UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

CHECAGEM DE CONFORMIDADE ARQUITETURAL NA MODERNIZAÇÃO ORIENTADA A ARQUITETURA

FERNANDO BEZERRA CHAGAS

ORIENTADOR: PROF. DR. VALTER VIEIRA DE CAMARGO

São Carlos – SP

Novembro-2016

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

CHECAGEM DE CONFORMIDADE ARQUITETURAL NA MODERNIZAÇÃO ORIENTADA A ARQUITETURA

FERNANDO BEZERRA CHAGAS

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação, área de concentração: Engenharia de Software

Orientador: Prof. Dr. Valter Vieira de Camargo

São Carlos – SP

Novembro-2016

Ficha catalográfica elaborada pelo DePT da Biblioteca Comunitária UFSCar
Processamento Técnico
com os dados fornecidos pelo(a) autor(a)

Chagas, Fernando Bezerra
C433c Checagem de conformidade arquitetural na modernização orientada a arquitetura / Fernando Bezerra Chagas. -- São Carlos : UFSCar, 2016.
78 p.

Dissertação (Mestrado) -- Universidade Federal de São Carlos, 2016.

1. Modernização apoiada por modelos. 2. Metamodelo de descoberta de conhecimento. 3. Checagem de conformidade arquitetural. I. Título.

UNIVERSIDADE FEDERAL DE SAO CARLOS

Centro de Ciências Exatas e de Tecnologia

Programa de Pós-Graduação em Ciência da Computação

Folha de Aprovação

Assinaturas dos membros da comissão examinadora que avaliou e aprovou a defesa de dissertação de mestrado do candidato Fernando Bezerra Chagas, realizada em 03/03/2016.

Prof. Dr. Valter Vieira de Camargo
(UFSCar)

Prof. Dr. Daniel Lucrédio
(UFSCar)

Prof. Dr. Ricardo Terra Nunes Bueno Villela
(UFLA)

Certifico que a sessão de defesa foi realizada com a participação à distância do membro Prof. Dr. Ricardo Terra Nunes Bueno Villela e, depois das arguições e deliberações realizadas, o participante à distância está de acordo com o conteúdo do parecer da comissão examinadora redigido no relatório de defesa do aluno Fernando Bezerra Chagas.

Prof. Dr. Valter Vieira de Camargo
Presidente da Comissão Examinadora
(UFSCar)

AGRADECIMENTOS

Uma caminhada, seja ela qual for, deixa as suas marcas. Tais marcas se mostram de diferentes maneiras, estando dentre elas as suas escolhas e as pessoas que passam por ela. Das escolhas que fiz, por ser humano e por mudar constantemente, certamente digo que poderia ter tomado caminhos diferentes durante a minha trajetória, porém, das pessoas que conheço e conheci durante este tempo, indubitavelmente, eu não quero mudar.

Por isso, hoje, agradeço à Deus, não por praxe, mas por ele não ter me abandonado, mesmo sendo eu um cristão desleixado com a sua fé.

Agradeço ao meu pai (Doutor) e a minha mãe (Leonice), por me darem uma vida que muitos não tiveram a oportunidade de ter, por me permitirem conhecer todo este Brasil sem que eles tenham mal saído do Maranhão e pelo meu caráter, do qual muitas das coisas foram construídas a partir do deles.

Agradeço à minha irmã (Luana Chagas), por fazer além de sua obrigação e trabalho. Atendendo e ouvindo os meus problemas durante a solidão em minha estada em São Carlos, quando na verdade eu que deveria ser o seu conselheiro.

Agradeço ao meu orientador Valter Camargo, novamente, não por praxe. Mas por ter me escolhido, por ter me ajudado com praticamente todas as minhas dificuldades e por ter sido o meu alvo nas horas de frustrações, pois sim, a todo momento jogamos a culpa nos orientadores, mas eles sabem que no fundo tudo isso faz parte do processo e que uma hora amadureceremos a ponto de entendermos que as dificuldades sempre estarão presentes em qualquer processo de aprendizado.

Agradeço ao Rener Baffa, irmão e compadre por opção, por ter sido mais do que um amigo e por ter me acolhido em sua casa, juntamente com a sua esposa (Francieli) e toda a sua família, que são pessoas de ótimos corações. As noites viradas no lolzinho não serão esquecidas, muito menos a companhia que me fez quando mais precisei!

Agradeço ao André Abade, irmão com quem dividi apartamento por quase toda a minha

estadia em São Carlos. Sábio, pois de sua boca saíram os conselhos que mais utilizei. Forte, por conseguir estar longe de sua filha para realizar esse sonho junto comigo. E companheiro, nas madrugadas se preparando para as provas, no boteco reclamando da vida e dos orientadores e ao telefone, quando precisei de sua palavra nas minhas tomadas de decisão.

Agradeço ao Pablo Bizzi, mais um irmão (são muitos né?), por me mostrar que na outra ponta do Brasil existiria uma pessoa de alma boa e coração gigante. Me ensinou a tomar chimarrão, me ensinou a lutar pelo sonho de empreender e me ajudou a fazer todos os vídeos para a amada nos momentos de saudade. Certamente amigo, TU não serás esquecido.

Agradeço ao Roberto Guimarães, irmão e compadre. Pois quem disse que precisa ter muito tempo junto para que uma amizade permaneça para sempre? O meu primeiro amigo de mestrado, me deu casa, comida e roupa lavada. Largou o mestrado e escolheu o mercado, mas eu o escolhi para ser meu irmão e ele me escolheu para ser o seu padrinho, obrigado.

Outros grandes amigos foram construídos durante este mestrado. Rafael Durelli, devo a ele o meu mestrado, pois com ele aprendi o poder da programação em pares e do pensamento em pares. Elias Adriano, o responsável por fazer o café que unia a turma que sempre tentava se afastar, também veio do nordeste e entende muito bem o que é estar sozinho. Ettore Tognoli, super inteligente e que acredita que dormir é algo psicológico. Luiz Pacini, mente brilhante e que nos presenteou por várias vezes com os seus ensinamentos em muitas oportunidades. E Eloísa, muito esforçada, tenha certeza que aquela salinha de estudos jamais nos esquecerá.

Aos outros, digo, não os esqueci. Augusto Chaves, Arthur Morais, Jãum Moreira, Guido Prado, Rafael Gastaldi, Renato Costa e Steve Tmat. Obrigado por terem me ajudado de alguma maneira até aqui, seja jogando um futebol na sexta, fazendo uma piada na quarta ou dando um conselho na segunda, obrigado.

RESUMO

Modernização Dirigida por Modelos (ADM) é uma iniciativa para a padronização dos processos de reengenharia. Dentre os metamodelos criados pela ADM, o mais importante é chamado de KDM (Metamodelo de Descoberta de Conhecimento), que é independente de plataforma e linguagem, além de ser padrão ISO. Uma importante etapa em uma Modernização Dirigida por Modelos é a Checagem de Conformidade Arquitetural (ACC), cujo objetivo é identificar violações entre as representações das arquiteturas planejada e atual de um sistema. Embora existam abordagens para ACC que atuam sobre código-fonte e modelos proprietários, não foram encontrados indícios desse tipo de abordagem para sistemas representados em KDM. Essa ausência de pesquisas na área dificulta a disseminação da ADM e aumenta o interesse em investigar a adequabilidade do KDM nesse contexto. Portanto, neste trabalho é apresentado o ArchKDM, uma abordagem para ACC baseado em KDM que depende exclusivamente do metamodelo KDM para representação i) do sistema legado a ser analisado; ii) da arquitetura planejada; iii) da arquitetura atual; e iv) das violações encontradas entre eles. ArchKDM é composta por três etapas: 1) Especificação da Arquitetura Planejada; 2) Extração da Arquitetura Atual; e 3) Checagem de Conformidade Arquitetural. O objetivo deste trabalho é investigar a adequabilidade do KDM como principal representação em todas as etapas da ACC, bem como fornecer uma abordagem para ACC no contexto da ADM. A abordagem foi avaliada utilizando dois sistemas reais e os resultados mostraram que não foram encontrados falsos positivos e negativos.

Palavras-chave: Modernização Apoiada por Modelos, Metamodelo de Descoberta de Conhecimento, Checagem de Conformidade Arquitetural.

ABSTRACT

Architecture-Driven Modernization (ADM) is a model-based initiative for standardizing reengineering processes. Its most important meta-model is KDM (Knowledge Discovery Metamodel), which is a platform and language-independent ISO standard. A important step in an Architecture-Driven Modernization is the Architectural Conformance Checking (ACC), whose goal is to identify the violations between the Planned (PA) and Current Architectures (CA) of a system. Although there are ACC approaches that act on source-code or proprietary models, there is none for systems represented as KDM. This absence hinders the dissemination of ADM and increases the interest for research that investigates the suitability of KDM in this context. Therefore, in this paper, we present ArchKDM, a KDM-based ACC approach that relies exclusively on the KDM meta-model for representing i) the legacy system under analysis; ii) the PA; iii) the CA; and iv) the violations between them. ArchKDM is composed of three tool-supported steps: 1) Specifying the Planned Architecture; 2) Extracting the Current Architecture; and 3) Performing the Checking. Our goal is to investigate the suitability of KDM as the main representation in all ACC steps as well as to deliver an ACC approach in the ADM context. We evaluated steps 2 and 3 of the approach using two real-world systems and the results showed no false positives and negatives.

Keywords: Architecture-Driven Modernization, Knowledge-Discovery Metamodel, Architectural Conformance Checking.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

GLOSSÁRIO

HTML – *HyperText Markup Language*

KDM – *Knowledge Discovery Metamodel*

MDA – *Model-Driven Architecture*

MDD – *Model-Driven Development*

MDE – *Model-Driven Engineering*

MOF – *Meta Object Facility*

OMG – *Object Management Group*

SQL – *Structure Query Language*

UML – *Unified Modeling Language*

XMI – *XML Metadata Interchange*

XML – *EXtensible Markup Language*

Capítulo 1

INTRODUÇÃO

1.1 Contextualização

No campo de pesquisa em manutenção de software, Sommerville (2004) define um sistema legado como aquele cujos custos de manutenção e evolução encontram-se fora dos níveis aceitáveis para uma organização, mas que ainda são úteis para apoiar seus processos internos. Frequentemente esse tipo de software possui sua estrutura inconsistente com a documentação, tornando a manutenção uma tarefa mais custosa. Porém, substituir completamente um sistema legado pode ser uma tarefa muito cara e propensa a erros.

De acordo com Pressman (2010), a manutenção é responsável por 90% do custo total de um software, portanto é uma importante atividade de seu ciclo de vida. O alto custo de sucessivas manutenções pode levar a equipe de desenvolvimento à substituição do sistema. Uma alternativa a esta substituição de um sistema legado são os processos de reengenharia. Chikofsky e Cross J.H. (1990) explica que a reengenharia de software é o processo de exame e alteração de um sistema com a possibilidade de reconstituí-lo em uma nova forma e subsequentemente reimplementá-lo a partir dessa nova forma.

A arquitetura é um dos artefatos que possui grande impacto nos processos de reengenharia de software. De acordo com Bosch (2000), a arquitetura de software é uma das partes mais importantes dentro do ciclo de vida de um sistema de software, pois as decisões tomadas em nível arquitetural possuem impacto direto sobre o sucesso ou fracasso na realização dos objetivos funcionais, de negócio e de qualidade do sistema. Entretanto, Murphy, Notkin e Sullivan (1995) explicam que, como os modelos arquiteturais possuem um alto nível de abstração, violações na arquitetura de um software surgem facilmente durante o desenvolvimento e evolução de sistemas de software. Tais violações são encontradas na literatura como erosão arquitetural.

Portanto, violações arquiteturais podem ser objeto de processos de reengenharia de software. No entanto, de acordo com Gurp e Bosch (2002) e Medvidovic, Egyed e Grünbacher (), apesar da grande quantidade de pesquisas na área de arquitetura de software e das diversas soluções propostas para evitar o fenômeno de erosão arquitetural, ainda é pouco comum que os artefatos que compõem a arquitetura planejada de um sistema sejam mantidos em sincronia com a sua implementação.

As vantagens de um projeto arquitetural, tais como manutenibilidade, reusabilidade, escalabilidade e portabilidade são diminuídas por conta do processo de erosão arquitetural. Uma boa opção para a verificação de violações arquiteturais é a Checagem de Conformidade Arquitetural (*Architecture Compliance Checking* - CCA). Silva e Balasubramaniam (2012) explicam que a CCA é uma abordagem para preencher a ponte entre os modelos de alto nível do projeto arquitetural e o código-fonte. Knodel e Popescu (2007) definiram a Conformidade Arquitetural (*Architecture Conformance*) como uma medida para definir em que grau a arquitetura implementada no código-fonte está em conformidade com a arquitetura do software planejada. Uma CCA pode ser verificada de diversas maneiras, dentre elas, por meio de modelos de reflexão (MURPHY; NOTKIN; SULLIVAN, 1995), matrizes de dependências estrutural (SANGAL et al., 2005) e consulta em código fonte (MOOR et al., 2007), tais técnicas possuem bons resultados na abordagem de checagem.

Dentre os termos mais recentes relacionados a reengenharia de software, a Modernização Apoiada por Modelos (*Architecture Driven Modernization* - ADM) se destaca. A ADM começou em 2003 com a criação da ADMTF (*Architecture-Driven Modernization Task Force*), uma iniciativa para a utilização dos conceitos da MDA (*Model-Driven Architecture*) no contexto de modernização de software OMG (). A ADM defende a realização do processo de reengenharia considerando os princípios do desenvolvimento dirigido a modelos, isto é, ela cuida dos artefatos de software como protótipos, de maneira a facilitar a formalização de transformações entre modelos determinísticos. Izquierdo, Zapata e Molina (2010) dizem que o objetivo da ADM é definir um conjunto de metamodelos padrões para representar as informações envolvidas em um processo de modernização e facilitar a interoperabilidade entre ferramentas.

Dentre os metamodelos propostos pela ADM encontram-se o Metamodelo de Descoberta de Conhecimento (*Knowledge Discovery Metamodel* - KDM). De acordo com Perez-Castillo, Guzman e Piattini (2011), o KDM permite representar todos os artefatos de um sistema legado, em diferentes níveis de abstração, tais como código-fonte, interface de usuário, banco de dados, arquitetura e processos de negócio. O pacote do KDM mais importante no contexto deste trabalho é o de Estrutura (*Structure*), que permite representar elementos arquiteturais.

1.2 Motivação

Como visto anteriormente, um dos principais objetivos da ADM é permitir a realização de modernizações voltadas para a arquitetura, incluindo, claramente, as que permitem corrigir violações arquiteturais. A principal tarefa nesse tipo de modernização é identificar as violações arquiteturais existentes entre o sistema legado e a arquitetura planejada, para que, em seguida, possam ser aplicadas refatorações. Entretanto, apesar de a ADM fornecer a estrutura necessária para a modernização de sistemas, ela não oferece as ferramentas necessárias.

Foram encontradas abordagens que se concentram na proposta de alternativas para a realização de atividades de modernização. Rahimi e Khosravi (2010), Maffort et al. (2013), Deiters et al. (2009), Terra e Valente (2009) e Bittencourt et al. (2010) apresentaram propostas para a checagem de conformidade arquitetural. No entanto, nenhum desses trabalhos apresenta estudos sobre a viabilidade de realização da checagem de violações arquiteturais no contexto da modernização dirigida por modelos que utilizam o metamodelo KDM, que é padrão ISO e pouco estudado até o momento desta pesquisa. Isso foi um fator importante para a motivação deste trabalho, pois não foram encontradas abordagens que realizem modernizações por meio de padrões, o que impacta na dificuldade de reúso, portabilidade e interoperabilidade de aplicações.

O segundo fator motivador foi a ausência de estudos que apontam se é possível conduzir uma CCA utilizando o metamodelo KDM em sua forma original, sem extensões, além da apresentação de vantagens e desvantagens na utilização deste modelo na realização de uma CCA. Pois apesar de o KDM apresentar elementos que representam estruturas em alto nível, tais como camadas, componentes e subsistemas, não há evidência da qualidade dessa representação. Outro fator motivador foi a escassez de técnicas para identificação de violações arquiteturais independentes de linguagem de programação e plataforma. Tais técnicas permitem que o engenheiro de software conduza modernizações em um alto nível de abstração, resultando em aplicações modernizadas reusáveis.

Também foi fator motivador a ausência de estudos sobre a utilização de estilos arquiteturais para a especificação de arquiteturas de software. Portanto, deve-se verificar se uma CCA que segue as restrições impostas por estilos arquiteturais é capaz de reproduzir bons resultados na especificação de uma arquitetura. Também não foram encontradas evidências de que os elementos presentes no KDM para representação arquitetural sejam suficientes para demonstrar todos os detalhes existentes nesse nível de abstração. Dessa forma, investigar esses elementos e sugerir, se necessário, alterações no metamodelo KDM também foi uma das motivações deste

trabalho.

1.3 Abordagem desenvolvida e Resultados

A abordagem desenvolvida neste trabalho de mestrado, denominada ArchKDM, é capaz de realizar checagem de conformidade arquitetural no contexto da ADM. A saída desta abordagem é uma lista de possíveis violações arquiteturais entre um modelo da arquitetura planejada e outro da arquitetura atual do sistema analisado. ArchKDM foi dividida em três etapas: *I - Especificação da Arquitetura Planejada*, *II - Extração da Arquitetura Atual* e *III - Checagem de Conformidade*. Na primeira, o arquiteto é responsável por especificar a arquitetura planejada por meio da DCL-KDM, que é uma linguagem de descrição arquitetural (ADL) para especificação de arquiteturas de software considerando estilos arquiteturais. Na segunda etapa, é gerado um modelo em alto nível da arquitetura planejada. Por fim, os modelos são comparados e as possíveis violações são apresentadas.

O principal resultado obtido desta dissertação foi a criação de uma solução para a verificação de violações arquiteturais no contexto da modernização dirigida por modelos. Esta abordagem servirá como base para trabalhos que envolvam refatorações no metamodelo KDM a partir da identificação de violações arquiteturais. Dentre as soluções existentes nessa categoria, podem ser mencionadas as abordagens de Rahimi e Khosravi (2010), Maffort et al. (2013), Deiters et al. (2009), Terra e Valente (2009) e Bittencourt et al. (2010).

Outro resultado obtido foi verificar as vantagens e desvantagens de se utilizar o metamodelo KDM para realizar tal checagem. Também foi consequência deste trabalho a realização da especificação de arquiteturas de software utilizando estilos arquiteturais, por exemplo, usando camadas e componentes como elementos arquiteturais, além de definir possíveis restrições automaticamente, baseadas nesses estilos. Por fim, foi implementado um *plug-in* para o ambiente Eclipse para que a aplicação de checagem de conformidade arquitetural seja realizada.

Outros resultados deste projeto foram:

- Uma adaptação de uma ADL existente para contemplar a especificação de arquiteturas com a utilização de estilos arquiteturais;
- Uma técnica para conversão do pacote *code/kdm* para *structure/kdm*;
- Uma técnica para extração da arquitetura atual do sistema legado a partir de uma instância do KDM legado; e

- Uma ferramenta de apoio computacional para auxiliar o engenheiro durante a atividade de checagem de conformidade arquitetural no contexto da ADM de forma eficiente.

1.4 Grupo AdvanSE - Colaboração no Grupo de Pesquisa

Este trabalho é mais uma contribuição para o grupo de pesquisa AdvanSE (Advanced Research on Software Engineering), da Universidade Federal de São Carlos (UFSCar). O grupo possui pesquisas em andamento sobre extensões, refatorações, mineração, métricas e validações de arquitetura em modelos KDM.

1.5 Organização do Trabalho

No Capítulo 2, são descritas as fundamentações teóricas necessárias para o entendimento da proposta de pesquisa e sínteses da bibliografia fundamental. No Capítulo 3, são apresentados os trabalhos relacionados à checagem de conformidade arquitetural na modernização dirigida por modelos. No Capítulo 4, são apresentados os objetivos, metodologia a ser seguida, o método de avaliação que será aplicado e resultados esperados. No Capítulo 5, o presente trabalho é avaliado. Por fim, no Capítulo 6, são apresentadas as contribuições, limitações e trabalhos futuros.

Capítulo 2

FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são descritos os principais conceitos necessários para o entendimento do projeto. Ideias relacionadas à arquitetura de software são apresentadas na Seção 2.1. Na Seção 2.2 encontram-se os principais conceitos relacionados à checagem de conformidade arquitetural. Na Seção 2.3, a ADM é apresentada. Na Seção 2.4, o metamodelo KDM e suas camadas são descritas. Por fim, na Seção 2.5 são apresentadas as considerações finais.

2.1 Arquitetura de Software

Na década de 70, o maior custo da computação trocava de liderança, da qual o software assumia a primeira colocação, deixando o hardware em segundo lugar. Essa época foi caracterizada por Dijkstra (1972) como a crise do software. A complexidade dos problemas crescia e faltavam técnicas suficientes para a realização do desenvolvimento dessa alta demanda por software.

A partir dos problemas mencionados surgiu a Engenharia de Software, com o objetivo de apoiar o desenvolvimento desses sistemas complexos. Além disso, notou-se que em uma aplicação de software o código fonte é a parte que mais sofre com alterações. Outro ponto importante percebido é que mudanças nos requisitos de uma aplicação, mesmo que pequenas, podem resultar em uma quantidade enorme de alterações em código-fonte.

Um sistema de software considerado complexo possui componentes e conexões abstratas que serão aplicadas na parte de código-fonte do sistema. Tais sistemas devem possuir mecanismos de representação para identificar seus componentes, relações e propriedades em um nível de abstração compatível. A partir dessas ideias de alta complexidade de sistemas, surgiu a Arquitetura de Software que, de acordo com Shaw e Garlan (1996), é responsável pelo

estudo dos sistemas de software como um todo, além de seus componentes e seus respectivos relacionamentos.

Outra característica importante das arquiteturas de sistemas de software é que elas funcionam gerenciando de forma eficiente as preocupações dos *stakeholders*, como uma ferramenta conceitual. Além disso, a arquitetura é responsável por descrever a organização de seus componentes, relacionamentos, restrições e os princípios de concepção e evolução ao longo do tempo (Fowler (2003), Shaw e Garlan (1996) e Garlan (2000)). Uma documentação completa deve ser fornecida para que todos os interesses dos *stakeholders* sejam cobertos. Uma arquitetura pode ser descrita por diferentes perspectivas, fornecendo múltiplas visões arquiteturais de sistemas de software, como em Hofmeister, Nord e Soni (), Kruchten (1995) e Rozanski e Woods (2011).

Além das definições previamente citadas, na literatura são encontradas alternativas para a definição da arquitetura de um software, bem como a de Bass, Clements e Kazman (1998), cujos autores afirmam que arquiteturas de software são estruturas compostas por componentes e suas propriedades e relacionamentos. Estes elementos arquiteturais compõem a abstração do sistema auxiliando o gerenciamento de sua complexidade, em outras palavras, a especificação de uma arquitetura liga o problema à solução do negócio.

Braga (2013) cita aspectos que justificam a importância e a necessidade da arquitetura de software no processo de desenvolvimento de um sistema:

- Os requisitos e a implementação do sistema são ligados por meio da arquitetura do software;
- A arquitetura de software é responsável por atender às regras de negócio e os aspectos técnicos do sistema;
- A arquitetura representa um modelo de como o sistema deve ser organizado e como os seus componentes interagem; e
- De acordo com Bass, Clements e Kazman (1998), a arquitetura permite antecipar decisões de projeto, tais como o tempo de desenvolvimento, custo, manutenção, restrições de implementação e definição da estrutura organizacional.

Os elementos, as notações e as linguagens de uma arquitetura de software podem ser representados por meio de diversas maneiras, geralmente com objetivos diferentes. Braga (2013) diz que essas representações podem ser divididas em formais, semiformais e informais. A representação informal é considerada a mais simples delas. Descrições textuais e diagramas de caixas e linhas compõem esse tipo de representação.

As descrições textuais são feitas em linguagem natural e fornecem detalhes dos principais componentes da arquitetura de um sistema. Como ponto positivo desse tipo de representação, destaca-se o fato de que a linguagem natural proporciona facilidade no entendimento dos componentes do sistema. Porém, a parte negativa é que essa representação exige um grande esforço do leitor para que ela possa ser compreendida, além da dificuldade para encontrar informações específicas inseridas no texto de descrição da arquitetura. Além disso, um problema recorrente desse tipo de representação se dá devido ao espalhamento das descrições entre as relações e interações dos componentes.

Diagramas de caixas e linhas são um tipo de representação arquitetural amplamente difundido. De acordo com Garlan et al. (2010), as caixas representam os componentes enquanto as linhas são responsáveis pelos relacionamentos entre esses componentes. Esse tipo de representação geralmente é insuficiente para mostrar todos os detalhes da arquitetura de um sistema. No entanto, geralmente são utilizados comentários e/ou legendas para melhorar a especificação, o que resulta em um novo problema, a sobrecarga da representação. Além disso, caixas de formatos diferentes e linhas com diversas cores são utilizadas para diferenciar os componentes. Por um lado, os diagramas aparentam ser fáceis e intuitivos, porém tendem a ser incompletos e imprecisos. Portanto, a representação pode levar a múltiplas interpretações que dependem de cada elemento utilizado na descrição da arquitetura.

A segunda categoria de representação é chamada de semiformal, nela são atribuídos significados aos elementos de descrição da arquitetura de um sistema, resultando na diminuição da informalidade da representação. A UML e o modelo de visões arquiteturais 4+1 são os principais exemplos desse tipo de representação (KRUCHTEN, 1995).

De acordo com Booch, Rumbaugh e Jacobson (2005), a UML (*Unified Modeling Language*) é uma notação de modelagem de propósito geral que especifica sistemas de software orientados a objetos. Além disso, a UML oferece várias perspectivas de visualização de um sistema de software. Os pontos positivos são que a maioria dos desenvolvedores estão familiarizados com os diagramas da UML e a enorme quantidade de ferramentas disponíveis para mapeamento dos diagramas diretamente com a implementação. Por outro lado, pode acontecer dos diagramas se tornarem muito complexos, diminuindo assim a sua utilidade para análises ou, em outros casos podem resultar em diagramas muito simples, em que a sua expressividade para análise é insuficiente.

Kruchten (1995) descreve o Modelo de Visões Arquiteturais 4+1 como uma maneira de descrever a arquitetura de um sistema de software que utiliza quatro visões concorrentes, no qual cada uma se preocupa com um conjunto de interesses específicos dos participantes do sistema.

A visão lógica é dada pelos especialistas, clientes e usuários finais, ao passo que a visão de desenvolvimento é responsabilidade dos programadores e gerentes de projetos. Além das visões citadas, existe uma chamada de cenários, na qual são trabalhados casos de uso juntamente com cenários que abordam as outras quatro visões do modelo. Por um lado, a arquitetura pode ser organizada por meio de múltiplas visões, porém, essa independência impossibilita uma padronização.

No contexto deste trabalho, a representação formal é considerada uma boa maneiras para representar arquiteturas de sistemas de software, e se destacam por conta das Linguagens para Descrição de Arquitetura (*Architecture Description Language* - ADL), que podem ser divididas entre as de domínio específico ou de propósito geral (Medvidovic e Taylor (2000)). As ADLs se preocupam em definir uma linguagem para especificar as estruturas de alto nível de um sistema de software, deixando de lado os detalhes de implementação. Dessa maneira, Shaw e Garlan (1996) abordam que uma ADL deve se preocupar em especificar a descrição da arquitetura de software, na qual são definidos os seus respectivos componentes, comportamentos, padrões e mecanismos de interação entre eles.

O que caracteriza uma ADL como formal é que tais linguagens possuem um sintaxe bem formada e os seus elementos têm significados bem definidos. Dessa maneira, Shaw e Garlan (1996) descreveram seis propriedades que caracterizam uma ADL ideal:

- Heterogeneidade: possibilidade de combinar descrições arquiteturais múltiplas e heterogêneas;
- Reúso: possibilidade de reutilizar elementos arquiteturais;
- Composição: possibilidade de criar um sistema a partir da composição de elementos distintos e independentes;
- Configuração: possibilidade dar suporte à reconfiguração dinâmica e descrição da estrutura de um sistema independentemente dos elementos;
- Abstração: possibilidade de descrição dos elementos arquiteturais de um software de maneira que os papéis abstratos sejam claros e explícitos;
- Análise: possibilidade de realizar análises variadas e ricas nas descrições.

2.2 Checagem de Conformidade Arquitetural

De acordo com Knodel et al. (2006), um problema enfrentado pelos arquitetos de software é assegurar que um sistema possua a sua implementação (arquitetura atual) de acordo com o sistema planejado (arquitetura planejada), pois durante a implementação e evolução do sistema é comum serem encontradas diferenças devido a diversos motivos, tais como requisitos conflitantes, curtos prazos de entrega, falta de conhecimento dos desenvolvedores, entre outros. O Acúmulo de más decisões de implementação pode gerar o fenômeno conhecido como erosão ou violação arquitetural, descrito por Perry e Wolf (1992).

Dessa maneira, surgiu o termo Checagem de Conformidade Arquitetural, que conforme Knodel e Popescu (2007) é uma atividade chave para o controle de qualidade de software, cujo objetivo é revelar as diferenças entre a arquitetura do software planejada e o código fonte. Mais especificamente, a Checagem de Conformidade Arquitetural expõe o que não confere entre as restrições impostas na arquitetura planejada do sistema e as expressões, declarações e instruções no código fonte.

A conformidade arquitetural pode ser verificada estaticamente, comparando o código fonte à visão arquitetural do sistema, ou dinamicamente, verificando o código fonte em tempo de execução ou suas versões ao longo do tempo, comparando-as com a visão arquitetural. Knodel e Popescu (2007) comparam técnicas para checagem de conformidade arquitetural e citam os modelos de reflexão e as regras de relações de conformidade como duas das principais técnicas de verificação estática.

A Checagem de Conformidade Arquitetural avalia as dependências entre os componentes. Os resultados podem ser divididos em:

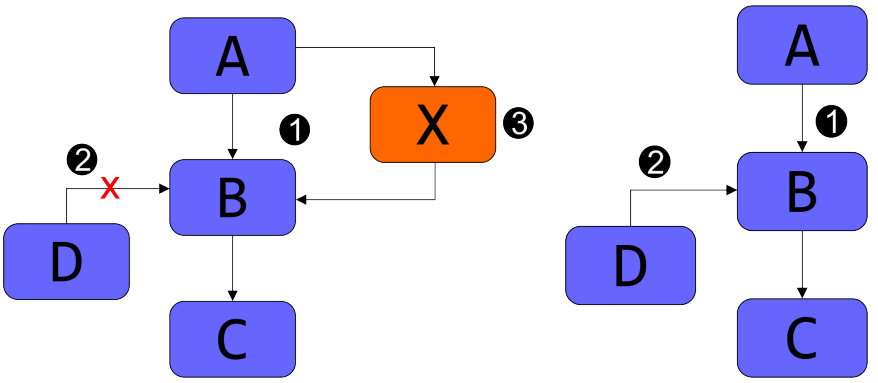
Convergência - é uma relação entre dois componentes que é permitida e foi implementada como pretendida. A convergência indica que a implementação é compatível com a arquitetura planejada.

Divergência - é uma relação entre dois componentes que não é permitida e não foi implementada como pretendida. A divergência aponta que a implementação não é compatível com o modelo arquitetural planejado.

Ausência - é uma relação entre dois componentes que era pretendida, porém foi implementada. A ausência indica que as relações na arquitetura planejada não foram encontradas na implementação.

Na Figura 2.1, é mostrado um exemplo de como funciona a checagem de conformidade ar-

quitetural entre dois modelos de alto nível representando as arquiteturas planejada (Figura 2.1(a)) e atual (Figura 2.1(b)) do sistema. As caixas representam elementos arquiteturais e as linhas representam os relacionamentos entre eles.



(a) Modelo de arquitetura planejada

(b) Modelo de arquitetura atual

Figura 2.1: Exemplo de Checagem de conformidade arquitetural

É possível notar que a relação entre os elementos **A** e **B**, tanto na arquitetura planejada quanto na atual estão representados da mesma maneira, portanto, há uma relação convergente, pois é permitida pelo arquiteto do sistema e foi implementada corretamente (como visto nas Figura 2.1(a) ❶ e 2.1(b) ❶). Em seguida, verifica-se que o arquiteto do sistema não permitiu relações partindo do elemento **D** para o elemento **B** na arquitetura planejada (Figura 2.1(a) ❷), porém, esta relação foi implementada na arquitetural atual (Figura 2.1(b) ❷), neste caso, a relação apresentada é considerada divergente. Por fim, as relações entre o elemento **X** e os elementos **A** e **B** (Figura 2.1(a) ❸) foi prevista, porém não foi implementada na arquitetura atual (Figura 2.1(a)), portanto, esta dependência é considerada do tipo ausente.

2.2.1 Modelos de Reflexão

Murphy, Notkin e Sullivan (1995) apresentam os modelos de reflexão. Inicialmente, a técnica foi proposta para ajudar o engenheiro a utilizar um modelo de alto nível de um sistema existente como uma lente para ver o modelo de código-fonte. Eles foram aplicados nos casos em que havia pouca ou nenhuma informação sobre o sistema de software e sua arquitetura.

Um exemplo encontrado na literatura é descrito por Knodel et al. (2006) e trata-se da ferramenta SAVE (*Software Architecture Visualization and Evaluation*), que realiza a análise estática de arquiteturas de software. SAVE realiza a comparação da arquitetura planejada do sistema com o seu código-fonte. De acordo com a abordagem, primeiramente os arquitetos de software devem definir um modelo de alto nível que represente a arquitetura planejada de um sistema.

Esse modelo é composto por diversos elementos, entre eles estão os componentes do sistema e as relações entre estes componentes. Para que os modelos sejam gerados, um passo importante é o mapeamento entre os elementos do modelo arquitetural e o código-fonte. Por fim, as relações são classificadas em convergentes, divergentes e ausentes.

2.2.2 Regras de relações de conformidade

As regras de relações de conformidade são descritas por Knodel e Popescu (2007) e especificam restrições entre elementos arquiteturais. As restrições se dividem em permitir, proibir ou impor alguma relação entre as entidades arquiteturais. Por exemplo, o componente **A** não pode acessar elementos não públicos da componente **B**. Em comparação com os modelos de reflexão, as regras de relação não necessitam de um modelo estrutural explícito, ou seja, as relações podem ser verificadas sem definir elementos arquiteturais.

Uma regra de relação de conformidade é composta por um tipo de relação, um componente de origem, um de destino e o tipo da regra de conformidade. Os componentes de origem e destino são definidos na expressão regular por meio dos nomes dos componentes. Por sua vez, o tipo de regra de conformidade determina se a relação entre os componentes é proibida ou se deve (obrigação) existir entre os componentes. O tipo de relação define qual o tipo de dependência existente entre os componentes, por exemplo, uma chamada de método ou declaração de variável.

Por exemplo, Knodel e Popescu (2007) explicam a regra "A **must exist** B", a qual estabelece que uma relação deve existir iniciando em um componente A e terminando em um componente B. Em outro caso, a regra de relação "C **is forbidden** D"define que o componente C está proibido de acessar elementos pertencentes ao componente D.

2.3 Modernização Apoiada por Modelos - ADM (Architecture-Driven Modernization)

O crescente interesse na Modernização Dirigida Por Engenharia Reversa (MDRE, *Model-Driven Reverse Engineering*) motivou a OMG a lançar a Modernização Apoiada por Modelos (ADM, *Architecture-Driven Modernization*), cujo objetivo foi estabelecer padrões para processos de reengenharia ADM (2015). Essa iniciativa foi motivada considerando o grande volume de projetos de reengenharia realizados sem sucesso que foram reportados na literatura e por fábricas de software, como descrito em Sneed (2005).

Perez-Castillo, Guzman e Piattini (2011) comentam que a ADM soluciona o problema de formalização porque é capaz de representar todos os artefatos envolvidos no processo de re-engenharia a partir de modelos padrões. A ADM trata todos os modelos homogeneamente, permitindo a criação de transformações Modelo para Modelo (M2M, *Model-to-Model*) entre eles.

De acordo com a ADM (2015), o seu objetivo não é substituir o processo tradicional de reengenharia, mas sim melhorá-lo por meio do uso da MDA (*Model-driven Architecture*). A ADM consiste em uma adaptação do modelo bem conhecido chamado ferradura. Na Figura 2.2 é explicado o modelo ferradura adaptado pela ADM.

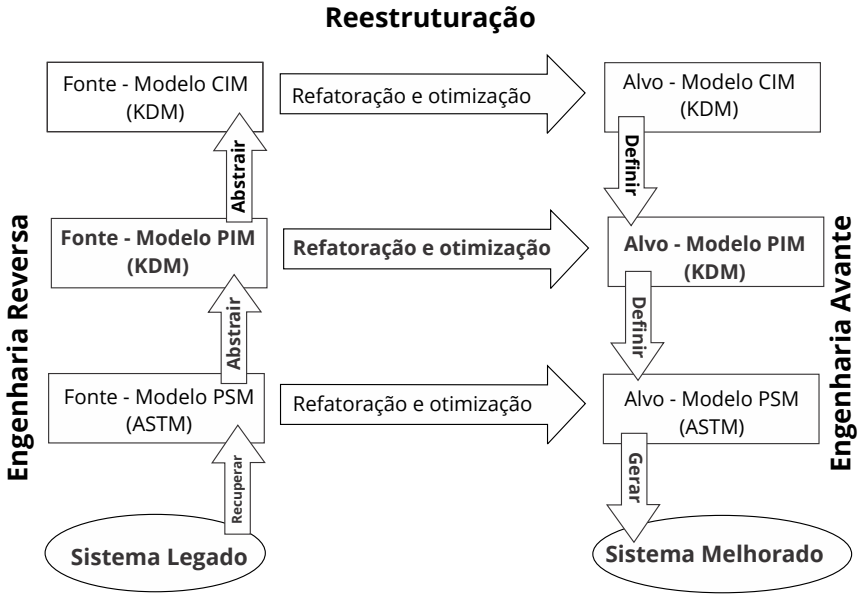


Figura 2.2: Modelo de Modernização Ferradura Figura adaptada de ADM (2015).

O modelo ferradura contém todas as fases tradicionais da MDA, que geram modelos em diferentes níveis de abstração. As fases tradicionais da ADM são:

- **Engenharia Reversa**: nesta fase a engenharia reversa é realizada. A entrada é dada por meio de um sistema legado a ser modernizado, então o conhecimento é extraído e o Modelo Específico de Plataforma (*Platform Specific Model* - PSM) é gerado. O modelo serve como base para a geração de um Modelo Independente de Plataforma (*Platform Independent Model* - PIM), que é chamado de KDM(ver Seção 2.4).

- **Reestruturação**: nesta fase um conjunto de reestruturações/refatorações pode ser aplicado no modelo KDM gerado por meio das transformações M2M.

- **Engenharia Avante**: nesta fase é realizada a engenharia avante e o código-fonte do sistema alvo modernizado é gerado.

2.4 Metamodelo de Descoberta de Conhecimento - KDM (*Knowledge Discovery Metamodel*)

Metamodelo de Descoberta de Conhecimento (KDM, *Knowledge Discovery Metamodel*) é um metamodelo para representação de sistemas de software, seus elementos, associações e ambiente operacional. De acordo com Perez-Castillo, Guzman e Piattini (2011), o objetivo do KDM é ser um metamodelo que permite ao engenheiro de software criar ferramentas para trocar metadados da aplicação por meio de diferentes aplicações, linguagens, plataformas e ambientes. Mais especificamente, a ADM (2015) especifica que os objetivos do KDM são:

- Representar artefatos de sistemas legados como entidades, relacionamentos e atributos;
- Incluir artefatos externos que realizem interação com os artefatos do software;
- Suportar uma variedade de plataformas e linguagens;
- Disponibilizar um núcleo independente de plataforma e linguagem, com extensões quando necessário;
- Definir uma terminologia unificada para artefatos de sistemas legados;
- Descrever a estrutura física e lógica de sistemas legados;
- Agregar ou modificar um sistema legado, i.e., refatorar;
- Facilitar o rastreamento de artefatos da estrutura lógica para a estrutura física; e
- Representar o comportamento dos artefatos.

O KDM consiste em quatro camadas de abstração, como é mostrado na Figura 2.3 (destacado em cinza). O nome das camadas são: Infraestrutura (`InfrastructureLayer`) ❶, Elementos do Programa (`ProgramElementsLayer`) ❷, Tempo de Execução (`RuntimeResourceLayer`) ❸, e Abstrações (`AbstractionsLayer`) ❹. Ainda na Figura 2.3, é possível ver que essas camadas interagem uma com a outra. Como consequência, se alguma mudança acontecer em uma camada específica, ela será propagada para as outras camadas.

Cada camada é organizada em pacotes. Mais especificamente, o KDM contém 12 pacotes, em que cada um deles é composto por um ou mais diagramas de classe e define um conjunto de elementos do metamodelo no qual o propósito é representar uma parte independente do conhecimento relacionado a sistemas de software existentes. A camada de infraestrutura ❶

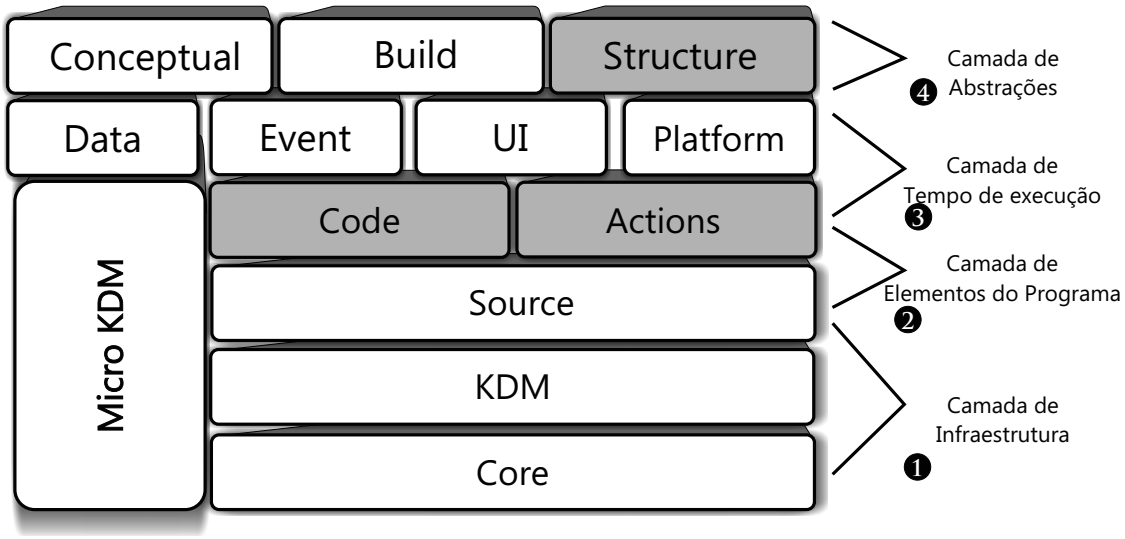


Figura 2.3: Camadas, pacotes e separação de interesses no KDM

consiste nos seguintes pacotes: `Core`, `KDM` e `Source`. Esses pacotes definem elementos do metamodelo comuns que constituem a infraestrutura para outros pacotes. Por exemplo, o pacote `Source` define o `InvetoryModel`, que enumera os artefatos de um sistema de software existente e define o mecanismo de rastreabilidade entre os elementos e suas representações originais no código-fonte do sistema.

A Camada de Elementos de Programa ❷ consiste nos pacotes `Code` e `Action`. Esses pacotes coletivamente definem o modelo `CodeModel`, que representa artefatos do nível de implementação de um sistema de software existente, determinado pela linguagem de programação utilizada. O pacote `Code` possui o foco nos itens do código-fonte e nos relacionamentos básicos entre eles. O pacote `Action` diz respeito as descrições de comportamento e controle dos relacionamentos de fluxo de dados entre eles. O pacote `Action` é estendido por outros pacotes KDM para abstrair a descrição de comportamentos de alto nível que são elementos chave no conhecimento sobre sistemas de software.

A Camada de Recursos em Tempo de Execução ❸ consiste em quatro pacotes. O `Platform` define recursos utilizados em tempo de execução pelo sistema, por exemplo, relacionamentos determinados em tempo de execução. O pacote `UI` define elementos do metamodelo que representa aspectos da interface com o usuário. O pacote `Event` representa aspectos dirigidos a eventos, tal como estados, transições, entre outros. Finalmente, o pacote `Data` define elementos que representam aspectos de persistência de dados em sistemas de software.

A Camada de Abstrações ❹ consiste nos seguintes pacotes: `Conceptual`, `Build` e `Structure`. Mais especificamente, o pacote `Structure` define elementos que representam componentes arquiteturais, tais como subsistemas, camadas, pacotes, entre outros. Além disso, é capaz de

definir a rastreabilidade desses elementos para outros elementos KDM do mesmo sistema. O pacote `Conceptual` representa elementos de domínio específico do sistema de software e o pacote `Build` define elementos que representam artefatos relacionados ao processo de construção de sistemas de software. Nas próximas subseções, são mostrados os principais pacotes que compõem o contexto para o desenvolvimento deste trabalho.

2.4.1 Pacote Code

O objetivo do pacote `Code` é representar por meio de seus elementos, chamados de metaclasse, unidades de um programa em nível de implementação em conjunto com as suas respectivas associações. As metaclasses nele existentes são responsáveis por representar os elementos de programa que são comuns em linguagens de programação distintas, como exemplo, são citados os tipos de dados, classes, *templates*, macros e protótipos.

Em uma instância KDM dada, cada elemento do pacote `Code` representa alguma construção em uma linguagem de programação utilizada no sistema. Na Figura 2.4 o `CodeModel`[1] é retratado e representa partes do pacote `Infraestructure`. A metaclasse `CodeModel` ❶ é um modelo que possui coleções de fatos sobre o sistema de software, correspondentes ao domínio `Code`. Ela possui uma associação `codeElement` : `AbstractCodeElement`$[0..*]$, que significa a possibilidade de arranjo com outros elementos de código, por exemplo, métodos, atributos, entre outros. O `AbstractCodeRelationship` ❷ representa qualquer relacionamento determinado por uma linguagem de programação. A metaclasse `ComputationalObject` ❸ descreve objetos computacionais em tempo de execução, por exemplo, métodos e variáveis.

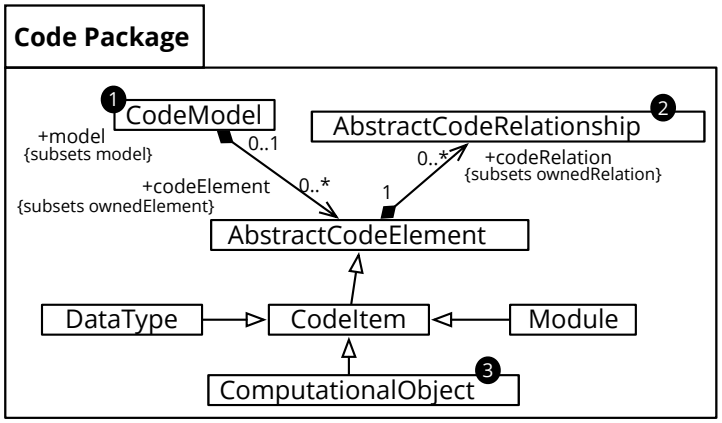


Figura 2.4: Diagrama de classes - `CodeModel` (Adaptado do grupo OMG (ADM, 2015))

O pacote `Code` consiste em 24 metaclasses e contém todos os elementos abstratos para mo-

[1] O diagrama de classes do `CodeModel` mostrado só representa o conjunto de metaclasses e os seus respectivos relacionamentos lógicos. Para informações completas, verifique o **KDM Specification (OMG, 2013)**.

Tabela 2.1: Elementos de Código-fonte e suas respectivas metaclasses KDM

Elemento em código-fonte	Metaclasse KDM
Interface	`InterfaceUnit`
Classe	`ClassUnit`
Atributo	`StorableUnit`
Método	`MethodUnit`
Variável Local	`MemberUnit`
Parâmetro	`ParameterUnit`
Associação	`KDMRelationship`

delar estruturas estáticas do código-fonte. Na Tabela 2.1 são identificadas as metaclasses KDM referentes a estruturas similares ao código-fonte. Algumas metaclasses podem ser diretamente mapeadas, como uma `Class` referente a linguagens orientadas a objetos pode ser facilmente mapeada para uma `ClassUnit` do KDM.

Como afirmado anteriormente, uma `ClassUnit` representa classes definidas por usuários de linguagens orientadas a objeto. Uma `ClassUnit` pode conter uma coleção de elementos que é chamado de `CodeItem`, por exemplo, `StorableUnit` ou `MethodUnit`. A metaclasse `ClassUnit` contém um meta atributo `isAbstract` : `Boolean` utilizado para especificar se uma classe é ou não abstrata. `ClassUnit` também possui uma meta associação chamada de `codeElement` : `CodeItem`$[0..*]$, que é utilizada para agrupar todos os membros da classe, por exemplo, construtores, métodos, entre outros.

Similarmente, a metaclasse `InterfaceUnit` representa o conceito comum a várias linguagens de programação. Ela é uma subclasse de `DataType`, assim como `ClassUnit`. `InterfaceUnit` também possui uma metaassociação chamada `codeElement` : `CodeItem`$[0..*]$ que representa tipos de dados tal como `MethodUnit`.

`StorableUnit` representa uma variável em um sistema de software. Ele é usado para representar as variáveis globais e locais e contém um meta-atributo `String` usado para definir o nome das variáveis. `StorableUnit` também tem a associação `type` : `DataType`$[1]$ usada para especificar o tipo da variável e uma enumeração `kind` : `StorableUnit`, que descreve várias propriedades comuns de um `StorableUnit` relacionado ao seu ciclo de vida, à visibilidade e ao tipo de memória.

`MethodUnit` representa as funções possuídas por uma `ClassUnit` ou `InterfaceUnit`. Ele também é usado para representar os operadores definidos pelo usuário, construtores e destrutores. Possui como meta-atributos `name` : `String`, `kind` : `MethodKind` e `export` : `ExportKind`. O primeiro é usado para descrever o nome de um método. O segundo meta-atributo é uma enumeração que define especificações adicionais do tipo de método, ou seja, é possível especificar

se a instância do método é um construtor, destrutor, etc. O último representa a visibilidade do método, ou seja, público, privado e protegido.

A fim de entender completamente como o KDM é usado para representar o código-fonte de um programa específico, na Listagem 2.1 mostra-se um exemplo simplificado em Java. O KDM simplificado correspondente é mostrado na Figura 2.5. Por questões de simplicidade, é possível notar que este diagrama representa o código-fonte como uma árvore de nós contendo metaclasses de alguns elementos KDM. Como pode ser visto na Figura 2.5, a metaclasse raiz é a `Segment`, que é um recipiente para um conjunto significativo de fatos sobre um sistema de software existente. Cada `Segment` pode incluir uma ou mais instâncias de modelos KDM, como `CodeModel` e `StructureModel`.

Analisando tanto a Listagem 2.1 quanto a Figura 2.5 é evidente que cada estrutura estática do código-fonte tem uma metaclasse em KDM para representá-la. Por exemplo, o modelo de pacote na Linha 1 da Listagem 2.1 ❶ é representada em KDM pela metaclasse chamada `package`, como visto na Figura 2.5 ❶. Além disso, a classe `Car` é declarada de acordo com a Listagem 2.1 ❷. Por sua vez, herda da classe `Vehicle`, no Java isso é feito usando a palavra-chave `extends` seguida do nome de uma classe, conforme a Listagem 2.1 ❸ e ❹. A metaclasse `Extends` representa a herança em modelos KDM. Como mostrado na Figura 2.5 ❸ a metaclasse `Extends` possui associações `to` e `from`. O primeiro representa a classe pai (superclasse) e o último a classe filha (subclasse). Neste contexto, a classe `Vehicle` é classe pai de `Car`, como mostrado na Figura 2.5 ❹. Final

mente, a variável `name` e o método `getName()` (consulte a Listagem 2.1 ❺ e ❻) são mapeados para os correspondentes elementos do KDM, `StorableUnit` e `MethodUnit` (ver Figura 2.5 ❺ e ❻).

[!ht]

```
1    ❶ package modelo;
2    ❷ public class Funcionario ❸
3    ❹ extends Pessoa{
4    ➥❺ private String nome;
5    ➥❻ public String getNome(){
6    ...
7    }
8    }
```

Código 2.1: Exemplo de código Java.

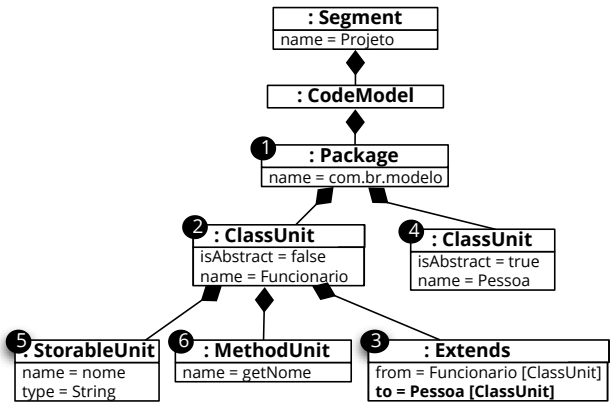


Figura 2.5: KDM referente a Listagem 2.1

2.4.2 Pacote Action

O pacote `Action` define um conjunto de metaclasses cujo propósito é o de representar descrições de comportamento em nível de implementação estabelecido por linguagens de programação, por exemplo, declarações, operadores, condições, *features*, bem como as suas associações, por exemplo, fluxo controle e de dados. A Figura 2.6 ❶ mostra o pacote `Action` e algumas de suas metaclasses.

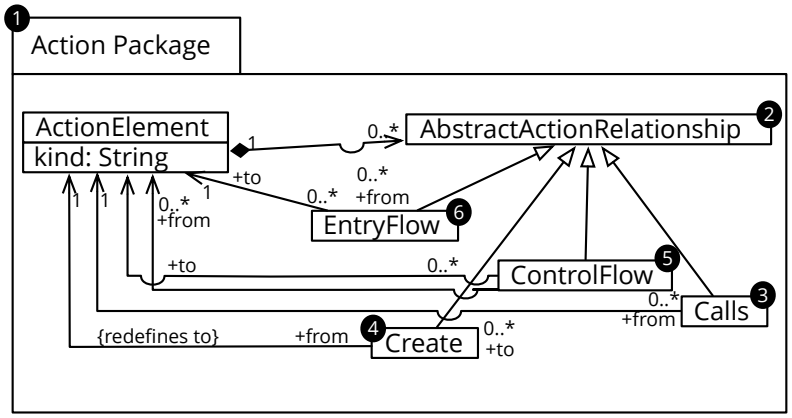


Figura 2.6: Diagrama de classes pacote `Action` (adaptado de ADM (2015))

O pacote `Action` é composto por 11 diagramas de classe e depende dos pacotes `Core`, `KDM` e `Source` e principalmente, do pacote `Code`. O pacote `Action` se desvia de um padrão uniforme para os modelos KDM porque ele não define um modelo KDM separado, mas é estendido a partir do pacote `Code`, que é apresentado na Subseção 2.4.1. Juntos, os pacotes `Code` e `Action` constituem a Camada de Elementos de Programa, como mostrado na Figura 2.3. A metaclasse `AbstractionActionRelationship`, apresentada na Figura 2.6 ❷, é a classe pai usada para representar várias relações KDM que se originam a partir de um `ActionElement`. A metaclasse `AbstractionActionRelationship` possui metaclasses específicas. Algumas delas estão representadas na Figura 2.6, por exemplo, as metaclasses `Calls` ❸, `Creates` ❹, `ControlFlow` ❺ e `EntryFlow` ❻.

O relacionamento `Calls` corresponde a uma chamada para um procedimento, um método estático, um método não-estático de uma instância particular de um objeto, um método virtual, ou um elemento de interface. `Calls` possui duas associações, são elas: `ActionElement`[1] e `ControlElement`[1]. O primeiro representa o elemento de ação a partir do qual a relação chamada origina-se, enquanto que a segunda associação representa o alvo.

A metaclasse `Creates` representa uma associação entre um elemento de ação que cria uma nova instância de um determinado elemento de dados para o tipo de dados correspondente de acordo com a semântica da linguagem de programação do sistema de software existente. Ele

também possui duas associações: `ActionElement`[1] e `DataType`[1]. Similarmente à metaclasse `Calls`, a primeira associação representa o elemento que possui o relacionamento. A segunda ilustra o elemento de dados que é instanciado pelo `ActionElement`.

O `ControlFlow` é um elemento de modelagem genérica que representa relação de fluxo de controle entre dois `ActionElements`. Além disso, é uma subclasse com elementos de modelagem mais específicos. O `EntryFlow` é um elemento de modelagem que representa um fluxo inicial de controle em um elemento KDM. O relacionamento `EntryFlow` é usado de uma maneira uniforme para descrever os pontos de entrada para outros elementos de código KDM.

2.4.3 Pacote Structure

A ADM (2015) define metaclasses que representam componentes da arquitetura de sistemas de software existentes, como subsistemas, camadas, componentes, etc., e definem a rastreabilidade desses elementos para outros em uma mesma instância do sistema, por meio do pacote `Structure`. As visões de arquitetura com base no ponto de vista definido pelo pacote `Structure` representam a forma como os elementos estruturais do sistema de software estão relacionados com os módulos definidos em código, que correspondem ao pacote `Code`. Na Figura 2.7, é mostrado o pacote `Structure` em forma de diagrama de classes.

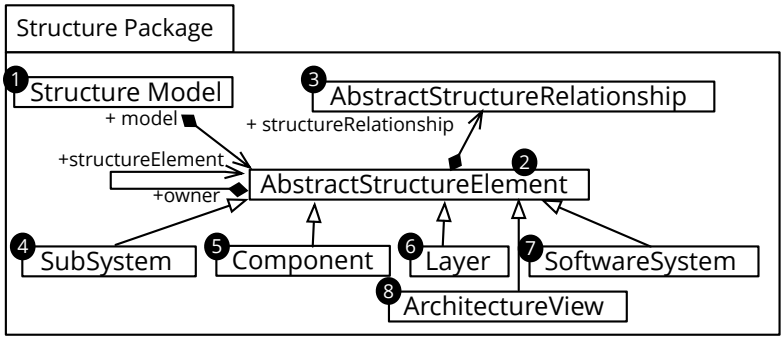


Figura 2.7: Diagrama de classes do pacote `Structure` (Adaptado do Grupo OMG (ADM, 2015))

Por meio da utilização das suas metaclasses é possível relacionar todos os elementos estruturais do sistema com os seus respectivos elementos computacionais, em outras palavras, pode-se especificar os elementos estruturais do sistema. Na Figura 2.7 é mostrado que o pacote `Structure` e suas metaclasses são usadas em combinação com os pacotes `Code`, `Data`, `Platform`, `UI` e `Inventory`.

O pacote `Structure` possui uma coleção de elementos estruturais, representado pela metaclasse `AbstractStructureElement`[0..$*$] ❷. Ela representa a parte arquitetural relacionada à organização do sistema de software em módulos e possui quatro associações. A primeira as-

sociação representa os elementos pertencentes ao modelo e é chamada de `structureElement` : `AbstractStructureElement`$[0..*]$. Essa metaclasse pode ser instanciada por meio das metaclasses: `SubSystem` ❹, `Component` ❺, `Layer` ❻, `SoftwareSystem` ❼ e `ArchitectureView` ❽. Cada uma delas representa um elemento arquitetural.

A metaclasse: `AbstractStructureRelationship` ❸ é usada para representar todos os relacionamentos em nível arquitetural, enquanto a metaclasse `AggregatedRelationship`$[0..*]$ representa uma relação abstrata entre dois elementos do KDM, na qual é possível definir relações concretas. Além das metaclasses mencionadas, também é feita uma associação chamada de `implementation` : `KDMEntity`$[0..*]$, que é a ligação entre um : `AbstractStructureElement`$[0..*]$ e um `KDmEntity`, que é usada para especificar elementos computacionais (do pacote `Code`, ou seja, `Package`, `ClassUnit`, `InterfaceUnit`, etc.) que representam o elemento estrutural.

Na Figura 2.8 é descrita uma arquitetura hipotética mostrada para ilustrar como o KDM pode ser utilizado para representar elementos arquiteturais. Todos os elementos arquiteturais são representados com a seguinte padronização: o nome da metaclasse que representa os elementos arquiteturais, ':' seguido pelo seu nome.

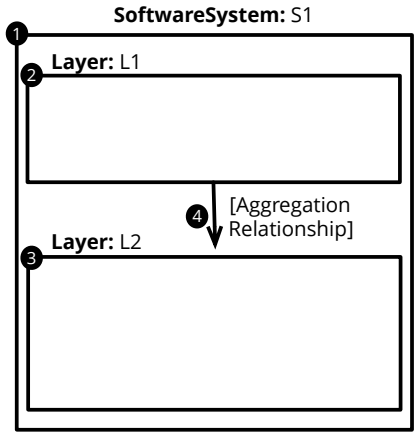


Figura 2.8: Representação de uma arquitetura

O primeiro elemento representado é um `SoftwareSystem` ❶, dentro dele, existem duas metaclasses do tipo `Layer` (L1 ❷ e L2 ❸) que representam camadas dentro do sistema de software. Por fim, a metaclasse `AggregatedRelationship` ❹ é responsável por mostrar o relacionamento entre as camadas L1 e L2.

O correspondente simplificado da instância KDM é mostrado na Figura 2.9. Todos os elementos arquiteturais são subclasses de `StructureModel`. As camadas são representadas pela metaclasse `Layer`, como pode ser visto na Figura 2.9 ❷ e ❸. A metaclasse mais importante a se destacar nesta figura é `AggregatedRelationship`, que representa a relação entre a `Layer` `L1` e a `Layer` `L2`. Ela possui meta-atributos que têm como objetivo fornecer infor-

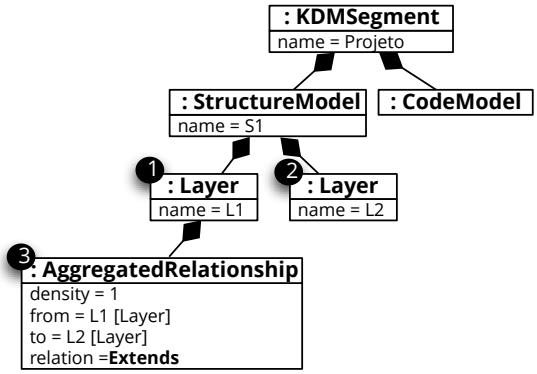


Figura 2.9: Instância KDM correspondente a Figura 2.8

mações sobre o relacionamento. Por exemplo, o meta-atributo `density` ilustra o número de relações primitivas entre estas camadas. Na Figura 2.9, o meta-atributo `density` possui o valor 3 (quatro). Outros dois meta-atributos são o `from` e o `to`, que representam os elementos arquiteturais de origem e destino, respectivamente. Eles são usados para especificar que a `Layer L1` em `SoftwareSystem S1` pode acessar a `Layer L2` também em `SoftwareSystem S1` de alguma forma. Finalmente, o meta-atributo `relation` representa como a `Layer L1` pode acessar a `Layer L2`, neste contexto, por das metaclasses `Extends`, `Calls`, e `Implements`. Na Figura 2.10 é mostrado em detalhes como as relações entre dois elementos arquiteturais são considerados neste estudo.

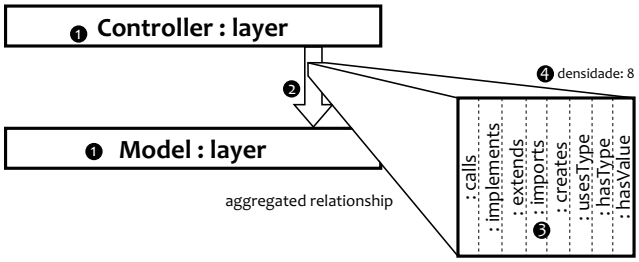


Figura 2.10: Relacionamento entre dois elementos arquiteturais

Primeiramente os elementos arquiteturais ❶ são apresentados por meio da representação das camadas `Controller` e `Model`. Como visto anteriormente, um relacionamento em nível arquitetural acontece entre dois elementos em nível estrutural (camadas, componentes, subsistemas, etc). Em seguida, um `aggregatedRelationship` contendo todos os possíveis relacionamentos entre dois elementos (chamadas de métodos, herança, etc.) é mostrado ❸. A seta ❷ representa o fluxo de entrada e saída dos relacionamentos. Em outras palavras, esses relacionamentos representam as possíveis permissões de acesso iniciando na camada `Controller` para a camada `Model`. Finalmente, a densidade ❹ é definida como oito, que representa o número total de relacionamentos dentro da metaclasse.

Atualmente, MoDisco, idealizada por Bruneliere et al. (2010), é a ferramenta mais madura

e disseminada para dar suporte a geração de instâncias KDM a partir de código-fonte Java. Ela permite realizar engenharia reversa em aplicações por meio de um *Discoverer*, que analisa o código-fonte Java e gera instâncias KDM em nível de código. Observa-se que, naturalmente, MoDisco não é capaz de gerar pacotes de nível superior, pois a maioria deles requer interação do usuário para fornecer as informações necessárias. Por exemplo, não é possível para o MoDisco adivinhar que específicos pacotes Java representam uma camada chamada Modelo; tal informação deve ser fornecida por um usuário. A fim de atacar tais limitações, criou-se o ArchKDM, no Capítulo 4, a qual é apresentada com mais detalhes.

2.5 Considerações Finais

Neste capítulo, foram apresentados os principais conceitos para o desenvolvimento deste trabalho. Dentre eles, foram apresentados os conceitos de arquitetura de software e suas formas de representação, conceitos fundamentais para a especificação e recuperação de arquiteturas de software. Em seguida, a ideia-chave no contexto deste trabalho foi apresentada, que é a Checagem de Conformidade Arquitetural, na qual são realizadas comparações entre duas versões de arquitetura, uma contendo a especificação da arquitetura planejada e outra com a arquitetura atual recuperada. Por fim, como este trabalho trata da realização da checagem por meio de modelos, também foram apresentados a Modernização Apoiada por Modelos e o metamodelo KDM. O primeiro explica os principais conceitos para a realização de modernizações utilizando modelos. O segundo é responsável por representar todos os artefatos utilizados neste trabalho, tais como: representação em modelo das arquiteturas planejadas e atuais, além do resultado da checagem de conformidade.

Capítulo 3

TRABALHOS RELACIONADOS

Neste capítulo são apresentados os trabalhos encontrados na literatura que foram considerados diretamente relacionados ao tema deste projeto. Pesquisadores têm proposto abordagens para Checagem de Conformidade Arquitetural com base em diferentes modelos, dos quais foram divididos nos quatro grupos seguintes: i) abordagens CCA baseadas em AST; ii) abordagens CCA baseadas em Grafos; iii) abordagens CCA baseadas em MDE; e iv) outras abordagens CCA. Portanto, cada seção deste capítulo apresenta uma categoria relacionada ao projeto de pesquisa proposto. Na Seção 3.1, são apresentadas as abordagens baseadas em AST (*Abstract Syntax Tree*). Na Seção 3.2, são apresentados os trabalhos baseados em grafos. Na Seção 3.3, as abordagens baseadas em modelos são apresentadas. Na Seção 3.4 são apresentadas as abordagens para CCA baseadas em outras formas de representação. Por fim, na Seção 3.5, as considerações finais sobre os trabalhos relacionados são apresentadas.

3.1 Abordagens CCA baseadas em AST

DCL (Terra e Valente (2009)), ArchJava (Aldrich, Chambers e Notkin (2002)), e ArchLint (Maffort et al. (2015)) dependem da AST (Árvore sintática abstrata - **Abstract Syntax Tree**) como o modelo subjacente para a realização da checagem de conformidade arquitetural. DCL (Terra e Valente (2009)) é uma linguagem de domínio específico, declarativa e estaticamente verificável que permite definir restrições estruturais entre módulos. O objetivo da DCL é mostrar ao engenheiro de software violações estruturais que contribuem para a erosão arquitetural do sistema. DCL permite capturar divergências por meio de dependências do tipo podem (*only can* e *can-only*) ou não podem (*cannot*). Além disso, DCL também captura ausências por meio do tipo deve (*must*). Essas dependências são capturadas entre módulos, que são, de maneira simplificada, um conjunto de classes. Para utilizar a DCL, o arquiteto de software

deve descrever a arquitetura do sistema em dois passos: declaração de módulos e declaração de dependências. Para declarar um módulo é necessário que o arquiteto utilize a palavra chave *module* definida na DCL, seguida do nome que irá identificar o módulo e depois dos elementos pertencentes ao módulo definido.

ArchJava (Aldrich, Chambers e Notkin (2002)) é uma extensão Java que une arquitetura e implementação. ArchJava realiza construções de modelagem arquitetural que unificam a arquitetura de software com a aplicação, garantindo que a implementação está em conformidade com as restrições arquiteturais. Além disso, ArchJava garante a integridade da comunicação entre arquitetura e implementação em qualquer estágio do ciclo de vida de um software.

ArchLint (Maffort et al. (2015)) foi a primeira ferramenta de verificação de conformidade arquitetural que depende da combinação de análise estática e histórica de código-fonte. Como resultado, é fornecida uma abordagem para a verificação de conformidade arquitetural que não necessita de refinamentos sucessivos em modelos arquiteturais de alto nível e também não requer a especificação de uma extensa lista de restrições arquiteturais, como acontece com linguagens específicas de domínio. Por outro lado, *ArchLint* pode gerar resultados falsos positivos, comuns na maioria das ferramentas de busca de erros com base na análise estática.

Esses três estudos compartilham a mesma fraqueza. Embora alcancem bons níveis durante a realização da checagem de conformidade arquitetural, eles não suportam múltiplas linguagens, estilos arquiteturais e hierarquia entre os elementos arquiteturais explicitamente definida, como ArchKDM é capaz de fazer.

3.2 Abordagens CCA baseadas em Grafos

ConQAT (Deissenboeck et al. (2010)), SAVE (Knodel et al. (2006), Duszynski, Knodel e Lindvall (2009)) e SotoArch (hello2morrow (2015)) realizam a checagem de conformidade arquitetural baseada em grafos como modelos subjacente. ConQAT (Deissenboeck et al. (2010)) identifica divergências e ausências com base na comparação da arquitetura planejada e o conhecimento das dependências, automaticamente extraídas do código fonte. ConQAT não é limitado a uma única linguagem ou a um único tipo de dependência. A arquitetura planejada é especificada por meio de um editor gráfico que também permite o mapeamento entre os elementos arquiteturais e os artefatos do sistema.

Com base em conceitos puros de modelos de reflexão, SAVE (Knodel et al. (2006), Duszynski, Knodel e Lindvall (2009)) destaca relações do tipo convergente, divergente e ausentes entre o modelo de alto nível e o modelo de código-fonte, que também são extraídas automaticamente a

partir do código fonte. O mapeamento é realizado manualmente entre elementos estruturais de alto nível e os de código fonte.

SotoArc (hello2morrow (2015)) fornece meios para visualizar e compreender a estrutura estática de um sistema de software, incluindo a modelagem de sua arquitetura. Em SotoArc, não há informações suficientes no código-fonte para detectar as estruturas arquiteturais automaticamente. Por esta razão, SotoArc possibilita a definição de modelos arquiteturais que descrevem os elementos estruturais de um sistema de software, tais como arquivos, pacotes e diretórios. Tais elementos podem ser combinados com entidades arquiteturais que possuem um maior nível de abstração, os quais, por fim, podem se relacionar.

Embora essas ferramentas sejam completas e precisas, elas dependem de modelos proprietários para representar a arquitetura planejada. ArchKDM, por outro lado, depende de um metamodelo ISO (KDM) para representar as arquiteturas atuais e planejadas. Isso significa que os pesquisadores que estão familiarizados com KDM podem desenvolver e melhorar qualquer um dos passos do ArchKDM, por exemplo, a implementação de um algoritmo mais sofisticado para a extração da arquitetura atual ou realizar refatorações de alto nível para as violações identificadas.

3.3 Abordagens CCA baseados em MDE

ArcConf (Abi-Antoun e Aldrich (2008)), ReflexML (Adersberger e Philippsen (2011)) e a abordagem de Herold e Rausch (2013) utilizam modelos MDE para realizarem a checagem de conformidade arquitetural. ArcConf (Abi-Antoun e Aldrich (2008)) gera uma visão de conformidade e calcula métricas entre os componentes e conectores. Eles uniformemente representam várias linguagens na forma de metamodelo. As regras são definidas para elementos de alto nível, o padrão MOF (**Meta-Object Facility**) é utilizado para representar o metamodelo e a linguagem OCL (**Object Constraint Language**) é utilizada para representar as regras. Em seguida, os modelos são integrados em um único. Por fim, as regras são especificadas de acordo com as expressões OCL, que avalia a conformidade entre código fonte e arquitetura.

ReflexML (Adersberger e Philippsen (2011)) define a rastreabilidade dos modelos por meio de componentes UML para o código fonte usando expressões orientadas a aspectos. A maior contribuição do estudo foi um mapeamento eficiente de arquitetura para elementos de código que é mais expressiva do que as abordagens tradicionais em AOP. Outra contribuição do trabalho foi a realização de checagem consistentes. Essas checagens são baseadas no modelo de componentes UML e nos princípios de arquitetura de software. Além disso, as checagens per-

mitem detectar uma grande diversidade de violações arquiteturais apenas utilizando estereótipos em componentes UML em conjunto com expressões para modelos de reflexão.

Herold e Rausch (2013) realizam a checagem de conformidade arquitetural por meio de regras arquiteturais em conjunto com modelos UML mapeados para código fonte. A representação do conhecimento é utilizada com um sistema de software para verificar se as regras de conformidade estão satisfeitas para um determinado conjunto de modelos.

Embora as abordagens baseadas em MDE promovam a reutilização, elas não representam fielmente os detalhes de implementação. ArchKDM, no entanto, depende de KDM, que fornece uma especificação completa dos elementos arquiteturais e permite que elementos de código fonte sejam representados com precisão um-para-um.

3.4 Outras abordagens

LDM (Sangal et al. (2005)) depende de Matrizes de Dependência Estrutural (DSMs) para realizar a checagem de conformidade arquitetural. A DSM é uma matriz quadrada cujas linhas e colunas denotam classes de um sistema orientado a objetos. O número de referências que `B` contém em `A` denota que a classe `B` depende da classe `A`, o que pode ser representado pela célula $(\texttt{A},\texttt{B})$. Embora DSM seja importante para fins de documentação e comunicação com as partes interessadas, a DSM não é uma especificação de arquitetura, que é independente da implementação do sistema. O principal objetivo da LDM é revelar padrões arquiteturais e detectar dependências que possam indicar violações arquiteturais. Para realizar a CCA, LDM provê uma linguagem simples para declarar regras de projeto que devem ser seguidas pela implementação, em outras palavras, LDM utiliza regras do tipo **can-use** e **cannot-use**.

Na linha de pesquisa de análise dinâmica, DiscoTect (Yan et al. (2004)) monitora dinamicamente sistemas de software em execução para obter a sua arquitetura. Assim, os arquitetos podem explorar as regularidades na implementação e o conhecimento de estilos arquiteturais para criar um mapeamento que pode ser aplicado em qualquer sistema que esteja em conformidade com as convenções de implementação, com o objetivo obter uma visão da arquitetura.

Da mesma forma, ConArch (Ciraci, Sozer e Tekinerdogan (2012)) é uma abordagem para verificação em tempo de execução que realiza a detecção de inconsistências entre o comportamento dinâmico da arquitetura documentada e o comportamento em tempo de execução do sistema. ConArch converte automaticamente chamadas padrões de código fonte em fatos Prolog. O usuário provê mapeamento entre o modelo arquitetural e o código fonte por meio de regras Prolog. ConArch executa essas regras e gera uma especificação em forma de máquina de

estados em conjunto com o monitoradores em tempo de execução.

No entanto, esses estudos compartilham o mesmo problema em comparação com abordagens estáticas; abordagens não KDM neste caso. Mapeamentos entre observações de baixo nível de sistemas de software e eventos arquiteturais normalmente não são um-para-um, portanto, não é simples indicar padrões de implementação que representam a arquitetura planejada.

3.5 Considerações Finais

Nesse capítulo foram apresentados trabalhos relacionados ao projeto proposto divididos em quatro categorias: i) abordagens CCA baseadas em AST; ii) abordagens CCA baseadas em Grafos; iii) abordagens CCA baseadas em MDE; e iv) outras abordagens CCA.

As abordagens apresentadas baseadas em AST possuem resultados positivos na realização da checagem de conformidade arquitetural, porém não suportam diversas funcionalidades existentes em ArchKDM, tais como estilos arquiteturais, múltiplas linguagens e hierarquia explícita entre elementos arquiteturais.

As abordagens CCA baseadas em grafos são completas e também alcançaram bons níveis durante a verificação, porém trabalham sobre modelos proprietários na representação da arquitetura planejada enquanto que ArchKDM trabalha com o metamodelo KDM em todas as suas representações. Pesquisadores que conhecem KDM podem adaptar e/ou melhorar qualquer uma das etapas da checagem de conformidade.

As abordagens CCA baseadas em MDE possuem diversos pontos fontes, dentre eles, a reutilização. Porém, um dos seus maiores problemas está no fato da incapacidade em representar detalhes de implementação. Por outro lado, KDM fornece metaclasses para a representação completa de um código fonte, além de possuir um nível de abstração capaz de lidar com a arquitetura de software.

Por fim, as outras abordagens CCA apresentadas neste trabalho possuem como principal ponto negativo a dificuldade na representação de arquiteturas planejadas, pois o mapeamento de observações de sistemas de software em baixo nível não são um-para-um em relação ao código fonte.

Vale ressaltar que todos os trabalhos relacionados descritos nesse capítulo não apresentam um estudo sobre a viabilidade de realização da checagem de conformidade arquitetural no contexto da Modernização Dirigida por Modelos (ADM), que é uma área nova de pesquisa que utiliza os princípios da MDA e por isso herda as suas características que podem ajudar na con-

dução de processos de modernização, tal como portabilidade e reúso.

Capítulo 4

ARCHKDM - CHECAGEM DE CONFORMIDADE ARQUITETURAL

Neste capítulo é apresentado todo o contexto para a realização da checagem de conformidade arquitetural. Na Seção 4.1, é apresentada a visão geral da abordagem proposta. Na Seção 4.2, a linguagem de descrição arquitetural DCL-KDM é mostrada, além disso, um exemplo de utilização por meio da descrição de uma arquitetura e instanciação do pacote `Structure` são realizados. Na Seção 4.3, a abordagem para extração da arquitetura de um sistema é descrita. Na Seção 4.4, são mostrados os passos para a realização da checagem de conformidade arquitetural. Por fim, na Seção 4.5, as considerações finais sobre este capítulo são apresentadas.

4.1 Visão Geral

Como discutido no Capítulo 2, em geral, arquiteturas de sistemas legados apresentam inconsistências em relação às arquiteturas planejadas. Uma alternativa para esse problema é conduzir uma modernização apoiada por modelos por meio de uma checagem de conformidade arquitetural. Dentre os metamodelos disponibilizados pela ADM, o principal é o KDM, que objetiva representar um sistema legado em vários níveis de abstração, inclusive o arquitetural.

Para a realização de uma modernização seguindo os conceitos da ADM, é preciso um fluxo de requisitos que é dividido em três fases: engenharia reversa, refatoração e engenharia avante. No contexto deste trabalho, a checagem de conformidade arquitetural é realizada no nível PIM da Engenharia Reversa.

O processo checagem de conformidade no contexto desse trabalho ocorre durante a fase de engenharia reversa. A arquitetura planejada é obtida por meio da descrição arquitetural feita

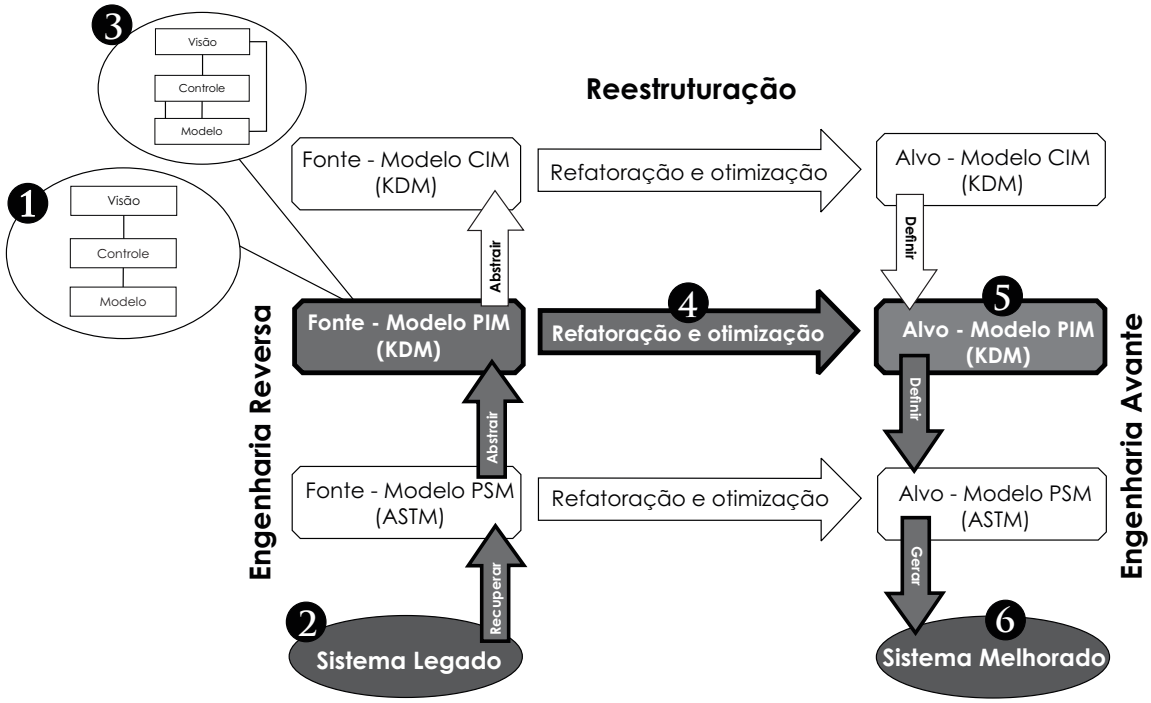


Figura 4.1: Modernização no contexto da ADM

na DCL-KDM (ver Seção 4.2). O resultado é uma instância do metamodelo KDM em nível arquitetural ❶ (Figura 4.1). A arquitetura atual é provida por meio de transformações, na qual é gerada uma instância do KDM a partir do sistema legado ❷ (Figura 4.1) que representa todo o sistema, em seguida, os elementos arquiteturais presentes na arquitetura planejada são adicionados e mapeados com os elementos de código fonte, que juntos irão gerar uma nova instância do pacote estrutural ❸ (Figura 4.1) com as relações presentes entre estes elementos. A checagem é realizada ao final da fase de engenharia reversa por meio de comparação entre as duas instâncias do KDM, conforme mostrado na Figura 4.2. Para que seja conduzida uma modernização de maneira completa, os resultados obtidos pelo processo de checagem de conformidade podem ser utilizados na fase de refatoração do processo de modernização ❹ (Figura 4.1), e em seguida, a instância KDM pode ser transformada novamente para código fonte (❺ e ❻ (Figura 4.1)), correspondendo a fase de engenharia avante.

Na instância que representa a arquitetura planejada do sistema ❶ (Figura 4.2) estão contidos os elementos arquiteturais e as relações permitidas entre eles, enquanto que, na instância que representa a arquitetura atual ❷ (Figura 4.2), são mostrados os elementos arquiteturais mapeados com elementos de código e as relações reais que acontecem entre eles. As duas instâncias são comparadas e as diferenças entre elas são adicionadas a uma terceira instância ❸ (Figura 4.2).

A abordagem implementada para a checagem de conformidade arquitetural no contexto da ADM é chamada de ArchKDM. O principal objetivo é apresentar violações arquiteturais entre

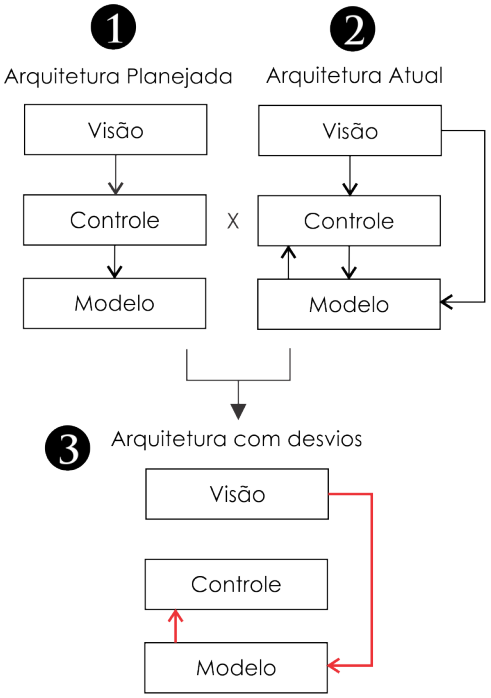


Figura 4.2: Checagem de Conformidade Arquitetural

um modelo da arquitetura planejada e outro da arquitetura atual do sistema analisado. No contexto deste trabalho, foi escolhido criar duas representações em um mesmo nível de abstração (arquitetura planejada e atual) e compará-las, com o intuito de facilitar a implementação dos algoritmos, tornando-os mais reusáveis para trabalhos relacionados com modernização por meio das tecnologias da OMG. Alternativas para condução de uma CCA seria realizar a verificação entre um modelo de alto nível e o código-fonte (como visto em Terra e Valente (2009)) ou entre modelos de níveis de abstração parecidos (como visto em Bittencourt et al. (2010), Herold e Rausch (2013)). Na Figura 4.3, a abordagem proposta é apresentada, e, como pode ser observado, ela é dividida em três etapas principais: *I - Especificação da Arquitetura Planejada*, *II - Extração da Arquitetura Atual* e *III - Checagem de Conformidade*.

A etapa **I** é composta por uma única atividade (atividade **A**), que corresponde a especificação de arquiteturas. O principal objetivo é prover a especificação da arquitetura planejada de um sistema contendo os seus respectivos elementos arquiteturais e restrições entre eles. No contexto deste trabalho, os elementos arquiteturais são descritos de acordo com os elementos presentes no pacote `Structure` do KDM: `layer`, `component`, `subsystem`, `softwareSystem` e `architectureView`. As restrições são especificadas de acordo com a abordagem proposta por Terra e Valente (2009), os quais responsáveis por regular os relacionamentos permitidos ou não entre os elementos arquiteturais. A saída dessa etapa é a representação da arquitetura planejada por meio de uma instância do pacote `Structure` do KDM.

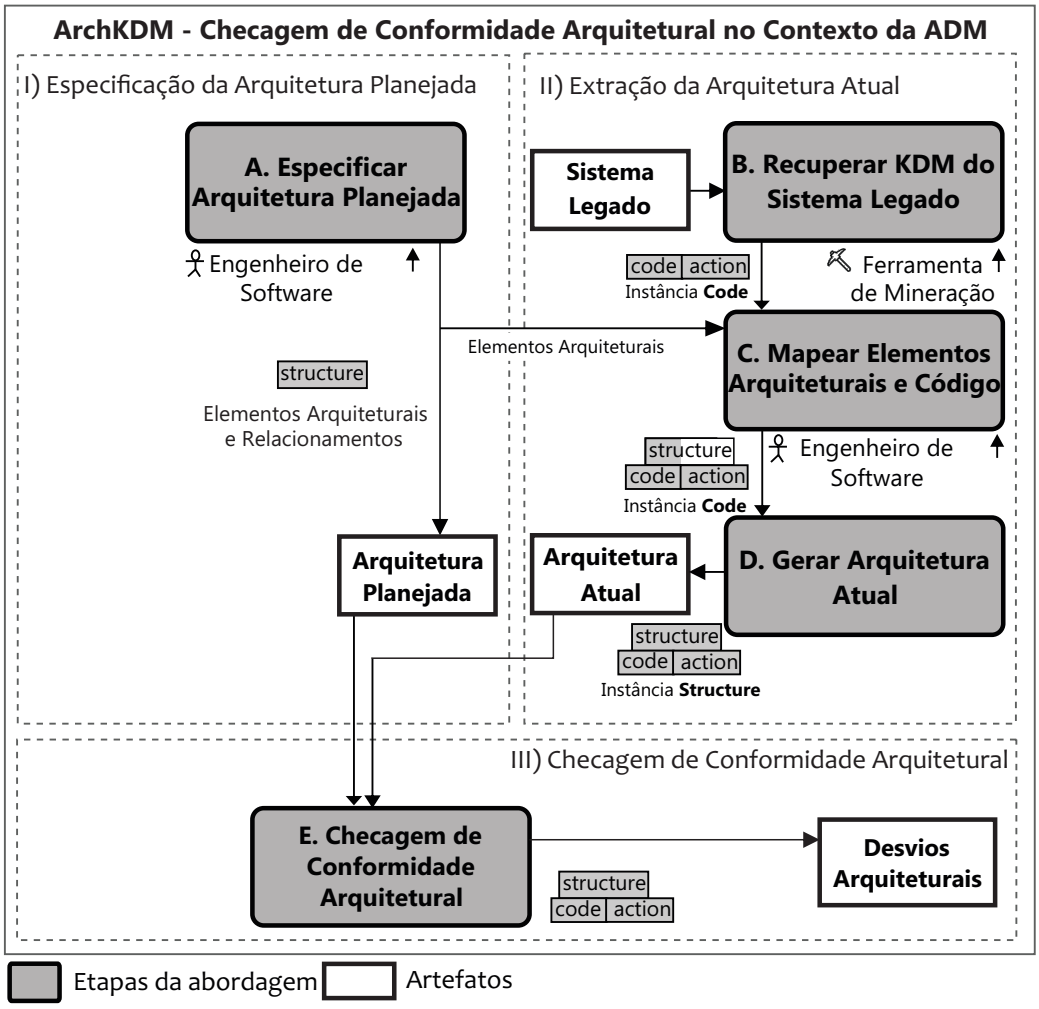


Figura 4.3: Abordagem ArchKDM

A etapa **II** é responsável pela extração da arquitetura atual e é composta por três atividades (**B**, **C** e **D**), como pode ser visto na parte **II** do Figura 4.3. Na atividade **B**, a ferramenta MoDisco é utilizada para obter uma instância KDM que representa todos os artefatos de código fonte (classes, ações, atributos, etc) do sistema legado. Na atividade **C**, o engenheiro de software deve mapear os elementos arquiteturais (arquitetura planejada) para os elementos de código fonte (sistema legado). Ao final desta atividade, a saída é um instância KDM que representa a arquitetura atual do sistema incompleta, pois faltam as relações entre os elementos arquiteturais. O processo de mapeamento é realizado por um *plug-in* do Eclipse que permite ao usuário relacionar os elementos arquiteturais aos de código fonte. Após o mapeamento, na atividade **D**, um algoritmo processa o KDM gerado na atividade **C** e gera uma instância KDM contendo todos os elementos de código e a arquitetura atual do sistema com seus respectivos relacionamentos.

A última etapa da abordagem proposta é responsável por realizar a checagem de conformidade, e é composta por uma única atividade (atividade **E**). As representações da arquitetura

planejada e atual devem ser comparadas. Essa comparação é feita verificando a presença ou ausência dos relacionamentos da arquitetura atual em relação à arquitetura planejada. Como saída, espera-se que sejam apontados indícios de violações arquiteturais, que são as diferenças encontradas entre as representações fornecidas. O resultado da checagem será fornecido por meio de uma nova instância do KDM, contendo os elementos que infringiram as regras definidas na representação da arquitetural planejada do sistema, essa instância deverá ser analisada por um engenheiro responsável pelo sistema com o intuito de apontar oportunidades de refatoração. Nas seções seguintes serão detalhadas cada uma das etapas envolvidas neste estudo.

4.2 Especificação da Arquitetura Planejada com DCL-KDM

Nesta seção, é apresentada a linguagem de descrição arquitetural com foco na representação de arquiteturas que utilizam estilos arquiteturais, chamada DCL-KDM. A especificação realizada na DCL-KDM irá gerar uma instância estrutural do KDM que servirá como entrada para o processo de checagem de conformidade arquitetural descrito na Seção 4.4. DCL-KDM é uma extensão de DCL, inicialmente proposta por Terra e Valente (2009), que é apresentada na Subseção 4.2. A extensão realizada constitui-se da inclusão de novas palavra-chave para a especificação de estilos arquiteturais. As principais palavras-chave incluídas são: `layer`, `component`, `subSystem`, `softwareSystem` e `interface`.

4.2.1 DCL-KDM

Com a inclusão de novas palavras-chave na DCL é possível que o engenheiro de software possa ser bem mais detalhista durante a especificação de uma arquitetura. Outro detalhe importante a respeito da extensão realizada é que as regras arquiteturais existentes entre elementos podem ser definidas automaticamente. Por exemplo, ao se criar três camadas (`layer`), deve-se informar obrigatoriamente a hierarquia entre elas para que a DCL-KDM seja capaz de gerar automaticamente as regras de restrição de acesso.

Como este trabalho encontra-se no contexto da ADM, as palavra-chave criadas foram motivadas a partir dos elementos arquiteturais presentes no metamodelo KDM. Assim, como pode ser visto na Figura 2.7 (pacote `Structure` do KDM), a maioria das metaclasses existentes nesse metamodelo podem agora ser representadas com a DCL-KDM. Entretanto, vale ressaltar que embora o KDM apresente esses elementos arquiteturais, o metamodelo não se preocupa com as restrições existentes em estilos arquiteturais.

Como previamente mencionado, a DCL foi atualizada para contemplar estilos arquiteturais e para instanciar a arquitetura planejada de sistemas no metamodelo KDM. Assim, o arquiteto de software passa a ter a segurança de que a sua especificação segue as regras impostas para cada estilo arquitetural. Em alguns dos estilos, como em camadas, basta que o arquiteto defina o nome da camada e o nível a que ela pertence. Dessa maneira, as restrições passam a ser impostas automaticamente na arquitetura. Em outros casos, as restrições são escritas pelo arquiteto de software, por exemplo, no estilo por componentes, em que o arquiteto deve definir as interfaces de cada componente e como elas interagem entre si. No entanto, a DCL-KDM acompanha os passos do arquiteto para que haja o mínimo de possibilidades de erros durante a especificação, esse acompanhamento é feito de duas maneiras: (i) a própria sintaxe da linguagem, que obriga o arquiteto a definir os elementos de acordo com cada estilo arquitetural e (ii) o uso do x-text[1], que é um *framework* para o desenvolvimento de linguagens de programação, ele obriga o arquiteto de software a realizar a especificação na ordem exata em que cada palavra chave da DCL-KDM deve ser escrita.

Um ponto importante a ser destacado é que para representar as dependências da DCL-KDM foi necessário buscar quais eram as metaclasses correspondentes no metamodelo KDM. É necessário identificar essas metaclasses por dois motivos: (a) na arquitetura planejada, elas representam os tipos de dependências permitidas entre os módulos, e (b) na arquitetura atual, elas são responsáveis por indicar que tipo de relacionamento está acontecendo entre os elementos em baixo nível. Na Tabela 4.1, é possível ver um comparativo entre as dependências da abordagem deste projeto e as metaclasses que as representam em KDM.

Tabela 4.1: Dependências DCL-KDM e metaclasses correspondentes em KDM

Dependência DCL-KDM	**Metaclasse KDM**
Access	Calls
Declare	HasType, Type
Handle	Calls, HasType, Type
Create	Creates
Extend	Extends
Implement	Implements
Derive	Extends, Implements
Throw	Calls, ExceptionFlow
Useannotation	HasValue
Depend	Todas citadas.

[1]https://eclipse.org/Xtext/

4.2.1.1 Camadas

De acordo com Garlan et al. (2010), camadas são um agrupamento de elementos que juntos oferecem um conjunto de serviços coesivos para outros elementos. As relações entre camadas devem ser unidirecionais, ou seja, uma camada que utiliza recursos de outra camada não pode fornecer recursos para ela. Garlan et al. (2010) comentam que o estilo arquitetural em camadas possui uma propriedade fundamental: as camadas são criadas para interagir de acordo com uma relação de ordem estrita. Se duas camadas `A` e `B` se relacionam, isso significa que a implementação da camada `A` pode acessar qualquer elemento público da camada `B`.

O estilo arquitetural em camadas é representado na DCL-KDM por meio da palavra-chave `layer`. Para inserir uma camada na especificação da arquitetura, o arquiteto deve descrevê-la de acordo com o exemplo abaixo:

```
1 layer l1, level 1
2 layer l2, level 2
3 layer l3, level 3
```

Código 4.1: Exemplo de descrição do estilo em camadas no DCL-KDM

A expressão para especificação de uma camada é dividida em duas partes obrigatórias e uma opcional, a primeira identifica o elemento estrutural, por meio do uso da palavra chave `layer`, seguido do nome da camada. Na segunda parte, deve ser definido o nível (`level`) em que a camada se encontra na especificação. A parte opcional é utilizada quando uma camada está contida em um subsistema, devendo ser definido a qual elemento a camada pertence. As regras de acesso que regem o estilo arquitetural, foram formalizadas abaixo:

$$L_i \textbf{ cannot} - \textbf{depend } L_j \mid (\mathrm{j} > \mathrm{i} \; or \; \mathrm{j} < \mathrm{i} - 1) \text{ and } L_j \in SS \mid C \mid L \tag{4.1}$$

L = layer, SS = subsystem, C = component

Equação 4.1: BNF que descreve camadas no contexto da DCL-KDM

O estilo arquitetural em camadas possui algumas restrições que são aplicadas automaticamente na DCL-KDM, bastando que o arquiteto defina qual o nível que cada camada pertence. A expressão acima reflete as restrições referentes as relações de **skip-calls** e **back-calls**, que são descritas por Sarkar, Rama e Shubha (2006) como nocivas a arquiteturas em camadas. Uma **Skip-call** acontece quando uma camada possui dependências com outra que não seja a imediatamente inferior a ela, representada no exemplo por $\mathtt{j} < \mathtt{i} - 1$. Uma **back-call** acontece quando uma camada possui dependências com alguma camada em nível superior ($\mathtt{j} > \mathtt{i}$). No Código 4.2 são mostradas as restrições que foram criadas implicitamente pela DCL-KDM após a definição da arquitetura pelo engenheiro de software, de acordo com o exemplo descrito no

Código 4.1.

```
1 l1 cannot-depend l2
2 l2 cannot-depend l3
3 l1 cannot-depend l3
4 l3 cannot-depend l1
```

Código 4.2: Restrições geradas automaticamente pela DCL-KDM

As restrições mostradas são criadas automaticamente durante a geração da instância estrutural que representa a arquitetura. Nas linhas 1-3 são mostradas as restrições referentes as relações de **back**, enquanto que a linha 4 representa as restrições de **skip**. No Código 4.3 é mostrada uma instância KDM que representa os exemplos arquiteturais descrito nos Códigos 4.1 e 4.2.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <kdm:Segment xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
3 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4 xmlns:action="http://kdm.omg.org/action" xmlns:code="http://kdm.omg.org/code"
5 xmlns:kdm="http://kdm.omg.org/kdm"
6 xmlns:structure="http://kdm.omg.org/structure">
7 <model xsi:type="structure:StructureModel" name="SystemExample">
8 <structureElement xsi:type="structure:Layer" name="l1"/>
9 <structureElement xsi:type="structure:Layer" name="l2">
10 <aggregated from="//@model.0/@structureElement.1"
11 to="//@model.0/@structureElement.0" relation="//@model.1/@codeRelation.0 [...]
12 //@model.1/@codeElement.0/@codeElement.0/@codeElement.0/@actionRelation.2" density="8"/>
13 </structureElement>
14 <structureElement xsi:type="structure:Layer" name="l3">
15 <aggregated from="//@model.0/@structureElement.2"
16 to="//@model.0/@structureElement.1"
17 relation="//@model.1/@codeElement.0/@codeElement.0/@actionRelation.0  [...]
18 //@model.1/@codeElement.0/@codeElement.0/@codeElement.0/@actionRelation.2" density="8"/>
19 </structureElement>
20 </model>
21 </kdm:Segment>
```

Código 4.3: Código KDM que representa a descrição de uma arquitetura simples em camadas

Na linha 7 é especificado o modelo `StructureModel`, responsável por determinar que uma nova especificação arquitetural será feita. Em seguida, nas linhas 7, 8 e 14 são declaradas as três camadas por meio da metaclasse `Layer`. Os relacionamentos entre essas camadas são representados pela metaclasse `AggregatedRelationship` (linhas 10-12), como características principais deve-se destacar que um relacionamento em alto nível é composto por uma origem e um destino, além dos relacionamentos em baixo nível que o compõem, como chamadas a métodos (`calls`) e implementação de interfaces (`implements`). Por fim, a densidade é definida (linha 12), que indica o total de relacionamentos aceitos entre a origem e o destino.

4.2.1.2 Componentes e interfaces

De acordo com a definição de Garlan et al. (2010), componentes são as principais unidades de processamento, elas possuem um conjunto de portas (interface/conectores) pelas quais é possível interagir com outros componentes. As restrições são: a) componentes só podem ser anexados a conectores e b) conectores só podem ser anexados a componentes. Na DCL-KDM, componentes são descritos por meio da palavra-chave `component`. A descrição de um componente pode ser feita de acordo com a Equação 4.2.

$$\text{Component} ::= \textbf{component}\ \textit{componentId}\ [,\ (\textbf{inLayer}\ \textit{layerID})\ |\ \textbf{inSubSystem}\ \textit{subSystemID}\ |\ \textbf{inComponent}\ \textit{componentID})] \quad (4.2)$$

Equação 4.2: BNF que descreve componentes no contexto da DCL-KDM.

A DCL-KDM permite que um componente pertença a uma camada, subsistema ou outro componente. Para especificá-lo, o arquiteto deve definir o nome do componente, e em seguida, caso haja necessidade, defina a camada, subsistema ou componente a qual ele pertence. Porém, como citado anteriormente, componentes necessitam de estruturas auxiliares para representar dependências, esses elementos são chamados de interfaces. Portanto, a dependência é realizada com sucesso quando uma interface de um componente consome recursos de outro componente, como descrita na Equação 4.3.

$$\begin{aligned}\text{Interface} ::= \textbf{interface}\ \textit{interfaceId}\ \textbf{ofComponent}\ \textit{componentId},\\ \text{type (provided | required)}\end{aligned} \quad (4.3)$$

Equação 4.3: BNF que descreve interfaces no contexto da DCL-KDM.

A interface deve ser declarada por meio da palavra-chave `interface` seguida de um identificador. A especificação de uma interface também necessita de que o arquiteto defina a qual componente ela pertence, para tal, o termo `ofComponent` deve ser utilizado seguido do identificador. Diferentemente das camadas, o estilo arquitetural definido por meio de componentes necessita que as suas restrições sejam especificadas explicitamente, suponha a definição descrita no Código 4.4.

```
1 component c1;
2 component c2;
3 interface c1_p1 ofComponent c1;
4 interface c2_r1 ofComponent c2;
5 c2_r1 can-depend-only c1_p1;
```

Código 4.4: Trecho DCL-KDM para especificação de interfaces e suas restrições

Nas linhas 1-2, os componentes são especificados. Na linha 3, a interface `c1_p1` é definida para o componente `c1`, em seguida, na linha 4, a interface `c2_r1` é definida para o componente `c2`. A restrição arquitetural é definida na linha 5, que determina que a interface `c2_r1` só pode depender da interface `c1_p1`. Em seguida, a descrição do Código 4.4 é mostrada em uma instância KDM.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <kdm:Segment xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
3 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4 xmlns:action="http://kdm.omg.org/action" xmlns:code="http://kdm.omg.org/code"
5 xmlns:kdm="http://kdm.omg.org/kdm"
6 xmlns:structure="http://kdm.omg.org/structure">
7 <model xsi:type="structure:StructureModel" name="ComponentExample">
8 <structureElement xsi:type="structure:Component" name="c1"/>
9 <structureElement xsi:type="structure:Component" name="c2">
10 <aggregated from="//@model.1/@codeElement.0/@codeElement.1"
11 to="//@model.1/@codeElement.0/@codeElement.0"
12 relation="//@model.2/@codeElement.0/@codeElement.0/@actionRelation.0 [...]
13 //@model.2/@codeElement.0/@codeElement.0/@actionRelation.2" density="8"/>
14 </structureElement>
15 </model>
16 <model xsi:type="code:CodeModel">
17 <codeElement xsi:type="code:Package">
18 <codeElement xsi:type="code:InterfaceUnit" name="c1_p1"/>
19 <codeElement xsi:type="code:InterfaceUnit" name="c2_r1"/>
20 </codeElement>
21 </model>
22 </kdm:Segment>
```

Código 4.5: Código KDM que representa a descrição de uma arquitetura em componentes

O Código 4.5 apresenta um exemplo de instância KDM referente a descrição de uma arquitetura composta por componentes e interfaces. Nas linhas 8-9 são declarados os componentes. Em seguida, nas linhas 18-19 as metaclasses referentes as interfaces são apresentadas, vale ressaltar que essa metaclasse não representa um elemento arquitetural, mas sim um elemento de código. Por fim, nas linhas 10-13 é descrito o relacionamento entre os dois componentes.

4.2.1.3 Subsistemas

Garlan et al. (2010) explicam que um subsistema pode ser executado independente de qualquer outro elemento arquitetural. A palavra-chave utilizada para especificar subsistemas na DCL-KDM é `subSystem`. A Equação 4.4 descreve um subsistema na abordagem proposta.

```
SubSystem ::= subSystem subSystemId [, inSubSystem]
```

Figura 4.4: BNF que descreve subsistemas no contexto da DCL-KDM

Os requisitos da especificação de um subsistema são: a) o subsistema deve possuir a palavra-chave `subSystem`, seguido de um identificador, e b) ele pode estar dentro de um outro subsistema. As restrições são definidas pelo arquiteto de software e funcionam de maneira semelhante aos módulos de Terra e Valente (2009). Uma diferença importante entre os dois elementos (subsistemas e módulos) é o posicionamento de subsistemas na hierarquia de uma arquitetura, pois a DCL-KDM prevê que um subsistema pode conter: camadas, componentes e outros subsistemas. Em seguida é descrita uma arquitetura composta por subsistemas de acordo com o metamodelo KDM.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <kdm:Segment xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
3 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4 xmlns:action="http://kdm.omg.org/action" xmlns:code="http://kdm.omg.org/code"
5 xmlns:kdm="http://kdm.omg.org/kdm"
6 xmlns:structure="http://kdm.omg.org/structure">
7 <model xsi:type="structure:StructureModel">
8 <structureElement xsi:type="structure:Subsystem" name="ss1">
9 <aggregated from="//@model.0/@structureElement.0"
10 to="//@model.0/@structureElement.0/@structureElement.0"
11 relation="//@model.1/@codeElement.0/@codeElement.0/@actionRelation.0 [...]
12 //@model.1/@codeElement.0/@codeElement.0/@actionRelation.2" density="8"/>
13 <structureElement xsi:type="structure:Subsystem" name="ss2"/>
14 </structureElement>
15 </model>
16 </kdm:Segment>
```

Código 4.6: Código KDM que representa a descrição de uma arquitetura em subsistemas

No Código 4.6 é mostrada a descrição de uma arquitetura composta por dois subsistemas organizados hierarquicamente. Os subsistemas representados pela metaclasse `Subsystem` estão definidos nas linhas 8-14. Vale ressaltar que o subsistema `ss2` está contido no `ss1`, portanto um relacionamento foi criado para contemplar a estrutura hierárquica (linhas 9-12).

4.2.1.4 Sistema de Software

Todas as regras descritas na DCL-KDM consideram que, por padrão, existe um elemento raiz na hierarquia da descrição arquitetural, que representa o sistema de software e é chamado de `softwareSystem`. Dessa maneira, ele é adicionado automaticamente à instância estrutural do KDM, evitando assim a necessidade de adicioná-lo à descrição da DCL-KDM.

4.2.1.5 Módulo

A DCL-KDM manteve o uso do elemento arquitetural `Module`, pois este contempla a maioria dos casos de regras escritas de maneira explícita, do mesmo modo que ele é utilizado na DCL. Dessa maneira, a ADL continua aberta para a descrição de arquiteturas independentes de estilos arquiteturais.

4.2.1.6 Especificação de arquiteturas na DCL-KDM

Esta seção tem como objetivo apresentar exemplos básicos de uma arquitetura especificada na DCL-KDM. Além disso, é mostrado como a instância estrutural do KDM é gerada após a especificação. Para isso, foi criada uma arquitetura na qual é possível representar diferentes estilos arquiteturais e uma quantidade significativa de possibilidades, que é ilustrada na Figura 4.5.

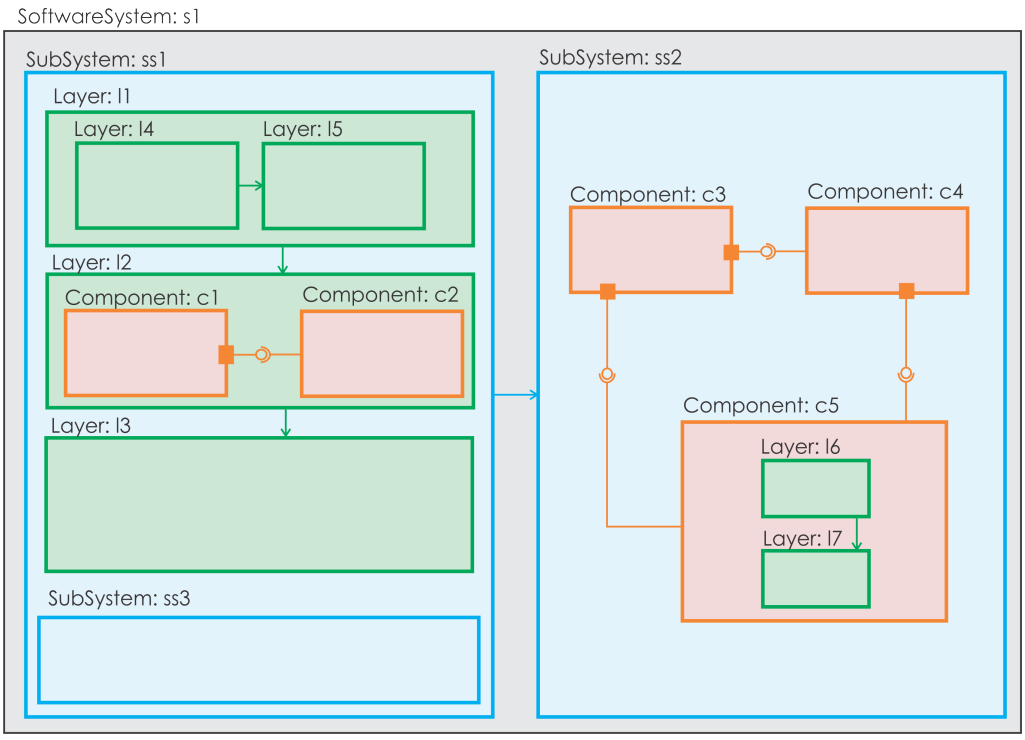


Figura 4.5: Arquitetura avaliada

```
1 architeturalElements {
2 subSystem ss1;
3 layer l1, level 3, inSubSystem: ss1;
4 layer l2, level 2, inSubSystem: ss1;
5 layer l3, level 1, inSubSystem: ss1;
6 layer l4, level 2, inLayer: l1;
7 layer l5, level 1, inLayer: l1;
8 component c1, inLayer: l2;
9 interface c1_p1 ofComponent c1;
10 component c2, inLayer: l2;
```

```
11 interface c2_r1 ofComponent c2;
12 subSystem ss3, inSubSystem: ss1;
13 subSystem ss2;
14 component c3, inSubSystem: ss2;
15 interface c3_p1 ofComponent c3;
16 interface c3_p2 ofComponent c3;
17 component c4, inSubSystem: ss2;
18 interface c4_r1 ofComponent c4;
19 interface c4_p1 ofComponent c4;
20 component c5, inSubSystem: ss2;
21 interface c5_r1 ofComponent c5;
22 interface c5_r2 ofComponent c5;
23 layer l6, level 2, inComponent: c5;
24 layer l7, level 1, inComponent: c5;
25 }
26 restrictions {
27 c2_r1 can-depend-only c1_p1;
28 c4_r1 can-depend-only c3_p1;
29 c5_r1 can-depend-only c4_p1;
30 c5_r2 can-depend-only c3_p2;
31 ss1 can-depend-only ss2;
32 }
```

Código 4.7: Arquitetura representada na DCL-KDM

A arquitetura apresentada foi criada com o intuito de avaliar de maneira abrangente todos os elementos e combinações suportados pela DCL-KDM: `softwareSystem`, `subSystem`, `layer`, `component` e `interface`. O sistema apresentado é dividido da seguinte maneira: no ponto mais alto de abstração está um sistema de software (`s1`), que é dividido em dois subsistemas (`ss1` e `ss2`).

O subsistema `ss1` (linha 2, Listagem 4.7) possui em sua arquitetura uma parte composta pelo subsistema `ss3` (linha 12, Listagem 4.7) e a outra parte por uma sub arquitetura que possui as camadas `l1`, `l2` e `l3` (linhas 3-5, Listagem 4.7), na qual a camada `l1` pode acessar elementos de `l2`, que por sua vez pode acessar elementos de `l3`. A arquitetura da camada `l1` é composta pelas camadas `l4` e `l5` (linhas 6-7, Listagem 4.7), na qual a camada `l4` pode acessar elementos de `l5`, enquanto que a camada `l2`, que é composta por dois componentes `c1` e `c2` (linhas 8-10, Listagem 4.7), na qual o componente `c1` provê recurso por meio de uma interface para o consumo do componente `c2`.

O subsistema `ss2` (linha 13, Listagem 4.7) tem a sua arquitetura composta pelos componentes `c3`, `c4` e `c5`, na qual o componente `c3` (linha 14, Listagem 4.7) provê recurso para os componentes `c4` (linha 17, Listagem 4.7) e `c5` (linha 20, Listagem 4.7) e o componente `c4` provê recurso para o componente `c5`. A arquitetura do componente `c5` é composta pelas camadas `l6` e `l7` (linhas 23, 24, Listagem 4.7), na qual a camada `l6` pode acessar os elementos da camada

`l7`. A arquitetura mostrada na Figura 4.5 foi descrita na DCL-KDM e o resultado é mostrado no Código 4.7.

A especificação da arquitetura feita na DCL-KDM é dividida em duas etapas: (a) descrição dos elementos arquiteturais e (b) definição das restrições entre os elementos. Na primeira etapa, os elementos arquiteturais são especificados e os detalhes relacionados à hierarquia são definidos, por exemplo, na linha 9 fica evidente que a interface `c1_p1` pertence ao componente `c1`, a hierarquia foi definida por meio da palavra chave `ofComponent`; enquanto isso, o componente `c1` está contido (`inLayer`) na camada `l2` (linha 8), que por sua vez está contida (`inSubsystem`) no subsistema `ss1` (linha 4).

Na segunda etapa, as restrições relacionadas aos elementos são definidas. Por exemplo, na linha 27, é definido que a interface `c2_r1` pode consumir apenas (**can-depend-only**) recursos da interface `c1_p1`. Um detalhe importante a ser mencionado é que as relações de restrição hierárquicas são automaticamente construídas pela DCL-KDM, por exemplo, o fato da camada `l2` estar contida no subsistema `ss1` implica que a DCL-KDM gerou uma regra que permite que elementos do subsistema `ss1` possam acessar elementos da camada `l2`.

A primeira etapa da abordagem proposta neste trabalho é a descrição arquitetural realizada na DCL-KDM, a saída desta etapa são dois arquivos: um contendo a descrição propriamente dita, para caso o arquiteto deseje posteriormente realizar alterações em sua arquitetura e, em outro, uma instância estrutural com os elementos arquiteturais e relacionamentos entre eles. O segundo arquivo serve de entrada para as etapas seguintes deste trabalho.

Ao final da especificação de uma arquitetura, uma instância KDM `Struture` deve ser gerada. Por exemplo, na Figura 4.6 é mostrada uma representação gráfica de uma arquitetura planejada.

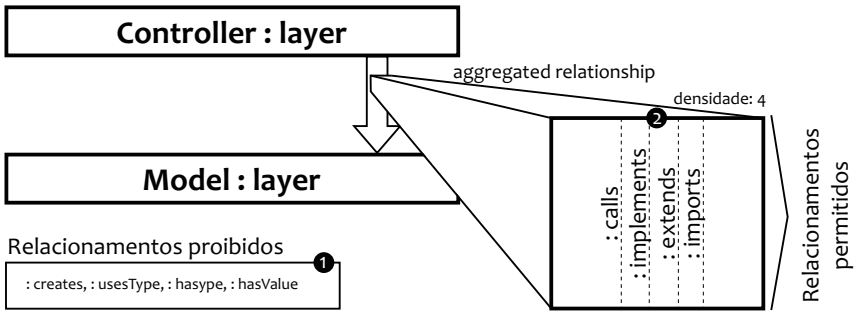


Figura 4.6: Exemplo Arquitetura Planejada

No exemplo da Figura 4.6, é mostrado uma representação gráfica da arquitetura planejada. O `AggregatedRelationship` possui quatro relacionamentos: `calls`, `implements`, `extends` e `imports` ❷. Em outras palavras, na arquitetura planejada do sistema podem existir apenas

os relacionamentos citados, todos os que estão fora dos mencionados ❶ podem caracterizar a possibilidade de violação arquitetural. Outro ponto importante é que para efeito de demonstração, todas as figuras relacionadas a esse exemplo levam em consideração duas camadas e os relacionamentos entre eles, como mostrado na Figura 2.10.

4.3 Extração da Arquitetura Atual

Esta seção apresenta o processo de extração da arquitetural atual do sistema alvo. Para que este processo seja realizado com sucesso, três atividade (B, C e D)s devem ser realizadas, conforme apresentadas na etapa **II** da Figura 4.3. Para realizar estas atividades, um *plug-in* para Eclipse foi desenvolvido, chamado **ExtrArch**.

Na atividade **B** (Figura 4.3, etapa **II**), **Recuperar instância do KDM legado**, uma representação em alto nível do sistema legado é gerado por um *plug-in* chamado de MoDisco. Essa ferramenta gera uma instância do KDM que possui os elementos de código fonte (pacote `code`) e outros elementos de baixo nível.

Na atividade **C** (Figura 4.3, etapa **II**), **Mapear elementos arquiteturais e código fonte**, um mapeamento manual é realizado entre os elementos arquiteturais e a instância KDM que representa o sistema legado. Para isso, o engenheiro de software deve mapear os elementos arquiteturais para elementos de código fonte, por exemplo, cada camada (elemento arquitetural) pode se referenciado a um ou mais pacotes (elementos de código fonte). É importante notar que a nossa abordagem permite um baixo nível de granularidade, por exemplo, além de pacotes, também é possível realizar a seleção de classes e interfaces para o mapeamento. Em seguida, uma instância `Structure` do KDM é adicionada à instância do sistema legado, porém, sem os preenchimento dos relacionamentos.

ExtrArch provê um interface gráfica para esta etapa, como mostrado na Figura 4.7. Os elementos arquiteturais extraídos da especificação da arquitetura planejada são apresentados no lado esquerdo da janela ❶. Cada um desses elementos é mapeado para um ou mais elementos de código ❷. O mapeamentos já realizados são mostrados na parte inferior da Figura 4.7 ❸. Após o mapeamento, a ferramenta cria uma versão inicial da arquitetura planejada contendo o mapeamento sem os relacionamentos extraídos.

Na atividade **D** (Figura 4.3, etapa **II**), **Gerar arquitetura atual** do sistema. Para isso, o algoritmo implementado nesta abordagem verifica todos os elementos de código em busca de ações entre diferentes elementos arquiteturais, por exemplo, encontrar todas as chamadas de método da uma camada A para uma camada B. Além dos relacionamentos, o algoritmo também

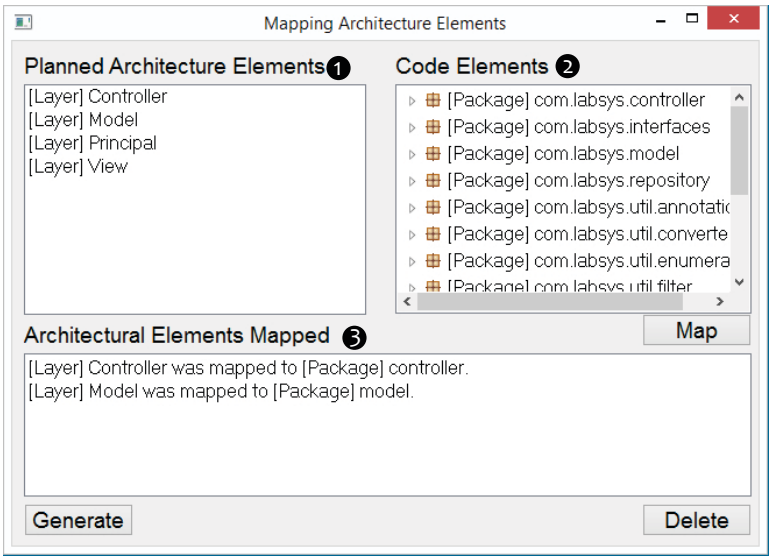


Figura 4.7: Mapeamento dos elementos arquiteturais para instância KDM

busca por declaração de variáveis, criação de objetos, extensão de classes, implementação de interfaces, tratamento de exceções e uso de anotações. No Algoritmo 1, é mostrado como funciona a extração.

Na linha 2, um método é chamado para realizar a busca de todos os relacionamento primitivos na instância do KDM gerada na atividade **B** desta etapa. Para isso, uma busca em profundidade é aplicada. Nas linhas 3-4 são recuperados os elementos de código que representam a origem (**from**) e destino (**to**) de cada relacionamento, pois identificá-los é parte importante para que os relacionamentos em nível arquitetural sejam criados. Em seguida, nas linhas 5-6 são buscados todos os elementos arquiteturais que contém os de código fonte (linhas 3-4). Por exemplo, é identificada a camada que um determinado método pertence. Na linha 7 é verificado se o elemento arquitetural `ae1` possui algum `aggregatedRelationship`, ou seja, se alguma relação em alto nível pertence àquele elemento arquitetural. Caso possua, todos `ar` do `ae1` serão analisados para verificar qual possui o argumento `to` igual ao `ae2` (linha 10), ou seja, verifica-se se existe algum relacionamento entre os elementos arquiteturais que contém o destino (`to`) como argumento. Se algum resultado for encontrado, o relacionamento deve ser adicionado ao `ar` (linha 11) e a densidade de `ar` deve ser incrementada em um, senão, um novo `aggregatedRelationship` deve ser criado (linhas 14-17). Por fim, se o `ae1` não possuir nenhum `aggregatedRelationship`, um novo deve ser criado (linhas 21-23).

O resultado da extração é a atualização da instância que possui o sistema legado, incluindo uma visão estrutural da arquitetura atual do sistema. Dessa maneira, a checagem de conformidade pode ser realizada conforme os requisitos identificados na Seção 4.1. Na Figura 4.8, é mostrado um exemplo da arquitetura atual do sistema.

Algorithm 1: ExtrArch - Extração da arquitetura atual do sistema

Input: Instância KDM com os pacotes code e action totalmente preenchidos e o pacote structure parcialmente.

Output: Instância KDM com os pacotes code, action totalmente preeenchidos (arquitetura atual do sistema).

```
1  begin
2      listPR = kdmUtil.getAllPrimitiveRelationships() foreach outgoing listPR do
3          from = listPR.getFrom()
4          to = listPR.getTo()
5          ae1 = seekArchitecturalElement(from)
6          ae2 = seekArchitecturalElement(to).
7          if ae1 has ae1.ar then
8              ar = ae1.getAR
9              foreach outgoing ar do
10                 if ar.getTo = ae2 then
11                     ar.add(listPR)
12                     ar.density(ae1.ar.getDensity+1)
13                 end
14                 if ar is the last then
15                     ar = new AR
16                     ar.add(listPR)
17                     ar.density(1)
18                 end
19             end
20         else
21             ar = new AR
22             ar.add(listPR)
23             ar.density(1)
24         end
25     end
26 end
```

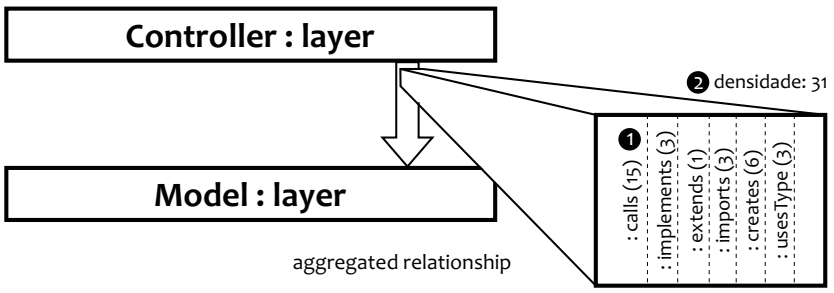


Figura 4.8: Representação da arquitetura atual

O primeiro ponto a ser observado é que o modelo de representação é mantido (comparado as Figuras 2.10 e 4.6). Portanto, as etapas da abordagem consideram o mesmo nível de abstração em todas as representações para facilitar e tornar mais clara a checagem de conformidade. No

entanto, nessa representação em particular, é possível notar que existem múltiplas instâncias dos relacionamentos (como visto em ❶). A camada `Controller` possui 15 (quinze) chamadas de métodos e atributos que pertencem a camada `Model`, porque todos os relacionamentos entre essas camadas são armazenados na arquitetura atual do sistema. Logo, o valor da densidade ❷ muda substancialmente.

4.4 Checagem de Conformidade Arquitetural no contexto da ADM

Conforme visto anteriormente (Seção 4.1), ArchKDM foi dividida em três etapas principais. Após a especificação da arquitetura planejada do sistema (etapa I) e o mapeamento e extração da arquitetura atual (etapa II), é possível a realização da checagem de conformidade arquitetural que, na abordagem proposta por este trabalho, é realizada por meio da comparação das representações arquiteturais (planejada e atual), como visto na Figura 4.3, etapa **III**. ArchKDM foi desenvolvida como uma evolução de DCL-KDM (Seção 4.2) e ExtrArch (Seção 4.3).

Para realizar a checagem no contexto deste trabalho, faz-se necessário que a arquitetura planejada (DCL-KDM) e atual (ExtrArch) estejam especificadas. O principal objetivo da checagem de conformidade é realizar a comparação das especificações arquiteturais e informar as possíveis violações presentes entre as representações. Para isso, deve-se verificar quais as metaclasses existentes na arquitetura atual que não foram previstas na arquitetura planejada. Por exemplo, caso uma chamada de método entre as camadas A e B na arquitetura atual não seja permitida na arquitetura planejada, i.e., esteja ausente - há caracterização de uma violação arquitetural. Ao final dessa etapa, uma instância do metamodelo KDM contendo todas as possibilidades de violações é gerada. Mais importante, essa instância pode ser utilizada como entrada para futuras modernizações independentes de plataforma e linguagem de programação.

Para que a checagem de conformidade seja realizada com sucesso, é preciso que o algoritmo seja capaz de realizar corretamente a comparação entre duas instâncias KDM e apontar a diferença entre elas. Por ser um dos passos independentes de intervenção humada mais importantes para o sucesso da abordagem, ela foi avaliada no Capítulo 5. No Algoritmo 2, é mostrado como a checagem de conformidade foi implementada.

Na linha 3, são recuperados todos os relacionamentos (`aggregatedRelationship`) da arquitetura atual do sistema. Em seguida, cada elemento da lista de `aggregatedRelationship` (AR) é verificado com o objetivo de compará-lo com as relações (`calls`, `implements`, `extends`, etc) da arquitetura planejada (linha 4). Em seguida, o AR da arquitetura planejada correspon-

Algoritmo 2: ArchKDM - Checagem de Conformidade Arquitetural

Input: Instâncias KDM que representam as arquiteturas planejada*planned* e atual *current*.
Output: Instância KDM contendo os violações arquiteturais *r*.

```
1  begin
2      currentArcRelationships = kdmUtil.seekAllAggregatedRelationships(current)
3      foreach outgoing currentArcRelationships currentRelationship do
4          plannedRelationship = seekCorrespondentRelationship(currentRelationship)
5          currentRelations = currentRelationship.getRelations()
6          foreach outgoing currentRelations relation do
7              if not relation.contains(plannedRelationship.getRelations() then
8                  aggRelationshipList.add(relation)
9              end
10         end
11     end
12 end
```

dente a arquitetura atual é recuperado (linha 5). Finalmente, para cada relação (linha 6) é verificado se as relações não estão contidas no AR da arquitetura planejada (linha 7). Em caso positivo, esse relacionamento é incluído na instância KDM que contém os violações arquiteturais (linha 8).

O primeiro passo é recuperar todos os relacionamentos da arquitetura atual (linha 2), para isso, foi implementada uma função que verifica todas as metaclasses presentes na instância KDM referente a arquitetura atual, dessa maneira, chamadas de métodos, implementação de interfaces, heranças e as outras categorias de relacionamentos são buscadas pelo algoritmo. Em seguida, para cada um destes relacionamentos, deve-se seguir os passos descritos a seguir. Primeiro, são buscados os relacionamentos em nível arquitetural dentro da arquitetura planejada que são correspondentes aos da arquitetura atual (linha 4), por exemplo, todos os relacionamentos entre as camadas A e B, com origem na camada A. Em seguida, deve-se recuperar as relações primitivas do relacionamento da arquitetura atual que está sendo verificado (linha 5), ou seja, chamadas de métodos, implementação de interfaces, tratamento de exceções, entre outros. Na linha 7, esses relacionamentos primitivos são comparados com os equivalentes da arquitetura planejada (linha 7), para que todo relacionamento primitivo da arquitetura atual que não foi especificado na arquitetura planejada seja adicionado à lista de possíveis violações arquiteturais (linha 8). Portanto, ao final da execução do algoritmo deve-se existir um KDM que possua a lista de violações.

Na Figura 4.9 é mostrado o processo de checagem que ocorre no ArchKDM, de acordo com as diretrizes mostradas no Algoritmo 2. As ações pertencentes à parte branca do `Aggregated-`

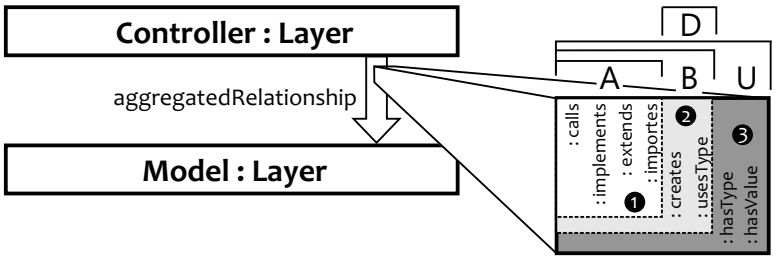


Figura 4.9: Checagem de conformidade arquitetural

`Relationship` ❶ representam as restrições impostas na arquitetura planejada, elas são formalizas como conjunto A: `calls`, `implements`, `extends` e `imports`. A parte cinza claro do `AggregatedRelationship` ❷ somada ao conjunto A representam as ações existentes na arquitetura atual, formalizadas como conjunto B: `calls`, `implements`, `extends`, `imports`, `creates` e `usesType`. Por fim, a parte cinza escura ❸ representa o universo de ações que a abordagem é capaz de encontrar, formalizada como conjunto U.

As violações arquiteturais encontradas pela abordagem são apresentadas no conjunto D. O cálculo realizado para encontrar oportunidades de refatoração é feito por meio da checagem dos elementos do conjunto B (atual) que não estão presentes no conjunto A (planejada) ($D = B - A$). Cada um dos elementos pertencentes a esse conjunto é adicionado a uma nova instância `Structure` do KDM que irá conter apenas as violações arquiteturais, como explicado anteriormente. Portanto, de acordo com o exemplo apresentado, o conjunto D deve ser composto pelos relacionamentos `creates` e `usesType`, que estão presentes na arquitetural atual (B) e não são explicitamente permitidos pela arquitetura planejada (A).

4.5 Considerações Finais

Neste capítulo foram abordados alguns pontos importantes em relação a este trabalho, tais como: ArchKDM, DCL, DCL-KDM, ExtrArch e CCA no contexto da ADM. ArchKDM foi a abordagem para a realização da checagem de conformidade. DCL foi a linguagem específica de domínio em que este trabalho foi inspirado para a realização da especificação da arquitetura planejada, a qual é responsável por definir restrições entre módulos. DCL-KDM é a adaptação de DCL para permitir representar estilos arquiteturais e instanciar o resultado dessa especificação por meio do metamodelo KDM, que é a entrada para as outras etapas da checagem de conformidade arquitetural propostas neste trabalho.

É importante destacar que a DCL-KDM sofreu modificações em relação a DCL quanto aos tipos de elementos arquiteturais que é capaz de representar, pois representa estilos arquitetu-

rais. Além de também permitir a descrição arquitetural hierárquica explícita, tornando mais clara a leitura de uma especificação arquitetural. Com a inclusão dos estilos arquiteturais, foi possível que a DCL-KDM gerasse automaticamente algumas restrições, dependendo do estilos arquiteturais (e.g., camadas) e hierarquias. Por fim, a DCL-KDM incluiu a geração da instância KDM estrutural da representação criada com o intuito de fornecer a entrada necessária para as outras etapas da abordagem apresentada neste projeto. É importante mencionar que os tipos de dependências permaneceram as mesmas, pois não eram o foco deste trabalho.

Em seguida, foi mostrada a abordagem ExtrArch que foi utilizada para realizar a recuperação de arquiteturas de sistemas legados. Por fim, foi apresentada uma visão geral do contexto que envolve a abordagem de checagem. Foram mostradas as etapas de uma modernização que este trabalho se enquadra, o que é necessário para a abordagem funcionar no contexto da ADM e também como é possível identificar dependências no metamodelo KDM. Além disso, foi apresentada a checagem de conformidade arquitetural realizada entre as instância geradas nas etapas anteriores da abordagem.

Capítulo 5

AVALIAÇÃO

Neste capítulo é apresentada a avaliação da abordagem proposta. O mais importante objetivo dessa abordagem é analisar as violações arquiteturais encontradas. Porém, o melhor resultado possível é detectar as violações em um dado sistema sem encontrar falsos positivos e negativos. No entanto, a efetividade de toda a abordagem depende de três pontos principais: i) a correta especificação da arquitetura planejada (Etapa **I**); ii) a correta extração da arquitetura atual (Etapa **II**); e iii) a qualidade do algoritmo de checagem. Dessa maneira, na Seção 5.1, é apresentado um estudo empírico. Na Seção 5.2, uma análise crítica e realizada com base nos resultados obtidos no estudo. Na Seção 5.3, são apresentadas as ameaças à validade do estudo. Por fim, na Seção 5.4, são apresentadas as considerações finais.

5.1 Estudo Empírico

O experimento foi realizado por meio de estudos empíricos com o intuito de averiguar se é possível conduzir uma boa checagem de conformidade arquitetural utilizando o metamodelo KDM. O planejamento do estudo foi realizado de acordo com as diretrizes propostas por Wohlin et al. (2000).

5.1.1 Definição do Estudo Empírico

5.1.1.1 Objetivo

O objetivo geral é investigar a efetividade da abordagem em detectar as violações arquiteturais no contexto da ADM. No entanto, a avaliação foi dividida em duas partes. i) avaliação do algoritmo de extração (Atividade **II.C** - Figura 4.3), que é responsável pela recuperação

da arquitetural atual, e ii) avaliação do algoritmo de checagem (Etapa **III** - Figura 4.3), que é responsável por identificar as violações arquiteturais entre as arquiteturas planejada e atual.

5.1.1.2 Objeto de estudo

Os algoritmos de checagem da abordagem proposta.

5.1.1.3 Abordagem Quantitativa

O potencial da abordagem é medido por meio da porcentagem de relações corretas encontradas na extração da arquitetura atual e a violações encontradas pelo algoritmo durante a checagem.

5.1.1.4 Perspectiva

Os estudos foram conduzidos da perspectiva dos arquitetos de software.

5.1.1.5 Objeto de estudo

As relações extraídas na arquitetura atual e as violações detectadas pelo algoritmo de checagem.

5.1.2 Planejamento do Estudos

O planejamento da avaliação foi realizado em torno das seguintes questões: 1) Assumindo que o engenheiro de modernização proveu o mapeamento correto na Etapa **II**, o algoritmo de extração é capaz de recuperar a arquitetura atual corretamente? 2) Assumindo que as especificações das arquiteturas planejada e atual estão corretas, o algoritmo de checagem de conformidade é capaz de detectar todas as violações arquiteturais? Para responder essas questões de pesquisa, foram coletados os seguintes dados:

1. Todos os relacionamento entre elementos de código (classes, interfaces) extraídos automaticamente pela ferramenta.
2. As violações detectadas pelo algoritmo de checagem.

No Item 1, o foco é verificar se o algoritmo de extração é capaz de gerar todos os relacionamentos presentes no código fonte de um sistema dado. No Item 2, o foco é analisar se o

algoritmo de checagem é capaz de identificar violações entre todos os tipos de elementos (camadas e componentes, camadas e subsistemas, camadas e módulos, etc.) e restrições (chamadas de métodos, herança, implementação de interfaces, etc.), além de verificar se as violações encontradas foram devidamente capturadas.

5.1.2.1 Seleção de Contexto

Foram utilizados dois sistemas reais na avaliação: LabSys (*Laboratory System*) e SIGA (*Integrated System of Academic Management*). LabSys é um sistema gerenciador do uso de laboratórios que é atualmente utilizado pela Universidade Federal do Tocantins (UFT) e o SIGA é um sistema de gerenciamento acadêmico atualmente utilizado pela Universidade Federal de São Carlos (UFSCar).

O LabSys foi escolhido por envolver em sua arquitetura uma variedade considerável de estilos arquiteturais. Enquanto que o sistema SIGA foi escolhido por possuir um código fonte extenso e por seus engenheiros estarem interessados em detectar as violações arquiteturais que o sistema possui.

5.1.2.2 Instrumentação

Foram utilizados os seguintes instrumentos para aplicação da abordagem: a) o guia utilizado pelos arquitetos; b) a documentação dos sistemas que deram suporte a especificação da arquitetura planejada; c) o código fonte utilizado para extração da arquitetura; e d) os dados coletados depois da avaliação (relacionamentos da arquitetura atual e o resultado da checagem).

5.1.3 Operação

5.1.3.1 Preparação

Foram construídos dois oráculos para o LabSys; o primeiro contém todas as violações arquiteturais e o segundo possui todos os relacionamentos entre as metaclasses da arquitetura atual. Além disso, foram especificadas as arquiteturais planejadas para ambos os sistemas (A especificação da arquitetura do LabSys pode ser vista na Listagem 4.7). No sistema SIGA a construção de oráculos é inviável devido ao seu tamanho. Por fim, foram verificadas se as violações arquiteturais encontradas pela abordagem são falsas positivas, em outras palavras, se elas atualmente referem-se as decisões arquiteturais não prescritas na arquitetura planejada do sistema.

5.1.3.2 Execução

A abordagem é executada seguindo as seguintes etapas. Inicialmente, deve-se utilizar o DCL-KDM para especificar a arquitetura planejada do sistema a ser analisado. Em seguida, ExtrArch é aplicado para mapear os elementos arquiteturais (etapa anterior) com os elementos de código fonte do sistema legado. Por fim, ArchKDM é executado e a lista violações arquiteturais é obtida.

5.1.3.3 Dados

Foram coletados todos os relacionamentos e violações para posterior comparação com os oráculos feitos para o sistema LabSys. As violações arquiteturais foram analisadas a partir de duas perspectivas: quantidade de falsos positivos e restrições entre elementos diferentes (exemplo, camada e componente, camada e subsistema, etc). A quantidade de violações falsas positivas são utilizadas para averiguar se a checagem foi realizada com sucesso. Finalmente, são checadas se foram encontradas violações entre todas as combinações possíveis entre elementos arquiteturais.

5.1.4 Análise do processo de Extração

Nesta etapa da avaliação o objetivo é determinar se o algoritmo de extração é capaz de gerar a arquitetura atual corretamente. A parte crítica desta etapa é a extração dos relacionamentos existentes, pois um erro por parte do algoritmo pode levar a abordagem a não gerar resultados corretos. O algoritmo trabalha sobre uma instância KDM (gerada pelo MoDisco) que representa o sistema legado e é complementada com detalhes arquiteturais. A Tabela 5.1 lista os relacionamentos encontrados no sistema LabSys combinados com os resultados obtidos por meio do oráculo. É importante notar que cada relacionamento existente no código fonte representa uma instância de uma metaclasse específica no KDM, por exemplo, na Linha 3 da Tabela 5.1 é possível verificar que o algoritmo gera 240 instâncias da metaclasse `Calls` entre as camadas `Controller` e `Model`.

A análise da Tabela refcap6:tab-rellabsys permite verificar alguns pontos relevantes para a abordagem. Inicialmente, foi possível notar que todos os relacionamentos encontrados pela ferramenta também estavam presentes no oráculo. Portanto, o ArchKDM foi capaz de detectar corretamente todos os relacionamentos presentes na arquitetura atual. Outro ponto relevante verificado foi que todos os tipos de relacionamento foram encontrados nessa extração, dessa maneira pode-se afirmar preliminarmente que ArchKDM consegue recuperar corretamente os

Tabela 5.1: Relacionamentos recuperados da arquitetura atual do sistema LabSys

Elemento Arquitetural 1	Elemento Arquitetural 2	Calls	Implements	Extends	Imports	Creates	UsesType	HasType	HasValue
		A/O	A/O	A/O	A/O	A/O	A/O	A/O	A/O
view - Camada	controller - Camada								
view - Camada	model - Camada								
controller - Camada	model - Camada	240/240		21/21	52/52		19/19	150/150	
view - Camada	repository - Módulo								
controller - Camada	repository - Módulo	208/208			24/24			11/11	
model - Camada	repository - Módulo	6/6		24/24	37/37	1/1	2/2	128/128	3/3
view - Camada	generic - Componente								
controller - Camada	generic - Componente	108/108			8/8				
model - Camada	generic - Componente								
repository - Módulo	generic - Componente				25/25				
view - Camada	validator - Módulo								
controller - Camada	validator - Módulo	16/16			8/8			2/2	
model - Camada	validator - Módulo	9/9	8/8		8/8	1/1	1/1	9/9	1/1
repository - Módulo	validator - Módulo				12/12				
generic - Componente	validator - Módulo								
view - Camada	converter - Componente								
controller - Camada	converter - Componente				1/1				
model - Camada	converter - Componente	2/2	12/12		12/12	1/1			
repository - Módulo	converter - Componente	1/1			1/1				
generic - Componente	converter - Componente								
validator - Módulo	converter - Componente								

A = Algoritmo, O = Oráculo

oito tipos de relacionamentos que foram determinados na proposta da abordagem.

5.1.5 Análise do Algoritmo de Checagem

Nesta etapa da avaliação o objetivo é averiguar as violações arquiteturais detectadas pelo algoritmo de checagem. Cada um dos indícios foi analisado juntamente com o Engenheiro de Software responsável pela aplicação para que este pudesse fazer uma análise baseada na arquitetura planejada e verificasse se o indício apontado pela ferramenta estava correto. Para a aplicação LabSys, foi montado um oráculo para verificar os valores de falsos positivos e negativos, enquanto que esta atividade não foi possível para o SIGA.

Dessa maneira, um estudo comparativo foi realizado entre as violações encontradas pela abordagem e as encontradas pelo oráculo. Todos os indícios de desvio foram adicionados a uma nova instância do KDM para análise do engenheiro de software, em outras palavras, os indícios de violação são apresentados na interface disponibilizada pelo KDM-SDK e o engenheiro pôde verificar em que parte do sistema se encontram. Na Tabela 5.2 é sumarizada a quantidade total de violações encontradas pelo ArchKDM e pela verificação manual. Os resultados são apresentados por restrição, ou seja, cada uma das restrições especificadas na arquitetura planejada do sistema foi adicionada a tabela. Vale ressaltar, que além das restrições explícitas, também foram contabilizadas as restrições geradas automaticamente pela ArchKDM, por exemplo, restrições entre camadas e restrições hierárquicas.

Como pode ser observado, todos os indícios de violações arquiteturais foram encontrados

Tabela 5.2: Violações encontradas por restrição

N	Restrição	ArchKDM	Manualmente	Diferença
%restrições explicitas				
1	repository **can-depend-only** controller	58	58	0
2	validator **can-depend-only** controller	70	70	0
3	genericInterface **can-depend -only** controller	3	3	0
4	converterInterface **can-depend -only** controller	29	29	0
%restrições implícitas				
5	model **can-depend-only** controller	8	8	0
6	controller **can-depend-only** view	3	3	0
Total		171	171	0

pelo ArchKDM. Por exemplo, a restrição 1 define que o módulo `repository` pode depender apenas da camada `controller`, porém, foram encontradas violações de classes não pertencentes ao componente `controller` com o módulo `repository`. Outro exemplo é a restrição 4, que define que a interface `converterInterface` pode depender apenas da camada `controller`, no entanto foram encontradas violações na qual a camada `model` também consome os recursos providos pela interface `converterInterface`.

Os indícios de violações arquiteturais encontrados pela ArchKDM foram sumarizados na Tabela 5.2 e em seguida comparados com o oráculo, essa verificação mostrou que todas as violações encontradas existiam no sistema avaliado, inclusive os que foram gerados automaticamente pela ferramenta. Outro ponto importante foi averiguar se as regras geradas automaticamente refletiam as reais restrições que o estilo arquitetural deveria ter. Após verificar as violações encontradas pelo oráculo e pela ferramenta, foi possível confirmar que a geração das restrições e captura de violações foi realizada com sucesso em todos os casos. Na Tabela 5.3 são mostrados os valores sumarizados de violações arquiteturais encontradas em ambos os sistemas testados.

Tabela 5.3: Violações arquiteturais presentes nos sistemas SIGA e LabSys

Aplicação	Violações encontradas	Violações oráculo	Falsos Positivos	Falsos Negativos
LabSys	171	171	0	0
SIGA	230	-	0	-

Como mencionado anteriormente, os resultados mostraram que foram encontrados 171 in-

dícios de violações arquiteturais no sistema LabSys, todos presentes no oráculo, portanto não foram encontrados falsos positivos. Além disso, não foram encontradas violações presentes oráculo e ausentes na checagem, portanto, não foram encontrados falsos negativos. Para o sistema SIGA, devido ao seu tamanho, não foi encontrado nenhum falso positivo, reforçando o resultado apresentado para a aplicação LabSys, que é de pequeno porte. Acredita-se que foi possível alcançar tais resultados na busca por violações arquiteturais por dois motivos: 1) a implementação do algoritmo de checagem e 2) a facilidade de manipulação/inspeção do meta-modelo KDM e de suas instâncias.

Outro ponto a ser avaliado é a habilidade do algoritmo de checagem em detectar violações entre diferentes tipos de elementos arquiteturais. Por exemplo, entre camadas e componentes, camadas e subsistemas, camadas e camadas, etc. Na Tabela 5.4 são apresentados os resultados. No sistema LabSys, apenas a combinação subsistema/subsistema não foi averiguada, pois a arquitetura do LabSys não possuía dois subsistemas. No entanto, no sistema SIGA, foram encontradas violações entre todas as possíveis combinações. Portanto, a abordagem foi capaz de detectar violações entre todas as possibilidades.

Tabela 5.4: Violações entre elementos arquiteturais

Elemento 1	Elemento 2	LabSys	SIGA
Camada	Camada	Sim	Sim
Camada	Subsistema	Sim	Sim
Camada	Componente	Sim	Sim
Camada	Módulo	Sim	Sim
Subsistema	Componente	Sim	Sim
Subsistema	Módulo	Sim	Sim
Módulo	Componente	Sim	Sim
Subsistema	Subsistema	Não	Sim
Módulo	Módulo	Sim	Sim
Componente	Componente	Sim	Sim

5.2 Análise Crítica

O estudo de caso LabSys e as verificações realizadas pelos autores permitiram a relização de uma análise crítica sobre a abordagem de checagem de conformidade arquitetural proposta neste artigo. A análise foi feita de acordo com os seguintes critérios:

Expressividade: A abordagem proposta possui foco na realização da checagem de conformidade arquitetural por meio da comparação de dois modelos de alto nível que representam a arquitetura planejada e atual de um sistema. A avaliação mostrou uma quantidade significativa

de combinações entre elementos e estilos arquiteturais diferentes. A corretude para o caso de estudo apresentado foi de 100%, portanto, o ArchKDM não gerou falsos positivos, porém, um caso de teste não foi executado no estudo proposto (subsistema-subsistema), portanto, a cobertura das possibilidades de restrições entre elementos arquiteturais de tipos diferentes foi de 90%. Dessa maneira, pode-se verificar que a ArchKDM apresentou bons resultados quanto à qualidade de sua verificação de conformidade, definição de restrições e uso de estilos arquiteturais. Por outro lado, não foi realizado um estudo para a verificação de possíveis falsos negativos presentes no sistema que não foram mapeados pela ferramenta. Outro ponto importante a ser abordado é a incapacidade da ArchKDM em representar alguns estilos arquiteturais presentes na literatura, essa limitação é dada pela restrição de opções dos elementos disponíveis no KDM para representação de elementos arquiteturais.

Adequabilidade: O pacote estrutural do KDM fornece alguns elementos para a representação de elementos arquiteturais, porém, dependendo do estilo arquitetural a ser abordado, o KDM não consegue representar a menos que sejam utilizados estereótipos, por exemplo, portas e conectores. Fatores estes que dificultam a criação de abordagens para a checagem de conformidade arquitetural utilizando estilos arquiteturais, pois adaptar um metamodelo para atender a uma demanda específica de um algoritmo dificulta a reutilização das instâncias deste metamodelo por outras ferramentas que queiram realizar outras etapas do processo de refatoração.

Aplicabilidade: Até o momento desta pesquisa não foi encontrado nenhum trabalho que realize checagem de conformidade arquitetural levando em consideração estilos arquiteturais, porém, Pruijt, Koppe e Brinkkemper (2013) publicaram um estudo comparativo entre ferramentas que realizam checagem de conformidade e verificam estilos arquiteturais, em grande maioria, de maneira indireta, ou seja, aplicando restrições manuais a elementos arquiteturais, diferentemente do que é realizado neste trabalho, em que camadas, por exemplo, tem as suas restrições definidas automaticamente pela ArchKDM. A ferramenta proposta por este trabalho é capaz de representar os estilos arquiteturais em camadas, componentes/interfaces e subsistemas, porém, ainda existe uma boa quantidade de estilos arquiteturais que podem ser alvo de trabalhos futuros, por exemplo, filtros e dutos, aspectos (CLEMENTS et al., 2003), etc. Além disso, também não foram encontrados na literatura trabalhos que contemplem a verificação de conformidade utilizando o metamodelo KDM, que é independente de plataforma e linguagem, que podem ser alvos de estudos para verificação das vantagens e desvantagens em se utilizar modelos para condução destes processos.

5.3 Ameaças à validade

Foram relatadas duas ameaças a validade da avaliação apresentada. Primeiro, apesar de utilizarmos dois sistemas reais que possuem diferentes arquiteturas e restrições, não é possível afirmar que a abordagem irá prover resultados equivalentes em outros sistemas, como de costume em estudos empíricos na engenharia de software (*validação externa)*. Segundo, há dependência de dois engenheiros de software para avaliar a quantidade de falsos positivos. Como é típico em avaliações com humanos (*human-based*), os resultados podem ter sido afetados por algum grau de subjetividade (*validação de construção)*. No entanto, é importante lembrar que foram entrevistados os engenheiros de software que especificaram as arquiteturas, além de serem responsáveis pela manutenção e evolução dos sistemas.

5.4 Considerações Finais

Neste capítulo foi apresentada a avaliação da abordagem proposta neste trabalho (ArchKDM). Ela foi dividida em duas etapas: i) análise do processo de extração e ii) análise do algoritmo de checagem. Os resultados mostraram que a abordagem proposta nesse trabalho consegue conduzir uma checagem de conformidade arquitetural no contexto da ADM utilizando como base o metamodelo KDM, sem a utilização de extensões, leves ou pesadas. Um ponto importante a ser ressaltado é que o ArchKDM foi capaz de recuperar corretamente uma arquitetura a partir de um código fonte. Outro ponto relevante para o trabalho conduzido é que a abordagem alcançou bons níveis na busca por violações arquiteturais. Assim sendo, foi possível verificar que ArchKDM é capaz de funcionar corretamente em todas as etapas de uma checagem de conformidade arquitetural.

Capítulo 6
CONCLUSÃO

Neste trabalho foi apresentada uma nova abordagem para a realização de checagem de conformidade arquitetural (CCA). A motivação desta pesquisa partiu da observação de que a maioria das abordagens CCA empregam modelos diferentes para a representação da arquitetura planejada e arquitetura atual, ou modelos que não foram criados para representar conceitos arquiteturais (TERRA; VALENTE, 2009; ABI-ANTOUN; ALDRICH, 2008; ALDRICH; CHAMBERS; NOTKIN, 2002; CIRACI; SOZER; TEKINERDOGAN, 2012; DEISSENBOECK et al., 2010; YAN et al., 2004; MAFFORT et al., 2015; DUSZYNSKI; KNODEL; LINDVALL, 2009; HELLO2MORROW, 2015; ADERSBERGER; PHILIPPSEN, 2011; HEROLD; RAUSCH, 2013; SANGAL et al., 2005). Por isso, a avaliação foi concentrada em averiguar a adequação e impacto para representar AP e AA a partir da utilização de um metamodelo exclusivo (KDM). A hipótese era que o uso de um metamodelo único, capaz de representar conceitos arquiteturais, levaria a uma CCA com bons níveis de cobertura e precisão. Os resultados obtidos foram satisfatórios, como pode ser visto na seção de avaliação.

Uma característica da abordagem apresentada é que ela pode ser usada para verificar a conformidade de sistemas implementados em qualquer linguagem de programação. Isto é possível porque todos os algoritmos são dependentes apenas da terminologia KDM. Além disso, como nossos algoritmos são totalmente baseado em um padrão ISO, eles têm um grande potencial para reutilização. Isso não acontece quando algoritmos são desenvolvidos ao longo de um modelo proprietário ou dependentes do linguagem.

Ao observar a literatura, é possível verificar que em algumas abordagens (TERRA; VALENTE, 2009; DUSZYNSKI; KNODEL; LINDVALL, 2009; ALDRICH; CHAMBERS; NOTKIN, 2002) as etapas I e II da Figura 4.3) são realizadas em conjunto. No entanto, decidimos separar esses passos devido das seguintes razões: i) para permitir a especificação de arquiteturas planejadas a serem executadas em estágios iniciais do desenvolvimento, quando os detalhes de implementação não são conhecidos e ainda não é possível elaborar o mapeamento; e ii) propiciar que os artefatos

resultantes possam evoluir de forma independente.

Como foi mostrado, ArchKDM compreende três etapas. Na primeira, deve-se fornecer uma especificação da arquitetura planejada por meio da DCL-KDM. Como a abordagem depende de KDM, é necessário trazer todos os conceitos do metamodelo `Structure` ao nível do utilizador. Por isso, a decisão de estender DCL (TERRA; VALENTE, 2009) com estilos arquiteturais foi motivada principalmente pelas metaclasses do pacote `Structure` e para auxiliar os engenheiros de software com algumas restrições arquiteturais comuns e pré-concebidas, que são geradas automaticamente pela DCL-KDM.

Um detalhe importante na primeira etapa é que a saída é uma instância do modelo `Structure` representando a arquitetura planejada. Como o metamodelo `Structure` não foi originalmente criado para representar arquiteturas planejadas, não há metaclasses para representar restrições ou normas que regem as relações entre os elementos. Portanto, utilizou-se o `Aggregated−Relationship` para representar as regras, isto é, a existência de uma relação primitiva dentro de uma relação agregada indica que a relação é permitida entre os elementos arquiteturais, enquanto que a ausência indica o oposto.

Na segunda etapa, o objetivo é obter uma instância KDM contendo o sistema legado (obtida pelo Modisco) e a arquitetura atual. O engenheiro de modernização deve fornecer o mapeamento entre o elementos arquiteturais da arquitetura planejada e do código-fonte. Depois disso, a ferramenta gera um modelo que representa a arquitetura atual do sistema.

Um dos pontos avaliados foi a capacidade de adequação do KDM para representar arquiteturas de software. O pacote `Code` é claramente capaz de representar detalhes de baixo nível em uma qualidade muito boa, no entanto, a qualidade do pacote `Structure` é mais difícil de ser avaliada. Por exemplo, embora o pacote `Structure` possua as metaclasses mais convencionais para representar detalhes arquiteturais, ainda assim ele necessita de algumas outras importantes, por exemplo: filtros, conectores, portas, interfaces, etc. Para o caso da abordagem apresentada neste trabalho, essas classes não foram necessárias. Vale a pena notar que o KDM não foi estendido para representar detalhes arquiteturais, por exemplo, portas e conectores. Assim, a abordagem se limitou a trabalhar com as abstrações provenientes do KDM.

Na última etapa, as duas representações (planejada e atual) são comparadas e uma lista de violações arquiteturais é obtida. Como ambas as representações do sistema são instâncias do mesmo meta-modelo, o algoritmo (ver Algoritmo 2) torna-se mais claro, fácil de entender e, como consequência, mais fácil de manter, reutilizar e evoluir. Em síntese, com base na avaliação realizada, a CCA foi aplicada com sucesso. O uso de KDM não tem impacto sobre a qualidade do processo, principalmente porque o metamodelo `Code` está em um nível de abs-

tração muito semelhante ao código fonte. Dessa maneira, todos os detalhes para executar uma checagem arquitetural, tais como ações dinâmicas de código (chamadas, instanciações, etc) estão disponíveis.

Outro ponto importante a se notar é que a principal vantagem do KDM é a sua padronização. Assim, ferramentas de modernização agora possuem boas razões para adotar KDM como o principal metamodelo subjacente. A razão é que provavelmente haverá uma grande quantidade de recursos (por exemplo, algoritmos de CCA/refatoração/ferramentas/técnicas) disponíveis, que podem ser reutilizados. Nossos algoritmos são reutilizáveis em todas as ferramentas de conformidade que utilizem KDM porque seu código-fonte apenas menciona as metaclasses originais do KDM. Ou seja, mesmo considerando as ameaças da avaliação, pode-se dizer que há boas evidências de que é perfeitamente possível conduzir CCA no contexto da ADM e obter bons resultados em termos de detecção de violação. Isso acontece porque o pacote `Code` do KDM representa todos os detalhes de baixo nível necessárias em uma CCA.

6.1 Contribuições

- Uma abordagem para auxiliar a atividade de reestruturação de sistemas legados independente de plataforma e linguagem de programação. Em outras palavras, foi criada uma abordagem para a realização de checagem de conformidade arquitetural utilizando o metamodelo KDM, sem a utilização de extensões leves ou pesadas.

- Uma ADL para especificação de arquiteturas com a utilização de estilos arquiteturais, criada para auxiliar o engenheiro de software a conduzir a especificação da arquitetura de um sistema.

- Uma técnica para conversão do pacote *code/kdm* para *structure/kdm*. Apesar da ADM fornecer os metamodelos para utilização, nenhum desses possui maneiras de extração/-conversão automática. Portanto, se fez necessária a criação de mecanismos de conversão e adaptação de pacotes para que se pudesse alcançar níveis maiores de abstração dentro do KDM.

- Uma técnica para extração da arquitetura atual do sistema legado a partir de uma instância do KDM legado. A partir do mapeamento entre os elementos arquiteturais e os elementos de código fonte, o KDM legado foi analisado e foram extraídos todos os relacionamentos entre elementos de baixo nível (`code`/`kdm`) e adicionados as relações de alto nível (`structure`/`kdm`).

- Uma ferramenta de apoio computacional para auxiliar o engenheiro durante a atividade de checagem de conformidade arquitetural no contexto da ADM de forma eficiente, denominada ArchKDM.

6.2 Limitações

Um problema notado durante o desenvolvimento da abordagem apresentada é que o metamodelo KDM limitou a quantidade de estilos arquiteturais representados. Para que essa limitação fosse superada a curto prazo, extensões no metamodelo KDM deveriam ser feitas para que existisse suporte a elementos diferentes e com propriedades distintas, o que não é objetivo deste trabalho.

Outra limitação importante verificada foi que o MoDisco não gera elementos KDM para páginas web. Dessa maneira, as verificações de conformidade que envolvem esses arquivos não são realizadas. Com a quantidade de aplicações web desenvolvidas atualmente, realizar a verificação levando em consideração páginas web da aplicação é quase que obrigatório para se alcançar uma grande fatia das aplicações produzidas.

6.3 Trabalhos Futuros

Como trabalho futuro, pretende-se realizar outros tipos de avaliações. Por exemplo, como um número de restrições são geradas automaticamente. É importante caracterizar as vantagens e desvantagens de tal abordagem na liberdade de utilizar a linguagem proposta. Além disso, também pretende-se aplicar a abordagem deste projeto de mestrado em sistemas de maior porte.

Outro trabalho importante é gerar instâncias KDM compatíveis com páginas web. Por fim, sugerir e realizar refatorações automáticas e semi-automáticas a partir dos resultados gerados pela abordagem também caracterizam um bom tópico de pesquisa para trabalhos futuros.

REFERÊNCIAS

ABI-ANTOUN, M.; ALDRICH, J. Tool support for the static extraction of sound hierarchical representations of runtime object graphs. In: *23rd Conference on Object-oriented Programming Systems Languages and Applications (OOPSLA)*. [S.l.: s.n.], 2008. p. 743–744.

ADERSBERGER, J.; PHILIPPSEN, M. Reflexml: Uml-based architecture-to-code traceability and consistency checking. In: *Software Architecture*. [S.l.]: Springer Berlin Heidelberg, 2011, (Lecture Notes in Computer Science, v. 6903). p. 344–359.

ADM. *Architecture-Driven Modernization.* 2015. Disponível em: `http://adm.omg.org/`. Acesso em: 20 fev. 2016.

ALDRICH, J.; CHAMBERS, C.; NOTKIN, D. ArchJava: connecting software architecture to implementation. In: *24rd International Conference on Software Engineering (ICSE)*. [S.l.: s.n.], 2002. p. 187–197.

BASS, L.; CLEMENTS, P.; KAZMAN, R. *Software Architecture in Practice*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1998.

BITTENCOURT, R. et al. Improving automated mapping in reflexion models using information retrieval techniques. In: *17th Working Conference on Reverse Engineering (WCRE)*. [S.l.: s.n.], 2010. p. 163–172.

BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. *Unified Modeling Language User Guide*. [S.l.]: Addison-Wesley Professional, 2005.

BOSCH, J. *Design and Use of Software Architectures: Adopting and Evolving a Product-line Approach*. [S.l.: s.n.], 2000.

BRAGA, T. H. *Recuperação da arquitetura de software para manutenção de sistemas*. Tese (Doutorado) — Universidade Federal de Minas Gerais - UFMG, 2013.

BRUNELIERE, H. et al. Modisco: A generic and extensible framework for model driven reverse engineering. In: *IEEE/ACM International Conference on Automated Software Engineering*. [S.l.: s.n.], 2010. (ASE '10), p. 173–174.

CHIKOFSKY, E.; CROSS J.H., I. Reverse engineering and design recovery: a taxonomy. *IEEE Software*, v. 7, n. 1, p. 13–17, 1990.

CIRACI, S.; SOZER, H.; TEKINERDOGAN, B. An approach for detecting inconsistencies between behavioral models of the software architecture and the code. In: *36th Annual Computer Software and Applications Conference (COMPSAC)*. [S.l.: s.n.], 2012. p. 257–266.

CLEMENTS, P. et al. *Documenting Software Architectures: Views and Beyond*. [S.l.]: Addison-Wesley, 2003.

DEISSENBOECK, F. et al. Flexible architecture conformance assessment with conqat. In: *32nd International Conference on Software Engineering (ICSE)*. [S.l.: s.n.], 2010. p. 247–250.

DEITERS, C. et al. Rule-based architectural compliance checks for enterprise architecture management. In: *Enterprise Distributed Object Computing Conference*. [S.l.: s.n.], 2009. p. 183–192.

DIJKSTRA, E. W. The humble programmer. *ACM*, v. 15, n. 10, p. 859–866, 1972.

DUSZYNSKI, S.; KNODEL, J.; LINDVALL, M. SAVE: Software architecture visualization and evaluation. In: *13th European Conference on Software Maintenance and Reengineering (CSMR)*. [S.l.: s.n.], 2009. p. 323–324.

FOWLER, M. *Patterns of Enterprise Application Architecture*. [S.l.]: Addison-Wesley, 2003.

GARLAN, D. Software architecture: A roadmap. In: *Conference on The Future of Software Engineering*. [S.l.: s.n.], 2000. p. 91–101.

GARLAN, D. et al. *Documenting Software Architectures: Views and Beyond*. 2nd. ed. [S.l.]: Addison-Wesley Professional, 2010.

GURP, J. van; BOSCH, J. Design erosion: Problems and causes. *Journal of Systems and Software*, v. 61, n. 2, p. 105–119, 2002.

HELLO2MORROW. *SotoArc*. 2015. Disponível em: <http://www.hello2morrow.com/products/sotoarc>.

HEROLD, S.; RAUSCH, A. Complementing model-driven development for the detection of software architecture erosion. In: *5th International Workshop on Modeling in Software Engineering (MiSE)*. [S.l.: s.n.], 2013. p. 24–30.

HOFMEISTER, C.; NORD, R.; SONI, D. [S.l.: s.n.].

IZQUIERDO, J. L. C.; ZAPATA, B. C.; MOLINA, J. G. Definición y ejecución de métricas en el contexto de adm. 2010.

KNODEL, J. et al. Static evaluation of software architectures. In: *10th European Conference on Software Maintenance and Reengineering*. [S.l.: s.n.], 2006. p. 10 pp.–294.

KNODEL, J.; POPESCU, D. A comparison of static architecture compliance checking approaches. In: *The Working IEEE/IFIP Conference on Software Architecture*. [S.l.: s.n.], 2007. p. 12–12.

KRUCHTEN, P. The 4+1 view model of architecture. *IEEE Software*, v. 12, n. 6, p. 42–50, 1995.

MAFFORT, C. et al. Heuristics for discovering architectural violations. In: *20th Working Conference on Reverse Engineering (WCRE)*. [S.l.: s.n.], 2013. p. 222–231.

MAFFORT, C. et al. Mining architectural violations from version history. *Empirical Software Engineering Journal*, p. 1–41, 2015.

MEDVIDOVIC, N.; EGYED, A.; GRüNBACHER, P. Stemming architectural erosion by coupling architectural discovery and recovery. In: *International Software Requirements to Architectures Workshop (STRAW 2003)*. [S.l.: s.n.]. p. 61–68.

MEDVIDOVIC, N.; TAYLOR, R. N. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, v. 26, n. 1, p. 70–93, 2000.

MOOR, O. d. et al. Keynote address: .QL for source code analysis. In: *IEEE International Working Conference on Source Code Analysis and Manipulation*. [S.l.: s.n.], 2007. p. 3–16.

MURPHY, G. C.; NOTKIN, D.; SULLIVAN, K. Software reflexion models: Bridging the gap between source and high-level models. In: *Symposium on Foundations of Software Engineering*. [s.n.], 1995. p. 18–28. Disponível em: <http://doi.acm.org/10.1145/222124.222136>.

OMG. Object Management Group. ADM White Papers and Roadmap. Disponível em: <http://adm.omg.org/ADMTF20Roadmap.pdf>. Acesso em: 5 novembro, 2013.

OMG. Object Management Group. 2013. KDM Specification. Disponível em: <http://www.omg.org/spec/KDM/1.3/PDF/>. Acesso em: 18 março, 2013.

PEREZ-CASTILLO, R.; GUZMAN, I.-R. D.; PIATTINI, M. Knowledge discovery metamodel–ISO/IEC 19506: A standard to modernize legacy systems. *Computer Standards and Interfaces*, v. 33, n. 6, p. 519–532, 2011.

PERRY, D. E.; WOLF, A. L. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, v. 17, n. 4, p. 40–52, 1992.

PRESSMAN, R. S. *Software Engineering: A Practitioner's Approach*. 7th. ed. [S.l.: s.n.], 2010.

PRUIJT, L.; KOPPE, C.; BRINKKEMPER, S. Architecture compliance checking of semantically rich modular architectures: A comparative study of tool support. In: *29th IEEE International Conference on Software Maintenance (ICSM)*. [S.l.: s.n.], 2013. p. 220–229.

RAHIMI, R.; KHOSRAVI, R. Architecture conformance checking of multi-language applications. In: *IEEE/ACS International Conference on Computer Systems and Applications (AICCSA)*. [S.l.: s.n.], 2010. p. 1–8.

ROZANSKI, N.; WOODS, E. *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives*. [S.l.]: Addison-Wesley, 2011.

SANGAL, N. et al. Using dependency models to manage complex software architecture. In: *20th Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*. [S.l.: s.n.], 2005. p. 167–176.

SARKAR, S.; RAMA, G.; SHUBHA, R. A method for detecting and measuring architectural layering violations in source code. In: *Software Engineering Conference, 2006. APSEC 2006. 13th Asia Pacific*. [S.l.: s.n.], 2006. p. 165–172.

SHAW, M.; GARLAN, D. *Software Architecture: Perspectives on an Emerging Discipline*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996.

SILVA, L. de; BALASUBRAMANIAM, D. Controlling software architecture erosion: A survey. *J. Syst. Softw.*, v. 85, n. 1, p. 132–151, 2012.

SNEED, H. Estimating the costs of a reengineering project. In: *12th Working Conference on Reverse Engineering*. [S.l.: s.n.], 2005. p. 9–119.

SOMMERVILLE, I. *Software Engineering*. [S.l.]: Pearson Addison Wesley, 2004.

TERRA, R.; VALENTE, M. T. A dependency constraint language to manage object-oriented software architectures. *Software: Practice and Experience*, v. 39, n. 12, p. 1073–1094, 2009.

WOHLIN, C. et al. *Experimentation in Software Engineering: An Introduction*. [S.l.: s.n.], 2000.

YAN, H. et al. DiscoTect: A system for discovering architectures from running systems. In: *26th International Conference on Software Engineering*. [S.l.: s.n.], 2004. p. 470–479.