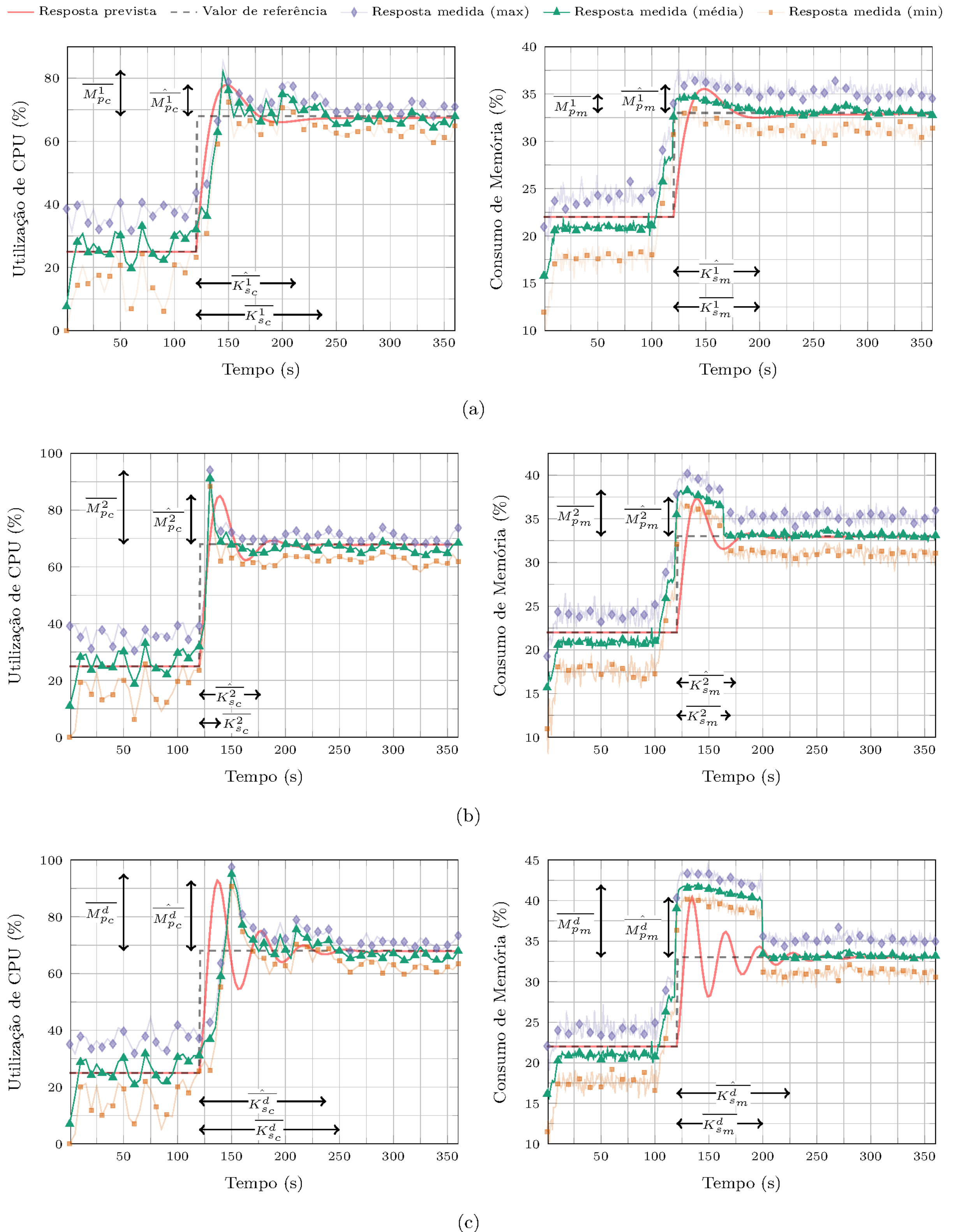


Figura 10.7.: Dinâmica do `httpd`, em malha fechada, utilizando os controladores que implementam as arquiteturas candidatas c_1 (a), c_2 (b) e c_d (c).

outro lado, o protótipo da arquitetura c_2 apresentou tempo de estabilização menor que aquele apresentado pelo protótipo da arquitetura c_1. O protótipo da arquitetura c_d apresentou tempo de estabilização e sobressinal maiores que aqueles apresentados pelos protótipos das arquiteturas

c_1 e c_2.

Finalmente, mas não menos importante, discrepâncias foram observadas entre os valores previstos e medidos de tempo de estabilização e sobressinal. Conforme observado no gráfico da esquerda na Figura 10.7(a), a implementação da arquitetura c_1 apresentou – para o controle da utilização de CPU – um sobressinal medido ($\overline{M_{p_c}^1} = 19.11\%$) um pouco maior que o sobressinal previsto ($\hat{\overline{M_{p_c}^1}} = 16.17\%$). O tempo de estabilização medido ($\overline{K_{s_c}^1} = 120$s) também foi maior que o valor previsto ($\hat{\overline{K_{s_c}^1}} = 92.4$s). Os protótipos das outras arquiteturas apresentam discrepâncias semelhantes.

Algumas razões para tais diferenças podem ser identificadas. Primeiro, a presença de um componente estocástico forte – conforme frequentemente observado em medições de utilização de CPU – pode causar tais discrepâncias caso não seja devidamente considerado. O uso de técnicas mais avançadas de filtragem de sinais pode contribuir para reduzir esta influência. Segundo, todas as técnicas de controle capturadas no SA:DuSE assumem como premissa que o sistema dinâmico a ser controlado é linear e invariante no tempo. Não-linearidades na dinâmica do `httpd` podem ter causado tais diferenças. Técnicas para controle adaptativo são frequentemente utilizadas para minimizar estes problemas. Terceiro, os modelos FOPDT e de espaço de estados produzidos pelo ensaio ao degrau realizado neste experimento são de primeira ordem e, portanto, provavelmente não capturam as nuances de dinâmica necessárias para previsões mais acuradas. Modelos de mais alta ordem podem trazer resultados melhores. Tais aspectos não foram analisados nesta tese e são foco de trabalhos futuros.

Para melhor analisar as discrepâncias apresentadas entre valores previstos e medidos, investigou-se – com base nas 31 replicações de execução dos protótipos de cada arquitetura candidata c_i – as seguintes hipóteses:

$$\begin{array}{ll} H_{0_K}^i: \ \overline{K_s^i} > \hat{\overline{K_s^i}} \\ H_{1_K}^i: \ \overline{K_s^i} \leq \hat{\overline{K_s^i}} \end{array} \quad \text{e} \quad \begin{array}{ll} H_{0_M}^i: \ \overline{M_p^i} > \hat{\overline{M_p^i}} \\ H_{1_M}^i: \ \overline{M_p^i} \leq \hat{\overline{M_p^i}} \end{array} \tag{10.3}$$

Para este teste de hipótese, o nível de significância (α) desejado não foi antecipadamente definido pois deseja-se saber o maior nível de confiança ($\gamma = 1 - \alpha$) que faz com que H_0 seja rejeitada. Este número informa com qual confiança pode-se afirmar que os valores medidos ($\overline{K_s^i}$ e $\overline{M_p^i}$) são menores ou iguais àqueles previstos ($\overline{K_s^i}$ e $\overline{M_p^i}$). Visto que o *p-value* do teste utilizado representa o menor valor de α que faz com que H_0 seja rejeitada, o nível de confiança máximo é dado por $1-$*p-value*. Quanto maior for este valor, mais confiança tem-se que as discrepâncias apresentadas no experimento foram devidas ao acaso.

Os mesmos procedimentos realizados no Estudo 1 para verificação das premissas para uso de testes paramétricos foram realizados. A Tabela 10.5 apresenta os resultados obtidos. Em relação ao tempo médio de estabilização $\overline{K_s}$, a arquitetura candidata com maior *p-value* foi a c_2 (0.291644), indicando que a confiança que podemos ter na acurácia destas predições é menor que $70.83\% = 100 \cdot (1 - 0.291644)$. Já para o sobressinal médio $\overline{M_p}$, o maior *p-value* foi também apresentado pela arquitetura c_2 (0.255307), o que equivale a uma confiança menor que 74.47% $= 100 \cdot (1 - 0.255307)$.

10.2.2.2. Cluster Elástico para Aplicações MapReduce

A avaliação da acurácia das métricas de qualidade do SA:DuSE, em relação ao modelo do *cluster* elástico para aplicações *MapReduce*, foi realizada através de implementações de *loops* de controle para o *Apache Hadoop* [320]. O *Hadoop* é uma implementação *open source* de uma plataforma

c_i	$\overline{K_s^i}$	***p-value* de Anderson-Darling**	***p-value* de Levene**	**Teste Utilizado**	***p-value* (*t-test* / *Wilcoxon SR*)**
c_1	100.0	0.177067	0.863098	*t-test*	0.207467
c_2	35.0	0.040538	0.695380	*Wilcoxon SR*	0.291644
c_d	105.0	0.037652	0.487348	*Wilcoxon SR*	0.238657
			***p-value* máximo**		0.291644
			γ máximo		0.708356

c_i	$\overline{M_p^i}$	***p-value* de Anderson-Darling**	***p-value* de Levene**	**Teste Utilizado**	***p-value* (*t-test* / *Wilcoxon SR*)**
c_1	12.58	0.287484	0.476245	*t-test*	0.241646
c_2	27.42	0.394569	0.387601	*t-test*	0.255307
c_d	35.69	0.287606	0.310452	*t-test*	0.195475
			***p-value* máximo**		0.255307
			γ máximo		0.744693

Tabela 10.5.: Resultados dos testes de hipótese para verificação da acurácia das predições de tempo de resposta e sobressinal no `httpd`.

para armazenamento e processamento distribuído de grandes bases de dados, criado em 2005 e atualmente mantida pela *Apache Foundation*.

O *Hadoop* disponibiliza duas tecnologias principais: um sistema de arquivos distribuído (HDFS – *Hadoop Distributed File System*) e uma plataforma para computação distribuída (YARN – *Yet Another Resource Negotiator*). O HDFS provê suporte escalável e tolerante a falhas para armazenamento de grandes bases de dados. O YARN suporta uma variedade de modelos para computação paralela, dentre eles o *MapReduce*.

O *MapReduce* [76] é um estilo arquitetural e um modelo computacional para processamento distribuído, amplamente utilizado na análise de grandes bases de dados. O *MapReduce* assume que os dados a serem processados estão armazenados em um sistema de arquivos distribuído e disponibiliza um *framework* que facilita o processamento paralelo e a posterior consolidação (redução) dos dados de saída gerados. A arquitetura é inerentemente escalável e com bom suporte a falhas em nós do *cluster*.

O HDFS representa grandes conjuntos de dados sob a forma de múltiplos blocos, armazenados de forma distribuída nas máquinas que compõem o *cluster*. Ao iniciar a execução de um *job*, uma tarefa de *map* é executada para cada bloco do conjunto e tem como objetivo produzir uma saída parcial do *job*, referente ao bloco em questão. Este procedimento ocorre em paralelo, nas múltiplas máquinas do *cluster*.

No *MapReduce*, os dados de entrada e saída das tarefas são representados como pares do tipo <chave, valor>. À medida em que as tarefas de *map* concluem o processamento, seus dados de saída são passados como entrada a uma ou mais tarefas de *reduce*. O objetivo da tarefa de *reduce* é manter os dados ordenados pela chave e, após a conclusão das tarefas de *map*, executar um procedimento de consolidação dos dados, definido pelo programador do *job*. As decisões adotadas no *MapReduce* viabilizam o processamento massivo de dados da ordem de centenas de *petabytes*, em *clusters* formados por milhares de máquinas.

A Figura 10.8 apresenta os principais serviços disponibilizados pelo HDFS e YARN, bem como aqueles presentes durante a execução de um *job MapReduce*. O HDFS é formado por dois serviços

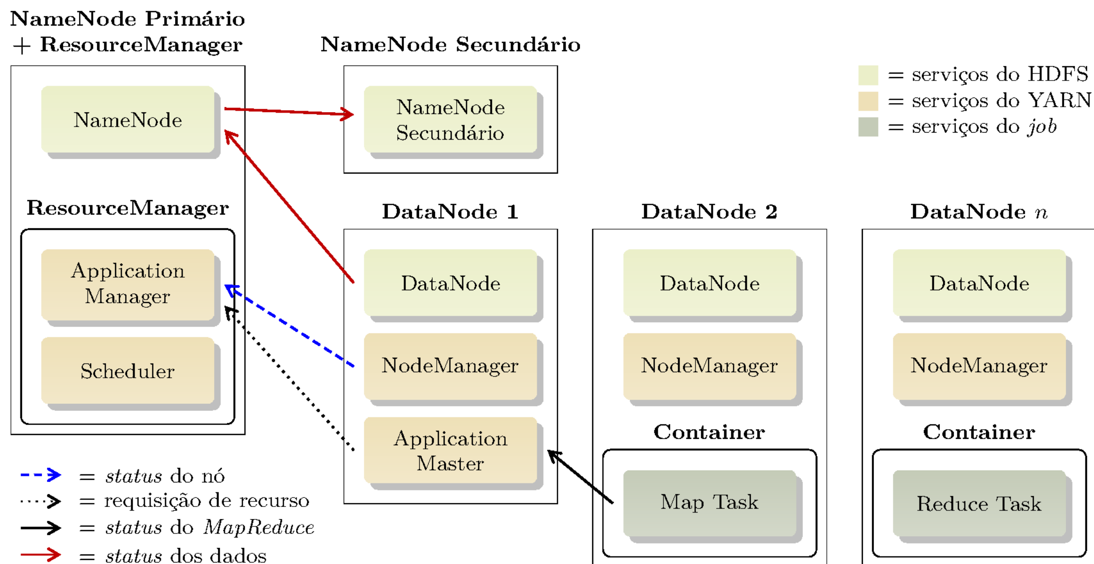


Figura 10.8.: Componentes fundamentais do HDFS e do YARN (*Hadoop* 2.3.0).

básicos: o `NameNode` e o `DataNode`. O `NameNode` é responsável pelo gerenciamento global dos blocos e réplicas que representam arquivos armazenados no sistema de arquivos distribuído. É o serviço que é consultado quando deseja-se recuperar um arquivo do sistema, através da coleta e integração dos diversos blocos armazenados de forma distribuída no *cluster*.

O `DataNode` é um serviço executado em cada nó de armazenamento do *cluster* e tem como objetivo o gerenciamento dos dados que residem naquela máquina. O `DataNode` envia periodicamente ao `NameNode` informações sobre o *status* dos dados armazenados. Um `NameNode` secundário é utilizado para suportar falhas no `NameNode` primário, evitando a inviabilização de todo o *cluster*. Recursos para federação de `NameNodes`, com o objetivo de melhorar a escalabilidade do serviço, estão também disponíveis no *Hadoop*.

O YARN é formado por dois serviços principais: o `ResourceManager` e o `NodeManager`. O `ResourceManager`, por sua vez, possui duas atribuições primárias: coordenar a execução de *jobs* no *cluster* e alocar recursos (memória, CPU e facilidades de comunicação) a *jobs* em execução. Tais atividades são executadas pelos componentes `ApplicationManager` e `Scheduler`, respectivamente. O `NodeManager` é um cliente do `ResourceManager`, presente em cada nó do *cluster* que está habilitado a executar tarefas de *map* ou *reduce*.

Ao receber uma solicitação de execução de *job*, o YARN seleciona uma máquina do *cluster* para hospedar a execução do `ApplicationMaster` – serviço de coordenação daquele *job* em particular. O `ApplicationMaster` é responsável pela solicitação, ao `ResourceManager`, dos recursos necessários à execução do *job*. O `Scheduler` solicita a alocação de contêineres de execução de tarefas de *map*, preferencialmente naquelas máquinas que contêm blocos do arquivo de entrada a ser processado (minimizando o tráfego de dados na rede). Dependendo do número de tarefas de *reduce* configurado para o *job*, uma ou mais máquinas do *cluster* são selecionadas para execução destas tarefas.

O desempenho de *jobs* executados no *Hadoop* é influenciado pela natureza e organização dos dados de entrada e também por valores atribuídos a mais de 190 parâmetros, referentes ao HDFS, YARN e *MapReduce*. Embora o *Hadoop* defina valores *default* para estes parâmetros, estudos [28, 157, 161] demonstram que ganhos de desempenho de até 50% podem ser obtidos com configurações mais especializadas. Entretanto, conhecer os parâmetros e os valores ótimos para um determinado *job* requer amplo conhecimento da infraestrutura e do modo de operação do HDFS, do YARN e do *MapReduce*. Estratégias adaptativas para auto-configuração, como as

abordadas neste trabalho, são, portanto, de fundamental utilidade nestes casos.

Configuração da Medição. Para a realização do Estudo 2 com o modelo do *cluster* elástico para aplicações *MapReduce*, o *Hadoop* v2.4 foi instalado em um *cluster* formado por 10 máquinas atuando como `DataNodes + NodeManagers` e uma máquina atuando como `NameNode + ResourceManager`. Todos os serviços foram executados em máquinas Intel QuadCore, com 4GB de memória RAM, 500GB de disco, utilizando o sistema operacional GNU/Linux, *kernel* 3.16.

O experimento teve como objetivo regular a vazão total V_t observada, no *cluster*, durante o processamento de um determinado *job*. Para isso, *feedback control loops* locais a cada máquina atuando como `DataNodes` + `NodeManagers` foram implantados. O *feedback control loop* global para controle do número de máquinas presentes no *cluster* não foi adotado neste experimento. O *job* em execução durante as adaptações foi o `TeraSort`, um *benchmark* disponibilizado pelo *Hadoop* e comumente utilizado na análise do desempenho dos *clusters* instalados. O `TeraSort` permite a geração, armazenamento no HDFS e ordenação via *MapReduce* de um conjunto arbitrário de dados, com tamanho definido pelo usuário (originalmente 1TB em grandes *clusters*).

Os *feedback control loops* locais regulavam a vazão parcial (V_t/número de máquinas no *cluster*) e o consumo de memória através de dois parâmetros: `mapreduce.tasktracker.map.tasks.maximum` e `mapreduce.task.io.sort.factor`. O parâmetro `tasks.maximum` indica o número máximo de tarefas de *map* executadas simultaneamente na máquina em questão. Já o parâmetro `sort.factor` regula o tamanho do *buffer* utilizado, pelas tarefas de *reduce*, nas operações de ordenação de chaves. Obviamente, estes parâmetros têm considerável impacto, respectivamente, na vazão parcial e no consumo de memória observados em cada máquina do *cluster*. Os *feedback control loops* locais atuaram com um período de amostragem de 5s.

O sensor de consumo de memória foi implementado de forma similar ao apresentado para o caso do servidor *web* MIMO. Já o sensor de vazão parcial de cada máquina foi implementado com base na infraestrutura de métricas disponibilizada pelo *Hadoop*. As configurações das métricas são armazenadas no arquivo `conf/hadoop-metrics.properties`:

Trecho do arquivo `conf/hadoop-metrics.properties`:

```
mapred.class=org.apache.hadoop.metrics.file.FileContext
mapred.period=5
mapred.fileName=/tmp/mrmetrics.log
```

A configuração acima instala um consumidor de métricas com armazenamento em arquivo (classe `FileContext`, disponibilizada pelo *Hadoop*), período de coleta de 5s (parâmetro `mapred.period`) e local de armazenamento (parâmetro `mapred.fileName`) igual ao arquivo `/tmp/mrmetrics.log`.

Trecho do arquivo de métricas gerado (`/tmp/mrmetrics.log`):

```
mapred.job:  counter=Combine input records, ..., sessionId=, user=philip, value=0.0
mapred.job:  counter=Map output records, ..., sessionId=, user=philip, value=20.0
mapred.job:  counter=Launched reduce tasks, ..., sessionId=, user=philip, value=1.0
```

O sensor de vazão parcial, desenvolvido em *Shell Script*, obtém o valor do contador `"Map output records"` apresentado no arquivo acima. Este contador informa que 20 registros (pares <chave, valor>) de saída foram produzidos, pelas tarefas de *map* em execução, no último intervalo de 5s analisado.

Os atuadores para manipulação dos parâmetros `tasks.maximum` e `sort.factor` foram implementados em *Shell Script*, utilizando o mesmo mecanismo de *templates* adotado no caso do servidor *web* MIMO:

Trecho do *template* do arquivo de configuração `conf/mapred-site.tpl`:

```
<property>
    <name>mapreduce.tasktracker.map.tasks.maximum</name>
    <value>{% MaxMap %}</value>
</property>
<property>
    <name>mapreduce.task.io.sort.factor</name>
    <value>{% SortF %}</value>
</property>
```

***Script* de atuação ($1 - novo valor de `tasks.maximum`; $2 - novo valor de `sort.factor`):**

```
cat mapred-site.tpl | sed "s/{% MaxMap %}/$1/g"
                    | sed "s/{% SortF %}/$2/g" > /etc/hadoop/conf/mapred-site.conf
/usr/bin/systemctl restart hadoop-nodemanager.service
```

Concluídas as instrumentações necessárias ao monitoramento e controle do *Hadoop*, os procedimentos de ensaio ao degrau foram realizados com um período de amostragem de 5s. Os parâmetros que descrevem a dinâmica do sistema foram obtidos e criou-se o modelo arquitetural inicial devidamente anotado. Uma execução de otimização utilizando os mesmos valores de parâmetros descritos no Estudo 1 para o *cluster* elástico para aplicações *MapReduce* foi realizada. Após as 300 interações, duas arquiteturas do *Pareto-front* (c_1 e c_2) e uma arquitetura dominada (c_d) foram selecionadas. Estas soluções são apresentadas na Tabela 10.6.

c_i	Dimensão de Projeto	Ponto de Variação `IMaxMap`	Ponto de Variação `ISortFactor`	$\hat{\overline{K_s^i}}$	$\hat{\overline{M_p^i}}$
	DD2 (SISO x MIMO)	VP21 (SISO)	VP21 (SISO)		
	DD3 (Lei de Controle)	VP35 (PID)	VP33 (PI)		
c_1	DD4 (Técnica de Sintonia)	VP41 (CHR-0OS-DR)	VP45 (ZN)	132.5	14.7
	DD5 (Robustez de Controle)	VP51 (Ganho Fixo)	VP51 (Ganho Fixo)		
	DD6 (Forma de Cooperação)	VP61 (Sem Cooperação)	VP61 (Sem Cooperação)		
	DD2 (SISO x MIMO)	VP21 (SISO)	VP21 (SISO)		
	DD3 (Lei de Controle)	VP33 (PI)	VP35 (PID)		
c_2	DD4 (Técnica de Sintonia)	VP43 (CHR-20OS-DR)	VP42 (CHR-0OS-RT)	85.6	22.3
	DD5 (Robustez de Controle)	VP51 (Ganho Fixo)	VP51 (Ganho Fixo)		
	DD6 (Forma de Cooperação)	VP61 (Sem Cooperação)	VP61 (Sem Cooperação)		
	DD2 (SISO x MIMO)	VP22 (MIMO)	VP22 (MIMO)		
	DD3 (Lei de Controle)	VP36 (SS Feedback)	VP36 (SS Feedback)		
c_d	DD4 (Técnica de Sintonia)	VP47 (LQR)	VP47 (LQR)	197.5	38.1
	DD5 (Robustez de Controle)	VP51 (Ganho Fixo)	VP51 (Ganho Fixo)		
	DD6 (Forma de Cooperação)	VP61 (Sem Cooperação)	VP61 (Sem Cooperação)		

Tabela 10.6.: Arquiteturas implementadas nos protótipos reais de sistemas gerenciadores para o *Hadoop*.

A arquitetura c_1 utiliza um controlador SISO em cada porta controlável `IMaxMap` e `ISortFactor`. O controlador da porta `IMaxMap` utiliza a lei de controle PID, sintonizado com a técnica CHR-0OS-DR. Já o controlador da porta `ISortFactor` utiliza a lei de controle PI, sintonizado com a técnica ZN. Tais decisões produzem *feedback control loops* com tempo médio de estabilização previsto ($\hat{\overline{K_s^1}}$) de 132.5s e sobressinal médio previsto ($\hat{\overline{M_p^1}}$) de 14.7%.

A arquitetura c_2 também utiliza dois controladores SISO, porém adotando lei de controle PI com sintonia CHR-20OS-DR para a porta `IMaxMap` e lei de controle PID com sintonia CHR-0OS-RT para a porta `ISortFactor`. Novamente, esta arquitetura é tão efetiva quando c_1 pois representa

uma alternativa onde obtém-se um tempo de estabilização menor ($\widehat{\overline{K_s^2}} = 85.6 < \widehat{\overline{K_s^1}}$) às custas de um maior sobressinal ($\widehat{\overline{M_p^2}} = 22.3 > \widehat{\overline{M_p^1}}$).

Finalmente, arquitetura c_3 utiliza um único controlador MIMO para a gerência das duas portas controláveis. A lei de controle adotada é *Static State Feedback Control* com sintonia realizada pela técnica LQR. Esta solução é dominada por ambas as arquiteturas c_1 e c_2, pois apresenta o maior tempo de estabilização ($\widehat{\overline{K_s^3}} = 197.5 > \widehat{\overline{K_s^1}} > \widehat{\overline{K_s^2}}$) e também o maior sobressinal ($\widehat{\overline{M_p^3}} = 38.1 > \widehat{\overline{M_p^2}} > \widehat{\overline{M_p^2}}$).

Experimentações iniciais, em malha aberta, apresentaram vazão total média de 538924 registros/s e consumo médio de memória de 25%, utilizando os valores *default* de `tasks.maximum` (2) e `sort.factor` (10) nas 10 máquinas do *cluster*. Com este desempenho, consegue-se realizar a ordenação de 58.2GB de dados em 18 minutos (cada registro tem um tamanho de 100 *bytes*). O maior valor de `tasks.maximum` possível de ser utilizado sem saturar o *cluster* foi 64, produzindo uma vazão total de 1185632 registros/s. Com `sort.factor` igual a 200, o valor observado para o consumo de memória em cada máquina foi de 80%.

De forma similar ao caso do servidor *web* MIMO, os controladores PI, PID e *Static State Feedback* foram implementados em *Shell Script* e executados na mesma frequência com a qual o ensaio ao degrau foi realizado (5s). Cada um dos três sistemas finais resultantes (*cluster* + $c_{1,2,d}$) foram executados e monitorados por 540s. Nos primeiros 240s, os valores de referência para vazão total e consumo de memória foram, respectivamente, 540000 registros/s e 25%. No 241º segundo, tais valores foram modificados para 1000000 registros/s e 60%, respectivamente.

Resultados Obtidos. As Figuras 10.9(a), 10.9(b) e 10.9(c) apresentam os resultados de 31 replicações deste experimento utilizando, respectivamente, os protótipos das arquiteturas c_1, c_2 e c_d. Cada figura apresenta, à esquerda, a vazão parcial ao longo do tempo e, à direita, o consumo de memória ao longo do tempo, obtidas em uma das máquinas do *cluster*. O sinal de referência, os valores máximos, médios e mínimos da vazão parcial e do consumo de memória e a dinâmica prevista são apresentados (ver legenda).

Análise dos Dados. De forma similar ao discutido no experimento com o servidor `httpd`, o *trade-off* característico entre tempo de estabilização e sobressinal foi novamente observado nos *feedback control loops* aplicados ao *Hadoop*. O controlador utilizado na arquitetura c_2 produz um tempo de estabilização menor que aquele gerado pela arquitetura c_1, obviamente às custas de um maior sobressinal. Note como a ausência do fator integral no controlador adotado na arquitetura c_d (*Static State Feedback Control*) produz um erro em regime estacionário (e_{ss}) não nulo (Figura 10.9(c)).

Todos os sistemas finais (*Hadoop* + controlador) apresentaram comportamento BIBO-estável. A efetividade relativa apresentada pelos protótipos das arquiteturas c_1, c_2 e c_d estão compatíveis com as informações previstas pela otimização. Algumas discrepâncias foram novamente observadas entre os valores previstos e medidos de tempo de estabilização e sobressinal. As maiores divergências ocorreram no tempo de estabilização da vazão parcial apresentado pela arquitetura c_1 ($\widehat{\overline{K_{s_v}^1}}$= 130s e $\overline{K_{s_v}^1}$ = 200s) e no tempo de estabilização do consumo de memória apresentado pela arquitetura c_d ($\widehat{\overline{K_{s_m}^d}}$ = 185s e $\overline{K_{s_m}^d}$ = 235s). As mesmas razões discutidas no caso do servidor `httpd` podem ter contribuído para as divergências apresentadas no experimento com o *Hadoop*.

Para melhor analisar as diferenças apresentadas entre valores previstos e medidos, as mesmas hipóteses consideradas no exemplo do servidor `httpd` foram novamente investigadas, utilizando os dados das 31 replicações do experimento com o *Hadoop*:

$$\begin{aligned} H_{0_K}^i &: \overline{K_s^i} > \widehat{\overline{K_s^i}} \\ H_{1_K}^i &: \overline{K_s^i} \leq \widehat{\overline{K_s^i}} \end{aligned} \quad \text{e} \quad \begin{aligned} H_{0_M}^i &: \overline{M_p^i} > \widehat{\overline{M_p^i}} \\ H_{1_M}^i &: \overline{M_p^i} \leq \widehat{\overline{M_p^i}} \end{aligned} \tag{10.4}$$

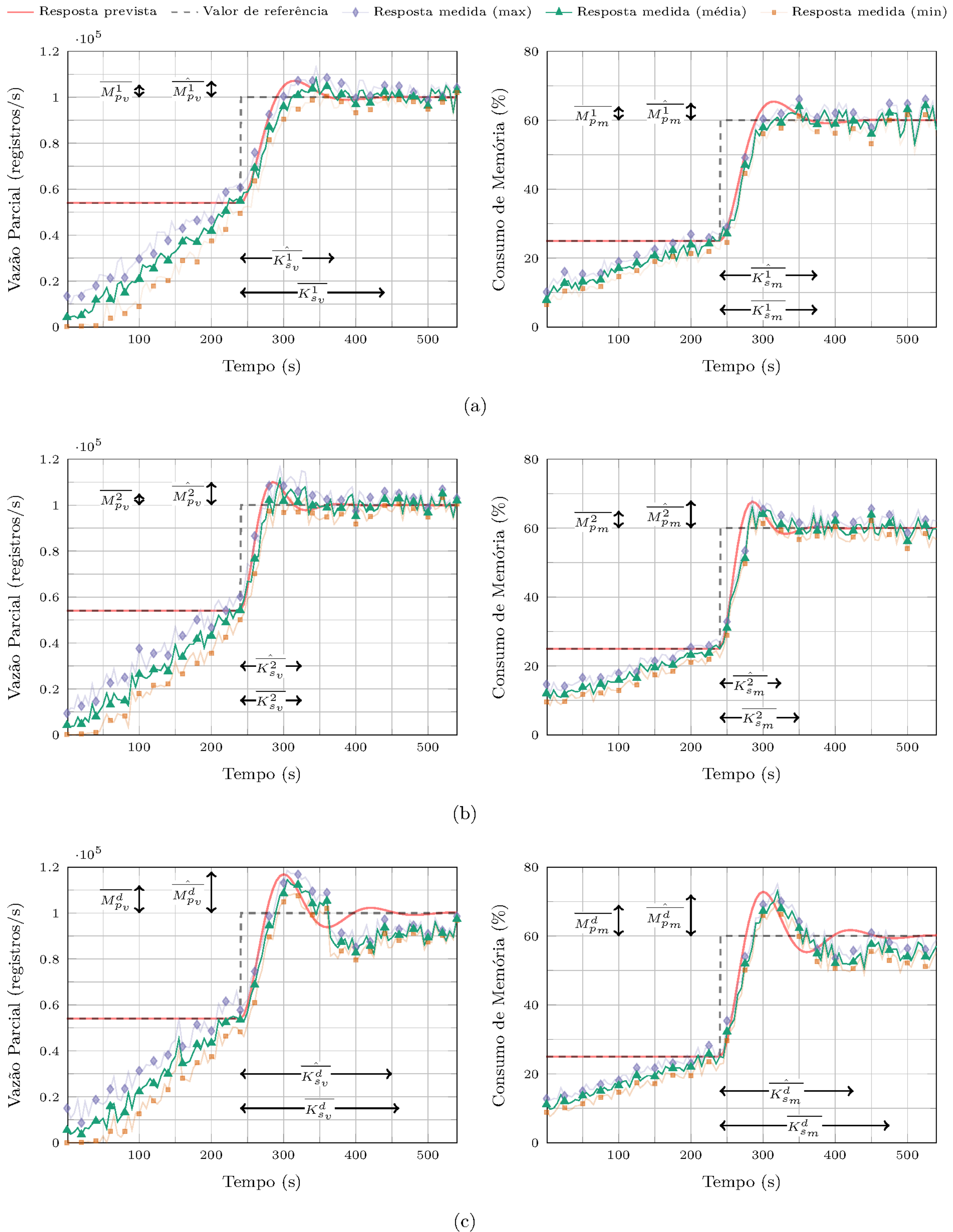


Figura 10.9.: Dinâmica do *Hadoop*, em malha fechada, utilizando os controladores que implementam as arquiteturas candidatas c_1 (a), c_2 (b) e c_d (c).

Novamente, o objetivo foi saber o maior nível de confiança ($\gamma = 1 - \alpha$) que faz com que H_0 seja rejeitada. A Tabela 10.7 apresenta os resultados obtidos. Em relação ao tempo médio de estabilização $\overline{K_s}$, a arquitetura candidata com maior *p-value* foi a c_d (0.218676), indicando que a confiança que podemos ter na acurácia destas predições é menor que $78.13\% = 100 \cdot (1 - 0.218676)$.

Já para o sobressinal médio $\overline{M_p}$, o maior *p-value* foi apresentado pela arquitetura c_1 (0.259787), o que equivale a uma confiança menor que $74.02\% = 100 \cdot (1 - 0.259787)$.

c_i	$\overline{K_s^i}$	***p-value* de Anderson-Darling**	***p-value* de Levene**	**Teste Utilizado**	***p-value* (*t-test* / *Wilcoxon SR*)**
c_1	165.0	0.029674	0.587047	*Wilcoxon SR*	0.208543
c_2	100.0	0.028697	0.793756	*Wilcoxon SR*	0.187746
c_d	225.0	0.049785	0.653769	*Wilcoxon SR*	0.218676
			***p-value* máximo**		0.218676
			γ máximo		0.781324

c_i	$\overline{M_p^i}$	***p-value* de Anderson-Darling**	***p-value* de Levene**	**Teste Utilizado**	***p-value* (*t-test* / *Wilcoxon SR*)**
c_1	11.14	0.795385	0.285676	*t-test*	0.259787
c_2	15.77	0.034569	0.673059	*Wilcoxon SR*	0.210689
c_d	24.81	0.487957	0.105867	*t-test*	0.178675
			***p-value* máximo**		0.259787
			γ máximo		0.740213

Tabela 10.7.: Resultados dos testes de hipótese para verificação da acurácia das predições de tempo de resposta e sobressinal no *Hadoop*.

10.2.2.3. Discussão

Os dois experimentos acima apresentados (servidor `httpd` e *cluster Hadoop*) investigaram com qual acurácia os valores de atributos de qualidade obtidos pelas métricas preditivas do SA:DuSE são de fato observados em implementações reais das arquiteturas de *feedback control loops* encontradas pelas buscas (questão de avaliação QA2). Em resumo, o pior nível de confiança nas predições (70.83%) foi o do tempo médio de estabilização da utilização de CPU e consumo de memória do `httpd`, ao passo que o melhor nível de confiança (78.13%) foi o do tempo médio de estabilização da vazão parcial e consumo de memória do *Hadoop*.

Algumas considerações podem ser realizadas sobre as afirmações acima. Primeiro, as razões de tais discrepâncias e as técnicas comumente adotadas para minimizá-las – apresentadas na seção de análise dos dados do `httpd` (pág. 208) – constituem problemática originada pela Teoria de Controle empregada. Tais desafios estarão sempre presentes no projeto de sistemas *self-adaptive* baseados em *feedback control loops*, utilizando ou não a abordagem baseada em busca apresentada nesta tese.

Segundo, a despeito das discrepâncias encontradas, todas as implementações das arquiteturas preservaram as relações de dominância e não-dominância previstas na otimização. Embora, para alguns casos, o nível de confiança na predição – obtido nos experimentos acima apresentados – não seja satisfatório, saber se uma determinada arquitetura candidata é melhor, pior ou igualmente efetiva (não-dominada) a outra já constitui informação valiosa para o arquiteto de sistemas *self-adaptive*.

Terceiro, atividades mais minuciosas de sintonia – realizadas com o sistema final já implementado – podem ajudar a reduzir as discrepâncias apresentadas. Finalmente, futuras extensões do

SA:DuSE podem incluir constructos para utilização de modelos de mais alta ordem na representação da dinâmica do sistema-alvo. Um amplo conjunto de modelos paramétricos de mais alta ordem e as suas respectivas técnicas de sintonia está disponível na literatura [312]. Vale ressaltar que não é objetivo deste trabalho a proposta de novas leis de controle ou técnicas de sintonia voltadas para a redução das discrepâncias acima discutidas.

10.2.3. Estudo 3: Benefícios do Uso de Abordagens Baseadas em Busca no Projeto de Sistemas Self-Adaptive

Este estudo investigou se a abordagem de projeto arquitetural baseado em busca proposta nesta tese contribui para uma maior efetividade e menor complexidade das arquiteturas de sistemas *self-adaptive* produzidas pelo processo, quando comparada a abordagens tradicionais de projeto arquitetural (questões de avaliação QA3.1 e QA3.2). Adicionalmente, deseja-se saber se a abordagem aqui apresentada facilita a aquisição, por arquitetos iniciantes, de conhecimento refinado de projeto na área de sistemas *self-adaptive* (questão de avaliação QA3.3). A Seção 10.2.3.1 apresenta a definição do experimento controlado, enquanto a Seção 10.2.3.2 discute as atividades de planejamento do experimento. A análise e discussão dos resultados obtidos são apresentados na Seção 10.2.3.3. Finalmente, ameaças à validade do experimento são consideradas na Seção 10.2.3.4.

10.2.3.1. Definição do Experimento

O objetivo deste estudo foi coletar evidências empíricas para comprovar/refutar hipóteses relacionadas aos aspectos acima apresentados. Para isso, um experimento controlado com 24 estudantes de um curso de especialização em Computação Distribuída e Ubíqua foi realizado. O objetivo do experimento pode ser definido, utilizando o *template* GQM (*Goal Quality Metric*) [29], como:

Analizar o projeto de sistemas *self-adaptive*;

com o propósito de avaliar a abordagem de projeto arquitetural baseada em busca aqui apresentada e uma abordagem tradicional de projeto arquitetural baseada em catálogos de estilos arquiteturais;

em relação à efetividade e complexidade das arquiteturas resultantes, bem como ao potencial da abordagem em promover a aquisição, por arquitetos iniciantes, de conhecimento refinado de projeto na área de sistemas *self-adaptive*;

sob ponto de vista dos pesquisadores desta tese;

no contexto de estudantes de pós-graduação adicionando capacidades de autogerenciamento a sistemas computacionais.

O estudo apresentado nesta seção é classificado como um *quasi-experimento* com comparação emparelhada de blocos sujeito-objeto (*blocked subject-object study*) [322]. Um estudo empírico é denominado um *quasi-experimento* quando não há aleatoriedade total na escolha dos sujeitos (participantes do experimento) a partir da população e na atribuição de tratamentos (*treatments* – valores permitidos para a variável independente) a sujeitos. Visto que apenas estudantes atuaram como sujeitos, o estudo aqui apresentado é um *quasi-experimento*.

Na comparação emparelhada de blocos sujeito-objeto, cada participante do experimento utiliza os dois tratamentos sendo considerados (neste caso, as abordagens de projeto arquitetural). Este mecanismo aumenta o número de amostras e, portanto, a relevância estatística dos resultados

obtidos. Note que, tanto a característica de *quasi-experimento* quanto a comparação emparelhada de blocos sujeito-objeto introduzem ameaças à validade. Medidas para minimização destas ameação foram tomadas durante o projeto do experimento. Tais aspectos são discutidos a seguir.

10.2.3.2. Planejamento do Experimento

O experimento foi realizado no âmbito de uma disciplina de 32h sobre Engenharia de Software para Sistemas Distribuídos, oferecida pela Especialização em Computação Distribuída e Ubíqua[5] do Instituto Federal de Educação, Ciência e Tecnologia da Bahia (IFBa). Conforme apresentado na Tabela 10.8, a disciplina foi dividida em três partes: *i)* aulas; *ii)* avaliação e treinamentos; e *iii)* experimento.

Fase	Dia (4h cada)	Atividades Realizadas
Aulas	1	Fundamentos de Sistemas *Self-Adaptive* (motivação, arquitetura de referência MAPE-K, abordagens atuais, desafios)
	2	Introdução a *Feedback Control* (metas de controle, propriedades de controle, abordagens SISO com ganho fixo)
	3	*Feedback Control* (abordagens MIMO e adaptativas)
	4	Sistemas *Self-Adaptive* – estudos de caso
Avaliação e Treinamentos	5	Primeira hora: discussão Próximas 3 horas: avaliação escrita
	6	Primeira hora: discussão da avaliação escrita Próximas 3 hoas: treinamento (DuSE-MT e catálogo de estilos arquiteturais para sistemas *self-adaptive*)
Experimento	7	Primeiros 110 minutos: testes 1 e 2 Próximos 110 minutos: testes 3 e 4 Próximos 20 minutos: testes 5 e 6 (questionário)
	8	Primeiros 110 minutos: testes 7 e 8 Próximos 110 minutos: testes 9 e 10 Próximos 20 minutos: testes 11 e 12 (questionário)

Tabela 10.8.: Cronograma de preparação e realização do experimento.

Nos primeiros quatro dias da disciplina, os estudantes tiveram aulas expositivas sobre os fundamentos de sistemas *self-adaptive* e de *feedback control*. As principais técnicas para controle SISO e MIMO (ver Seção 3.3.4.1, pág. 45) foram discutidas e uma visão geral sobre controle adaptativo (vide Seção 3.3.4.3, pág. 57) foi apresentada. Os estudantes atuavam, em sua maioria, como desenvolvedores de sistemas ou administradores de redes e tiveram experiência prévia – em uma outra disciplina do mesmo programa de especialização – nas áreas de arquitetura de *software* e modelagem de *software*. Nenhuma informação explícita sobre os *trade-offs* envolvendo propriedades de controle foi apresentada durante as aulas, visto que este aspecto era parte das hipóteses investigadas no experimento.

No quinto dia da disciplina, realizou-se uma discussão sobre o assunto apresentado nas aulas anteriores e uma avaliação individual escrita foi aplicada aos estudantes. No sexto dia, a avaliação foi corrigida e debatida. Em seguida, uma sessão de treinamento de 3h sobre a ferramenta DuSE-MT e sobre o catálogo de estilos arquiteturais para sistemas *self-adaptive* foi realizada. O experimento aconteceu nos últimos dois dias da disciplina.

[5] `http://gsort.ifba.edu.br`.

Os estudantes foram aleatoriamente divididos em dois grupos de igual tamanho (grupo 1 e grupo 2). O experimento realizou comparação emparelhada de blocos sujeito-objeto utilizando três objetos de experimento (modelo do servidor *web* MIMO, modelo do *cluster* elástico para aplicações *MapReduce* e questionário) e dois tratamentos (abordagem baseada em busca e abordagem baseada em catálogo). Dessa forma, um total de 12 testes (8 de projeto arquitetural e 4 de resposta a questionário) foi aplicado no experimento. Os testes são apresentados na Tabela 10.9 e discutidos mais à frente.

Objetos do Experimento. Os testes de projeto arquitetural realizados no experimento tiveram como objetivo a criação de sistemas gerenciadores (*loops* de adaptação) para dois modelos iniciais (sistemas gerenciados apresentados na Figura 10.10): um servidor *web* MIMO e um *cluster* elástico para aplicações *MapReduce*. Todos os testes de projeto arquitetural tiveram como meta a extensão do modelo inicial com alguma arquitetura de *feedback control loop* que regulasse uma métrica de desempenho e que minimizasse (apresentasse *trade-off* equilibrado) o tempo de estabilização, sobressinal máximo e *overhead* de controle.

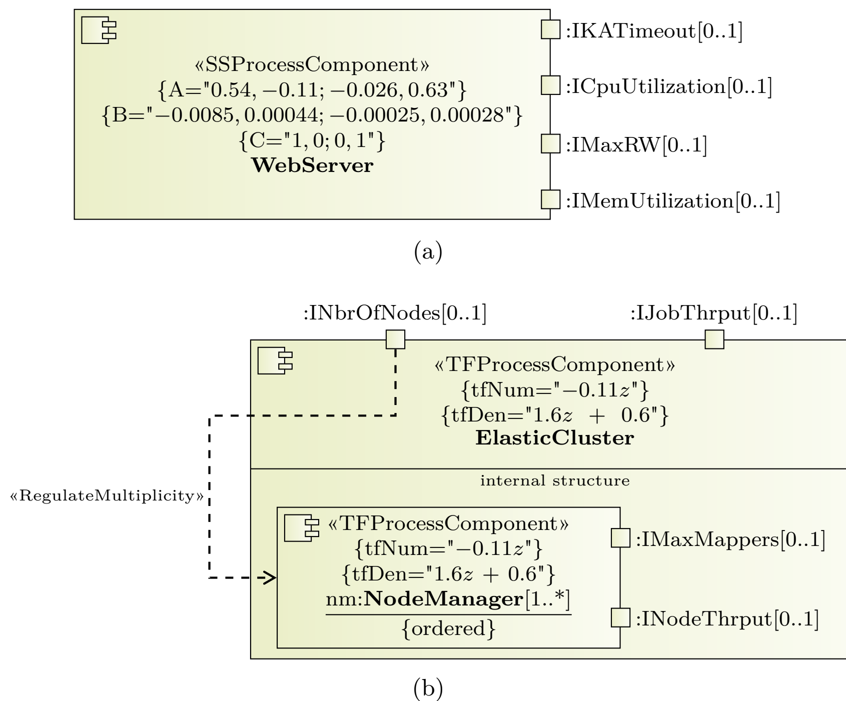


Figura 10.10.: Objetos de projeto arquitetural utilizados no experimento.

Ambos os grupos usaram o DuSE-MT como ferramenta de modelagem. Entretanto, todas as funcionalidades para navegação no espaço de projeto e otimização de arquiteturas foram desabilitadas quando utilizando a abordagem baseada em catálogo. De forma similar, os estudantes não tiveram acesso ao catálogo de estilos quando utilizando a abordagem baseada em busca. Vale ressaltar que o experimento não investigou variáveis tais como produtividade de projeto e densidade de falhas dos modelos gerados, visto que tais aspectos são obviamente favorecidos pelo uso de abordagens automatizadas.

Seleção de Variáveis. No estudo reportado nesta seção, analisou-se o impacto da abordagem de projeto adotada em três variáveis dependentes: *i)* efetividade do sistema gerenciador resultante; *ii)* complexidade da arquitetura do sistema gerenciador; e *iii)* potencial do método em promover a aquisição, por arquitetos iniciantes, de conhecimento refinado de projeto na área de sistemas *self-adaptive*. Esta seção descreve as métricas adotadas para quantificar tais variáveis.

1) Medindo efetividade (Generational Distance): esta variável foi quantificada em termos de quão próximos os atributos de qualidade da(s) arquitetura(s) resultante(s) está(ão) de um conjunto de soluções Pareto-ótimas previamente obtido. Visto que abordagens baseadas em metaheurísticas – como a aqui proposta – não garantem a obtenção de ótimos globais, realizou-se um conjunto de 50 execuções de otimização e calculou-se o *Pareto-front* P^* do conjunto formado por todas as saídas das execuções. Assume-se que P^* (conjunto de triângulos na Figura 10.11) é um bom representante das arquiteturas candidatas mais efetivas e constitui um valor de referência razoável para avaliar quão efetivas são as arquiteturas produzidas pelos sujeitos do experimento. Este procedimento foi realizado para ambos os objetos de projeto (modelos iniciais SW – servidor *web* e C – *cluster* elástico para aplicações *MapReduce*) usados no experimento, produzindo os *Pareto-fronts* de referência P^*_{SW} e P^*_C.

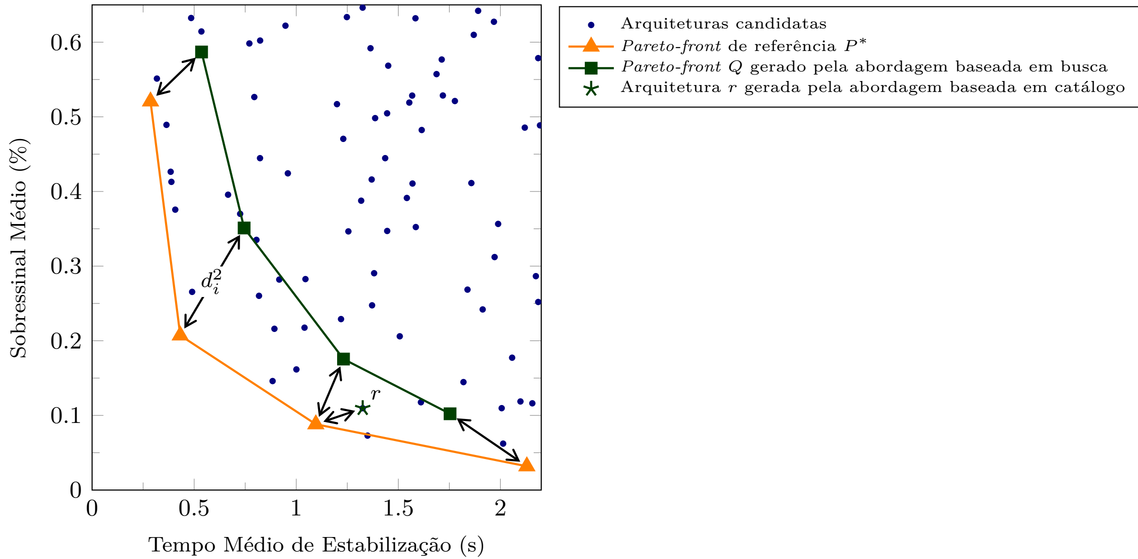


Figura 10.11.: Exemplo de *Pareto-front* de referência P^*e *Generational Distance* (GD).

A métrica *Generational Distance* (GD – apresentada na Seção 4.4.1, pág. 84) avalia a proximidade entre dois *Pareto-fronts* Q e P^*. A métrica encontra a distância média entre as soluções de Q e P^* (ou entre uma arquitetura particular r e aquela solução de P^* mais próxima):

$$GD = \frac{(\sum_{i=1}^{|Q|} d_i^p)^{1/p}}{|Q|}; \text{ onde } d_i^2 = \min_{k=1}^{|P*|} \sqrt{\sum_{m=1}^{M} (f_m^{(i)} - f*_m^{(k)})^2} \tag{10.5}$$

Qualquer *norma-L^p* (norma de Lebesgue) pode ser utilizada na *Generational Distance*. Quando $p = 2$, como apresentado acima, d_i^2 é a distância euclidiana entre uma solução $i \in Q$ e o membro de P^* que se encontra mais próximo. A *Generational Distance* foi utilizada neste experimento porque é mais adequada que alternativas como *Error Ratio* e *Set Coverage* (vide Seção 4.4.1, pág. 84) na comparação de *Pareto-fronts* disjuntos e porque pode ser avaliada a um custo computacional mínimo.

Em relação à arquitetura produzida pela abordagem baseada em catálogo, inicialmente calculou-se a sua posição no espaço-objetivo (valores para as métricas de qualidade apresentadas na Seção 7.3, pág. 161) e, em seguida, mediu-se a distância para o *Pareto-front* de referência P^* usando *Generational Distance*. Note que um *Pareto-front* Q, obtido pela abordagem baseada em busca (ex: conjunto de quadrados na Figura 10.11), não necessariamente é tão efetivo quanto o *Pareto-front* de referência P^* utilizado. Tal fato é decorrente da aleatoriedade inerente da técnica evolucionária de otimização multiobjetivo adotada (NSGA-II).

Embora r seja apresentada, na Figura 10.11, mais próxima do *Pareto-front* de referência P^* do

que qualquer solução do *Pareto-front* Q (obtido em uma única execução da abordagem baseada em busca), acredita-se que tal situação não seja comum em projetos realizados por arquitetos iniciantes na área de sistemas *self-adaptive*. Portanto, um dos objetivos do experimento aqui descrito foi encontrar alguma evidência que comprove/refute a hipótese que abordagens baseadas em busca produzem arquiteturas mais efetivas no atendimento das metas de adaptação.

2) Medindo complexidade de projeto (Component Point): a segunda variável dependente analisada no experimento foi complexidade de projeto, em função do seu impacto direto no esforço de desenvolvimento necessário para implementar as arquiteturas propostas. A abordagem *Component Point* (CP) [321] foi utilizada para quantificar este aspecto, motivada pela sua adequação à análise de modelos UML e pela existência de evidência empírica relacionada à sua validade e utilidade [321]. O *Component Point* mede complexidade de projeto em termos da complexidade das interfaces dos componentes e complexidade das interações entre componentes. A complexidade CC_c de um componente c é definida como:

$$CC_c = IFCI_c + ITCI_c = \frac{IFC_c}{n_c} + \frac{ITC_c}{m_c} \tag{10.6}$$

O $IFCI_c$ (*Interface Complexity per Interface*) é definido como sendo a complexidade da interface do componente (IFC_c) dividida pelo número de interfaces providas pelo componente (n_c). De forma semelhante, o $ITCI_c$ (*Interaction Complexity per Interaction*) é obtido dividindo-se a complexidade de interação do componente (ITC_c) pelo número de interações do componente (m_c). O cálculo dos valores de IFC_c e ITC_c é descrito a seguir.

O primeiro passo no cálculo do IFC_c é a classificação das interfaces em dois grupos: ILF (*Internal Logical Files*) e EIF (*External Interface Files*). Interfaces ILF são aquelas que contêm operações que modificam atributos de outras interfaces. As demais são classificadas como EIF. O *Component Point* especifica também como um nível de complexidade (baixo, médio ou alto) é atribuído a cada interface, baseado no número de operações e de parâmetros de operações que ela apresenta. A partir daí, o IFC_c é então definido como:

$$IFC_c = \sum_{j=1}^{2} \sum_{k=1}^{3} I_{jk} \times W_{jk} \tag{10.7}$$

I_{jk} é o número de interfaces do tipo j (1=ILF e 2=EIF) com nível de complexidade k (1=baixo, 2=médio e 3=alto). W_{jk} é um peso, definido pela abordagem *Component Point*, para interfaces do tipo j com nível de complexidade k.

O ITC_c, por sua vez, é avaliado em termos da frequência de interação (*Interaction Frequency* – IF_{ij}) da j-ésima operação da i-ésima interface do componente c. Adicionalmente, uma medida de complexidade (*Complexity Measure* – CM_{ijk}) do k-ésimo tipo de dado envolvido na execução da j-ésima operação da i-ésima interface é também utilizada.

IF_{ij} é definido como a razão entre o número de interações realizado pela operação (N_O) e o número de interações realizadas por todas as operações da interface (N_I). CM_{ijk}, por sua vez, é definido como:

$$CM_{ijk}(D, L) = L + \sum_{n=1}^{m} CM(DT_n, L + 1) \tag{10.8}$$

D é o tipo de dado sendo medido, L é o nível do grafo de composição de tipos onde D aparece (o nível inicial possui valor 1), DT_N é o tipo de dado do n-ésimo membro constituinte de D e

m é o número de membros constituintes de D. Finalmente, ITC_c é definido como:

$$ITC_c = \sum_{i=1}^{p} \sum_{j=1}^{q} \left(IF_{ij} \times \sum_{k=1}^{r} CM_{ijk} \right) \quad (10.9)$$

p é o número de interfaces providas pelo componente c, q é o número de operações que a i-ésima interface disponibiliza e r é o número de tipos de dados envolvidos na execução da j-ésima operação da i-ésima interface. A complexidade geral de uma arquitetura a (*Architectural Complexity* – AC_a) é a soma dos valores de CC_i de todos os componentes presentes em a.

3) Medindo aquisição de conhecimento refinado de projeto (questionário): a terceira variável dependente investigada no experimento foi o potencial do método em promover a aquisição, por arquitetos iniciantes, de conhecimento refinado de projeto na área de sistemas *self-adaptive*. Para isso, um questionário formado por 10 questões de múltipla escolha – relacionadas à compreensão e análise de *trade-offs* no projeto de arquiteturas de controle – foi utilizado. O questionário foi respondido por cada estudante ao final de cada dia de experimento e notas foram atribuídas de acordo com o número de questões corretamente respondidas. O objetivo foi avaliar até que ponto a abordagem de projeto adotada pôde promover a aquisição de conhecimento relacionado aos *trade-offs* de projeto comumente presentes.

Formulação das Hipóteses. No *quasi-experimento* reportado nesta seção, comparou-se o uso de duas abordagens para projeto arquitetural: a baseada em busca, proposta nesta tese, e uma abordagem convencional baseada em catálogos de estilos arquiteturais de domínio específico. O objetivo foi analisar as arquiteturas resultantes em relação à efetividade na obtenção das propriedades de controle desejadas e em relação à complexidade dos modelos gerados. Adicionalmente, desejou-se saber se algum método contribui mais significativamente para a aquisição, por arquitetos iniciantes, de conhecimento refinado de projeto. O objetivo do estudo foi descrito em três hipóteses nulas (H_0) e suas hipóteses alternativas (H_1) correspondentes:

- H_0^1: não há diferença em efetividade de projeto (medida em termos da métrica *Generational Distance* – GD) entre um projeto de *feedback control loop* criado com a abordagem baseada em catálogo (abordagem de referência – AR) e um projeto de *feedback control loop* criado com a abordagem baseada em busca (abordagem de intervenção – AI).

$$H_0^1 : \mu_{GD_{AR}} = \mu_{GD_{AI}}$$
$$H_1^1 : \mu_{GD_{AR}} > \mu_{GD_{AI}}$$

- H_0^2: não há diferença em complexidade de projeto (medida em termos da métrica *Architectural Complexity* – AC) entre um projeto de *feedback control loop* criado com a abordagem baseada em catálogo (abordagem de referência – AR) e um projeto de *feedback control loop* criado com a abordagem baseada em busca (abordagem de intervenção – AI).

$$H_0^2 : \mu_{AC_{AR}} = \mu_{AC_{AI}}$$
$$H_1^2 : \mu_{AC_{AR}} > \mu_{AC_{AI}}$$

- H_0^3: não há diferença na aquisição de conhecimento refinado de projeto (medida em termos da nota atribuída ao questionário pós-experimento – NQ) entre projetar um *feedback control loop* usando a abordagem baseada em catálogo (abordagem de referência – AR) e projetar um *feedback control loop* usando a abordagem baseada em busca (abordagem de intervenção – AI).

$$H_0^3 : \mu_{NQ_{AR}} = \mu_{NQ_{AI}}$$
$$H_1^3 : \mu_{NQ_{AR}} < \mu_{NQ_{AI}}$$

Projeto do Experimento. Conforme mencionado anteriormente, cada sujeito do experimento utilizou ambos os tratamentos (abordagens de projeto) e os efeitos puderam, portanto, ser comparados em pares. Visto que os alunos possuiam experiência profissional em duas áreas de atuação distintas (desenvolvimento de sistemas e administração de redes), tal informação foi utilizada como fator de blocagem (*blocking factor*). Com isso, elimina-se o impacto indesejado do perfil técnico do sujeito nos valores obtidos para as variáveis dependentes, melhorando a precisão do experimento.

Teste	Objeto	Tratamento	Sujeitos
01	Servidor *Web* MIMO	Abordagem baseada em catálogo	Grupo 1
02	*Cluster* Elástico para Aplicações *MapReduce*	Abordagem baseada em busca	Grupo 2
03	*Cluster* Elástico para Aplicações *MapReduce*	Abordagem baseada em catálogo	Grupo 1
04	Servidor *Web* MIMO	Abordagem baseada em busca	Grupo 2
05	Questionário	Abordagem baseada em catálogo	Grupo 1
06	Questionário	Abordagem baseada em busca	Grupo 2
07	*Cluster* Elástico para Aplicações *MapReduce*	Abordagem baseada em busca	Grupo 1
08	Servidor *Web* MIMO	Abordagem baseada em catálogo	Grupo 2
09	Servidor *Web* MIMO	Abordagem baseada em busca	Grupo 1
10	*Cluster* Elástico para Aplicações *MapReduce*	Abordagem baseada em catálogo	Grupo 2
11	Questionário	Abordagem baseada em busca	Grupo 1
12	Questionário	Abordagem baseada em catálogo	Grupo 2

Tabela 10.9.: Testes definidos para o experimento.

Estudantes de cada área de atuação foram aleatoriamente e igualmente atribuídos aos grupos 1 e 2, produzindo uma proporção igual de desenvolvedores e administradores em cada grupo. Conforme apresentado na Tabela 10.9, 8 testes de projeto arquitetural e 4 testes de resposta a questionário foram conduzidos e cada grupo trabalhou com todas as combinações possíveis de objeto de experimento e tratamento.

No primeiro dia de experimento, o grupo 1 aplicou a abordagem baseada em catálogo – primeiro no modelo do servidor *web* MIMO e depois no modelo do *cluster* elástico para aplicações *MapReduce*. Neste mesmo dia, o grupo 2 aplicou a abordagem baseada em busca na ordem inversa de objetos de experimento utilizada pelo grupo 1 (primeiro no modelo do *cluster* elástico para aplicações *MapReduce* e depois no modelo do servidor *web* MIMO). Ao final do primeiro dia, ambos os grupos responderam o questionário. No segundo dia de experimento, os grupos trocaram o tratamento utilizado e o aplicaram aos objetos na ordem inversa à realizada por eles no dia anterior. O mesmo questionário foi novamente aplicado ao final do segundo dia.

Para minimizar os efeitos decorrentes da experiência obtida entre o primeiro e segundo dia de experimento, atribuiu-se sistematicamente qual combinação objeto-tratamento é vivenciada pela primeira vez em cada grupo. A ordem de condução dos testes é apresentada na Tabela 10.9. Para evitar a dedução de hipótese (*hypothesis guessing*) e outras ameaças sociais, nenhuma consideração sobre as arquiteturas resultantes foi realizada e os alunos não estavam cientes do experimento até a sua completa finalização.

10.2.3.3. Análise dos Resultados e Discussão

Após a finalização do estudo, 20 dos 24 estudantes produziram dados válidos da comparação emparelhada de *Generational Distance* (*GD*), *Architectural Complexity* (*AC*) e Nota do Questionário (*NQ*). Com a ajuda da ferramenta DuSE-MT, todos os modelos UML resultantes foram serializados em arquivos XMI e relacionados aos valores de atributos de qualidade (localização

no espaço-objetivo) por eles produzidos. Tais valores foram usados para calcular a *Generational Distance* e a *Architectural Complexity*, conforme apresentado na Seção 10.2.3.2, pág. 220.

Análise. A Figura 10.12 e a Tabela 10.10 apresentam as estatísticas descritivas dos valores obtidos para cada variável dependente, de acordo com o tratamento utilizado. A diferença emparelhada de cada variável dependente é também apresentada.

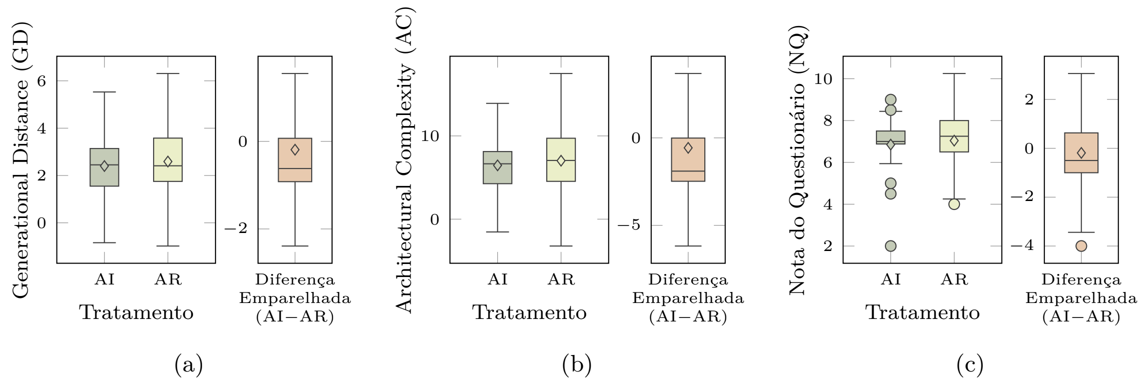


Figura 10.12.: Gráficos "*box and whispers*" para efetividade de projeto (a), complexidade de projeto (b) e potencial de aquisição de conhecimento refinado de projeto (c).

Generational Distance (GD)			
	média (μ)	**mediana**	**desvio padrão**
Abordagem baseada em busca (AI)	2.40	2.45	1.08
Abordagem baseada em catálogo (AR)	2.59	2.41	1.03
Diferença (AI−AR)	-0.19	-0.62	1.32

Architecture Complexity (AC)			
	média (μ)	**mediana**	**desvio padrão**
Abordagem baseada em busca (AI)	6.46	6.65	2.77
Abordagem baseada em catálogo (AR)	7.02	7.05	2.70
Diferença (AI−AR)	-0.57	-1.90	3.47

Nota do Questionário (NQ)			
	média (μ)	**mediana**	**desvio padrão**
Abordagem baseada em busca (AI)	6.85	7.00	1.43
Abordagem baseada em catálogo (AR)	7.04	7.25	1.27
Diferença (AI−AR)	-0.19	-0.50	1.51

Tabela 10.10.: Estatísticas descritivas para as variáveis dependentes do experimento.

O primeiro passo realizado na análise dos dados obtidos foi verificar se as premissas usuais para utilização de testes paramétricos de hipótese eram atendidas. O teste de Anderson-Darling [68] foi utilizado para avaliar se as diferenças emparelhadas eram normalmente distribuídas. Já a hipótese nula de heteroscedasticidade entre a abordagem baseada em catálogo e a abordagem baseada em busca foi investigada com o teste de Brown-Forsythe [115]. A Tabela 10.11 apresenta os resultados destes testes.

Variável Dependente	p-value de Anderson-Darling	p-value de Brown-Forsythe
Generational Distance	1.29814505086953E-007	0.9009324909
Architecture Complexity	2.88672812219819E-010	0.7207666486
Nota do Questionário	0.635529605	0.7167840476

Tabela 10.11.: Resultados do teste de normalidade de Anderson-Darling e teste de heteroscedasticidade de Brown-Forsythe.

Com um nível de significância (α) de 0.05, observou-se que somente a diferença emparelhada da Nota do Questionário (NQ) pôde ser considerada normalmente distribuída (*p-value* de Anderson-Darling > 0.05). Adicionalmente, para todas as variáveis dependentes, a hipótese nula de heteroscedasticidade não pôde ser rejeitada (*p-value* de Brown-Forsythe > 0.05). Visto que todas as premissas devem ser atendidas, somente a hipótese nula H_0^3 foi avaliada via teste paramétrico. As diferenças emparelhadas da *Generational Distance* e da *Architecture Complexity* não foram consideradas normalmente distribuídas (*p-value* de Anderson-Darling < 0.05) e, portanto, as hipóteses nulas H_0^1 e H_0^2 foram avaliadas via testes não-paramétricos.

H_0^i	Teste Utilizado	Critério	Conclusão
1	Wilcoxon Signed-Rank	T (410) > t-crítico (378)	Rejeitada
2	Wilcoxon Signed-Rank	T (367) > t-crítico (361)	Rejeitada
3	Paired t-test	p-value = 0.5488018266	Não Rejeitada

Tabela 10.12.: Resultados do teste de normalidade de Anderson-Darling e teste de heteroscedasticidade de Brown-Forsythe.

Conforme apresentado na Tabela 10.12, o teste *Wilcoxon Signed-Rank* [322, 68] foi utilizado para avaliar as hipóteses nulas H_0^1 e H_0^2, enquanto a hipótese nula H_0^3 foi avaliada utilizando o *Paired t-test* [322]. Com um nível de significância (α) de 0.05, H_0^1 e H_0^2 foram rejeitadas. As evidências obtidas não foram suficientes para rejeitar a hipótese nula H_0^3.

Discussão. As estatísticas descritivas e os resultados dos testes de hipótese mostraram que o uso de abordagens baseadas em busca melhora a efetividade e reduz a complexidade das arquiteturas de controle produzidas. Nada pôde ser afirmado em relação à promoção de aquisição, por arquitetos iniciantes, de conhecimento refinado de projeto. Na verdade, estudantes que utilizaram primeiro a abordagem baseada em busca obtiveram notas ligeiramente inferiores (6.85) às dos estudantes do outro grupo (7.04).

Visto que o conhecimento de projeto presente no catálogo de estilos arquiteturais é o mesmo presente no espaço de projeto utilizado, duas razões para estes dados podem ser identificadas. Primeiro, estudantes expostos à abordagem baseada em busca podem ter vivenciado um conjunto maior de arquiteturas candidatas, dificultando a assimilação de conclusões sobre os *trade-offs* envolvidos. Segundo, os *trade-offs* envolvidos podem, eventualmente, não ser tão difíceis de serem assimilados, mesmo utilizando abordagens tradicionais de projeto. Daí, a diferença pode ter ocorrido devido ao acaso. Outros experimentos devem ser realizados para melhor investigar este aspecto.

A *Generational Distance* foi, em média, 7% menor ao utilizar a abordagem baseada em busca (2.40) quando comparada ao valor obtido ao utilizar a abordagem baseada em catálogo (2.59). A *Architecture Complexity* é, em média, também 7% menor com a abordagem baseada em busca (6.46) quando compara à complexidade das arquiteturas resultantes da abordagem baseada em catálogo (7.02). Tais valores indicam alguma melhoria nas arquiteturas resultantes, mas estudos adicionais devem ser realizados para investigar até que ponto valores melhores para estas métricas podem ser obtidos.

10.2.3.4. Ameaças à Validade

Esta seção apresenta as ameaças à validade [322] identificadas para o experimento realizado no Estudo 3.

Ameaças à Validade de Constructo. Validade de constructo é o grau com que os objetos e as medições adotadas no experimento refletem os constructos equivalentes do mundo real. Três ameaças deste tipo foram identificadas para o experimento: *i)* explicação pré-operacional de constructos inadequada; *ii)* dedução de hipótese; e *iii)* representatividade dos objetos.

Primeiro, visto que a teoria utilizada pelos *feedback control loops* envolve áreas tais como sistemas e sinais, modelagem de comportamento dinâmico e análise no domínio da frequência, os estudantes podem não ter tipo tempo suficiente para assimilar tais fundamentos de uma forma mais consolidada. Para reduzir esta ameaça, procurou-se utilizar o mínimo necessário de tal conhecimento na realização do experimento. Adicionalmente, o máximo possível de apoio por ferramenta foi utilizado.

Segundo, os estudantes podem ter se esforçado mais na produção de arquiteturas utilizando a abordagem baseada em busca por suspeitar que este tratamento foi proposto pelos executores do experimento. Para minimizar este problema, estudantes não foram informados da realização do experimento até a sua conclusão e notas foram atribuídas às atividades realizadas com ambas as abordagens.

Terceiro, os objetos utilizados no experimento (modelo do servidor *web* MIMO, modelo do *cluster* elástico para aplicações *MapReduce* e questionário) podem não ser representantes legítimos dos problemas e decisões enfrentadas rotineiramente no projeto de sistemas *self-adaptive*. Visto que os dois sistemas *self-adaptive* adotados vêm sendo amplamente investigados em pesquisa recentes, acredita-se que eles constituem exemplos interessantes e representativos da prática corrente.

Ameaças à Validade Interna. A validade interna analisa até que ponto fatores desconhecidos podem afetar as variáveis dependentes, em relação a causalidade. Duas destas ameaças foram identificadas para o experimento. A primeira é denominada maturação – sujeitos adquirindo experiência e melhorando seus desempenhos em função de vivências previamente realizadas. Para reduzir esta ameaça, os objetos foram atribuídos, nos dois dias de experimento, de forma alternada entre grupos. A segunda ameaça diz respeito à instrumentação utilizada. Visto que uma nova ferramenta de modelagem foi utilizada no experimento (DuSE-MT), algum impacto pode ter sido gerado em relação à capacidade dos sujeitos de produzir os modelos arquiteturais.

Ameaças à Validade Externa. A validade externa diz respeito à possibilidade de generalização dos resultados do experimento para outras situações semelhantes. Pelo fato de os sujeitos do experimento terem sido estudantes de um curso de pós-graduação em Computação Distribuída e Ubíqua, este perfil pode não representar o público comumente esperado.

Ameaças à Validade de Conclusão. A validade de conclusão examina a capacidade de promoção dos resultados obtidos à categoria de teoria ou conceito geral que suporte o experimento. Visto que os objetos do experimento foram desenvolvidos pelos executores do estudo, existe a ameaça potencial que estes objetos não representem realmente o problema sendo investigado. Esta ameaça pode ser minimizada com o uso de consultores externos. Adicionalmente, estudos mais avançados devem ser realizados para investigar meios para transformação do uso dos espaços de projeto – aqui propostos – em teorias de projeto, conforme descritas por Gregor & Jones em [158] e [117].

Mais informações sobre o experimento podem ser encontradas em [21]. Todo o material utilizado (aulas, catálogo de estilos arquiteturais para sistemas *self-adaptive*, questionário e dados obtidos) está disponível no *website* `http://wiki.ifba.edu.br/tr-ce012014`.

10.3. Resumo do Capítulo

Este capítulo apresentou as atividades de avaliação da abordagem de projeto arquitetural automatizado de sistemas *self-adaptive* proposta neste trabalho. Para isso, três diferentes metas de avaliação foram definidas. A primeira meta investigou até que ponto as dimensões de projeto, pontos de variação e métricas de qualidade definidos no SA:DuSE capturam os *trade-offs* comumente envolvidos no projeto de sistemas *self-adaptive*. A segunda meta analisou a acurácia com a qual os valores apresentados pelas métricas do SA:DuSE são de fato observados em protótipos reais das arquiteturas de controle encontradas pelas buscas. Finalmente, a terceira meta verificou se a abordagem de projeto arquitetural automatizado aqui apresentada produz arquiteturas mais efetivas e menos complexas, quando comparada às técnicas de projeto arquitetural atualmente adotadas.

A primeira meta de avaliação foi investigada no Estudo 1. As populações finais produzidas pelas execuções da otimização para os modelos arquiteturais iniciais do servidor *web* MIMO e do *cluster* elástico para aplicações *MapReduce* foram analisadas. A cardinalidade dos *Pareto-fronts* de referência encontrados (13 para o servidor *web* MIMO e 18 para o *cluster* elástico para aplicações *MapReduce*) indicaram que o SA:DuSE capturou corretamente o caráter multiobjetivo do projeto de sistemas *self-adaptive*. Para ambos os modelos, observou-se que os *trade-offs* usuais entre tempo de estabilização e sobressinal, bem como entre efetividade de controle e *overhead* de controle, foram corretamente produzidos pelo SA:DuSE.

O indicador *hypervolume* foi utilizado para quantificar a convergência e espalhamento do *Pareto-front* produzido por uma única execução da otimização. 31 replicações da otimização foram realizadas para investigar a hipótese de convergência a 95% do *Pareto-front* de referência. Verificou-se – com um nível de significância de 0.01 – que tal convergência é obtida antes de 228 iterações de otimização para o modelo do servidor *web* MIMO e antes de 215 iterações para o modelo do *cluster* elástico para aplicações *MapReduce*.

O Estudo 2 investigou a segunda meta de avaliação. Para isso, protótipos reais dos *feedback control loops* produzidos pelo SA:DuSE foram implementados para o servidor *web* Apache HTTP (`httpd`) e para a plataforma de aplicações *MapReduce Apache Hadoop*. Duas arquiteturas do *Pareto-front* e uma arquitetura dominada foram escolhidas, para cada modelo arquitetural inicial, e implementadas. Verificou-se que as relações de dominância e não-dominância foram corretamente observadas nas execuções, tanto do `httpd` quanto do *Hadoop*.

31 replicações de medições de tempo de estabilização e sobressinal foram utilizadas para encontrar o maior nível de confiança λ que faz com que o intervalo de confiança da métrica medida contenha o valor da métrica prevista. Concluiu-se, para as três arquiteturas analisadas, que o tempo de estabilização – previsto pelo SA:DuSE – para o controle da utilização de CPU e

consumo de memória do `httpd` é observado, no pior caso, em 70.83% das execuções dos protótipos. O sobressinal previsto para o controle de tais variáveis é observado, no pior caso, em 74.46% das execuções. Para as experimentações realizadas com o Hadoop, concluiu-se que o tempo de estabilização previsto para o controle da vazão parcial do *job* e consumo de memória é observado, no pior caso, em 78.13% das execuções. O sobressinal previsto para o controle do *Hadoop* é observado, no pior caso, em 74.02% das execuções dos protótipos.

O Estudo 3 descreveu a realização de um experimento controlado que investigou se o SA:DuSE melhora a efetividade e reduz a complexidade das arquiteturas produzidas, quando comparado a processos de projeto baseados em catálogos de estilos arquiteturais. Adicionalmente, investigou-se se a abordagem baseada em busca promove uma melhor aquisição, por arquitetos iniciantes, de conhecimento refinado de projeto na área de sistemas *self-adaptive*. Para isso, 24 estudantes de pós-graduação realizaram a construção de arquiteturas de sistemas gerenciadores para os modelos iniciais do servidor *web* MIMO e do *cluster* elástico para aplicações *MapReduce*.

A efetividade das arquiteturas produzidas foi medida através do indicador *Generational Distance*, que avalia quão proximo a arquitetura produzida está de um *Pareto-front* supostamente ótimo previamente encontrado. A técnica *Component Point* foi utilizada para medir a complexidade de projeto das arquiteturas, enquanto um questionário pós-experimento avaliou a aquisição de conhecimento refinado de projeto. Com um nível de significância de 0.05, concluiu-se que o uso da abordagem baseada em busca melhora a efetividade e reduz a complexidade das arquiteturas produzidas. Nenhuma conclusão pôde ser obtida em relação à aquisição de conhecimento refinado de projeto.

Capítulo 11

Trabalhos Correlatos

As much as I live I shall not imitate them or hate myself for being different to them.
Orhan Pamuk

Esta tese apresentou uma abordagem automatizada para projeto e análise de arquiteturas de sistemas *self-adaptive*. Para isso, uma infraestrutura genérica para representação sistemática de conhecimento refinado de projeto foi elaborada, bem como um mecanismo para otimização multiobjetivo de arquiteturas de *software*, viabilizando a manifestação explícita dos *trade-offs* envolvidos no projeto de tais arquiteturas. Esta infraestrutura disponibilizou todos os mecanismos necessários para a instanciação de um espaço de projeto (*design space*) concreto que captura as principais estratégias de construção de sistemas *self-adaptive* baseados em *feedback control loops*.

Este capítulo apresenta as dimensões de correlação referentes ao trabalho aqui apresentado e os principais estudos em cada uma dessas dimensões. São apresentados os objetivos e técnicas utilizadas em cada trabalho, bem como os pontos que se assemelham e diferem das contribuições apresentadas nesta tese. A Figura 11.1 apresenta o roteiro deste capítulo.

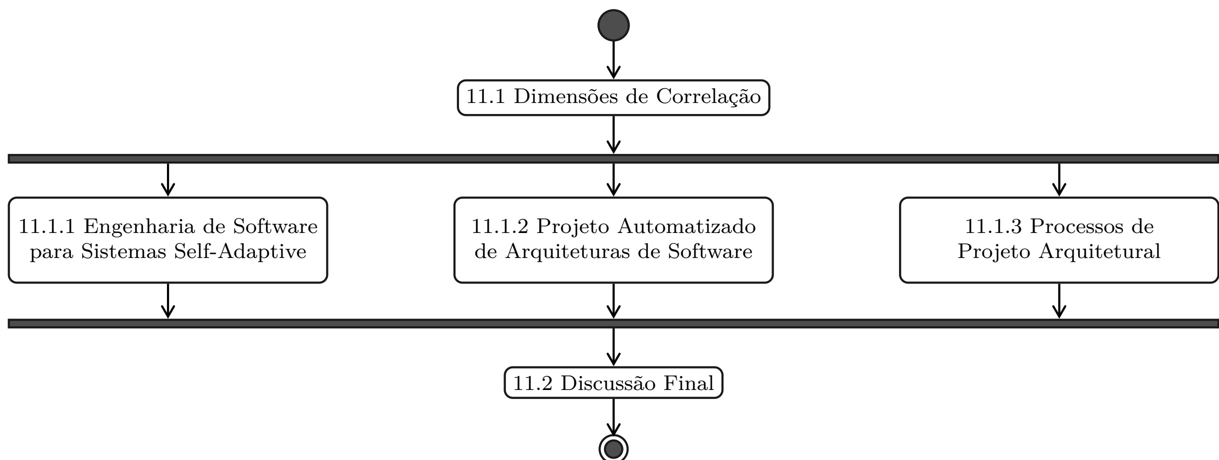


Figura 11.1.: Roteiro do capítulo 11.

11.1. Dimensões de Correlação

As abordagens apresentadas nesta tese estão associadas à utilização de mecanismos existentes e à contribuição de novas soluções em três amplas áreas de pesquisa: *i)* engenharia de *software*

para sistemas *self-adaptive*, *ii)* projeto automatizado (baseado em busca) de arquiteturas de *software*; e *iii)* processos de projeto arquitetural baseados em atributos de qualidade. Portanto, a revisão de literatura apresentada neste capítulo considera trabalhos correlatos em cada uma dessas frentes de pesquisa. Os trabalhos são apresentados em ordem cronológica decrescente.

11.1.1. Engenharia de Software para Sistemas Self-Adaptive

Embora os primeiros sistemas baseados em algoritmos adaptativos tenham surgido há algumas décadas, os esforços para organização sistemática de conhecimento de engenharia voltado para sistemas *self-adaptive* são relativamente recentes. Desde 2006, conferências especializadas no assunto – tais como o SEAMS (*Software Engineering for Adaptive and Self-Managing Systems*) [272], o SASO (*IEEE International Conference on Self-Adaptive and Self-Organizing Systems*) [148] e os seminários Dagstuhl [271] – disponibilizam espaços importantes para formação da comunidade de pesquisa e troca de experiências referentes a engenharia de *software* para sistemas *self-adaptive*. Dentre os tópicos estudados, destacam-se: processos para engenharia de requisitos de autogerenciamento, arquiteturas dinâmicas e autogerenciáveis, validação e verificação em tempo de execução, modelagem explícita de *feedback control loops* e tratamento de incertezas.

Diversos trabalhos para modelagem de *feedback control loops* como elementos arquiteturais de primeira classe vêm sendo propostos nos últimos anos. A motivação é o papel crucial que tais estruturas exercem no atendimento dos requisitos de adaptação e o objetivo é permitir a análise antecipada das propriedades exibidas pelo produto final de *software*. Esta seção apresenta os principais estudos que envolvem técnicas genéricas para modelagem e análise de sistemas *self-adaptive* baseados em *feedback control loops*. Vale ressaltar que um amplo conjunto de soluções para problemas particulares de autogerenciamento está disponível na literatura [239] e não serão aqui abordados por não se tratarem de abordagens genéricas de engenharia de *software* (aplicáveis em cenários diversos de autogerenciamento).

Abordagens Baseadas em Busca para Projeto de Sistemas Self-Adaptive. Os mecanismos apresentados nesta tese constituem, tanto quanto sabemos, a primeira abordagem para projeto de sistemas *self-adaptive* que utiliza soluções baseadas em busca e otimização multiobjetivo [20, 19]. É importante ressaltar que mecanismos para otimização multiobjetivo podem ser utilizados, no projeto de sistemas *self-adaptive*, para viabilizar duas diferentes metas:

1. Otimização como lei de atuação aplicada em tempo de execução (*on-line*): neste cenário, a execução do algoritmo de otimização tem como objetivo a derivação das mudanças arquiteturais (adaptações) que devem ser aplicadas de modo a trazer o sistema de volta aos níveis de serviço desejados. Dessa forma, a otimização constitui alternativa às outras leis de atuação comumente utilizadas – tais como PID ou *Static State Feedback Control* (apresentadas no Capítulo 3, Seção 3.3.4.3).
2. Otimização como técnica *off-line* para projeto arquitetural de sistemas gerenciadores: neste cenário, a otimização é aplicada em tempo de projeto (*off-line*) para decidir – dentre uma série de leis de controle e arquiteturas para autogerenciamento – qual solução deverá ser implementada e implantada. Esta solução (arquitetura) representa a decisão de *trade-off* selecionada, do *Pareto-front*, pelo arquiteto.

As contribuições apresentadas nesta tese estão relacionadas à meta 2 e nenhum trabalho relacionado com mesmo objetivo foi encontrado, tanto quanto sabemos, até o presente momento. Embora seja viável a transferência da infraestrutura aqui proposta para tempo de execução – de modo a suportar controladores reconfiguráveis[1] – tal aspecto não foi avaliado nesta tese e

[1] Controladores que suportam a mudança não só dos valores de parâmetros (controladores adaptativos) mas também da estrutura que define a lei de atuação, transdutores, sensores e atuadores utilizados.

constitui potencial trabalho futuro.

Trabalhos que utilizam otimização para atendimento da meta 1 constituem soluções da área de *Dynamic Adaptive Search-Based Software Engineering* (DASBSE), descrita no Capítulo 3, Seção 3.3.2. Diversos trabalhos baseiam-se em otimização como mecanismo de controle em sistemas *self-adaptive*, com considerável prevalência na área de *cloud computing*. Técnicas de otimização com modelos lineares são utilizadas em [194] para alocação de recursos virtuais. Algoritmos genéticos para otimização de implantações baseada em demandas de processamento e atrasos em filas são utilizados em [310]. MIP (*Mixed Integer Programming*) [311] é utilizada em [59] para otimização de implantação de máquinas virtuais baseada em capacidade de computação, armazenamento e largura de banda e em [34] para otimizar o consumo de energia.

Bin-packing [306] é utilizado em [242] para otimização de alocação de tarefas em ambientes virtualizados e em [289] para otimização de distribuição de carga entre nós virtuais. Heurísticas e algoritmos de fluxo-máximo [108] são utilizados em [296] para gerenciar alocação de recursos em larga escala. *Hill Climbing* [53] vem sendo utilizado para maximização de vazão em serviços [199], minimização do número necessário de réplicas para satisfação de requisitos de qualidade de serviço [216] e busca de componentes ótimos para qualidade de serviço em sistemas distribuídos [327]. Redes de fluxo [6] têm sido utilizadas para particionamento de carga entre nós [41] e maximização de utilização de recursos [299].

Outras Abordagens para Projeto de Sistemas Self-Adaptive. Um tema central nos trabalhos realizados pela comunidade de engenharia de *software* para sistemas *self-adaptive* é a prospecção de métodos para modelagem explícita de *feedback control loops*. Visto que propriedades de tais estruturas exercem impacto substancial nos atributos de qualidade apresentados pela solução final, a representação de *feedback control loops* como elementos arquiteturais de primeira classe é fundamental para viabilizar a análise antecipada das propriedades de autogerenciamento desejadas. Dentre os objetivos, destacam-se: investigação de propriedades SASO (*Stability*, *Accuracy*, *Settling time* e *Overshoot*), estimativa de *overheads* e de robustez de controle, predição de interações não desejadas entre múltiplos *loops*, dentre outros. Alguns trabalhos atuais – analisados a seguir – apresentam contribuições importantes em relação aos objetivos acima descritos.

Em [176], Křikava et al. utilizam modelos arquiteturais centrados em Teoria de Controle e modelagem específica de domínio para suportar a integração facilitada de diversas estratégias de autogerenciamento. Os autores apresentam uma nova linguagem de domínio específico – denominada FCDL (*Feedback Control Definition Language*) – para a definição de arquiteturas de *feedback control loops*. A linguagem suporta operações de composição, distribuição e reflexão, viabilizando a coordenação de múltiplos *loops*. A abordagem foi implementada em uma ferramenta denominada ACTRESS, baseada nas tecnologias *Eclipse Modeling Framework* (EMF) [288] e Xtext/Xbase [94]. A abordagem suporta geração automática de código para a plataforma Akka [149] e checagem de consistência de modelos através de restrições definidas pelo usuário em OCL/Xbase ou usando ferramentas externas de *model checking*.

O trabalho apresentado nesta tese também baseia-se na definição de uma nova linguagem (atividade de metamodelagem), porém o domínio alvo não é diretamente a especificação de *feedback control loops* mas a construção de espaços de projeto voltados a domínios particulares de aplicação. Tal linguagem (DuSE) suportou a criação do modelo (espaço de projeto concreto) que representa as principais dimensões de projeto da área de sistemas *self-adaptive* (SA:DuSE). Embora disponibilize uma linguagem com expressividade considerável no domínio em questão, o trabalho de Křikava não suporta projeto automatizado de arquiteturas e não utiliza mecanismos para manifestação de *trade-offs* entre soluções. Consequentemente – sob o ponto de vista de métodos para captura de conhecimento refinado de projeto – a curva de aprendizado de tal linguagem requer um esforço que não abrevia o tempo necessário para o projeto de boas arquiteturas para sistemas *self-adaptive*. Em contraste, a abordagem aqui proposta requer o uso deste

conhecimento refinado apenas uma vez – durante a elaboração do espaço de projeto concreto. A partir daí, a automação do projeto contribui para uma avaliação mais efetiva das diferentes arquiteturas candidatas disponibilizadas.

Em [110], Iglesia propõe um conjunto de *templates* formais para especificação e verificação de comportamento de autogerenciamento em uma família de sistemas *self-adaptive* distribuídos. Os *templates* contêm padrões para especificação de comportamento dos componentes MAPE-K de um sistema gerenciador e padrões para especificação de propriedades que suportam a verificação da corretude das adaptações criadas. Os autores utilizam autômatos temporizados (*timed automata*) [14], TCTL (*Timed Computation Tree Logic*) [15] e a ferramenta Uppaal [302] para análise das propriedades de autogerenciamento desenvolvidas.

O trabalho de Iglesia compartilha os mesmos objetivos do trabalho de Křikava apresentado acima, diferindo somente no tipo e grau de formalidade da notação utilizada na construção dos modelos. Dessa forma, as mesmas considerações apontadas para o trabalho de Křikava valem para o de Iglesia. Vale ressaltar, entretanto, que o uso de especificações comportamentais formais são importantes para a realização de análises arquiteturais mais sofisticadas.

Uma abordagem para tratamento de mudanças não antecipadas baseada em aprendizado (FUSION) é apresentada por Esfahani et al. em [92]. Os autores propõem o uso de modelos de *features* [113] para representação de conhecimento de projeto e de mecanismos de aprendizado *on-line* [13] para suportar as decisões de adaptação sem requerer uma representação explícita da estrutura interna da aplicação. Uma das grandes vantagens da abordagem é suportar adaptação a mudanças não antecipadas (imprevistas). Embora o FUSION não se limite a uma abordagem de aprendizado em particular, os autores utilizam a técnica *M5 model tree* [285] em função da sua facilidade de treinamento e capacidade de eliminação de *features* insignificantes.

Embora a abordagem proposta nesta tese considere somente *feedback control* como mecanismo viabilizador da lei de atuação, outras estratégias – tais como aprendizado de máquina, gramáticas de grafos ou *policies* – podem ser modeladas como novos pontos de variação da dimensão de lei de controle. O espaço de projeto resultante será, obviamente, ainda maior. Acredita-se, entretanto, que a estratégia de busca baseada em metaheurísticas aqui adotada continuará disponibilizando soluções efetivas (mesmo sendo sub-ótimas) em um tempo de execução aceitável.

Em [307], Villegas et al. propõem um modelo de referência – denominado DYNAMICO – caracterizado pela execução de *loops* em três diferentes níveis: *i)* adaptação a mudanças nos objetivos de controle (ex: adição de novos *Service Level Agreements*); *ii)* adaptação da estrutura do sistema gerenciador que realiza o controle (controle reconfigurável); e *iii)* adaptação dos mecanismos de monitoramento de sistema e de ambiente utilizados. A abordagem define os principais componentes e estratégias para separação de *concerns* entre diferentes *feedback control loops* e justifica a necessidade de autogerenciamento nos três níveis informados acima como importante para obter-se uma maior flexibilidade e robustez nas ações de adaptação.

O DYNAMICO apresenta uma descrição, em notação *ad-hoc*, de uma arquitetura de referência para sistemas *self-adaptive*. Embora represente uma contribuição significativa para arquitetos com pouca experiência no domínio, não há suporte efetivo para a comparação e análise de arquiteturas candidatas e os aninhamentos de *loops* se resumem aos três níveis propostos no trabalho. Em contraste, a abordagem aqui apresentada suporta um número arbitrário de *feedback control loops* aninhados, bem como a detecção de *loops* com interações não desejadas.

Em [317], Weyns et al. apresentam o FORMS: um modelo de referência para especificação formal de sistemas *self-adaptive* distribuídos, que unifica conceitos de três diferentes áreas: reflexão computacional, coordenação de processos distribuídos e adaptação via MAPE-K. A abordagem disponibiliza um pequeno conjunto de elementos de modelagem que captura os principais conceitos de projeto no domínio de sistemas *self-adaptive*. O FORMS utiliza a linguagem de especificação Z [245] para suportar a modelagem e análise de *feedback control loops*. O modelo

de referência é aplicado na especificação de um sistema *self-adaptive* para monitoramento de tráfego de veículos.

Embora o FORMS disponibilize um conjunto expressivo de constructos para o domínio de sistemas *self-adaptive* e apresente bons recursos para análise formal de arquitetural de *feedback control*, nenhum mecanismo para projeto automatizado é apresentado e sua infraestrutura rigorosa para especificação formal de comportamento de autogerenciamento pode implicar em uma curva de aprendizado mais acentuada.

Vogel & Giese, em [308], propõem uma nova linguagem de modelagem para descrição explícita de *feedback control loops* baseada em megamodelos em tempo de execução (múltiplos *models@runtime*) Adicionalmente, um interpretador definindo semânticas de execução de modelos viabiliza a realização das adaptações arquiteturais. A abordagem é caracterizada por quatro aspectos principais: *i)* os *feedback control loops* são explicitamente descritos nos megamodelos; *ii)* a adaptação é especificada em um nível de abstração mais alto, através de operações que atuam nos modelos que constituem os megamodelos; *iii)* os megamodelos são implantados juntos à solução e, portanto, podem ser modificados em tempo de execução; e *iv)* a plataforma disponibiliza suporte nativo à comunicação entre modelos, facilitando a especificação de *loops* de controle aninhados.

Em contraste, a abordagem aqui apresentada baseia-se na utilização de linguagens de modelagem padronizadas e de ampla adoção na indústria, tais como a MOF e a UML. Embora tais aspectos sejam aqui utilizados como mecanismo *off-line* de projeto arquitetural, trabalhos futuros incluem a transferência desta infraestrutura de otimização para o tempo de execução, caracterizando uma solução também baseada em *models@runtime*.

Um *profile* UML para modelagem de *feedback control loops* é apresentado por Hebig em [135]. A abordagem suporta a especificação de múltiplos *loops* e disponibiliza mecanismos básicos para verificação de regras de boa formação de modelos (*warning signals*). O *profile* suporta a especificação de esquemas de controle centralizado e descentralizado. No trabalho aqui apresentado, entretanto, os *profiles* UML representam um mecanismo de suporte à especificação de *feedback control loops*, viabilizando a identificação dos locais de decisão arquitetural, automação das atividades de projeto e detecção de arquiteturas candidatas inválidas.

Finalmente, em [63], Cheng et al. apresentam uma linguagem de adaptação baseada em *utility functions* para suporte a *self-adaptation* na presença de múltiplos objetivos. As possíveis adaptações são modeladas utilizando estruturas rotuladas de Kripke [190] – um tipo particular de autômato finito não-determinístico para especificação comportamental de sistemas. Adaptações conflitantes são resolvidas através da especificação de *utility functions*.

Conforme apresentado no Capítulo 4, métodos de otimização com articulação *a priori* de preferência – tais como *utility functions* – convertem um problema de otimização multiobjetivo em um problema de objetivo único. Porém, a efetividade do método é fortemente influenciada pela escolha do vetor de fatores de preferência. A abordagem aqui apresentada, por outro lado, acomoda a natureza multiobjetivo do problema de projeto de sistemas *self-adaptive* como aspecto essencial do método, como consequência da utilização de técnicas com articulação *a posteriori* de preferência.

A Tabela 11.1 apresenta um resumo das abordagens de engenharia de *software* para sistemas *self-adaptive* descritas nesta seção. Para cada trabalho, são informados: *i)* a lei de atuação utilizada; *ii)* o suporte à manifestação de *trade-offs* de projeto, às atividades de *Model-Driven Engineering* (MDE), à especificação comportamental e a regras de boa formação de soluções; e *iii)* o método de avaliação utilizado no trabalho. O suporte aos itens descritos em *ii)* é indicado como total (✚), parcial (✚) ou ausente (✖).

Trabalho	**Lei de Atuação**	**Manifestação de Trade-offs**	**Atividades MDE**	**Especificação de Comportamento**	**Regras de Boa Formação**	**Avaliação**
Andrade & Macêdo [20]	Teoria de Controle	✚ *Pareto-front*	✖	✖	✚ OCL	Indicadores, Estudo de Caso e Experimento Controlado
Křikava et al. [176]	Teoria de Controle	✖	✙ Geração de Código	✚ *Ptolemy*	✚ OCL	Estudo de Caso
Iglesia [110]	*Policies*	✖	✖	✚ TCTL	✚ TCTL	Estudo de Caso
Esfahani et al. [92]	Aprendizado de Máquina + *Feature Models*	✙ *Utility Functions*	✖	✚ xADL	✖	Estudo de Caso
Villegas et al. [307]	Teoria de Controle	✖	✖	✖	✖	Estudo de Caso
Weyns et al. [317]	Reflexão + Adaptação Arquitetural	✖	✖	✚ Z	✚ Z	Estudo de Caso
Vogel & Giese [308]	*Triple Graph Grammars* (TGG)	✖	✖	✖	✚ TGG	Exemplos Isolados
Hebig [135]	Teoria de Controle	✖	✖	✖	✚ *Warning Signals*	Estudo de Caso
Cheng et al. [63]	*Policies*	✙ *Utility Functions*	✖	✚ Estruturas de Kripke	✖	Estudo de Caso

Tabela 11.1.: Abordagens de engenharia de *software* para sistemas *self-adaptive*.

11.1.2. Projeto Automatizado de Arquiteturas de Software

Esta seção apresenta os principais trabalhos referentes a projeto arquitetural automatizado sem, no entanto, estarem direcionados à construção de sistemas *self-adaptive* em particular. Embora as primeiras ideias referentes a espaços de projeto (ainda não estruturados) tenham aparecido há quase 25 anos [188], trabalhos atuais [276] ainda evidenciam a importância de tais estratégias, dado o número relativamente pequeno de abordagens para sistematização de espaços de projeto presentes na literatura.

Uma abordagem multiobjetivo para síntese de arquiteturas de *software* a partir de diagramas de casos de uso UML é apresentada por Räihä et al. em [254]. Os autores apresentam um espaço de projeto (*design space*) contendo um conjunto de soluções arquiteturais padronizadas e adotam otimização multiobjetivo para evidenciar *trade-offs* entre facilidade de modificação (*modifiability*) e eficiência – ambos a serem maximizados. As soluções padronizadas incluem estilos arquiteturais tais como *message dispatcher* e *client-server*, bem como padrões de projeto tais como *facade*, *mediator*, *strategy*, *adapter* e *template method*.

Em contraste, a abordagem aqui apresentada tem como objetivo realizar o reprojeto de uma arquitetura de *software* inicial (sem recursos para autogerenciamento) de modo que ela passe a apresentar comportamento *self-adaptive*. Entretanto, espaços de projeto que recebem como entrada casos de uso UML (bem como qualquer outro constructo baseado na MOF) podem

também ser definidos na abordagem aqui proposta.

Em [173], Koziolek apresenta uma infraestrutura de metamodelagem para realização de melhorias em arquiteturas, com foco em desempenho e outros atributos. O projeto automatizado é guiado por uma abordagem de otimização multiobjetivo que utiliza Redes de Filas em Camadas (*Layered Queue Network* - LQN) [103] e simulação como funções de *fitness*. A abordagem inclui ainda extensões para utilização de táticas de domínio específico e de uma fase pós-otimização para intensificação das soluções obtidas. Koziolek compara os desempenhos com e sem as extensões propostas, concluindo que uma convergência mais rápida é obtida com o uso das extensões.

Embora a abordagem aqui apresentada adote técnicas similares de otimização e metamodelagem, as mudanças arquiteturais que geram as estratégias de *feedback control* envolvidas demandaram um metamodelo mais expressivo para a construção do espaço de projeto concreto aqui apresentado. O metamodelo proposto nesta tese (linguagem para especificação de espaços de projeto arquitetural) suporta o tratamento de dependências entre dimensões de projeto e regras para detecção de arquiteturas candidatas inválidas. O trabalho de Koziolek tem como objetivo a melhoria de arquiteturas em relação a desempenho e custo de implementação, enquanto a abordagem apresentada nesta tese tem como meta a geração de arquiteturas de sistemas gerenciadores com boas propriedades de controle. Adicionalmente, a abordagem de Koziolek assume que as modificações arquiteturais não mudam funcionalidades (somente refatoração), ao passo que a abordagem aqui proposta permite a inclusão de novos elementos arquiteturais. O trabalho de Koziolek não suporta mudanças em propriedades multivaloradas do modelo. Por outro lado, seu trabalho não apresenta a limitação de que os pontos de variação de uma dimensão não mudam em função do modelo de entrada, conforme apresentado pela abordagem apresentada nesta tese.

Saxena & Karsai propõem em [269] um *metaframework* para exploração de espaços de projeto. O objetivo é a disponibilização de uma infraestrutura flexível para exploração de espaços de projeto de modo independente de domínio e utilizando diversas estratégias de busca. Uma versão preliminar do *metaframework* foi desenvolvida com base em tecnologias tais como o GME (*Generic Modeling Environment*) [192] e apresenta ainda uma linguagem abstrata para exploração de espaços de projeto (ADSEL), uma linguagem para especificação de restrições (CSL) e interpretadores para transformação automática de modelos em problemas de otimização.

A abordagem aqui apresentada adota estratégias similares de exploração de espaços de projeto, porém com foco em projeto arquitetural em vez de resolução geral de problemas (alocação de recursos, roteamento, escalonamento, etc), conforme afirmado pelos autores acima. Adicionalmente, o trabalho aqui apresentado é baseado em tecnologias padronizadas para modelagem de sistemas, em contraste à proposta de novas linguagens apresentada por Saxena & Karsai.

Finalmente, em [218], Mkaouer et al. apresentam uma abordagem escalável para otimização multiobjetivo de operações de refatoração de *software*. Os autores utilizam o NSGA-III para encontrar boas refatorações em relação a 15 métricas de qualidade (funções-objetivo) diferentes. As métricas utilizadas foram: *Weighted Method per Class* (WMC), *Response for a Class* (RC), *Lack of Cohesion of Methods* (LCOM), *Cyclomatic Complexity* (CC), *Number of Attributes* (NA), *Attribute Hiding Factor* (AH), *Method Hiding Factor* (MH), *Number of Lines of Code* (NLC), *Coupling Between Object Classes* (CBO), *Number of Associations* (NAS), *Number of Classes* (NC), *Depth of Inheritance Tree* (DIT), *Polymorphism Factor* (PF), *Attribute Inheritance Factor* (AIF) e *Number of Children* (NOC). A abordagem foi implementada e avaliada em sete projetos de *software* livre e concluiu-se que, em média, 92% dos *code-smells* foram corrigidos com a otimização. A análise estatística de 31 execuções da otimização mostrou que o desempenho é significativamente melhor quando comparado à otimização com o NSGA-II.

Visto que esta tese não aborda problemas de otimização com mais de quatro objetivos, o NSGA-II foi utilizado como *backend* de otimização. Entretanto, a plataforma implementada permite o

uso de diferentes abordagens de otimização e a utilização do NSGA-III para suportar problemas com mais funções-objetivo é parte dos trabalhos futuros. Enquanto a abordagem de Mkaouer et al. tem como meta a refatoração de sistemas, a solução apresentada nesta tese tem como objetivo a extensão de uma arquitetura que representa o sistema gerenciado para inclusão do sistema gerenciador (adição de *feedback control loops*).

A Tabela 11.2 apresenta um resumo das abordagens para automação de projeto arquitetural descritas nesta seção. Para cada trabalho, são informados: *i)* o objetivo; *ii)* a técnica utilizada na automação do projeto; *iii)* os artefatos de entrada e saída; *iv)* as funções de *fitness* empregadas e; *v)* o método de avaliação utilizado.

Trabalho	Objetivo	Técnica de Otimização	Artefato de Entrada	Artefato de Saída	Funções de Fitness	Avaliação
Andrade & Macêdo [20]	Reprojeto Arquitetural	NSGA-II	Diagrama de Componentes e de Implantação	Diagrama de Componentes e de Implantação	Teoria de Controle + definidas pelo usuário	Indicadores, Estudo de Caso e Experimento Controlado
Räihä et al. [254]	Projeto Arquitetural	Algoritmos Genéticos	Diagramas de Caso de Uso e de Sequência	Diagrama de Classes	Métricas CK	Estudo de Caso
Koziolek [173]	Reprojeto Arquitetural	NSGA-II	Diagrama de Componentes e de Implantação	Diagrama de Componentes e de Implantação	LQN + simulação	Indicadores e Estudo de Caso
Saxena & Karsai [269]	Resolução de Problemas Genéricos	Várias	Modelo Representando o Problema	Modelo Representando a Solução	Definidas pelo Usuário	Estudo de Caso
Mkouer et al. [218]	Refatoração	NSGA-III	Código-fonte	Código-fonte	WMC, RFC, LCOM, CC, NA, AH, MH, NLC, CBO, NAS, NC, DIT, PF, AIF e NOC	Experimento com 7 Sistemas

Tabela 11.2.: Abordagens de projeto automatizado de arquiteturas de *software*.

11.1.3. Processos de Projeto Arquitetural Baseados em Atributos de Qualidade

Conforme apresentado no Capítulo 2, os processos de projeto arquitetural guiados por atributos de qualidade baseiam-se na identificação antecipada de tais atributos a partir de requisitos arquiteturalmente significativos e na seleção de táticas de projeto arquitetural que induzem o atendimento destes atributos. Ao colocar os atributos de qualidade como elemento central do processo, dá-se ênfase à prospecção de um mecanismo mais rigoroso de mapeamento entre táticas arquiteturais e grau de atendimento desses atributos de qualidade. Esta seção apresenta os principais processos de projeto arquitetural guiados por atributos de qualidade.

Um método baseado em ontologias para projeto arquitetural (Quark) é apresentado por Ameller em [16]. A solução integra requisitos não-funcionais ao processo de projeto e disponibiliza meios para auxílio à tomada de decisão. O trabalho tem como meta a disponibilização de uma ferramenta para gerenciamento de *design rationale* (ArchiTech) e, em contraste à abordagem aqui apresentada, não suporta projeto automatizado nem meios para manifestação de *trade-offs*.

O método *Attribute-Driven Design* (ADD) é apresentado por Wojcik et al. em [323]. O ADD

define um processo recursivo de projeto que decompõe o sistema através da aplicação de táticas e padrões arquiteturais de modo a satisfazer os atributos de qualidade desejados. As iterações do ADD consistem de fases de planejamento (*plan*), execução (*do*) e verificação (*check*). Na fase de planejamento, os atributos de qualidade mais importantes são selecionados, enquanto na fase de execução os elementos mais adequados ao atendimento dos atributos selecionados são instanciados. Finalmente, na fase de verificação, analisa-se o projeto resultante para determinar se os requisitos são satisfatoriamente atendidos. O ADD, diferente da abordagem proposta nesta tese, define um método mais qualitativo e não automatizado para projeto arquitetural baseado em atributos de qualidade.

Em [8], Al-Naeem et al. apresentam uma abordagem quantitativa para descoberta de arquiteturas com melhor atendimento de metas de qualidade conflitantes. Os autores utilizam o método AHP (*Analytic Hierarchy Process*) [263] para calcular uma pontuação para cada arquitetura candidata. De forma similar às abordagens baseadas em *utility functions*, o AHP também requer o uso de um vetor de fatores de preferência e difere, portanto, do método com articulação *a posteriori* de preferência aqui adotado.

Finalmente, Matinlassi propõe em [209] um método não-automatizado para transformação de modelos arquiteturais (QADA) de acordo com requisitos de qualidade predefinidos. Os autores utilizam um banco de dados de estilos arquiteturais e um conjunto de regras de transformação para guiar a busca por novas arquiteturas. Em contraste, a abordagem aqui apresentada disponibiliza um método mais quantitativo, supondo que métodos confiáveis para avaliação de qualidade estão disponíveis e são utilizados.

A Tabela 11.3 apresenta um resumo das abordagens para projeto arquitetural baseado em atributos de qualidade descritas nesta seção. O método de avaliação de atributos proposto em cada trabalho é identificado como quantitativo ou qualitativo e a técnica de mapeamento entre atributos de qualidade e táticas arquiteturais é apresentada. São informados ainda os atributos de qualidade considerados no trabalho, a forma de representação de conhecimento de projeto e o método de avaliação da abordagem proposta.

Trabalho	**Tipo de Abordagem**	**Técnica de Mapeamento**	**Atributos Considerados**	**Representação de Conhecimento de Projeto**	**Avaliação**
Andrade & Macêdo[20]	Quantitativo	Métricas + Busca (metaheurísticas)	De Domínio Específico (sistemas *self-adaptive*)	Espaços de Projeto	Indicadores, Estudo de Caso e Experimento Controlado
Ameller [16]	Qualitativo	Ontologias → Decisões	Confiabilidade e Usabilidade	Ontologias	Questionários
Wojcik et al. [323]	Qualitativo	Estilos → Atributos	Nenhum Específico	Estilos	Estudos de Caso
Al-Naeem et al. [8]	Quantitativo	Métricas	Facilidade de Modificação, Escalabilidade, Desempenho e Custo	Banco de Dados de Estilos	Estudo de Caso
Matinlassi [209]	Qualitativo	Estilos → Atributos	Portabilidade, Confiabilidade, Disponibilidade e Extensibilidade	Estilos	Estudo de Caso

Tabela 11.3.: Abordagens de processo de projeto arquitetural baseado em atributos de qualidade.

11.2. Discussão Final

O projeto de arquiteturas de *software* que sejam efetivas em relação ao atendimento dos atributos de qualidade desejados é uma tarefa que requer o uso de conhecimento refinado e cuja sistematização apresenta desafios. Com algumas poucas exceções, a maioria dos métodos para projeto e análise de arquiteturas de *software* definem procedimentos qualitativos e manuais, cujo sucesso depende fortemente da experiência e habilidade do arquiteto encarregado de tais atividades. Dentre os principais desafios presentes na definição de processos para projeto e análise de arquiteturas de *software*, destacam-se:

1. Documentação de conhecimento refinado (táticas arquiteturais e arquiteturas candidatas): a transformação de conhecimento tácito em conhecimento estruturado de forma sistemática é fundamental para alavancar o desenvolvimento de arquiteturas mais efetivas. A prática comum envolve o uso de arquiteturas de referência e catálogos de estilos e padrões arquiteturais. Embora tais abordagens já tragam benefícios para uma formação abreviada de bons arquitetos, a falta de suporte para manipulação direta por ferramentas e a dificuldade de associação e comparação de arquiteturas candidatas limita os benefícios destas abordagens no uso rotineiro pela indústria.
2. Mapeamento entre táticas e indução de atributos de qualidade: embora a documentação sistemática de conhecimento refinado já contribua significativamente para o projeto de boas arquiteturas, o objetivo final é a obtenção de produtos de *software* que apresentem os atributos de qualidade desejados. Arquiteturas de notória efetividade são frequentemente caracterizadas por uma combinação sensata de estilos arquiteturais, conectores sofisticados e aspectos comportamentais bem elaborados. Conhecer e documentar quais táticas arquiteturais são mais adequadas à indução de determinados atributos de qualidade são atividades valiosas pois favorecem o reuso de soluções interessantes e tornam o processo de projeto mais previsível.
3. Técnicas eficazes para análise e predição de propriedades: a realização de análises mais rigorosas de modelos arquiteturais permite a detecção antecipada de problemas e a comparação de arquiteturas alternativas em relação a um ou mais atributos de qualidade. Para isso, notações de modelagem com suporte à especificação formal de aspectos estruturais e comportamentais devem ser utilizadas. O objetivo é antecipar, com certa precisão, como os aspectos arquiteturais adotados impactam o atendimento dos atributos de qualidade envolvidos.
4. Documentação de *rationale*: conforme apresentado no Capítulo 2, o conjunto formado pelas intenções, premissas, razões e ponderações adotadas pelo arquiteto durante o processo de projeto é denominado *rationale*. O *rationale* é a descrição do caminho trilhado pelo arquiteto, as decisões por ele tomadas e os motivos para tais ações. Esta informação dificilmente está presente em algum modelo ou diagrama representando a arquitetura e, portanto, precisa ser descrito em algum outro artefato e associado aos elementos arquiteturais correspondentes.
5. Rastreamento entre arquitetura e artefatos de implementação: após o término do seu projeto, uma arquitetura geralmente tem seus módulos detalhados até o ponto em que possam ser efetivamente implementados. Tal atividade geralmente implica em mapeamentos não biunívocos, o que dificulta rastreamentos futuros entre elementos arquiteturais e de implementação, e vice-versa. Estes rastreamentos são importantes para manter os modelos arquiteturais sempre atualizados ou para guiar atividades subsequentes de testes e evolução.
6. Manifestação de *trade-offs* e suporte à tomada de decisão: a natureza frequentemente conflitante dos atributos de qualidade envolvidos torna mais difícil uma ponderação bem informada e criteriosa sobre as arquiteturas candidatas disponíveis. *Trade-offs* não conhecidos

ou a ausência de mecanismos para manifestação explícita de tais *trade-offs* como soluções não-dominadas constituem obstáculos ao projeto de arquiteturas efetivas pois tornam mais provável a adoção de arquiteturas inferiores, decorrentes de algum viés tecnológico, falsa intuição ou conhecimento parcial do espaço de solução em questão.

Os trabalhos correlatos apresentados neste capítulo constituem abordagens interessantes para alguns dos desafios acima citados, mas ainda há espaço para a proposta de soluções que ataquem tais problemas de forma mais completa, integrada e efetiva. Abordagens para projeto automatizado de arquiteturas de *software* requerem a consideração direta dos desafios 1, 3 e 6, enquanto aqueles dedicados a processos de projeto arquitetural baseados em atributos de qualidade atacam primariamente os desafios 1, 2 e 4. Os estudos da área de engenharia de *software* para sistemas *self-adaptive* estão mais interessados no desafio 3, embora os desafios 1 e 2 tenham passado a receber maior atenção recentemente.

As soluções apresentadas nesta tese estão primariamente relacionadas aos desafios 1, 2 e 6 acima apresentados. O item 3 é parcialmente abordado – as funções de *fitness* adotadas permitem algum grau de predição de propriedades mas notações rigorosas para especificação de comportamento não são aqui aplicadas. Embora esta abordagem não ataque diretamente todos os desafios acima citados, acredita-se que o uso de técnicas padronizadas de metamodelagem e abordagens escaláveis para otimização multiobjetivo viabilizam a investigação facilitada dos demais aspectos no futuro.

Capítulo 12

Conclusão

I tramp a perpetual journey.
Walt Whitman (Song of Myself)

Esta tese apresentou uma nova abordagem para projeto e análise automatizados de arquiteturas de *software*, com aplicação particular no domínio de sistemas *self-adaptive* baseados em Teoria de Controle. O trabalho foi motivado pela ausência de mecanismos mais sistemáticos para documentação de conhecimento refinado de projeto de arquiteturas de *software* e pela grande complexidade envolvida na construção de sistemas *self-adaptive* modernos. A captura sistemática de conhecimento de projeto para domínios particulares de aplicação viabiliza um uso rotineiro mais eficiente das abordagens arquiteturais, promove uma análise mais produtiva de soluções alternativas e alavanca o uso de abordagens automatizadas.

Este capítulo está organizado da seguinte forma. A Seção 12.1 apresenta um resumo da tese, reafirmando os problemas abordados, técnicas adotadas e avaliações realizadas. A Seção 12.2 resume as contribuições realizadas neste projeto, enquanto limitações e deficiências da abordagem proposta são discutidas na Seção 12.3. Por fim, a Seção 12.4 apresenta as oportunidades de investigação futura decorrentes do trabalho aqui realizado.

12.1. Resumo da Tese

Embora os paradigmas e tecnologias da engenharia de *software* propostas nos últimos anos tenham promovido melhorias significativas na gerência da complexidade envolvida na construção de sistemas computacionais modernos, acredita-se que os limites da capacidade humana para compreensão e manipulação de artefatos de *software* inevitavelmente se tornará um fator limitante [145]. Dois importantes desdobramentos podem ser identificados nesta afirmação.

Primeiro, a adoção de mecanismos mais sistemáticos e estruturados para a captura, utilização e disseminação de conhecimento refinado de projeto [275] é fundamental para o avanço da prática corrente. Trabalhos relacionados a teorias em engenharia de *software* [282] – em particular, teorias de projeto (*design theories*) [117, 158] – e gerenciamento de conhecimento arquitetural [27] são algumas das frentes atuais de pesquisas que atacam este problema.

Segundo, mesmo com a realização de avanços significativos nas questões acima apresentadas, os limites da capacidade humana para gerenciamento de complexidade vêm sendo continuamente questionados frente às demandas introduzidas pelos sistemas computacionais atuais [168]. Dessa

forma, transferir certas atividades de um processo de desenvolvimento de *software* para os próprios sistemas parece ser um caminho inexorável. Tais capacidades de autogerenciamento em tempo de execução são a principal característica dos sistemas *self-adaptive* [145, 266].

Frente às motivações acima descritas, esta tese apresentou uma abordagem genérica para projeto e análise automatizados de arquiteturas de *software*, baseada na representação sistemática de conhecimento refinado de projeto e na utilização de técnicas de otimização multiobjetivo. Esta infraestrutura foi instanciada de modo a capturar as dimensões de projeto mais proeminentes da área de sistemas *self-adaptive*. O objetivo foi a construção de uma plataforma para manipulação automática de modelos arquiteturais, viabilizando a derivação e análise das diversas arquiteturas candidatas que compõem o espaço de projeto em questão.

Cada arquitetura candidata representa uma possibilidade de projeto. Algumas são menos efetivas, outras, entretanto, podem representar soluções de alta qualidade, difíceis de serem encontradas por arquitetos menos experientes. O formalismo disponibilizado pelas técnicas de otimização multiobjetivo viabilizou a implementação da descoberta automática das arquiteturas mais efetivas do espaço de projeto. Adicionalmente, a abordagem aqui apresentada oferece um mecanismo mais fundamentado para a comparação de arquiteturas candidatas e a descoberta de *trade-offs* arquiteturais.

Um espaço de projeto é modelado neste trabalho como um conjunto finito de dimensões ortogonais de projeto. Um espaço de projeto é sempre criado para um domínio de aplicação em particular e cada dimensão captura um aspecto arquitetural associado a um tipo de decisão importante do domínio. Cada dimensão de projeto define um conjunto de soluções alternativas (pontos de variação) para o aspecto arquitetural em questão. Um ponto de variação, por sua vez, define o conjunto de modificações arquiteturais que implementam a solução representada pelo ponto de variação. Constructos para descoberta de combinações inválidas de pontos de variação permitem a identificação de soluções conflitantes.

A abordagem para projeto automatizado de arquiteturas aqui descrita foi construída de modo a ser independente de domínio de aplicação. O objetivo é que espaços de projeto para outros domínios sejam criados no futuro e que a mesma plataforma de busca seja utilizada para a descoberta de arquiteturas efetivas. Para isso, uma nova linguagem de modelagem – denominada DuSE – foi projetada e construída como parte desta tese. O metamodelo da linguagem implementa os constructos apresentados acima e serve como base para que o motor de otimização desenvolvido manipule as arquiteturas com base em um espaço de projeto para um domínio particular (descrito na linguagem DuSE). A linguagem DuSE, por sua vez, foi especificada utilizando a linguagem de (meta)modelagem MOF (*Meta Object Facility*) [231].

O motor de otimização desenvolvido nesta tese permite a busca automática por soluções que minimizam/maximizam um conjunto de métricas de avaliação arquitetural, definidas como parte do espaço de projeto em uso. O objetivo é viabilizar a descoberta daquelas soluções que representam arquiteturas mais sutis, caracterizadas por combinações menos óbvias de pontos de variação e, portanto, difíceis de serem projetadas por arquitetos iniciantes no domínio em questão. O algoritmo evolucionário de otimização multiobjetivo NSGA-II foi utilizado neste trabalho, mas outras abordagens podem ser facilmente incluídas no futuro.

A linguagem DuSE foi utilizada como metamodelo para criação do SA:DuSE – espaço de projeto para sistemas *self-adaptive* desenvolvido neste trabalho. O SA:DuSE captura decisões arquiteturais referentes a cinco dimensões importantes deste domínio, focadas no desenvolvimento de sistemas *self-adaptive* baseados em Teoria de Controle. Tais dimensões são: cardinalidade de controle (*Single-Input Single-Output* ou *Multiple-Input Multiple-Output*), lei de controle (sete alternativas de solução), técnica de sintonia (sete alternativas de solução), grau de adaptabilidade do controlador (cinco alternativas de solução) e forma de cooperação entre múltiplos *loops* (seis alternativas de solução). Quatro métricas para avaliação de propriedades referentes ao

desempenho das adaptações realizadas pelas arquiteturas foram também definidas como parte do SA:DuSE.

A abordagem foi completamente implementada em uma ferramenta de apoio (DuSE-MT) e três estudos de avaliação foram conduzidos utilizando, cada um, dois modelos de sistemas *self-adaptive*: um servidor *web* e um *cluster* elástico para aplicações *MapReduce*. O estudo 1 teve como objetivo a investigação da expressividade do SA:DuSE na descoberta dos *trade-offs* comumente encontrados no projeto de sistemas *self-adaptive*. A descoberta de 13 arquiteturas alternativas para o caso do servidor *web* e 18 para o caso do *cluster* elástico para aplicações *MapReduce* é um indício da correta captura, no SA:DuSE, do caráter multiobjetivo do problema de projeto de arquiteturas para sistemas *self-adaptive*. Adicionalmente, observou-se – com um nível de significância de 0.01 – que as arquiteturas encontradas em uma única execução de otimização apresentaram efetividade de 95% antes da 229ª iteração no caso do servidor *web* e antes da 216ª iteração no caso do *cluster* elástico para aplicações *MapReduce*.

O estudo 2 investigou com qual acurácia os valores de atributos de qualidade apresentados pelo SA:DuSE são observados em implementações reais das arquiteturas produzidas pelas otimizações. Para isso duas arquiteturas não-dominadas e uma arquitetura dominada foram implementadas com base no servidor *Apache HTTP Server* (para o caso do servidor *web* MIMO) e na plataforma *Apache Hadoop* (para o caso do *cluster* elástico para aplicações *MapReduce*). Os resultados indicam, no pior caso, uma acurácia máxima menor que 70.89% e 74.46% na previsão, respectivamente, do tempo de estabilização e do sobressinal no controle do *Apache HTTP Server*. Para os experimentos com o *Apache Hadoop*, tais valores foram, respectivamente, 78.13% e 74.02%.

Finalmente, o estudo 3 teve como objetivo verificar se a adoção de abordagens baseadas em busca melhora a efetividade e diminui a complexidade das arquiteturas de sistemas *self-adaptive* produzidas, quando comparadas às arquiteturas geradas por processos convencionais de projeto. Adicionalmente, investigou-se se a abordagem baseada em busca facilita a aquisição, por arquitetos iniciantes, de conhecimento refinado de projeto. Para isso, um experimento controlado com 24 estudantes de pós-graduação comparou os produtos da abordagem aqui apresentada com as arquiteturas produzidas por um método baseado em catálogos de estilos arquiteturais para sistemas *self-adaptive*. Os resultados concluem, com um nível de significância de 0.05, que a abordagem baseada em busca melhora a efetividade e diminui a complexidade das arquiteturas. Nada pôde ser concluído em relação à aquisição de conhecimento refinado.

12.2. Contribuições

Os produtos desta tese podem ser classificados em dois grupos: contribuições para projeto e análise arquitetural independentes de domínio e contribuições para projeto e análise de arquiteturas para sistemas *self-adaptive*. As principais contribuições independentes de domínio são:

- A formalização da infraestrutura para espaços de projeto: os constructos apresentados no Capítulo 6 capturam as informações básicas para representação de modificações arquiteturais e definição de soluções alternativas. Esta infraestrutura é genérica o suficiente para ser utilizada em domínios de aplicação diferentes daquele aqui estudado (sistemas *self-adaptive*).
- Uma linguagem para modelagem de espaços de projeto de domínio específico (DuSE): esta linguagem, definida sobre os mesmos pilares que sustentam a linguagem UML, viabiliza a utilização de ferramentas convencionais de modelagem de *software* para a criação, por arquitetos experientes, de novos espaços de projeto para domínios específicos. O metamodelo definido para a linguagem DuSE permite que, no futuro, ferramentas de diferentes

fabricantes implementem funcionalidades para criação e exploração de espaços de projeto arquitetural, mantendo ainda aspectos de interoperabilidade. Detalhes da linguagem DuSE foram discutidos no Capítulo 9.

- Um mecanismo para otimização multiobjetivo de arquiteturas de *software*: o motor de otimização de arquiteturas apresentado no Capítulo 8 disponibiliza os principais recursos para exploração automática de espaços de projeto arquitetural. Esta funcionalidade é importante nos casos, não raros, onde o problema de projeto arquitetural é caracterizado por um espaço de projeto de altíssima cardinalidade. Adicionalmente, os conceitos de solução dominada e *Pareto-optimality* viabilizaram o uso de um mecanismo mais fundamentado para a identificação de *trade-offs* e justificativa de escolha por soluções particulares.
- Uma biblioteca para operações de modelagem e metamodelagem (`Qt5Modeling`): a implementação dos recursos para criação, manipulação e armazenamento de modelos de *software* foi realizada de forma totalmente independente de metamodelo. Com isso, novas linguagens podem ser facilmente suportadas no futuro.
- Um *application framework* para otimização multiobjetivo (`Qt5Optimization`): este módulo define um conjunto de *hot-spots* para configuração do algoritmo de otimização utilizado e dos operadores de seleção, recombinação e mutação adotados. A implementação do algoritmo NSGA-II foi realizada como parte do `Qt5Optimization`.
- Uma ferramenta de suporte ao processo de projeto automatizado de arquiteturas aqui apresentado (DuSE-MT): todas as atividades do processo automatizado de projeto arquitetural apresentado nesta tese foram implementadas, de forma integrada, na ferramenta DuSE-MT. Toda a implementação das operações de busca por arquiteturas foi realizada de forma independente de domínio. Novos espaços de projeto – abordando aspectos arquiteturais de outros domínios de aplicação – podem ser imediatamente utilizados no DuSE-MT.

As contribuições específicas da área de projeto de sistemas *self-adaptive* são:

- Espaço de projeto para sistemas *self-adaptive* (SA:DuSE): um conjunto considerável de conhecimento refinado de projeto na área de sistemas *self-adaptive* foi capturado, de forma sistemática, no espaço de projeto SA:DuSE. A possibilidade de investigação automática de arquiteturas de *feedback control loops* em relação aos cinco aspectos arquiteturais capturados é importante para uma análise mais completa das alternativas disponíveis e adoção daquela arquitetura mais efetiva para o problema em questão. O SA:DuSE pode ser compreendido como uma coleção de possibilidades de projeto arquitetural na área de sistemas *self-adaptive* – semelhante a um catálogo de estilos arquiteturais – porém, neste caso, o conhecimento é sistematicamente armazenado em uma forma mais estruturada e, portanto, passível de manipulação por ferramentas.
- Avaliação experimental do SA:DuSE: além dos estudos que avaliaram o desempenho das execuções de otimização, esta tese inclui experimentos com protótipos reais de *feedback control loops* para o servidor *Apache HTTP* e para a plataforma de *MapReduce Apache Hadoop*. Servidores *web* e ambientes de *cloud computing* são dois dos mais frequentes cenários com demandas de autogerenciamento na atualidade. Adicionalmente, até onde sabemos, o experimento controlado apresentado no Capítulo 10 (Seção 10.2.3) é o segundo experimento controlado envolvendo sistemas *self-adaptive*[1].

12.3. Limitações

Algumas limitações deste trabalho foram identificadas e discutidas nos capítulos anteriores. Primeiro, embora a abordagem aqui proposta possa ser reescrita para outros meta-metamodelos,

[1] O primeiro é aquele realizado por Weyns et al. em [316].

a implementação da linguagem DuSE, apresentada no Capítulo 9, é fortemente dependente da linguagem MOF (*Meta Object Facility*) [231]. Entretanto, a MOF (bem como a UML) é uma linguagem com ampla aceitação na indústria e frequentemente utilizada como base para especificação de novas linguagens de modelagem.

Segundo, o espaço de projeto SA:DuSE proposto nesta tese se limita ao uso de Teoria de Controle como mecanismo viabilizador de autogerenciamento em sistemas *self-adaptive*. Entretanto, outros mecanismos podem ser facilmente suportados através da criação de novas dimensões de projeto. As estratégias evolucionárias para otimização aqui adotadas são particularmente adequadas à exploração de grandes espaços de busca. Portanto, acredita-se que novas dimensões não tenham impacto expressivo na qualidade das arquiteturas encontradas. Esta afirmação, entretanto, não foi investigada neste trabalho.

Terceiro, o uso de técnicas de otimização multiobjetivo limita a abordagem de projeto arquitetural aqui proposta àquelas situações onde o problema é passível de ser modelado como um espaço de busca e um conjunto de funções de *fitness* (métricas de qualidade). Abordagens mais qualitativas, mesmo que parcialmente automatizadas, podem contribuir para uma melhor efetividade do processo. Quarto, a adoção de abordagens de busca baseadas em metaheurísticas implica na não garantia de obtenção de um *Pareto-front* globalmente ótimo. Naquelas situações onde a obtenção da arquitetura globalmente ótima é a única solução aceitável, a abordagem aqui apresentada passa a ser de pouca valia. Acredita-se, entretanto, que o uso de arquiteturas localmente ótimas já constitua solução mais efetiva que aquelas arquiteturas inferiores decorrentes de viés tecnológico ou da falta de análise de um conjunto maior de arquiteturas candidatas.

Finalmente, táticas específicas de domínio para melhoria do desempenho das otimizações arquiteturais não foram adotadas neste trabalho. Definir mecanismos flexíveis para a inclusão de tais táticas pode trazer melhorias na velocidade de convergência das iterações de otimização e na qualidade do *Pareto-front* final encontrado. Limitações mais pontuais, referentes à formalização da infraestrutura para espaços de projeto apresentada no Capítulo 6, foram apresentadas e discutidas naquele capítulo.

12.4. Trabalhos Futuros

Uma série de oportunidades de investigação futura podem ser identificadas para este trabalho. As avaliações apresentadas no Capítulo 10 foram baseadas em dois modelos arquiteturais iniciais: o servidor *web* MIMO e o *cluster* elástico para aplicações *MapReduce*. Novas experimentações podem ser realizadas com o objetivo de investigar como o desempenho das otimizações e a qualidade do *Pareto-front* final obtido variam em função do tamanho do modelo arquitetural inicial. Adicionalmente, estes aspectos podem ser avaliados também em função do tamanho do espaço de projeto utilizado.

Embora de reconhecida eficiência na resolução de problemas de otimização multiobjetivo, o NSGA-II apresenta limitações quando aplicado a problemas com mais de quatro funções-objetivo. Tais limitações são consequência, dentre outras coisas, do alto número de operações realizadas no cálculo da distância de aglomeração. Outras abordagens para otimização multiobjetivo, tais como o SPEA2, NS-PSO e NSGA-III podem ser futuramente analisadas. Em particular, o NSGA-III tem sido aplicado com sucesso em problemas de engenharia de *software* envolvendo até 15 funções-objetivo [218].

Os estudos de avaliação apresentados no Capítulo 10 utilizaram implementações reais de *feedback control loops* com diferentes escolhas para as dimensões de projeto DD2 (cardinalidade de controle), DD3 (lei de controle) e DD4 (técnica de sintonia). Futuras investigações podem incluir implementações dos pontos de variação identificados para as dimensões DD5 (grau de

adaptabilidade do controlador) e DD6 (forma de cooperação entre múltiplos *loops*). Em particular, para o caso do *cluster* para aplicações *MapReduce* baseado no *Hadoop*, diferentes formas de cooperação entre múltiplos *loops* podem ter impacto direto no desempenho e escalabilidade de controle percebidos. Observações empíricas devem comprovar/refutar a hipótese de que a forma de cooperação escolhida impacta diretamente a efetividade de controle e o *overhead* de controle. Adicionalmente, a derivação de métricas preditivas de efetividade de controle e *overhead* de controle podem tornar as operações de análise arquitetural mais completas.

Uma outra contribuição futura promissora é o desenvolvimento de uma plataforma que permita a modificação, em *run-time*, do ponto de variação escolhido para a dimensão DD6 (forma de cooperação entre múltiplos *loops*). Em ambientes altamente dinâmicos, adotar uma única forma de cooperação entre *loops* pode não ser suficiente para apresentar os atributos de qualidade desejados. Variar dinamicamente, portanto, a forma de cooperação entre *loops* (por exemplo, de um controle mais centralizado para outro com algum grau de descentralização) em função das características observadas do sistema e do ambiente pode trazer benefícios interessantes.

Esta tese apresentou um mecanismo para otimização *off-line* de arquiteturas de *software*. Arquitetos devem realizar a criação dos modelos arquiteturais iniciais, executar as otimizações e selecionar a arquitetura que será efetivamente implementada. Um outro potencial trabalho futuro é a transferência de todo o mecanismo de otimização aqui proposto para *run-time*, caracterizando uma abordagem que se aproxima do que hoje é conhecido como *Dynamic Adaptive Search-Based Software Engineering* [127, 128, 130]. Obviamente, o tempo e o *overhead* de execução das otimizações passam a ter importância mais crítica neste cenário.

Experimentações futuras podem investigar algumas premissas assumidas neste trabalho e discutidas no Capítulo 10, tais como a viabilidade de criação, pelos arquitetos, dos modelos arquiteturais iniciais e a capacidade de interpretação dos resultados de otimização obtidos. Algumas limitações apresentadas no Capítulo 6 podem ser atacadas futuramente, tais como a impossibilidade de definição de dimensões cujo número de pontos de variação depende do modelo arquitetural inicial utilizado.

Em relação à ferramenta DuSE-MT, novas funcionalidades para uso de indicadores de otimização, suporte à replicação de execuções de otimização e visualização das soluções obtidas podem ser implementadas futuramente. Espaços de projeto para outros domínios de aplicação podem ser criados e se beneficiar diretamente de toda a infraestrutura independente de domínio para otimização de arquiteturas desenvolvida nesta tese. Como exemplos de domínios de aplicação que se beneficiariam de tais espaços de projeto, pode-se citar: sistemas *self-organizing*, sistemas tolerantes a falhas e sistemas concorrentes.

Finalmente, mas não menos importante, trabalhos a médio e longo prazo podem envolver a prospecção de mecanismos para transformação de espaços de projeto em teorias de projeto (*design theories*), conforme concebidas por Gregor & Jones em [117, 158].

Referências Bibliográficas

[1] Java virtual machine tool interface (jvm ti). `http://docs.oracle.com/javase/7/docs/technotes/guides/jvmti/`. Accesso: 28-05-2014.

[2] An artificial intelligence perspective on autonomic computing policies. In *Proceedings of the Fifth IEEE International Workshop on Policies for Distributed Systems and Networks*, POLICY '04, Washington, DC, USA, 2004. IEEE Computer Society.

[3] Wasif Afzal and Richard Torkar. On the application of genetic programming for software engineering predictive modeling: a systematic review. *Expert Syst. Appl.*, 38(9):11984–11997, 2011.

[4] Wasif Afzal, Richard Torkar, and Robert Feldt. A systematic review of search-based testing for non-functional system properties. *Inf. Softw. Technol.*, 51(6):957–976, Junho 2009.

[5] Nazareno Aguirre and Tom Maibaum. A temporal logic approach to the specification of reconfigurable component-based systems. In *Proceedings of the 17th IEEE International Conference on Automated Software Engineering*, ASE '02, Washington, DC, USA, 2002. IEEE Computer Society.

[6] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[7] Sarel Aiber, Dagan Gilat, Ariel Landau, Natalia Razinkov, Aviad Sela, and Segev Wasserkrug. Autonomic self-optimization according to business objectives. In *ICAC*, pages 206–213. IEEE Computer Society, 2004.

[8] Tariq Al-Naeem, Ian Gorton, Muhammed Ali Babar, Fethi Rabhi, and Boualem Benatallah. A quality-driven systematic approach for architecting distributed software applications. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 244–253, New York, NY, USA, 2005. ACM.

[9] Ahmad Al-Shishtawy, Joel Höglund, Konstantin Popov, Nikos Parlavantzas, Vladimir Vlassov, and Per Brand. Distributed control loop patterns for managing distributed applications. In *Proceedings of the 2008 Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops*, SASOW '08, pages 260–265, Washington, DC, USA, 2008. IEEE Computer Society.

[10] Shaukat Ali, Lionel C. Briand, Hadi Hemmati, and Rajwinder Kaur Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test case generation. *IEEE Trans. Software Eng.*, 36(6):742–762, 2010.

[11] Robert Allen, Rémi Douence, and David Garlan. Specifying and analyzing dynamic software architectures. In *FASE*, pages 21–37, 1998.

[12] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Trans. Softw. Eng. Methodol.*, 6(3):213–249, Julho 1997.

[13] Ethem Alpaydin. *Introduction to Machine Learning.* The MIT Press, 2nd edition, 2010.

[14] Rajeev Alur and David L Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.

[15] Rajeev Alur and Thomas A Henzinger. Logics and models of real time: A survey. In *Real-Time: Theory in Practice*, pages 74–106. Springer, 1992.

[16] David Ameller. *Non-Functional Requirements as drivers of Software Architecture Design.* PhD thesis, Departament de Llenguatges i Sistemes Informàtics, 2014.

[17] Pierre America, Eelco Rommes, and J. Henk Obbink. Multi-view variation modeling for scenario analysis. In Frank van der Linden, editor, *PFE*, volume 3014 of *Lecture Notes in Computer Science*, pages 44–65. Springer, 2003.

[18] Jesper Andersson, Rogério Lemos, Sam Malek, and Danny Weyns. Software engineering for self-adaptive systems. chapter Modeling Dimensions of Self-Adaptive Software Systems, pages 27–47. Springer-Verlag, Berlin, Heidelberg, 2009.

[19] Sandro Santos Andrade and Raimundo José de Araújo Macêdo. Architectural design spaces for feedback control concerns in self-adaptive systems (S). In *The 25th International Conference on Software Engineering and Knowledge Engineering, Boston, MA, USA, June 27-29, 2013.*, pages 741–746. Knowledge Systems Institute Graduate School, 2013.

[20] Sandro Santos Andrade and Raimundo José de Araújo Macêdo. A search-based approach for architectural design of feedback control concerns in self-adaptive systems. In *7th IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2013, Philadelphia, PA, USA, September 9-13, 2013*, pages 61–70. IEEE Computer Society, 2013.

[21] Sandro Santos Andrade and Raimundo José de Araújo Macêdo. Do search-based approaches improve the design of self-adaptive systems ? A controlled experiment. In *2014 Brazilian Symposium on Software Engineering, Maceió, Brazil, September 28 - October 3, 2014*, pages 101–110. IEEE, 2014.

[22] DARPA Broad Agency Announcement. Self-adaptive software. Technical Report 98-12, DARPA (Defense Advanced Research Projects Agency), 1997.

[23] Andrea Arcuri and Xin Yao. A novel co-evolutionary approach to automatic software bug fixing. In *IEEE Congress on Evolutionary Computation*, pages 162–168. IEEE, 2008.

[24] J. Arlow and I. Neustadt. *UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design.* Pearson Education, 2005.

[25] K. J. Åström and T. Hägglund. *PID Controllers: Theory, Design, and Tuning.* Instrument Society of America, Research Triangle Park, NC, 2 edition, 1995.

[26] Karl Johan Åström and Bjorn Wittenmark. *Adaptive Control.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1994.

[27] Muhammad Ali Babar, Torgeir Dingsyr, Patricia Lago, and Hans van Vliet. *Software Architecture Knowledge Management: Theory and Practice.* Springer Publishing Company, Incorporated, 1st edition, 2009.

[28] Shivnath Babu. Towards automatic optimization of mapreduce programs. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 137–142, New York, NY, USA, 2010. ACM.

[29] Victor R Basili and H Dieter Rombach. The tame project: Towards improvement-oriented software environments. *Software Engineering, IEEE Transactions on*, 14(6):758–773, 1988.

[30] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, 3rd edition, 2012.

[31] Alex E. Bell. Death by uml fever. *Queue*, 2(1):72–80, 2004.

[32] Alex E. Bell. Uml fever: Diagnosis and recovery. *Queue*, 3(2):48–56, 2005.

[33] B. Bequette. *Process control: modeling, design, and simulation*. Prentice Hall Press, Upper Saddle River, NJ, USA, first edition, 2002.

[34] Luciano Bertini, Julius C. B. Leite, and Daniel Mossé. Power optimization for dynamic configuration in heterogeneous web server clusters. *J. Syst. Softw.*, 83(4):585–598, April 2010.

[35] Finn Olav Bjørnson and Torgeir Dingsøyr. Knowledge management in software engineering: A systematic review of studied concepts, findings and research methods used. *Inf. Softw. Technol.*, 50(11):1055–1068, Outubro 2008.

[36] Paul Blanchard, Robert L. Devanev, and Glen R. Hall. *Differential Equations (with DE Tools Printed Access Card)*. Cengage Learning, 2011.

[37] V.A. Bloomfield. *Using R for Numerical Analysis in Science and Engineering*. Chapman & Hall/CRC The R Series. Taylor & Francis, 2014.

[38] Barry W Boehm, Ray Madachy, Bert Steece, et al. *Software Cost Estimation with Cocomo II with Cdrom*. Prentice Hall PTR, 2000.

[39] Rainer Böhme and Felix C Freiling. On metrics and measurements. In *Dependability metrics*, pages 7–13. Springer, 2008.

[40] Rainer Böhme and Ralf Reussner. Validation of predictions with measurements. In *Dependability metrics*, pages 14–18. Springer, 2008.

[41] S. H. Bokhari. Partitioning problems in parallel, pipeline, and distributed computing. 37(1):48–57, Janeiro 1988.

[42] Grady Booch. Goodness of fit. *IEEE Software*, 23(6):14–15, 2006.

[43] Grady Booch. The irrelevance of architecture. *IEEE Software*, 24(3):10–11, 2007.

[44] Grady Booch. Architecture as a shared hallucination. *IEEE Software*, 27(1):96–95, 2010.

[45] D.P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly Media, 2005.

[46] Jeremy S. Bradbury, James R. Cordy, Juergen Dingel, and Michel Wermelinger. A survey of self-management in dynamic software architecture specifications. In *Proceedings of the 1st ACM SIGSOFT Workshop on Self-managed Systems*, WOSS '04, pages 28–33, New York, NY, USA, 2004. ACM.

[47] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, 2012.

[48] Frederick P. Brooks. *The Design of Design: Essays from a Computer Scientist*. Addison-Wesley Professional, 1st edition, 2010.

[49] Frederick P. Brooks, Jr. No silver bullet essence and accidents of software engineering. *Computer*, 20(4):10–19, April 1987.

[50] Yuriy Brun, Ron Desmarais, Kurt Geihs, Marin Litoiu, Antónia Lopes, Mary Shaw, and Michael Smit. A design space for self-adaptive systems. In de Lemos et al. [74], pages 33–50.

[51] Edmund Burke, Emma Hart, Graham Kendall, JimNewall, Peter Ross, and Sonia Schulenburg. Hyper-heuristics: An emerging direction in modern search technology, 2003.

[52] Edmund K. Burke, Michel Gendreau, Matthew Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and Rong Qu. Hyper-heuristics: A survey of the state of the art. *Journal of the Operational Research Society*, To Appear.

[53] E.K. Burke and G. Kendall. *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques.* SpringerLink: Springer e-Books. Springer, 2006.

[54] Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. *Pattern-Oriented Software Architecture, Volume 4: A Pattern Language for Distributed Computing.* Wiley, Chichester, UK, 2007.

[55] E.F. Camacho and C. Bordons. *Model Predictive Control.* Advanced Textbooks in Control and Signal Processing. Springer London, 2004.

[56] Carlos Canal, Ernesto Pimentel, and José M. Troya. Specification and refinement of dynamic software architectures. In Patrick Donohoe, editor, *WICSA*, volume 140 of *IFIP Conference Proceedings*, pages 107–126. Kluwer, 1999.

[57] Gianni Di Caro and Marco Dorigo. Ant colonies for adaptive routing in packet-switched communications networks. In A. E. Eiben, Thomas Bäck, Marc Schoenauer, and Hans-Paul Schwefel, editors, *PPSN*, volume 1498 of *Lecture Notes in Computer Science*, pages 673–682. Springer, 1998.

[58] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. Comput. Syst.*, 19(3):332–383, Agosto 2001.

[59] Sivadon Chaisiri, Bu-Sung Lee, and Dusit Niyato. Optimal virtual machine placement across multiple cloud providers. In Markus Kirchberg, Patrick C. K. Hung, Barbara Carminati, Chi-Hung Chi, Rajaraman Kanagasabai, Emanuele Della Valle, Kun-Chan Lan, and Ling-Jyh Chen, editors, *APSCC*, pages 103–110. IEEE, 2009.

[60] Darren Chamberlain, David Cross, and Andy Wardley. *Perl Template Toolkit.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2003.

[61] Luis Chaparro. *Signals and Systems using Matlab.* Academic Press, 2010.

[62] Betty H. Cheng, Rogério Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M. Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi A. Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle. Software engineering for self-adaptive systems. chapter Software Engineering for Self-Adaptive Systems: A Research Roadmap, pages 1–26. Springer-Verlag, Berlin, Heidelberg, 2009.

[63] Shang-Wen Cheng, David Garlan, and Bradley Schmerl. Architecture-based self-adaptation in the presence of multiple objectives. In *Proceedings of the 2006 International Workshop on Self-adaptation and Self-managing Systems*, SEAMS '06, pages 2–8, New York, NY, USA, 2006. ACM.

[64] Paul Clements and Mary Shaw. "The Golden Age of Software Architecture"Revisited. *IEEE Software*, 26(4):70–72, 2009.

[65] P.C. Clements and L.M. Northrop. *Software Product Lines: Practices and Patterns.* The SEI series in software engineering. Addison Wesley Professional, 2002.

[66] Qt Project Community. Qt: Cross-platform application & UI development framework. http://www.qt.io, Outubro 2014.

[67] A. Corazza, S. Di Martino, F. Ferrucci, C. Gravino, F. Sarro, and E. Mendes. How effective is tabu search to configure support vector regression for effort estimation ? In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, PROMISE '10, pages 4:1–4:10, New York, NY, USA, 2010. ACM.

[68] G.W. Corder and D.I. Foreman. *Nonparametric Statistics for Non-Statisticians: A Step-by-Step Approach*. Wiley, 2009.

[69] P.B. Crosby. *Quality is free: the art of making quality certain*. Mentor executive library. New American Library, 1980.

[70] Carlos E. Cuesta, Pablo de la Fuente, and Manuel Barrio-Solárzano. Dynamic coordination architecture through the use of reflection. In *Proceedings of the 2001 ACM Symposium on Applied Computing*, SAC '01, pages 134–140, New York, NY, USA, 2001. ACM.

[71] Ítalo S. Cunha, Jussara M. Almeida, Virgilio Almeida, and Marcos Santos. Self-adaptive capacity management for multi-tier virtualized environments. In *Integrated Network Management*, pages 129–138. IEEE, 2007.

[72] Thomas H. Davenport and Laurence Prusak. *Working Knowledge: How Organizations Manage What They Know*. Harvard Business School Press, Boston, Mass., 2000.

[73] Fabricio Gomes de Freitas and Jerffeson Teixeira de Souza. Ten years of search based software engineering: A bibliometric analysis. In Myra B. Cohen and Mel Ó Cinnéide, editors, *SSBSE*, volume 6956 of *Lecture Notes in Computer Science*, pages 18–32. Springer, 2011.

[74] Rogério de Lemos, Holger Giese, Hausi A. Müller, and Mary Shaw, editors. *Software Engineering for Self-Adaptive Systems II - International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers*, volume 7475 of *Lecture Notes in Computer Science*. Springer, 2013.

[75] V. C. C. de Paula. *A Formal Framework for Specifying Dynamic Software Architectures*. PhD thesis, 1999.

[76] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Janeiro 2008.

[77] K. Deb. *Optimization for Engineering Design: Algorithms and Examples*. Prentice-Hall of India, 2009.

[78] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *Trans. Evol. Comp*, 6(2):182–197, April 2002.

[79] Kalyanmoy Deb and Himanshu Jain. An evolutionary many-objective optimization algorithm using reference-point based non-dominated sorting approach, part I: Solving problems with box constraints. *IEEE*.

[80] Kalyanmoy Deb and Deb Kalyanmoy. *Multi-Objective Optimization Using Evolutionary Algorithms*. John Wiley & Sons, Inc., New York, NY, USA, 2001.

[81] Kalyanmoy Deb, Lothar Thiele, Marco Laumanns, and Eckart Zitzler. *Scalable test problems for evolutionary multiobjective optimization*. Springer, 2005.

[82] Elisabetta Di Nitto, Carlo Ghezzi, Andreas Metzger, Mike Papazoglou, and Klaus Pohl. A journey to highly dynamic, self-adaptive service-based applications. *Automated Software Engg.*, 15(3-4):313–341, Dezembro 2008.

[83] Meinolf Dierkes, editor. *Handbook of organizational learning and knowledge.* Oxford Univ. Press, Oxford, 2001.

[84] Simon Dobson, Spyros Denazis, Antonio Fernández, Dominique Gaïti, Erol Gelenbe, Fabio Massacci, Paddy Nixon, Fabrice Saffre, Nikita Schmidt, and Franco Zambonelli. A survey of autonomic communications. *ACM Trans. Auton. Adapt. Syst.*, 1(2):223–259, Dezembro 2006.

[85] Richard C. Dorf and Robert H. Bishop. *Modern Control Systems (12th Ed.).* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2010.

[86] Norman Richard Draper and Harry Smith. *Applied regression analysis.* Wiley series in probability and mathematical statistics. Wiley, New York [u.a.], 1966.

[87] Michael Earl. Knowledge management strategies: Toward a taxonomy. *J. Manage. Inf. Syst.*, 18(1):215–233, Maio 2001.

[88] M. Easterby-Smith and M.A. Lyles. *The Blackwell Handbook of Organizational Learning and Knowledge Management.* Handbooks in management. Wiley, 2005.

[89] Jens Ehlers and Wilhelm Hasselbring. A self-adaptive monitoring framework for component-based software systems. In Ivica Crnkovic, Volker Gruhn, and Matthias Book, editors, *ECSA*, volume 6903 of *Lecture Notes in Computer Science*, pages 278–286. Springer, 2011.

[90] Jens Ehlers, André van Hoorn, Jan Waller, and Wilhelm Hasselbring. Self-adaptive software system monitoring for performance anomaly localization. In Hartmut Schmeck, Wolfgang Rosenstiel, Tarek F. Abdelzaher, and Joseph L. Hellerstein, editors, *ICAC*, pages 197–200. ACM, 2011.

[91] Markus Endler and Jiawang Wei. Programming generic dynamic reconfigurations for distributed applications. In *CDS*, pages 68–79. IEEE, 1992.

[92] Naeem Esfahani, Ahmed Elkhodary, and Sam Malek. A learning-based framework for engineering feature-oriented self-adaptive software systems. *IEEE Transactions on Software Engineering*, 39(11):1467–1493, 2013.

[93] Christoph Evers, Romy Kniewel, Kurt Geihs, and Ludger Schmidt. The user in the loop: Enabling user participation for self-adaptive applications. *Future Generation Computer Systems*, 2013.

[94] Moritz Eysholdt and Heiko Behrens. Xtext: implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 307–309. ACM, 2010.

[95] Davide Falessi, Giovanni Cantone, and Philippe Kruchten. Do architecture design methods meet architects' needs ? In *Proceedings of the Sixth Working IEEE/IFIP Conference on Software Architecture*, WICSA '07, Washington, DC, USA, 2007. IEEE Computer Society.

[96] Sheikh Umar Farooq, SMK Quadri, and Nesar Ahmad. Metrics, models and measurements in software reliability. In *Applied Machine Intelligence and Informatics (SAMI), 2012 IEEE 10th International Symposium on*, pages 441–449. IEEE, 2012.

[97] Jacques Ferber. *Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1999.

[98] José Luiz Fiadeiro, Antónia Lopes, and Michel Wermelinger. A mathematical semantics for architectural connectors. In *Generic Programming*, pages 178–221, 2003.

[99] Hasso-Plattner-Institute for Software Systems Engineering. Fmc - the apache modeling project. http://www.fmc-modeling.org/projects/apache, Outubro 2014.

[100] Apache Foundation. The apache HTTP server project. http://httpd.apache.org/, Outubro 2014.

[101] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[102] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3 edition, 2003.

[103] Greg Franks, Tariq Omari, C. Murray Woodside, Olivia Das, and Salem Derisavi. Enhanced modeling and solution of layered queueing networks. *IEEE Trans. Software Eng.*, 35(2):148–161, 2009.

[104] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[105] David Garlan, Felix Bachmann, James Ivers, Judith Stafford, Len Bass, Paul Clements, and Paulo Merson. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley Professional, 2nd edition, 2010.

[106] David Garlan, Shang-Wen Cheng, and Andrew Kompanek. Reconciling the needs of architectural description with object-modeling notations. *Sci. Comput. Program.*, 44(1):23–49, 2002.

[107] David Garlan and Bradley Schmerl. Model-based adaptation for self-healing systems. In *Proceedings of the First Workshop on Self-healing Systems*, WOSS '02, pages 27–32, New York, NY, USA, 2002. ACM.

[108] Saul Gass. *An annotated timeline of operations research an informal history*. Kluwer Academic Publishers, New York, 2005.

[109] Michel Gendreau and Jean-Yves Potvin. *Handbook of Metaheuristics*. Springer Publishing Company, Incorporated, 2nd edition, 2010.

[110] Didac Gil de la Iglesia. *A Formal Approach for Designing Distributed Self-Adaptive Systems*. PhD thesis, Linnaeus UniversityLinnaeus University, School of Computer Science, Physics and Mathematics, Department of Media Technology, 2014.

[111] Paolo Giorgini, Manuel Kolp, John Mylopoulos, and Marco Pistore. The tropos methodology: An overview. In *In Methodologies and Software Engineering for Agent Systems, Kluwer*, page 505. Kluwer Academic Press, 2003.

[112] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.

[113] Hassan Gomaa. *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[114] C. Gomez. *Engineering and Scientific Computing with Scilab*. Engineering and scientific computing with Scilab. Birkhäuser Boston, 1999.

[115] Phillip I Good. *Permutation, parametric and bootstrap tests of hypotheses*, volume 3. Springer, 2005.

[116] Ian Gorton. *Essential Software Architecture*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[117] Shirley Gregor. The nature of theory in information systems. *MIS Q.*, 30(3):611–642, Setembto 2006.

[118] Object Management Group. Common warehouse metamodel 1.1 specification. Specification Version 1.1, Volume 1, Object Management Group, March 2003.

[119] Object Management Group. Xml metadata interface (XMI) specification. `http://www.omg.org/spec/XMI/`, Outubro 2014.

[120] Paul Grunbacher, Alexander Egyed, and Nenad Medvidovic. Reconciling software requirements and architectures with intermediate models. *Software and System Modeling*, 3(3):235–253, 2004.

[121] F.M. Gryna. *Quality Planning and Analysis: From Product Development Through Use*. McGraw-Hill Series in Industrial Engineering and Management Science. McGraw-Hill, 2001.

[122] Sherif A. Gurguis and Amir Zeid. Towards autonomic web services: achieving self-healing using web services. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–5, 2005.

[123] Mark Harman. Search based software engineering for program comprehension. In *ICPC*, pages 3–13. IEEE Computer Society, 2007.

[124] Mark Harman. Open problems in testability transformation. In *ICST Workshops*, pages 196–209. IEEE Computer Society, 2008.

[125] Mark Harman. The relationship between search based software engineering and predictive modeling. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, PROMISE '10, pages 1:1–1:13, New York, NY, USA, 2010. ACM.

[126] Mark Harman. Why the virtual nature of software makes it ideal for search based optimization. In David S. Rosenblum and Gabriele Taentzer, editors, *FASE*, volume 6013 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2010.

[127] Mark Harman. Software engineering: An ideal set of challenges for evolutionary computation. In *Proceeding of the Fifteenth Annual Conference Companion on Genetic and Evolutionary Computation Conference Companion*, GECCO '13 Companion, pages 1759–1760, New York, NY, USA, 2013. ACM.

[128] Mark Harman, Edmund K. Burke, John A. Clark, and Xin Yao. Dynamic adaptive search based software engineering. In Per Runeson, Martin Höst, Emilia Mendes, Anneliese Amschler Andrews, and Rachel Harrison, editors, *ESEM*, pages 1–8. ACM, 2012.

[129] Mark Harman and John Clark. Metrics are fitness functions too. In *Proceedings of the Software Metrics, 10th International Symposium*, METRICS '04, pages 58–69, Washington, DC, USA, 2004. IEEE Computer Society.

[130] Mark Harman, John Clark, and Mel Ó Cinnéide. Dynamic adaptive search based software engineering needs fast approximate metrics. San Francisco, USA, 05/2013 2013.

[131] Mark Harman and Bryan F. Jones. Search-based software engineering. *Information & Software Technology*, 43(14):833–839, 2001.

[132] Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. Search-based software engineering: Trends, techniques and applications. *ACM Comput. Surv.*, 45(1):11:1–11:61, Dezembro 2012.

[133] Monson H. Hayes. *Schaum's Outline of Digital Signal Processing*. McGraw-Hill, Inc., New York, NY, USA, 1st edition, 1998.

[134] Jifeng He, Xiaoshan Li, and Zhiming Liu. Component-based software engineering. In *ICTAC*, pages 70–95, 2005.

[135] Regina Hebig, Holger Giese, and Basil Becker. Making control loops explicit when architecting self-adaptive systems. In *Proceedings of the Second International Workshop on Self-organizing Architectures*, SOAR '10, pages 21–28, New York, NY, USA, 2010. ACM.

[136] George T. Heineman and William T. Councill, editors. *Component-based Software Engineering: Putting the Pieces Together*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[137] Brandon Heller, Srini Seetharaman, Priya Mahadevan, Yiannis Yiakoumis, Puneet Sharma, Sujata Banerjee, and Nick McKeown. Elastictree: Saving energy in data center networks. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 17–17, Berkeley, CA, USA, 2010. USENIX Association.

[138] Joseph L. Hellerstein, Yixin Diao, Sujay Parekh, and Dawn M. Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.

[139] Herodotos Herodotou and Shivnath Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *PVLDB*, 4(11):1111–1122, 2011.

[140] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. Starfish: A self-tuning system for big data analytics. In *CIDR*, pages 261–272, 2011.

[141] Michael A. Heroux. Software challenges for extreme scale computing: Going from petascale to exascale systems. *Int. J. High Perform. Comput. Appl.*, 23(4):437–439, Novembro 2009.

[142] C. Hofmeister, P. Kruchten, R.L. Nord, H. Obbink, A. Ran, and P. America. Generalizing a model of software architecture design from five industrial approaches. In *Software Architecture, 2005. WICSA 2005. 5th Working IEEE/IFIP Conference on*, pages 77 –88, 2005.

[143] Christine Hofmeister, Robert Nord, and Dilip Soni. *Applied Software Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[144] Steven Holzner. *Differential Equations for Dummies*. For Dummies, 2008.

[145] Markus C. Huebscher and Julie A. McCann. A survey of autonomic computing – degrees, models, and applications. *ACM Comput. Surv.*, 40(3):7:1–7:28, Agosto 2008.

[146] IBM. Autonomic computing toolkit: Developers guide. Technical report, IBM, Dezembro 2004.

[147] IBM. Build to manage tool. `http://www.ibm.com/developerworks/eclipse/btm`, Dezembro 2013.

[148] IEEE. SASO | IEEE International Conference on Self-Adaptive and Self-Organizing Systems. `http://www.iis.ee.imperial.ac.uk/saso2014/history.php`, Junho 2014.

[149] Typesafe Inc. Akka. `http://akka.io/`, Junho 2014.

[150] Software Engineering Institute. Community software architecture definitions. `http://www.sei.cmu.edu/architecture/start/glossary/community.cfm`, Dezembro 2013.

[151] Petros Ioannou. *Adaptive Control Tutorial (Advances in Design and Control)*. SIAM, 2006.

[152] ISO/IEC. *ISO/IEC 9126. Software engineering – Product quality*. ISO/IEC, 2001.

[153] Himanshu Jain and Kalyanmoy Deb. An evolutionary many-objective optimization algorithm using reference-point based non-dominated sorting approach, part II: Handling constraints and extending to an adaptive approach. *IEEE*.

[154] Raj Jain. *The art of computer systems performance analysis - techniques for experimental design, measurement, simulation, and modeling.* Wiley professional computing. Wiley, 1991.

[155] Anton Jansen and Jan Bosch. Software architecture as a set of architectural design decisions. In *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture*, WICSA '05, pages 109–120, Washington, DC, USA, 2005. IEEE Computer Society.

[156] Mehdi Jazayeri, Alexander Ran, and Frank Van Der Linden. *Software Architecture for Product Families: Principles and Practice.* Addison-Wesley, 2000.

[157] Dawei Jiang, Beng Chin Ooi, Lei Shi, and Sai Wu. The performance of MapReduce: An in-depth study. *Proceedings of the VLDB Endowment*, 3(1-2):472–483, 2010.

[158] David Jones and Shirley Gregor. The anatomy of a design theory. *J. AIS*, 2007.

[159] ShiftOne JRat. Java runtime analysis toolkit. `http://jrat.sourceforge.net/`, Dezembro 2013.

[160] Konstantinos Kakousis, Nearchos Paspallis, and George Angelos Papadopoulos. A survey of software adaptation in mobile and ubiquitous computing. *Enterp. Inf. Syst.*, 4(4):355–389, Novembro 2010.

[161] Karthik Kambatla, Abhinav Pathak, and Himabindu Pucha. Towards optimizing Hadoop provisioning in the cloud. In *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing*, HotCloud'09, Berkeley, CA, USA, 2009. USENIX Association.

[162] Stephen H. Kan. *Metrics and Models in Software Quality Engineering.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[163] Michael Kay. *XSLT 2.0 and XPath 2.0 Programmer's Reference (Programmer to Programmer).* Wrox Press Ltd., Birmingham, UK, UK, 4 edition, 2008.

[164] Rick Kazman, Jai Asundi, and Mark Klein. Quantifying the costs and benefits of architectural decisions. In *Proceedings of the 23rd international conference on Software engineering*, pages 297–306. IEEE Computer Society, 2001.

[165] Rick Kazman, Mark Klein, Mario Barbacci, Tom Longstaff, Howard Lipson, and Jeromy Carriere. The architecture tradeoff analysis method. In *Engineering of Complex Computer Systems, 1998. ICECCS'98. Proceedings. Fourth IEEE International Conference on*, pages 68–78. IEEE, 1998.

[166] Stephen Kell. Rethinking software connectors. In *International Workshop on Synthesis and Analysis of Component Connectors: In Conjunction with the 6th ESEC/FSE Joint Meeting*, SYANCO '07, pages 1–12, New York, NY, USA, 2007. ACM.

[167] Steven Kelly and Risto Pohjonen. Worst practices for domain-specific modeling. *IEEE Software*, 26(4):22–29, 2009.

[168] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, Janeiro 2003.

[169] Jeffrey O. Kephart and Rajarshi Das. Achieving self-management via utility functions. *IEEE Internet Computing*, 11(1):40–48, 2007.

[170] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[171] Mieczyslaw M. Kokar, Kenneth Baclawski, and Yonet A. Eracar. Control theory-based foundations of self-controlling software. *IEEE Intelligent Systems*, 14(3):37–45, Maio 1999.

[172] Fabio Kon, Manuel Román, Ping Liu, Jina Mao, Tomonori Yamane, Claudio Magalhã, and Roy H. Campbell. Monitoring, security, and dynamic configuration with the dynamictao reflective orb. In *IFIP/ACM International Conference on Distributed Systems Platforms*, Middleware '00, pages 121–143, Secaucus, NJ, USA, 2000. Springer-Verlag New York, Inc.

[173] Anne Koziolek. *Automated Improvement of Software Architecture Models for Performance and Other Quality Attributes*. PhD thesis, Institut für Programmstrukturen und Datenorganisation (IPD), Karlsruher Institut für Technologie, Karlsruhe, Germany, July 2011.

[174] Jeff Kramer and Jeff Magee. The evolving philosophers problem: Dynamic change management. *IEEE Trans. Softw. Eng.*, 16(11):1293–1306, Novembro 1990.

[175] Heather Kreger, Ward Harold, and Leigh Williamson. *Java and JMX: Building Manageable Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[176] Filip Křikava, Philippe Collet, and Romain Rouvoy. Integrating adaptation mechanisms using control theory centric architecture models: A case study. In *11th International Conference on Autonomic Computing (ICAC 14)*, Philadelphia, PA, Junho 2014. USENIX Association.

[177] Klaus Krogmann, Michael Kuperberg, and Ralf Reussner. Using genetic search for reverse engineering of parametric behavior models for performance prediction. *IEEE Trans. Software Eng.*, 36(6):865–877, 2010.

[178] P. Kruchten. An ontology of architectural design decisions in software intensive systems. In *2nd Groningen Workshop on Software Variability*, pages 54–61, Dec 2004.

[179] Philippe Kruchten. *The Rational Unified Process: An Introduction*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3 edition, 2003.

[180] Philippe Kruchten, Patricia Lago, and Hans van Vliet. Building up and reasoning about architectural knowledge. In *Proceedings of the Second International Conference on Quality of Software Architectures*, QoSA'06, pages 43–58, Berlin, Heidelberg, 2006. Springer-Verlag.

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

[182] H. T. Kung, F. Luccio, and F. P. Preparata. On finding the maxima of a set of vectors. *J. ACM*, 22(4):469–476, Outubro 1975.

[183] Yu-Kwong Kwok, Shanshan Song, and Kai Hwang. Selfish grid computing: game-theoretic modeling and nas performance results. In *CCGRID*, pages 1143–1150. IEEE Computer Society, 2005.

[184] Robert Laddaga. Active software. In *IWSAS*, pages 11–26, 2000.

[185] Linda M. Laird and M. Carol Brennan. *Software Measurement and Estimation: A Practical Approach (Quantitative Software Engineering Series)*. Wiley-IEEE Computer Society Pr, 2006.

[186] I.D. Landau, R. Lozano, M. M'Saad, and A. Karimi. *Adaptive Control: Algorithms, Analysis and Applications*. Communications and Control Engineering. Springer, 2011.

[187] Thomas G. Lane. Studying software architecture through design spaces and rules. Technical Report CMU-CS-90-175, Carnegie-Mellon University.Computer science. Pittsburgh (PA US), 1990.

[188] Thomas G Lane. Studying software architecture through design spaces and rules. 1990.

[189] William B. Langdon and Mark Harman. Evolving a cuda kernel from an nvidia template. In *IEEE Congress on Evolutionary Computation*, pages 1–8. IEEE, 2010.

[190] François Laroussinie, Nicolas Markey, and Philippe Schnoebelen. On model checking durational kripke structures. In *Foundations of Software Science and Computation Structures*, pages 264–279. Springer, 2002.

[191] Kung-Kiu Lau and Zheng Wang. Software component models. *IEEE Trans. Softw. Eng.*, 33(10):709–724, Outubro 2007.

[192] Akos Ledeczi, Miklos Maroti, Arpad Bakay, Gabor Karsai, Jason Garrett, Charles Thomason, Greg Nordstrom, Jonathan Sprinkle, and Peter Volgyesi. The generic modeling environment. In *Workshop on Intelligent Signal Processing, Budapest, Hungary*, volume 17, 2001.

[193] Rogério Lemos, Holger Giese, HausiA. Müller, Mary Shaw, Jesper Andersson, Marin Litoiu, Bradley Schmerl, Gabriel Tamura, NorhaM. Villegas, Thomas Vogel, Danny Weyns, Luciano Baresi, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Ron Desmarais, Schahram Dustdar, Gregor Engels, Kurt Geihs, KarlM. Göschka, Alessandra Gorla, Vincenzo Grassi, Paola Inverardi, Gabor Karsai, Jeff Kramer, Antónia Lopes, Jeff Magee, Sam Malek, Serge Mankovskii, Raffaela Mirandola, John Mylopoulos, Oscar Nierstrasz, Mauro Pezzè, Christian Prehofer, Wilhelm Schäfer, Rick Schlichting, DennisB. Smith, João Pedro Sousa, Ladan Tahvildari, Kenny Wong, and Jochen Wuttke. Software engineering for self-adaptive systems: A second research roadmap. In Rogério Lemos, Holger Giese, HausiA. Müller, and Mary Shaw, editors, *Software Engineering for Self-Adaptive Systems II*, volume 7475 of *Lecture Notes in Computer Science*, pages 1–32. Springer Berlin Heidelberg, 2013.

[194] Alexander Lenk, Markus Klems, Jens Nimis, Stefan Tai, and Thomas Sandholm. What's inside the cloud ? an architectural map of the cloud landscape. In *Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, CLOUD '09, pages 23–31, Washington, DC, USA, 2009. IEEE Computer Society.

[195] Harold C. Lim, Shivnath Babu, and Jeffrey S. Chase. Automated control for elastic storage. In *Proceedings of the 7th International Conference on Autonomic Computing*, ICAC '10, pages 1–10, New York, NY, USA, 2010. ACM.

[196] Harold C. Lim, Shivnath Babu, Jeffrey S. Chase, and Sujay S. Parekh. Automated control in cloud computing: Challenges and opportunities. In *Proceedings of the 1st Workshop on Automated Control for Datacenters and Clouds*, ACDC '09, pages 13–18, New York, NY, USA, 2009. ACM.

[197] Mikael Lindvall, Ioane Rus, Rajasekhar Jammalamadaka, and Rikin Thakker. Software Tools for Knowledge Management: A DACS State-of-the-Art Report. Technical report, Fraunhofer Center for Experimental Software Engineering, Dezembro 2001.

[198] Ibiso Wokoma Lionel, Lionel Sacks, and Ian Marshall. Biologically inspired models for sensor network design. In *University College London*, page 12, 2002.

[199] Marin Litoiu, Jerome A. Rolia, and Giuseppe Serazzi. Designing process replication and activation: A quantitative approach. *IEEE Trans. Software Eng.*, 26(12):1168–1178, 2000.

[200] Lin Liu and Eric Yu. From requirements to architectural design: Using goals and scenarios. Toronto, Canada, 14/05/2001 2001.

[201] WenQian Liu and Steve M. Easterbrook. Eliciting architectural decisions from requirements using a rule-based framework. In *STRAW*, pages 94–99, 2003.

[202] Zhen Liu, Nicolas Niclausse, and César Jalpa-Villanueva. Traffic model and performance evaluation of web servers. *Perform. Eval.*, 46(2-3):77–100, Outubro 2001.

[203] Lennart Ljung, editor. *System Identification (2Nd Ed.): Theory for the User*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1999.

[204] Antónia Lopes, Michel Wermelinger, and José Luiz Fiadeiro. Higher-order architectural connectors. *ACM Trans. Softw. Eng. Methodol.*, 12(1):64–104, Janeiro 2003.

[205] S. Lynden and O.F. Rana. Coordinated learning to support resource management in computational grids. In *Peer-to-Peer Computing, 2002. (P2P 2002). Proceedings. Second International Conference on*, pages 81– 89, 2002.

[206] Jeff Magee and Jeff Kramer. Dynamic structure in software architectures. In *Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '96, pages 3–14, New York, NY, USA, 1996. ACM.

[207] Sam Malek, George Edwards, Yuriy Brun, Hossein Tajalli, Joshua Garcia, Ivo Krka, Nenad Medvidovic, Marija Mikic-Rakic, and Gaurav S. Sukhatme. An architecture-driven software mobility framework. *Journal of Systems and Software*, 83(6):972–989, 2010.

[208] Robert C Martin. Java and c++ a critical comparison. *Retrieved November*, 10:2006, 1997.

[209] M. Matinlassi, E. Niemelä, and L. Dobrica. *Quality-driven Architecture Design and Quality Analysis Method: A Revolutionary Initiation Approach to a Product Line Architecture.* VTT publications: Valtion Teknillinen Tutkimuskeskus. Technical Research Centre of Finland, 2002.

[210] MATLAB. *version 7.10.0 (R2010a).* The MathWorks Inc., Natick, Massachusetts, 2010.

[211] Christian Alan Mattmann. *Software Connectors for Highly Distributed and Voluminous Data-intensive Systems.* PhD thesis, Los Angeles, CA, USA, 2007. AAI3291773.

[212] Phil McMinn. Search-based software test data generation: A survey: Research articles. *Softw. Test. Verif. Reliab.*, 14(2):105–156, Junho 2004.

[213] Nenad Medvidovic, David S. Rosenblum, David F. Redmiles, and Jason E. Robbins. Modeling software architectures in the unified modeling language. *ACM Trans. Softw. Eng. Methodol.*, 11(1):2–57, Janeiro 2002.

[214] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. Softw. Eng.*, 26(1):70–93, Janeiro 2000.

[215] Nikunj R. Mehta, Nenad Medvidovic, and Sandeep Phadke. Towards a taxonomy of software connectors. In *Proceedings of the 22Nd International Conference on Software Engineering*, ICSE '00, pages 178–187, New York, NY, USA, 2000. ACM.

[216] Daniel A. Menascé, Emiliano Casalicchio, and Vinod Dubey. A heuristic approach to optimal service selection in service oriented architectures. In *Proceedings of the 7th International Workshop on Software and Performance*, WOSP '08, pages 13–24, New York, NY, USA, 2008. ACM.

[217] Daniel Le Metayer. Describing software architecture styles using graph grammars. *IEEE Trans. Software Eng.*, 24(7):521–533, 1998.

[218] Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, Kalyanmoy Deb, and Mel Ó Cinnéide. High dimensional search-based software engineering: Finding tradeoffs among 15 objectives for automating software refactoring using NSGA-III. Technical report, 2010.

[219] Mirko Morandini, Loris Penserini, and Anna Perini. Towards goal-oriented development of self-adaptive systems. In *Proceedings of the 2008 International Workshop on Software Engineering for Adaptive and Self-managing Systems*, SEAMS '08, pages 9–16, New York, NY, USA, 2008. ACM.

[220] David Mosberger and Tai Jin. Httperf – a tool for measuring web server performance. *SIGMETRICS Perform. Eval. Rev.*, 26(3):31–37, Dezembro 1998.

[221] Norman S. Nise. *Control Systems Engineering (6th Ed.)*. John Wiley & Sons, Inc., 2010.

[222] Ikujiro Nonaka and Hirotaka Takeuchi. *The Knowledge-Creating Company. How Japanese Companies Create the Dynamics of Innovation*. Oxford University Press, New York, 1995.

[223] D.A. Norman. *The Design of Everyday Things*. Basic Books, 2002.

[224] Linda M. Northrop. Does scale really matter ? ultra-large-scale systems seven years after the study (keynote). In *ICSE*, page 857, 2013.

[225] Object Management Group. Ontology definition metamodel (omg) version 1.0. Technical Report formal/2009-05-01, Object Management Group, 5 2009.

[226] Object Management Group. OCL 2.2 Specification, Fevereiro 2010.

[227] Gabriela Ochoa, James Walker, Matthew Hyde, and Tim Curtois. Adaptive evolutionary algorithms and extensions to the hyflex hyper-heuristic framework. In *Proceedings of the 12th international conference on Parallel Problem Solving from Nature (PPSN '12)*, pages 418–427. Springer, 2012.

[228] Katsuhiko Ogata. *Modern Control Engineering (5th Ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2009.

[229] OMG. *MOF QVT Final Adopted Specification*. Object Modeling Group, June 2005.

[230] OMG. Software Process Engineering Metamodell SPEM 2.0 OMG Draft Adopted Specification. Technical report, OMG, 2006.

[231] OMG. OMG Meta Object Facility (MOF) Core Specification, Version 2.4.1, August 2011.

[232] OMG. OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1, August 2011.

[233] Alan V. Oppenheim, Alan S. Willsky, and S. Hamid Nawab. *Signals &Amp; Systems (2Nd Ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[234] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, Maio 1999.

[235] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Runtime software adaptation: Framework, approaches, and styles. In *Companion of the 30th International Conference on Software Engineering*, ICSE Companion '08, pages 899–910, New York, NY, USA, 2008. ACM.

[236] Suraj Pandey, Linlin Wu, Siddeswara Guru, and Rajkumar Buyya. A Particle Swarm Optimization-Based Heuristic for Scheduling Workflow Applications in Cloud Computing Environments. In *Proceedings of the 24th IEEE International Conference on Advanced Information Networking and Applications (AINA)*, pages 400–407, Washington, DC, USA, April 2010. IEEE Computer Society.

[237] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Dover Publications, July 1998.

[238] Gisele Pappa, Gabriela Ochoa, Matthew Hyde, Alex Freitas, John Woodward, and Jerry Swan. Contrasting meta-learning and hyper-heuristic research: the role of evolutionary algorithms. *Genetic Programming and Evolvable Machines*, pages 1–33, 2013.

[239] T. Patikirikorala, A. Colman, Jun Han, and Liuping Wang. A systematic survey on the design of self-adaptive software systems using control engineering approaches. In *Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 33 –42, june 2012.

[240] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, and Gerard Florin. Jac: A flexible solution for aspect-oriented programming in java. In Akinori Yonezawa and Satoshi Matsuoka, editors, *Reflection*, volume 2192 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2001.

[241] Marcelo Perazolo. A symptom extraction and classification method for self-management. In Lisandro Zambenedetti Granville, Luciano Paschoal Gaspary, Bruno Schulze, José Neuman de Souza, Iara Machado, and Michael Stanton, editors, *LANOMS*, pages 201–212. UFRGS, 2005.

[242] D.C. Pérez and Universitat Politècnica de Catalunya. Departament d'Arquitectura de Computadors. *Adaptive Execution Environments for Application Servers*. 2008.

[243] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17(4):40–52, Outubro 1992.

[244] Andrei Popovici, Thomas Gross, and Gustavo Alonso. Dynamic weaving for aspect-oriented programming. In *Proceedings of the 1st International Conference on Aspect-oriented Software Development*, AOSD '02, pages 141–147, New York, NY, USA, 2002. ACM.

[245] Ben Potter, David Till, and Jane Sinclair. *An introduction to formal specification and Z*. Prentice Hall PTR, 1996.

[246] John G. Proakis and Dimitris G. Manolakis. *Digital Signal Processing (4rd Ed.): Principles, Algorithms, and Applications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[247] Qt Project. The meta-object system - QtCore 5.3 - documentation. `http://qt-project.org/doc/qt-5/metaobjects.html`, Outubro 2014.

[248] Qt Project. Model/view programming - QtCore 5.3 - documentation. `http://qt-project.org/doc/qt-5/model-view-programming.html`, Outubro 2014.

[249] Qt Project. Object model - QtCore 5.3 - documentation. `http://qt-project.org/doc/qt-5/object.html`, Outubro 2014.

[250] Qt Project. Object trees and ownership - QtCore 5.3 - documentation. `http://qt-project.org/doc/qt-5/objecttrees.html`, Outubro 2014.

[251] Qt Project. The property system - QtCore 5.3 - documentation. `http://qt-project.org/doc/qt-5/properties.html`, Outubro 2014.

[252] Qt Project. Signals & slots - QtCore 5.3 - documentation. `http://qt-project.org/doc/qt-5/signalsandslots.html`, Outubro 2014.

[253] Outi RäIhä. Survey: A survey on search-based software design. *Comput. Sci. Rev.*, 4(4):203–249, Novembro 2010.

[254] Outi Räihä, Kai Koskimies, and Erkki Mäkinen. Multi-objective genetic synthesis of software architecture. In Natalio Krasnogor and Pier Luca Lanzi, editors, *GECCO (Companion)*, pages 249–250. ACM, 2011.

[255] Colin R. Reeves, editor. *Modern Heuristic Techniques for Combinatorial Problems*. John Wiley & Sons, Inc., New York, NY, USA, 1993.

[256] J. Rekers and Andy SchÃŒrr. Defining and parsing visual languages with layered graph grammars. *J. Vis. Lang. Comput.*, 8(1):27–55, 1997.

[257] Peter Roy, Seif Haridi, Alexander Reinefeld, Jean-Bernard Stefani, Roland Yap, and Thierry Coupaye. Formal methods for components and objects. chapter Self Management for Large-Scale Distributed Systems: An Overview of the SELFMAN Project, pages 153–178. Springer-Verlag, Berlin, Heidelberg, 2008.

[258] W.E. Royce and W. Royce. Software architecture: Integrating process and technology. *TRW Quest*, 14(1):2–15, 1991.

[259] N. Rozanski and E. Woods. *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives*. Pearson Education, 2011.

[260] Ioana Rus and Mikael Lindvall. Guest editors' introduction: Knowledge management in software engineering. *IEEE Softw.*, 19(3):26–38, Maio 2002.

[261] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009.

[262] Krzysztof Rzadca, Denis Trystram, and Adam Wierzbicki. Fair game-theoretic resource management in dedicated grids. In *CCGRID*, pages 343–350. IEEE Computer Society, 2007.

[263] Thomas L Saaty. How to make a decision: the analytic hierarchy process. *European journal of operational research*, 48(1):9–26, 1990.

[264] Seyed Masoud Sadjadi, Philip K. McKinley, Betty H. C. Cheng, and R. E. Kurt Stirewalt. Trap/j: Transparent generation of adaptable java programs. In Robert Meersman and Zahir Tari, editors, *CoopIS/DOA/ODBASE (2)*, volume 3291 of *Lecture Notes in Computer Science*, pages 1243–1261. Springer, 2004.

[265] R. K. Sahoo, I. Rish, A. J. Oliner, M. Gupta, J. E. Moreira, S. Ma, R. Vilalta, and A. Sivasubramaniam. Autonomic computing features for large-scale server management and control. In *In AIAC Workshop, IJCAI 2003*, 2003.

[266] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2):14:1–14:42, Maio 2009.

[267] Vivek Sarkar, William Harrod, and Allan E Snavely. Software challenges in extreme scale systems. *Journal of Physics: Conference Series*, 180(1):012045, 2009.

[268] T. Saxena and G. Karsai. A meta-framework for design space exploration. In *Engineering of Computer Based Systems (ECBS), 2011 18th IEEE International Conference and Workshops on*, pages 71–80, 2011.

[269] Tripti Saxena and Gabor Karsai. A meta-framework for design space exploration. In *ECBS*, pages 71–80. IEEE Computer Society, 2011.

[270] Scilab Enterprises. *Scilab: Free and Open Source software for numerical computation*. Scilab Enterprises, Orsay, France, 2012.

[271] selfadaptive.org. Daghstul seminars | software engineering for self-adaptive systems. `http://www.hpi.uni-potsdam.de/giese/public/selfadapt/dagstuhl-seminars/`, Junho 2014.

[272] selfadaptive.org. SEAMS | software engineering for self-adaptive systems. `http://www.hpi.uni-potsdam.de/giese/public/selfadapt/seams/`, Junho 2014.

[273] J. S. Shamma. *The Control Handbook*, chapter Linearization and Gain Scheduling. Electrical Engineering Handbook. Taylor & Francis, Oxford, 1996.

[274] Mary Shaw. Procedure calls are the assembly language of software interconnection: Connectors deserve first-class status. In *Selected Papers from the Workshop on Studies of Software Design*, ICSE '93, pages 17–32, London, UK, UK, 1993. Springer-Verlag.

[275] Mary Shaw. Research toward an engineering discipline for software. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, FoSER '10, pages 337–342, New York, NY, USA, 2010. ACM.

[276] Mary Shaw. The role of design spaces. *IEEE Software*, 29(1):46–50, 2012.

[277] Mary Shaw and Paul Clements. The golden age of software architecture. *IEEE Softw.*, 23(2):31–39, MarÃ§o.

[278] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[279] Herbert A. Simon. *The Sciences of the Artificial (3rd Ed.)*. MIT Press, Cambridge, MA, USA, 1996.

[280] C. L. Simons and I. C. Parmee. Single and multi-objective genetic operators in object-oriented conceptual software design. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, GECCO '06, pages 1957–1958, New York, NY, USA, 2006. ACM.

[281] Bruno Sinopoli, Courtney Sharp, Luca Schenato, Shawn Schaffert, and S. Shankar Sastry. Distributed control applications within sensor networks. In *IEEE Proceedings Special Issue on Distributed Sensor Networks*, pages 1235–1246, 2003.

[282] Dag I. K Sjøberg, Tore Dybå, Bente Cecilie Dahlum Anda, and Jo Erskine Hannay. *Building Theories in Software Engineering*, chapter 12, pages 312–336. Springer-Verlag London, 2008.

[283] J.J.E. Slotine and W.A. LI. *Applied Nonlinear Control*. Prentice Hall, 1991.

[284] Steven W. Smith. *The Scientist & Engineer's Guide to Digital Signal Processing*. California Technical Pub. http://www.dspguide.com, 1997.

[285] Dimitri P Solomatine. Data-driven modelling: paradigm, methods, experiences. In *Proc. 5th international conference on hydroinformatics*, pages 1–5, 2002.

[286] W.M. Spears. *Evolutionary Algorithms: The Role of Mutation and Recombination*. Natural Computing Series. Springer, 2000.

[287] Thomas Stahl, Markus Voelter, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006.

[288] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.

[289] Malgorzata Steinder, Ian Whalley, and David M. Chess. Server virtualization in autonomic management of heterogeneous workloads. *Operating Systems Review*, 42(1):94–95, 2008.

[290] Roy Sterritt, Mike Hinchey, and Emil Vassev. Self-managing software. In *Encyclopedia of Software Engineering*, pages 1072–1081. 2010.

[291] Allen Stubberud. *Schaum's Outline of Feedback and Control Systems*. McGraw-Hill, Inc., New York, NY, USA, 2nd edition, 1994.

[292] Jerry Swan, Ender Özcan, and Graham Kendall. Co-evolving add and delete hyper-heuristics. In *Proceedings of the 9th International Conference on the Practice and Theory of Automated Timetabling (PATAT'12)*, 2012.

[293] Daniel Sykes, William Heaven, Jeff Magee, and Jeff Kramer. From goals to components: A combined approach to self-management. In *Proceedings of the 2008 International Workshop on Software Engineering for Adaptive and Self-managing Systems*, SEAMS '08, pages 1–8, New York, NY, USA, 2008. ACM.

[294] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[295] Gabriele Taentzer. Agg: A graph transformation environment for modeling and validation of software. In John L. Pfaltz, Manfred Nagl, and Boris Böhlen, editors, *AGTIVE*, volume 3062 of *Lecture Notes in Computer Science*, pages 446–453. Springer, 2003.

[296] Chunqiang Tang, Malgorzata Steinder, Michael Spreitzer, and Giovanni Pacifici. A scalable application placement controller for enterprise data centers. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 331–340, New York, NY, USA, 2007. ACM.

[297] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009.

[298] Gerald Tesauro. Reinforcement learning in autonomic computing: A manifesto and case studies. *IEEE Internet Computing*, 11(1):22–30, 2007.

[299] L. Beril Toktay and Reha Uzsoy. A capacity allocation problem with integer side constraints. *European Journal of Operational Research*, 109(1):170–182, 1998.

[300] Walt Truszkowski, Mike Hinchey, James L. Rash, and Christopher Rouff. Nasa's swarm missions: The challenge of building autonomous software. *IT Professional*, 6(5):47–52, 2004.

[301] Jeff Tyree and Art Akerman. Architecture decisions: Demystifying architecture. *IEEE Softw.*, 22(2):19–27, MarÃ§o 2005.

[302] Uppsala and Aalborg Universities. Uppaal. `http://www.uppaal.org/`, Junho 2014.

[303] Tijs van Dam and Koen Langendoen. An adaptive energy-efficient mac protocol for wireless sensor networks. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*, SenSys '03, pages 171–180, New York, NY, USA, 2003. ACM.

[304] Axel van Lamsweerde. From system goals to software architecture. In *SFM*, pages 25–43, 2003.

[305] Luis M. Vaquero, Luis Rodero-Merino, and Rajkumar Buyya. Dynamically scaling applications in the cloud. *SIGCOMM Comput. Commun. Rev.*, 41(1):45–52, Janeiro 2011.

[306] Vijay V. Vazirani. *Approximation Algorithms*. Springer-Verlag New York, Inc., New York, NY, USA, 2001.

[307] Norha Villegas, Gabriel Tamura, Hausi Müller, Laurence Duchien, and Rubby Casallas. DYNAMICO: A Reference Model for Governing Control Objectives and Context Relevance in Self-Adaptive Software Systems. In Rogerio de Lemos, Holger Giese, Hausi Müller, and Mary Shaw, editors, *Software Engineering for Self-Adaptive Systems 2*, volume 7475 of *LNCS*, pages 265–293. Springer, Agosto 2012.

[308] Thomas Vogel and Holger Giese. A language for feedback loops in self-adaptive systems: Executable runtime megamodels. In *Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2012)*, pages 129–138. IEEE Computer Society, 6 2012.

[309] Pieter Vromant, Danny Weyns, Sam Malek, and Jesper Andersson. On interacting control loops in self-adaptive systems. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '11, pages 202–207, New York, NY, USA, 2011. ACM.

[310] Hiroshi Wada, Junichi Suzuki, and Katsuya Oba. Queuing theoretic and evolutionary deployment optimization with probabilistic slas for service oriented clouds. In *SERVICES I*, pages 661–669. IEEE Computer Society, 2009.

[311] Christopher Wallace. *Mixed Integer Programming Heuristics*. ProQuest / UMI, 2011.

[312] Qing-Guo Wang. Handbook of pi and pid controller tuning rules, aidan o'dwyer, imperial college press, london, 375pp, isbn 1-86094-342-x, 2003. *Automatica*, 41(2):355–356, 2005.

[313] Xiaoying Wang, Dong-Jun Lan, Xing Fang, Meng Ye, and Ying Chen. A resource management framework for multi-tier service delivery in autonomic virtualized environments. In *NOMS*, pages 310–316. IEEE, 2008.

[314] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2 edition, 2003.

[315] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 364–374, Washington, DC, USA, 2009. IEEE Computer Society.

[316] Danny Weyns, M Usman Iftikhar, and Joakim Söderlund. Do external feedback loops improve the design of self-adaptive systems ? a controlled experiment. In *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 3–12. IEEE Press, 2013.

[317] Danny Weyns, Sam Malek, and Jesper Andersson. Forms: Unifying reference model for formal specification of distributed self-adaptive systems. *TAAS*, 7(1):8, 2012.

[318] Danny Weyns, Bradley R. Schmerl, Vincenzo Grassi, Sam Malek, Raffaela Mirandola, Christian Prehofer, Jochen Wuttke, Jesper Andersson, Holger Giese, and Karl M. Göschka. On patterns for decentralized control in self-adaptive systems. In de Lemos et al. [74], pages 76–107.

[319] David R. White, John Clark, Jeremy Jacob, and Simon M. Poulding. Searching for resource-efficient programs: Low-power pseudorandom number generators. In *Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation*, GECCO '08, pages 1775–1782, New York, NY, USA, 2008. ACM.

[320] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.

[321] Thareendhra Wijayasiriwardhane and Richard Lai. Component Point: A system-level size measure for component-based software systems. *J. Syst. Softw.*, 83(12):2456–2470, Dezembro 2010.

[322] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, and Björn Regnell. *Experimentation in Software Engineering*. Springer, 2012.

[323] Rob Wojcik, Felix Bachmann, Len Bass, Paul C. Clements, Paulo Merson, Robert Nord, and William G. Wood. Attribute-Driven Design (ADD), Version 2.0. Technical report, Software Engineering Institute, Novembro 2006.

[324] Dingyu Xue, YangQuan Chen, and Derek P. Atherton. *Linear Feedback Control: Analysis and Design with MATLAB (Advances in Design and Control)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1 edition, 2008.

[325] Da-Qian Zhang, Kang Zhang, and Jiannong Cao. A context-sensitive graph grammar formalism for the specification of visual languages. *Comput. J.*, 44(3):186–200, 2001.

[326] Yuanyuan Zhang, Anthony Finkelstein, and Mark Harman. Search based requirements optimisation: Existing work and challenges. In Barbara Paech and Colette Rolland, editors, *REFSQ*, volume 5025 of *Lecture Notes in Computer Science*, pages 88–94. Springer, 2008.

[327] Tao Zheng, C. Murray Woodside, and Marin Litoiu. Performance model estimation and tracking using optimal filters. *IEEE Trans. Software Eng.*, 34(3):391–406, 2008.

[328] Eckart Zitzler, Joshua Knowles, and Lothar Thiele. Multiobjective optimization. chapter Quality Assessment of Pareto Set Approximations, pages 373–404. Springer-Verlag, Berlin, Heidelberg, 2008.

[329] Eckart Zitzler and Lothar Thiele. Multiobjective optimization using evolutionary algorithms - a comparative case study. In *Parallel problem solving from nature - PPSN V*, pages 292–301. Springer, 1998.

Apêndice A

Especificação do SADuSE UML Profile

A.1. Visão Geral

Este apêndice descreve o *profile* UML criado para suportar a espaço de projeto para sistemas *self-adaptive* proposto nesta tese (SA:DuSE).

A.2. Descrição dos Elementos do Profile

A.2.1. Componentes e Interfaces de Processo

A.2.1.1. ProcessComponent

Indica que o component UML em questão é um componente de processo, ou seja, apresenta uma dinâmica passível de ser gerenciada por algum tipo de controlador. `ProcessComponent` é um estereótipo abstrato, utilizado para viabilizar expressões OCL que identificam genericamente os diversos componentes de processo. Estereótipos derivados de `ProcessComponent` adicionam atributos e restrições específicos das classes de componentes de processo sendo por eles modeladas.

Extensões: `UML::Component`

Generalizações: nenhuma

Associações:

- /processPort: `Port` [1..*]
  {subsets `UML::EncapsulatedClassifier::ownedPort`, ordered, union}
  - Conjunto formado pelas portas do componente cujo tipo é uma interface anotada com o estereótipo `ProcessInterface` (ou algum subtipo dele). É um subconjunto (*subsets*) de `UML::EncapsulatedClassifier::ownedPort`. A porta não deve ser conjugada, ou seja, ela representa uma interface provida pelo componente.
  - Especificação OCL:

```
1: self.base_Component.ownedPort->select(
2:    not(isConjugated) and
3:    type.oclIsTypeOf(Interface) and
4:    type.oclAsType(Interface).
5:       extension_ProcessInterface <> null
6: )->asOrderedSet()
```

- /monitorablePort: `Port` [1..*]
  {subsets `processPort`, ordered}
  - Conjunto formado pelas portas do componente cujo tipo é uma interface anotada com o estereótipo `MonitorableInterface`. É um subconjunto (*subsets*) de `processPort`.
  - Especificação OCL:

```
1: self.processPort->select(
2:    type.oclAsType(Interface).
3:       extension_MonitorableInterface <> null
4: )->asOrderedSet()
```

- /controllablePort: `Port` [1..*]
  {subsets `processPort`, ordered}
  - Conjunto formado pelas portas do componente cujo tipo é uma interface anotada com o estereótipo `ControllableInterface`. É um subconjunto (*subsets*) de `processPort`.
  - Especificação OCL:

```
1: self.processPort->select(
2:    type.oclAsType(Interface).
3:       extension_ControllableInterface <> null
4: )->asOrderedSet()
```

Atributos: nenhum

Restrições: nenhuma

Operações Adicionais: nenhuma

A.2.1.2. SISOProcessComponent

Indica que o component UML em questão é um componente de processo cuja dinâmica é um sistema SISO: *Single-Input Single-Output* (vide Seção 3.2, pág. 30). `SISOProcessComponent` é um estereótipo abstrato.

Extensões: nenhuma

Generalizações: `ProcessComponent` (pág. 269)

Associações: nenhuma

Atributos:

- /monitorablePort: `Port` [1]
  {redefines `SADuSE::ProcessComponent::monitorablePort`}
  - Redefine a propriedade para que o componente possua somente uma interface monitorável.

- /controllablePort: `Port` [1]
  {redefines `SADuSE::ProcessComponent::controllablePort`}
  - Redefine a propriedade para que o componente possua somente uma interface controlável.

Restrições:

[1] Um `SISOComponentProcess` descreve um processo SISO e, portanto, deve possuir exatamente uma porta monitorável:

```
1: self.monitorablePort->size() = 1
```

[2] Um `SISOComponentProcess` descreve um processo SISO e, portanto, deve possuir exatamente uma porta controlável:

```
1: self.controllablePort->size() = 1
```

[3] O tipo da porta controlável deve possuir (atuar como *client* de) apenas uma dependência anotada com o estereótipo `SISOImpact`:

```
1: self.controllablePort.type.clientDependency->one(
2:     extension_SISOImpact <> null
3: )
```

[4] O alvo (*supplier*) da dependência anotada com o estereótipo `SISOImpact` deve ser o tipo da porta monitorável do componente:

```
1: self.controllablePort.type.clientDependency->exists(
2:     extension_SISOImpact <> null and
3:     self.monitorablePort.type = supplier->first()
4: )
```

A.2.1.3. TFProcessComponent

Indica que o component UML em questão é um componente de processo SISO cuja dinâmica é descrita através de uma função de transferência no domínio do tempo discreto (vide Seção 3.3.4.2, pág. 51).

Extensões: nenhuma

Generalizações: `SISOProcessComponent` (pág. 270)

Associações: nenhuma

Atributos:

- tfNum: `Real` [1..*]
  {ordered}
  - Coeficientes do polinômio (em função de z) que representa o numerador da função de transferência.
- tfDen: `Real` [1..*]
  {ordered}

  - Coeficientes do polinômio (em função de z) que representa o denominador da função de transferência.

Restrições:

[1] A ordem do polinômio presente no numerador deve ser menor ou igual à ordem do polinômio presente no denominador (o sistema deve ser causal):

```
1: self.tfNum->size() <= self.tfDen->size()
```

Operações Adicionais:

[1] A operação `systemOrder()` retorna a ordem do sistema dinâmico representado pelo componente de processo:

```
1: context TFProcessComponent::systemOrder():Integer
2:    return = tfDen->size()-1
```

A.2.1.4. FOPDTProcessComponent

Indica que o component UML em questão é um componente de processo SISO cuja dinâmica é descrita através de um modelo FOPDT – *First-Order Plus Dead Time* (vide Seção 3.3.4.2, pág. 53).

Extensões: nenhuma

Generalizações: `SISOProcessComponent` (pág. 270)

Associações: nenhuma

Atributos:

- R: `Real` [1]
  - Ganho estático.
- L: `Real` [1]
  - *Dead time* (também conhecido como *time lag*). É obtido através da intersecção do eixo do tempo com a reta tangente de maior inclinação durante a resposta transiente do sistema.
- T: `Real` [1]
  - *Time constant*. É definido como o tempo necessário, após o *dead time*, para que o sistema atinja 63% $(1 - e^{-1})$ da sua saída medida, em regime estacionário.

Restrições: nenhuma

Operações Adicionais: nenhuma

A.2.1.5. MIMOProcessComponent

Indica que o component UML em questão é um componente de processo cuja dinâmica é um sistema MIMO: *Multiple-Input Multiple-Output* (vide Seção 3.2, pág. 30). Anotações SISO podem ser conjuntamente utilizadas neste estereótipo. `MIMOProcessComponent` é um estereótipo abstrato.

Extensões: nenhuma

Generalizações: `ProcessComponent` (pág. 269)

Associações: nenhuma

Atributos:

- /mo: `Integer` [1]
  - Número de sinais de saída.
  - Especificação OCL:

```
1: self.monitorablePort->size()
```

- /mi: `Integer` [1]
  - Número de sinais de entrada.
  - Especificação OCL:

```
1: self.controllablePort->size()
```

Restrições:

[1] Um `MIMOComponentProcess` descreve um processo MIMO e, portanto, deve possuir mais de uma porta monitorável:

```
1: self.mo > 1
```

[2] Um `MIMOComponentProcess` descreve um processo MIMO e, portanto, deve possuir mais de uma porta controlável:

```
1: self.mi > 1
```

[3] Os tipos de cada porta controlável devem possuir (atuar como *client* de) apenas uma dependência anotada com o estereótipo `MIMOImpact`:

```
1: self.controllablePort->forAll(type.clientDependency->one(
2:    extension_MIMOImpact <> null
3: ))
```

[4] Os alvos (*suppliers*) da dependência anotada com o estereótipo `MIMOImpact` devem ser os tipos das portas monitoráveis do componente:

```
1: self.controllablePort->forAll(
2:    type.clientDependency->exists(
3:       extension_MIMOImpact <> null and
4:       self.monitorablePort.type->includesAll(supplier)
5:    )
6: )
```

Operações Adicionais: nenhuma

A.2.1.6. SSProcessComponent

Indica que o component UML em questão é um componente de processo MIMO cuja dinâmica é descrita através de um modelo de espaço de estados (vide Seção 3.3.4.2, pág. 52).

Extensões: nenhuma

Generalizações: `MIMOProcessComponent` (pág. 272)

Associações: nenhuma

Atributos:

- n: `Integer` [1]
  - Número de variáveis de estado.
- A: `Real` [1..*]
  {ordered}
  - Matriz que indica como cada variável de estado é influenciada por valores passados de cada variável de estado.
- B: `Real` [1..*]
  {ordered}
  - Matriz que indica como cada variável de estado é influenciada por valores passados de cada sinal de entrada.
- C: `Real` [1..*]
  {ordered}
  - Matriz que indica como cada sinal de saída pode ser calculado em função das variáveis de estado.

Restrições:

[1] A matriz **A** deve ser uma matriz $n \times n$:

```
1: self.A->size() = n*n
```

[2] A matriz **B** deve ser uma matriz $n \times m_I$:

```
1: self.B->size() = n*mi
```

[3] A matriz **C** deve ser uma matriz $m_O \times n$:

```
1: self.C->size() = mo*n
```

Operações Adicionais: nenhuma

A.2.1.7. ProcessInterface

Indica que a interface UML em questão é uma interface que controla ou monitora a dinâmica de um componente de processo. Esta interface só pode ser utilizada para definir tipos de portas presentes em componentes de processo. `ProcessInterface` é um estereótipo abstrato.

Extensões: `UML::Interface`

Generalizações: nenhuma

Associações: nenhuma

Atributos:

- x: `Integer [0..1]`
  - Índice da interface quando presente em componentes de processo MIMO.

Restrições:

[1] Interfaces de processo só podem ser utilizadas para especificar tipos de portas de componentes anotados com o estereótipo `ProcessComponent` ou `Controller` (ou qualquer subtipo deles):

```
1: self.base_Interface.namespace.allOwnedElements()->select(
2:    oclIsKindOf(TypedElement) and
3:    oclAsType(TypedElement).type = self.base_Interface
4: )->forAll(
5:    oclIsTypeOf(Port) and
6:    (
7:       oclAsType(Port).owner.oclAsType(Component)->forAll(
8:          .base_ProcessComponent <> null or
9:          .base_Controller <> null
10:      )
11:   )
12: )
```

Operações Adicionais: nenhuma

A.2.1.8. MonitorableInterface

Indica que a interface UML em questão é uma interface monitorável, ou seja, aquela que disponibiliza uma API para obtenção dos valores da saída medida do sistema-alvo. Esta interface só pode ser utilizada para definir tipos de portas presentes em componentes de processo.

Extensões: nenhuma

Generalizações: `ProcessInterface` (pág. 274)

Associações: nenhuma

Atributos: nenhum

Restrições: nenhuma

Operações Adicionais: nenhuma

A.2.1.9. ControlableInterface

Indica que a interface UML em questão é uma interface controlável, ou seja, aquela que disponibiliza uma API para informar os valores da entrada de controle do sistema-alvo. Esta interface só pode ser utilizada para definir tipos de portas presentes em componentes de processo.

Extensões: nenhuma

Generalizações: `ProcessInterface` (pág. 274)

Associações: nenhuma

Atributos: nenhum

Restrições:

[1] Uma interface controlável deve possuir (atuar como *client* de) uma dependência anotada com o estereótipo `SISOImpact` ou `MIMOImpact`:

```
1: self.base_Interface.clientDependency->exists(
2:     extension_SISOImpact <> null or extension_MIMOImpact <> null
3: )
```

Operações Adicionais: nenhuma

Índice Remissivo

A
Adaptação
- baseada em agentes inteligentes, 38
- baseada em busca, 40
- baseada em formalização de estrutura, 41
- baseada em Teoria de Controle, 42

Algoritmo genético, 74
Arquitetura candidata, 118
Arquitetura de software, 12
Arquitetura inválida, 119
Atributos de qualidade
- de arquitetura, 17
- de domínio específico, 16
- de negócio, 16
- de sistema, 16
- definição, 16

C
Componente de software, 13
Conector de software, 13
Configuração arquitetural, 14
Conjunto de soluções não-dominadas, 68
Conjunto Pareto-ótimo global, 69
Conjunto Pareto-ótimo local, 69
Controle
- adaptativo, 55
- em malha aberta, 44
- em malha fechada, 44
- espaço de estados, 54
- PID, 53

D
Dimensão de projeto, 115
Dominância, 68

E
Equação-diferença, 49
Escalonamento de ganho, 52
Espaço arquitetural viável, 119
Espaço de busca, 64
Espaço de decisão, 64, 118
Espaço de projeto, 116
Espaço de projeto específico de aplicação, 117
Espaço-objetivo, 66

F
FOPDT, 51
Função de transferência, 49
Função-objetivo, 64

G
Gestão de conhecimento de projeto arquitetural, 20

H
Heurística, 65
Hiper-heurística, 65

I
Instância de dimensão de projeto, 116
IPDT, 52

L
Linguagem de domínio específico, 22
Linguagem de propósito geral, 22
Loop de adaptação
- análise, 35
- atuação, 37
- definição, 31
- monitoramento, 34
- planejamento, 36

M
Mecanismo de articulação de preferência, 67
Meta Object Facility (MOF), 23
Metaheurística, 65
Metamodelagem, 22
Metamodelo, 22
Metamodelo arquitetural, 109

Métrica de qualidade, 119
Model-driven (Software) Engineering (MDE), 21
Model-Driven Architecture (MDA), 22
Modelo arquitetural, 14, 110
Modelo de espaço de estados, 50
Modelo paramétrico de aproximação, 51
Modificação arquitetural, 112

N
Notação de modelagem arquitetural, 15
NSGA-II, 78

O
Operador
  de mutação, 77
  de recombinação, 76
  de seleção, 76
  de seleção por torneio aglomerado, 80
Ordenação por aglomeração, 79
Ordenação por não-dominância, 78
Otimização, 63

P
Pareto-front, 69
Ponto de variação, 113
Precisão (accuracy), 46
Problema convexo de otimização multiobjetivo, 67
Problema de otimização multiobjetivo, 65
Problema de programação linear, 65
Problema de programação não-linear, 64
Processo
  de desenvolvimento de software, 17
  de projeto arquitetural, 17

R
Robustez, 48

S
Sinal, 43
Sistema
  BIBO-estável, 46
  definição, 43
  dinâmico, 44
  estático (memoryless), 44
  invariante no tempo, 46
  linear, 45
  MIMO, 44
  SISO, 44
Sistema de controle, 43
Sistema self-adaptive, 28
Sobressinal (overshoot), 47

T
Tempo de estabilização (settling time), 47

U
Unified Modeling Language (UML), 24

V
Vetor candidato, 118
Vetor de fatores de preferência, 67