



UNIVERSIDADE FEDERAL DA PARAÍBA
CENTRO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

Seleção Automatizada de Componentes de Software
Orientada por Métricas Estruturais e
Informações de Reúso

Jailton Maciel Alexandre Segundo

João Pessoa – PB
Agosto de 2013

Jailton Maciel Alexandre Segundo

**Seleção Automatizada de Componentes de Software
Orientada por Métricas Estruturais e
Informações de Reúso**

Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Informática da Universidade Federal da Paraíba como requisito parcial para obtenção do título de Mestre em Informática (Sistemas de Computação).

Orientador: Prof. Dr. Glêdson Elias da Silveira

João Pessoa – PB

Agosto de 2013

A382s Alexandre Segundo, Jailton Maciel.

Seleção automatizada de componentes de software orientada por métricas estruturais e informações de reúso / Jailton Maciel Alexandre Segundo.-- João Pessoa, 2013.

111f. : il.

Orientador: Glêdson Elias da Silveira

Dissertação (Mestrado) - UFPB/CCEN

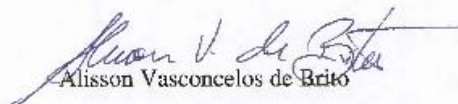
1. Informática. 2. Desenvolvimento Baseado em Componentes (DBS). 3. Engenharia de Software Baseada em Busca (ESBB). 4. Seleção de componentes. 5. Métricas de software.

UFPB/BC

CDU: 004(043)

Ata da Sessão Pública de Defesa de Dissertação de
Mestrado de **Jailton Maciel Alexandre Segundo**,
candidato ao Título de Mestre em Informática na
Área de Sistemas de Computação, realizada em 30
de Agosto de 2013.

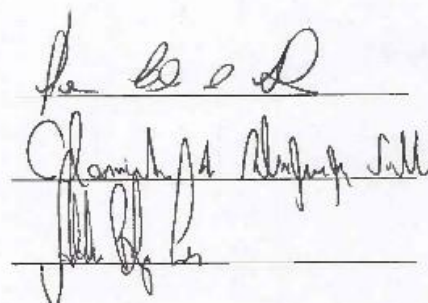
Ao trigésimo dia do mês de Agosto do ano dois mil e treze, às nove horas, no laboratório 2 da Escola Superior de Redes - Universidade Federal da Paraíba, reuniram-se os membros da Banca Examinadora constituída para examinar o candidato ao grau de Mestre em Informática, na área de "*Sistemas de Computação*", na linha de pesquisa "*Computação Distribuída*", o Sr. **Jailton Maciel Alexandre Segundo**. A comissão examinadora foi composta pelos professores doutores: GLEDSON ELIAS DA SILVEIRA (PPGI-UFPB), Orientador e Presidente da Banca, CLAUIRTON DE ALBUQUERQUE SIEBRA (PPGI-UFPB), examinador interno e ADILSON BARBOZA LOPES (UFRN) como examinador externo. Dando início aos trabalhos, o professor GLEDSON ELIAS DA SILVEIRA cumprimentou os presentes, comunicou aos mesmos a finalidade da reunião e passou a palavra ao candidato para que o mesmo fizesse, oralmente, a exposição do trabalho de dissertação intitulado "*Seleção Automatizada de Componentes de Software Orientada por Métricas Estruturais e Informações de Reuso*". Concluída a exposição, o candidato foi arguido pela Banca Examinadora que emitiu o seguinte parecer: "*Aprovado*". Assim sendo, deve a Universidade Federal da Paraíba expedir o respectivo diploma de Mestre em Informática na forma da lei e, para constar, eu, professor Alisson Vasconcelos de Brito, Coordenador deste programa, servindo de secretário, lavrei a presente ata que vai assinada por mim e pelos membros da Banca Examinadora. João Pessoa, 30 de Agosto de 2013.


Alisson Vasconcelos de Brito

Prof. Dr. Gledson Elias da Silveira
Orientador (PPGI-UFPB)

Prof. Dr. Clauirton de Albuquerque Siebra
Examinador Interno (PPGI-UFPB)

Prof. Dr. Adilson Barboza Lopes
Examinador externo (UFRN)



AGRADECIMENTOS

Primeiramente gostaria de expressar minha profunda gratidão a Deus.

Agradeço aos meus pais, pelo esforço que fizeram para me proporcionar uma educação de qualidade. Além de me fornecerem conselhos e carinho, demonstraram paciência nesse processo da minha vida que é o mestrado. É ilimitado o número de palavras para descrever toda a estrutura que meus pais me ofereceram para eu prosseguir na minha pesquisa. Também agradeço a minha irmã pela companhia enquanto eu escrevia esta dissertação.

Ao restante da família, gostaria de agradecer todo o apoio e torcida durante minha trajetória na pós-graduação. Sempre demonstravam que se importavam com meu progresso acadêmico, e isso me empenhava mais ainda quando, porventura, eu enfrentasse momentos difíceis.

Em especial, demonstro gratidão e reconhecimento pelo meu orientador Prof. Glêdson Elias, que com base em sua extensa experiência me guiou e me motivou para a conclusão desse trabalho.

Agradeço a Talles Brito pela ajuda e suporte desde minha iniciação científica, onde o mesmo me acolheu no laboratório de pesquisa Compose e me introduziu os conhecimentos básicos para a pesquisa acadêmica.

Agradeço também aos professores da banca avaliadora pelas contribuições do trabalho desenvolvido: Prof. Adilson e Prof. Claurton.

Aos integrantes, Bruno Luna, Diego Morais, Larissa Barbosa, Vinícius Souza, do grupo de pesquisa Compose no período de minha passagem pelo laboratório, agradeço imensamente pelos momentos de discussões sobre a pesquisa de cada um, assim como pelos momentos de diversão.

E por fim, agradeço aos meus amigos, Gutenberg Botelho e Thiago Ribeiro, de todo o curso e mestrado pelo companheirismo e amizade, em que compartilhamos bons momentos.

RESUMO

A grande dificuldade de selecionar componentes de software ainda é um obstáculo para alcançar o sucesso do Desenvolvimento Baseado em Componentes (DBC). Com o crescimento do mercado de componentes, a tendência é sempre aumentar o número de opções para montagem de aplicações em diferentes contextos, tornando impraticável a seleção manual. Tratando de um problema com gigantesco espaço de busca e complexo, é requerida a automatização efetuada por técnicas de otimização. O trabalho proposto visa automatizar o processo de seleção de componentes utilizando técnicas da Engenharia de Software Baseada em Busca (ESBB), cuja técnica de otimização é orientada por métricas estruturais (avaliam as conexões entre componentes de uma arquitetura de software) e informações de reúso (i.e., valores correspondentes ao reúso de software agregados ao próprio componente). As métricas utilizadas neste contexto de seleção de componentes têm o propósito de avaliar a perspectiva estrutural de uma instância arquitetural, já que elas preveem possíveis problemas de integração entre implementações de componentes produzidas por terceiros, além de que muitos problemas ignoram essa perspectiva e só focam no componente em si. Já as informações de reúso trazem consigo uma alternativa para representar a percepção dos desenvolvedores sobre os atributos de qualidade dos componentes de software em cenários de reúso, tais como: o grau de satisfação dos consumidores que já adquiriram tal componente e o número de *downloads* do mesmo. A avaliação da proposta é conduzida através de experimentos, que são validados aplicando testes estatísticos.

Palavras-chave: DBC, ESBB, seleção de componentes, métricas de software.

ABSTRACT

The great difficulty of selecting software components is still an obstacle to achieving the success of Component-Based Development (CBD). With the growing market for components, the tendency is always to increase the number of options for assembling applications in different contexts, making it impractical to manual selection. Dealing with a problem of gigantic and complex search space, it is required automation performed by optimization techniques. The proposed approach aims to automate the process of selecting components using techniques of Search-Based Software Engineering (SBSE), whose optimization technique is driven by structural metrics (i.e., connections between components of a software architecture) and information reuse (i.e., aggregated values to the component itself). The metrics used in this component selection context are intended to assess the structural perspective of an architectural instance, since they predict possible integration problems between implementations of components produced by third parties. Note that other proposals ignore this perspective and focus only on the component itself. In addition, reuse information can bring an alternative to represent the perception of the developers about the quality attributes of the software components in a reuse scenario as: the degree of consumers' satisfaction who have already purchased this component and the number of downloads of it. The proposed evaluation is carried out through experiments, which are validated by applying statistical tests.

Keywords: CBD, SBSE, component selection, software metrics.

LISTA DE ILUSTRAÇÕES

Figura 1 – Espaço dessas soluções em uma paisagem de adequação.....	26
Figura 2 – Ciclo de um algoritmo genético simples.....	30
Figura 3 – Visão geral da abordagem proposta.	33
Figura 4 – Exemplo visual de uma arquitetura de software.	34
Figura 5 – Descrição da especificação de componente <i>ReservationSystem</i>	36
Figura 6 – Descrição da especificação de interface <i>IMakeReservation</i>	37
Figura 7 – Trecho de um artefato X-ARM e a árvore correspondente a esse trecho.	42
Figura 8 – Expressão regular da linguagem definida pela técnica <i>BranchGuide</i>	43
Figura 9 – Exemplos de consultas com ramificações.....	43
Figura 10 – Consulta com operador lógico.	45
Figura 11 – Especificação de componente cujos candidatos serão buscados.....	47
Figura 12 – Consultas que representam uma operação	47
Figura 13 – Listas de interfaces similares que serão utilizadas para buscar componentes candidatos.....	48
Figura 14 – Consultas que retornarão especificações de componentes candidatas.....	49
Figura 15 – Diferenças entre interfaces e suas consequências.	52
Figura 16 – Avaliação de uma conexão entre componentes candidatos.	53
Figura 17 – Relações entre os conjuntos.	54
Figura 18 – Métrica referente à avaliação da conexão.	55
Figura 19 – Classificação das relações entre os conjuntos de operações	56
Figura 20 – Comparação por similaridade	58
Figura 21 – Arquitetura na perspectiva de grafos.....	65
Figura 22 – Ciclo de via do AG proposto neste trabalho	66
Figura 23 – Representação genética escolhida	69

Figura 24 – Recombinação uniforme	72
Figura 25 – Criação de componentes candidatos que são variações de alguma especificação.	79
Figura 26 – Comparação do desempenho qualitativo das soluções.	81
Figura 27 – Hipóteses nula e alternativa do teste estatístico.	82
Figura 28 – Arquitetura-modelo para a avaliação experimental.	84
Figura 29 – <i>Fitness</i> da estrutura da arquitetura	88
Figura 30 – <i>Fitness</i> correspondente a avaliação na perspectiva dos componentes candidatos..	90

LISTA DE TABELAS

Tabela 1 – Exemplos de problemas de busca nas principais áreas da engenharia de software.	23
Tabela 2 – Elementos pertencentes à arquitetura apresentada anteriormente.	35
Tabela 3 – Identificadores dos conjuntos de operações.	54
Tabela 4 – Comparação quantitativa entre as abordagens algorítmicas investigadas neste trabalho.	89
Tabela 5 – Comparação entre os trabalhos relacionados.....	97

LISTA DE ABREVIATURAS E SIGLAS

AG	Algoritmo Genético
COTS	<i>Commercial Off-The-Shelf</i>
DBC	Desenvolvimento Baseado em Componentes
ESBB	Engenharia de Software Baseada em Busca
SOA	<i>Service-Oriented Architecture</i>
UML	<i>Unified Modeling Language</i>
XML	<i>eXtensible Markup Language</i>
X-ARM	<i>XML Asset Representation Model</i>

SUMÁRIO

Agradecimentos	iv
Resumo	v
Abstract	vi
Lista de Ilustrações	vii
Lista de Tabelas	ix
Lista de Abreviaturas e Siglas	x
Sumário	xi
Introdução	14
1.1. Motivação.....	15
1.2. Objetivos do Trabalho.....	16
1.3. Organização do Trabalho	17
Fundamentação Teórica	18
2.1. Desenvolvimento Baseado em Componentes	18
2.2. Arquitetura de Software	21
2.3. Engenharia de Software Baseada em Busca	22
2.3.1. Ingredientes Essenciais.....	24
2.3.2. Meta-heurísticas Comumente Utilizadas.....	27
2.3.3. Problemas de Otimização Multi-objetivos	28
2.4. Algoritmos Genéticos.....	29
2.5. Considerações Finais.....	31
Abordagem Proposta.....	32
3.1. Visão Geral da Abordagem Proposta.....	33
3.2. Projeto Arquitetural.....	34

3.3. Recuperação de Informações de Reúso e Componentes.....	38
3.3.1. Informações de Reúso	39
3.3.2. Consultas ao Serviço de Busca.....	41
3.3.3. Consultas dos Componentes Candidatos.....	45
3.4. Definição de Métricas	49
3.4.1. Métricas em Termos da Estrutura Arquitetural	50
3.4.2. Métricas em Termos dos Componentes Candidatos	57
3.4.2.1. Métrica Referente ao Nível de Similaridade	57
3.4.2.2. Métrica Referente às Informações de Reúso	60
3.4.2.3. Métrica Geral Referente aos Componentes Candidatos.....	62
3.5. Seleção de Componentes.....	63
3.5.1. Problema NP-Completo	63
3.5.2. Meta-Heurística Escolhida	66
3.5.3. Representação Genética	68
3.5.4. Seleção.....	69
3.5.5. Operadores.....	71
3.5.6. Função de adequação.....	73
3.6. Considerações Finais.....	75
Avaliação Experimental.....	76
4.1. Criação do Ambiente Experimental	77
4.2. Avaliação.....	80
4.3. Realização dos Experimentos	83
4.3.1. Avaliação em Termos da Estrutura da Arquitetura	87
4.3.2. Avaliação em Termos dos Componentes	89
4.3.3. Considerações Referentes aos Experimentos	91
Trabalhos Relacionados.....	93

5.1. Algoritmos de Aproximação para o Problema da Seleção de Componentes de Software.....	93
5.2. Uma Abordagem Evolucionária Baseada em Métricas para o Problema de Seleção de Componentes	95
5.3. Melhoria do Processo de Seleção de Componentes Usando Algoritmos Genéticos para Desenvolvimento Baseado em Componentes.....	96
5.4. Comparativo entre os Trabalhos Relacionados.....	97
Considerações Finais	100
6.1. Limitações e Trabalhos Futuros	102
Referências	105

Capítulo 1

Introdução

Com o passar do tempo, o mercado competitivo impõe sempre um crescimento tecnológico e espera-se que sistemas de software também cresçam continuamente, ou seja, aumentando a sua escala e complexidade. Além disso, novos produtos devem ser construídos o mais rápido possível para não se tornarem obsoletos em relação aos seus concorrentes. Então é requerido que os ciclos de produção dos elementos de software tenham custos e tempo minimizados. Avanços na Engenharia de Software têm contribuído para o aumento da produtividade, e um campo interessante dessa disciplina é o reúso de software, o qual impacta positivamente na redução do tempo de desenvolvimento, dos custos e dos riscos [Sametinger97]. Portanto, o reúso melhora não só o ciclo de desenvolvimento do produto, mas também o próprio produto.

Umas das abordagens de pesquisa em reúso de software é o Desenvolvimento Baseado em Componentes (DBC), em que as unidades para composição de sistemas de software são denominadas componentes. Esta abordagem apresenta o importante fundamento de, no desenvolvimento de sistemas, não ter que demandar tempo em elaborar uma solução que fora resolvida por componentes desenvolvidos para outros contextos de aplicação. Várias aplicações em diferentes domínios são construídas em etapas bem definidas (guiadas pelos conceitos de DBC), como selecionar componentes para posteriormente integrar os componentes escolhidos.

Um dos problemas mais desafiadores do DBC é realizar a seleção de componentes, onde o arquiteto de software ou algum tomador de decisão (i.e., gerente) defronta-se com o desafio de montar uma configuração (instância) arquitetural, com cada componente dessa instância tendo uma grande variedade de opções com funcionalidades equivalentes ou similares, porém com características qualitativas distintas. A tendência de tal variedade é sempre aumentar devido à existência do mercado de componentes de software *Commercial Off-The-Shelf* (COTS), ou seja, componentes desenvolvidos por terceiros e posteriormente distribuídos no mercado [Torchiano04].

Idealmente, sistemas construídos por componentes de software seriam integrados perfeitamente uns aos outros, onde cada um executaria suas funcionalidades normalmente. Entretanto, na prática, os componentes desenvolvidos por terceiros geralmente não se adequam entre si, necessitando de esforço para adaptação caso eles sejam escolhidos para fazerem parte de algum sistema [Becker06]. Isso evidencia que a questão de incompatibilidades de integração entre componentes deve ser tratada na etapa de seleção do DBC, pois caso componentes muito inconsistentes entre si sejam escolhidos, o custo para adaptação será muito alto e impactará no tempo e custo do desenvolvimento. O tratamento de incompatibilidades de integração deve ser reconhecido como inevitável, uma tarefa crucial para a Engenharia de Software Baseada em Componentes [Becker06].

1.1. Motivação

Mesmo obtendo relatos de sucesso e avanços, a indústria ainda não aderiu massivamente ao reúso de software, e, conseqüentemente, ao reúso de componentes. Dentre as limitações enumeradas por Sametinger em relação ao reúso de software, podemos destacar a enorme dificuldade de selecionar componentes de software reusáveis [Sametinger97]. Questões como incompatibilidades de integração prejudicam alcançar a consistência entre uma arquitetura de um sistema e os componentes disponíveis no mercado, resultando em um problema persistente na Engenharia de Software Baseada em Componentes [Mancebo05][Becker06]. A seleção de componentes tem se mostrado ser a etapa de maior risco do Desenvolvimento Baseado em Componentes [Port04].

Não só o meio acadêmico, como o industrial vem demonstrando um crescente interesse no problema de seleção de componentes. Nessa linha de pesquisa, ambos os meios podem ser considerados interdependentes, pois a indústria aplica e compartilha (em publicações) suas abordagens para o problema, além de financiar pesquisas. Do outro lado, a pesquisa acadêmica gera diversos estudos que são analisados, comparados, porventura combinados, e adotados em algum domínio de aplicação industrial [Land08]. Pesquisas no âmbito da indústria referentes à seleção de componentes podem ser observadas em trabalhos como [Ardimento04].

A maioria dos processos de seleção leva em consideração apenas atributos de qualidade do próprio componente (i.e., performance, manutenibilidade, etc) para avaliá-lo. De fato, a qualidade dos componentes influencia diretamente na qualidade da arquitetura [Vitharana03b], porém métricas estruturais também são interpretadas como métricas de

qualidade [Sedigh-Ali01], pois estima a efetividade da integração de todos os componentes na arquitetura de software. Quanto maior tal efetividade, menor será a quantidade de incompatibilidades geradas nessa integração, consequentemente, o custo para adaptação dos componentes diminui. Incompatibilidades de componentes têm sido investigadas por ambas as perspectivas: arquitetural (i.e., conexões entre os componentes da arquitetura) e informações de reuso (i.e., valores correspondentes ao reuso de software agregados ao próprio componente) [Becker06].

Escolher cada componente de software adequado para uma configuração arquitetural é uma tarefa muito importante e complexa para ser realizada de forma manual (*ad hoc*), que é baseada apenas na experiência de um arquiteto de software. Como cada componente escolhido da arquitetura possui vários candidatos com funcionalidades correspondentes, a quantidade de possibilidades geradas apresenta complexidade exponencial, onde a base é o número médio de componentes candidatos e o expoente é número de componentes da arquitetura, fazendo com que os fatores existentes nessa perspectiva sejam impossíveis de serem analisados sem nenhum auxílio computacional.

Como o espaço de busca do processo de seleção de componentes de software é gigantesco e complexo para a capacidade humana, automatizar tal processo guiado por objetivos conhecidos (i.e., métricas de software) também não é uma tarefa trivial, já que é classificado como um problema de complexidade computacional NP-Completo (não são conhecidos algoritmos que solucionem o problema em tempo polinomial) e de combinatória [Petty03]. O esforço humano deve ser concentrado no que é requerido e não como é construído [Harman01]. Cabe, então, recorrer a abordagens como a Engenharia de Software Baseada em Buscas, onde problemas da Engenharia de Software são tratados como problemas de busca para serem otimizados por técnicas de meta-heurísticas ou computação evolutiva como, por exemplo, algoritmos genéticos [Harman06].

1.2. Objetivos do Trabalho

O objetivo geral do presente trabalho é de utilizar técnicas da Engenharia de Software Baseada em Buscas para minimizar os problemas pertencentes à etapa de seleção de componentes de software. Para tal, a abordagem proposta adota métricas estruturais (conexões entre os componentes) e métricas intrínseca ao componente como similaridade (grau de adequação funcional) e informações de reuso (grau de satisfação dos consumidores que

utilizaram outrora certo componente e também o número de *downloads* do mesmo) para avaliar não apenas a estrutura da arquitetura, como também os componentes candidatos.

Para alcançar o objetivo geral do trabalho, foram definidos os seguintes objetivos específicos:

- Levantamento bibliográfico dos principais trabalhos relacionados à seleção de componentes, mais precisamente nos que utilizam abordagens automatizadas para sua computação;
- Elaboração/definição das métricas mais adequadas para serem aplicadas neste processo de seleção de componentes;
- Definição da meta-heurística que é utilizada na implementação e validação do trabalho;
- Implementação do processo de automatização da seleção de componentes com a meta-heurística escolhida;
- Teste e validação da proposta a partir de experimentos provenientes de estudos de casos.

1.3. Organização do Trabalho

O trabalho está organizado como descrito a seguir. No capítulo 2 são introduzidos alguns conceitos fundamentais relacionados à abordagem proposta, com o objetivo de facilitar a sua compreensão. O capítulo 3 apresenta a abordagem proposta, descrevendo o fluxo de execução do trabalho, assim como as métricas quantitativas definidas para avaliar configurações arquiteturais formadas por componentes candidatos, e também o algoritmo baseado em meta-heurísticas para buscar soluções satisfatórias. Já no capítulo 4 é realizada a avaliação experimental com testes estatísticos visando validar o trabalho proposto. O capítulo 5 apresenta os trabalhos correlatos, destacando as semelhanças e diferenças em relação ao presente trabalho, e, assim, evidenciando suas principais contribuições. Por fim, no capítulo 6 são apresentadas as considerações finais, evidenciando as contribuições, limitações e trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Este capítulo apresenta as áreas de estudo mais relevantes para o presente trabalho e cujo conhecimento é necessário para seu entendimento como um todo: *Desenvolvimento Baseado em Componentes*, com foco no aspecto de reúso de software, na definição do componente e suas interfaces, e o ciclo de vida do processo de construção de sistemas baseado em componentes (mostrando as etapas anteriores e posteriores do processo de seleção de componentes); *Arquitetura de Software*, com ênfase no uso de métricas para avaliar uma configuração arquitetural; *Engenharia de Software Baseada em Buscas*, com foco no uso de meta-heurísticas como estratégia para um problema com espaço de busca complexo e gigantesco; e *Algoritmos Genéticos* com foco em apresentar sua definição e seu ciclo básico para facilitar o entendimento do leitor.

2.1. Desenvolvimento Baseado em Componentes

Várias abordagens de desenvolvimento têm sido propostas nas pesquisas alusivas ao reúso de software, onde artefatos de software já existentes são utilizados para construir novos artefatos de software. Essas abordagens objetivam reduzir a complexidade, o tempo e o custo de desenvolvimento de software, assim como melhorar a qualidade, produtividade, confiabilidade e manutenibilidade dos produtos.

Enfrentar problemas sem algum suporte de reúso, seja intelectual ou material, demanda bastante esforço para alcançar soluções satisfatórias, tornando tal fator inapropriado em um mercado competitivo. É intrínseco do ser humano reusar padrões de boas soluções em problemas idênticos ou similares, cabendo adaptar possíveis inconsistências provenientes do novo contexto. É inegável que o processo de adaptação requer esforço, todavia é muito menor comparado à elaboração de uma solução [Becker06]. Então, o Desenvolvimento Baseado em Componentes (DBC) surge como uma possível alternativa para o reúso de software, onde as unidades de composição são denominadas componentes.

De acordo com Sametinger, componentes de software reusáveis são auto-contidos, artefatos facilmente identificáveis que descrevem e/ou executam funções específicas e têm interfaces claras, documentação apropriada e um estado de reuso definido [Sametinger97]. Cada uma dessas propriedades é detalhada abaixo:

- **Auto-contido:** um componente é classificado como auto-contido quando ele é reusado sem a necessidade de incluir outros componentes para o reuso. Ou seja, o componente é uma unidade independente de algum outro determinado componente para servir sua funcionalidade.
- **Identificação:** componentes devem ter uma localização definida, não sendo misturados com outros artefatos. O termo artefato indica que componentes podem ter uma variedade de diferentes formatos, seja código fonte, código executável, etc.
- **Funcionalidade:** componentes têm uma funcionalidade claramente especificada que eles executam ou descrevem.
- **Interfaces:** componentes devem possuir interfaces claras e ocultar detalhes que não são necessários para o reuso.
- **Documentação:** é indispensável para o reuso. O componente mais útil para certo domínio de aplicação perde sua utilidade para os propósitos de reuso se não apresentar uma documentação disponível.
- **Estado de Reuso:** contém informações sobre quem desenvolveu o componente, quem realiza manutenção, quem pode ser contactado caso surja algum problema e qual o estado de qualidade do componente.

Em um sistema de software geralmente não há possibilidade de que apenas um componente atenda todos os seus requisitos, então cada componente (possuindo suas respectivas funcionalidades) se integra com outros componentes para atender as necessidades do sistema em desenvolvimento. Os componentes se conectam por interfaces, onde uma interface representa um subconjunto das interações que um componente pode sustentar com outros elementos de software. As interfaces definem também um subconjunto de operações que são disponíveis por um componente [Sametinger97].

Um componente de software dá suporte a interfaces providas se o mesmo contém uma implementação de todas as operações definidas pela interface. Um componente necessita de uma interface requerida se o mesmo solicita uma interação definida em uma interface e o componente espera algum outro elemento de software para dar suporte àquela interface. Caso uma interface requerida não seja satisfeita para algum componente, o mesmo pode não prover

uma interface porque dependeria de algum elemento de software para poder realizar sua computação e então, satisfazer sua operação que o componente potencialmente provê. Idealmente, um componente deveria sempre ter suas interfaces providas solicitadas e suas interfaces requeridas atendidas por outros componentes [Heineman01].

As interfaces são atribuídas para um componente por intermédio da especificação de componente, as quais definem uma implementação e os limites da mesma no momento de implantação do componente em domínios de aplicação. A especificação determina como operações devem ser implementadas em termos do uso de outras interfaces. A implementação de componente é um modelo concreto de computação, pois realmente foi construída e está disponível para ser executada em algum sistema. Já a especificação de componente é um modelo abstrato, apenas define a especificação de um software e não seus detalhes de implementação.

Há diversas variações de processos DBC adotados no mercado de desenvolvimento de sistemas de software, entretanto será apresentado um modelo de ciclo de vida de um processo de DBC adaptado com o intuito de melhorar o entendimento, baseado em [Vitharana03a] e estabelecido da seguinte forma (apresentadas em ordem):

- **Análise de domínio:** planejamento dos requisitos do sistema diante o domínio de aplicação.
- **Projeto arquitetural:** elabora o modelo da arquitetura do sistema, que é constituída por especificações de componentes e suas interfaces.
- **Recuperação de componentes:** baseada nas especificações da arquitetura, na qual recuperam de repositórios os componentes candidatos para constituir o sistema em desenvolvimento.
- **Seleção de componentes:** escolhe os componentes mais adequados para pertencerem a um sistema que será construído.
- **Montagem do sistema:** recebe os componentes selecionados e os conectam para obter um sistema executável.

O trabalho aqui proposto concentra-se na etapa de seleção de componentes, todavia recebe artefatos do desenvolvimento como: modelo da arquitetura do sistema em desenvolvimento (produzido pelo projeto arquitetural), e informações descritivas dos componentes candidatos baseadas nas especificações da arquitetura (resultado da recuperação

dos componentes candidatos) provenientes de outras etapas, assim como fornece um conjunto de componentes selecionados para serem conectados na etapa de montagem do sistema.

2.2. Arquitetura de Software

A arquitetura de software engloba um conjunto de decisões significativas a respeito da organização de um sistema de software que inclui a seleção de elementos estruturais e suas interfaces do qual tal sistema é composto [Garlan94]. Ela é concentrada nas relações formadas por suas interfaces e não nos seus elementos estruturais, pois detalhes de implementações devem ser internos a ele.

Na perspectiva de componentes de software, uma arquitetura baseada em componentes pode ser definida como um conjunto de componentes, juntamente com suas relações estruturais e suas dependências comportamentais. Esta definição independe de tecnologia onde o sistema será implantado. Obter uma arquitetura de componentes proporciona analisar e entender o comportamento estrutural do sistema, por exemplo, observar o nível de acoplamento entre os componentes [Cheesman01]. Minimizar o acoplamento entre componentes é uma boa prática de design, pois resulta na diminuição da complexidade do sistema. Um sistema fortemente acoplado dificulta atributos de qualidade como manutenibilidade e confiabilidade.

A avaliação arquitetural tem como objetivo prever a qualidade de uma arquitetura de software antes que um sistema seja construído, e também possibilita estimar os principais efeitos da arquitetura [Kazman93]. O propósito dessa avaliação é identificar riscos potenciais e verificar se os requisitos de qualidades foram incorporados no projeto da arquitetura [Li93].

Há várias abordagens para a avaliação de uma arquitetura de software, conforme abordado em [Oliveira Junior10], podem ser divididas em duas categorias, consoante os objetivos subjacentes:

- **Avaliação de atributos de qualidade:** estudos que realizam a avaliação de atributos como performance, reusabilidade, confiabilidade, etc;
- **Avaliação estrutural:** estudos que realizam a avaliação arquitetural conforme sua estrutura em termos de métricas que analisam interfaces, assinaturas de métodos, utilização de serviços, entre outras.

A avaliação dos atributos de qualidade e estrutural são adequadas para serem utilizadas no processo de seleção de componentes, visto que deve-se avaliar configurações

arquiteturais candidatas. Os componentes pertencentes à arquitetura contribuem para a avaliação de atributos de qualidades da arquitetura, pois a qualidade das parcelas (componentes) reflete no todo (arquitetura). As interfaces dos componentes (estabelecem conexões) contribuem para a avaliação estrutural.

A avaliação dos atributos de qualidade geralmente é executada das seguintes formas [Clements02][Vitharana03b]:

- Cada componente é avaliado individualmente e a soma do escore qualitativo de todos os componentes pertencentes à arquitetura é atribuída a ela.
- São realizadas simulações entre os componentes, como tempo de resposta entre componentes.

Na perspectiva estrutural, a avaliação de incompatibilidades entre conexões de componentes possui o benefício de aumentar a probabilidade de que tais inconsistências de integração entre componentes sejam identificadas antes que o sistema seja construído, potencialmente prevenindo grande perda de esforço e tempo [Egyed00].

2.3. Engenharia de Software Baseada em Busca

Este campo da Engenharia de Software emprega técnicas e métodos que reformulam e modelam problemas dessa disciplina como aqueles que podem ser resolvidos usando técnicas de otimização baseadas em busca meta-heurística [Harman01]. Este tipo de busca é utilizado em problemas que não se tem conhecimento de algoritmos eficientes para sua resolução. Atividades realizadas no desenvolvimento de projetos de Engenharia de Software, como *balanceamento de restrições concorrentes* [Freitas09], são atreladas a problemas complexos os quais apresentam uma gama enorme de possibilidades, e são definidos como NP-Completo. Além de automatizar a atividade, a busca meta-heurística propõe-se a obter resultados aceitáveis (soluções próximas da ótima) ou até mesmo pode encontrar a solução ótima.

Práticas referentes ao conceito de Engenharia de Software Baseada em Busca surgiram a partir da década de 1970, e eram tratadas com pouca atenção pela comunidade acadêmica. Como a tendência é que os sistemas de software se tornem mais complexos e desafiadores com o passar do tempo, houve a necessidade de explorar novas vertentes. Esse cenário contribuiu para que, recentemente, esse campo de pesquisa emergisse na esfera acadêmica, onde o interesse e prática desse conceito crescem vertiginosamente. Essa abordagem

emergente demonstra potencial para se consolidar na Engenharia de Software, uma vez que as técnicas de otimização são aplicadas com sucesso em outras disciplinas de engenharia (mecânica, elétrica, civil, etc).

Conforme Harman [Harman10], algoritmos de otimização são adequados para um conjunto ideal de aplicações oriundos da Engenharia de Software, pois a essência virtual dessa disciplina permite a concordância com as buscas meta-heurísticas, seja por mapeamento ou reformulações. As características desses algoritmos refletem na Engenharia de Software Baseada em Busca, tornando-a: **escalável** (o desempenho do algoritmo não demonstra decaimento significativo, pelo motivo da alta capacidade de paralelização dessas abordagens algorítmicas), **genérica** (algoritmos compatíveis com diversos problemas, visto que eles possuem metodologias que facilitam a representação do domínio do problema nesses algoritmos), **robusta** (habilidade de lidar com fatores adversos como ruídos e dados incompletos), **rica em introspecção** (pode gerar resultados além da imaginação dos engenheiros, possibilitando incluí-los em estratégias anteriormente debatidas) e **realística** (lida com múltiplos objetivos potencialmente conflitantes como no dia-a-dia, em que reduzir o tempo de um projeto impacta na qualidade, no escopo, etc.) [Harman10].

Problemas de busca complexos e passíveis de serem reformulados matematicamente estão distribuídos em diversas áreas da Engenharia de Software. Atividades como alocação de recursos, presente no campo de engenharia de requisitos, ou seleção de testes, inserido no campo de testes de software, são exemplos desses problemas. Apesar de espalhada em vários estágios referentes ao ciclo de vida do desenvolvimento de software, a distribuição a aplicação de meta-heurísticas não é uniforme; a concentração das aplicações está inserida em poucas áreas. O campo de teste de software chega a possuir mais da metade das aplicações, consoante ao levantamento realizador por [Harman09]. A Tabela 1 evidencia exemplos de problemas de busca nas principais áreas da Engenharia de Software que se aplicam técnicas de otimização [Freitas09].

Tabela 1 – Exemplos de problemas de busca nas principais áreas da engenharia de software.

Área	Exemplo de Problemas
Planejamento de projeto	Estimativa de custo
Engenharia de requisitos	Priorização de requisitos
Design de software	Melhoria de qualidade do design arquitetural

Implementação/codificação	Otimização de código-fonte
Integração	Integração de componentes de software
Teste/validação	Priorização de casos de teste
Deployment	Implantação de componentes de software
Manutenção	Refatoração automática

A reformulação do problema é altamente dependente do domínio do problema, pois cada mapeamento é exclusivo de certo problema da Engenharia de Software [Harman01]. Uma meta-heurística possui inicialmente caráter genérico, e então é lapidada para o problema em questão. De fato há esforço intelectual para a modelagem do algoritmo, porém o esforço humano empregado em tentar buscar a solução é maior-igual do que a modelagem algorítmica da meta-heurística. A solução encontrada por esforço humano tem o risco de aparentar ser aceitável, entretanto, após vários ciclos de iteração de projeto, pode-se perceber que a solução encontrada geraria resultados desastrosos; consequentemente resultaria em investimento perdido, fator inadmissível na perspectiva do mercado atual. Portanto, o esforço humano empregado deve ser focado na descrição da solução (guia para busca automatizada), e não na sua construção [Harman10].

2.3.1. Ingredientes Essenciais

Para que a reformulação do problema tenha sucesso, há alguns ingredientes essenciais que devem ser seguidos. Entre eles são: representação do problema, função de adequação, operadores de manipulação [Harman01]. De certa forma, o engenheiro de software geralmente terá facilidade em representar seu problema de busca em alguma meta-heurística, visto que rotineiramente o mesmo enfrenta situações na qual deve mapear ou simular modelos concretos em dados. Outra facilidade pode ser observada no fato de que muitos problemas da Engenharia do Software possuem uma gama de métricas de software relacionadas consigo e que elas são potenciais candidatas para serem funções de adequação, ou mesmo se tornarem modelos para a construção destas funções [Harman07][Harman10]. Segue abaixo uma síntese explicativa de cada ingrediente essencial para reformulação do problema.

A *representação do problema* consiste em transformar conceitos do problema em símbolos para que seja possível realizar manipulações simbólicas, como na matemática [Harman01]. Assim, esta transformação gera uma solução candidata que poderia ser avaliada

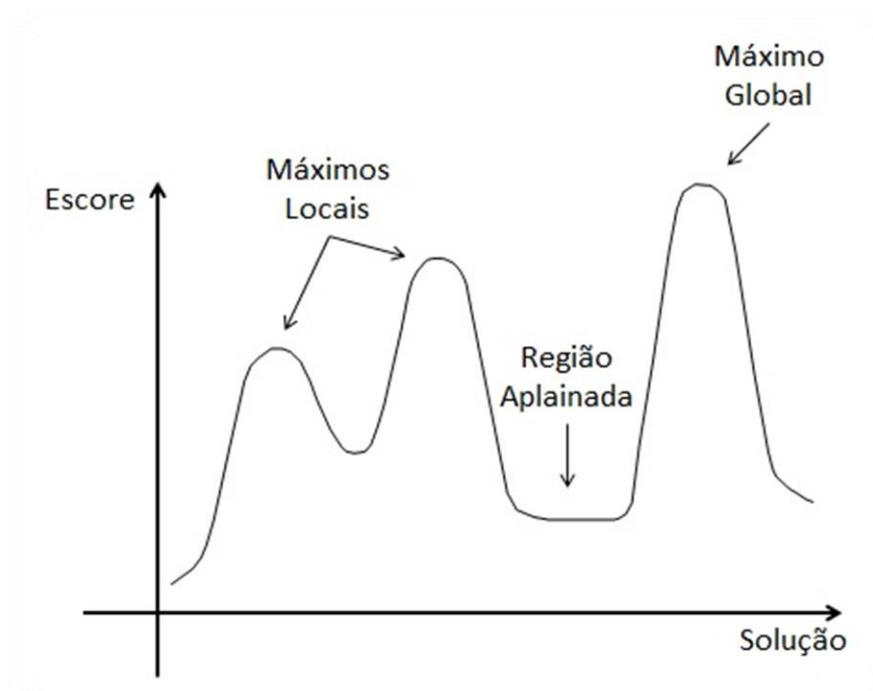
por funções que as classificassem como boas ou más soluções. Para serem computadas, as soluções candidatas são representadas como: vetores, matrizes, grafos, código binário, etc. O engenheiro de software deve escolher uma representação que tente adequar ao máximo possível do problema em questão [Harman07]. Uma má representação corromperia ou acarretaria na perda de informação que, por conseguinte, impactaria nos resultados provenientes da busca.

A *função de adequação* é uma função criada em termos da representação do problema que precisa ser resolvido, onde soluções boas teriam níveis satisfatórios de adequação, diferentemente para soluções ruins [Harman01]. Elaborar tal função não é uma tarefa trivial, é possível que seja preciso algumas tentativas para definir termos de propriedades que devem ser mensuradas nas soluções candidatas, antes de encontrar uma função ideal para o problema [Linden08].

Partindo da comparação entre soluções candidatas, a função de adequação deve impor uma escala de medida, onde uma solução considerada melhor que outra receberá um escore maior que a última. Então, a solução ótima teria que obter o escore máximo, consequentemente nenhuma outra solução poderia ultrapassá-la [Harman01]. Como o escore gerado pela função de adequação tem teor numérico, cada solução é mapeada para um valor e essa função transforma o espaço dessas soluções em uma paisagem de adequação (gráfico), vide Figura 1. Nela, o eixo das soluções é discreto e o eixo dos escores é contínuo; a curva é gerada por interpolação.

Cada ponto no espaço tem um valor numérico que corresponde ao grau de adequação de uma solução candidata. O máximo global representa a solução ótima, já um máximo local é caracterizado por uma solução com o maior escore comparado as soluções vizinhas. Para a função de adequação gerar um espaço favorável à busca, é mister atender as seguintes propriedades:

- Não deve possuir uma paisagem aplainada, pois as soluções candidatas são muito similares, e então a busca poderá perambular sem direção definida, causando dificuldade de encontrar soluções consideravelmente melhores.
- Evitar possuir um espaço com máximos agudos; caso a busca encontre uma solução que representa um máximo local agudo, ela ficará “presa” porque as soluções vizinhas são muito ruins em relação a esse máximo local e a busca para por não encontrar soluções melhores.

Figura 1 – Espaço dessas soluções em uma paisagem de adequação.

É importante salientar que o engenheiro de software não tem posse da paisagem de adequação, pois ela representa todo o espaço de busca avaliado, e que apenas a busca por força-bruta poderia gerá-lo. Então evitar máximos agudos e uma paisagem aplainada irá depender de análises, as quais são realizadas em um subconjunto de soluções avaliadas pela função de adequação. Caso as análises apontem imperfeições, é requerido o ajuste da função de adequação. É evidente que mesmo com tais análises e ajustes, os problemas citados ainda podem persistir diante de um espaço de busca gigantesco e complexo, por isso é importante escolher uma meta-heurística que possua estratégias para escapar dessas adversidades.

A computação da função de adequação deve ser eficiente porque as meta-heurísticas necessitam realizar várias iterações do seu algoritmo, e uma função ineficiente se tornaria um gargalo no processo algorítmico da otimização. O processo para obter informações conceituais do problema – inseridas nas soluções candidatas – deve ser o mais direto possível, justamente para não causar *overhead* de processamento ou armazenamento.

Os *operadores de manipulação* são capazes de realizarem computação sob as representações de soluções candidatas existentes para gerar possivelmente novas soluções, na expectativa de explorar de forma inteligente o espaço de busca [Harman01]. Uma má escolha de operadores pode restringir a diversidade das soluções encontradas, diminuindo a efetividade do algoritmo de otimização e aumentando a possibilidade de parada em ótimos

locais. Diferentes técnicas de busca adotam operadores distintos, os quais ainda podem sofrer modificações visando melhorar a adequação ao problema que se pretende resolver.

2.3.2. Meta-heurísticas Comumente Utilizadas

Meta-heurísticas são heurísticas genéricas que são mapeadas para problemas de algum domínio específico para buscar soluções aproximadas da ótima. Abaixo seguem as principais meta-heurísticas abordadas em pesquisas no campo da Engenharia de Software Baseada em Busca [Clarke03].

- **Algoritmos Genéticos:** método de busca evolutivo baseado no processo de seleção natural;
- **Arrefecimento Simulado:** método de busca local probabilística que melhora iterativamente a solução inicial do espaço de busca;
- **Enxame de Partículas:** técnica estocástica baseada em populações que implementa uma metáfora do comportamento social da interação entre indivíduos (i.e., partículas) de um grupo (i.e., enxame);
- **Busca Tabu:** estratégia baseada em memória adaptativa que guia um método de busca local que melhora iterativamente a solução inicial do espaço de busca;
- **Colônia de Formigas:** método de busca coletiva, paralela e dinâmica que permite resolver problemas que podem ser reduzidos a encontrar bons caminhos em grafos.

Nenhuma busca utilizando a abordagem de meta-heurística garante encontrar a solução ótima, pois seu processo de execução é não-determinístico, ou seja, diferentes soluções finais podem ser apresentadas, as quais são consideradas razoavelmente boas, dentre elas pode estar a solução ótima. O não-determinismo pode ser justificado por: fatores aleatórios inerentes às meta-heurísticas, solução inicial, parâmetros, critério de parada. Apesar de ser impossível prever a solução final, é necessário que os resultados provenientes da meta-heurística sejam melhores que uma busca puramente aleatória [Harman01].

Dentre as meta-heurísticas citadas acima, não existe alguma considerada superior à outra de forma universal, ou seja, para qualquer aspecto que se queira comparar nem sempre uma será melhor que a outra. Irá depender do domínio do problema, cada um pode ter afinidade melhor com certa meta-heurística. Segundo [Wolpert97], todas as meta-heurísticas possuem o mesmo desempenho quando é realizada a média das avaliações relacionadas a infinitos problemas distintos. Dessa forma, se para certo número de problemas a meta-

heurística *A* proporcionou melhor desempenho, também existe outro conjunto de problemas onde a meta-heurística *B* apresenta melhor eficácia [Linden08].

Outra propriedade do trabalho de [Wolpert97] afirma que um algoritmo projetado especialmente para algum problema possui melhor desempenho do que outro de cunho genérico. O problema de busca deve ser reformulado para a meta-heurística que melhor absorva as propriedades e restrições do problema em questão. Quanto mais detalhes do problema a meta-heurística apresentar, melhor será seu desempenho. É importante considerar que a função de adequação deve ser computacionalmente eficiente, ou seja, a meta-heurística deve ser capaz de mapear o máximo de detalhes do domínio do problema, e ainda assim manter sua eficiência computacional. Além disso, a reformulação do problema deve ser fácil de entender para gerar maior confiança na percepção das soluções.

2.3.3. Problemas de Otimização Multi-objetivos

A otimização multi-objetiva envolve a minimização ou maximização de múltiplas funções de adequação. Este tipo de problema está relacionado com situações que ocorrem *trade-off* (dois ou mais objetivos conflitantes), i.e., fatores como tempo, custo e qualidade competem entre si. É comum encontrar este tipo de problema na vida real.

Uma abordagem comum para otimização multi-objetiva consiste em combinar funções de adequação individuais em uma única função composta. A determinação desta função é possível com métodos como soma ponderada, teoria da utilidade [Neumann44], etc. Mas o problema é direcionado para a seleção dos pesos que serão atribuídos a cada objetivo conflitante. Algum especialista deve atribuir os pesos precisamente baseado no domínio do problema [Konak06].

Este método é utilizado de maneira direta e apresenta resultados que fazem sentido intuitivamente. Entretanto, a definição dos pesos para um problema com muitos objetivos torna-se difícil [Kontio96]. Normalmente, a capacidade humana demonstra que ponderar até dez alternativas ao mesmo tempo têm-se resultados satisfatórios [Saaty82]. Quanto mais os objetivos são conflitantes entre si, maior será a complexidade para atribuir os pesos.

A Equação 1 representa o conjunto de funções de adequações referentes aos objetivos (critérios) que fazem parte da otimização multi-objetiva. Logo em seguida, a Equação 2 evidencia o conjunto dos pesos que remetem aos graus de importância de cada objetivo. E por fim, a Equação 3 demonstra a combinação dos objetivos por soma ponderada pelos graus de importância, e assim gerando o valor de adequação geral. Os possíveis valores dos pesos

pertencem ao conjunto dos números reais e do intervalo $[0,1]$, e a soma de todos os valores é igual a 1.

$$F(x) = [f_1(x), f_2(x), \dots, f_n(x)] \quad (1)$$

$$W = [w_1, w_2, \dots, w_n] \quad (2)$$

$$F(x) = \sum_{i=1}^n w_i \times f_i(x) \quad (3)$$

Para facilitar o processo de definição dos pesos de relevância alusivo aos critérios de uma função multi-objetivo, é necessário que tais critérios tenham suas avaliações quantitativas na mesma escala numérica, caso contrário, se o especialista encarregado de atribuir a ponderação escolhesse os mesmos pesos para cada critério com pontuação quantitativa média, haveria variação de importância entre si. Contudo não deveria ocorrer tal variação, pois o especialista idealizou o mesmo grau de importância de cada critério.

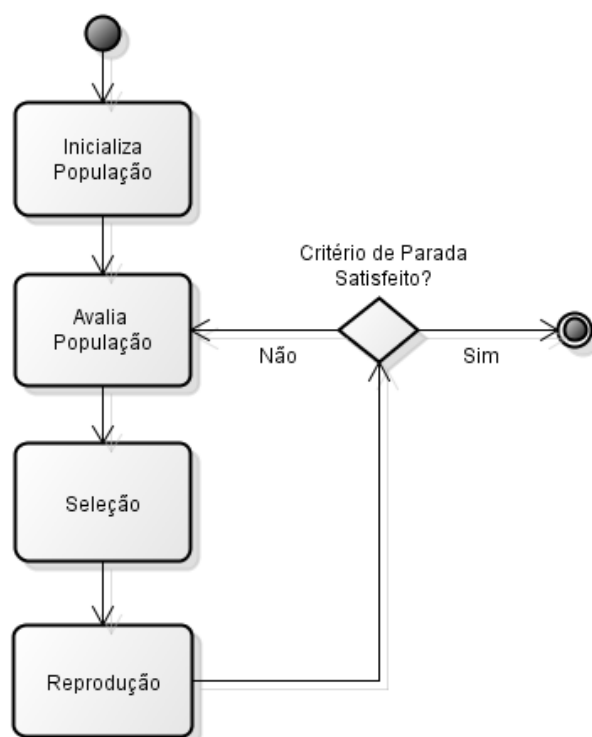
Quando se trata de otimização multi-objetiva, existem funções de adequação que quanto maiores são seus valores numéricos, melhor é a solução avaliada, e para outras ocorre justamente o inverso. Isso se dá porque há critérios cuja natureza qualitativa melhora na medida em que sua avaliação quantitativa é maximizada ou minimizada. Por exemplo, o custo e o tempo para desenvolvimento de software são objetivos do tipo minimização, por outro lado, a performance e a robustez de um software são do tipo maximização. Como a função de otimização multi-objetiva geralmente combina várias funções de adequação, é fácil ocorrer de que ambos objetivos, de minimização e maximização, façam parte do problema de otimização. Portanto, para não ocorrerem inconsistências como a soma de objetivos qualitativamente inversos, todos os objetivos de maximização são multiplicados por -1 , e assim, convertidos objetivos de minimização. De forma análoga, também se pode fazer a conversão dos objetivos de minimização em maximização. Fica a critério do engenheiro de software, que irá modelar a função multi-objetiva, qual das duas vertentes ele irá escolher.

2.4. Algoritmos Genéticos

Algoritmos genéticos (AG) [Holland75] são uma meta-heurística de busca adequada para a utilização em problemas de otimização multi-objetivos [Zhang07]. Algoritmos genéticos são parte da classe de algoritmos evolucionários e buscam a otimização de problemas a partir da simulação de técnicas e conceitos presentes na evolução natural (e.g., *crossover*, mutação, seleção, etc.).

Nos algoritmos genéticos, uma população de indivíduos (cromossomos) é inicialmente gerada, onde cada cromossomo representa uma possível solução para o problema a ser otimizado. Cada um dos cromossomos possui um valor de adequação, calculado a partir de uma função de adequação (*fitness*), que representa a qualidade da solução representada pelo mesmo. A partir da população corrente, uma lista de cromossomos pais é selecionada e, com operações de crossover (i.e., combinação genética entre dois cromossomos) e mutação (i.e., modificação genética de um cromossomo), uma nova população é gerada. Este processo se repete até que uma condição de parada seja atingida. A combinação da população com os operadores garante uma busca eficiente do espaço de soluções mesmo que ele seja relativamente desconhecido. A Figura 2 ilustra o ciclo de um algoritmo genético, com as etapas descritas anteriormente.

Figura 2 – Ciclo de um algoritmo genético simples.



Em cada etapa do AG tenta-se escolher a melhor técnica ou método para o processamento do algoritmo obter resultados consistentes. Essas escolhas são em termos do problema que se propõe a resolver e até mesmo metodologias consolidadas em vários domínios de aplicação do AG. E o critério de parada geralmente é realizado pelo critério de número de iterações, tempo de processamento ou qualidade dos cromossomos.

2.5. Considerações Finais

Este capítulo ofereceu um conteúdo baseado na revisão da literatura das principais áreas de estudo envolvidas no presente trabalho e seu conhecimento é necessário para a compreensão do trabalho como um todo: *Desenvolvimento Baseado em Componentes* é a base teórica do problema abordado por este trabalho, que é a seleção de componentes de software, então as definições de componentes e seus elementos, além das etapas do processo de DBC para desenvolver sistemas foram exploradas em sua seção; *Arquitetura de Software*, formada por componentes de software, ela representa a especificação estrutural do sistema e neste trabalho o foco é em sua definição e no uso de métricas para avaliar a arquitetura sob diferentes perspectivas; *Engenharia de Software Baseada em Buscas*, com ênfase na descrição da utilização de algoritmos de otimização como estratégia de busca, como também na reformulação do problema nesses algoritmos; *Algoritmos Genéticos*, são algoritmos de otimização (meta-heurística) escolhidos para o presente trabalho, portanto são apresentados sua definição e seu ciclo de vida. Como a literatura é ampla acerca de tais áreas, o capítulo somente explorou os tópicos relevantes para o trabalho, todavia leituras adicionais são amplamente recomendadas para um estudo aprofundado sobre cada uma das áreas de pesquisa.

Capítulo 3

Abordagem Proposta

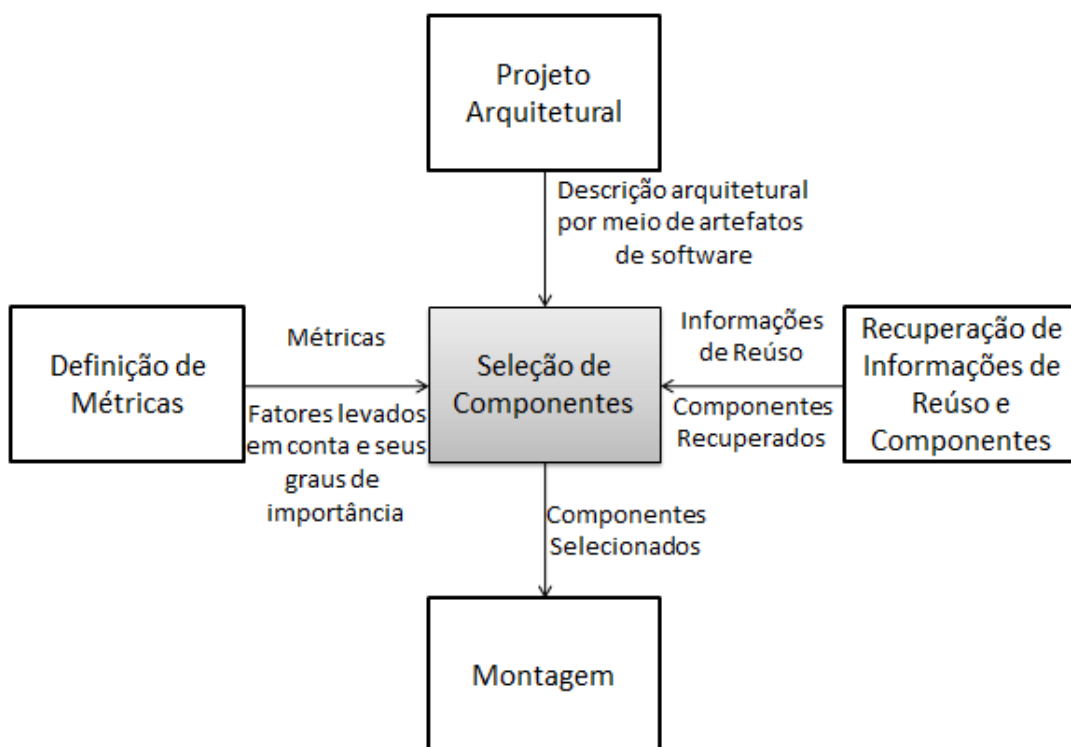
O trabalho proposto visa automatizar o processo de seleção de componentes utilizando técnicas da Engenharia de Software Baseada em Busca, cuja meta-heurística é orientada por métricas estruturais (i.e., conexões entre componentes de uma arquitetura de software) e métricas agregadas ao próprio componente como similaridade (funcionalidade) e informações de reúso (i.e., valores de reúso do componente). Com isto, espera-se que a seleção de componentes seja capaz de escolher uma configuração arquitetural satisfatória baseada no projeto da arquitetura de software (formada por especificações de componentes), que é elaborada pelo arquiteto de software. Este processo visa gerar uma solução (configuração arquitetural) com o mínimo de incompatibilidades estruturais possíveis, ou seja, aumenta a probabilidade que essas inconsistências estruturais entre componentes sejam identificadas antes que o sistema de software seja construído, consequentemente o custo e o esforço para interação e adaptação dessas inconsistências também são minimizados [Egyed00]. Também são levadas em consideração as informações de reúso, as quais implicam em aspectos qualitativos dos componentes integrantes da arquitetura de software, pois componentes de boa qualidade irão refletir suas características qualitativas na arquitetura como um todo, proporcionando que o sistema a ser desenvolvido seja um produto de, também, boa qualidade [Vitharana03b][Roy08]. Essas duas vertentes de avaliação de arquitetura de software – a estrutura arquitetural e o próprio componente em si – são combinadas neste trabalho aqui proposto, tornando a meta-heurística multi-objetiva. Ambas possuem relação qualitativa entre si, até porque seus componentes são os mesmos, mas cada vertente pode ser visualizada por uma ótica torna-se um objetivo diferente.

A abordagem de *seleção automatizada de componentes de software orientada por métricas estruturais e informações de reúso* terá seus aspectos fundamentais detalhados neste capítulo. De início é apresentada uma visão geral do trabalho para que o leitor possa perceber a estrutura do modelo do trabalho proposto, assim como o escopo. Logo em seguida serão apresentadas as etapas da abordagem proposta, evidenciando suas propriedades e seu fluxo.

3.1. Visão Geral da Abordagem Proposta

Considerando um processo de Desenvolvimento Baseado em Componentes (DBC), a etapa que antecede o processo de *Seleção de Componentes* é caracterizada pelo *Projeto Arquitetural* (modelagem do sistema), e ela fornece uma descrição arquitetural para o estágio do trabalho proposto. Já a etapa posterior consiste na *Montagem do Sistema de Software*, que solicita do processo de seleção um conjunto de implementações de componentes para serem conectadas e formarem uma aplicação. A Figura 3 ilustra as etapas que compõem a abordagem proposta. Como pode ser observada, a etapa de Seleção de Componentes recebe níveis de importância (pesos atribuídos pelo arquiteto de software) referentes às *Métricas Definidas* escolhidas para avaliar configurações arquiteturais. Assim, caso o arquiteto queira que uma métrica contribua mais para uma avaliação que outra, simplesmente basta aumentar sua importância. A *Recuperação de Informações de Reúso e Componentes* tem o objetivo de recuperar componentes e informações através do *Serviço de Repositório* e do *Serviço de Busca*. As informações de reúso são obtidas por intermédio de um *Serviço de Repositório*, que deve disponibilizar várias categorias de dados (i.e. grau de satisfação dos consumidores de certo componente) que contribuirão para gerar informações a respeito dos artefatos armazenados no repositório. O *Serviço de Repositório* também deve disponibilizar artefatos, que descrevem os

Figura 3 – Visão geral da abordagem proposta.



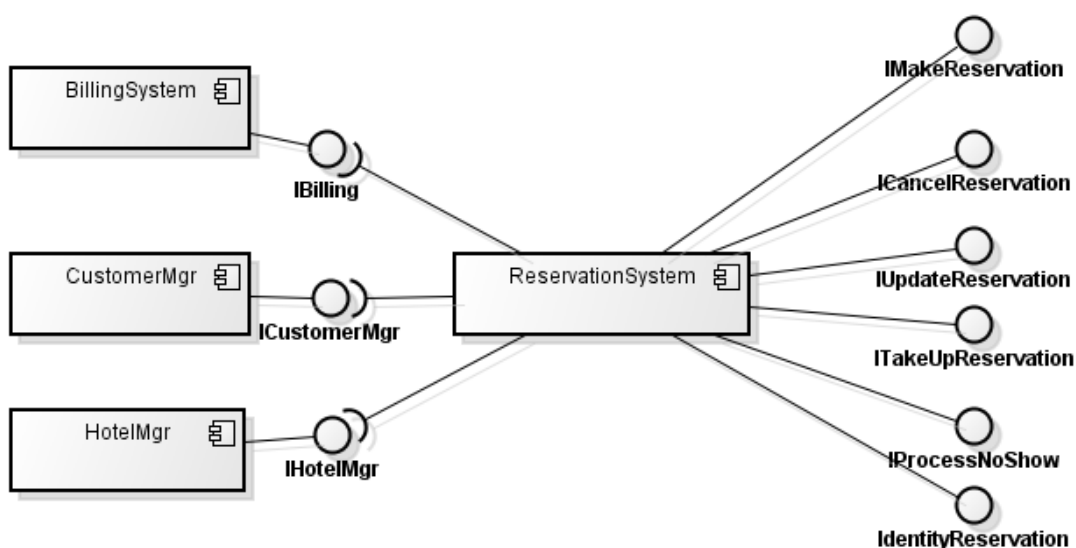
componentes de software candidatos, para a etapa de montagem, onde tais artefatos serão selecionados para então recuperá-los. Nesta proposta, será adotado o serviço de repositório *X-Core* [Oliveria07][Oliveira08], que foi desenvolvido pelos pesquisadores do grupo de pesquisa Compose (www.compose.ufpb.br). As consultas nesse repositório são realizadas por meio do *Serviço de Busca*.

3.2. Projeto Arquitetural

No momento da modelagem do Projeto Arquitetural, o arquiteto de software efetua a modularização do sistema (separação de conceitos) e os conectam por intermédio de interfaces [Eden03]. Esta separação de conceitos melhora a compreensão do desenvolvimento do sistema, bem como ajuda a visualizar aspectos estruturais como coesão e acoplamento [Hoek03]; e ainda é um meio para garantir a abstração, pois o projeto arquitetural suprime alguns detalhes do software, i.e., a arquitetura comumente não possui decisões relacionadas à implementação de código. Na Figura 4, adaptada de [Santos06], é mostrado um exemplo de descrição arquitetural construída com UML¹ [OMG13], sendo os módulos representados por especificações de componente e as conexões por especificação de interfaces providas e requeridas.

É assim que o arquiteto idealiza o sistema, e o mesmo espera que em etapas seguintes cada especificação de componente receba uma instância (implementação de componente) selecionada pelo trabalho proposto.

Figura 4 – Exemplo visual de uma arquitetura de software.



¹ Considerada uma descrição arquitetural pelo SEI – Software Engineering Institute

Segundo [Cheesman01], o diagrama do Projeto Arquitetural baseado em componentes contém apenas especificações de componentes e interfaces, e que todas as dependências pertencentes a uma especificação são caracterizadas por intermédio das interfaces. Essas dependências são partes da definição da especificação de componente e devem ser tratadas por todas as implementações de componente que implementam tal especificação.

A Tabela 2 corresponde ao conjunto de artefatos gerados no processo de Projeto Arquitetural, conforme [Santos06][Cheesman01], e os mesmos serão utilizados na etapa seguinte do DBC. No contexto desses trabalhos, o termo *asset* remete a qualquer artefato de software produzido no processo de DBC. Artefatos que descrevem, representam ou modelam outro *asset*, também é considerado um *asset*. Essa tabela refere ao Projeto Arquitetural que produziu a Figura 4, cada elemento proveniente de tal projeto é classificado por entidades que as representam, e também são descritos em *assets* de metadados que fornecem informações usadas não apenas em repositórios de componentes (pelos consumidores na identificação, aquisição e uso de *assets*), mas também para apoiar as atividades dos desenvolvedores de *assets*. O modelo de descrição utilizado neste trabalho é o X-ARM [Santos06], que tem como objetivo representar os tipos de *assets* cuja descrição se faz necessária em um repositório de componentes. Os elementos possuem identificação única para garantirem a unicidade, independência e transparência de localização dos *assets* armazenados em um repositório.

Tabela 2 – Elementos pertencentes à arquitetura apresentada anteriormente.

Entidade	Identificador do Asset	Tipo de Asset
Interfaces	br.compose.iimakereservation-1.0	Interface Independente
	br.compose.iicancelreservation-1.0	
	br.compose.iiupdatereservation-1.0	
	br.compose.iitakeupreservation-1.0	
	br.compose.iiprocessnoshow-1.0	
	br.compose.iiidentityreservation-1.0	
	br.compose.iibilling-1.0	
	br.compose.iicostumermgr-1.0	
	br.compose.iihotelmgr-1.0	
Especificação de Componentes	br.compose.ireservationsystem-1.0	Especificação de Componente Independente
	br.compose.ibilling-1.0	
	br.compose.icostumermgr-1.0	
	br.compose.ihotelmgr-1.0	
Arquitetura	br.compose.arquitetura-1.0	Resource

A arquitetura (vista como um artefato) é descrita pelo *asset* do tipo *Resource*. Este tipo de *asset* descreve as propriedades de quaisquer artefatos gerados por um processo de desenvolvimento; exceto: componentes, interfaces, eventos e exceções. Esses tipos de *assets* diferentes do *Resource* além de descreverem, também representam um modelo de especificação com conteúdo estruturado para facilitar a obtenção de suas informações por motores de busca. A descrição da especificação de componente *ReservationSystem* (observada abaixo na Figura 5) é realizada pelo artefato do tipo *Especificação de Componente Independente*, que representa a especificação de um componente de forma independente de modelo de componente (forma abstrata). Finalmente, a especificação das interfaces (Figura 6) é descrita pelo *asset* do tipo *Interface Independente*, que a representa de forma independente de modelos uma interface que será provida ou requerida por alguma especificação.

Figura 5 – Descrição da especificação de componente *ReservationSystem*.

```
<asset id="br.compose.ireservationsystem-1.0">
  <component-specification>
    <interface>
      <required>
        <related-asset name="ICustomerMgt"
          id="br.compose.iicustomermt-1.0"
          relationship-type="dependentInterface" required="true"/>
      </required>
      <required>
        <related-asset name="IHotelMgt"
          id="br.compose.iihotelmt-1.0"
          relationship-type="dependentInterface" required="true"/>
      </required>
      <required>
        <related-asset name="IBilling"
          id="br.compose.iibilling-1.0"
          relationship-type="dependentInterface" required="true"/>
      </required>
      <provided>
        <related-asset name="IMakeReservation"
          id="br.compose.iimakereservation-1.0"
          relationship-type="dependentInterface" required="true"/>
      </provided>
      <provided>
        <related-asset name="ICancelReservation"
          id="br.compose.iiancelreservation-1.0"
          relationship-type="dependentInterface" required="true"/>
      </provided>
      <provided>
        <related-asset name="IUpdateReservation"
          id="br.compose.iiupdatereservation-1.0"
          relationship-type="dependentInterface" required="true"/>
      </provided>
      <provided>
        <related-asset name="ITakeUpReservation"
          id="br.compose.iitakeupreservation-1.0"
          relationship-type="dependentInterface" required="true"/>
      </provided>
      <provided>
        <related-asset name="IProcessNoShow"
          id="br.compose.iiprocessnoshow-1.0"
          relationship-type="dependentInterface" required="true"/>
      </provided>
      <provided>
        <related-asset name="IIdentifyReservation"
          id="br.compose.iiidentifyreservation-1.0"
          relationship-type="dependentInterface" required="true"/>
      </provided>
    </interface>
  </component-specification>
  ...
</asset>
```

Figura 6 – Descrição da especificação de interface *IMakeReservation*.

```
<asset id="br.compose.iimakereservation-1.0" ...>
  <interface>
    <operation name="getHotelDetails">
      <parameter name="match" type="String" direction="in"/>
      <return type="HotelDetails[]"/>
    </operation>
    <operation name="getRoomInfo">
      <parameter name="res" type="ReservationDetails" direction="in"/>
      <parameter name="availability" type="Boolean" direction="out"/>
      <parameter name="price" type="Currency" direction="out"/>
      <return type="String"/>
    </operation>
    <operation name="makeReservation">
      <parameter name="res" type="ReservationDetails" direction="in"/>
      <parameter name="cus" type="CustomerDetails" direction="in"/>
      <parameter name="resRef" type="String" direction="out"/>
      <return type="Integer"/>
    </operation>
  </interface>
  ...
</asset>
```

A Figura 6 representa a descrição da especificação da interface *IMakeReservation* (presente na arquitetura mostrada na Figura 4), onde inclui uma lista de operações com assinaturas bem definidas que a interface irá prover/requerer. Já a Figura 5 corresponde a descrição da especificação do componente *ReservationSystem* (também presente na Figura 4), nela é observada uma lista de interfaces providas e requeridas. Essa lista referencia interfaces, especificadas outrora, por meio de seus identificadores (nota-se o reúso de *assets*). Observa-se que a especificação de componente *ReservationSystem* referencia a interface *IMakeReservation* e a classifica como provida, ou seja, *ReservationSystem* provê a interface *IMakeReservation*. Conforme definido no modelo X-ARM, as operações pertencentes às interfaces são descritas dentro dessas interfaces, ao invés de serem referenciadas.

Vimos nesta seção os tipos de artefatos produzidos pelo Projeto Arquitetural, os quais serão tomados como referência para buscar componentes candidatos ao sistema em construção. O Serviço do Repositório também possui artefatos descritos em X-ARM, então os artefatos dos componentes candidatos serão retornados para posteriormente serem comparados com os artefatos elaborados no Projeto Arquitetural por meio de métricas de software que irão direcionar quais componentes candidatos devem ser escolhidos.

3.3. Recuperação de Informações de Reúso e Componentes

O presente trabalho investiga o Serviço de Repositório [Oliveira08] e o Serviço de Busca [Viana11] do *framework ComponentForge* [Oliveira06], o qual utiliza uma abordagem arquitetural orientada a serviços (SOA – *Service-Oriented Architecture*). Este *framework* possibilita que quaisquer artefatos produzidos no processo de desenvolvimento de software sejam armazenados, indexados, buscados, recuperados, certificados e negociados via serviços que são distribuídos, compartilhados, independentes e fracamente acoplados. Além disso, tais serviços se comunicam e colaboram entre si por meio de interfaces e protocolos de comunicação bem definidos. Dentre quatro módulos de serviço do *framework*, destacam-se o Serviço de Repositório (responsável por armazenar, recuperar e gerenciar artefatos) e o Serviço de Busca (provê a descoberta dos artefatos armazenados no Serviço de Repositório).

Uma das motivações para adoção do *ComponentForge* se dá pelo fato de que o mesmo adota o X-ARM como modelo de representação (baseado em metadados XML) de artefatos de software, já que consoante [Frakes05][Orso00], sistemas de repositórios que dão suporte a componentes de software devem armazenar também os metadados que descrevem tais componentes. Outra característica importante é que esse modelo de representação/descrição não apresenta dificuldades quanto à interoperabilidade e portabilidade, já que é baseado em XML [Schuenck07]. Como o X-ARM pode possuir caráter representativo de um componente, em um processo de seleção de componentes não é necessário ter o componente concreto para extrair suas informações particulares, ou seja, apenas com artefatos descritivos contidos no repositório é possível obter informações essenciais sobre o componente.

O Serviço de Repositório proporciona – além das propriedades supracitadas – aspectos de segurança, gerência de métricas de reúso, controle de versão e ainda provendo toda a infraestrutura para os demais serviços do *ComponentForge*, dessa forma, atuando como um middleware de propósito especial [Oliveira08]. No contexto do trabalho proposto, este Serviço de Repositório armazena uma quantidade gigantesca de implementações de componentes COTS (desenvolvidos por terceiros e prontos para uso) juntamente com seus artefatos descritivos em X-ARM, e ainda provê o suporte a um conjunto de informações de reúso (que o repositório gerencia). Na etapa de seleção de componentes do processo de DBC, as informações de reúso, provenientes desse repositório, contribuirão na escolha da configuração arquitetural do sistema em construção. Todas as descrições de artefatos candidatas serão recuperadas e posteriormente avaliadas para definir quais implementações de componentes realmente serão selecionadas, e logo em seguida serem recuperadas.

Já o Serviço de Busca realiza as operações de indexação e busca. Na indexação, esse serviço percorre o repositório com o intuito de recuperar metadados de artefatos armazenados para indexá-los, e então consolidar tal operação. Para promover a busca, algum consumidor realiza a consulta (descrevendo suas necessidades) e posteriormente recuperar os artefatos identificados. O Serviço de Busca adota técnicas que tratam eficientemente essas operações citadas acima, permitindo que os consumidores desse serviço construam consultas levando em consideração as relações estruturais dos artefatos, como também as textuais [Viana11].

3.3.1. Informações de Reúso

A indústria de software vem enfrentando dificuldades relacionadas à avaliação dos atributos de qualidade dos seus produtos. Cada vendedor de software apresenta características dos produtos da maneira que lhes convém, como por exemplo, um componente de sistema de cadastros pode ser comercializado sob o pretexto que possui o tempo de resposta não maior que 30 milissegundos, já outro componente com as mesmas funcionalidades pode ser comercializado com a asserção de que o *throughput* de dados máximo cadastrados em 1 segundo é aproximadamente 200 requisições de cadastro. Caso o consumidor queira analisar as duas opções de produtos sob a perspectiva do atributo performance, ele teria que utilizar de meios subjetivos para definir qual o melhor produto que satisfaça seus critérios e suas necessidades. É comum deparar-se com a problemática de avaliar dois ou mais produtos que possuem algum atributo de qualidade com grandezas e escalas quantitativas diferentes, e também há produtos que informam suas características com termos como: bom, ruim, alto, baixo, etc. Portanto, conclui-se que atributos de qualidade (i.e. performance, usabilidade, portabilidade, etc) são compreendidos de formas distintas por pessoas diferentes [Henningsson02], onde é possível imaginar que um mesmo produto de software seja descrito qualitativamente de forma diferente para cada fabricante que se proponha a comercializá-lo.

Outro fator importante no contexto da indústria de software, principalmente para componentes COTS, se deve ao conjunto de atributos de qualidade que descrevem o produto disponível para venda. Podem ocorrer situações em que tal produto possui a descrição de quatro atributos de qualidades, mas o consumidor necessita de outras três categorias de atributos, pois no domínio de aplicação que o mesmo aborda, essas três categorias são relevantes. Cada produto pode seguir uma taxonomia qualitativa única, gerando também uma não-conformidade hierárquica dos atributos, porque um pode ser subconjunto do outro, acarretando no aumento da complexidade necessária para analisar dois produtos.

Conforme os argumentos evidenciados acima, avaliar os atributos de qualidade dos componentes COTS armazenados em repositórios é uma tarefa complexa e muito vulnerável a falhas, podendo comprometer o processo de seleção de componentes, que formarão algum sistema de software. Com a consciência de que os métodos atuais de avaliação por atributos de qualidade são eficazes somente em domínios de aplicação que adotem os mesmos padrões desses métodos, o presente trabalho propõe introduzir informações de reúso como alternativa para amenizar os problemas que surgem nas avaliações comparativas entre produtos candidatos a serem selecionados. Essas informações são intrínsecas a cada componente e podem ser exemplificadas como: grau de satisfação dos consumidores e a taxa de seleção do componente. Essas informações de reúso são de suma importância, pois dão suporte à seleção de componentes [Preiss02]. Já a utilização de modelos como ontologia e SIG [Chung99] podem ser alternativas de pesquisa para contribuir na padronização de características qualitativas dos produtos de vendedores no mercado software, pois auxiliam na hierarquia e relação entre os atributos de qualidade. Mas cada vendedor pode adotar diferentes modelagens e também podem interpretar certos atributos de qualidades de maneira distinta, portanto a problemática de comparar produtos qualitativamente permanece.

O Serviço de Repositório gerencia informações de reúso sob o pretexto de que elas facilitam a tomada de decisão dos consumidores sobre a adoção de um determinado artefato, como uma forma de proporcionar um *feedback* desse artefato para futuros consumidores. Já o trabalho proposto idealiza essas informações como métricas de software, visando substituir a abordagem clássica de avaliações de software por intermédio de atributos de qualidade. O Serviço de Repositório gerencia o grau de satisfação da maneira que cada artefato selecionado por um consumidor receberá deste uma nota de 0 a 10, e também o repositório informa o número de vezes que cada artefato foi selecionado, apontando a sua popularidade.

Um componente com alta satisfatibilidade e popularidade é considerado relevante, que remete a importância do mesmo (no meio em que se encontra), assim como o grau qualitativo que ele possui, ou seja, de forma implícita ou indireta a importância representa a percepção dos desenvolvedores sobre variados atributos de qualidade dos componentes [Bay04]. Por exemplo, um componente de software relevante possui boas qualidades, visto que foi bem avaliado em diferentes domínios de aplicação. Esta questão enriquece o processo de seleção, pois atributos de qualidade que, por ventura, não sejam tratados diretamente nessa etapa serão representados pelas informações de reúso do componente. Além disso, há atributos de qualidade importantes que são muito abstratos para serem representados [Preiss02].

Componentes com alta reputação, essencialmente têm bom grau de satisfação diante seus consumidores e bom índice de relevância, além de serem unidades bem testadas e potencialmente à prova de *bugs* [Vitharana03a].

3.3.2. Consultas ao Serviço de Busca

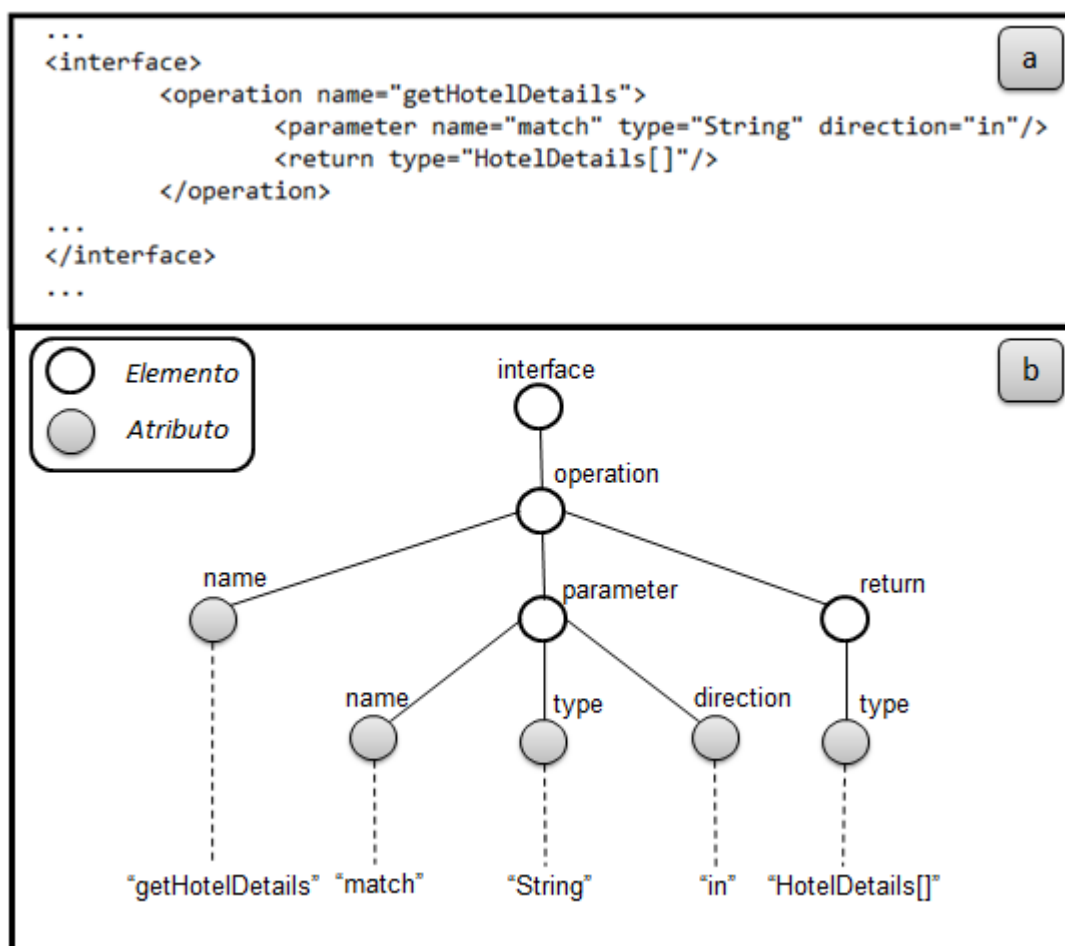
Esta etapa tem o objetivo de permitir que alguma ferramenta utilizada pelo arquiteto de software realize consultas, suportadas pelo Serviço de Busca, visando encontrar componentes de software que se aproximem dos que foram idealizados no projeto arquitetural. A consulta pode ser considerada como um meio de entrada que deve agregar o máximo de informação possível de algo que se deseja buscar, além de objetiva para não comprometer seu processamento, e que seja fácil de expressar as necessidades de informação visando evitar a ocorrência de erros semânticos no processo da consulta. A resposta da consulta é um conjunto de artefatos de software que representam componentes, os quais são candidatos para compor algum sistema em desenvolvimento.

O Serviço de Busca do *framework ComponentForge* concebeu uma técnica chamada *BranchGuide* para manipular uma classe consultas com ramificações, a qual corresponde a uma sequência de navegação que contém bifurcações, representando uma árvore sob um documento semi-estruturado, como o XML. A linguagem de construção de consultas do *BranchGuide* é baseada na linguagem XPath [W3C99], que permite localizar partes, i.e. nós, de documentos XML seguindo determinado padrão. Para ajudar e evitar a ocorrência de confusão semântica na construção de consultas (i.e. caso o usuário não domina completamente o modelo da estrutura dos documentos), o Serviço de Busca oferece interfaces gráficas para os usuários, as quais expõem explicitamente a estrutura da coleção de documentos semi-estruturados (artefatos descritos em XML), e assim guiam os usuários, como também os permitem especificar os elementos de seu interesse. A utilização dessa interface gráfica com o objetivo de facilitar a construção das consultas baseadas na estrutura de documentos é denominada de *interface gráfica de consulta dirigida via esquema* [Manning08].

Um documento XML é baseado em um modelo hierárquico e ordenado em que cada elemento (pai) pode conter um valor textual ou outro elemento (filho). Os elementos desse modelo se relacionam diretamente com ligações do tipo pai-filho como na estrutura de dados de árvores. Os documentos XML formam uma estrutura de árvore que iniciam na raiz e são ramificados até folhas [W3Schools13]. Cada elemento pode possuir um ou mais atributos, que

são partes dos desses elementos e proporcionam informações adicionais com adjetivos. O atributo possui a mesma lógica que uma variável de linguagens de programação, ou seja, são formados pelo par *nome=valor*, onde o *nome* é uma propriedade ou característica e o *valor* é um dado qualitativo ou quantitativo referente ao *nome*. No modelo X-ARM, os elementos são entidades estruturadas de forma hierárquica e os atributos descrevem tais entidades. A Figura 7 mostra um trecho de um documento que descreve uma interface, e ao mesmo tempo é evidenciada a árvore formada por elementos e atributos XML. Nota-se que todas as folhas da árvore são atributos, consequentemente os elementos são nós classificados como não-folha. Exemplificando, *interface*, *operation*, *parameter* e *return* são elementos; o restante como *name="getHotelDetails"* são atributos.

Figura 7 – Trecho de um artefato X-ARM e a árvore correspondente a esse trecho.



Segundo [Viana11], a técnica *BranchGuide* é resumidamente definida como “uma técnica projetada para ser empregada em sistemas de busca para dados semi-estruturados que idealmente disponibilizam uma interface gráfica de consulta via esquema, assim, através de consultas expressas através do eixo pai-filho (/) do XPath podem ser recuperados documentos de uma fonte de dados”.

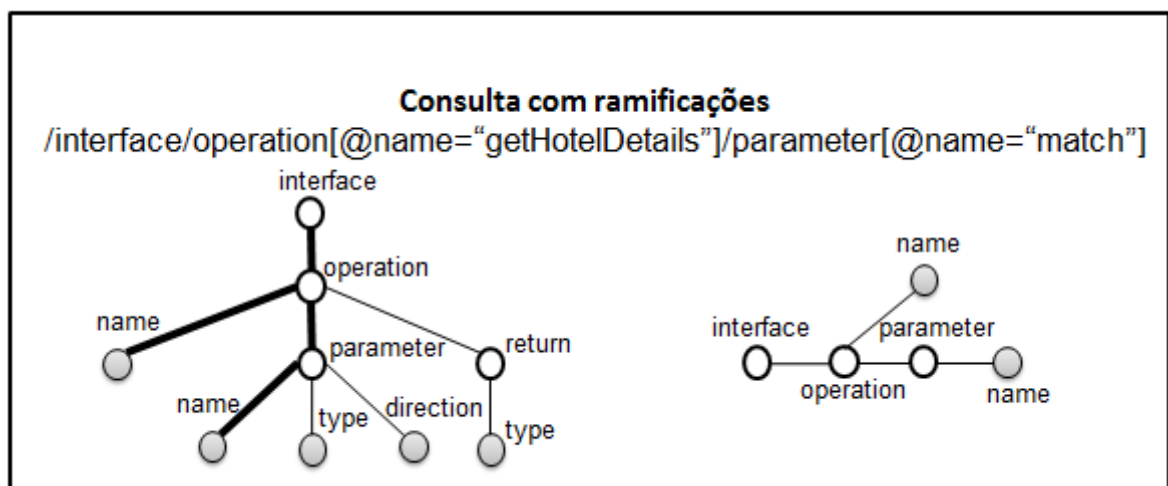
O padrão de consulta do Serviço de Busca foi concebido baseado na utilização da notação XPath [W3C99], conforme a expressão regular apontada na Figura 8 [Viana11]. A consulta deve ter pelo menos um elemento (a raiz da árvore), e cada elemento pode ter ou não um atributo que esteja associado a ele. O XPath é uma linguagem de consulta em documentos XML por meio de expressões que percorrem o caminho da árvore estruturada nesses documentos. O caractere (/) tem o mecanismo de dar um passo na árvore, mostrando que houve a transição do elemento pai para algum determinado filho, assim como em diretórios. Cada consulta retornará documentos XML compatíveis com a estrutura e o caminho elaborados por essa mesma consulta baseada na técnica *BranchGuide*, a qual utiliza a notação XPath.

Figura 8 – Expressão regular da linguagem definida pela técnica *BranchGuide*.

$$P = (/elemento([@atributo="valor"])^{0..1})^+$$

A técnica *BranchGuide* adota a abordagem de criar consultas com ramificações, pois todo elemento da consulta pode ser discriminado por algum atributo que o mesmo possua. Em uma consulta de caminho simples (sem ramificações) cada nível da árvore só pode ter um elemento ou um atributo, mas como os atributos são nós do tipo folha, as consultas só poderiam possuir no máximo um atributo. Seria embaraçoso para o cliente que quisesse buscar por uma interface que possua uma função de nome X, a qual também apresentasse um parâmetro de nome Y. A Figura 9 mostra um exemplo de consulta com ramificação, do documento XML em que é buscado. Observe que a consulta é construída sobre a regência da

Figura 9 – Exemplos de consultas com ramificações.



expressão regular da Figura 8. Outro fator importante é que toda ramificação possui apenas um nível de profundidade, visto que toda ramificação é um nó do tipo folha (atributo).

As consultas do Serviço de Busca também oferecem a propriedade de utilizar operadores lógicos em sua construção. Esta propriedade aumenta a expressividade da linguagem, pois sem ela os elementos só poderiam ser discriminados por apenas um atributo. Um exemplo é o elemento *parameter* que nas consultas representadas pela Figura 9 só é distinguido pelo atributo *name*. Mas se houver no repositório diversas operações com parâmetro de nome *match*, e se o usuário necessitasse de uma determinada operação com parâmetro de nome *match* e tipo *String*, seria recuperado muitos falsos-positivos em relação à operação que o usuário idealizou. Esta perspectiva de utilização do operador lógico refere-se ao operador *AND*.

O Serviço de Busca também adota a abordagem de recuperação de informação [Manning08], porque anteriormente ao processo de consulta, o usuário possui apenas uma noção do que ele procura, ou seja, um conteúdo parcial ou até vago. Em um repositório que contempla uma infinidade de componentes produzidos por terceiros (COTS), é impossível o usuário fornecer todas as características para recuperar algum determinado componente. Na perspectiva do trabalho proposto, o arquiteto de software, possuindo uma especificação de componente que ele planejou no Projeto Arquitetural (Seção 3.2), irá realizar a consulta que retornará componentes aproximados ao que foi planejado. Caso a busca não tivesse a abordagem de recuperação da informação, o sistema de busca só iria recuperar dados idênticos ao que fora consultado, todavia na prática, algum componente especificado em um projeto específico será quase impossível de ter uma implementação de componente (produzida por terceiros) exata ao que foi especificado por aquele projeto.

O trabalho de [Viana11], categoriza e distingue dois tipos de recuperação: dados e informação. Na recuperação de dados o usuário procura objetos que possuam atributos idênticos ao que está sendo consultado. Já a recuperação de informação o usuário procura por um conjunto de objetos que possuam apenas uma porcentagem de todos os atributos especificados na consulta. Conforme o último tipo, o usuário poderia encontrar objetos similares ao que foi idealizado e posteriormente especificado em sua consulta. Poderosos e consolidados sistemas de busca *Web* como o *Google* [Brin98] utilizam a abordagem de recuperação de informação, e sistemas de bancos de dados empregam a abordagem de recuperação de dados. Também é por meio dos operadores lógicos que a recuperação da

informação é inserida no mecanismo de consulta. Esta perspectiva de utilização do operador lógico refere-se ao operador *OR*.

O nível de precisão da consulta é definido pela forma como os operadores lógicos são usados. Uma consulta apenas com operadores *AND* restringe a busca para apenas objetos que possuam todos os atributos presentes na consulta. À medida que os operadores *AND* são substituídos por operadores *OR*, a precisão vai sofrendo relaxamento e o número de objetos recuperados tende a aumentar. A Figura 10 mostra uma consulta utilizando operadores lógicos, os quais são utilizados entre atributos irmãos (possuem o mesmo elemento pai).

Figura 10 – Consulta com operador lógico.

<code>/interface/operation[@name="getHotelDetails" ⊗ @return="String"]</code>
Onde ⊗ é um operador <i>AND</i> ou <i>OR</i>

Por intermédio de uma ferramenta, o arquiteto de software irá buscar implementações de componentes similares às especificações de componente elaboradas no Projeto Arquitetural, onde cada especificação pertencente à arquitetura de software terá um conjunto de componentes candidatos a compor essa arquitetura (montagem do sistema). O Serviço de Busca adota o documento XML como unidade de recuperação, ao invés de trechos possuindo elementos e atributos. A meta é recuperar a entidade descrita pelo documento XML, e não trechos dessa entidade. A próxima subseção apresentará como o arquiteto de software irá obter os componentes candidatos para serem selecionados, e posteriormente formarem um sistema em algum domínio de aplicação.

3.3.3. Consultas dos Componentes Candidatos

Com a posse de todos os artefatos X-ARM elaborados no Projeto Arquitetural, o arquiteto de software irá construir consultas elaboradas a partir de características descritivas dos mesmos com o objetivo recuperar artefatos que descrevem implementações de componentes similares, as quais serão candidatas para serem selecionadas. O Serviço do Repositório não armazena apenas as implementações de componentes, mas também todos os artefatos provenientes do seu desenvolvimento, e esses artefatos são descritos no modelo X-ARM. Neste processo, apenas são recuperados os artefatos de descrição provenientes do processo desenvolvimento das implementações de componente, pois há contextos em que a implementação de

componente só é fornecida após a sua compra, o que tornaria inviável. A compra da implementação só se justifica quando o componente realmente será utilizado.

A funcionalidade provida por um componente é definida pelo conjunto de interfaces que esse componente apresenta. Por outro lado, a funcionalidade das interfaces é dependente das suas operações. O X-ARM possui um artefato para descrever as especificações de componente e outro para as interfaces. As operações são descritas internamente ao artefato de descrição da interface. Portanto, os artefatos que serão recuperados são as descrições das especificações de componentes e as interfaces.

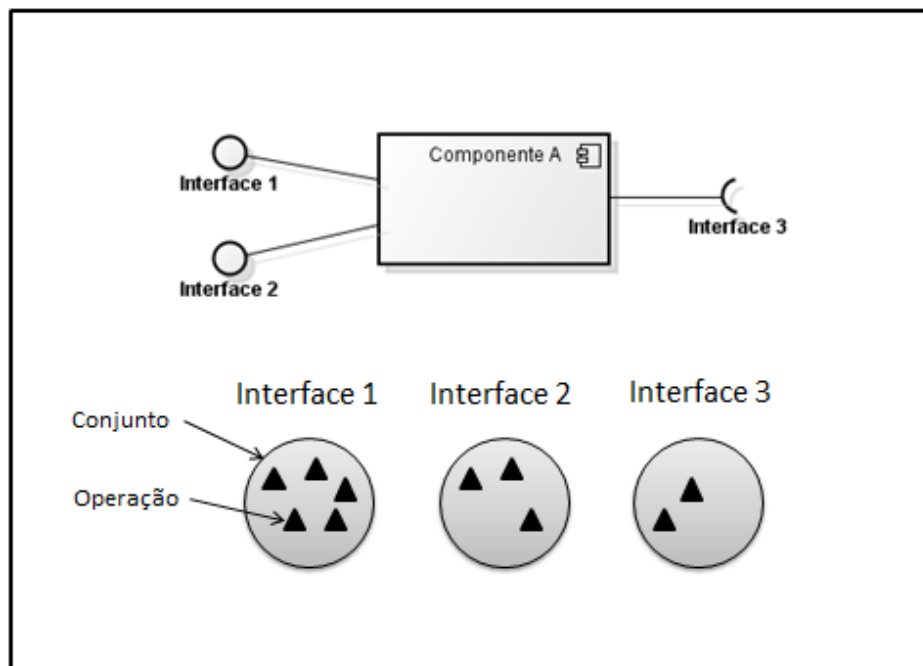
Uma descrição de especificação de componente referencia suas interfaces via identificador, que são atributos únicos de artefatos que descrevem interfaces. Ou seja, descrições de especificações de componentes apontam para descrições de interfaces. Já as interfaces descrevem suas operações, pois o modelo X-ARM estabeleceu que as operações devem ser descritas dentro das interfaces ao invés de possuírem sua categoria de artefato. Para melhor compreensão, observe a Figura 5 que descreve uma especificação de componente e a Figura 6 que representa uma interface.

Como o caminho da consulta é dependente da árvore (que é a estrutura do conteúdo inserido) em documentos XML que remetem a diversas categorias de artefatos, deverá recuperar interfaces nos documentos referentes às interfaces e posteriormente recuperar as especificações de componentes em documentos correspondentes a essas especificações. Não há como uma consulta percorrer a árvore de um documento alusivo a uma especificação, e quando chegar à referência para uma interface, percorrer o documento dessa interface.

A abordagem para procurar candidatos utilizando o Serviço de Busca é *bottom-up*, pois por meio de operações, serão encontradas interfaces candidatas, que encontrarão especificações candidatas. Para uma implementação de componente ser considerada candidata, sua especificação deve possuir pelo menos uma interface que apresente no mínimo uma operação idêntica a que esteja presente na especificação de componente definida no Projeto Arquitetural. Um diagrama de uma arquitetura apresenta várias especificações de componentes, e cada uma possuirá várias implementações de componentes candidatas. Portanto o arquiteto de software por intermédio de uma ferramenta automatizada executa essa abordagem *bottom-up*, bastando apenas apontar para as especificações de componente da arquitetura.

A Figura 11 exibe uma especificação de componente na perspectiva do processo de busca para a ferramenta. É construído um conjunto para cada interface dessa especificação e é preenchido com as respectivas operações. No conjunto da *Interface1*, a ferramenta fará consultas que buscarão por cada operação, visando recuperar interfaces similares. Cada consulta das operações é descrita na linguagem XPath, que quando processada no Serviço de Busca, retornará documentos de artefatos X-ARM que descrevem interfaces, pois as operações são descritas dentro dos documentos das interfaces. Os identificadores dos artefatos das interfaces retornadas serão armazenados em um conjunto sem duplicatas referente às interfaces similares à *Interface1*. Esse processo é análogo para as outras interfaces da especificação de componente que o arquiteto planejou no Projeto Arquitetural.

Figura 11 – Especificação de componente cujos candidatos serão buscados.



A Figura 12 evidencia a consulta na linguagem *BranchGuide* que representa uma dada operação, para poder realizar a busca das interfaces similares.

Figura 12 – Consultas que representam uma operação

```
/asset/interface/operation[@name="getHotelDetails"]/parameter[@name="match" AND @direction="in"
AND @type="String"]
/asset/interface/operation[@name="getHotelDetails"]/return[@type="HotelDetails[]"]
```

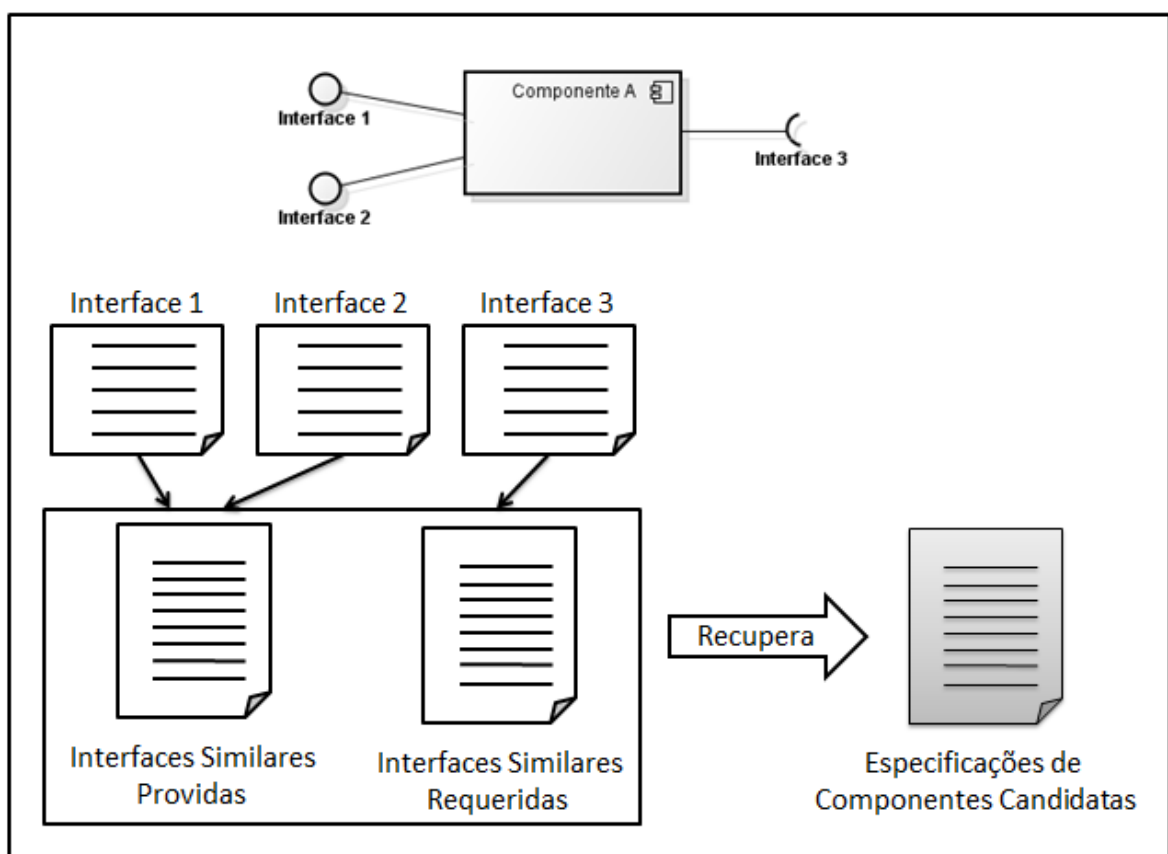
Uma interface similar a alguma outra da especificação de componente do arquiteto terá pelo menos uma operação idêntica, ou seja, todos os atributos dessa operação serão iguais. Porque sintaticamente essas operações possuem a mesma funcionalidade devido todos

os atributos serem iguais, e de certa forma semanticamente, já que possuem a mesma definição (nome), pois ela apresenta o propósito e o sentido da operação.

Após terem sido realizadas as consultas para cada operação do conjunto que remete a determinada interface especificada pelo arquiteto, todas as interfaces similares provenientes de cada operação são concatenadas em um conjunto sem duplicadas. Isso é feito para cada interface da especificação de componente. Consoante a Figura 11, há três conjuntos de interfaces similares, cada um com pelo menos uma operação igual a sua respectiva interface.

De acordo com Figura 13, os conjuntos das interfaces similares são concatenados para gerarem dois, onde um representa as interfaces que são similares às interfaces providas da especificação e o outro representa as interfaces similares requeridas. E de posse desses dois conjuntos, a ferramenta dará início a consultas que recuperarão especificações de componentes.

Figura 13 – Listas de interfaces similares que serão utilizadas para buscar componentes candidatos.



Os elementos desses dois conjuntos resultantes são identificadores alusivos às interfaces, os quais são referenciados nos documentos dos artefatos das especificações de componentes. O documento da especificação de componente não descreve interfaces em seu conteúdo, mas

apenas aponta para o identificador único referente a algum documento de artefato que descreve interfaces.

Os identificadores dos artefatos das interfaces são atributos no documento X-ARM das especificações de componente, portanto com as duas listas de interfaces similares apenas uma consulta para cada lista é possível recuperar especificações de componentes candidatas. Vide Figura 14.

Figura 14 – Consultas que retornarão especificações de componentes candidatas.

```
/asset/component-specification/interface/provided/related-asset[@id="InterfaceP1" OR  
@id="InterfaceP2" OR ...]  
/asset/component-specification/interface/required/related-asset[@id="InterfaceR1" OR  
@id="InterfaceR2" OR ...]
```

Observa-se a consulta para interface provida e a outra para interface requerida, ambas utilizam o operador lógico *OR* visando adotar a abordagem de recuperação da informação. Frisando mais uma vez, se pelo menos uma operação da especificação de componente elaborada no Projeto Arquitetural for a mesma que alguma operação pertencente a uma especificação de implementação armazenada no repositório, este componente armazenado será um candidato. Portanto, os candidatos possuem determinada parcela das funcionalidades do componente especificado no Projeto Arquitetural. Em repositórios de componentes, o arquiteto encontrará apenas componentes que satisfaçam parcialmente funcionalidades e comportamentos, pois é extremamente difícil uma implementação desenvolvida por terceiros possuir um *match* perfeito com uma especificação de componente construída para algum domínio de aplicação específico [Heineman99]. Por fim, esta seção apresentou o mecanismo de como o arquiteto de software recupera os componentes candidatos.

3.4. Definição de Métricas

Componentes oferecem funcionalidades de sistema através da interação, cooperação e coordenação entre eles, e quanto mais um par de componentes candidatos tender a uma interação com integração perfeita, maior a qualidade funcional do sistema. O arquiteto de software idealiza um sistema a ser desenvolvido com um Projeto Arquitetural construído por componentes abstratos (especificações de componente) e espera que as implementações de componentes selecionadas para construir o sistema sejam equivalentes às especificações presentes na arquitetura. Porém, como os componentes são desenvolvidos por diferentes desenvolvedores, potencialmente, são geradas incompatibilidades para a montagem dos sistemas, requerendo esforço de desenvolvimento para integrar e adaptar os componentes

candidatos. Portanto, é requerida a utilização de métricas para medir a eficiência do problema em questão [Narasimhan04].

3.4.1. Métricas em Termos da Estrutura Arquitetural

A respeito da ligação entre conectores (interfaces providas e requeridas), é natural que a integração entre componentes desenvolvidos por terceiros apresente incompatibilidades, onde interfaces com mesmo contexto funcional são parcialmente conectadas [Egyed00]. O contrato da conexão das interfaces pertencentes às especificações de componente criadas no Projeto Arquitetural deve seguir como modelo para analisar o grau do desvio da integração entre interfaces. É claro que quanto maior o desvio, maior será o número de inconsistências entre a conexão, consequentemente o esforço para reparar esse problema também aumenta.

O arquiteto de software é encarregado de montar o sistema, e projeta-o totalmente dependente do domínio de aplicação. A maneira como ele organiza a distribuição e conexão interfere no arranjo de operações de cada interface, pois o mesmo tem uma visão geral do sistema, e irá configurar as especificações de componentes para contribuírem na construção de uma boa arquitetura. Já o desenvolvedor de componentes apenas enxerga as capacidades que sua implementação de componente irá fornecer. O arranjo de operações dessa implementação é realizado da maneira que o desenvolvedor achar conveniente ou seguir algum padrão visando atender vários componentes e contribuir para produzir um componente com caráter mais genérico. Segundo [Heineman99], há dificuldades em especificar precisamente o sistema a ser construído, no fato em que um arquiteto tenta esboçar a arquitetura do sistema o mais próximo possível do especificado seguindo boas práticas de design, embora se outro arquiteto fosse executar a mesma tarefa, o esboço de sua arquitetura certamente teria alguma discrepância. A linha de raciocínio para o projeto de desenvolvimento de componentes de software também é a mesma. Todas essas possíveis variações justificam as habituais inconsistências no processo de integração de componentes de software COTS.

As interfaces possuem um conjunto de operações que foram definidas na especificação dessa interface. Se um componente provê via interface um conjunto de operações com algum contexto funcional em comum, existirá uma conexão perfeita com essa interface provida somente se houver outro componente cuja interface requeira exatamente o mesmo conjunto de operações ofertadas pela interface provida. Tem-se um *match* perfeito, pois o contrato/assinatura da interface provida e requerida foi o mesmo.

Com o objetivo de dar suporte ao processo de seleção de componentes, principalmente na análise de conexão entre componentes, foram criadas métricas de compatibilidade entre as conexões dos componentes. Esse tipo de métrica avalia a estrutura da arquitetura do sistema, visto que a mesma é construída por componentes e conectores, e os últimos demonstram as dependências e relações de como os componentes são arranjados na arquitetura. Uma boa integração entre componentes de software beneficia a qualidade geral da estrutura arquitetural.

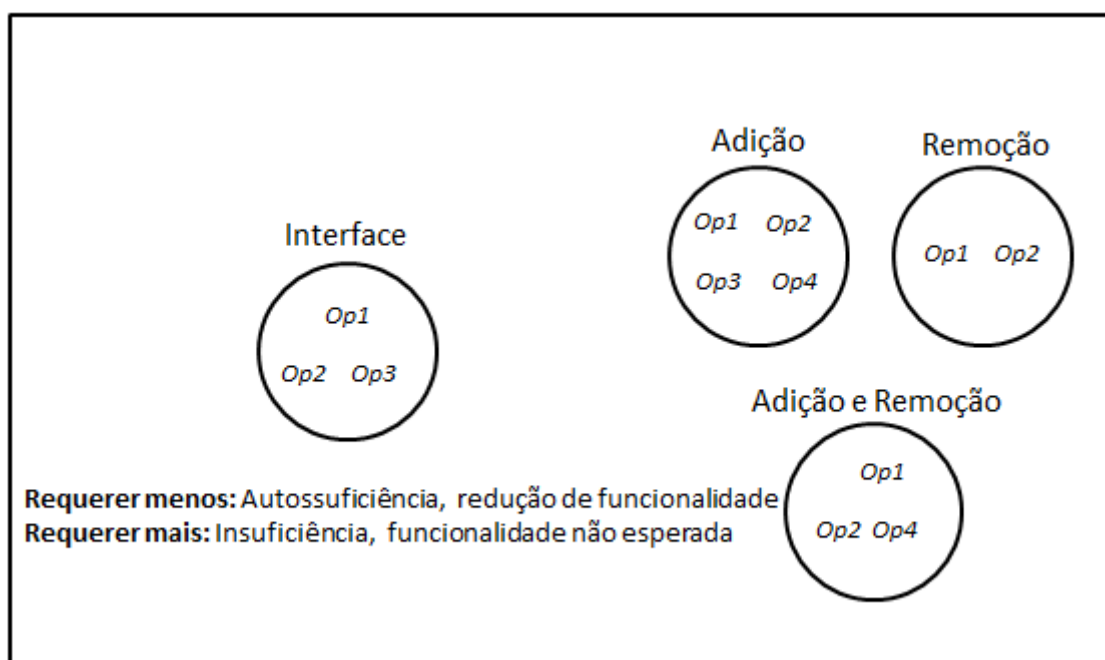
A função básica de comunicação de um componente com o outro se dá por interfaces providas e requeridas. E o ideal é que todas as operações requeridas por uma interface sejam providas por outra interface, numa conexão entre componentes. Existe a possibilidade de inconsistências quando interfaces similares (em relação as que exercem uma conexão perfeita) irão se conectar, pois pelo fato de serem similares, as interfaces providas podem ter: mais operações e menos operações em seu contrato, e isso também é análogo para a interface requerida.

Tendo uma especificação de componente que provê apenas uma interface, suponha que por ventura essa mesma especificação fosse comparada com outra especificação de componente idêntica, exceto pelo fato de que sua interface provida possui menos operações. Vamos denominar essas especificações de componentes como *Comp_A* e *Comp_B*, respectivamente. Como a interface de *Comp_B* oferece menos operações, consequentemente essa especificação proporcionará menos funcionalidades. O inverso dessa situação é quando a interface provida de *Comp_B* possui operações extras, que resultará na circunstância desse componente prover funcionalidades adicionais. Também pode ocorrer da interface de *Comp_B* possuir ambas situações citadas acima, que é a redução das funcionalidades originais e ao mesmo tempo a adição de funcionalidades diferentes.

Imaginemos de forma análoga a suposição acima, embora ao invés de interfaces providas, *Comp_A* e *Comp_B* agora possuem interfaces requeridas. A desnecessidade de requerer algumas operações reflete na especificação do componente *Comp_B* a redução de funcionalidade ou a autossuficiência. A primeira corresponde no impacto que a ausência da operação requerida poderia causar na especificação da interface provida. Se um componente depende de uma operação *Op1* para prover uma outra operação *Op2*, então a ausência da operação *Op1* acarreta também a ausência da operação *Op2*. A autossuficiência refere-se ao fato de que a operação *Op2* é provida porque a funcionalidade da operação *Op1*, a qual seria requerida, foi implementada pelo próprio componente.

Se o componente *Comp_B* requerer mais operações comparado ao componente *Comp_A*, aquele terá funcionalidades não esperadas ou apresentar insuficiência. Como o componente *Comp_B* é similar, o mesmo pode necessitar de mais interfaces em relação ao *Comp_A* porque aquele provê mais funcionalidades que este, assim as funcionalidades extras (que não foram esperadas pelo arquiteto de software) são direcionadas para a necessidade funcional, concebidas pelas interfaces requeridas. De outra forma, a insuficiência de um componente significa que componente *Comp_B* requer mais funcionalidades para suprir as mesmas interfaces providas que *Comp_A*. A ineficiência do projeto alusivo ao desenvolvimento do componente *Comp_B* acarretou em necessidades por funcionalidades extras, embora tal necessidade possa ser justificada pela pretensão de gerar mais qualidade para as interfaces providas. Observe a Figura 15.

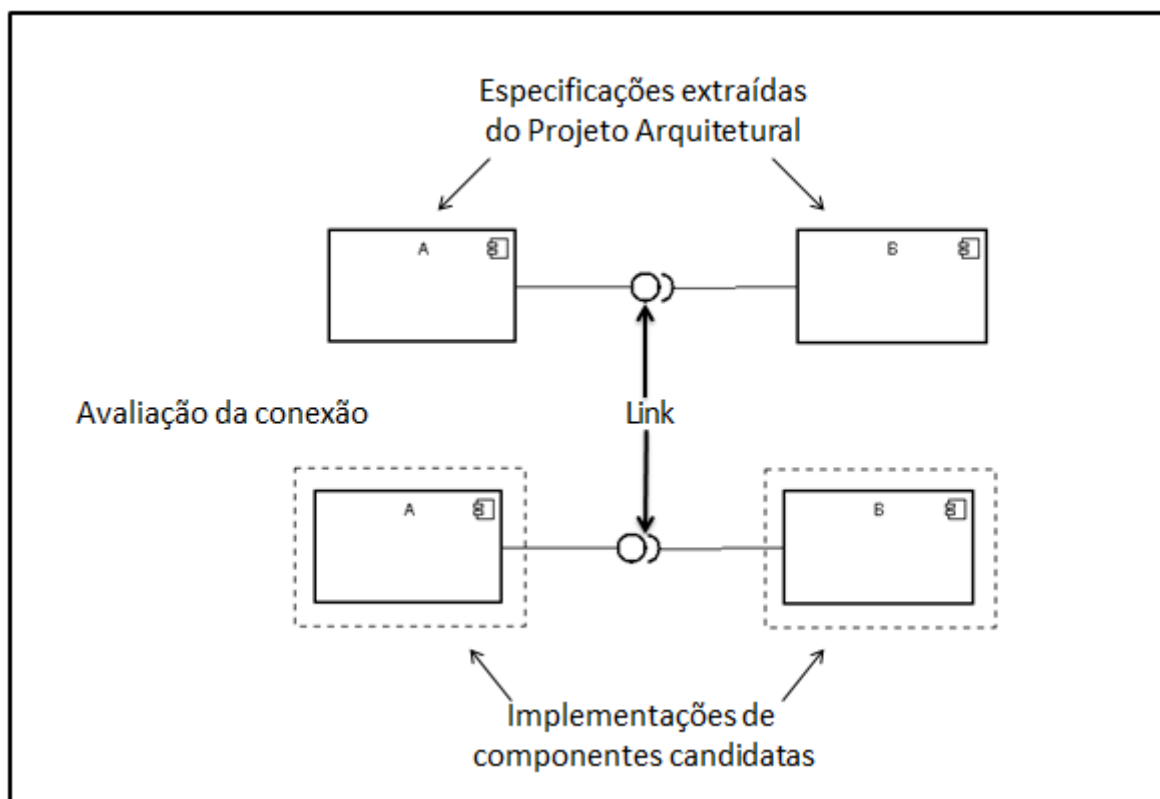
Figura 15 – Diferenças entre interfaces e suas consequências.



Para avaliar a conexão entre dois componentes candidatos deve-se analisar o grau da conformidade entre as operações da interface provida de um componente e as operações requeridas do outro, e ao mesmo tempo, analisar o quão essa conexão em avaliação corresponde a ligação das especificações de componentes que tais candidatos se referem. Não adianta obter dois componentes que se conectem perfeitamente, mas que a maioria dos itens dessa conexão são funcionalidades que não atendem à especificação do sistema a ser. A Figura 16 retrata a perspectiva de avaliação de uma conexão entre componentes candidatos. Tem-se duas especificações de componentes conectadas e seus respectivos componentes candidatos também conectados. Portanto, nessa avaliação temos quatro interfaces

participantes da avaliação. Como a conexão é formada através de interfaces, os elementos específicos para a avaliação são: interface provida da especificação de componente *A*, interface requerida da especificação de componente *B*, interface provida da implementação de componente candidata similar a especificação do componente *A*, e por fim, interface provida da implementação de componente candidata similar a especificação do componente *B*. Enfim, o processamento da métrica de avaliação de conexão entre componentes candidatos tem como entrada as quatro interfaces, representadas através de suas especificações descritas em artefatos X-ARM.

Figura 16 – Avaliação de uma conexão entre componentes candidatos.



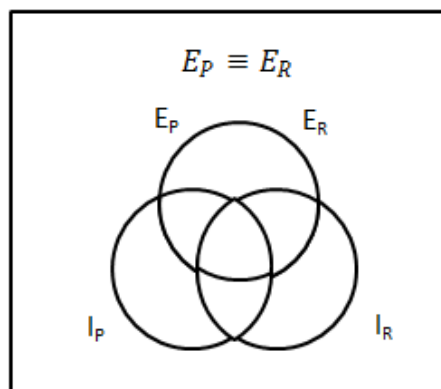
Os elementos para comparar as funcionalidades das interfaces dos componentes de software são as operações, em que cada uma representa uma parcela do sistema e computam alguma função a que ela se propõe. Das quatro interfaces envolvidas na avaliação da conexão entre componentes candidatos é estabelecido quatro conjuntos de operações referentes a cada interface dessa avaliação. Os conjuntos de operações que representam as interfaces são apresentados na Tabela 3 por identificadores, os quais facilitarão na elaboração das métricas.

Tabela 3 – Identificadores dos conjuntos de operações.

Identificador	Conjunto de Operações
E_P	Interface provida da especificação de componente
E_R	Interface requerida da especificação de componente
I_P	Interface provida da implementação de componente candidata
I_R	Interface requerida da implementação de componente candidata

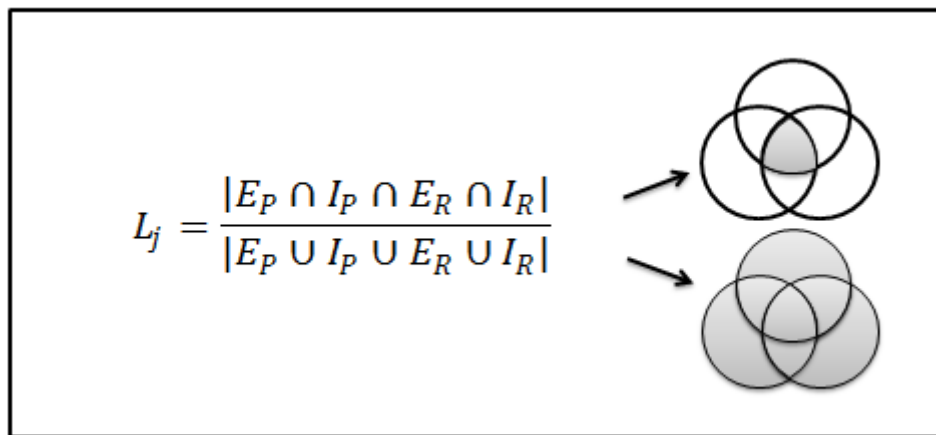
Um fato inerente a essa avaliação é que sempre os conjuntos E_P e E_R são equivalentes, ou seja, possuem o mesmo número de elementos, e cada elemento de E_P possui outro igual presente em E_R . Essa equivalência se dá pelo motivo de que o arquiteto de software quando constrói a arquitetura do sistema, ele efetua a conexão entre os componentes de modo que não existem incompatibilidades, porque é idealizado um sistema funcional com integração perfeita.

Conforme citado anteriormente, há duas dimensões que as interfaces dos componentes candidatos são avaliadas: a primeira representa a conformidade da conexão entre interfaces de componentes candidatos, e a segunda refere-se a adequação de uma interface de um componente candidato com sua interface correspondente da especificação de componente pertencente a arquitetura. O conjunto I_P é avaliado em função da sua interseção com o conjunto I_R (primeira dimensão), assim como também pela interseção entre o conjunto E_P (segunda dimensão). De maneira análoga e simétrica, o conjunto I_R é avaliado em função da sua interseção com o conjunto I_P (primeira dimensão), assim como também pela interseção entre o conjunto E_R (segunda dimensão). A Figura 17 mostra na perspectiva de diagramas de conjuntos as relações de interseções entre eles. Vê-se que os conjuntos E_P e E_R são representados por apenas um único conjunto, visto que são equivalentes.

Figura 17 – Relações entre os conjuntos.

Quanto mais elementos (i.e. operações) pertencentes à interseção entre os conjuntos (i.e. interfaces), melhor será a conexão entre os componentes candidatos. A interseção ($I_P \cap I_R$) entre I_P e I_R caracteriza a compatibilidade entre as funcionalidades dos componentes candidatos. Visando inserir a propriedade de que essa compatibilidade entre os componentes candidatos atende a especificação da arquitetura do sistema, temos $E_P \cap I_P$ e $E_R \cap I_R$. A operação de interseção é equivalente a operação lógica com *AND*, ou seja, é uma operação restritiva. Efetuando a interseção de ambas as dimensões, têm-se como resultado $E_P \cap I_P \cap E_R \cap I_R$. O ideal é que todas as operações estejam no subconjunto $Z = E_P \cap I_P \cap E_R \cap I_R$, pois demonstra que a conexão entre os componentes candidatos atende as duas dimensões comentadas anteriormente. A equação que avalia tal conexão é mostrada na Figura 18.

Figura 18 – Métrica referente à avaliação da conexão.



$$L_j = \frac{|E_P \cap I_P \cap E_R \cap I_R|}{|E_P \cup I_P \cup E_R \cup I_R|}$$

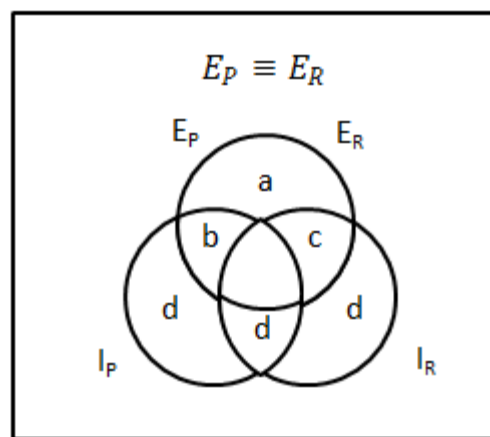
Na equação L_j , o numerador da divisão corresponde ao número de operações (elementos) que pertencem à interseção entre todos os conjuntos envolvidos na avaliação, tal interseção demonstra a conformidade simultânea com ambas as dimensões. Já o denominador representa todas as operações envolvidas na avaliação da conexão. Conforme as propriedades de conjuntos, o número de elementos da interseção é menor-igual ao da união, também se tem o fato que o intervalo do numerador é $[0, N]$ e o do denominador é $[1, N]$, resultando na equação L_j possuir valores numéricos pertencentes ao intervalo $[0, 1]$. Quanto mais próximo de 1 for o resultado, melhor será a conexão.

Esta equação penaliza na medida que as operações não atendem a ambas dimensões, pois há os seguintes problemas (vide Figura 19):

- a) Operações que pertencem às interfaces das especificações de componentes, mas não pertencem às interfaces das implementações candidatas.

- b) Operações que pertencem às interfaces das especificações de componentes e à interface provida de uma implementação candidata, mas não pertencem à interface requerida da outra implementação candidata.
- c) Operações que pertencem às interfaces das especificações de componentes e à interface requerida de uma implementação candidata, mas não pertencem à interface provida da outra implementação candidata.
- d) Operações que pertencem às interfaces das implementações candidatas, mas não pertencem às interfaces das especificações de componentes.

Figura 19 – Classificação das relações entre os conjuntos de operações



Abaixo segue a descrição para os problemas enumerados acima e também evidenciados na Figura 19.

- a) Problema relacionado a ausência de funcionalidades que foram especificadas na arquitetura de sistema, mas não foram aderidas pelas implementações candidatas.
- b) A implementação de componente que possui a interface requerida envolvida na conexão pode possuir funcionalidade insuficiente, pois como não requer as funcionalidades especificadas pelo arquiteto de software, possivelmente não proverá algumas operações. Caso o componente não requeira tais funcionalidades pelo motivo delas serem implementadas no próprio componente, haverá redundância funcional porque o outro componente também as provê. Também não há garantia que essas operações implementadas pelo próprio componentes são exatamente as mesmas que foram especificadas em interface requerida. Os componentes de software desenvolvido por

terceiros são caixa-preta, e não a como analisar quais operações requeridas impactam em operações providas.

- c) Ocorre o fato que implica na redução de funcionalidade, pois o componente deixar de prover operações especificadas na arquitetura do sistema.
- d) Esse problema se deve com a presença de operações que não foram especificadas pelo arquiteto de software. Funcionalidades adicionais providas, as quais não foram especificadas, podem consumir recursos computacionais que seriam dedicados às operações especificadas, além de que ainda essas operações adicionais podem requerer outras operações extras. Noutra vertente, as funcionalidades adicionais requeridas podem necessitar outras operações não especificadas (desviando o foco funcional do sistema), ou demonstrar que requerer apenas as operações especificadas para a implementação de componente em questão, não é o bastante para prover as operações especificadas, demonstrando insuficiência funcional.

3.4.2. Métricas em Termos dos Componentes Candidatos

Para avaliar a arquitetura de um sistema de software, a análise da qualidade de cada componente é de importância fundamental, pois eles refletem na qualidade geral da arquitetura que eles fazem parte. Quando se deseja adquirir algum componente desenvolvido por terceiros, todos os componentes de software são avaliados com base tanto nos seus requisitos funcionais quanto em seus requisitos não-funcionais, então o que melhor se encaixar nas necessidades do consumidor será selecionado.

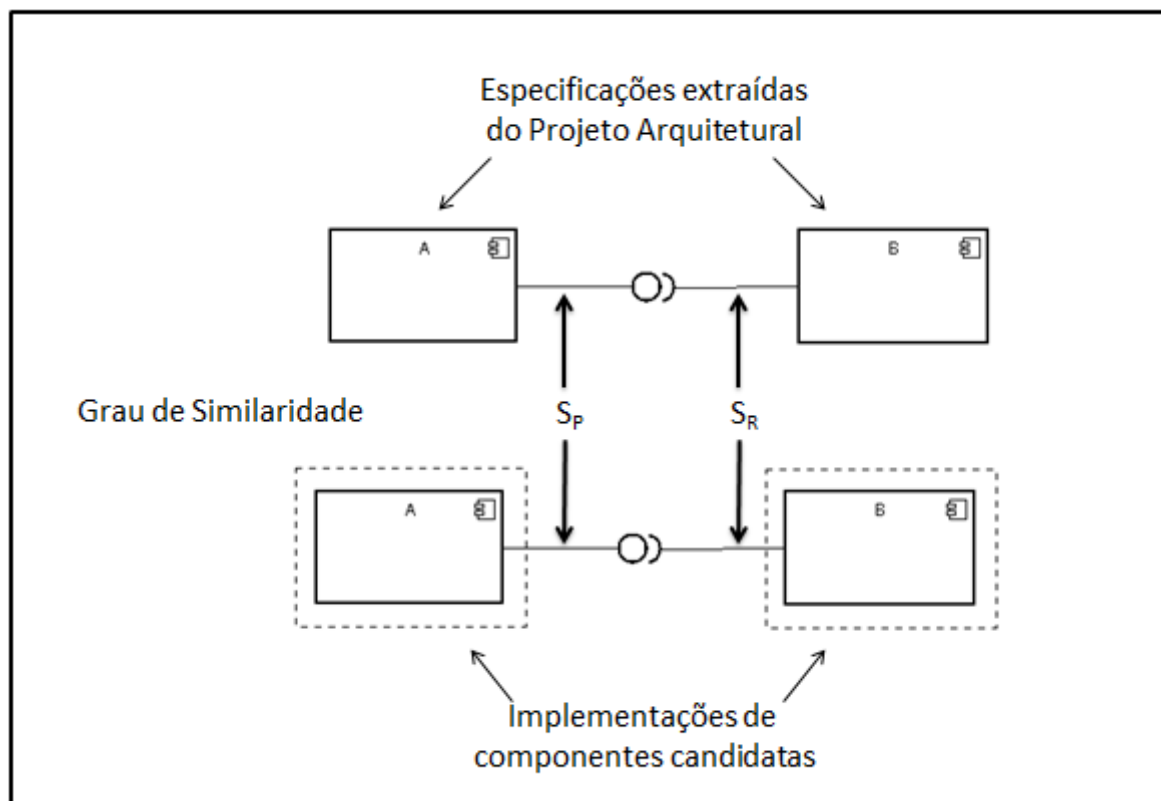
3.4.2.1. Métrica Referente ao Nível de Similaridade

Como as funcionalidades dos componentes de software são descritas via interfaces e operações, os requisitos funcionais podem ser representados por essas entidades. O arquiteto de software constrói sua arquitetura a partir de elementos que modularizam a representação do sistema. Esses elementos são as especificações de componentes, as quais possuem interfaces providas e requeridas, dessa maneira exibem suas funcionalidades. Portanto, uma implementação de componente candidata atende perfeitamente às funcionalidades especificadas se todas as suas interfaces forem equivalentes.

Partindo desse pressuposto, quanto mais similar o componente candidato for, em relação a especificação, melhor será compatível com as funcionalidades idealizadas pelo arquiteto de software. A Figura 20 evidencia a comparação por similaridade.

A comparação de uma implementação de componente candidata com a especificação de componente se dá pela medição da similaridade ou discrepância das interfaces entre ambos. Se a implementação candidata possuir menos ou mais interfaces correspondentes à especificação na arquitetura, a discrepância entre componentes acarreta na ausência de funcionalidades que foram especificadas na arquitetura do sistema, e a presença de funcionalidades que também não foram especificadas. É importante medir o nível do desvio funcional de um componente candidato a compor o sistema em construção.

Figura 20 – Comparação por similaridade



Como as interfaces correspondentes entre a implementação de componente e a especificação não são idênticas, há também a comparação em relação às operações dessas interfaces. A medição da similaridade entre a interface da especificação e sua interface candidata (similar) é formada pelo cálculo do índice de Jaccard [Jaccard1901], que foi matematicamente provado como sendo uma expressão para avaliar a similaridade entre conjuntos. Observe na equação abaixo.

$$S(C_1, C_2) = \frac{|C_1 \cap C_2|}{|C_1 \cup C_2|} \quad (4)$$

Essa equação tem resultado no intervalo $[0,1]$, à medida que os conjuntos C_1 e C_2 tendem a equivalência, a similaridade $S(C_1, C_2)$ calculada tende ao valor numérico 1. Em

contrapartida, na mesma proporção que um elemento possui mais ou menos elementos que o outro e também elementos distintos, o valor numérico dessa equação tende a 0. Essa equação possui boa adequação com a medição de similaridades entre interfaces, as quais podem ser consideradas um conjunto de operações. Para calcular a similaridade entre as interfaces providas S_P , tem-se o número de elementos (operações) da interseção entre os conjuntos (interfaces) E_P e I_P , onde representam a interfaces provida da especificação e da implementação, respectivamente.

$$S_P = \frac{|E_P \cap I_P|}{|E_P \cup I_P|} \quad (5)$$

Da mesma forma, é obtido o nível de similaridade entre as interfaces requeridas S_R a partir do cálculo de Jaccard aplicado às interfaces E_R e I_R .

$$S_R = \frac{|E_R \cap I_R|}{|E_R \cup I_R|} \quad (6)$$

Essas fórmulas avaliam apenas as interfaces do componente candidato através das operações, porém é preciso que se tenha uma medida geral para o componente, visando exibir o nível que o componente atente às funcionalidades da especificação, para isso teria que ter uma fórmula de Jaccard aplicada à comparação das interfaces, ou seja, um nível acima das fórmulas apresentadas anteriormente, pois comparam os elementos como operações. A fórmula de Jaccard é definida pelo número de elementos da interseção de dois conjuntos dividido pela união deles. Realizando o cálculo da similaridade de para cada interface do componente, têm-se valores S_P e S_R para as interfaces providas e requeridas do componente candidato. Esses níveis S_P e S_R só são calculados apenas por uma interface da especificação e uma interface da implementação candidata que possuam algum contexto funcional em comum, quer dizer correspondentes.

Comparando dois conjuntos quaisquer, os elementos correspondentes entre eles geram a interseção dos dois. Assim, todas as interfaces da implementação de componente candidata que tenha outra interface correspondente na especificação, são consideradas elementos participantes da interseção do conjunto das interfaces dos componentes que representam a especificação e a implementação. Já a união é caracterizada pela junção de todas as interfaces da especificação e da implementação. Portanto, a equação que mede o nível de similaridade funcional entre componentes está representada na equação abaixo.

$$Similaridade_i = \frac{\sum_{j=0}^n S_{P_j} + \sum_{k=0}^m S_{R_k}}{|P_E \cup P_I \cup R_E \cup R_I|} \quad (7)$$

Nessa equação, o numerador corresponde ao somatório das comparações das interfaces providas e das requeridas, onde o valor máximo de todas as somas é igual ao número de interfaces correspondentes. Valores de S_P e S_R diferentes de 1 significa que as interfaces comparadas não foram idênticas, mas possuem algum grau de semelhança, portanto o seu valor numérico faz parte do somatório da interseção. Essa equação explora as relações desde as operações até as interfaces para gerar o valor final agregado ao componente candidato. Essa equação também está presente no intervalo $[0,1]$, pois o número de interfaces correspondentes é menor-igual à união de todas as interfaces.

3.4.2.2. Métrica Referente às Informações de Reúso

Como planejado neste trabalho proposto, as informações de reúso fazem parte das métricas que proporcionarão um caminho para a etapa de seleção escolher os melhores candidatos para o sistema em construção. As métricas logo acima medem o nível de adequação funcional que um componente candidato satisfaz em comparação com a especificação que ele se propõe a atender. Esse critério de medida remete aos requisitos funcionais do projeto de construção do sistema, em contrapartida e ao mesmo tempo complementando tais requisitos funcionais, tem-se as informações de reúso. Como citado na seção 3.3.1, as informações de reúso têm o objetivo de representar os requisitos não-funcionais de forma geral, em outras palavras, os atributos de qualidades alusivos aos requisitos não-funcionais são mapeados implicitamente para as informações de reúso.

Na seção 3.3.1 observamos que a grande dificuldade de medir determinados componentes desenvolvidos por terceiros – assim como definições para certos atributos de qualidade variam de projeto para projeto – motiva a utilização de informações de reúso. Mesmo que o desenvolvedor ofereça um subconjunto de informações a respeito de algum componente que o mesmo tenha desenvolvido, sempre haverá algum domínio de aplicação que necessitará de mais atributos de qualidade para poder avaliar se o componente candidato deve ser selecionado. Ainda há o fato de que algumas implementações de componentes só podem ser testadas em algum domínio de aplicação, quando as mesmas forem compradas. Depender das informações qualitativas oferecidas pelos seus desenvolvedores é arriscado, já que os métodos que medem seus atributos de qualidade podem apresentar viés para o componente possuir um bom índice qualitativo.

O Serviço de Repositório do framework *ComponentForge* armazena as informações de reúso dos componentes e/ou artefatos que foram outrora selecionados por consumidores.

Há duas categorias de informações de reuso abordadas por este trabalho: a média do grau de satisfação dos consumidores que já adquiriu o artefato, e o número de vezes em que o artefato ou componente foi selecionado pelos consumidores. A primeira categoria corresponde a avaliação que o consumidor estabeleceu da adequabilidade qualitativa do componente em seu domínio de aplicação específico. Baseado nos requisitos não-funcionais do seu sistema em construção, o consumidor que selecionou algum artefato irá mapear a adequabilidade qualitativa dele em um valor numérico que representar o seu grau de satisfação. Quanto mais o componente selecionado desempenhar as restrições qualitativas estabelecidas pelo contexto do domínio do consumidor, maior será o grau de satisfação. A segunda categoria refere-se a popularidade do artefato/componente, onde um componente muito utilizado é considerado relevante, consequentemente são bem testados e potencialmente a prova de *bugs* [Vitharana03b].

A popularidade (relevância) de um componente remete à importância do mesmo, assim como o grau qualitativo que ele possui, ou seja, uma carga geral de diferentes qualidades, as quais estão ligadas implicitamente [Bay04]. Esta questão enriquece o processo de seleção, pois atributos de qualidade que, por ventura, não sejam tratados diretamente nessa etapa serão representados pela popularidade e grau de satisfação do componente. Além disso, há atributos de qualidade importantes que são muito abstratos para serem representados [Preiss02].

No ramo da estatística, a estimativa é um cálculo aproximado, onde um subconjunto de amostras apresentará o comportamento de um conjunto bem maior que é chamado de população. Quanto mais amostras o subconjunto tiver, maior será a credibilidade da estimação de medições relacionadas à população, ou seja, o erro amostral é diretamente proporcional ao número de amostras. Trazendo para a perspectiva das informações de reuso, quanto mais selecionado um artefato ou componente for, maior será a credibilidade da média do grau de satisfação dado pelos consumidores. A média do grau de satisfação obtida por um número pequeno de domínios de aplicação que algum componente foi implantado não poderia representar uma infinita gama de domínios de aplicações específicos. Partindo desse pressuposto e também com o objetivo de comparar componentes candidatos, a credibilidade do grau de satisfação é dependente das popularidades do componente em avaliação com o componente mais popular entre os candidatos. A equação abaixo demonstra a métrica para avaliar componentes ou artefatos em relação às informações de reuso.

$$\mathbf{Reuso}_i = \mathbf{g}_i \times \left(\frac{d_i}{d_{max}} \right) \quad (8)$$

Nessa equação, d_i corresponde número de vezes que o componente em avaliação foi selecionado anteriormente, d_{max} refere-se à quantidade que o componente mais selecionado dentre os candidatos foi escolhido, ou seja, d_{max} é maior-igual a d_i . E g_i representa a média do grau de satisfação atribuída pelos consumidores que selecionaram outrora o componente em avaliação. À medida que a popularidade do componente em avaliação se aproxima da popularidade do componente mais selecionado, maior será a credibilidade do grau de satisfação do componente. O parâmetro g_i está contido no intervalo $[0,1]$ e a divisão entre as popularidades também está contida no intervalo $[0,1]$, consequentemente o resultado da equação correspondente às informações de reúso está no intervalo $[0,1]$. Uma baixa popularidade em relação ao componente candidato que foi mais selecionado resultará em valor próximo de zero (na divisão), portanto penalizará mais a credibilidade do grau de satisfação.

3.4.2.3. Métrica Geral Referente aos Componentes Candidatos

Agora que foram definidas as métricas que mapeiam os requisitos funcionais e não-funcionais do componente, e como ambas qualificam os componentes candidatos, é possível inseri-las em uma única equação, que corresponde a uma métrica geral que avalia tais componentes candidatos. Considerando que se um componente atende totalmente os requisitos funcionais, mas possuem péssima qualidade referente aos requisitos não-funcionais, pouco importa a total adequabilidade funcional, visto que a baixa qualidade dos seus atributos de qualidade seria desastrosa para a aplicação. Da mesma forma, se o componente não atender os requisitos funcionais, desviará o foco ao que foi especificado na arquitetura do sistema em construção, então pouco importará ter excelentes atributos funcionais. Em suma, justifica-se o operador de multiplicação para unir as duas métricas e gerar apenas um único resultado que classifique a entidade do componente. A Equação 9 evidencia essa questão abordada.

$$\mathbf{C}_i = \mathbf{Similaridade}_i \times \mathbf{Reuso}_i \quad (9)$$

De posse das métricas estruturais e das métricas que correspondem às informações de reúso, têm-se formas de avaliar tanto a arquitetura do sistema quanto os componentes que as compõem. É importante medir os elementos da arquitetura, assim como as relações entre eles.

3.5. Seleção de Componentes

Nesta etapa do processo de Desenvolvimento Baseado em Componentes (DBC), a etapa de Seleção de Componentes tem o propósito de retornar uma configuração arquitetural que atenda aos requisitos especificados no Projeto Arquitetural, para então o sistema ser construído (integrar os componentes) na etapa de Montagem do Sistema. Como abordado nas seções anteriores deste capítulo, o Projeto Arquitetural fornece a especificação da arquitetura do sistema, cujos elementos (especificações de componentes) terão várias implementações de componentes candidatas, onde apenas uma será selecionada para ser implantada no sistema em construção. Cada implementação não é escolhida somente por seus atributos e características próprias, mas também a escolha depende dos outros componentes que o mesmo irá se relacionar diretamente (integração de componentes).

Após a conclusão do Projeto Arquitetural, o arquiteto de software obtém a descrição da especificação de cada componente da arquitetura do sistema, logo em seguida o mesmo irá realizar a busca no Serviço de Repositório para recuperar as descrições dos componentes candidatos alusivas às especificações de componente. Juntamente com as descrições dos candidatos, também são recolhidas suas informações de reúso.

As métricas de software aplicadas no processo de seleção de componentes deste trabalho proposto serão de suma importância para guiar o algoritmo de seleção visando escolher uma solução próxima da ótima (ou até mesmo a ótima) de forma automatizada. Todas as informações do problema serão mapeadas para a meta-heurística escolhida como o motor de seleção de componentes, dessa maneira, as descrições de componentes candidatos retornadas pela consulta do arquiteto, as informações de reúso e as métricas farão parte do processo de seleção. Essa abordagem configura-se em um problema da Engenharia de Software Baseada em Busca, pois aponta a situação de selecionar uma configuração arquitetural aceitável, diante de inúmeras combinações de componentes que possam ser formadas, então uma meta-heurística buscará por soluções de acordo com as métricas pertencentes ao contexto do problema.

3.5.1. Problema NP-Completo

No momento em que se tem um problema cuja resolução utilize meios computacionais, é importante descobrir a sua complexidade para poder modelar um algoritmo que o resolva. No trabalho proposto, utiliza-se da abordagem de Engenharia de Software Baseada em Busca, a qual é definida por tratar um problema de busca da Engenharia de

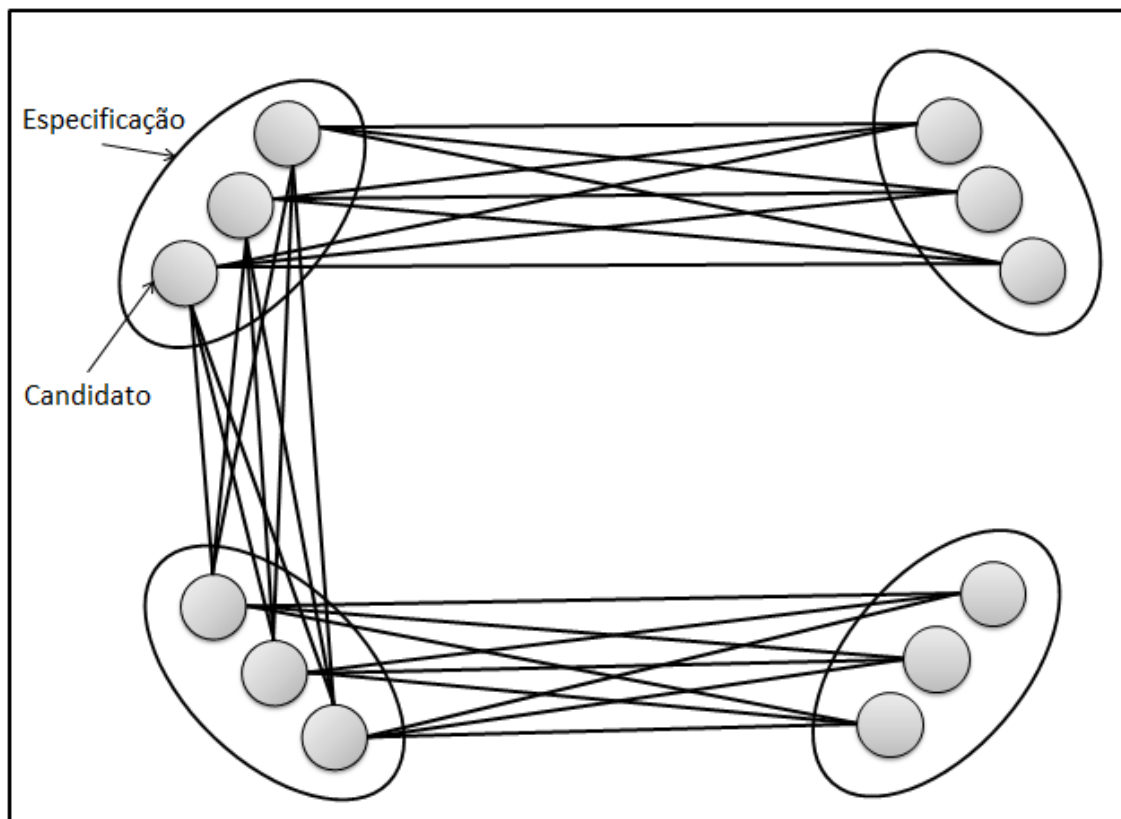
Software com algoritmos de otimização (meta-heurísticas). O problema deste trabalho é o de seleção de componentes de software, onde a partir de uma arquitetura de software produzida no Projeto Arquitetural, têm-se especificações de componentes com várias implementações de componentes candidatas.

Considerando que para cada especificação na arquitetura têm-se vários candidatos, e que o objetivo é selecionar a melhor configuração/instância arquitetural, então métricas de software irão avaliar a arquitetura sob diferentes perspectivas. Neste trabalho, são exploradas duas perspectivas de avaliação: o próprio componente e a estrutura da arquitetura (conexões entre os componentes). A primeira é considerada um problema de complexidade linear, pois para cada conjunto de componentes que corresponde a alguma especificação, escolhe-se o melhor. Detalhando melhor, cada especificação de componente da arquitetura possui um conjunto de candidatos, e de acordo com as métricas e critérios que avaliam o componente, é escolhido o melhor. Então percorre-se cada conjunto e escolhe-se o melhor. Desse modo, a complexidade é linear.

Já a outra perspectiva que avalia a estrutura da arquitetura se concentra nas conexões entre os componentes. Como cada componente pode se conectar com vários outros, a escolha de um componente candidato de uma especificação pode impactar noutros componentes candidatos. Cada decisão de selecionar um candidato impactará na escolha de outro e vice-versa (simétrico), assim como impactará em cadeia (transitivo). De forma semelhante ao problema do Caixeiro Viajante, a escolha de um caminho impacta no outro, e sabe-se que abordagens gulosas (escolhe sempre a situação mais vantajosa no turno) resultam em falhas.

Para facilitar o entendimento do problema referente à avaliação estrutural, o problema é mapeado para a visão de grafos. De acordo com a Figura 21, vê-se quatro conjuntos que representam quatro especificações de componentes de uma arquitetura.

Figura 21 – Arquitetura na perspectiva de grafos.



Cada conjunto correspondente a uma especificação que possui n implementações de componentes candidatas. Cada candidato (vértice de um grafo) de uma especificação possui ligações via arestas com todos os candidatos de outros conjuntos, exceto com os candidatos do próprio conjunto, pois a intenção é escolher um candidato para cada conjunto de especificação. Toda aresta possui um peso que corresponde ao valor quantitativo resultante da avaliação da métrica que analisa a conexão entre candidatos.

A estrutura da arquitetura é avaliada pelo somatório dos pesos das arestas entre os componentes candidatos escolhidos em cada conjunto. A definição do problema estabelece que quanto mais o valor resultante desse somatório aumenta, melhor é a qualidade geral da estrutura da arquitetura. Isso é caracterizado como um problema de maximização e que a escolha de um componente pode impactar em toda a arquitetura. O mesmo é um problema NP-Completo, e se define por maximizar a escolha de um subgrafo k -vértices de um grafo ponderado com k -partições [Feder06]. Dessa forma, o presente trabalho necessitou buscar uma abordagem algorítmica baseada em meta-heurísticas para tratar esse problema.

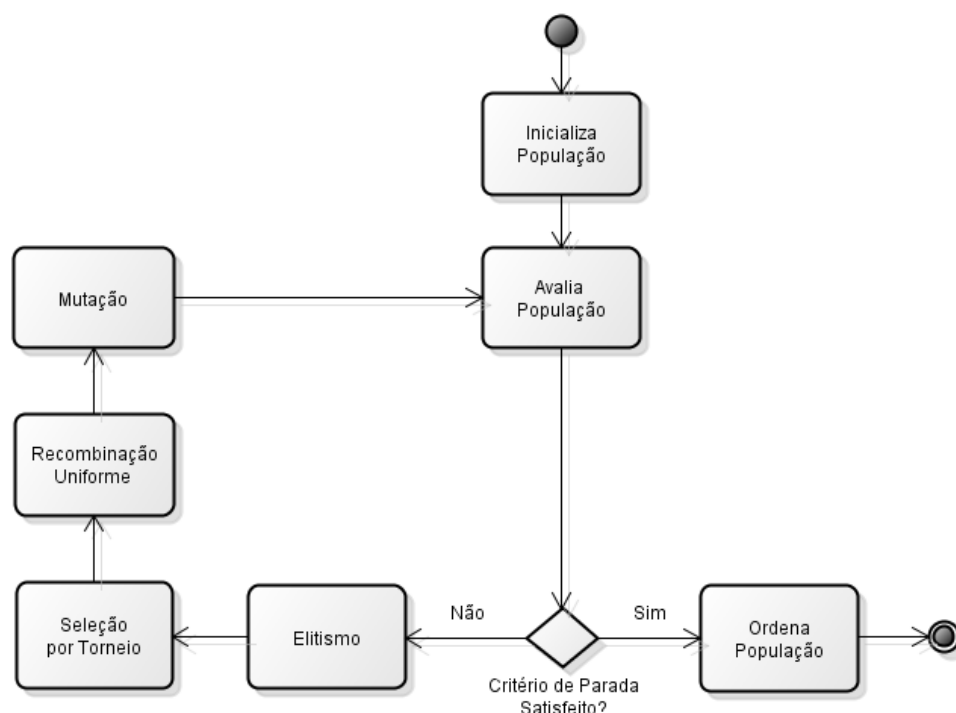
3.5.2. Meta-Heurística Escolhida

Este trabalho utiliza Algoritmos Genéticos multi-objetivo, uma meta-heurística que busca otimizar múltiplos objetivos potencialmente conflitantes. A escolha em utilizar esta meta-heurística se deve pelo motivo de que Algoritmos Genéticos (AG) são adequados para problemas de otimização combinatória (i.e., seleção de componentes de software) [Neumann04], assim como altamente eficientes para encontrar soluções multi-objetivo [Deb01]. Outro fator importante é que os AG proporcionam ao arquiteto um conjunto de soluções candidatas ordenadas por escores resultantes da função de adequação. Pois caso o arquiteto não tenha total segurança nos parâmetros da meta-heurística, o mesmo terá um conjunto de poucas soluções para reavaliar e, baseado em sua experiência, escolher uma das soluções candidatas.

A utilização de AG se fundamenta no processo evolucionista de seleção natural, onde os mais fortes sobrevivem diante dos mais fracos [Forrest96]. Sendo assim, é necessária a criação de uma função que avalie se uma configuração arquitetural é considerada boa ou ruim, definindo assim quem são os mais fortes. Essa função é conhecida como função de adequação. Outro fator importante da utilização de Algoritmos Genéticos é a intensa investigação e prática no cenário de problemas de otimização multi-objetivo, sendo esta meta-heurística a abordagem mais popular para solucionar tais problemas [Konak06].

A Figura 22 ilustra o ciclo de vida um algoritmo genético multi-objetivo, com suas etapas

Figura 22 – Ciclo de vida do AG proposto neste trabalho



bem definidas desde o início do algoritmo até o seu término, ou seja, quando o critério de parada for satisfeito. Comparado com as etapas do AG simples, o trabalho proposto adota a prática elitista, onde os melhores indivíduos de uma população são preservados para fazerem parte da próxima seleção, isso reflete no fato de que a melhor solução de certa população não será pior que o melhor indivíduo da população anterior.

Ratificando o ciclo do AG, inicialmente é criada a população de indivíduos, os quais são soluções candidatas, de forma aleatória visando evitar possíveis vieses e que a população se espalhe ao máximo no espaço de busca. Logo em seguida, a população é avaliada pela função de adequação e são gerados os escores para cada indivíduo. Sempre após a avaliação, o algoritmo testa se o critério de parada estabelecido foi satisfeito, caso afirmativo o algoritmo parte para iniciar mais uma iteração, caso contrário o algoritmo se prepara para seu término e disponibiliza a população final.

Seguindo o ciclo, a etapa de elitismo consiste em copiar os melhores indivíduos da população anterior para fazerem parte da população que está sendo formada. Esta abordagem elitista garante que o desempenho do AG sempre cresça conforme o decorrer das iterações do algoritmo. A função que demonstra o melhor indivíduo da população no decorrer do tempo tem o comportamento monótono e crescente. Como o AG é estocástico por natureza (não possui memória), caso a população tivesse um indivíduo que possuísse a solução ótima, essa mesma solução seria perdida, pois toda a população “morreria” para ser gerada uma nova população. Mesmo se essa solução ótima gerasse soluções filhas por meio dos operadores genéticos, essas novas soluções não seriam ótimas. Portanto é importante proporcionar ao AG uma pequena parcela de memória para que uma solução boa que fora conquistada não seja perdida.

Segundo [Linden08], o ato de utilizar a abordagem elitista, apesar de ser simples, usualmente colabora de forma dramática para a melhoria do desempenho do AG. As melhores soluções são mantidas dentro da população e são responsáveis por contribuir para que boas soluções sejam geradas a partir da recombinação com os melhores indivíduos da população anterior.

O próximo passo do algoritmo consiste na etapa de seleção por torneio, a qual selecionará os indivíduos da população para gerarem novos indivíduos (por intermédio dos operadores genéticos) e assim, construir a nova população. Esses novos indivíduos se juntarão aos melhores indivíduos da população anterior. Os operadores genéticos pertencentes ao AG proposta são divididos em duas etapas: a recombinação uniforme e a mutação. A recombinação recebe da etapa de seleção quaisquer dois indivíduos pais e retornam dois indivíduos filhos. Diversas recombinações são executadas até se ter o número de indivíduos para a nova população. Depois

do fim da etapa de recombinações, o operador de mutação atua apenas nos indivíduos provenientes da etapa anterior. Os indivíduos referentes à prática elitista se mantêm intactos, ou seja, não sofrem mutação.

Após o algoritmo obter a nova população, ela deve ser avaliada visando descobrir nos escores de cada indivíduo os seus níveis de adequação, em outras palavras, deve-se avaliar as novas soluções candidatas. Depois da etapa de seleção é iniciada mais uma nova iteração do AG proposto, e irá iterar repetidamente até que a condição de parada seja satisfeita. Depois de concretizada a condição que o algoritmo irá parar, é realizada a etapa de ordenação com o intuito de apresentar a população ordenada para o arquiteto de software obter um subconjunto das melhores soluções. Essa última pode ser considerada com um pós-processamento do AG.

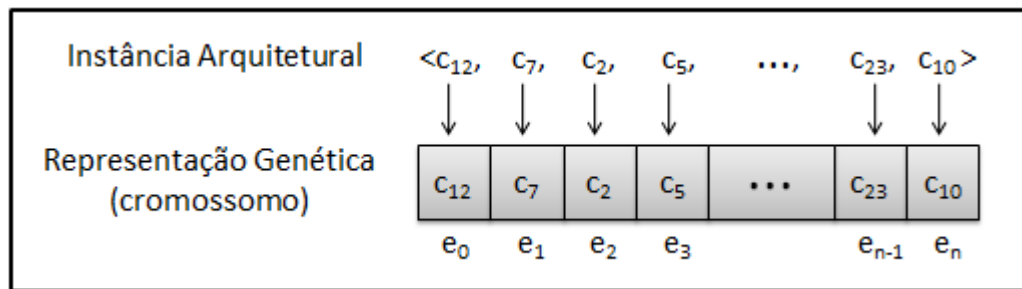
As seções seguintes detalham as técnicas escolhidas para os ingredientes essenciais do algoritmo genético criado no desenvolvimento do trabalho. Será mostrada a reformulação do problema para a representação genética, a técnica de seleção utilizada e detalhes da execução dos operadores genéticos.

3.5.3. Representação Genética

A representação genética consiste em transformar conceitos do problema em símbolos para que seja possível realizar manipulações simbólicas do algoritmo genético [Harman01]. Um cromossomo ou indivíduo do AG representa uma solução. Então, deve-se construir uma representação que mapeie soluções referentes ao problema em cromossomos. Na perspectiva do problema, uma solução corresponde a uma instância arquitetural, a qual é formada por componentes de software. De forma análoga, um cromossomo também corresponde a uma solução e é formado por genes. Assim, este fato implica que os componentes de software candidatos podem ser representados por genes.

Para o problema atual, em que se busca a solução mais adequada para uma arquitetura com n componentes distintos, a representação pensada para o algoritmo genético utiliza um vetor de n componentes, onde cada posição do vetor representa a escolha de um componente candidato entre as opções para determinada especificação de componente da arquitetura. A combinação de todas as posições do vetor representará a arquitetura proposta. A Figura 23 ilustra a representação escolhida. Nela, o cromossomo representa o vetor contendo a instância arquitetural, onde c_i é o componente escolhido para desempenhar o propósito da especificação de componente e_i .

Figura 23 – Representação genética escolhida



Caso alguma especificação de componente não possua candidatos (não foi encontrada no repositório nenhuma implementação de componente similar à especificação), então ela será removida do vetor, portanto o vetor só possuirá um tamanho correspondente ao número de especificações de componentes que possuem implementações candidatas. A ocorrência da ausência de candidatos para alguma especificação de componente não impactará na execução do algoritmo, pois o mesmo processará uma arquitetura como se aquela especificação não existisse, então, nenhum componente candidato enxergaria a especificação nula.

Não importa a ordem em que as especificações estão dispostas no vetor, o fato relevante é apenas saber quais implementações de componentes foram selecionadas para compor o cromossomo. Cada candidato encapsula informações de quais interfaces e operações ele possui, além das informações de reuso e quais componentes ele se conecta mediante informações descritas na arquitetura. Como o problema do trabalho proposto é de combinatória e as informações inerentes aos candidatos foram encapsuladas nos genes do cromossomo, então o mapeamento do problema para a representação genética foi direto. Adicionalmente, não ocorrem perdas de informação. Então o objetivo de inserir o máximo de informação do problema dentro da representação genética foi satisfeito com sucesso.

Esta representação permite que o processo de busca do algoritmo genético seja efetuado nas diversas possíveis instâncias de arquitetura para o problema a ser resolvido. Ao percorrer este espaço de busca, o algoritmo genético terá condições de encontrar boas arquiteturas, garantindo uma solução adequada para o problema de seleção de componentes.

3.5.4. Seleção

O processo de seleção é utilizado para escolher quais cromossomos da população atual serão considerados pais e gerarão a próxima população do algoritmo genético. É importante que a seleção dê alguma preferência para os melhores indivíduos da população, baseado na função de adequação, mas também permita que indivíduos que não sejam os melhores tornem-se

pais, garantindo diversidade genética e uma busca mais aprofundada no espaço de soluções [Goldberg91].

Para o algoritmo genético proposto, é sugerida a utilização da técnica de seleção por torneio. Nesta técnica, uma determinada quantidade de indivíduos da população é escolhida aleatoriamente, onde essa escolha aleatória não favorece a nenhum indivíduo. Em seguida, há competição direta destes indivíduos pelo direito de ser pai, de acordo com a função de adequação de cada um. O melhor indivíduo desse torneio será selecionado para ser pai. A escolha aleatória dos indivíduos para o torneio é efetuada de modo que um indivíduo que já foi escolhido para concorrer ao torneio também pode ser escolhido novamente. Como o pior elemento da população sempre perderá no torneio para qualquer outro indivíduo, então ele só poderá ser selecionado para ser pai se o torneio for formado apenas por esse pior indivíduo. Para o algoritmo não perder diversidade, é comum utilizar o valor numérico 2 como o número de indivíduos do torneio, para que os indivíduos não tão bons também tenham chance de ser pais. Além disso, evidências empíricas apontam que o método do torneio com tamanho de dois participantes é melhor que o método de seleção por roleta viciada [Linden08].

A seleção por torneio garante a menor perda de diversidade e maior variância de seleção em relação a outras técnicas comuns de seleção em algoritmos genéticos [Birckle95]. Por conta disto, é considerada uma das melhores opções e esperam-se bons resultados com sua utilização.

Em conjunto com a seleção por torneio, deve ser utilizado o elitismo no algoritmo genético. Elitismo é um conceito de algoritmos genéticos que garante a manutenção dos melhores indivíduos de uma geração na geração seguinte. A utilização de elitismo melhora a convergência em algoritmos evolucionários multi-objetivos [Deb01], como o proposto no corrente trabalho.

Tratando da etapa de seleção dos Algoritmos Genéticos, o esquema de seleção por torneio provavelmente é a abordagem mais popular devido a sua simplicidade de implementação e eficiência [Goldberg91][Xie09]. Em um problema clássico de combinatória como o problema do caixeiro viajante, o qual serve de exemplo para testar o desempenho de novas abordagens algorítmicas de meta-heurística, o método de torneio superou outros esquemas bastante utilizados como as seleções por roleta viciada e a seleção baseada em ranqueamento [Razali11].

A roleta viciada consiste em selecionar os indivíduos pais com base na proporção do valor resultante da função de adequação. Se um indivíduo A possuir uma *fitness* duas vezes maior que algum outro indivíduo B , a chance do indivíduo A ser selecionado também será duas vezes maior. Caso a população por ventura tenha alguns superindivíduos (indivíduos com *fitness* bem acima comparado ao restante da população), eles quase sempre serão selecionados e resultará na rápida convergência genética, impactando em uma baixa exploração do espaço de busca. Há também a situação em que a população tenha vários indivíduos com *fitness* bem próxima, então com esse método não haveria distinção considerável para seleção de indivíduos melhores.

A concepção da seleção baseada em ranqueamento produz a redução da pressão seletiva, onde a convergência genética demora um tempo maior para convergir, pois explora melhor o espaço de busca. Essa seleção requer uma população ordenada em relação às suas *fitness*, para então mapear os valores de tais *fitness* em um valor discreto, da maneira que usualmente o indivíduo com menor *fitness* obtém o valor numérico 1 e esse número é incrementado de 1 em 1 à medida que se vai atribuindo valores aos melhores elementos. Percebe-se que o escore do melhor indivíduo será N e o escore do pior indivíduo será 1. Apesar da seleção baseada em ranqueamento solucionar o problema intrínseco à presença de superindivíduos, se houver um indivíduo com *fitness* bem menor que o seu subsequente numa população ordenada, essa diferença significativa será mapeada para uma diferença pequena e uniforme. Isso provoca uma diferença monótona entre os indivíduos, e irá destruir a proporção comparativa de um indivíduo para com o outro. Além do mais, esse método necessita que a cada iteração do algoritmo seja executada a ordenação da população, assim é importante salientar que o processo de ordenação é extremamente oneroso em termos temporais para algoritmos que executam várias iterações [Linden08].

Possivelmente a única desvantagem do método de seleção por torneio se dá pelo motivo de que não é garantida a reprodução do melhor indivíduo, mas essa deficiência é solucionada com a utilização da prática elitista no AG [Matoušek08]. Adicionalmente, esse mesmo autor ressalta que a seleção por torneio possivelmente é o melhor mecanismo de seleção em vários quesitos.

3.5.5. Operadores

A escolha dos operadores de recombinação e mutação é fundamental no algoritmo genético. Uma má escolha de operadores pode restringir a diversidade genética das soluções

encontradas, diminuindo a efetividade do algoritmo e aumentando a possibilidade de parada em ótimos locais.

O operador de recombinação selecionado é o de recombinação uniforme. Neste operador, é gerada uma *string* binária, denominada máscara, que é utilizada para identificar quais genes dos pais serão permutados, e essa recombinação de genes irá gerar os filhos. Para cada posição do vetor do cromossomo, o vetor da máscara gera o valor 0 ou 1. Se o valor na posição i da máscara tiver valor 0, os genes dos cromossomos pais não serão permutados, caso o valor for 1, haverá a permutação. É importante frisar que os valores binários da máscara são gerados aleatoriamente a cada recombinação entre cromossomos pais. O crossover uniforme é provavelmente o mais utilizado por conta de sua eficiência em não apenas identificar e proteger os genes comuns, mas também recombinar os não-comuns [Hu07]. O operador é representado na Figura 24.

Figura 24 – Recombinação uniforme



O operador de recombinação uniforme é mais propício a problemas de combinatória, porque qualquer gene pode ser passível de permutação entre os pais, diferentemente da abordagem de recombinação simples em que os valores dos genes das pontas do cromossomo tendem a permanecerem juntos, e se caso essa união de valores implique em uma redução de qualidade no indivíduo, a destruição dessa união ficará a mercê da mutação genética. A recombinação uniforme possui maior chance de quebrar esquemas da estrutura do cromossomo, isso ajuda ao AG varrer melhor o espaço de busca, e se por ventura a recombinação uniforme destruir alguma solução muito boa, a abordagem elitista irá repô-la.

Conforme [Linden08], em média a recombinação uniforme possui melhor desempenho que as recombinações simples de um e dois pontos de corte.

O outro operador é o de mutação. No operador proposto, cada posição de um vetor de representação possui probabilidade p de sofrer variação em seu valor. A variação ocorre com a escolha de um valor aleatório dentre os possíveis para aquela posição que substitui o valor antigo. A mutação proporciona uma melhor diversificação na varredura do espaço de busca, permitindo escapar de máximos locais.

3.5.6. Função de adequação

Como o algoritmo genético utilizado neste trabalho tem múltiplos objetivos e cada objetivo possui níveis de importância variados, pois o arquiteto de software pode definir diferentes graus de importância para cada objetivo, é proposto utilizar o método da soma ponderada para compor uma *fitness* relativa a uma instância arquitetural. Esse método consiste em combinar funções de adequação individuais em uma única função composta, e cada função de adequação possui um peso agregado a ela para refletir na representação no seu grau de importância diante as outras funções de adequação.

Neste trabalho há dois objetivos, pois no processo de seleção de componentes é idealizado que as implementações de componentes escolhidas para construir o sistema possuam boa qualidade e que a integração entre esses componentes sejam as melhores possíveis. A qualidade geral do sistema é gerada pela qualidade de seus elementos (componentes de software) e a sua estrutura, a qual representa as conexões entre os componentes desse sistema. Dependendo do sistema, o grau de importância dessas duas vertentes pode diferir e a escolha de tais graus é realizada pelo tomador de decisão do projeto, onde é possível que o mesmo seja o arquiteto de software.

Nas seções anteriores foram apresentadas métricas que avaliam as implementações de componentes candidatas (Equação 9) e as suas conexões (Figura 18). Todas as conexões entre as implementações de componentes candidatas são avaliadas e cada valor resultante dessa avaliação deve contribuir para gerar um único valor que represente a qualidade da estrutura arquitetural. Portanto, a equação abaixo soma o valor qualitativo de cada avaliação e depois executa a divisão pelo número de conexões, visando normalizar a avaliação estrutural da arquitetura no intervalo $[0,1]$, pois o valor da avaliação de cada conexão também está inserido no intervalo $[0,1]$. Essa normalização se dá pelo motivo do problema proposto ser de

otimização multi-objetiva, e todos os objetivos conflitantes devem estar na mesma escala, pois facilita a atribuição dos graus de importância que são dados a cada objetivo.

$$A = \sum_{j=0}^m \frac{L_j}{m} \quad (10)$$

Nesta equação, A remete a avaliação estrutural de toda a arquitetura formada por implementações de componentes candidatas, onde é formada pelo somatório de contribuições quantitativas de cada conexão da instância arquitetural em avaliação. O número de conexões equivale a m , e L_j representa a avaliação de uma conexão (Figura 18).

Análogo à equação descrita anteriormente que representa a qualidade estrutural, tem-se também uma equação que representa a qualidade de todos os elementos da arquitetura, como vimos em seções anteriores, a qualidade dos elementos reflete na qualidade geral da arquitetura. Portanto, a qualidade geral da arquitetura em termos de seus elementos é representada pela equação abaixo.

$$C = \sum_{i=0}^n \frac{C_i}{n} \quad (11)$$

Nesta equação, C remete a avaliação qualitativa de todos os elementos da arquitetura formada por implementações de componentes candidatas, a qual é formada pelo somatório de contribuições quantitativas de cada elemento (componente) da instância arquitetural em avaliação. O número de implementações de componentes equivale a n , e C_i representa a avaliação de uma implementação de componente candidata (Equação 9).

Caso alguma especificação de componente presente na arquitetura do Projeto Arquitetural não possua implementações de componentes candidatas (a busca não retornou candidatos para àquela especificação), o componente e suas conexões serão anulados, ou seja, a avaliação arquitetural será realizada como tal especificação não estivesse presente no Projeto Arquitetural. A ausência de implementações candidatas para alguma especificação da arquitetura não compromete a computação da avaliação, evidenciando sua robustez.

Enfim, é apresentada a função de adequação (equação abaixo) do AG proposto por este trabalho, e ela agrupa os dois objetivos supracitados com seus pesos (w_1 e w_2) que serão estabelecidos por algum tomador de decisão.

$$\begin{aligned} F(x) &= w_1 C(x) + w_2 A(x) \\ \begin{cases} w_1 = \{a \in \mathbb{R}: 0 \leq a \leq 1\} \\ w_2 = \{a \in \mathbb{R}: 0 \leq a \leq 1\} \end{cases} \\ w_1 + w_2 &= 1 \end{aligned} \quad (12)$$

É importante frisar que as métricas utilizadas correspondem às funções de adequação do algoritmo genético proposto, pois na concepção dessas métricas foi planejado que elas seriam funções de adequações. Por esse motivo todas as métricas são normalizadas no intervalo $[0,1]$ para facilitar na combinação dos objetivos no AG multi-objetivo. E conforme [Harman09], métricas também são funções de adequação.

3.6. Considerações Finais

Neste capítulo, foi apresentada uma abordagem desenvolvida para selecionar componentes de software de maneira automatizada e guiada por métricas que avaliam os componentes candidatos no ponto de vista da estrutura arquitetural e das informações de reúso.

A etapa anterior do processo de seleção de componentes é o *Projeto Arquitetural*, onde é definida a arquitetura do sistema que será desenvolvido, e os artefatos em documentos X-ARM que descrevem as especificações de componentes dessa mesma arquitetura são fornecidos para etapa de *Seleção de Componentes* que irá recuperar implementações de componentes de software candidatas e irá escolher uma instância arquitetural com base nos artefatos da etapa anterior.

A recuperação é realizada através *Serviço de Repositório* juntamente com o *Serviço de Busca* que irão retornar artefatos X-ARM dos componentes candidatos, além de suas informações de reúso. Esses artefatos e suas informações de reúso serão avaliados por *Métricas* definidas com o propósito de avaliar componentes de software na perspectiva estrutural da arquitetura e do próprio componente (similaridade e informações de reúso).

Como o problema de seleção tratado neste trabalho possui complexidade exponencial, foi utilizada uma meta-heurística com o intuito de alcançar boas soluções e por consequência obter um sistema com qualidade, e com seu custo e tempo de integração reduzido. As funções de adequação dessa meta-heurística são as métricas definidas neste trabalho, assim, as soluções são guiadas em função das decisões e restrições definidas nas métricas. Após a meta-heurística apontar as melhores soluções alcançadas em função de tais métricas, a instância arquitetural escolhida é enviada para etapa de Montagem, onde as implementações de componentes serão integradas e formarão realmente um sistema concreto.

Capítulo 4

Avaliação Experimental

Neste capítulo será abordada a maneira como foi construído o ambiente para executar os experimentos do presente trabalho, para então efetuar a realização da validação do algoritmo de otimização construído. O ambiente gera e proporciona a massa de dados para que sejam processadas pelo algoritmo proposto. A execução de outros algoritmos como busca aleatória e por força bruta também serão realizadas com a intenção de compará-los com a implementação desenvolvida no presente trabalho. Essa comparação tem o objetivo de evidenciar que o algoritmo proposto possui a qualidade da solução próxima ou equiparável ao algoritmo de força bruta (com espaço de busca reduzido), e melhor desempenho que abordagens mais simples, demonstrando que o esforço empregado na concepção do algoritmo proposto resultou na melhoria qualitativa das soluções.

Após cada algoritmo apresentar suas soluções encontradas, são executados testes estatísticos como meio de provar que o algoritmo proposto possui desempenho melhor que a busca aleatória (abordagem mais simples). Também são apresentados os parâmetros para o algoritmo genético desenvolvido nesse trabalho, assim como o tempo de execução e o espaço de busca explorado.

Não foram realizadas comparações com algoritmos mais elaborados que a busca aleatória e até mesmo àqueles no estado da arte, pois a intenção do presente trabalho não é comparar, por exemplo, performance e/ou representação entre meta-heurísticas. Este trabalho demonstrou na seção 3.5.2 o motivo da escolha do algoritmo genético (adequabilidade do problema em relação à meta-heurística), então esta avaliação experimental tem a meta de evidenciar que o AG absorveu os detalhes e características do problema, e por isso, terá desempenho qualitativo melhor que a busca aleatória e terá soluções aproximadas à melhor solução possível (encontrada por força bruta).

4.1. Criação do Ambiente Experimental

Antes de dar início ao processo de validação do algoritmo proposto, deve-se ter um ambiente experimental propício que simule normalmente uma etapa de seleção de componentes automatizada por algoritmos genéticos. *A priori*, o término da etapa do Projeto Arquitetural fornece um diagrama que possui uma arquitetura formada por especificações de componentes, visto que, como a arquitetura será a base de referência para a criação do sistema em desenvolvimento, é necessário que ela seja convertida em uma estrutura de dados a qual seja passível de computação, para então automatizar o processo de seleção de componentes.

A estrutura de dados que melhor representa a arquitetura de software do trabalho proposto é um grafo, onde os vértices podem representar componentes de software e as arestas correspondem a conexões via interfaces. Esse grafo é direcionado, porque esse direcionamento representa uma conexão com uma interface provida e requerida. Quando um vértice *A* aponta para outro vértice *B* isso representa que existe uma conexão entre o componente *A* e o componente *B*, com o primeiro possuindo a interface requerida e o último a interface provida. Há também um vértice nulo, da maneira que ele possibilita representar as interfaces providas e requeridas que não realizam conexões. Por exemplo, se um componente apresentar uma interface requerida que não se conecta a nenhum outro componente, esse mesmo componente apontará para o vértice nulo visando representar uma interface requerida sem conexão. E se o vértice nulo apontar para algum componente, o mesmo receberá a representação de uma interface provida que não efetua conexão.

Cada aresta possui um identificador que é o mesmo de um artefato X-ARM, e isso tem o objetivo de mostrar que a aresta representa uma interface de software. Assim que o arquiteto de software mapeia o diagrama da arquitetura num grafo, haverá somente os vértices que representam os componentes e as arestas que são as conexões. Esse mapeamento é realizado por persistir a estrutura da arquitetura em uma matriz que corresponde ao grafo. A partir daí, um leitor XML acessa todos os artefatos que descrevem as interfaces representadas nas arestas e recupera as informações e as convertem para dados, então todas as informações desde as operações até a arquitetura como um todo estão disponíveis para serem passíveis de operações computacionais. Portanto, essa operação descrita logo acima obteve a representação da arquitetura em um grafo.

Agora que o mapeamento do diagrama para dados computacionais foi realizado, o próximo passo é também realizar o mapeamento dos componentes candidatos para dados e

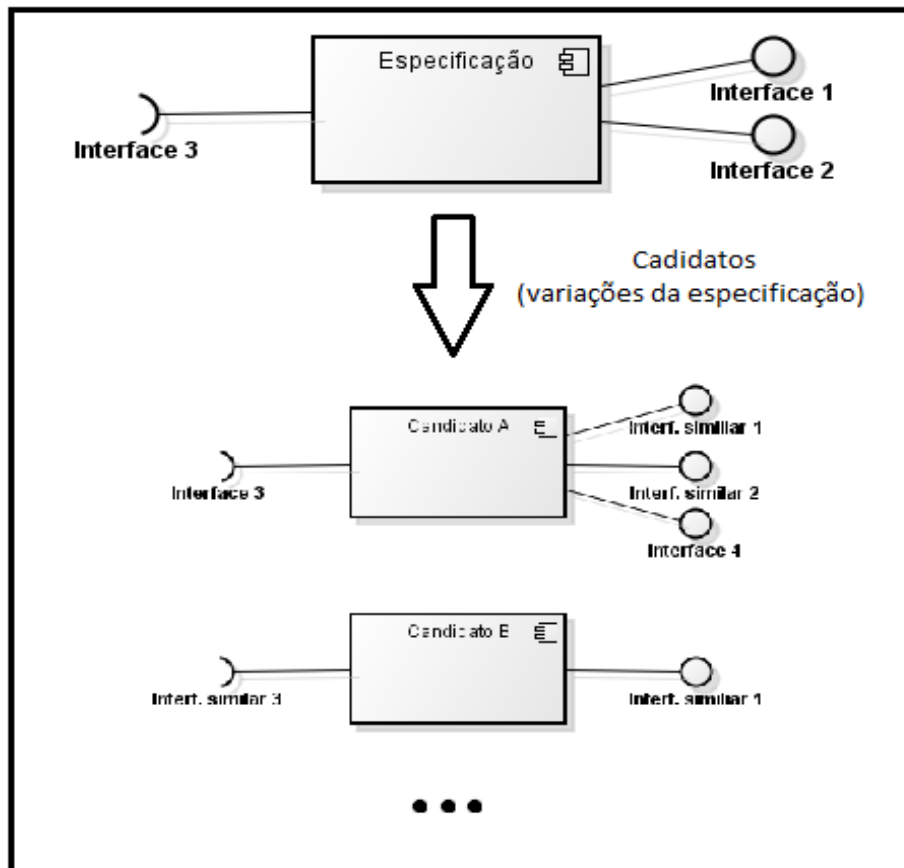
posteriormente reformulá-los para serem processados pelo algoritmo proposto. O arquiteto de software fornece descrições das especificações de componentes da arquitetura para uma ferramenta, e a mesma gera um conjunto de consultas com o propósito de buscar candidatos similares. Essas consultas podem ser processadas no Serviço do Repositório, para então serem recuperados os artefatos similares dos elementos da arquitetura (componentes e suas interfaces), e um leitor de arquivos XML irá mapear os artefatos (descritos no modelo X-ARM) em dados que representarão os componentes candidatos.

Para a elaboração do ambiente experimental não será necessário popular o repositório do *framework ComponentForge* com artefatos alusivos aos componentes candidatos para então executar consultas buscando tais artefatos candidatos (similares). O processo de busca dos artefatos similares não impacta na execução do algoritmo proposto. Portanto, abstrai-se desse passo e são gerados, via ferramenta, artefatos variantes das especificações de componentes da arquitetura (artefatos similares); e são considerados como se fossem provenientes da execução de busca no repositório. Apenas a arquitetura e os componentes candidatos são necessários para testar a meta-heurística.

Os componentes candidatos são gerados da seguinte forma: pega-se cada especificação de componente como referência e são criadas variações dela a nível de operações e interfaces. Exemplificando, um componente candidato faz uma cópia da especificação que ele tem como referência, posteriormente esse candidato terá modificações em cada interface (serão adicionadas novas operações e/ou retirada algumas existentes, porém a interface sempre terá pelo menos uma operação que corresponde a sua interface de referência), e também o candidato receberá novas interfaces e/ou perderá algumas interfaces. Esse processo cria um candidato referente a alguma especificação de componente da arquitetura, então ele é repetido n vezes, onde n é o número de componentes candidatos que se queira criar.

A Figura 25 ilustra como os componentes candidatos são gerados a partir de uma especificação pertencente à arquitetura do sistema em construção. O candidato *Componente A*

Figura 25 – Criação de componentes candidatos que são variações de alguma especificação.



dessa figura, possui uma interface provida a mais, e duas delas são similares às interfaces da especificação, contudo, a interface requerida foi idêntica. Já o candidato *Componente B* apresentou uma interface a menos, onde todas as interfaces desse candidato são similares as suas correspondentes na especificação. E assim por diante vão sendo geradas variações aleatórias da especificação até se obter um número n de candidatos.

No momento que todos os candidatos são criados, chega o momento do AG proposto ser iniciado. O primeiro passo do algoritmo é a criação da população, então os componentes candidatos são reformulados para os genes, que formarão cromossomos (soluções candidatas). O AG irá iterar suas etapas até que a condição de parada seja satisfeita. Enfim, nessa seção foi descrito como foi realizada a criação da massa de dados para os experimentos cuja meta é validar o trabalho proposto.

Para este experimento, as informações de reuso foram geradas aleatoriamente para os componentes candidatos, pois elas não dependem somente de tais candidatos para serem geradas, mas da relação deles com o repositório que estão incluídos. Como o ambiente experimental não é dependente de algum repositório específico, atribui-se valores randômicos no intervalo $[0,1]$ (faixa de valores possíveis para a métrica de informações de reuso) para os candidatos.

4.2. Avaliação

Após a concepção e implementação de uma meta-heurística é necessária a realização do processo de validação dessa abordagem algorítmica para dar confiança aos resultados produzido por ela. É comum utilizar testes comparativos nesse contexto de validação, como evidenciar o quão próximo os resultados da meta-heurística são em relação à solução ótima. Isso configura uma comparação entre os resultados do algoritmo em análise com os resultados de um algoritmo que retorna soluções ótimas. Como a maior motivação da elaboração de uma meta-heurística é a ausência de um algoritmo que encontre soluções ótimas em tempo polinomial, a comparação deve ser feita com um espaço de busca bem reduzido para que um algoritmo de força bruta encontre a solução ótima em um tempo viável. Se houvesse um algoritmo que encontrasse as soluções ótimas em tempo polinomial, não haveria necessidade de se criar uma meta-heurística, então como terá que ter a comparação entre a meta-heurística e o algoritmo de força bruta, o espaço de busca é reduzido.

Uma meta-heurística também deve ser validada pela comparação com alternativas mais simples, visando justificar o esforço empregado na elaboração dessa meta-heurística, pois se a mesma apresentar resultados equiparáveis a de uma alternativa mais simples, deveria utilizar esta ao invés daquela. Segundo [Ali09], a comparação com abordagens mais simples é uma necessidade quando se é proposta uma nova meta-heurística que é endereçada a tratar algum problema da Engenharia de Software. Como o trabalho proposto utiliza algoritmos genéticos, onde o mesmo é considerado um algoritmo aleatório guiado pela abordagem da seleção natural, então comumente a alternativa mais simples que é comparada com o AG será um algoritmo de busca aleatória. A intenção da escolha da busca aleatória para comparação se deve pelo motivo de mostrar que o AG implementado é diferenciado de uma mera busca aleatória, pois as informações do problema inseridas no AG devem guiar a busca para encontrar boas soluções.

Ratificando, a intenção desse processo de validação pretende mostrar que a reformulação do problema da Engenharia de Software enfrentado nesse trabalho em um AG foi executada de forma consistente, em que as soluções resultantes do AG proposto devem ser próximas ou iguais da solução ótima e também devem ser melhores que uma simples busca aleatória. A Figura 26 mostra a comparação performática da qualidade das soluções entre essas abordagens comentadas.

Figura 26 – Comparação do desempenho qualitativo das soluções.

$$\text{Busca Aleatória} < \text{Algoritmo Genético Proposto} \leq \text{Busca por Força Bruta}$$

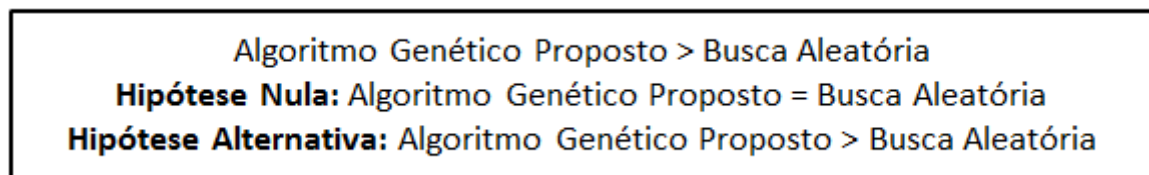
Para comparar abordagens algorítmicas em um determinado domínio de aplicação e apontar se uma é melhor que a outra ou se são equivalentes é necessária a utilização de testes estatísticos, pois eles avaliam se há evidência nos dados o bastante para afirmar que um conjunto de amostras resultantes de um algoritmo é superior ou inferior ao conjunto de amostras de outro algoritmo que se queira comparar. Um teste estatístico, também chamado de teste de hipótese, utiliza conceitos estatísticos para aceitar ou rejeitar uma hipótese nula. Tal hipótese nula geralmente condiz com a asserção de que não há diferenças entre dois algoritmos, e o teste estatístico indicará se deve aceitar ou rejeitar tal hipótese. Nesse contexto estatístico tem-se o nível de significância α , onde ele remete ao erro cuja definição afirma que a chance de que a diferença entre os algoritmos tenha ocorrido devido ao acaso é menor que uma probabilidade definida. Para expressar melhor, o nível α remete ao risco que se está disposto a aceitar para afirmar que a hipótese nula não é verdadeira.

Na prática dos testes estatísticos em diversos domínios de aplicação como até em ciências sociais, tipicamente o valor de α é convencionado em 0,05. Assim, equivale a dizer que o valor de α estabelecido em 0,05 defende que a aceitação da hipótese nula possui uma probabilidade de 5% de não ser verdadeira. Esse valor α pode representar uma região na curva de distribuição normal chamada de *região de rejeição*. E se algum parâmetro que defende pela hipótese nula estiver inserido nessa região, então a chance desse parâmetro ser verdadeiro é pequena demais para ser aceita, então a hipótese nula é rejeitada em favor de outra alternativa. Nesses testes tem-se o *valor-p* que corresponde à probabilidade de se rejeitar a hipótese nula quando ela for verdadeira. Basicamente, se o *valor-p* for menor-igual que o nível α , a hipótese nula será rejeitada.

A distribuição da curva normal possui duas caudas em suas extremidades, e os valores presentes nessas caudas são os mais raros dessa curva. A região de rejeição situa-se nessas pontas da curva normal e o tamanho dessa região na curva vai depender do nível α escolhido. O teste estatístico com uma curva que apresenta duas caudas possui como hipótese alternativa o fato de que os resultados de dois algoritmos não são equivalentes. Ao invés disso, a meta é evidenciar estatisticamente que um algoritmo é melhor que outro. Assim, o teste estatístico para o problema proposto não tem o objetivo de descobrir o nível qualitativo dos resultados dos algoritmos em análise, mas deve-se testar se um determinado algoritmo é sempre superior a algum outro, portanto a cauda de rejeição é situada apenas na cauda superior da curva.

A intenção desse teste é demonstrar que o algoritmo proposto sobrepõe a qualidade do algoritmo com uma abordagem mais simples, pois o esforço em construir um algoritmo mais complexo deve ser justificado por soluções com qualidade superior. Portanto a intenção é mostrar que a hipótese alternativa, ilustrada na Figura 27, é a hipótese aceita pelo teste.

Figura 27 – Hipóteses nula e alternativa do teste estatístico.



Sabemos que um algoritmo de força bruta sempre encontrará a solução ótima, e como a meta-heurística implementada neste trabalho se propõe a encontrar soluções próximas da ótima ou até mesmo a ótima, o teste estatístico nessa situação não é necessário, pois já se tem o conhecimento que os resultados provenientes do algoritmo por força bruta sempre irão superar os resultados de algum outro algoritmo. Dessa forma, apenas será mostrado o quão próximo das soluções ótimas os resultados do algoritmo proposto irão chegar.

O trabalho de [Arcuri11] oferece uma revisão sistemática dos trabalhos que utilizam algoritmos de cunho aleatório, como as meta-heurísticas, e os aplicam em problemas da Engenharia de Software. Os autores desse trabalho apontam que apenas uma minoria dos trabalhos analisados possui seus experimentos fundamentados em testes estatísticos. Assim, a validade da grande maioria desses trabalhos pode se tornar questionável. Nos trabalhos que utilizam testes estatísticos, há duas abordagens utilizadas: o teste paramétrico *t-test* e o teste não-paramétrico *Mann-Whitney U-test*. A abordagem utilizada por este trabalho proposto é o *U-test*, pois pelo ponto de vista prático [Arcuri11][Leech02] a grande motivação desse teste

não-paramétrico se deve em função do *t-test* não ser apropriado para algumas análises referentes ao contexto de algoritmos de cunho estatístico para Engenharia de Software.

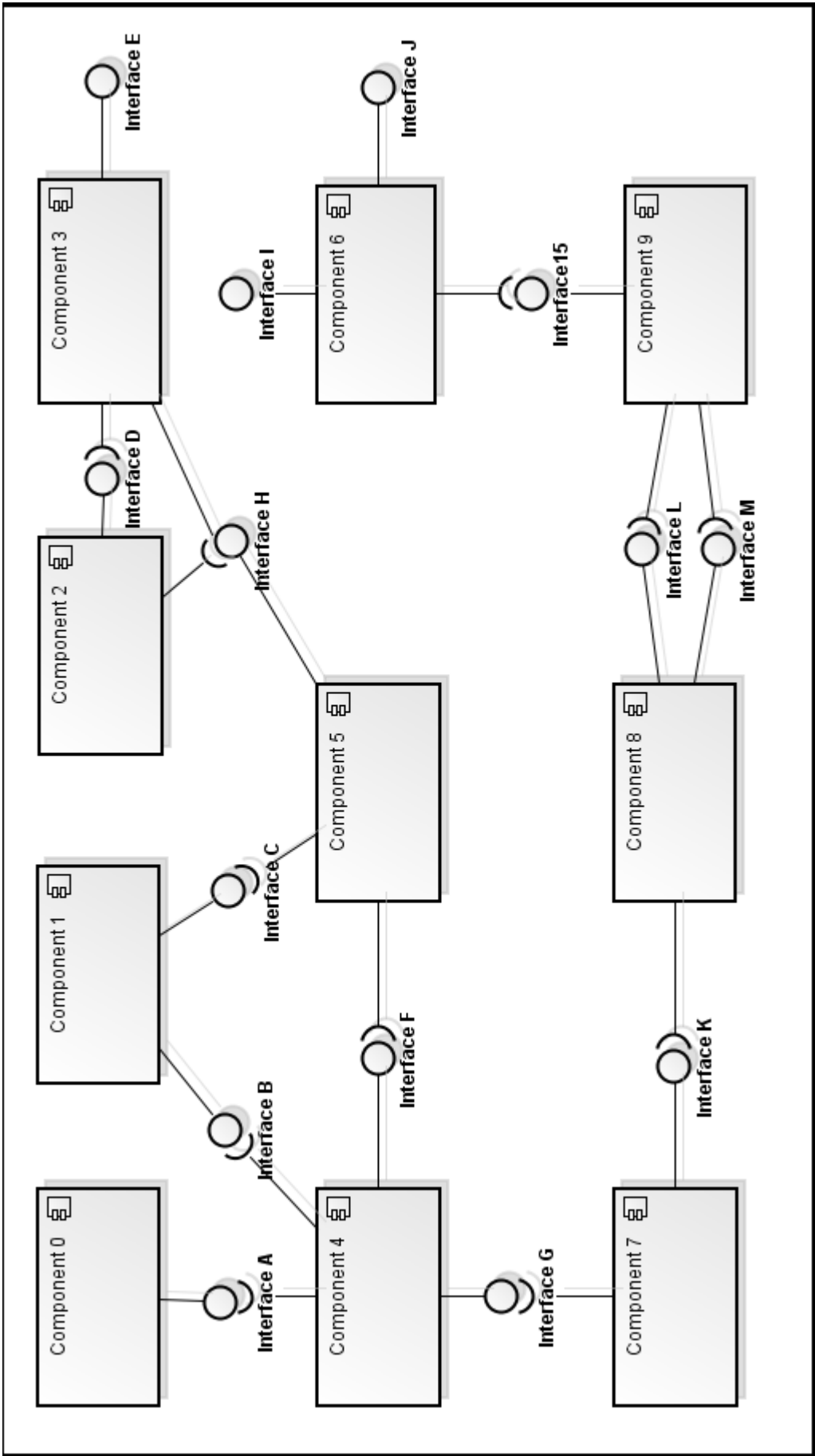
O teste estatístico não-paramétrico *Mann-Whitney U-test* é adequado para comparar duas distribuições de uma variável aleatória que seja ordinal e medida por amostras independentes. Esse tipo de teste estatístico é uma alternativa ao *t-test* de *Student* quando as amostras são independentes, como no caso do trabalho proposto, e os parâmetros (i.e. média e desvio padrão da população) desse teste não são válidos e então não é desejável garantir a robustez do teste quando seus pressupostos são violados. Isso geralmente acontece quando as amostras possuem pequena dimensão ou são muito diferentes, distribuições muito enviesadas, e as variâncias são muito heterogêneas [Marôco11].

4.3. Realização dos Experimentos

Esta seção tem a intenção de apresentar a forma como os experimentos são realizados com o suporte de testes estatísticos visando obter uma validação coerente. A princípio é necessário ter uma arquitetura-modelo para concretizar o ambiente experimental, dessa maneira, foi criado um diagrama UML para representar a arquitetura-modelo a qual os experimentos irão utilizar. A Figura 28 ilustra uma arquitetura formada por 10 especificações de componentes, donde são contabilizadas 12 conexões entre os componentes.

Para cada especificação de componente da arquitetura-modelo foram gerados 30 candidatos, que são implementações de componentes candidatas. Como a arquitetura possui 10 componentes, a quantidade de combinações possíveis é o número de candidatos elevado ao número de especificações na arquitetura, onde nesse experimento tem-se 30^{10} combinações possíveis. Portanto, buscar a solução ótima com esse número de combinações é bastante proibitivo, acarretando na impossibilidade de obter a solução por intermédio de um algoritmo de força bruta (exponencial) em um tempo realístico.

Figura 28 – Arquitetura-modelo para a avaliação experimental.



O algoritmo proposto por este trabalho é multi-objetivo, mais especificamente, o mesmo busca soluções satisfatórias sob a orientação de dois objetivos: avaliação da estrutura arquitetural e do componente em si. Nesse caso, os experimentos serão efetuados nas duas perspectivas ao invés de avaliar a função multi-objetiva que une ambas, pois seria complicado enxergar qual objetivo estaria apresentando ruído.

Como o algoritmo multi-objetivo apenas soma de forma ponderada os valores produzidos pelos dois objetivos, a computação de um objetivo não interfere na computação do outro. Ou seja, a execução da função de adequação de um objetivo não espera pela execução nem altera os valores do outro, e vice-versa. Então, serão realizados experimentos para analisar cada objetivo, tornando desnecessária a análise dos valores resultantes da junção dos objetivos, pois basta observar que ambos os objetivos possuem suas funções de adequação coerentes no algoritmo genético [Brownlee12][Hamdy11]. Quando os objetivos forem ponderados, cada um será consistente e contribuirá para seleção de componentes na proporção que foi definida. Além de que é muito complexo analisar uma solução multi-objetiva e detectar se uma característica da mesma foi resultante direto de um objetivo e/ou outro; chega até ser empírica tal análise.

É inegável que na ponderação os objetivos competirão entre si pela importância em guiar essa seleção, todavia esta decisão de atribuir pesos cabe ao arquiteto de software ou a um tomador de decisão. Como os dois objetivos abordados neste trabalho são essenciais para avaliar uma arquitetura, é sugerido que ambos por padrão possuam o mesmo grau de importância para a seleção de componentes, assim cada objetivo possui peso 0,5. Porém, dependendo da necessidade do arquiteto de software, esses pesos são ajustados.

O propósito da avaliação de cada objetivo tem a meta de mostrar que a reformulação do problema atacado pelo trabalho proposto para a meta-heurística foi executada de maneira consistente e que o esforço para realizar tal reformulação deve ser justificado numa melhor qualidade de seus resultados comparados aos de abordagens mais simples. Além do mais, tem o propósito de mostrar que o algoritmo proposto obtém soluções próximas da ótima ou até mesmo a ótima. A abordagem mais simplória que será comparada com o AG proposto é uma busca aleatória. A intenção dessa comparação implica em evidenciar estatisticamente que o AG produz resultados melhores.

Um dado importante a se considerar é o número de amostras que deve ser obtida para que o teste estatístico apresente um resultado com boa propriedade. De acordo com

[Arcuri11], 1000 amostras é um bom valor para realizar o teste estatístico nesse domínio de aplicação experimental. Assim, a confiabilidade dos testes realizados nesse trabalho é melhor que em outros trabalhos que utilizam meta-heurísticas na Engenharia de Software, porque eles na maioria das vezes se limitam a apenas 100 amostras.

Para se obter as 1000 amostras, a busca aleatória e o algoritmo genético são executados 1000 vezes e consequentemente cada um apresentará 1000 soluções. Então, têm-se duas populações com 1000 amostras, as quais serão analisadas estatisticamente visando mostrar que o AG sobrepõe a abordagem mais simples.

Para mostrar que as soluções do AG são próximas à solução ótima, a alternativa mais comum e prática é reduzir o espaço de busca para que o algoritmo de força bruta obtenha sua solução ótima em tempo tolerável, para então, comparar com as soluções do AG proposto. O espaço de busca da arquitetura-modelo é 30^{10} , onde o tempo de execução é proibitivo quando se executa o algoritmo por força bruta em um espaço dessa magnitude. Uma maneira trivial de reduzir o espaço de busca é diminuindo o número de candidatos e/ou o número de especificações de componentes na arquitetura. Entretanto, foi pensado em uma forma de reduzir o espaço de busca somente para o algoritmo de força bruta, resultando na vantagem de conhecer a solução ótima em um tempo hábil e analisar o processamento do AG em um espaço de busca mais realístico com a prática atual de seleção de componentes.

A ideia é criar o espaço de busca da seguinte maneira: como há 10 conjuntos de componentes candidatos (um para cada especificação de componente da arquitetura), em 5 deles, pega-se um candidato aleatoriamente e troca o mesmo por uma implementação de componente idêntica à especificação de componente que aquele conjunto corresponde. Esses componentes idênticos são dominantes quando são comparados com componentes similares, pois as métricas do presente trabalho são orientadas pela arquitetura elaborada no Projeto Arquitetural. Dessa forma, qualquer candidato se conectaria melhor com um componente idêntico (por ser perfeito ao que fora especificado na arquitetura) do que qualquer outro componente similar à especificação.

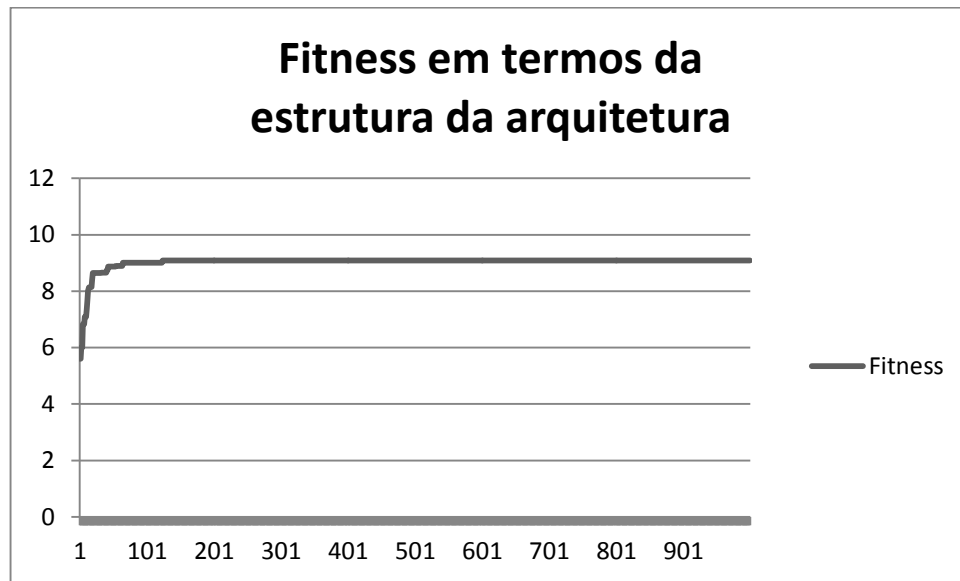
Como já se sabe que a solução ótima possuirá os 5 componentes candidatos que são idênticos as suas respectivas especificações, então apenas os outros 5 são desconhecidos, assim o espaço de busca para o algoritmo de força bruta é reduzido de 30^{10} para 30^5 combinações possíveis, tornando viável o tempo de execução para esse panorama experimental. O Algoritmo de força bruta irá iterar em 30^5 combinações, pois antes de sua

inicialização, os 5 componentes idênticos são fixados em todas as permutações. Já para a busca aleatória e o AG, tais componentes não são fixados, então outros componentes pertencentes aos mesmos conjuntos que os componentes idênticos também são testados. Portanto, o espaço de busca para essas duas abordagens algorítmicas é 30^{10} . Portanto, a meta de reduzir apenas o espaço de busca para o algoritmo de força bruta foi alcançado.

É importante salientar a execução do algoritmo de força bruta seria desnecessária se ao invés de 5 componentes idênticos, fossem utilizados os 10, para então verificar se o AG encontraria a solução ótima, que é formada por esses 10 componentes idênticos as suas especificações. Todavia, pelo motivo dos componentes idênticos serem perfeitos, consequentemente dominantes, esse fator contribui para a convergência genética do AG porque se um cromossomo trocar um gene que possuía um componente idêntico por qualquer componente similar, a *fitness* do cromossomo é reduzida. Assim, possuir conjuntos de candidatos sem a presença de componentes iguais aos da arquitetura torna o experimento mais realístico, pois nenhum componente similar domina todos os seus concorrentes para todas as combinações de conexões possíveis com outros componentes. Portanto, torna o experimento mais realístico.

4.3.1. Avaliação em Termos da Estrutura da Arquitetura

A curva que representa a melhor solução encontrada com o passar do tempo de execução do algoritmo genético é ilustrada na Figura 29. O eixo vertical corresponde a *fitness* da melhor solução encontrada até o momento e o eixo horizontal representa o número de iterações. A escala do eixo vertical vai até o número 12, pois existem 12 conexões presentes na arquitetura-modelo, já no eixo horizontal a escala numérica vai até o número 1000, já que a condição de parada do AG proposta é quando se atinge 1000 iterações. Em todas as 1000 iterações ocorreu este comportamento de crescimento, pois o algoritmo genético proposto utiliza a abordagem elitista, implicando, em potencial, da curva sempre crescer até atingir a solução ótima. Nesta implementação, todas as amostras obtiveram a solução ótima, e a mesma foi alcançada entre o intervalo de 62-214 iterações do AG.

Figura 29 – *Fitness* da estrutura da arquitetura

Juntamente com o algoritmo proposto, foram executadas a busca aleatória e a força bruta para serem os meios de comparação. A primeira por ser uma abordagem mais simples e a última por encontrar a solução ótima em um espaço de busca reduzido. A Tabela 4 demonstra o desempenho de apenas uma execução de cada abordagem algorítmica. O algoritmo genético desse experimento possui uma população de 100 indivíduos, uma taxa de mutação 20%, e todos os cromossomos que não fazem parte do subconjunto elitista são recombinados. O número de indivíduos elitistas na população é 4, assim os quatro melhores integrantes da população são preservados para não se perder soluções boas que foram obtidas outrora. A *fitness* da busca aleatória atingiu aproximadamente 68% da solução ótima e seu tempo de execução foi levemente menor que o algoritmo. O número máximo do espaço de busca explorado foi de $1 \cdot 10^5$ (caso não ocorressem repetições), esse número foi definido para ser equiparável com o espaço de busca explorado pelo algoritmo genético, pois o mesmo possui uma população de 100 indivíduos e o número de 1000 iterações, resultando em $100 \cdot 1000 = 10^5$. O espaço de busca do algoritmo por força bruta é de $2,43 \cdot 10^7$, que é o mesmo que 30^5 , pois existem 30 candidatos para 5 especificações de componentes que não se conhece o melhor componente. Os outros 5 componentes possuem entre seus candidatos uma implementação idêntica à especificação correspondente, então não precisa procurar por outros componentes, pois eles são “perfeitos”, ou seja, se for avaliar cada conexão possível desse componente, ela vai ser melhor ou igual a qualquer outro componente candidato. Na arquitetura do sistema do experimento há 12 conexões entre os componentes, assim se todos os conjuntos candidatos tivessem um componente idêntico ao da especificação, a solução

ótima seria 12. Entretanto, esse valor posteriormente é normalizado para a escala $[0,1]$, pois o problema abordado nesse trabalho é multi-objetivo, e todos os objetivos devem possuir uma escala em comum para serem combinados e gerarem uma *fitness* geral.

Tabela 4 – Comparação quantitativa entre as abordagens algorítmicas investigadas neste trabalho.

	<i>Fitness</i>	Tempo de Execução	Espaço de busca
Busca Aleatória	6,274	2,095 segundos	$1 \cdot 10^5$
Algoritmo Genético	9,207	2,317 segundos	$1 \cdot 10^5$
Força Bruta	9,207	300,830 segundos	$2,43 \cdot 10^7$

Como o algoritmo genético e a busca aleatória foram executados 1000 vezes foram gerados esse mesmo número de amostras visando a utilização do teste estatístico de *Mann-Whitney U-test* para validar que a abordagem algorítmica proposta é de fato melhor que uma abordagem mais simplória. Em todas as execuções do algoritmo genético foi alcançada a solução ótima. De forma contrária, a busca aleatória não encontrou a solução ótima, onde a amostra que mais chegou próxima atingiu aproximadamente 74% da solução ótima. O teste foi realizado e nessa situação obteve o *valor-p* igual a 0 em virtude de todos os valores da amostra do algoritmo genético serem maiores que as da busca aleatória, e como 0 é um valor menor que o nível de significância α que foi estabelecido (0,05), a hipótese nula foi rejeitada. Já que os testes estatísticos modelam seus experimentos na curva da distribuição normal, o teste realizado nesse trabalho apenas se concentrou na calda direita da distribuição para avaliar se a hipótese alternativa *Algoritmo Genético Proposto* > *Busca Aleatória* é escolhida em detrimento da hipótese nula. Então, foi provado estatisticamente que o algoritmo proposto é melhor que a busca aleatória em termos da qualidade da solução encontrada. É importante observar que o tempo do AG é equiparável ao da busca aleatória. E também, a solução ótima sempre foi encontrada.

4.3.2. Avaliação em Termos dos Componentes

O outro objetivo do algoritmo genético remete à análise qualitativa dos componentes pertencentes à arquitetura do sistema em desenvolvimento. Nessa perspectiva de avaliação dos componentes, nenhum componente depende de outro pra ser avaliado, mas somente dos seus atributos qualitativos. Assim, para cada grupo de candidatos correspondente a alguma

especificação da arquitetura, basta avaliar um por um com a métrica (Seção 3.4 – Equação 9) estabelecida no capítulo anterior, e escolher o que obtiver a melhor *fitness*. Portanto o algoritmo para encontrar a solução ótima desse objetivo por força bruta é linear.

Apesar da complexidade desse segundo objetivo ser linear, quando ele é combinado com o outro objetivo de complexidade exponencial para avaliar uma instância arquitetural candidata, a complexidade geral torna-se exponencial, pois os objetivos são conflitantes (ambos otimizam os mesmos componentes em perspectivas distintas) e a avaliação geral depende dos dois objetivos para avaliar a arquitetura como um todo.

Consoante o objetivo estrutural da avaliação da arquitetura para selecionar a melhor instância arquitetural, o comportamento da curva do objetivo referente à entidade do componente é o mesmo. A curva dessa execução do AG com a função de adequação em termos dos componentes candidatos é ilustrada na Figura 30. Dessa forma, quando ambos os objetivos são combinados para realizar a avaliação geral, o tempo de execução e a qualidade das solução encontradas são muito similares quando se comparam os objetivos em separado.

Figura 30 – *Fitness* correspondente a avaliação na perspectiva dos componentes candidatos.



Esse segundo objetivo foi testado para mostrar também a boa atuação do AG nessa outra perspectiva. Foram utilizados os mesmos parâmetros (tamanho da população, número de iterações, taxa de mutação) que os experimentos para o objetivo da avaliação estrutural. Nenhum dos dois objetivos é gargalo de desempenho de si mesmo no ponto de vista do algoritmo genético. A execução experimental da busca aleatória também possui as mesmas características comportamentais em comparação com o outro objetivo. Ao contrário, o algoritmo por força bruta possui complexidades diferentes entre os dois objetivos. No que

trata da avaliação estrutural a complexidade é exponencial, já o objetivo que avalia os componentes em si, possui complexidade linear. Todavia, quando são combinados os dois objetivos, o de maior complexidade engloba o de menor, resultando em um problema geral com complexidade exponencial, onde a busca pela solução ótima com algoritmos determinísticos é inviável pelo espaço de busca ser complexo e gigantesco.

O teste estatístico entre o AG e a busca aleatória nessa perspectiva também mostrou que a hipótese alternativa foi escolhida em detrimento da hipótese nula. Isso já era previsto, pois o comportamento da curva do objetivo que avalia os componentes foi semelhante ao teste anterior, assim como a busca aleatória, pois o espaço de busca foi o mesmo.

Os dados necessários para compor as variáveis da equação da fitness foram gerados aleatoriamente. Matematicamente falando, objetivos como similaridade e informações de reúso poderiam ser unidos ou trocados por qualquer outro objetivo qualitativo, pois não alteram a forma de execução do algoritmo – desde que os dados necessários para compor as variáveis sejam pré-calculados antes do algoritmo iniciar. Entretanto, o presente trabalho ressalta a importância dos objetivos, similaridade e informações de reúso, compondo o objetivo mais geral que refere-se em termos do componente, porque critérios como funcionalidade, atributos de qualidade e informações de mercado são englobados direta e indiretamente. Mensurar precisamente, por exemplo, o quanto as informações de reúso refletem esses critérios e seus subcritérios está além do escopo deste trabalho, visto que o campo de análise dessa esfera é empírico.

4.3.3. Considerações Referentes aos Experimentos

As avaliações dos experimentos para cada objetivo mostraram que a abordagem proposta encontrou a solução ótima em ambos os casos. Dessa forma, tem-se a percepção de que o algoritmo genético proposto é guiado de forma consistente por suas funções de adequação e também evoluem suas soluções por meio da seleção e dos operadores genéticos. Contudo, essas avaliações demonstraram apenas uma excelente reformulação do problema em uma meta-heurística. Como as métricas de software mapeadas em funções de adequações direcionam ao algoritmo genético proposto, é interessante analisar se a solução ótima realmente é melhor que outras soluções com *fitness* inferior. Ou seja, observar se uma arquitetura com *fitness* maior que outra realmente é melhor intuitivamente no ponto de vista da Engenharia de Software.

Na perspectiva do objetivo da avaliação estrutural da arquitetura, foram observadas 3 soluções: a ótima e as duas subsequentes com *fitness* ligeiramente inferior. Houve comparação entre as conexões dessas soluções, e foi percebido que quanto maior a *fitness*, menor a degradação dos problemas relacionados à integração entre os componentes candidatos. Assim, à medida que uma solução piora sua *fitness*, o número de *mismatches* detectados vão aumentando, portanto a consistência da métrica é observada de maneira intuitiva. Já para o objetivo em termos dos componentes candidatos a metodologia de comparação foi análoga à outra perspectiva. Ou seja, quanto menor a discrepância funcional entre as especificações e os candidatos, intuitivamente é observado o crescimento do valor da *fitness* (assumindo que todos possuam os mesmos valores para informações de reuso). Já para analisar as informações de reuso é trivial, pois ocorrer um mapeamento direto de valores (grau de satisfação e número de *downloads*) para outros valores (*fitness*). Quanto maior o valor das informações de reuso melhor será a *fitness*.

No âmbito da Engenharia de Software, quanto mais próximo a solução estiver da ótima, os ajustes necessários nos componentes selecionados serão mínimos e podem ser considerados pontuais – o que ajuda no tratamento do problema. E quanto mais distante a solução estiver da ótima, a complexidade nos ajustes na configuração arquitetural aumentará exponencialmente, pois deve-se observar a arquitetura como um todo (o problema deixa de ser pontual).

Capítulo 5

Trabalhos Relacionados

Este capítulo tem como objetivo apresentar alguns trabalhos que possuem relação direta com o presente trabalho, evidenciando os pontos positivos e negativos de cada um deles e, principalmente, mostrando onde os trabalhos relacionados diferem deste trabalho.

Foram escolhidos trabalhos de seleção automatizada de componentes que lidam com técnicas de otimização para sua computação. A busca por esses trabalhos foi realizada dentro de um repositório de publicações [SBSE13] de uma comunidade acadêmica que cataloga todos os trabalhos da Engenharia de Software Baseada em Busca, ou seja, trabalhos referentes a problemas da Engenharia de Software que são tratados por algoritmos de otimização, pois a busca é complexa e gigantesca. Dessa forma, foram filtrados apenas trabalhos que correspondem ao problema de seleção de componentes de software.

5.1. Algoritmos de Aproximação para o Problema da Seleção de Componentes de Software

O trabalho *Approximation Algorithms for Software Component Selection Problem*, de [Haghpanah07], declara a necessidade de projetar algoritmos eficientes para automatizar a seleção de componentes de software para organizações que utilizem DBC como modo de produção. Este processo NP-Completo de seleção tem como objetivo escolher um conjunto de componentes com custo mínimo para satisfazer um conjunto de objetivos como, por exemplo, funcionalidades. Devido à alta ordem de complexidade computacional, este trabalho investiga soluções aproximadas para tornar o problema da seleção de componentes praticável.

Este trabalho declara o problema de seleção da seguinte forma: a cada componente é atribuído um conjunto de funcionalidades que ele satisfaz. Os componentes podem possuir mais de uma funcionalidade e vice-versa. A cada componente é atribuído um custo, que é o custo total para adquiri-lo. E caso tal componente seja dependente de outros, o custo do componente candidato à seleção é a soma do seu custo juntamente com o de todos os outros que ele depende. O custo total de um conjunto de componentes selecionados é a soma dos

custos de todos os membros. Assumindo que cada requisito possa ser satisfeito por pelo menos um componente. A meta é escolher um conjunto com o menor custo possível entre os candidatos.

Os algoritmos utilizados para o problema de seleção descrito acima são Algoritmos Genéticos e Abordagem Gulosa visando obter soluções próximas da ótima. Abaixo segue a descrição da aplicação de ambos:

- **Algoritmos Genéticos:** são meta-heurísticas inspiradas em sistemas biológicos que melhoram a qualidade de soluções através de evolução [Holland75]. Vide a seção 2.4.
- **Abordagem Gulosa:** esta abordagem foi escolhida porque é um dos melhores algoritmos de aproximação em tempo polinomial para problemas de coberturas de conjuntos, que são problemas NP-Completo relacionados com a seleção de componentes. No decorrer da computação para obter uma solução, cada etapa é dirigida pela escolha que parece ser a melhor no momento. Neste caminho, um componente que custa 1 (uma) unidade e satisfaz 1 (um) requisito, é preferível a outro que satisfaz 2 (dois) requisitos, porém custam 100 (cem) unidades.

Vários experimentos foram realizados utilizando conjuntos de dados (representações de componentes) para comparar a performance de diferentes abordagens. Além das abordagens algorítmicas supracitadas, foi reunido um time de cinco especialistas para selecionar um conjunto de componentes que satisfaçam todos os requisitos propostos com o mínimo de custo possível. O número de requisitos era 16 e o número de componentes era de 40 a 60, cada requisito possuía um número de 1 a 6 candidatos. Os resultados apontaram que as abordagens algorítmicas obtiveram resultados melhores que o esforço humano, e a abordagem genética mostrou-se superior à gulosa.

Este trabalho apenas avalia a configuração arquitetural (construída por componentes selecionados) sob a perspectiva do custo de cada componente. Portanto, diferentemente da abordagem aqui proposta, não possui métricas que avaliam os atributos de qualidades dos componentes, nem métricas estruturais que avaliariam o custo necessário para adaptar componentes visando integrá-los entre si.

5.2. Uma Abordagem Evolucionária Baseada em Métricas para o Problema de Seleção de Componentes

Andreaa Vescan propôs em 2009 uma abordagem evolucionária para o problema de seleção de componentes (*A Metrics-based Evolutionary Approach for the Component Selection Problem*) [Vescan08]. Este trabalho considera que para obter a meta de selecionar a melhor configuração arquitetural é preciso avaliar os componentes, assim métricas de software são muito úteis no sentido de quantificar atributos considerados importantes para sistemas que serão construídos.

Este trabalho propõe uma abordagem de computação evolucionária de seleção que leva em consideração alguns atributos de qualidades dos componentes que serão avaliados por métricas. O problema é formulado como multi-objetivo, pois considera diferentes métricas.

Nesta pesquisa foram considerados os seguintes atributos para avaliar sistemas: custo e funcionalidade. E as métricas destes atributos de qualidades são descritas a seguir.

- **Métrica de custo:** o custo de um componente é definido pelo custo total para adquiri-lo e adaptá-lo. É idealizado encontrar uma solução que minimize o custo total do sistema.
- **Métrica de funcionalidade:** esta métrica representa para um componente a taxa entre o número de serviços requeridos pelo sistema que são providos por esse componente e o número total de serviços requeridos pelo sistema. Um componente com a métrica próxima de 1 (um) oferece quase todos os serviços para o sistema. Este trabalho trata o termo serviço como uma funcionalidade de software (definida por requisitos funcionais).

De acordo com métrica de funcionalidade deste trabalho [Vescan08], é ignorado o fato dos componentes possuírem funcionalidades extras (que não foram especificadas no modelo arquitetural). Isso demonstra que um sistema repleto de componentes com vários serviços extras sem serem utilizados terá boa funcionalidade. Essa característica desvia do contexto funcional da arquitetura produzida no Projeto Arquitetural, pois esta métrica pode guiar a seleção de componentes em escolher componentes com vários serviços extras, que não seriam utilizados porque não foram previstos, mas possuiriam boa funcionalidade. Serviços adicionais ao que foi especificado na arquitetura aumentam a complexidade do sistema, necessitam de mais recursos computacionais, além de requerem serviços adicionais para poderem prover tais serviços extras. Ao contrário deste trabalho, a abordagem de *Seleção*

Automatizada de Componentes de Software Orientada por Métricas Estruturais e Informações de Reúso proposta nesta pesquisa busca por componentes que se atém a estrutura estabelecida da arquitetura do sistema.

A otimização do problema é realizada com Algoritmos Genéticos e cada métrica estabelecida neste trabalho idealmente possui sua função de adequação, por se tratar de um problema multi-objetivo. Há várias abordagens para tratar problemas deste tipo, tendo o método da soma ponderada como a abordagem escolhida por este trabalho.

O problema de seleção, a representação das soluções e os operadores de manipulação são análogos ao trabalho discutido na seção anterior. Os experimentos computados mostraram que houve um número pequeno de diferentes soluções, porém apresentaram resultados satisfatórios. Populações com 50-100 amostras obtiveram resultados melhores que populações com 10-20 amostras. Pela baixa magnitude do número de amostras, este fato revela o motivo da ocorrência de resultados com pouca variabilidade de soluções. As métricas levadas em consideração foram apenas o custo e a funcionalidade.

5.3. Melhoria do Processo de Seleção de Componentes Usando Algoritmos Genéticos para Desenvolvimento Baseado em Componentes

O trabalho produzido por Vijayalakshmi, Ramaraj e Amuthakkannan (*Improvement of Component Selection Process Using Genetic Algorithm for Component-Based Software Development*) [Vijayalakshmi08], consiste em uma abordagem de seleção de componentes utilizando algoritmo genético para encontrar a melhor combinação de componentes diante de requisitos funcionais e não-funcionais. Esta abordagem sistematicamente avalia os componentes sob o critério de seleção baseado em funcionalidade, custo, complexidade e confiabilidade.

A função de adequação é um mecanismo usado para determinar o nível de confiança das soluções otimizadas de algum problema. Quanto maior o escore que uma solução obtiver, mais adequada ela será. Um componente que atente a todos os requisitos e avaliado como o melhor sob o critério de otimização, é considerado ótimo. O componente avaliado é por multi-objetivos, onde cada objetivo contribui para o escore final do componente. E a configuração arquitetural é avaliada pela soma dos escores de cada componente pertencente a ela.

A meta deste trabalho é encontrar o melhor componente possível, em tempo viável, para cada especificação do projeto arquitetural. Então, as métricas medem o quão um componente candidato atende à especificação desejada pelo arquiteto de software.

A métrica de funcionalidade deste trabalho remete à quantidade de serviços atendidos pelos componentes. Quanto mais serviços requeridos pela especificação de componente o candidato atender, melhor é sua funcionalidade no sistema. A métrica de complexidade é calculada baseada no acoplamento entre dois componentes. Dois componentes são acoplados se existe uma conexão entre eles, em que tal conexão significa uma solicitação de serviço. Acoplamento excessivo é uma qualidade negativa para práticas de design, pois eleva a complexidade de sistema, dificultando sua manutenção e atualização [Tian95]. A métrica de confiabilidade está relacionada com probabilidade de uma operação estar livre de falhas em um especificado ambiente por um tempo determinado. Para avaliar um componente, é requerida a simulação desse componente e o resultado é comparado com os dados qualitativos fornecidos por um vendedor. A métrica de custo é definida pelo valor de mercado de um componente.

A avaliação arquitetural desta abordagem é direcionada apenas nos próprios componentes, ou seja, despreza a análise arquitetural da própria arquitetura (interações entre componentes). Outro fato importante é que a avaliação de alguns atributos de qualidade espera valores qualitativos e quantitativos dos vendedores dos componentes a respeito de seus produtos, entretanto estes valores não são confiáveis, visto que no mercado de componentes há diversos modelos de componentes, dificultando a padronização.

5.4. Comparativo entre os Trabalhos Relacionados

A Tabela 2 mostra um comparativo resumido entre os trabalhos apresentados.

Tabela 5 – Comparação entre os trabalhos relacionados.

Trabalho	Algoritmos de Otimização	Otimização multi-objetiva	Métricas	Seleção orientada por Projeto Arquitetural
[Haghpanah07]	Algoritmos Genéticos (AG) e Abordagem Gulosa	Não Possui	Custo	Não

[Vescan08]	AG	Soma Ponderada	Atributos de qualidade: custo e funcionalidade	Não
[Vijayalakshmi08]	AG	Soma Ponderada	Atributos de qualidade: custo, funcionalidade, complexidade e confiabilidade	Sim
Trabalho Proposto	AG	Soma Ponderada	Estrutura da arquitetura, similaridade e informações de reúso	Sim

Todos os trabalhos correlatos discutidos nesta proposta utilizam Algoritmos Genéticos. A otimização multi-objetiva abordada por [Vescan08] e [Vijayalakshmi08] combinam suas funções de adequação individuais em uma única função composta por intermédio da soma ponderada. As métricas de atributos de qualidade e o custo apenas analisam os componentes, ignorando a estrutura de uma instância arquitetural formada pelos componentes selecionados. O processo de seleção orientado por Projeto Arquitetural proporciona que os componentes candidatos sejam avaliados comparativamente com a especificação de componente presente no Projeto Arquitetural. Este projeto é idealizado pelo arquiteto de software, onde o mesmo espera que a configuração arquitetural formada pelos componentes candidatos seja a mais próxima possível do seu projeto. Vê-se que os trabalhos correlatos não possuem validação de seus experimentos por meio de testes estatísticos, desta forma depreciam a credibilidade dos resultados [Arcuri11]. Por o presente trabalho utilizar informações de reúso, esta abordagem de forma implícita ou indireta representa a percepção dos desenvolvedores sobre variados atributos de qualidade dos componentes. Portanto, o processo de seleção de componentes proposto não é limitado em apenas alguns de atributos de qualidade para realizar as avaliações.

Um estudo [Land08], que compilou as melhores práticas de seleção de componentes tanto na academia quanto na indústria, recomenda que quatro tipos de critérios de avaliação deveriam ser considerados em um processo de seleção de componentes. Tais critérios são: funcionalidade, estrutural arquitetural, atributos de qualidade e informações de mercado.

Todos os quatro são explorados pelo presente trabalho. O primeiro critério supracitado é englobado pela métrica de nível de similaridade, pois quanto mais similar o componente candidato for em relação ao seu correspondente idealizado pelo arquiteto de software, maior

será sua satisfatibilidade funcional. O segundo critério é explorado diretamente. Os dois últimos são mapeados implicitamente para as informações de reúso, pois a relevância de um componente candidato depende de sua qualidade e do modo de como seu vendedor o disponibiliza no mercado. Portanto, o presente trabalho é o único da tabela comparativa que explora todos os critérios recomendados pela pesquisa de [Land08].

Capítulo 6

Considerações Finais

Neste trabalho foi apresentada uma proposta para uma Seleção Automatizada de Componentes de Software Orientada por Métricas Estruturais e Informações de Reúso. Conforme foi esclarecido, o processo de seleção de componentes é um trabalho muito complexo para ser resolvido apenas com a experiência do arquiteto de software, sendo necessário um auxílio computacional para obter sucesso nas escolhas dos componentes que constituirão um sistema de software.

Portanto, a presente proposta consiste no processo de seleção de componentes utilizando fatores diferentes dos que normalmente são evidenciados nestes processos de seleção que utilizam técnicas de otimização como estratégia para tratar o problema enfrentado. Esses fatores estão relacionados com métricas de avaliação arquitetural, as quais avaliam a estrutura da arquitetura e conseqüentemente medem o esforço de integração e adaptação dos componentes candidatos a compor uma aplicação. Os componentes candidatos a comporem o sistema em desenvolvimento também são avaliados por métricas, pois as qualidades dos elementos participantes de uma arquitetura influenciam na qualidade geral da mesma. A automatização desse processo de seleção é possível pela utilização de um algoritmo de otimização (meta-heurística) que faz com que se possa avaliar um problema demasiadamente complexo e com muitas soluções, e oferecer ao arquiteto de software uma série de recomendações de instâncias arquiteturais para o problema em questão.

A concepção e o delineamento do trabalho utilizam como suporte a abordagem de Engenharia de Software Baseada em Busca, onde a mesma serve como um guia para reformular problemas de busca com complexidade não-polinomial da Engenharia de Software em um algoritmo de otimização visando obter soluções próximas da ótima. Essa abordagem é considerada recente e emergiu com a motivação de explorar novas alternativas desse campo de pesquisa, pois a tendência é que os produtos de software se tornem cada vez mais complexos e que sua escala aumente em uma proporção cada vez maior, e a metodologia de desenvolvimento desses produtos devem se adaptar a novas fronteiras alcançadas por eles.

Os modelos atuais de seleção de componentes possuem a deficiência de não tratarem as relações estruturais, em outras palavras, somente se concentram nos componentes participantes da arquitetura modelada pelo arquiteto de software. A perspectiva de DBC (Desenvolvimento baseado em Componentes) remete ao desenvolvedor reutilizar componentes outrora implementados para construir um sistema em algum domínio de aplicação, e tais componentes são COTS (*Commercial Off-The-Shelf*) os quais são desenvolvidos por terceiros e distribuídos em repositórios (como produtos eletrônicos que são dispostos na prateleira para venda) para serem consumidos. Cada desenvolvedor de componentes para reúso na maioria dos casos utiliza seus modelos e padrões próprios, resultando em inconsistências no momento da integração entre os componentes selecionados para compor o sistema. É muito importante prever os problemas estruturais (conexões) relacionados à arquitetura para não comprometer o custo e o tempo de construção do sistema. O *mismatch* entre implementações de componentes é inevitável, portanto é mister a elaboração e utilização de métricas para medir o esforço que seria empregado para conectar componentes, assim evita surpresas ou retrabalho quando o estágio do DBC está numa etapa posterior a da seleção de componentes.

Partindo agora para a perspectiva de avaliação dos componentes como elementos da arquitetura, é importante mencionar o fato de que a grande maioria dos trabalhos que abordam seleções de componentes não teriam uma aplicabilidade prática em um cenário real, pois cada domínio de aplicação escolhe seus próprios atributos de qualidade para avaliar os componentes candidatos de alguma instância arquitetural. Quando se imagina em processo de seleção de componentes, o mesmo deve ajudar o arquiteto de software a escolher as implementações de componentes candidatas sob vários domínios de aplicação, ou seja, um processo que seja robusto e genérico o suficiente para atender essa etapa do DBC. Há problemas como a ausência de padronização de atributos de qualidade, assim como a dificuldade de expressar em métricas alguns desses atributos que são muito abstratos para representá-los computacionalmente. Por isso, a investigação das informações de reúso é tratada como uma alternativa para a avaliação dos componentes candidatos na perspectiva de utilização dos atributos de qualidade. As informações de reúso implicitamente apresentam uma carga genérica desses atributos de qualidades, assim como disponibilizam o nível de relevância do componente em avaliação. Isso beneficia a aplicabilidade prática do processo de seleção, pois todos os domínios de aplicação podem ser contemplados sem nenhum conflito conceitual.

Após a motivação da utilização das métricas produzidas para avaliação de arquiteturas de software, tem-se a preocupação de observar se tais métricas realmente estão guiando o algoritmo de otimização desenvolvido para tratar o problema de seleção de componentes. Assim como mostrar que o esforço utilizado na elaboração desse algoritmo teve seu propósito bem sucedido, ou seja, tal algoritmo proporciona melhores resultados do que abordagens mais simples. Dessa forma, é importante realizar a validação do algoritmo desenvolvido para evidenciar a credibilidade dos resultados apresentados. Diferentemente da grande maioria dos trabalhos que utilizam algoritmos de otimização no campo da Engenharia de Software, a validação deste trabalho proposto tem como suporte testes estatísticos para provar a dominância do novo algoritmo em relação a abordagens mais simples. Outro fator interessante foi que o algoritmo desenvolvido sempre encontrou a solução ótima, portanto, desempenhou certa equivalência qualitativa em comparação com um algoritmo de força bruta.

6.1. Limitações e Trabalhos Futuros

As limitações apresentadas nesta seção são justamente os trabalhos futuros que devem ser desenvolvidos em outra oportunidade. Aqui serão discutidos dois tipos de limitação presentes no trabalho proposto. Inicialmente, serão apontadas formas de automatizar a obtenção da massa de dados para a execução do algoritmo proposto. Por fim, serão abordados fatores conceituais dos componentes que não foram tratados no presente trabalho.

Seria importante e prática a criação de uma ferramenta que auxilie o arquiteto de software a recuperar, no Serviço do Repositório, artefatos dos componentes de software candidatos a partir do diagrama da arquitetura produzido no Projeto Arquitetural. Assim, após as concepções das especificações de componentes no diagrama da arquitetura, as consultas seriam construídas automaticamente na linguagem estabelecida pelo Serviço de Busca, a qual é a *BranchGuide*, para poupar o esforço ou até o esquecimento do arquiteto de alguma consulta, elemento ou atributo no momento da busca pelos artefatos referentes aos componentes candidatos. Portanto, a entrada para essa ferramenta seria um conjunto de especificações de componentes pertencentes à arquitetura e a saída seria um conjunto de consultas com o objetivo de buscar artefatos dos componentes candidatos, pois é por meio desses artefatos que um leitor de arquivos XML irá capturar todas as informações dos candidatos e as disponibilizarão para serem processadas pela meta-heurística desenvolvida.

No Desenvolvimento Baseado em Componentes, o processo de determinar quais operações pertencem a uma interface é considerado fatoração. Há duas vertentes distintas para

processo de fatoração: uma remete ao desenvolvimento do componente COTS e a outra se refere à especificação do componente em algum domínio de aplicação específico. Na primeira vertente, a equipe de desenvolvimento do componente idealizado para aplicação em diversos domínios (componente concebido para o reúso) pensará na fatoração de maneira genérica, e agrupará operações em alguma interface, e consequentemente criará um contrato para a mesma. Uma vez estabelecido o contrato das operações de uma interface, ela será invariável e a integração do componente que a utilizará será por meio desse contrato. Já a vertente das especificações dos componentes em domínios de aplicações específicos a metodologia para fatoração segue uma linha diferente. No momento que o arquiteto for planejar a arquitetura de um novo sistema, ele especificará essa arquitetura baseada nos requisitos específicos de cada projeto, e as especificações de componentes de software serão conectadas de acordo com a arquitetura planejada para se adequar ao sistema em construção. Também é importante salientar que as interfaces das especificações de componentes serão projetadas para beneficiar a qualidade estrutural da arquitetura como tentar oferecer as conexões com fraco acoplamento e alta coesão. Dependendo até da mudança de pessoa que irá fatorar uma interface, o seu design pode variar, cada um irá executar a fatoração utilizando boas práticas de design (i.e. utilizar fundamentos orientados a objetos) e sua própria experiência. Basicamente quando se pretende fatorar uma interface, sempre é necessário pensar em termos do reúso, e em uma aplicação orientada a componentes a unidade básica de reúso é a interface [Löwy05].

As operações de um componente são expostas via interfaces. Qualquer componente de software pode possuir mais de uma interface, e a sua quantidade vai depender da fatoração no momento em que for realizar a especificação. O número de possibilidades de exposição de interfaces é o mesmo número de partições que um conjunto pode ter. O componente tem um conjunto de operações, e são expostas por partições (subconjuntos) que correspondem às interfaces. Exemplificando, se um componente possuir três operações (*op1*, *op2* e *op3*), há cinco possibilidades de fatoração, que são:

- Operações distribuídas em uma interface: $\{op1, op2, op3\}$.
- Operações distribuídas em duas interfaces: $\{op1\}, \{op2, op3\}$.
- Operações distribuídas em duas interfaces: $\{op2\}, \{op1, op3\}$.
- Operações distribuídas em duas interfaces: $\{op3\}, \{op1, op2\}$.
- Operações distribuídas em três interfaces: $\{op1\}, \{op2\}, \{op3\}$.

O número de possibilidades (partições) explode facilmente, e no ramo da combinatória esse valor é chamado de *número de Bell* [Rota1964].

Comparando uma interface de certa especificação de componente com uma interface pertencente a alguma implementação de componente candidata, nem sempre se tem comparação injetora, em outras palavras, no panorama real de aplicação não é garantido que uma interface só pode ser comparada com apenas uma outra, por causa da fatoração de interfaces. Portanto, é uma linha de pesquisa que seria interessante investigar como trabalho futuro, pois certamente na prática ocorrerão inconsistências de fatoração entre interfaces.

Na mesma linha de raciocínio da fatoração de interfaces, o próprio componente de software pode ser formado pela composição de outros componentes. Exemplificando, um componente que realize funcionalidades de uma calculadora pode ser formado por um componente que realize operações de soma e adição, juntamente com outro componente que realize operações de multiplicação e divisão. Dessa forma, se no Projeto arquitetural for especificado um componente que possui várias funcionalidades, certamente será difícil encontrar em um repositório um componente que atenda todas as funcionalidades desejadas, entretanto seria fácil encontrar componentes que são subconjuntos das funcionalidades da especificação, e então uni-los para possuir uma equivalência funcional ao que foi especificado na arquitetura do Projeto Arquitetural. Esse tipo de situação é rotineiro para desenvolvedores de sistemas que utilizam a abordagem de DBC, e sua implementação nesse trabalho traria mais complexidade e aumentaria mais ainda o espaço de busca.

Por fim, o presente trabalho buscou investigar novos caminhos para melhorar a qualidade do processo de seleção de componentes de software, como a avaliação estrutural da arquitetura e uma avaliação genérica, apoiada por informações de reúso, para os componentes de software. A utilização da Engenharia de Software Baseada em Busca nesse trabalho mostrou que é um caminho a se seguir e realizar pesquisas mais a fundo, já que essa abordagem ainda é considerada emergente.

Referências

- [Ali09] Ali, S.; Briand, L.; Hemmati, H.; Panesar-Walawege, R., “*A systematic review of the application and empirical investigation of search-based test-case generation*”, IEEE Transactions on Software Engineering (TSE), 2009.
- [Arcuri11] Arcuri, A.; Briand, L., “*A practical guide for using statistical tests to assess randomized algorithms in software engineering*”, Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011), pp. 1-10, 2011.
- [Ardimento04] Ardimento, P.; Bianchi, A.; Visaggio, G.; “*Maintenance-oriented selection of software components*”, Proceedings of the 8th European Conference on Software Maintenance and Reengineering, pp. 115-124, 2004.
- [Bay04] Bay, T. G.; Pauls, K., “*Reuse Frequency as Metric for Component Assessment*”, Lecture Notes in Computer Science (Freie Universität Berlin), Vol. 3798/2005, pp. 164-176, 2004.
- [Becker06] Becker, S.; Brogi, A.; Gorton, I.; Overhage, S.; Romanovsky, A.; Tivoli, M., “*Towards an Engineering Approach to Component Adaptation*”, Springer-Verlang. 2006.
- [Birckle95] Birckle, T.; Thiele, L., “*A Comparison of Selection Schemes used in Genetic Algorithms*”, TIK-Report, 1995.
- [Brin98] Brin, S.; Page, L., “*Anatomy of a large-scale hypertextual web search engine*”, Proceedings of the 7th International World Wide Web Conference, 1998.
- [Brownlee12] Brownlee, A.; Wright, J. A., “*Solution analysis in multi-objective optimization*”, Building Simulation and Optimization, Loughborough, 2012.
- [Cheesman01] Cheesman, J.; Daniels, J., “*UML Components: A Simple Process for Specifying Component-Based Software*”, Addison-Wesley, Boston, 2001.

-
- [Clarke03] Clarke, J.; Dolado, J. J.; Harman, M.; Hierons, R.; Jones, B.; Lumkin, M.; Mitchell, B.; Mancoridis, S.; Rees, K.; Roper, M.; Shepperd, M., “*Reformulating software engineering as a search problem*”, IEEE Proceedings – Software, Vol. 150, Iss. 3, pp. 161-175, 2003.
- [Clements02] Clements, P.; Kazman, R.; Klein, M., “*Evaluating software architectures: methods and case studies*”, Addison-Wesley, 2002.
- [Chung99] Chung L.; Nixon B. ; Yu E.; Mylopoulos J., “*Non-Functional Requirements in software engineering*”, Springer, 1999.
- [Deb01] Deb, K., “*Multi-Objective Optimization Using Evolutionary Algorithms*”, John Wiley & Sons, Inc., USA, 2001.
- [Eden03] Eden, A.; Kazman, R., “*Architecture, Design, and Implementation*”, Proceedings of the 25th International Conference on Software Engineering, 2003.
- [Egyed00] Egyed, A.; Medvidovic, N.; Gacek, C., “*Component-Based Perspective on Software Mismatch Detection and Resolution*”, IEE Proceedings – Software, vol. 147, No. 6, pp. 225–236, 2000.
- [Feder06] Feder, T.; Subi, C., “*Partition into k -vertex subgraphs of k -partite graphs*”, Electronic Colloquium on Computational Complexity, Report No. 16, 2006.
- [Forrest96] Forrest, S., “*Genetic algorithms*”, ACM Computing Surveys, 1996.
- [Frakes05] Frakes, W.; Kyo, K., “*Software Reuse Research: Status and Future*”, IEEE Transactions on Software Engineering, Vol. 31, No. 7, 2005.
- [Freitas09] Freitas, F. G.; Maia, C. L. B.; Coutinho, D. P.; Campos, G. A. L.; Souza, J. T., “*Aplicação de Metaheurísticas em Problemas da Engenharia de Software: Revisão de Literatura*”, II Congresso Tecnológico Infobrasil, 2009.
- [Garlan94] Garlan, D.; Shaw, M., “*An Introduction to Software Architecture*”, Advances in Software Engineering, 1994.
- [Goldberg91] Goldberg, D. e Deb, K., “*A Comparative Analysis of Selection Schemes Used in Genetic Algorithms*”, Foundations of Genetic Algorithms, Morgan Kaufmann, USA, 91

-
- [Haghpanah07] Haghpanah, N.; Moaven, S., “*Approximation Algorithms for Software Component Selection Problem*”, Proceeding of 14th Asia-Pacific Software Engineering Conference, 2007.
- [Hamdy11] Hamdy, M.; Hasan, A.; Siren, K., “*Impact of adaptive thermal comfort criteria on building energy use and cooling equipment size using a multi-objective optimization scheme*”, Energy and Buildings, pp. 2055-2067, 2011.
- [Harman01] Harman, M.; Jones, B. F., “*Search-based Software Engineering*”, Information and Software Technology, Vol. 43, No. 14, pp. 833-839, 2001.
- [Harman06] Harman, M., “*Search based software engineering*”, 6th International Conference on Computational Science, pp. 740-747, 2006.
- [Harman07] Harman, M., “*The Current State and Future of Search Based Software Engineering*”, Proceedings of the Future of Software Engineering Workshop, 2007.
- [Harman09] Harman, M.; Mansouri, S. A.; Zhand, Y., “*Search Based Software Engineering: A Comprehensive Analysis and Review of Trends Techniques and Applications*”, ACM Computing Surveys, Vol. 45, Iss. 1, No. 11, 2009.
- [Harman10] Harman, M., “*Why the Virtual Nature of Software Makes It Ideal for Search Based Optimization*”, Proceedings of the 13th International Conference on Fundamental Approaches to Software Engineering, Vol. 6013, pp. 1-12, 2010.
- [Heineman99] Heineman, G., “*Adaptation of Software Components*”, 2nd International Workshop on Component-Based Software Engineering, 1999.
- [Heineman01] Heineman, G.; Council, W., “*Component-Based Software Engineering: Putting the Pieces Together*”, Addison-Wesley, 2001.
- [Henningsson02] Henningsson, K.; Wohlin, C., “*Understanding the Relations between Software Quality Attributes - A Survey Approach*”, Proceedings 12th International Conference for Software Quality, 2002.
- [Holland75] Holland, J. H., “*Adaptation in Natural and Artificial Systems*”, University of Michigan Press, USA, 1975.
- [Hoek03] Hoek, A.; Dincel, E.; Medvidovic, N., “*Using Service Utilization Metrics to Assess the Structure of Product Line Architectures*”, Proceedings of the 9th International Symposium on Software, 2003.

-
- [Hu07] Hu, X. and Di Paolo, E. A., “*An efficient genetic algorithm with uniform crossover for the multi-objective airport gate assignment problem*”, in 2007 IEEE Congress on Evolutionary Computation, CEC 2007, Singapore, September 25-28, 2007.
- [Jaccard1901] Jaccard, P., “*Étude comparative de la distribution florale dans une portion des Alpes et des Jura*”, Bulletin del la Société Vaudoise des Sciences Naturelles, 1901.
- [Kazman93] Kazman, R.; Bass, L.; Abowd, G.; Webb, M., “*Analyzing the Properties of User Interface Software*”, Technical Report, 1993.
- [Konak06] Konak, A.; Coit, D. W.; Smith, A. E., “*Multi-objective optimization using genetic algorithms: A tutorial*”, Reliability Engineering & System Safety, Vol. 91, Iss. 9, pp. 992-1007, 2006.
- [Kontio96] Kontio, J., “*A case study in applying a systematic method for COTS selection*”, Proceedings of the 18th international conference on Software engineering, pp. 201-209, 1996.
- [Land08] Land, R.; Blankers, L.; Chaudron, M.; Crnković, I., “*COTS Selection Best Practices in Literature and in Industry*”, Proceedings of the 10th international conference on Software Reuse: High Confidence Software Reuse in Large Systems, pp 100-111, 2008.
- [Leech02] Leech, N.; Onwuegbuzie, A., “*A Call for Greater Use of Nonparametric Statistics*”, Technical report, US Dept. Education, 2002.
- [Li93] Li, W.; Henry, S., “*Object-Oriented Metrics that Predict Maintainability*”, Systems and Software, 1993.
- [Linden08] Linden, R., “*Algoritmos Genéticos: Uma Importante Ferramenta da Inteligência Computacional*”, 2ª Edição, Rio de Janeiro, Brasport, 2008.
- [Löwy05] Löwy, J., “*Programming .NET Components*”, 2nd Edition, O'Reilly, 2005.
- [Mancebo05] Mancebo, E.; Andrews, A., “*A Strategy for Selecting Multiple Components*”, ACM Symposium on Applied Computing, 2005.
- [Manning08] Manning, C.; Raghavan, P.; Schütze, H., “*Introduction to Information Retrieval*”, Cambridge University Press, 2008.
- [Marôco11] Marôco, J., “*Análise Estatística com o SPSS Statistics*”, 5ª Edição, Pero Pinheiro, 2011.

-
- [Matoušek08] Matoušek, R., “*Genetic Algorithm and Advanced Tournament Selection Concept*”, Nature Inspired Cooperative Strategies for Optimization, Studies in Computational Intelligence, Volume 236, 2008.
- [Narasimhan04] Narasimhan, V.; Hendradjaya, B., “*A New Suite of Metrics for the Integration of Software Components*”. The University of Adelaide, 2004.
- [Neumann44] Neumann, J.; Morgenstern, O., “*Theory of Games and Economic Behavior*”, Princeton University Press, 1944.
- [Neumann04] Neumann, L., et al., “*Otimização combinatória empregando algoritmo genético aplicada na análise multivariada de medicamentos manipulados*”, Encontro Nacional de Engenharia de Produção, 2004.
- [Oliveira06] Oliveira, J. P. F.; Santos, M. S.; Elias, G., “*ComponentForge: Um Framework Arquitetural para Desenvolvimento Distribuído Baseado em Componentes*”, IV Workshop de Desenvolvimento Baseado em Componentes, 2006.
- [Oliveira07] Oliveira, J. P. F.; Brito, T.; Rabelo, S.; Elias, G., “*Um Serviço de Repositório Compartilhado e Distribuído para Suporte ao Desenvolvimento Baseado em Componentes*”, XXI Simpósio Brasileiro de Engenharia de Software, 2007.
- [Oliveira08] Oliveira, J. P., “*X-CORE: Um Serviço de Repositório Distribuído e Compartilhado de Componentes de Software*”, Universidade Federal da Paraíba, UFPB, 2008.
- [Oliveira Junior10] Oliveira Junior, E.; Maldonado, J. C.; Gimenes, I., “*Uma Revisão Sistemática sobre Avaliação de Linha de Produto de Software*”, 2010.
- [OMG13] OMG, “*Unified Modeling Language*”, (Online) <http://www.uml.org/>, 2013.
- [Orso00] Orso, A.; Harrold, M. J.; Rosenblum, D. S., “*Component Metadata for Software Engineering*”, Second International Workshop on Engineering Distributed Objects, pp. 129-144, 2000.
- [Petty03] Petty, M.; Weisel, E.; Mielke, R., “*Computational Complexity of Selecting Components for Composition*” Proceedings of the Fall 2003 Simulation Interoperability Workshop, 2003.
- [Port04] Port, D.; Chen, S., “*Assessing COTS Assessment: How Much Is Enough?*”, Lecture Notes in Computer Science, 2004.

-
- [Preiss02] Preiss, O, “*A Systems Perspective on Quality Description of Software Components*”, Proceedings of the 6th World Multiconference on Systemics, Cybernetics and Informatics, pp. 250-255, 2002.
- [Razali11] Razali, N. M.; Geraghty, J., “*Genetic Algorithm Performance with Different Selection Strategies in Solving TSP*”, Proceedings of the World Congress on Engineering 2011, Vol II, 2011.
- [Rota1964] Rota, G. C., “*The American Mathematical Monthly*”, Vol. 71, 5. pp. 498-504, 1964.
- [Roy08] Roy, B; Graham, T. C. N., “*Methods for Evaluating Software Architecture: A Survey*”, Technical Report No. 2008-545, School of Computing, Queen's University at Kingston, Ontario, Canada, 2003.
- [Saaty82] Saaty, T. L., “*Decision Making for Leaders*”, Lifetime Learning Publications, 1982.
- [Sametingner97] Sametingner, J., “*Software Engineering with Reusable Components. Berlin*”, Springer-Verlang, 1997.
- [Santos06] Santos, M. S., “*X-ARM : um modelo de representação de artefatos de software*”, Dissertação de Mestrado, Natal, 2006.
- [SBSE13] SBSE, “*Search Based Software Engineering Repository*”, Disponível em: http://crestweb.cs.ucl.ac.uk/resources/sbse_repository/, 2013.
- [Schuenck07] Schuenck, M.; Elias, G., “*Suporte à Certificação de Componentes no Modelo de Representação X-ARM*”, I Simpósio Brasileiro de Componentes, Arquiteturas e Reutilização de Software, 2007.
- [Sedigh-Ali01] Sedigh-Ali, S.; Ghafoor, A.; Paul, R., “*Metrics Guided Quality Management for Component-Based Software System*”, 25th Annual International Computer Software and Applications Conference, 2001.
- [Tian95] Tian, J.; Zelkowitz, M., “*Complexity measure evaluation and selection*”, IEEE Trans. on Software Engineering. 1995.
- [Torchiano04] Torchiano, M.; Morisio, M., “*Overlooked Facts on COTS-based Development*”, IEEE Software, Vol. 21, 2004.
- [Vescan08] Vescan, A., “*A Metrics-based Evolutionary Approach for the Component Selection Problem*”, Applications of Digital Information and Web Technologies, 2008.

-
- [Viana11] Viana, T., “*Uma Técnica de Indexação de Dados Semi-Estruturados para o Processamento Eficiente de Consultas com Ramificação*”, Universidade Federal da Paraíba, UFPB, 2011.
- [Vijayalakshmi08] Vijayalakshmi, K.; Ramaraj, N; Amuthakkannan, R., “*Improvement of component selection process using Genetic Algorithm for Component-Based Software Development*” International Journal of Information Systems and Change Management, 2008.
- [Vitharana03a] Vitharana, P., “*Risks and Challenges of Component-Based Software Development*”, Communications of the ACM, Vol. 46, No. 8, pp. 67-72, 2003.
- [Vitharana03b] Vitharana, P.; Zahedi, F.; Jain, H., “*Design, Retrieval, and Assembly in Component-based Software Development*”, Communication of the ACM, 2003.
- [W3C99] W3C, “*XML Path Language (XPath)*”, <http://www.w3.org/TR/xpath>, 1999.
- [W3Schools13] W3Schools, “*XML Tutorial*”, <http://www.w3schools.com/xml>, 2013.
- [Wolpert97] Wolpert, D. H.; Macready, W.G., “*No Free Lunch Theorems for Optimization*”, IEEE Transactions on Evolutionary Computation, Vol. 1, No. 1, 1997.
- [Xie09] Xie, H.; Zhang, M., “*Sampling Issues of Tournament Selection in Genetic Programming*”, Technical Report Series, School of Engineering and Computer Science, Victoria University of Wellington, New Zealand, 2009.
- [Zhang07] Zhang, Y.; Harman, M.; Mansouri, S., “*The Multi-Objective Next Release Problem*”, Proceedings of the 9th annual Conference on Genetic and Evolutionary Computation (GECCO '07), pp. 1129-1137, 2007.