UNIVERSIDADE DE SÃO PAULO
FACULDADE DE FILOSOFIA, CIÊNCIAS E LETRAS DE RIBEIRÃO PRETO
PROGRAMA INTERUNIDADES DE PÓS-GRADUAÇÃO EM BIOINFORMÁTICA

RICARDO CACHETA WALDEMARIN

Desenvolvimento baseado em modelos de serviços adaptadores para ferramentas de bioinformática

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001.

Ribeirão Preto
2021

RICARDO CACHETA WALDEMARIN

Desenvolvimento baseado em modelos de serviços adaptadores para ferramentas de bioinformática

Versão Corrigida

Tese de Doutorado apresentada ao Programa Interunidades de Pós-Graduação em Bionformática da Universidade de São Paulo para obtenção do título de Doutor em Ciências.

Área de Concentração: Bioinformática
Orientador: Prof. Dr. Cléver Ricardo Guareis de Farias

Ribeirão Preto

2021

Waldemarin, Ricardo Cacheta.
Desenvolvimento baseado em modelos de serviços adaptadores para ferramentas de bioinformática / Ricardo Cacheta Waldemarin; Orientador: Prof. Dr. Cléver Ricardo Guareis de Farias – Ribeirão Preto, 2021.
292p.

Tese (Doutorado) – Universidade de São Paulo, 2021.
1. Serviços Web. 2. Desenvolvimento orientado a modelos. 3. Ferramentas de análise. 4. Bioinformática. I. Cléver Ricardo Guareis de Farias, orient. II. Título

Obrigado, Ma.
Você esteve comigo
durante todo esse período
e foi a luz de meus dias
quando tudo parecia trevas.

Resumo

Um experimento *in silico* envolve a execução de um conjunto de atividades de análise normalmente suportadas por uma ou mais ferramentas dedicadas. Estas ferramentas são, em geral, produzidas como programas de linha de comando executados localmente. Um serviço *web* pode ser criado de modo a adaptar uma ferramenta de análise existente, provendo ao usuário uma interface *web* adequada e invocando de maneira transparente a ferramenta de análise sobre os dados submetidos. Porém, o desenvolvimento de serviços *web* adaptadores requer conhecimentos técnicos específicos pouco comuns a bioinformatas em geral e biologistas em particular. Adicionalmente, a falta de um modelo de referência para guiar o desenvolvimento destes serviços leva a proliferação de serviços de análise com diferentes características, o que dificulta o maior reúso dos serviços desenvolvidos. Nesse cenário, a disponibilidade de processos de desenvolvimento que facilitem a obtenção de um serviço adaptador maduro e padronizado facilitada é altamente desejável. Por esta razão, este trabalho teve por objetivo investigar o uso de abordagens de desenvolvimento baseadas em modelos para o suporte ao desenvolvimento e à utilização de serviços *web* de análise em bioinformática. Nesse sentido, este trabalho propõe inicialmente um modelo de referência para serviços de análise em bioinformática, de modo a guiar o desenvolvimento desses serviços e obter interfaces RESTful maduras. Em seguida, o trabalho propõe um processo de desenvolvimento orientado a modelos para serviços de análise adaptadores, seus clientes e descrições, e provê uma infraestrutura de suporte para esse processo de desenvolvimento Por fim, o processo de desenvolvimento proposto foi aplicado à reengenharia de um repositório de serviços para a realização de análises de expressão gênica, obtendo um novo conjunto de serviços mais maduros e com menor esforço de desenvolvimento que aquele despendido na criação dos serviços iniciais. Acreditamos que este trabalho fomenta o desenvolvimento de serviços de análise em bioinformática ao permitir ao especialista do domínio a produção de serviços adaptadores maduros, bem como de seus clientes e suas descrições, mais rapidamente.

Palavras-chave: *serviços web, desenvolvimento orientado a modelos, ferramentas de análise, bioinformática.*

Abstract

An *in silico* experiment demands the execution of a number of activities supported by one or more dedicated command-line tools. A analysis web service may be created to adapt an existing analysis tool, exposing a suitable web interface to its user while still executing the original tool over user-submitted data in the background. However, the development of an adapter web service requires technical knowledge unusually detained by bioinformaticians and biologists. Additionally, the lack of a well-known reference model to guide the development of mature analysis services leads to the proliferation of services that have distinct characteristics, hindering the reuse of these services in different contexts. Thus, the availability of a development process that facilitates the development of mature and consistent adapter services is highly desirable. This research aimed at studying the use of model-based approaches to support the development and use of analysis web services in bioinformatics. First, this work proposes a reference model for bioinformatics analysis services to guide the development of mature RESTful analysis services. Then, the work proposes a model-based development process for adapter services, their clients and their descriptions, and provides a support infrastructure for the development process. Finally, the work details the application of the proposed development process and support infrastructure in the reengineering of an existing service repository in the functional genomics domain. We believe our work promotes the development and use of analysis services in the bioinformatics domain by allowing the bioinformatician to quickly obtain a mature analysis service to adapt an existing analysis tool, as well as its clients and associate service description.

Keywords: *web services, model driven development, analysis tools, bioinformatics.*

Lista de ilustrações

Lista de tabelas

Lista de listagens

*

Sumário

1 Introdução

1.1 Contextualização

1.1.1 Atividades de análise e artefatos de suporte

Uma grande quantidade de dados biomédicos é produzida e analisada cotidianamente. Tais dados proveem de diferentes fontes e representam aspectos diferentes sobre um sistema biológico de interesse, como, por exemplo, o nível de expressão gênica e a marcação epigenética de uma célula em determinado tempo. O desejo de unir as diferentes visões de um mesmo organismo proporcionadas pelas diferentes fontes de dados levou ao desenvolvimento de áreas que possuem uma visão integrativa em bioinformática. Neste contexto, a computação tornou-se ferramenta essencial para o desenvolvimento de pesquisas na área das ciências biológicas, auxiliando na organização e na análise desses dados.

Uma análise *in silico* envolve a execução de um conjunto de atividades de análise de modo a obter uma informação biológica de interesse [1]. Essas atividades são normalmente suportadas por uma ou mais ferramentas dedicadas à automação dessas tarefas. Por sua vez, novas ferramentas de análise têm sido desenvolvidas em grande número nos últimos anos para automatizar diferentes atividades de análise ou testar novas hipóteses. Por exemplo, atualmente o registro BioTools [2] contém mais de 21 mil recursos para bioinformática catalogados, incluindo ferramentas e repositórios de dados. Contudo, no final de 2015 esse mesmo registro listava apenas 1785 recursos [3]. As ferramentas criadas, muitas vezes desenvolvidas na forma de programas ou *scripts*, podem ser reutilizadas se forem disponibilizadas de maneira a permitir seu uso em novas análises.

1.1.2 Serviços *web* em bioinformática

No domínio biomédico, a disponibilização das funcionalidades de uma ferramenta de análise por meio de um serviço *web* é uma solução cada vez mais utilizada para promover a integração entre ferramentas em bioinformática, bem como para prover a execução dessas ferramentas de maneira remota [4–7]. A disponibilização de um serviço *web* para o suporte à execução de uma atividade de análise tem como benefício principal o aumento da propensão ao uso da ferramenta. Um dos fatores para essa maior propensão ao uso está na facilidade que esses serviços proveem para a execução de atividade de análise suportada sem a necessidade de prover localmente toda a infraestrutura necessária para executá-la. Adicionalmente, o uso

de tecnologias padronizadas para a associação de informação semântica às descrições desses serviços facilita a criação de mecanismos para a descoberta (automática) e uso desses serviços [8].

De modo a utilizar um serviço *web* para a execução de uma dada atividade de análise, faz-se necessário conhecer a interface desse serviço, i.e., os *endpoints* providos pelo serviço, as operações suportadas por esses *endpoints* e as estruturas de dados utilizadas nas requisições e respostas dessas operações. Também é necessário compreender o modelo de interação do serviço, o qual define a sequência em que operações devem ser executadas sobre os *endpoints* definidos de modo a atingir um dado objetivo. A partir dessas informações, bibliotecas de suporte ao protocolo de comunicação HTTP podem ser utilizadas para interagir com o serviço de análise, invocando cada uma de suas operações.

Uma descrição de serviço consiste em um documento formal ou modelo abstrato que apresenta os *endpoints*, operações e estruturas de dados referentes a um dado serviço por meio de uma linguagem específica para essa representação. Descrições de serviços podem ser utilizadas para facilitar a compreensão da interface de um serviço de análise. Porém, aspectos comportamentais do serviço de análise, tais como seu modelo de interação, não são capturados pelas linguagens para descrição de serviços *web* atuais, o que dificulta a compreensão e a utilização de serviços complexos.

A complexidade existente no uso de um serviço de análise pode ser abstraída se o usuário possuir um cliente para este serviço. Um cliente de um serviço de análise consiste de uma aplicação ou biblioteca de *software* disponibilizada pelo provedor do serviço ou por terceiros que automatiza as interações necessárias para a execução da atividade de análise suportada por ele. Em geral, a implementação de um cliente de serviço está atrelada à interface e ao modelo de interação definido pelo serviço de análise, apresentando pouco acoplamento ao domínio de aplicação da atividade de análise que este serviço executa. Dessa maneira, um cliente para um serviço de análise pode ser mais simples e possuir menos dependências que uma ferramenta equivalente que executasse a mesma atividade de análise. Essas características facilitam a distribuição e o uso desses clientes, promovendo a maior exploração dos dados por meio de diferentes atividades/serviços durante o processo de análise.

Serviços de análise relacionados podem ser agregados em repositórios de serviços. Um repositório de serviços agrega serviços de análise providos por uma dada organização ou comunidade e provê mecanismos para a descoberta, manual ou automática, desses serviço por seus usuários. Serviços de análise em um mesmo repositório de serviços normalmente apresentam características semelhantes, como,

por exemplo, seu domínio de aplicação e o uso de um mesmo modelo de referência para a construção de suas interfaces. Essas características facilitam a descoberta de serviços de interesse e a integração de mais de um serviço em um dado repositório durante o processo de análise. Entre estes repositórios estão o repositório de serviços mantido pelo *European Bioinformatics Institute* (EBI) [9], o repositório mantido pelo *National Center for Biotechnology Information* (NCBI) [10] e o repositório *Gene Expression Analysis Services* (GEAS) [7].

De modo a permitir que ferramentas de análise sejam corretamente integradas e executadas, ambientes de suporte em bioinformática têm sido desenvolvidos [1,8,11–15]. Esses ambientes de suporte, também chamados de ambientes integrados de análise ou sistemas de gerenciamento de *workflows*, buscam abstrair os detalhes tecnológicos necessários à integração das ferramentas e serviços de modo a facilitar a criação de um processo de análise completo. Exemplos desses ambientes são as plataformas *Galaxy* [14], Taverna [11, 15], SemanticSCo [8], Gaggle [12] e BioWMS [13].

Novas ferramentas e serviços de análise podem ser introduzidos em um ambiente integrado de análise por meio do desenvolvimento de uma extensão que permita a invocação dessa ferramenta ou serviço pelo ambiente [16,17]. Neste sentido, três atividades principais devem ser realizadas de modo a criar um novo serviço *web* para adaptar uma ferramenta de análise a um ambiente integrado de análise: i) o desenvolvimento do serviço *web* propriamente dito; ii) o desenvolvimento de um cliente para esse serviço; e iii) o desenvolvimento de extensões específicas para cada ambiente integrado de análise.

1.1.3 Desenvolvimento de serviços *web* em bioinformática

Serviços de análise podem ser classificados em *serviços monolíticos* ou *serviços adaptadores*. Esta classificação considera a forma de integração entre o código que provê a interface do serviço e o código que implementa a execução da atividade de análise. Em um serviço monolítico, o código específico para a execução de uma atividade de análise e o código específico para o suporte à interface do serviço executam em um mesmo processo. Nestes serviços, a ferramenta de análise não é vista como um programa executável, sendo utilizada como qualquer outra biblioteca de *software* da qual o serviço depende ou mesmo sendo implementada *ab initio* como parte do serviço *web*. Para que isso possa ser realizado, tanto o serviço quando a ferramenta de análise precisam ser implementados por meio da mesma linguagem de programação ou utilizarem de suporte disponível para a invocação nativa de código escrito na outra linguagem.

Embora uma ferramenta de análise possa ser construída como um serviço *web* ou integrada nativamente neste serviço, também é possível construir um serviço

web que atue como um adaptador para uma ferramenta (executável) existente. Este serviço adaptador oculta a interface original da ferramenta (por exemplo, chamadas de linha de comando) e oferece acesso às funcionalidades da mesma por meio de interfaces padronizadas e protocolos padrões da *web*. Esta arquitetura demanda uma separação mais forte entre o serviço adaptador e a ferramenta de análise que aquela encontrada entre um serviço monolítico e o código embarcado para a execução da atividade de análise. Nesse sentido, enquanto um serviço monolítico pode invocar o código que executa a atividade de análise como invoca qualquer outra chamada de função ou procedimento, o serviço adaptador precisa utilizar os recursos do sistema operacional do hospedeiro, como chamadas de processo e sistemas de arquivos, para se comunicar com a ferramenta de análise sendo adaptada. Dessa maneira, a implementação de um serviço adaptador demanda que o desenvolvedor coordene a execução e comunicação entre dois (ou mais) processos, tornando um serviço adaptador mais complexo que o serviço monolítico equivalente. Por outro lado, um serviço adaptador tem como vantagem da capacidade de integrar uma ferramenta de análise de maneira independente da linguagem de programação utilizada no desenvolvimento do serviço e da ferramenta de análise adaptada.

De modo a facilitar o desenvolvimento de adaptadores para ferramentas de análise existentes, uma metodologia de desenvolvimento foi proposta em [7]. Esta metodologia foi aplicada na implementação de um conjunto de serviços RESTful disponibilizados por meio do repositório *Gene Expression Analysis Services* (GEAS) [8]. Porém, embora tenha sido utilizada com sucesso, essa metodologia não abstrai os aspectos técnicos necessários ao desenvolvimento desses serviços, de modo que o desenvolvedor que deseja utilizá-la ainda necessita dominar as tecnologias existentes para o desenvolvimento de serviços *web*. Por fim, essa metodologia não apresenta suporte para o desenvolvimento de clientes para os serviços implementados, de maneira que também não abstrai os detalhes desse serviço aos usuários que desejam utilizá-lo.

A necessidade de conhecimento de detalhes tecnológicos específicos para o desenvolvimento de serviços *web*, de clientes para esses serviços e de extensões para ambientes integrados de análise limita a maior disponibilização e uso de serviços de análise na bioinformática. Embora um bioinformata seja frequentemente capaz de desenvolver novas ferramentas de análise, este especialista não necessariamente possui conhecimentos de desenvolvimento *web*, o que dificulta o desenvolvimento de um serviço *web* maduro para a execução da atividade de análise.

A ausência de uma padronização entre os serviços disponibilizados por diferentes repositórios também representa uma dificuldade para o desenvolvimento e uso desses serviços. Uma vez que não há um modelo de referência comum para guiar a implementação dos serviços de análise entre os diversos repositórios, torna-se

necessário um maior esforço na conceitualização e documentação de cada novo serviço ou repositório. Dessa maneira, cada novo serviço de análise ou repositório de serviços pode apresentar especifidades que os distanciam dos serviços de análise já existentes. Essas especificidades também impõe um esforço maior ao usuário que deseja integrar diferentes serviços para a execução de sua análise em bioinformática, pois, na falta de clientes dedicados disponíveis, torna-se necessário o estudo e a compreensão da interface e do comportamento provido por cada um dos serviços sendo utilizado de forma integrada.

1.1.4 Desenvolvimento baseado em modelos

Metodologias de desenvolvimento baseadas em modelos podem ser utilizadas em cenários nos quais deseja-se obter uma solução de software complexa a partir de modelos descritivos sobre o domínio da aplicação [18]. Tais modelos facilitam a compreensão e o raciocínio envolvendo os elementos de domínio relevantes em um dado momento da trajetória de desenvolvimento, ao mesmo tempo em que detalhes não relevantes são abstraídos [19].

A aplicação de metodologias de desenvolvimento baseadas em modelos também tem o potencial de mover o esforço aplicado no desenvolvimento do novo software da atividade de implementação e de seus detalhes tecnológicos para a descrição do domínio por meio do uso de modelos abstratos. Neste sentido, metodologias de desenvolvimento baseadas em modelos promovem o desenvolvimento e uso de transformações automáticas para obter o código fonte da aplicação diretamente dos modelos de domínio iniciais. Adicionalmente, o uso de interpretações automatizadas dos elementos dos modelos de domínio iniciais em tempo de execução também é uma maneira de obter a solução de software desejada nestas metodologias [18, 20]. Porém, essas vantagens são apenas obtidas quando há suporte ferramental adequado disponível para a implementação e uso dessas transformações e interpretações [19].

Metodologias de desenvolvimento baseadas em modelos também têm sido utilizadas para o desenvolvimento de serviços e aplicações *web* [21–24]. Tais metodologias têm sido aplicadas a diferentes domínios do conhecimento, tais como *business intelligence* [25], sistemas de informação em saúde [26, 27] e manufatura de produtos conectados a serviços [28]. Porém, não temos conhecimento de metodologias de desenvolvimento baseadas em modelos específicas para o desenvolvimento de serviços de análise em bioinformática. Adicionalmente, as metodologias existentes não são adequadas para o desenvolvimento de serviços de análise a partir da adaptação de uma ferramenta de análise, dada a maior complexidade dessa tarefa.

Frente a esse cenário, a disponibilidade de metodologias e ferramentas de suporte que abstraiam a maior parte dos detalhes tecnológicos e permitam ao

bioinformata/biologista representar em alto nível de abstração os aspectos relevantes das ferramentas que desejam adaptar pode, potencialmente, beneficiar todo o domínio da bioinformática. Essas metodologias facilitariam o desenvolvimento sistemático e (semi) automático de um serviço adaptador para a ferramenta de interesse e de eventuais extensões para a integração desse serviço em um dado ambiente de análise.

1.2 Objetivos

O objetivo geral desse projeto é investigar o uso de abordagens de desenvolvimento baseadas em modelos para o suporte ao desenvolvimento e à execução programática de serviços *web* de análise em bioinformática. Neste sentido, este projeto tem os seguintes objetivos específicos:

O1 Investigar um modelo de referência para o desenvolvimento de serviços de análise em bioinformática, bem como uma abordagem de desenvolvimento baseada em modelos para serviços de análise para bioinformática a partir de ferramentas existentes;

O2 Investigar uma abordagem de desenvolvimento baseada em modelos para clientes e descrições de serviços de análise desenvolvidos, de modo a facilitar o uso desses serviços por parte de bioinformatas/biologistas;

O3 Prover uma infraestrutura de suporte ao desenvolvimento baseado em modelos de serviços de análise em bioinformática, bem como de clientes e descrições para esses serviços;

O4 Aplicar o processo de desenvolvimento proposto na reengenharia de um repositório de serviços de análise em bioinformática.

1.3 Metodologia

O desenvolvimento baseado em modelos facilita a obtenção de soluções de *software* para domínios específicos. Dessa maneira, uma metodologia baseada em modelos para o desenvolvimento de serviços *web* para bioinformática foi investigada. Esta metodologia facilita o desenvolvimento desses serviços por pessoas com conhecimentos computacionais potencialmente limitados, em particular quanto às tecnologias para o desenvolvimento *web*. Adicionalmente, essa metodologia provê suporte ao desenvolvimento de eventuais clientes e de extensões para o uso e a integração dos serviços criados em diferentes ambientes integrados de análise.

De modo a alcançar os objetivos deste trabalho, as seguintes atividades foram desenvolvidas ao longo deste projeto:

A1 **Revisão bibliográfica.** Esta atividade teve por objetivo definir a base conceitual necessária para este trabalho. Neste sentido, durante essa atividade foi realizado um estudo sobre a execução de atividades de análise em bioinformática, bem como da integração de ferramentas de análise por meio de ambientes de suporte. Também foi realizado um estudo bibliográfico sobre a arquitetura orientada a serviços e o estilo arquitetural REST para serviços *web*. Por fim, foi realizado um estudo sobre o suporte ao desenvolvimento de serviços de análise em bioinformática, bem como sobre o desenvolvimento baseado em modelos de serviços *web*;

A2 **Definição de um processo de desenvolvimento orientado a modelos para serviços de análise.** Esta atividade teve por objetivo definir os requisitos para a adaptação de uma ferramenta de análise por meio de um serviço RESTful. Durante esta atividade, também foi definido o ciclo de vida de uma atividade de análise provida por um serviço adaptador, bem como um processo de desenvolvimento baseado em modelos para serviços de análise, clientes desses serviços e suas descrições;

A3 **Definição de um modelo de referência para serviços de análise em bioinformática.** Esta atividade teve por objetivo definir um modelo de referência para guiar o desenvolvimento de serviços de análise em bioinformática. O modelo de referência proposto define como a interface *web* de um serviço de análise deve ser construída para o suporte à execução de uma atividade de análise, bem como também define a interação de um usuário e o comportamento esperado do serviço de análise durante a execução de instâncias dessa atividade;

A4 **Definição de uma arquitetura de metamodelagem base para o processo de desenvolvimento de um serviço de análise.** Esta atividade teve por objetivo definir os principais artefatos utilizados durante o processo de desenvolvimento baseado em modelos para serviços de análise, bem como as relações entre esses artefatos. Durante esta atividade também foi definido um conjunto de metamodelos utilizados durante esse processo de desenvolvimento;

A5 **Implementação de uma infraestrutura computacional para o suporte ao desenvolvimento de serviços de análise, bem como de clientes e descrições desses serviços.** Esta atividade teve por objetivo criar a infraestrutura de suporte à geração de serviços de análise adaptadores, de seus clientes e de suas descrições. Ao final dessa atividade, obtivemos um *framework* de suporte ao desenvolvimento desses serviços que abstrai os detalhes técnicos dessa implementação, bem como transformações modelo-para-texto que permitem obter clientes e descrições de serviço de maneira semiautomática;

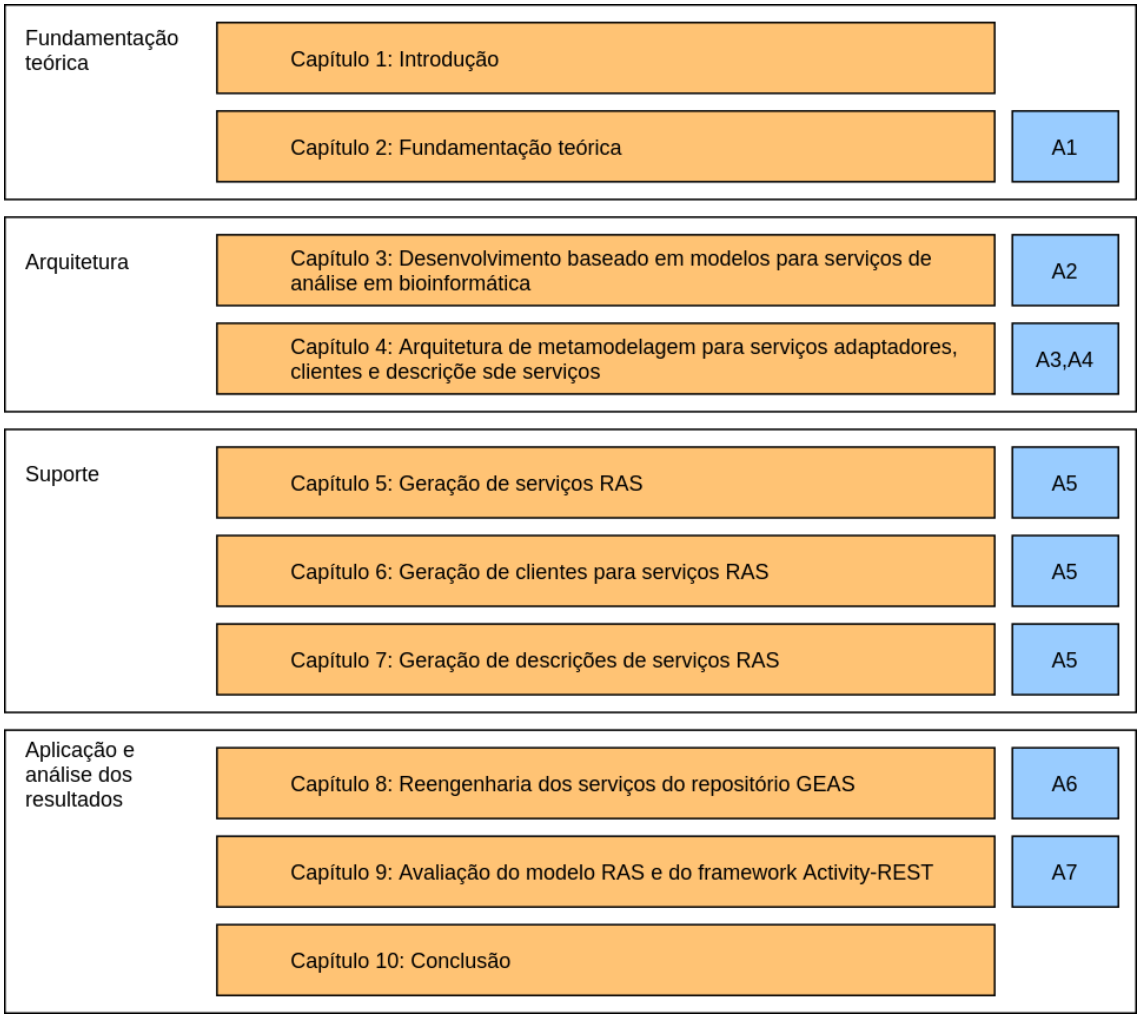
A6 **Aplicação do processo de desenvolvimento definido na reengenharia de um repositório de serviços de análise** Esta atividade teve por objetivo validar o processo de desenvolvimento de serviços de análise, seus clientes e suas descrições. Nesse sentido, realizamos a reengenharia de um conjunto de serviços de análise existentes no repositório GEAS [7] utilizando nosso processo de desenvolvimento baseado em modelos. Ao final dessa atividade, obtivemos um novo repositório contendo serviços de análise em conformidade ao modelo de referência proposto neste trabalho para o desenvolvimento de serviços de análise RESTful;

A7 **Avaliação da infraestrutura de suporte desenvolvida.** Esta atividade teve por objetivo avaliar o suporte desenvolvido para a criação de serviços de análise. Neste sentido, comparamos a maturidade dos serviços de análise desenvolvidos com nossa metodologia com os serviços de análise disponíveis em repositórios de serviços conhecidos. Também avaliamos o esforço necessário para o desenvolvimento de um serviço de análise usando o processo baseado em modelos proposto. Por fim, comparamos o desempenho dos serviços implementados por meio de nossa metodologia com os serviços de análise equivalentes existentes no repositório GEAS.

1.4 Organização do documento

A Figura 1 apresenta uma visão geral da estrutura desta tese e a relação de seus capítulos com as atividades de nossa metodologia. O restante desse documento está organizado da seguinte forma: o Capítulo 2 apresenta a fundamentação teórica utilizada neste trabalho, provendo uma visão geral sobre a arquitetura orientada a serviços, metodologias de desenvolvimento baseadas em modelos, o uso de serviços de análise em bioinformática e ambientes integrados de análise para bioinformática (Atividade A1); o Capítulo 3 apresenta uma conceitualização sobre a adaptação de uma ferramenta de análise por meio de um serviço *web*, requisitos que guiam o desenvolvimento de serviços de análise neste trabalho, bem como um processo de desenvolvimento baseado em modelos para a adaptação de uma ferramenta por meio de um serviço de análise (Atividade A2); o Capítulo 4 apresenta uma arquitetura de metamodelagem utilizada como base para este trabalho e um modelo de referência para serviços de análise em bioinformática (Atividades A3 e A4); os Capítulos 5 a 7 apresentam o suporte desenvolvido para a geração de um serviço adaptador para uma ferramenta de análise existente, o suporte desenvolvido para a geração de clientes de um serviço de análise e o suporte desenvolvido para a geração de descrições de serviços (Atividade A5); o Capítulo 8 apresenta um estudo de caso na reengenharia do repositório se serviços de análise GEAS utilizando o processo de

Figura 1 – Estrutura da tese



Fonte: Autoria própria.

desenvolvimento definidos por este trabalho (Atividade A6); o Capítulo 9 apresenta uma avaliação dos resultados de nosso trabalho, com foco no modelo de referência para serviços de análise definido e nos serviços de análise obtidos (Atividade A7); finalmente, o capítulo 10 discute as principais contribuições deste trabalho e apresenta as conclusões e os trabalhos futuros.

2 Fundamentação teórica

No domínio da bioinformática, análises *in silico* são realizadas sobre conjuntos de dados descrevendo processos biológicos. Essas análises consistem em um conjunto de atividades utilizadas para processar os conjuntos de dados existentes até a obtenção de uma dada informação de interesse. As atividades executadas durante uma análise costumam ser realizadas por meio do suporte de ferramentas de análise, as quais normalmente tomam a forma de programas e *scripts* implementados pelo pesquisador ou seus pares. Serviços *web* também têm sido criados para o suporte à execução de atividades de análise, provendo a capacidade de execução remota destas atividades e fomentando o reúso e a reprodutibilidade das análises em bioinformática. Abordagens de desenvolvimento baseadas em modelos têm sido utilizadas para abstrair os detalhes tecnológicos do processo de desenvolvimento de diferentes soluções de *software*, como, por exemplo, de serviços *web* simples.

Este capítulo apresenta uma síntese dos conceitos básicos necessários para o desenvolvimento deste trabalho e está estruturado da seguinte maneira: a Seção 2.1 apresenta uma visão geral do processo de análise *in silico* e de ambientes integrados de análise; a Seção 2.2 discorre sobre a arquitetura orientada a serviços e tecnologias relacionadas; por fim, a Seção 2.3 apresenta o desenvolvimento orientado a modelos e uma visão geral das abordagens de desenvolvimento de serviços *web* orientadas a modelo.

2.1 Análises *in silico*

Uma análise *in silico* consiste do processamento de conjuntos de dados que descrevem um determinado fenômeno biológico por meio do auxílio de computadores, de modo a obter uma determinada informação de interesse sobre esse fenômeno. Análises *in silico* tornaram-se mais importantes para as ciências biológicas conforme bancos de dados de informações biológicas tornaram-se disponíveis e os custos para o acesso e o processamento desses dados diminuiram. Tal fecilidade de acesso possibilitou o maior reúso de conjuntos de dados obtidos em experimentos anteriores e a combinação desses dados de novas formas (ou com novos conjuntos de dados) de modo a obter informações inéditas sobre os processos biológicos em estudo, fazendo melhor uso dos dados disponíveis e reduzindo o tempo para o desenvolvimento de pesquisas biológicas.

2.1.1 Análises *in silico* como atividades em um *workflow*

Uma análise *in silico* é muito semelhante, em sua definição, a um processo de negócio. Um *processo de negócio* consiste de um conjunto de atividades que representam os passos necessários para atingir um objetivo de interesse para uma organização [29]. Neste sentido, uma análise *in silico* pode ser vista como um processo de negócio cujo objetivo é a exploração sistemática de um conjunto de dados para obter uma resposta a um problema biológico. Durante uma análise *in silico*, o pesquisador executa um conjunto de atividades de análise para atingir um dado objetivo de pesquisa.

Uma atividade de análise consiste em um subprocesso executado sobre um conjunto de dados de entrada de modo a produzir um conjunto de dados de saída contendo alguma informação relevante para o pesquisador em um dado passo da análise. Conjuntos de dados de entrada e de saída, em geral, tomam a forma de um ou mais arquivos contendo os dados de interesse para uma atividade de análise ou os dados produzidos pela execução desta, respectivamente. Adicionalmente, a execução de uma atividade de análise pode ser guiada por um conjunto de parâmetros de execução, i.e., um conjunto adicional de informações utilizadas para guiar, de alguma maneira, o comportamento do processo de análise.

A execução de atividades de análise em bioinformática é frequentemente suportada por ferramentas computacionais adequadas. Estas ferramentas, geralmente implementadas como um programa de linha de comando, acessam os conjuntos de dados de entrada por meio do endereço em os arquivos que o compõem estão armazenados no sistema de arquivos local. De maneira semelhante, estas ferramentas armazenam o conjunto de dados de saída na forma de um ou mais arquivos informados por meio dos caminhos em que estes arquivos deverão ser encontrados no sistema de arquivos. Por fim, os parâmetros de execução são normalmente passados de maneira literal para a ferramenta de análise, durante a invocação desta ferramenta.

Conjuntos de dados de saída produzidos pela execução de uma dada atividade de análise podem, posteriormente, ser utilizados como conjuntos de dados de entrada de atividades de análise subsequentes. Dessa maneira, diferentes atividades de análise podem ser compostas para atingir o objetivo primário do processo de análise e obter uma informação biológica relevante. Durante essa composição, as ferramentas de análise que provêem suporte a atividades de análise subsequentes devem ser integradas de maneira adequada. Neste sentido, diferenças sintáticas e/ou semânticas dos arquivos que compõe os conjuntos de dados de entrada e saída passados a essas ferramentas subjacentes devem ser tratadas.

Um *workflow* consiste de uma facilitação, total ou parcial, de um dado processo de negócio [30]. Em um *workflow*, cada atividade do processo de negócio é descrita

de forma abstrata, denotando uma ação específica e parametrizada (processamento) que irá consumir dados de entrada e produzir dados de saída [31]. Adicionalmente, também as regras, os participantes e o fluxo de execução das atividades são descritas. Este *workflow* abstrato é, então, utilizado para realizar execuções concretas do processo de negócio.

Workflows proveem uma solução apropriada para a separação entre a lógica do processo de negócio e suporte (computacional) à execução de cada atividade deste processo. Ainda que *workflows* possam ser organizados e executados manualmente, estes geralmente são executados no contexto de um sistema computacional que suporte a automação procedural deste processo. Assim, um sistema de gerenciamento de *workflows* consiste de um sistema computacional que permite a um usuário definir, gerenciar e executar as diferentes atividades logicamente representadas por um *workflow* de forma ordenada [30].

2.1.2 Ambientes integrados de análise

Ambientes integrados de análise em bioinformática podem ser vistos como sistemas de gerenciamento de *workflows* que disponibilizam aos seus usuários um conjunto de atividades de análise, bem como auxiliam na interconexão e execução automática das atividades de interesse [8]. Estes ambientes auxiliam no processamento e na exploração interativos dos dados em análise por meio da integração facilitada de um conjunto de ferramentas. Neste sentido, ambientes integrados de análise buscam abstrair os processos necessários à integração das ferramentas e dados biológicos de modo a permitir que o usuário concentre-se na criação do *workflow* de análise e na descoberta da informação biológica de interesse. O armazenamento, o compartilhamento e o reúso de *workflows* de análise podem ser suportados pelo ambiente. Desse modo, estes *workflows* podem ser posteriormente reutilizados em novas análises e/ou avaliados por outros pesquisadores.

Diversos ambientes integrados de análise foram desenvolvidos para o domínio da bioinformática [8,11–15]. Estes ambientes podem ser classificados de acordo com a estratégia de integração utilizada e quanto a sua extensibilidade. Neste sentido, estes ambiente podem ser classificados em ambientes baseados em estratégias de integração local, ambientes baseados em estratégias de integração federadas ou distribuídas e ambientes baseados em estratégias de integração híbridas [4].

Ambientes baseados em estratégias de integração local são criados para integrar ferramentas que coexistem em uma mesma máquina. Por outro lado, ambientes baseados em estratégias de integração distribuídas focam no acesso a funcionalidades de ferramentas e a repositórios de dados armazenados de forma remota. Por fim, é possível que um ambiente possibilite a integração de ferramentas de análise locais e

ferramentas de análises remotamente distribuídas. Neste caso, o ambiente apresenta uma estratégia de integração híbrida.

As diferentes estratégias de integração possuem vantagens e desvantagens. Ambientes baseados em *estratégias de integração local* normalmente necessitam de esforço adicional por parte do usuário para a sua manutenção e atualização frente à evolução das ferramentas que integram estes ambientes [4]. Porém, o compartilhamento de dados entre as diversas ferramentas torna-se mais rápido, uma vez que as ferramentas e bases de dados integradas estão centralizadas em um único ponto. Por sua vez, ambientes baseados em *estratégias de integração distribuídas* geralmente necessitam de um esforço menor por parte do usuário para a atualização dos conjuntos de dados e das ferramentas em uso [4]. Neste caso, dados e ferramentas podem ser mantidos atualizados por seus próprios criadores (ou terceiros) e essa atualização é refletida em todos os usuários do recurso. Os mantenedores desses dados e ferramentas remotas podem cobrar pelo acesso a esses recursos, dividindo o custo de sua manutenção entre seus usuários e tendo a oportunidade de obter recursos financeiros para novas pesquisas. Adicionalmente, estratégias de integração distribuídas possibilitam alcançar maior escalabilidade e reúso de ferramentas nas análises [32], porém podem apresentar como gargalo a velocidade de compartilhamento dos dados entre os diversos recursos remotos.

Finalmente, ambientes baseados em *estratégias de integração híbridas* podem aproveitar vantagens de ambas as estratégias de integração anteriores. Por exemplo, uma estratégia de integração híbrida possibilita o uso de ferramentas locais para o compartilhamento rápido de grandes conjuntos de dados entre ferramentas, evitando o gargalo provocado pela largura da banda de rede, enquanto mantém a possibilidade de utilizar serviços remotos especializados e/ou não disponíveis como ferramentas locais [4].

Ambientes integrados de análise também podem ser classificados quanto a sua extensibilidade. Neste sentido, estes ambientes podem ser divididos em *ambientes fechados* e *ambientes abertos* [33]. Ambientes fechados disponibilizam um conjunto final de ferramentas para seus usuários. Assim, estes ambientes limitam o usuário ao uso das ferramentas disponibilizadas pelo desenvolvedor, o qual é o único capaz de adicionar novas funcionalidades. Por sua vez, ambientes abertos permitem ao usuário a adição de novas ferramentas por meio da criação de extensões ao ambiente. Estas extensões podem ser realizadas por meio da alteração do código fonte do ambiente, configurações adicionais ou pela inclusão de outros artefatos ao mesmo.

As diferentes estratégias de integração utilizadas pelos diferentes ambientes integrados de análise fazem com que métodos e artefatos próprios precisem ser desenvolvidos para suporte à extensão desses ambientes. Essa diversidade é um desafio

ao maior compartilhamento de ferramentas entre esses ambientes. Adicionalmente, a extensão desses sistemas demanda conhecimentos tecnológicos específicos que, muitas vezes, são desconhecidos de um bioinformata que queria utilizar uma nova ferramenta de maneira integrada aos ambientes disponíveis.

2.1.3 Plataforma *Galaxy*

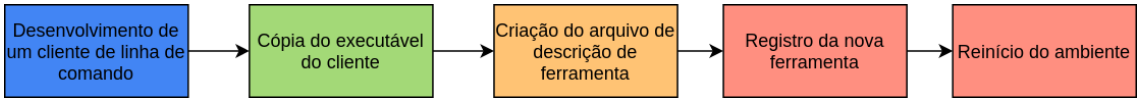
A plataforma Galaxy [14, 34] é um ambiente integrado de análise de código aberto para análises epigenômicas. Galaxy provê um ambiente de análise colaborativo transparente, com suporte à pesquisa reproduzível e acessível a usuários sem conhecimentos tecnológicos específicos. Neste sentido, Galaxy provê uma interface *web* gráfica para a criação das análises, bem como suporte para o acesso à plataforma por meio de dispositivos móveis [35].

O ambiente Galaxy é aberto à extensão e utiliza uma estratégia de integração local. Uma ferramenta do Galaxy é essencialmente um adaptador de um programa executável em linha de comando. Adaptadores Galaxy são tipicamente definidos por meio de arquivos XML que descrevem entradas e saídas da ferramenta adaptada. Estas extensões também descrevem a interface *web* disponibilizada ao usuário para o uso da ferramenta no ambiente Galaxy. Neste sentido, o ambiente Galaxy suporta nativamente apenas ferramentas para as quais todos os parâmetros para a execução são passados previamente à execução. Ferramentas que necessitem de interações entre usuário e ferramenta durante a execução, tais como ferramentas com interfaces textuais ou gráficas, não são suportadas.

Embora o programa invocado pelo ambiente deva residir na mesma máquina que o servidor Galaxy, a execução desse programa pode acessar remotamente outros recursos. Neste sentido, uma ferramenta local pode ser criada de modo a receber os parâmetros e dados da análise, enviá-los a um serviço externo e recuperar o resultado. Adicionalmente, esforços para a criação de extensões à plataforma de modo a permitir o uso de serviços *web* são conhecidos [16].

A Figura 2 apresenta os passos necessário para a extensão do ambiente Galaxy com a adição de uma nova ferramenta de análise implementada como um serviço *web*.

Figura 2 – Etapas da extensão do ambiente Galaxy por meio de um serviço de análise



Fonte: Autoria própria.

O primeiro passo consiste no desenvolvimento de um cliente de linha de

comando para o serviço, no qual o desenvolvedor deve implementar uma ferramenta que receberá os dados de análise a partir de argumentos de linha de comando e invocará (remotamente) o serviço *web* de interesse. Em seguida, o segundo passo consiste da cópia do executável do cliente desenvolvido para o diretório `tools/` da instalação do Galaxy.

O terceiro passo consiste da criação do arquivo de descrição de ferramenta. Neste arquivo, informações sobre a ferramenta de análise devem ser representadas em XML. Atributos do cliente desenvolvido devem ser declarados, tais como identificador e descrição textual, tipos de entradas e saídas, o comando a ser executado e o interpretador utilizado para executar este comando. Adicionalmente, testes para a invocação deste cliente podem ser definidos.

O quarto passo consiste na configuração do ambiente Galaxy de modo a reconhecer a existência da nova ferramenta. Neste sentido, uma entrada contendo o nome da ferramenta, seu identificador e o caminho para o arquivo de definição de ferramenta deve ser adicionada ao arquivo `tool_conf.xml`.

Por fim, o passo final consiste do reinício da instância atual do Galaxy. A nova ferramenta será reconhecida durante a próxima inicialização do sistema e disponibilizada ao uso.

Ferramentas Galaxy podem ser compartilhadas fazendo uso de repositórios de ferramentas (*tool sheds*). Um *tool shed* consiste de um repositório versionado de ferramentas Galaxy publicadas por desenvolvedores dessas ferramentas. Este repositório inclui os arquivos de definição de ferramentas (adaptadores Galaxy), bem como o executável destas ferramentas adaptadas. *Tool sheds* podem ser públicos ou privados.

2.2 Arquitetura orientada a serviços e tecnologias de serviços *web*

O uso de serviços *web* para a disponibilização das funcionalidades de uma ferramenta de análise é uma solução que está tornando-se popular no domínio biomédico. Porém, o desenvolvimento de serviços *web* maduros depende do domínio da arquitetura orientada a serviços, bem como das restrições e tecnologias associada ao estilo arquitetural utilizado. Esta seção apresenta uma visão geral da arquitetura orientada a serviços e sua concretização por meio de serviços *web*, bem como das principais linguagens para a descrição de serviços *web* disponíveis.

2.2.1 Arquitetura orientada a serviços

Entidades, quer sejam pessoas ou empresas, podem desenvolver soluções computacionais para o suporte às atividades de seus processos de negócio. Diferentes entidades podem executar processos de negócio semelhantes, de maneira que soluções equivalentes podem ser desenvolvidas. Porém, o desenvolvimento e a manutenção de novas soluções demanda investimentos que, por vezes, podem ser amortizados por meio do uso compartilhado das soluções desenvolvidas por diferentes entidades. De forma a promover esta visão, a *Arquitetura Orientada a Serviços* (*Service Oriented Architecture* ou, simplesmente, SOA) foi definida como uma arquitetura de referência para a organização e utilização de recursos distribuídos sob diferentes domínios administrativos [36].

Os principais conceitos presentes em uma arquitetura orientada a serviços incluem *serviço*, *cliente de serviço*, *provedor de serviço*, *descrição de serviço*, *serviço de diretório*, *política* e *contrato*. Um *serviço* consiste em uma abstração de um recurso capaz de executar um conjunto determinado de atividades e prover uma dada funcionalidade [36, 37]. De acordo com a arquitetura orientada a serviços, as soluções computacionais desenvolvidas por diferentes entidades são disponibilizadas como serviços localizados remotamente e invocáveis sob demanda. *Cliente de serviço* representa uma entidade que utiliza as funcionalidades providas por um serviço de modo a atingir um dado objetivo de negócio. A invocação de um serviço produz um efeito visível para o cliente, tal como a recuperação de uma informação e/ou a alteração no estado de um dado recurso.

Provedor de serviço representa uma entidade que implementa os recursos computacionais necessários ao suporte funcional de um ou mais serviços. Uma mesma funcionalidade pode ser implementada de diferentes maneiras, utilizando diferentes tecnologias, bem como por diferentes provedores. No entanto, detalhes sobre a implementação de um serviço são normalmente ocultos e seus clientes conhecem apenas a descrição do serviço disponibilizada pelo provedor [36]. Assim, a *descrição de um serviço* apresenta formalmente a informação necessária para o uso de um serviço e os efeitos esperados deste uso, bem como a localização do recurso computacional que o suporta. Adicionalmente, uma descrição pode apresentar outros atributos funcionais e não-funcionais do serviço descrito. A descrição de um serviço padroniza como clientes e provedores interagem. Dessa maneira, a arquitetura orientada a serviços provê um desacoplamento entre funcionalidade e implementação de um serviço, tornando clientes indiferentes à escolha dos provedores que o concretizam.

A descrição de um serviço precisa ser conhecida pelos seus clientes de modo que este serviço possa ser utilizado. Um *serviço de diretório* normalmente é utilizado para compartilhar descrições de serviços com clientes potenciais. Por intermédio de

um serviço de diretório, provedores podem publicar as descrições dos serviços que implementam e torná-las acessíveis aos clientes sob identificadores padronizados. O serviço de diretório pode, também, prover facilidades para a pesquisa e a descoberta de serviços segundo categorias e/ou atributos definidos em sua descrição. Dessa maneira, clientes podem pesquisar novos serviços de acordo com suas necessidades, bem como acessar a descrição destes serviços por meio do serviço de diretório.

Políticas e contratos podem interferir no uso de um serviço por seus clientes. Uma *política* representa uma restrição ao uso de um serviço definida por seu provedor ou por seu cliente de maneira unilateral. Políticas podem cobrir diversos aspectos do uso do serviço, como frequência máxima desse uso por um cliente e a obrigação de troca de mensagens encriptadas. Por sua vez, um *contrato* descreve um conjunto de valores mensuráveis acordados pelos participantes da interação os quais representam expectativas e responsabilidades durante a interação, por exemplo, atributos de qualidade do serviço e custo de uso. O processo de definição de um contrato pode envolver diferentes aspectos comerciais ou legais, os quais não fazem parte do escopo deste modelo arquitetônico.

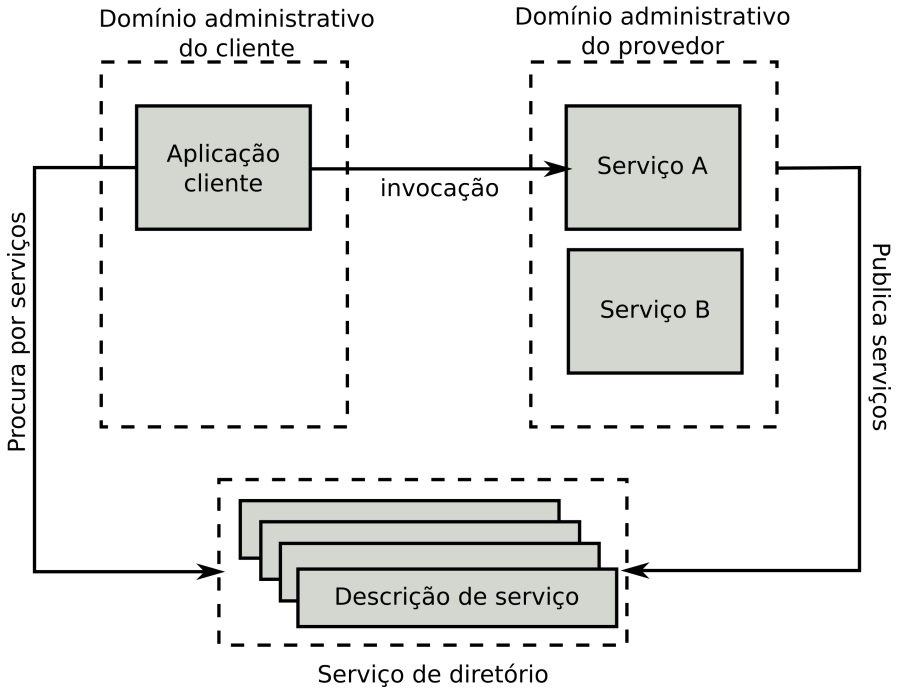
A Figura 3 ilustra alguns elementos de uma arquitetura orientada a serviços. Cliente (Aplicação cliente) e serviços residem em diferentes domínios administrativos, essencialmente, máquinas diferentes em uma rede de computadores. De modo a possibilitar a interação entre clientes e serviços providos, o provedor de serviços torna público as descrições destes serviços por meio do serviço de diretório. O cliente consulta o serviço de diretórios em busca de serviços que correspondam às suas necessidades, acessando a descrição de um serviço de interesse. Por meio desta descrição, o cliente é capaz de invocar o serviço desejado e obter o efeito que necessita.

2.2.2 Serviços *web*

Um serviço *web* consiste em um sistema computacional desenvolvido para prover interação máquina a máquina por meio de padrões da *web* [37, 38], tais como o *HyperText Transfer Protocol* (HTTP) [39] e a *Extensible Mark-up Language* (XML) [40]. Dessa maneira, serviços *web* representam uma solução tecnológica para a implementação da arquitetura orientada a serviços.

O uso dos padrões abertos da *web* permite a proliferação livre desses serviços de maneira independente da plataforma de implementação utilizada e/ou tecnologias proprietárias. Neste sentido, serviços *web* são normalmente desenvolvidos segundo duas abordagens distintas: serviços SOAP e serviços RESTful.

Figura 3 – Visão geral das entidades em uma arquitetura SOA.



Adaptado de [33]

2.2.2.1 Serviços SOAP

Simple Object Access Protocol (SOAP) consiste de um protocolo definido pelo W3C para a representação de informação compartilhada em ambientes distribuídos e descentralizados [41–44]. SOAP define como dados estruturados podem ser transmitidos por meio de mensagens representadas em XML entre os participantes de uma arquitetura orientada a serviços.

O protocolo SOAP define apenas um paradigma de troca de mensagem única entre participantes do sistema distribuído. Porém, aplicações podem utilizar o protocolo para criar padrões de interação mais complexos, como requisição-resposta ou requisição e múltiplas respostas. Adicionalmente, SOAP assume que uma mensagem pode transitar por diversos nós intermediários até alcançar o destinatário final. Neste sentido, SOAP permite definir regras de como os nós intermediários devem processar uma mensagem recebida e modificá-la antes de retransmiti-la. Essas regras podem ser utilizadas por aplicações que utilizam o protocolo para implementar mecanismos de controle próprios, como mecanismos de segurança e transações.

Uma mensagem SOAP consiste em um conjunto de informações representadas em XML. Estas informações são encapsuladas por meio de um envelope (elemento `env:Envelope`), o qual possui duas partes distintas: corpo (elemento `env:Body`) e cabeçalho (elemento `env:Header`). O corpo de uma mensagem SOAP é um elemento obrigatório que carrega a carga útil da mensagem. Por sua vez, o cabeçalho de uma

mensagem SOAP é um elemento opcional utilizado para a extensão da mensagem pela associação de informação adicional que não faz parte da carga útil da mensagem. Por exemplo, cabeçalhos SOAP podem ser usados para a troca de informação utilizada por mecanismos de controle da interação entre os participantes ou para enviar informação contextual que deve ser considerada durante o processamento da mensagem.

O conteúdo do cabeçalho e do corpo de uma mensagem SOAP são específicos de aplicação e, portanto, não são definidos pelo protocolo. Porém, SOAP define alguns atributos para os elementos contidos nesses blocos, bem como define a maneira que os elementos que possuam tais atributos devem ser processados em um nó na rede.

Mensagens SOAP podem ser trocadas entre os participantes de um sistema distribuído por meio de diferentes protocolos de troca de mensagens, tais como o *Simple Mail Transfer Protocol* (SMTP) [45], o *Transmission Control Protocol* (TCP) [46] e o *Hypertext Transfer Protocol* (HTTP) [39]. Dessa maneira, cabe ao desenvolvedor da aplicação escolher o protocolo utilizado para a troca de mensagem e considerar como problemas dessa escolha devem ser tratados. Porém, o HTTP representa o padrão *de facto* para a troca de mensagens SOAP.

Descrições de serviços SOAP são representadas por meio da linguagem *Web Service Description Language* (WSDL) [47–49]. WSDL consiste em uma linguagem baseada em XML que descreve serviços *web* em um nível abstrato e em um nível concreto. Em cada nível, a linguagem WSDL contém um conjunto de construtos que proveem suporte ao reúso de uma descrição, bem como à separação de diferentes aspectos do projeto de um serviço.

WSDL permite também definir um serviço *web* de forma concreta a partir da definição de *bindings*, *endpoints* e por meio da concretização do serviço como um conjunto funcional. Neste sentido, *Bindings* definem formatos de serialização e transporte para uma ou mais *interfaces*. *Endpoints* associam os endereços de rede utilizados para o acesso a um serviço com *bindings* que o serviço implementa. Por fim, serviços agrupam *endpoints* que, em conjunto, implementam uma *interface* definida abstratamente.

Descrições WSDL podem ser publicadas em serviços de diretório *Universal Description, Discovery and Integration* (UDDI) [50]. UDDI representa um dos pilares do padrão de interoperabilidade de serviços *web* (*Web Service Interoperability* ou WS-I) definido pela *Organization for the Advancement of Structured Information Standards* (OASIS) [51]. UDDI define um serviço de diretório independente de plataforma e acessível por mensagens SOAP que pode ser utilizado para a publicação e descoberta de descrições WSDL.

Um serviço de diretório UDDI permite a consulta e a descoberta de informação sobre serviços *web* usando três diferentes componentes: um *serviço de páginas amarelas*, o qual provê a descoberta de serviços por meio de uma taxonomia padrão; um *serviço de páginas brancas*, o qual provê informação de contato de um provedor de serviços e permite a descoberta dos serviços publicados por este provedor; e um *serviço de páginas verdes*, o qual provê informação sobre a concretização necessária para o acesso e uso de um serviço de interesse, como os *bindings* e *endpoints* desse serviço. Dessa maneira, um cliente pode buscar por novos serviços de interesse por meio de sua classificação taxonômica ou seu provedor, bem como acessar diretamente a descrição WSDL de um serviço publicado.

2.2.2.2 Serviços RESTful

REpresentational State Transfer (REST) [52, 53] representa o estilo arquitetural padrão *de facto* para o desenvolvimento de novos serviços *web*. Este estilo arquitetural foi desenvolvido de modo a prover um modelo de interação para a *web* e permitir o desenvolvimento de aplicações escaláveis, com baixo acoplamento entre cliente e servidor, bem como eficientes no uso da rede.

REST é caracterizado por quatro restrições arquiteturais [52, 53]: i) separação clara dos componentes da aplicação entre clientes e servidores; ii) respostas provindas de um servidor a uma dada requisição de um cliente são passíveis de serem armazenadas em *cache*; iii) interações entre cliente e servidor devem ser independentes do estado interno do servidor; iv) uso de uma interface uniforme para a manipulação dos recursos disponíveis.

REST define que recursos providos por um servidor devem ser facilmente identificáveis e devem ser manipulados pelo cliente via representações. Neste sentido, um recurso consiste de uma informação nomeável qualquer, como “usuário”, “coleção de dados de análise” ou “resultado do processamento”. Todo recurso é identificado por meio de um *Uniform Resource Identifier* (URI). Recursos podem ser divididos em recursos simples e coleções. Recursos simples representam uma dada entidade ou objeto que é acessível por um URI. Por sua vez, coleções agrupam outros recursos logicamente associados sob um mesmo URI, permitindo a navegação por este conjunto de uma maneira adequada.

Clientes não acessam os recursos disponíveis diretamente. Neste sentido, um cliente que acessa um recurso recebe apenas uma representação do estado atualizado do mesmo. O cliente manipula a representação recebida e, se for de seu interesse, retorna a representação atualizada para o servidor de modo que a manipulação seja refletida no estado do recurso original. REST não define o formato das representações trocadas entre cliente e serviço, de forma que um dado recurso pode ser representado

de diferentes maneiras. Por exemplo, um recurso pode ser representado e manipulado como um documento XML, um objeto JSON [54] ou um objeto YAML [55].

REST também define que as requisições enviadas por um cliente a um servidor devem ser autodescritivas e independentes de informações providas por requisições anteriores. Dessa maneira, o servidor não mantém estado interno em relação a um dado cliente (*stateless*) e o cliente torna-se responsável por manter a informação do estado da interação.

Por fim, o estilo arquitetural REST define que ligações de hipermídia devem ser utilizadas como motor de aplicação (*Hypermedia As The Engine Of Application State*, o HATEOAS). Segundo o princípio de HATEOAS [52], ligações de hipermídia são incluídas nas respostas de um serviço. Estas ligações associam identificadores padronizados ao endereço de recursos e operações acessíveis ao cliente do serviço em um dado momento. O cliente do serviço pode utilizar os identificadores retornados para selecionar os endereços retornados e navegar pelos recursos do serviço. Dessa maneira, o princípio de HATEOAS permite que estes clientes interajam com o serviço sem conhecer de antemão os recursos disponibilizados. Adicionalmente, em interações mais longas e que demandam diversos ciclos de requisição e resposta para atingir um dado objetivo, o cliente pode identificar o estado da interação ao inspecionar as ligações de hipermídia retornadas a cada resposta. Um serviço *web* que implemente todas essas restrições e capacidades é dito *RESTful*.

Os métodos definidos pelo protocolo HTTP representam a interface uniforme utilizada por serviços RESTful. Neste sentido, os métodos HTTP POST, GET, PUT, PATCH e DELETE são utilizados para representar semanticamente operações de criação, recuperação, atualização e remoção dos recursos disponíveis (operações CRUD). Parte dessas operações são definidas como operações idempotentes, ou seja, operações que podem ser aplicadas repetidas vezes sem que o resultado se altere após a primeira aplicação. Adicionalmente, essas operações podem ser definidas de forma diferente quando aplicadas sobre recursos ou sobre coleções.

Embora requisições RESTful compulsoriamente sejam autodescritivas e independentes de informação anterior, por vezes uma única requisição não é suficiente para alcançar um dado objetivo de negócio. Neste sentido, um conjunto de interações entre o cliente e os provedores que disponibilizam os recursos RESTful de interesse podem ser necessárias, resultando em uma conversa RESTful [56]. Uma conversa RESTful define um conjunto de interações RESTful entre um cliente e um ou mais provedores de serviço necessárias para atingir um dado objetivo de negócio. Uma conversa RESTful ocorre de maneira independente de estado para o(s) provedor(es) envolvido(s). O cliente mantém ciência do estado dos recursos acessados e continua a conversa ao seguir as ligações de hipermídia incluídas nas representações destes

recursos. A conversa prossegue até que o objetivo de negócio seja alcançado. Dessa maneira, interações mais complexas podem ocorrer a partir de um conjunto de interações básicas.

2.2.3 Descrição de serviços *web*

Diversos padrões foram propostos para a modelagem e a descrição de diferentes aspectos de um serviço *web*. Por exemplo, a interface de um serviço *web* pode ser modelada utilizando a *Web Service Description Language* (WSDL) *a priori* da implementação desses serviço ou descrita por documentos nessa linguagem após o serviço ser implementado. WSDL é uma linguagem de descrição de serviços baseada em XML definida pelo *World Wide Web Consortium* (W3C). Inicialmente focada na descrição de serviços SOAP, a segunda versão da linguagem WSDL também incluiu elementos necessários para a descrição de serviços RESTful.

A linguagem WSDL descreve um serviço *web* de forma abstrata a partir da definição de mensagens, operações e *interfaces* associadas a este serviço. Adicionalmente, WSDL descreve uma mensagem de maneira independente do formato de serialização específico utilizado durante a troca de mensagens. Esta descrição é realizada por meio um sistema de definição de tipos, normalmente *XML Schema Definition* (XSD) [57, 58]. Uma operação define um padrão de troca de mensagens, identificando uma sequência de mensagens enviadas ou recebidas entre os participantes da interação. Por fim, uma *interface* agrupa um conjunto de operações sem comprometimento com o formato de serialização e/ou transporte de mensagens.

A linguagem WSDL é definida para ser representada por meio de XML, o que torna esta linguagem facilmente processável por máquinas. Porém, a necessidade de usar *tags* de abertura e fechamento de cada elemento, imposta pelo XML, bem como a necessidade de importar um conjunto de *namespaces* padrões para os elementos básicos da linguagem, tornam WSDL um tanto verbosa. Neste sentido, desenvolvedores de serviços em geral preferem utilizar linguagens (textuais) mais sucintas e compreensíveis aos humanos. De modo a suprir esta necessidade, iniciativas para a definição de linguagens para a descrição e documentação de serviços RESTful de maneira mais adequada à compreensão humana surgiram.

Uma dessas iniciativas é representada pela linguagem de descrição de serviço OpenAPI, padronizada pela *OpenAPI Initiative* [59]. A *OpenAPI Initiative* consiste de um consórcio liderado pela *The Linux Foundation* que busca definir uma interface padrão e aberta para a descrição, a documentação e a descoberta de APIs RESTful de forma compreensível por máquinas e seres humanos. A *OpenAPI Initiative* adotou a linguagem Swagger [60], definida inicialmente pela *SmartBear Software*, como a linguagem padrão para a descrição de serviços RESTful, renomeando essa linguagem

como OpenAPI. Atualmente, OpenAPI é o padrão *de facto* para a descrição de serviços RESTful, e extenso suporte ferramental é provido para o desenvolvimento e o uso de descrições OpenAPI no desenvolvimento de *software*.

Outras linguagens para descrição de serviços RESTful foram propostas, porém tiveram pouca adoção. Por exemplo, a *Web Application Description Language* (WADL) [61] é uma linguagem baseada em XML utilizada na descrição de serviços RESTful. WADL modela um serviço por meio dos recursos que o compõe, permitindo definir os métodos HTTP aceitos por estes recursos e os formatos de representação passíveis de serem utilizados em requisições e respostas. Embora WADL tenha sido criada especificamente para a descrição de serviços RESTful, esta linguagem nunca chegou ao *status* de padrão recomendado, tendo sido substituída pela linguagem WSDL (versão 2.0) para este propósito e, mais recentemente, pela OpenAPI.

Dado que WSDL e OpenAPI representam as principais soluções para a descrição de um serviço *web*, uma visão geral dessas linguagens será apresentada em seguida.

2.2.3.1 Visão geral de uma especificação WSDL

Uma especificação WSDL consiste de um documento XML, com estrutura bem definida, utilizado para descrever um dado serviço *Web*. O elemento raiz de uma especificação WSDL (`<wsdl:description>`) agrega um conjunto de elementos de modo a descrever as principais características de um serviço *Web*. Tais elementos podem ser definidos internamente ao elemento `<wsdl:description>` ou podem ser definidos em documentos externos e serem, então, referenciados internamente à especificação WSDL.

O elemento `<wsdl:description>` define o *namespace* padrão para a descrição do serviço. Um *namespace* XML representa um particionamento lógico dos elementos presentes em um conjunto de documentos XML, utilizado para definir referências unívocas a esses elementos. O *namespace* padrão de um documento é indicado por meio do atributo `@targetNamespace`, contendo como valor um identificador universal de recurso (URI) alvo. Recomenda-se que o URI alvo do *namespace* padrão seja o mesmo URI pelo qual o documento poderá ser encontrado por seus usuários. Adicionalmente, o elemento `<wsdl:description>` deve informar também um conjunto de *namespaces* XML cujos elementos são importados e utilizados no documento. O uso de elementos de outro *namespace* XML é indicado por meio de um atributo `@xmlns:`*`id`*`="`*`NamespaceURI`*`"`, no qual *`id`* define um identificador local para o *namespace* e *`NamespaceURI`* indica o URI associado. O identificador local de um *namespace* consiste de uma *string* alfanumérica que pode ser definida pelo desenvolvedor de uma dada especificação WSDL de maneira independente

aos *namespaces* definidos em outros documentos WSDL. Isso ocorre pois cada identificador local é válido apenas nos contexto dos elementos aninhados ao elemento que importa o *namespace* referenciado. Neste sentido, elementos em um mesmo *namespace* podem ser referenciados por meio de diferentes identificadores locais em dois documentos XML.

A Tabela 1 apresenta o conjunto de *namespaces* necessários para a descrição de um serviço RESTful utilizando WSDL. Nessa tabela são apresentados o identificador local escolhido para cada *namespace*, seu URI e uma visão geral dos elementos definidos por ele. Outros *namespaces* podem ser definidos de acordo com as necessidades de uma dada especificação WSDL, por exemplo, para o reúso de tipos de mensagens cliente-serviço definidos em documentos externos.

Tabela 1 – *Namespaces* utilizado em uma especificação WSDL.

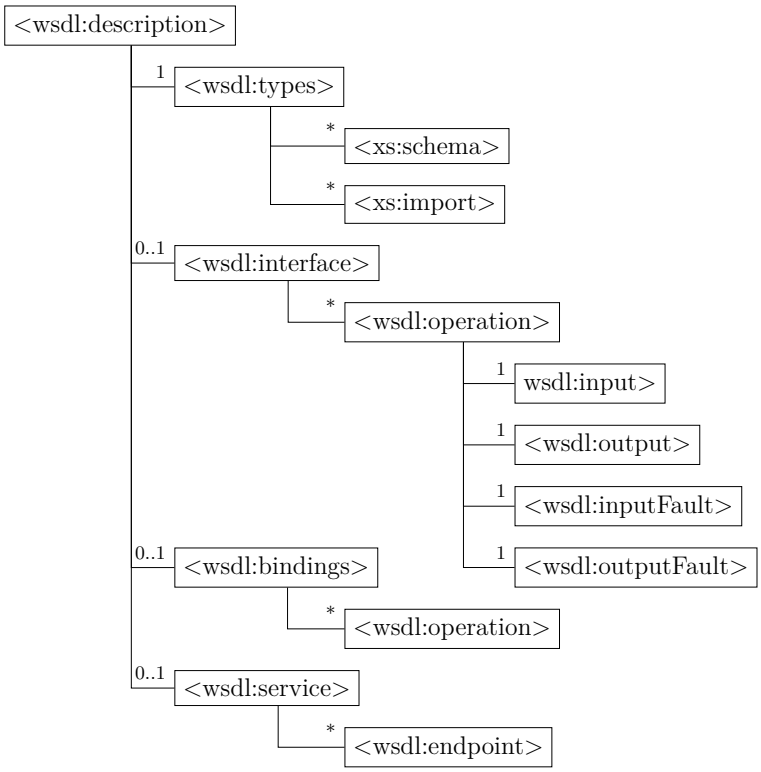
Ident. local	Namespace URI	Descrição
`wsdl`	`http://www.w3.org/ns/wsdl`	Define a estrutura geral de uma descrição WSDL.
`xs`	`http://www.w3.org/2001/XMLSchema`	Define a estrutura de um documento XSD.
`whttp`	`http://www.w3.org/ns/wsdl/http`	Define um conjunto de elementos XML utilizados para a descrição de operações executáveis por meio de um modelo de interação HTTP / REST.
`wsdlx`	`http://www.w3.org/ns/wsdl-extension`	Define um conjunto de atributos que estendem os elementos de uma especificação WSDL.
`tns`	*`URI de acesso à descrição WSDL`*	Define um indicador local para o *namespace* padrão dos elementos presentes na descrição. Esse mesmo URI deve estar indicada no atributo `targetNamespace` do elemento raiz da descrição WSDL.

Fonte: Autoria própria.

Os elementos de `<wsdl:description>` incluem a definição dos tipos XML das mensagens trocadas entre cliente e serviço (`<wsdl:types>`), a definição abstrata das

operações providas pelo serviço (`<wsdl:interface>`), a definição da concretização das operações abstratas (`<wsdl:binding>`) e os endereços bases de acesso ao serviço (`<wsdl:service>`). A Figura 4 apresenta uma visão geral dos principais elementos em uma especificação WSDL como uma árvore de elementos XML aninhados.

Figura 4 – Visão geral da estrutura de uma especificação WSDL.



Fonte: Autoria própria. Elementos XML presentes na especificação WSDL são representados por meio de retângulos nomeados. Uma aresta entre um elemento A superior e um elemento B inferior representa que o elemento B é um elemento definido no contexto de A. Essas arestas possuem rótulos representando a cardinalidade esperada para o elemento B em A. Neste sentido, `1` representa que A deve possuir 1, e apenas 1, elemento B; `0..1` representa que A pode possuir 1, e apenas 1, elemento B; por fim, `*` representa que A pode possuir qualquer número natural de elementos B.

O elemento `<wsdl:types>` define a estrutura dos tipos de mensagens trocadas entre cliente e serviço utilizando a linguagem XSD. Este elemento deve estar presente em uma especificação WSDL, podendo tanto definir um conjunto de tipos XML para a representação dessas mensagens por meio de declarações `<xs:schema>` quanto importar tais definições de outros documentos XSD por meio de declarações `<xs:import>`.

O elemento `<wsdl:interface>` define a interface abstrata do serviço. Neste sentido, este elemento inclui um conjunto de elementos `<wsdl:operation>`, os quais declaram operações abstratas providas por um serviço aos seus clientes. Cada elemento `<wsdl:operation>` pode incluir um elemento `<wsdl:input>` e um elemento `<wsdl:-`

output>. O elemento <wsdl:input> referencia um tipo XML para representar o tipo das informações enviadas em uma mensagem de requisição. O tipo referenciado pelo elemento <wsdl:input> deve ter sido definido em <wsdl:types>. De maneira semelhante, o elemento <wsdl:output> referencia um tipo XML, também definido em <wsdl:types>, para representar o tipo das informações recebidas em uma mensagem de resposta quando a operação é executada com sucesso. Adicionalmente, é possível definir a resposta de uma operação frente à ocorrência de uma falha ou exceção. Neste sentido, um elemento <wsdl:fault> é definido pelo elemento <wsdl:interface> e referenciado por elementos <wsdl:infault> e <wsdl:outfault> presentes na definição da operação, de modo a sinalizar falhas de invocação ou execução da operação.

O elemento <wsdl:binding> define a forma na qual cada operação presente no elemento <wsdl:interface> pode ser concretizada. Neste sentido, <wsdl:binding> inclui também um conjunto de elementos <wsdl:operation>. Estes elementos referenciam os <wsdl:operation> presentes na definição da interface do serviço, associando a estes elementos diferentes tipos de informação.

O atributo @type de <wsdl:binding> permite associar um protocolo de transporte à concretização da operação. Operações que utilizam o protocolo HTTP podem definir ainda o método utilizado para o acesso à operação (atributo @whttp:method), bem como a localização do recurso que a provê (atributo @whttp:location) tendo por base o endereço do serviço. O atributo @whttp:location pode conter referências a elementos do conjunto de informações da requisição definido anteriormente por um elemento <wsdl:input>. Essas referências são representadas no valor de @whttp:location por meio dos nomes desses elementos entre chaves, e.g. {*nome*}. Estas referências serão substituídas pelos conteúdos dos elementos previamente à requisição ao serviço. Adicionalmente, atributos @whttp:inputSerialization e @whttp:outputSerialization podem ser usados para associar o formato de serialização esperado para a requisição e a resposta à concretização da operação por meio de tipos MIME.

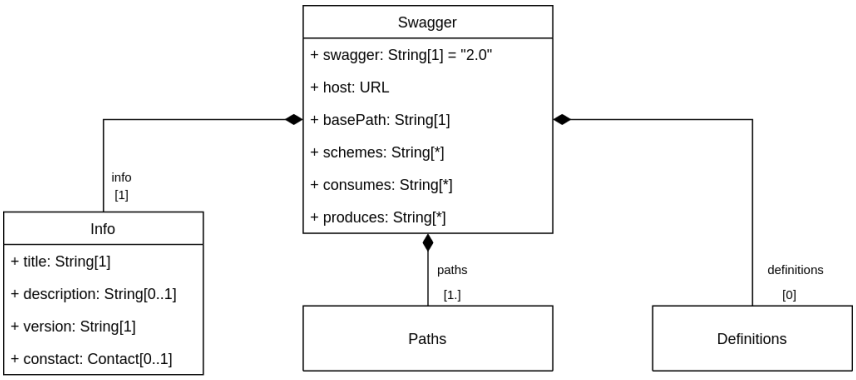
Por fim, o elemento <wsdl:service> provê os endereços bases para o acesso concreto ao serviço. Este elemento indica quais operações abstratas o serviço provê por meio de uma referência ao elemento <wsdl:interface> definido anteriormente. Adicionalmente, <wsdl:service> contém um ou mais elementos de endereçamento do serviço (<wsdl:endpoint>). Um elemento <wsdl:endpoint> provê a definição do endereço base do serviço, associando-o a uma concretização definida anteriormente em <wsdl:binding>. Este endereço base é formado pelo identificador de protocolo de transporte, endereço de rede, porta e caminho do recurso base do serviço.

2.2.3.2 Visão geral de uma descrição OpenAPI

OpenAPI especifica uma linguagem para a descrição e a documentação de serviços *web* RESTful com seus modelos representados em JSON [54] ou YAML [55]. A Figura 5 apresenta uma visão geral das metaclasses raízes de uma descrição OpenAPI versão 2.0. A metaclasse `Swagger` representa a descrição OpenAPI versão 2.0. Esta metaclasse possui um conjunto de atributos que representam informações básicas da descrição de um serviço RESTful. Dentre os atributos definidos em `Swagger` são encontrados atributos para descrever a versão OpenAPI utilizada (atributo `swagger`, que deve conter o valor `2.0`), o nome de domínio ou IP da máquina hospedeira (atributo `host`), o caminho base do serviço (atributo `basePath`), os tipos de comunicação possíveis de serem realizadas com o serviço descrito (atributo `schemes`) e os formatos de representação consumidos e produzidos pelo serviço (atributos `consumes` e `produces`, respectivamente).

`Swagger` possui referências a elementos que vão prover informações adicionais sobre o serviço (referência `info`), a identificação dos recursos disponíveis para serem acessados (referência `paths`), e a definição da estrutura e dos tipos de dados utilizados pelo serviço (referência `definitions`). A metaclasse `Info` representa uma informação adicional sobre o serviço descrito. `Info` permite definir um título, uma versão e uma descrição textual para o serviço (atributos `title`, `description` e `version`, respectivamente), bem como o contato do responsável pela manutenção do serviço (atributo `contact`).

Figura 5 – Visão geral das metaclasses raiz de uma descrição OpenAPI.



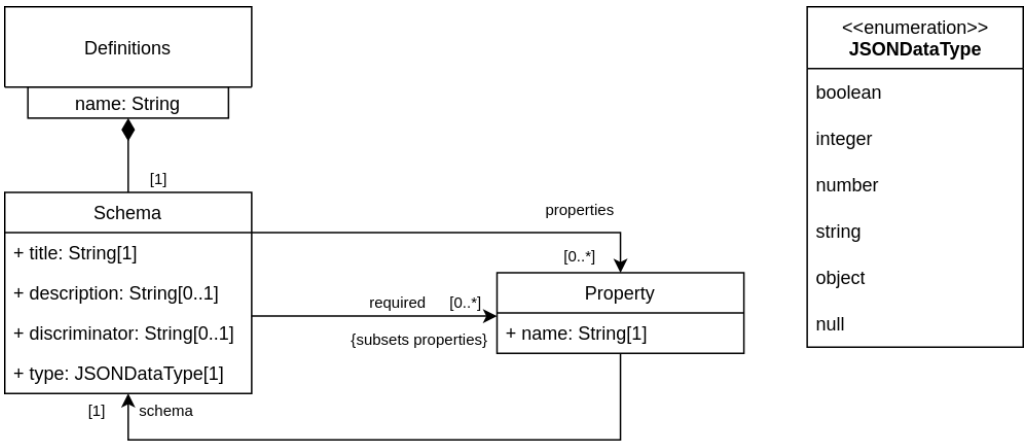
Fonte: Autoria própria.

A Figura 6 apresenta uma visão geral das metaclasses relacionadas à definição de tipos de dados em uma descrição OpenAPI. `Definitions` representa o elemento raiz que agrega as definições dos tipos de dados reutilizáveis na descrição. Por sua vez, a metaclasse `Schema` representa um tipo de dados utilizado pelo serviço. Esta metaclasse é uma extensão à metaclasse análoga provida pelo JSON Schema [62]. Dessa maneira, um elemento `Schema` pode representar um elemento de um dado tipo

definido no JSON Schema (atributo `type` do tipo `JSONDataType`), o formato dos valores aceitos por esse elemento ou uma enumeração de valores aceitos (atributos `format` e `enum`, respectivamente), uma descrição textual (atributo `description`) e um valor padrão (atributo `default`), entre outros. Aos atributos já definidos pelo JSON Schema, a OpenAPI adiciona um conjunto de outros atributos adicionais a `Schema`, como um título para o tipo de dados definido (atributo `title`) e se os valores da estrutura de dados são imutáveis (atributo `readOnly`). `Definitions` possui uma associação qualificada com diferentes elementos `Schema`, mapeando um nome único (`name`) ao elemento `Schema` correspondente. Dessa maneira, um dado elemento `Schema` pode ser referenciado em diferentes pontos de uma descrição OpenAPI por meio da *string* associada a esse elemento em `Definitions`.

Uma instância de `Schema` pode possuir um conjunto de `Property` (referência `properties`). `Property` representa uma propriedade da estrutura de dados definida por um elemento `Schema` do tipo objeto (atributo `type` com valor `#object`). `Property` provê um nome a essa propriedade (atributo `name`) e referencia um elemento `Schema` que indicará o tipo de dado aceito por aquela propriedade (referência `schema`). `Schema` pode, opcionalmente, indicar quais propriedades são requeridas para o objeto ser bem-formado (referência `required`).

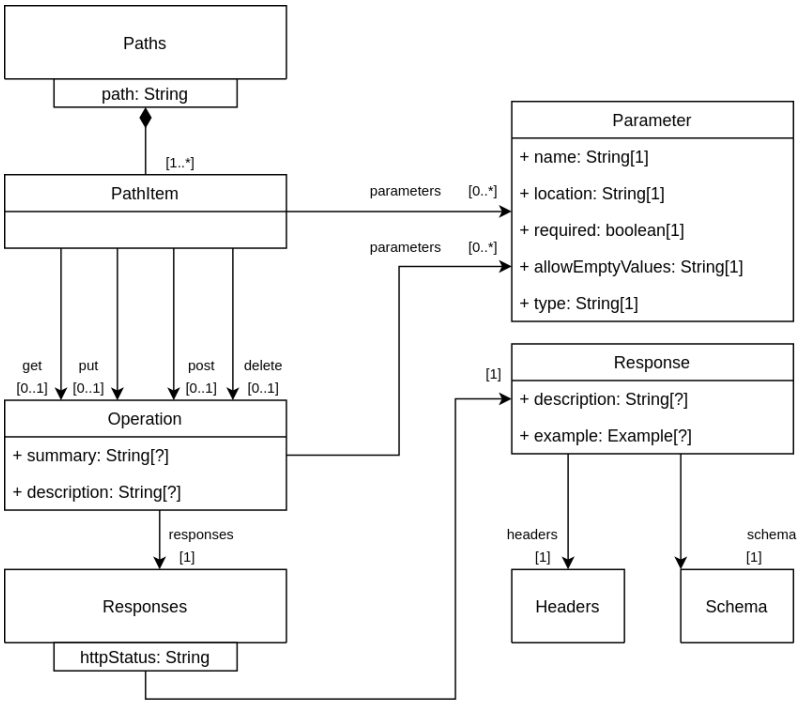
Figura 6 – Visão geral das metaclasses relacionadas à definição de tipos de dados no OpenAPI.



Fonte: Autoria própria.

A Figura 7 apresenta um visão geral das metaclasses relacionados à definição dos caminhos de recursos, operações e respostas no OpenAPI Metamodel. A metaclasse `Paths` representa o elemento que agrega a definição dos recursos do serviço descrito. Por sua vez, a metaclasse `PathItem` representa um conjunto de recursos de um determinado tipo presentes no serviço. Estes recursos são identificados pelo caminho relativo em relação à base do serviço, o qual são representados pelo atributo `path` da associação qualificada entre `Paths` e `PathItem`.

Figura 7 – Visão geral das metaclasses relacionadas à definição dos caminhos de recursos, operações e respostas no OpenAPI.



Fonte: Autoria própria.

`PathItem` permite definir um conjunto de parâmetros para o caminho relativo do recurso, bem como para definir parâmetros que podem ser reutilizados em diferentes operações sobre esse recurso (referência `parameters`). A metaclasse `Parameter` representa um parâmetro nomeado (atributo `name`) utilizável em diferentes locais de uma requisição HTTP, tais como o caminho do recurso, o corpo da requisição ou o cabeçalho (atributo `location`). `Parameter` também permite indicar se o parâmetro é obrigatório (atributo `required`), bem como se esse parâmetro aceita valores vazios (atributo `allowEmptyValues`). Adicionalmente, `Parameter` indica o tipo de dados aceito pelo parâmetro (atributo `type`).

A metaclasse `Operation` representa um operação realizável sobre os recursos definidos por `PathItem`. `Operation` permite associar uma descrição textual da operação (atributo `description`), bem como uma descrição textual curta (atributo `summary`). `Operation` também permite referenciar um conjunto de `Parameter` que podem ser utilizados nessa operação, de maneira a adicionar valores obrigatórios presentes no corpo da requisição HTTP ou em seus cabeçalhos. `PathItem` define o método HTTP associado ao elemento `Operation` associando o elemento `Operation` a uma referência com o mesmo nome do método HTTP (referências `post`, `get`, `put` e `delete`, entre outras).

A metaclasse `Response` representa uma possível resposta HTTP recebida

ao executar uma determinada operação sobre um dado recurso. Cada resposta é identificada pelo código HTTP retornado (atributo `httpStatus` da associação qualificada pertencente à metaclasse `Responses`) e permite associar uma descrição e exemplos (atributos `description` e `examples`, respectivamente). Por fim, `Response` também permite especificar a estrutura de dados retornada no corpo da resposta HTTP (referência `schema`) e associar um conjunto de cabeçalhos HTTP retornados (referência `headers`).

2.3 Modelos, metamodelos e desenvolvimento de *software*

Um modelo pode ser visto como um artefato concreto que contém informações sobre um dado sistema de interesse. Essa característica permite o uso desses modelos em abordagens de desenvolvimento baseadas em modelos. Nessas abordagens, técnicas computacionais são utilizadas para automatizar a geração de (parte de) um *software* a partir de um conjunto de modelos iniciais mais abstratos. Esta seção apresenta alguns fundamentos de modelagem conceitual, bem como apresenta uma visão geral do desenvolvimento de *software* baseado em modelos.

2.3.1 Fundamentos de modelagem conceitual

Um modelo consiste de um conjunto de declarações formais sobre um sistema em estudo criado de forma a prover suporte à análise e/ou projeto desse sistema [18]. Nesse sentido, um modelo representa as principais características estruturais e/ou comportamentais necessárias a um determinado tipo de análise sobre um dado sistema, ao mesmo tempo que abstrai dessa representação os detalhes menos relevantes, facilitando a compreensão sobre o sistema em estudo.

Dois fatores devem ser considerados para a definição dos elementos que devem ser representados em um modelo: o ponto de vista da análise e o nível de abstração do modelo. O ponto de vista da análise define o conjunto de elementos importantes ao entendimento de uma dada característica do sistema. Dessa maneira, diferentes modelos podem apresentar um mesmo sistema segundo diferentes pontos de vistas, segmentando esse sistema conforme diferentes interesses de análise [63]. O nível de abstração de um modelo define quão próximo esse modelo está da solução computacional que esse modelo representa. Portanto, as declarações em um modelo em alto nível de representação são definidas em uma linguagem mais próxima do domínio que o sistema está inserido e mais afastada do domínio de implementação. Dessa maneira, o nível de abstração define, dentro de um dado ponto de vista, quais características de um sistema devem ser representadas e quais características podem ser omitidas.

As declarações contidas em um modelo devem ser representadas em uma linguagem adequada de modo que expressem um significado bem definido. Dessa maneira, ferramentas computacionais poderão ser utilizadas para a análise e validação automática desse modelo [19]. Adicionalmente, métodos para transformação desse modelo em outros artefatos de *software*, tais como código em uma determinada plataforma de implementação e documentação, também poderão ser utilizados [19,20].

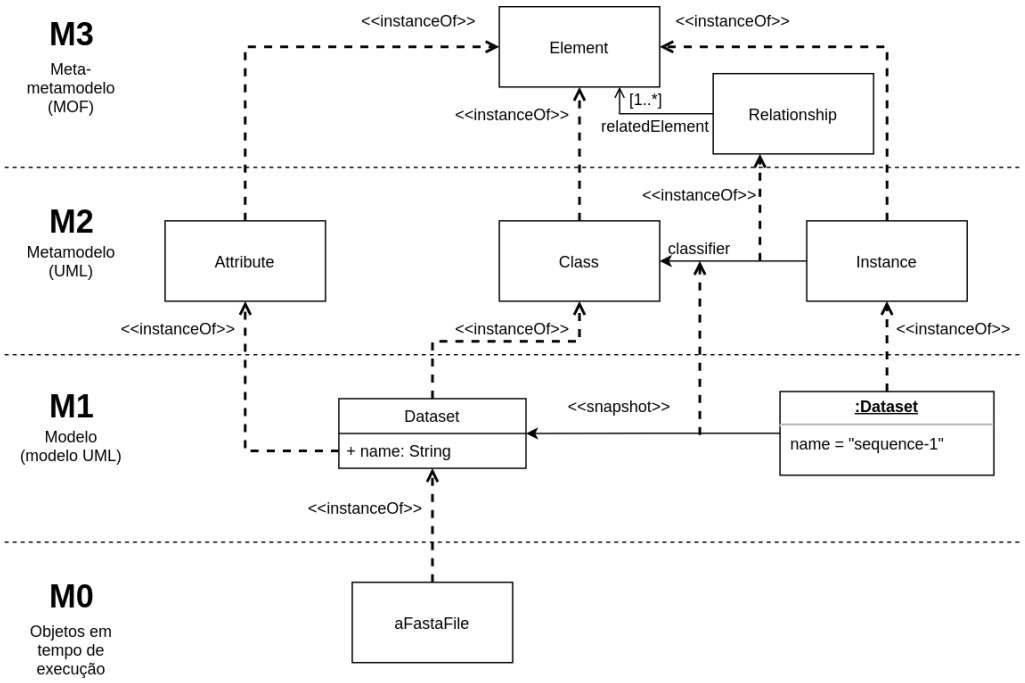
Uma linguagem de modelagem pode ser vista como um construto de três partes: sintaxe abstrata, sintaxe concreta e semântica. A *sintaxe abstrata* de uma linguagem define os elementos estruturais dessa linguagem de maneira independente da representação utilizada para concretizar as declarações em si. Neste sentido, a sintaxe abstrata define um conjunto de elementos passíveis de representação por meio de declarações da linguagem, bem como as relações e combinações possíveis de serem realizadas entre esses elementos, i.e., um vocabulário formal e um conjunto de regras de composição.

Por sua vez, a *sintaxe concreta* de uma linguagem provê uma representação para os elementos definidos pela sintaxe abstrata, i.e., um alfabeto de representação para a linguagem. Diferentes sintaxes concretas podem ser providas para uma mesma sintaxe abstrata. Por fim, a *semântica* de uma linguagem consiste em um mapeamento (externo) dos elementos da sintaxe abstrata aos elementos do domínio em estudo. Este mapeamento semântico é utilizado para definir o significado das declarações presentes em um modelo em relação ao sistema em estudo.

Um modelo pode ser utilizado para descrever formalmente a sintaxe abstrata de uma classe de outros modelos [64]. Por exemplo, um modelo *m1* pode ser utilizado para definir os elementos válidos e representáveis em um modelo *m0*. Nesse sentido, o modelo *m1* é chamado de metamodelo do modelo *m0*. A sintaxe abstrata dos elementos representáveis no modelo *m1* pode também ser definida pelos elementos descritos em um modelo *m2*. Neste caso, *m2* é o metamodelo de *m1* e o meta-metamodelo de *m0*.

O uso de metamodelos formais facilita o desenvolvimento de ferramentas capazes de lidar de maneira uniforme com os diversos modelos criados a partir desses metamodelos [64]. O número de camadas da arquitetura de metamodelagem pode se estender conforme a necessidade do desenvolvedor da linguagem. Porém, em geral essas arquiteturas são desenvolvidas com um número baixo de camadas. Por exemplo, a linguagem UML [65] define uma arquitetura de metamodelagem em quatro camadas: M0, ou camada dos objetos, na qual residem os objetos presentes em um sistema em tempo de execução; M1, ou camada dos modelos, na qual residem os modelos UML contendo informações de um dado domínio de interesse; M2, ou camada do metamodelo, na qual é definida a linguagem UML em si, utilizada para

Figura 8 – Arquitetura em quatro camadas da UML.



Fonte: Autoria própria. Baseado em [67].

a representação de modelos UML; e M3, ou camada do meta-metamodelo, na qual é definida a linguagem *Meta-Object Facility* [66], utilizada como metamodelo da linguagem UML.

A Figura 8 apresenta um exemplo contendo alguns elementos da arquitetura de metamodelagem em quatro camadas da UML. Todos os elementos existentes em uma camada inferior são instâncias dos elementos definidas na camada imediatamente superior. Por exemplo, os elementos `Attribute`, `Class` e `Instance` na camada M2 (metamodelo UML) são instâncias de `Element` definido na camada M3 (meta-metamodelo MOF), enquanto a referência `classifier` existente entre `Instance` e `Class` é uma instância do elemento `Relationship` existente em M3.

Embora linguagens de modelagem de propósito geral, como UML, estejam disponíveis, uma linguagem específica de domínio (*Domain Specific Language* ou DSL) pode ser definida para permitir a descrição formal dos elementos de um domínio em particular. Neste sentido, uma DSL adequada permite que especialistas de um domínio descrevam esse domínio por meio de um modelo utilizando um vocabulário familiar. Tecnologias orientadas a modelos podem, então, ser utilizadas para obter uma implementação completa ou parcial da solução para o problema definido. Dessa maneira, a definição e uso de DSLs adequadas e tecnologias de suporte permitem reduzir a distância semântica entre a descrição do problema (domínio da linguagem) e as soluções computacionais desejadas [68].

2.3.2 Desenvolvimento orientado a modelos

O Desenvolvimento Orientado a Modelos (*Model-Driven Development* ou MDD) [18, 19, 64] é uma abordagem de desenvolvimento de sistemas computacionais que visa o refinamento e a transformação de modelos em alto nível de abstração até a obtenção final de um sistema computacional em uma linguagem de programação. A premissa básica do desenvolvimento orientado a modelos é que (parte de) uma dada aplicação seja gerada a partir de um conjunto de modelos iniciais. Nesse processo, busca-se aproveitar ao máximo o suporte à automação do processo de desenvolvimento por meio do uso de padrões de modelagem e tecnologias para a transformação de modelos em código.

A principal vantagem do Desenvolvimento Orientado a Modelos reside, em um primeiro momento, na criação de modelos independentes de uma dada tecnologia de implementação e mais próximos do domínio de aplicação. Essa característica facilita a compreensão do problema e a especificação da solução, bem como a manutenção desse *software* após sua implementação. Adicionalmente, modelos são menos suscetíveis a mudanças tecnológicas que a implementação desses modelos em uma linguagem de programação. Dessa maneira, a existência de suporte adequado pode permitir o reúso de um mesmo conjunto de modelos no desenvolvimento de aplicações para diferentes tecnologias e/ou a atualização facilitada do código da aplicação após a atualização de uma dada plataforma de execução.

As declarações presentes em um modelo podem ser expressas sob diferentes pontos de vista e representadas utilizando-se um vocabulário próprio de um domínio (linguagens específicas de domínio). Após a definição dos modelos iniciais, o suporte a transformações automáticas pode ser utilizado de modo a obter uma implementação para a aplicação (ou para uma parte dela). Assim, a existência de suporte ao desenvolvimento orientado a modelos em um dado domínio de aplicação pode auxiliar o especialista deste domínio na obtenção de soluções de *software* adequadas às suas necessidades.

2.3.2.1 Arquitetura orientada a modelos

Diversas metodologias e *frameworks* apoiam o desenvolvimento orientado a modelos. Por exemplo, a Arquitetura Orientada a Modelos (*Model-Driven Architecture* ou simplesmente MDA) [69–72] é uma abordagem de desenvolvimento orientado a modelos proposta pela OMG para o desenvolvimento de sistemas computacionais. De acordo com esta arquitetura, modelos são definidos a partir de três pontos de vista [69]: *modelos independentes de computação*, os quais são criados para capturar conceitos relacionados aos requisitos do sistema e seu ambiente de execução, sem apresentar detalhes da estrutura e comportamento do sistema; *modelos independentes*

de plataforma, os quais representam a estrutura e o comportamento do sistema de maneira abstrata, escondendo detalhes necessários para sua execução em uma plataforma; por fim, *modelos específicos de plataforma*, os quais acrescentam detalhes específicos de uma dada plataforma aos modelos independentes de plataforma.

A trajetória de desenvolvimento MDA inicia-se com a captura do ambiente e dos requisitos do sistema em modelos independentes de computação. Em seguida, os esforços são concentrados na definição de detalhes sobre a estrutura e as operações do sistema. Esses detalhes são formalizados em modelos independentes de plataforma. Por fim, transformações automáticas devem ser definidas e utilizadas para transformar esses modelos em implementações adequadas a diferentes plataformas. Dessa maneira, podemos reduzir os custos e os esforços necessários para a criação de diferentes implementações do sistema.

2.3.2.2 *Frameworks* e ferramentas de suporte

Eclipse Modeling Project (EMP) [73] é uma iniciativa da *Eclipse Foundation* que visa agregar e fomentar projetos para o suporte ao desenvolvimento orientado a modelos no ambiente de desenvolvimento *Eclipse* [74]. Os projetos promovidos pelo EMP suportam diferentes aspectos do desenvolvimento orientado a modelos, tais como o desenvolvimento de novos metamodelos, o desenvolvimento de sintaxes para a representação de modelos de forma textual ou gráfica, bem como o desenvolvimento de transformações automáticas de modelos.

O projeto *Eclipse Modeling Framework* (EMF) [75] é um framework para (meta) modelagem e desenvolvimento de ferramentas baseadas em modelos. Este *framework* é utilizado como base de integração entre os demais projetos do EMP. EMF provê um metamodelo reflexivo, chamado *Ecore*, desenvolvido com base na especificação do *Essential MOF* (EMOF) [76]. Modelos *Ecore* podem ser utilizados para capturar uma conceitualização sobre um dado domínio de forma semelhante a diagramas de classes UML.

O *framework* EMF provê transformações dos elementos de um modelo *Ecore* para um conjunto de classes Java equivalentes, bem como facilidades para a manipulação e serialização de instâncias Java criadas com base nesse modelo conceitual. Adicionalmente, o metamodelo *Ecore* também pode ser utilizado como um metametamodelo para a criação de metamodelos dedicados às necessidades de um usuário. As características reflexivas do *Ecore* permitem que esse metamodelo seja manipulado de forma transparente por outros *frameworks* baseados no EMF. Dessa maneira, o usuário pode utilizar o suporte provido por outros *frameworks* em diferentes atividades, tais como na representação e na transformação dos modelos criados.

2.3.3 Desenvolvimento orientado a modelos de serviços *web*

Diferentes abordagens orientadas a modelos têm sido propostas para o desenvolvimento de serviços e aplicações *web*. Pahl [77] propõe um *framework* suportado por ontologias para a representação, o raciocínio e a transformação de modelos durante o desenvolvimento de serviços *web*. Neste sentido, este framework envolve a definição de um conjunto de conceitos e relacionamentos de interesse para os diferentes pontos de vista da arquitetura de suporte ao desenvolvimento orientado a modelos de serviços *web*. Esses conceitos e relacionamentos definem um conjunto de metamodelos abstratos. O usuário do *framework* concretiza cada metamodelo abstrato por meio da seleção de ontologias adequadas. Desta maneira, o trabalho propõe o uso de um suporte lógico-semântico formal na definição dos modelos e transformações de modelos necessárias ao desenvolvimento de um serviço *web*.

Schreier [78] apresenta um metamodelo baseado em *Ecore* para a modelagem estrutural e comportamental de serviços RESTful. Este metamodelo foi desenvolvido a partir de uma terminologia de domínio que facilita o desenvolvimento dessas aplicações. Duas fases de modelagem de um serviço *web* RESTful são propostas: a modelagem estrutural dos recursos RESTful e a modelagem comportamental destes mesmos recursos. Inicialmente, o desenvolvedor descreve os recursos segundo os tipos propostos pela terminologia definida, bem como atributos, relações, interfaces e representações associados a esses recursos. Em seguida, o desenvolvedor descreve o comportamento interno de cada recurso por meio de máquinas de estado. Por fim, o metamodelo proposto também permite modelar as representações dos recursos, utilizadas nas respostas às requisições recebidas.

Haupt *et al.* [79] propõem uma arquitetura para o projeto e o desenvolvimento orientado a modelos de serviços RESTful. A arquitetura proposta visa reforçar a obediência às restrições RESTful, enquanto define o uso e a transformação sucessiva de diferentes modelos até a obtenção de uma implementação RESTful de um serviço *web*. Adicionalmente, os autores propõem um modelo de papéis para os desenvolvedores desses serviços. Este modelo de papéis identifica três diferentes desenvolvedores de modo a haver uma divisão de responsabilidades durante o desenvolvimento: o especialista de domínio, o especialista REST e o especialista de aplicação. Cada papel é associado ao desenvolvimento de diferentes modelos da arquitetura proposta. Finalmente, uma extensão ao ambiente de desenvolvimento *Eclipse* foi implementada para prover suporte a esta metodologia. Este trabalho foi estendido posteriormente de modo a introduzir um metamodelo centrado em conversas RESTful para a modelagem de interações cliente-servidor utilizado durante o desenvolvimento de serviços *web* [56].

Sistemas integrados de análise são utilizados, estendidos e mantidos princi-

palmente por cientistas, engenheiros e outros especialistas de domínio com pouco conhecimento de engenharia de software. Dessa maneira, métodos acessíveis a especialistas de domínio devem ser providos para suporte a estas atividades. Um dos desafios para a extensão desses sistemas reside no desenvolvimentos de serviços de acesso a dados persistidos em sistemas de armazenamento heterogêneos e/ou distribuídos. Neste sentido, a necessidade de lidar com diferentes aspectos técnicos, tais como a heterogeneidade das representações dos conceitos armazenados e das interfaces de acesso aos sistemas de armazenamento, representam desafios ao desenvolvimento rápido destes serviços.

Frente a esses desafios, Bender *et al.* [21] propõem um *framework* de engenharia orientada a modelos para o desenvolvimento de serviços *web*. Este *framework* visa prover o acesso a dados heterogêneos para sistemas integrados de análise e simulação em ciências e engenharia. Neste sentido, um desenvolvedor pode modelar conceitualmente os dados a serem armazenados por meio do *framework*. A partir do modelo de dados definido, serviços RESTful são gerados automaticamente para prover acesso aos dados armazenados. O compartilhamento de dados entre clientes e o serviço de acesso é realizado por meio de representações em XML definidas segundo o metamodelo disponibilizado pelo *framework*.

O *framework* disponibiliza diferentes tipos de sistemas de armazenamento para serem utilizados na camada de persistência dos dados, tais como bancos de dados relacionais, bancos de dados baseados em documentos e bancos de dados baseados em grafos. Clientes podem recuperar e persistir objetos de dados representados dessa maneira por meio das interfaces de acesso automaticamente geradas. Assim, este *framework* busca resolver o problema do compartilhamento de dados heterogêneos sem o uso de abordagens semânticas, mas pelo uso de um formato padrão de compartilhamento a partir do qual clientes e serviços devem mapear e extrair as entidades armazenadas. Por fim, o *framework* permite a obtenção de um serviço de acesso a dados como um arquivo WAR [80], bem como a criação automática de um servidor de aplicação local contendo o serviço de acesso aos dados modelados.

Ed-douibi *et al.* [24] propõem uma abordagem para a geração semiautomática de serviços *web* RESTful a partir de metamodelos baseados no *framework* EMF. Uma API JavaScript para o acesso aos serviços desenvolvidos também é gerada automaticamente de modo a auxiliar na interação de clientes com estes serviços. Os serviços criados segundo essa abordagem proveem acesso a operações CRUD sobre os recursos/modelos instanciados remotamente utilizando os princípios REST. O uso do EMF como base da proposta permite utilizar o mecanismo de validação automática de modelos providos por este *framework*, garantindo a consistência dos dados armazenados. Finalmente, uma extensão ao ambiente de desenvolvimento *Eclipse* é provida para o suporte à abordagem proposta.

Sferruzza [81] propõe estender a linguagem OpenAPI de modo a permitir a inclusão de detalhes de implementação dos *endpoints* e suas operações, bem como a utilização de um mecanismo de geração de código para obter o código do serviço. Neste sentido, extensões a essa linguagem permitiriam incluir identificadores de componentes a serem utilizados para cada *endpoint* e operação. Cada componente é definido de maneira atômica, implementado separadamente em uma dada linguagem de programação. Um mecanismo de transformação e geração de código é, então, utilizado para combinar o código dos componentes segundo a definição contida na descrição OpenAPI, obtendo, ao final, o código fonte do serviço. Dessa maneira, o trabalho busca promover a implementação *top-down* de serviços *web*, tornando a descrição OpenAPI do serviço o principal artefato dessa implementação.

Mohseni *et al.* [82] propõe uma arquitetura MDA para a modelagem de serviços *web* semânticos a partir de UML. Neste sentido, o trabalho provê um perfil UML para estender a linguagem e permitir a modelagem desses serviços. Aspectos estruturais dos serviço em desenvolvimento são modelados por meio de diagramas de classes, enquanto aspectos comportamentais são expressos por meio de diagramas de atividade e diagramas de sequência, bem como de expressões na linguagem *Object Constraint Language* (OCL) [83]. Um conjunto de transformações são, então, utilizadas para mapear os elementos do modelo UML e obter uma descrição de serviço representada por meio de *Web Service Modeling Language* (WSML) [84]. WSML foi criada para prover sintaxe e semântica para a *Web Service Modeling Ontology* (WSMO) [85, 86], uma ontologia para descrever semanticamente um serviço *web* quanto a seus aspectos estruturais e comportamentais, facilitando a descoberta desses serviços. Dessa maneira, o trabalho busca produzir descrições formais de um serviço *web* a ser desenvolvido a partir de modelos em linguagens de modelagem de propósito geral.

3 Desenvolvimento baseado em modelos para serviços de análise em bioinformática

A disponibilização de uma ferramenta de análise por meio de um serviço *web* promove o maior compartilhamento e o uso dessa ferramenta. No entanto, o desenvolvimento de um serviço *web* demanda um esforço considerável, bem como o domínio de um conjunto de tecnologias e padrões normalmente desconhecidos pelo especialista em bioinformática. Esse esforço pode ser reduzido se metodologias de desenvolvimento orientado a modelos forem utilizadas de modo a obter os serviços de análise desejados por meio da criação e manipulação de modelos em alto nível de abstração. Porém, a definição de um processo de desenvolvimento baseado em modelos demanda um esforço prévio na conceitualização do domínio ao qual esse processo será aplicado. Em vista disso, este capítulo apresenta uma conceitualização do domínio de adaptação de ferramentas de análise por meio de serviços *web* e os requisitos estruturais e comportamentais dos artefatos necessários para essa adaptação. Por fim, este capítulo apresenta uma visão geral do processo de desenvolvimento baseado em modelos para serviços de análise em bioinformática proposto por este trabalho.

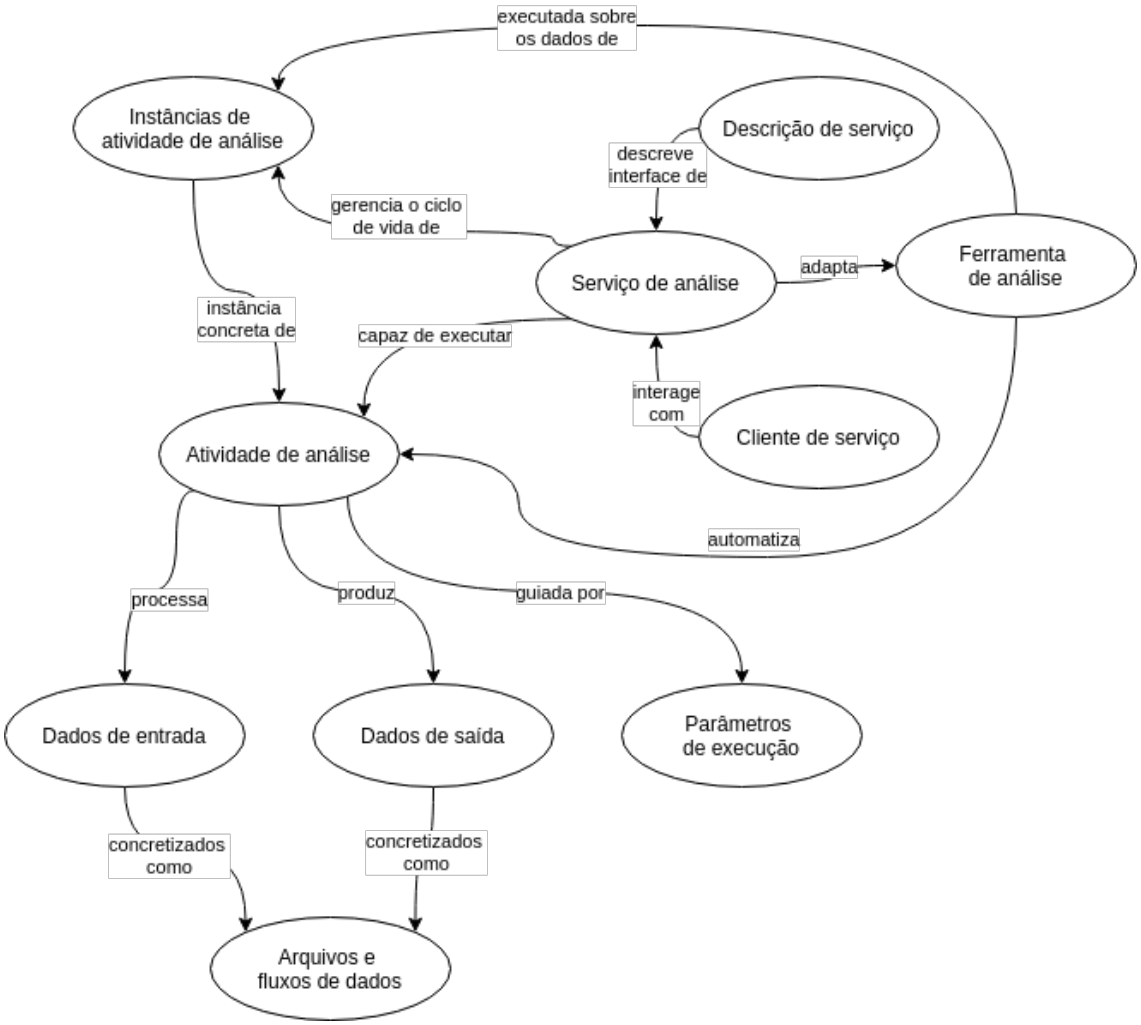
O restante do capítulo está estruturado da seguinte maneira: a Seção 3.1 apresenta um modelo conceitual da adaptação de uma atividade de análise por meio de um serviço RESTful; a Seção 3.2 apresenta o ciclo de vida de uma instância de atividade de análise executada por meio de um serviço RESTful; a Seção 3.3 apresenta um conjunto de requisitos estruturais e comportamentais para serviços de análise, bem como para clientes e descrições desses serviços; a Seção 3.4 apresenta um processo de desenvolvimento baseado em modelos para serviços de análise em bioinformática, seus clientes e descrições; por fim, a Seção 3.5 apresenta algumas considerações finais.

3.1 Modelo conceitual da adaptação de uma atividade de análise por meio de um serviço *web*

Este trabalho visa facilitar o desenvolvimento e uso de serviços de análise em bioinformática por meio da adaptação de ferramentas de análise existentes. Nesse sentido, inicialmente definimos uma conceitualização sobre as principais entidades relacionadas à essa adaptação. A Figura 9 ilustra o modelo conceitual da adaptação

de uma ferramenta existente por meio de um serviço de análise.

Figura 9 – Modelo conceitual da adaptação de uma ferramenta de análise por meio de um serviço *web*.



Fonte: Autoria própria. Uma elipse representa um conceito. Uma seta entre duas elipses representa um relacionamento entre os conceitos associados.

Atividade de análise representa um tipo de procedimento realizado durante uma análise de bioinformática com o objetivo de manipular dados biológicos de uma determinada classe e obter novos dados de interesse. Adicionalmente, uma atividade de análise pode ser referenciada por um dado nome como, por exemplo, *alinhamento de amostras de RNA-Seq a um genoma de referência*. Cada execução concreta de uma atividade de análise é denominada uma *instância* dessa atividade de análise, e diferentes instâncias de uma mesma atividade de análise podem ser executadas sobre dados biológicos diferentes e utilizando diferentes parâmetros de análise.

Uma atividade de análise processa um ou mais *conjuntos de dados de entrada* de modo a produzir um ou mais *conjuntos de dados de saída*. Um *conjunto de dados de entrada* define uma classe de dados com sintaxe e semântica determinados que a

atividade de análise recebe para processamento, por exemplo, *amostras de RNA-Seq não-normalizadas representadas no formato FASTA*. Por sua vez, um *conjunto de dados de saída* define uma classe de dados com sintaxe e semântica determinados que a atividade de análise produz a partir do processamento dos conjuntos de dados de saída. Um exemplo de conjunto de dados de saída para a atividade de análise *alinhamento de amostras de RNA-Seq a um genoma de referência* seria *amostras de RNA-Seq alinhadas representadas no formato SAM*. Os conjuntos de dados de entrada e conjuntos de dados de saída de uma atividade de análise são concretizados como *arquivos ou outros fluxos de dados* recebidos ou produzidos pela execução da instância de atividade de análise.

O processamento dos conjuntos de dados de entrada é guiado por um conjunto de *parâmetros de execução*. Um *parâmetro de execução* representa um conjunto de valores de semântica definida que direcionam o processamento dos conjuntos de dados de entrada e, por consequência, a produção dos conjuntos de dados de saída. Por exemplo, um parâmetro de execução booleano para a atividade de alinhamento de amostras de RNA-Seq pode indicar se, para cada sequência presente na amostra não-normalizada, partes desta sequência podem ser alinhadas de maneira não contígua no genoma de referência. Parâmetros de execução de uma atividade de análise recebem valores concretos e determinados anteriormente à execução de uma instância dessa atividade de análise.

A execução da atividade de análise pode ser apoiada por uma *ferramenta de análise*. Uma *ferramenta de análise* consiste em um programa executável que recebe valores concretos para conjuntos de dados de entrada e parâmetros de execução, aplica automaticamente os passos definidos pela atividade de análise sobre os dados iniciais e produz conjuntos de dados de saída. Dessa maneira, uma ferramenta de análise automatiza a execução de uma atividade de análise sobre os dados de novas instâncias dessa atividade. Mais de uma ferramenta de análise pode existir para apoiar a execução de uma atividade de análise. Por exemplo, as ferramentas STAR [87] e Bowtie [88] podem ser utilizadas para a atividade de alinhamento de amostras de RNA-Seq a um genoma de referência.

Um serviço de análise adaptador, ou, simplesmente, *serviço de análise*, representa um serviço RESTful que adapta uma ferramenta de análise existente de modo a executar uma dada atividade de análise em resposta a um conjunto de interações com um usuário. Um serviço de análise gerencia o ciclo de vida das instâncias da atividade de análise que estão sob seu controle, provendo acesso a um conjunto de operações para a manipulação dessa instância de atividade para seus usuários. Nesse sentido, o serviço de análise armazena os arquivos submetidos para conjuntos de dados de entrada e os valores submetidos para os parâmetros de execução de cada instância de atividade de análise. Após a execução da instância da atividade de

análise, o serviço de análise provê acesso aos arquivos gerados para os conjuntos de dados de saída. A execução da instância de atividade de análise propriamente dita pode ser realizada por meio de uma ferramenta de análise existente, a qual o serviço de análise adapta de modo a expor apenas uma interface *web* padronizada.

Um *cliente de serviço* consiste em um programa executável pelo qual um usuário pode interagir com um serviço de análise de maneira simplificada e programática. Dessa maneira, um cliente de serviço torna transparente as diferentes interações necessárias para a execução remota da atividade de análise por meio do serviço.

Por fim, uma *descrição de serviço* consiste de um modelo formal que descreve a interface RESTful do serviço de análise. A descrição de um serviço permite que desenvolvedores de *software* compreendam os *endpoints* e operações disponibilizados por esse serviço e sejam capazes de desenvolver novos clientes para utilizá-lo, se necessário.

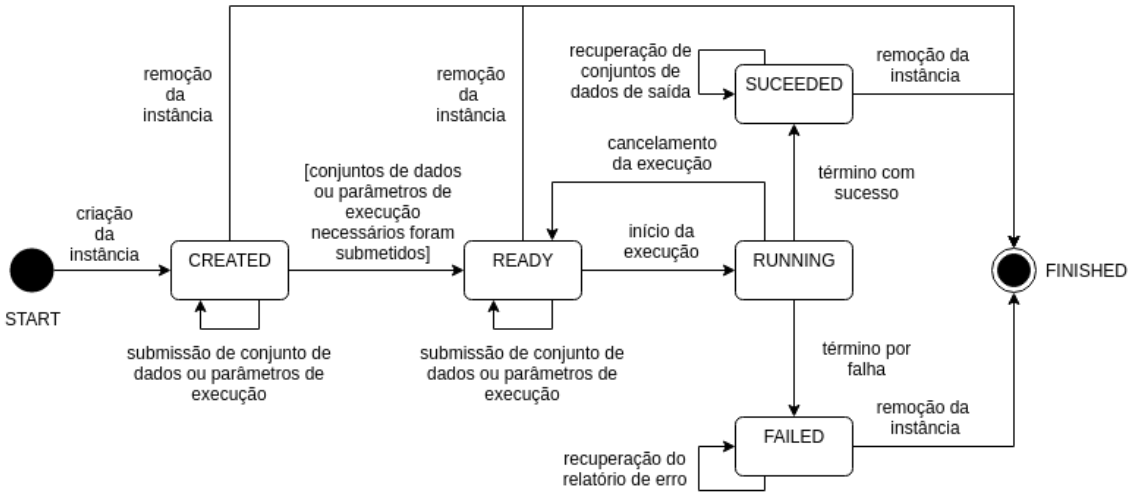
3.2 Ciclo de vida de uma instância de atividade de análise

Uma atividade de análise pode ser comparada a um programa computacional, i.e., ambos consistem em uma descrição de um conjunto de passos que devem ser executados de modo a obter um resultado desejado a partir de um conjunto de dados de entrada com valor parametrizado, porém ainda não determinado. Neste sentido, um programa computacional pode ser instanciado diversas vezes na forma de diferentes processos. Cada um desses processos recebe um conjunto de argumentos de execução independente e produz resultados próprios a partir desses argumentos. De maneira semelhante, a atividade de análise também pode ser instanciada diversas vezes. Cada instância de uma atividade de análise recebe um conjunto de dados de entrada e de parâmetros de execução, produzindo um conjunto de dados de saída independentemente da execução de outras instâncias.

Assim como um processo assume diferentes estados durante sua execução, uma instância de atividade de análise assume diferentes estados ao longo de sua execução mediante a ocorrência de um conjunto de eventos. Assim, o ciclo de vida de uma instância de atividade de análise é formado pelo conjunto dos estados nos quais uma instância de atividade de análise percorre desde sua criação até seu término, bem como pelo conjunto de eventos que disparam as transições entre esses estados. A Figura 10 apresenta uma visão geral do ciclo de vida de uma atividade de análise por meio de um diagrama de estados UML.

Inicialmente, a instância da atividade de análise não existe. Tal característica é representada pelo estado `START`. Eventos subsequentes irão criar, inicializar e executar a instância da atividade de análise, recuperando um ou mais conjuntos de

Figura 10 – Ciclo de vida de uma atividade de análise em bioinformática.



Fonte: Autoria própria.

dados de saída ao final.

O primeiro evento no ciclo de vida apresentado é a criação de uma nova instância da atividade de análise. Essa instância toma a forma de um recurso concreto e endereçável. Dessa maneira, é possível associar conjuntos de dados de entrada, parâmetros de execução e outras informações a essa instância. A criação desse contexto de execução leva a instância de atividade de análise do estado `START` ao estado `CREATED`.

Durante o estado `CREATED` a instância de atividade de análise não está pronta para ser executada. O usuário precisa ainda submeter conjuntos de dados de entrada e parâmetros necessários à execução. Após o usuário informar todos os parâmetros de execução sem valor padrão e a quantidade mínima de arquivos para os conjuntos de dados de entrada, a instância de atividade de análise transita para o estado `READY`. Nesse sentido, a submissão de eventuais parâmetros de execução com valores padrões definidos é opcional, não sendo necessária para disparar essa transição.

Uma instância de atividade de análise no estado `READY` pode ser processada imediatamente. Parâmetros de execução e conjuntos de dados de entrada ainda podem ser modificados pelo usuário. Desse modo, caso o usuário ainda necessite enviar arquivos adicionais para os conjuntos de dados de entrada da atividade de análise, ele ainda é capaz de fazê-lo nesse estado. A transição para o estado `RUNNING` dispara o início do processamento da instância de atividade de análise.

Durante a fase de processamento, a instância de atividade de análise se mantém no estado `RUNNING`. A ferramenta computacional que provê suporte à execução da atividade de análise é invocada durante a transição para este estado. Essa ferramenta recebe os parâmetros e os conjuntos de dados de entrada informados

e executa a atividade de análise até seu término com sucesso ou até que uma falha de processamento ocorra.

Durante a fase de processamento, o usuário não interage diretamente com a instância da atividade de análise, pois neste estado não é possível alterar parâmetros ou dados de entrada e os resultados da execução ainda não estão disponíveis. Assim, o usuário aguarda o término do processamento verificando periodicamente o estado dessa execução por meio de uma referência ou sendo notificado do término da execução de maneira assíncrona. Essa referência também pode permitir que o usuário cancele a execução da instância de atividade de análise, retornando essa instância ao estado `READY` e permitindo o reinício da fase de processamento posteriormente. Porém, esse cancelamento pode não ser sempre possível ou bem sucedido, sendo dependente de suporte à restauração do estado anterior à execução e da ausência de efeitos residuais.

O término da fase de processamento resulta na transição para o estado `SUCCEEDED`, em caso de término com sucesso, ou para o estado `FAILED`, em caso de término com falha. No estado `SUCCEEDED`, o usuário pode recuperar os conjuntos de dados de saída criados durante a execução, enquanto que no estado `FAILED` o usuário pode recuperar um relatório de erro contendo a causa provável da falha.

Em qualquer estado, o usuário pode remover a instância da atividade de análise e liberar recursos associados a ela. A remoção da instância de atividade de análise resulta na transição para o estado `FINISHED`. Após a transição para este estado, o usuário não pode interagir com a instância novamente. Adicionalmente, a transição para o estado `FINISHED` também pode ser provocada pelo sistema hospedeiro, no qual a instância da atividade de análise é processada, quando mecanismos de coleta de lixo são (automaticamente) executados.

O Apêndice A apresenta informações adicionais sobre os estados de uma atividade de análise e sobre as transições entre esses estados.

3.3 Requisitos estruturais e comportamentais

Um conjunto de requisitos estruturais e comportamentais foi definido para o desenvolvimento dos serviços de análise, clientes desses serviços e suas descrições formais segundo a metodologia proposta nesse trabalho. Estes requisitos definem restrições à concretização dos artefatos a serem produzidos de modo a guiar a trajetória de desenvolvimento proposta ao final deste capítulo.

De modo a obter esses requisitos, consideramos o ciclo de vida de uma atividade de análise em bioinformática e estudamos serviços de análise disponibilizados pelo *National Center for Biotechnology Information* (NCBI) [89] e pelo *European*

Bioinformatics Institute (EBI) [90], bem como o repositório *Gene Expression Analysis Services* (GEAS) [7]. Uma visão geral dos serviços presentes nesses repositórios será apresentada no capítulo 9.

3.3.1 Requisitos estruturais e comportamentais de um serviço de análise

A interface *web* de um serviço de análise deve permitir que clientes desse serviço o utilizem para a execução de instâncias da atividade de análise suportada pelo serviço. Dessa maneira, o desenvolvimento de um serviço de análise deve considerar um conjunto de requisitos estruturais e comportamentais das instâncias dessas atividades. De modo a garantir a conveniência e a flexibilidade dos serviços implementados a partir de nossa metodologia, o seguinte conjunto de requisitos para os serviços de análise (RS) foi definido:

RS-1 **Prover uma interface RESTful madura.** O estilo arquitetural REST é um padrão *de facto* para o desenvolvimento de novos serviços *Web*. Dessa maneira, o serviço de análise deve prover uma interface RESTful madura segundo as melhores práticas desse estilo arquitetural;

RS-2 **Permitir a submissão de dados de entrada e de parâmetros de execução de forma independente.** Uma atividade de análise pode possuir múltiplos conjuntos de dados de entrada. Dado que o tamanho de um conjunto de dados de entrada pode variar consideravelmente para cada atividade de análise (ou mesmo entre execuções de diferentes instâncias de uma mesma atividade de análise), a interface RESTful deve prover suporte à submissão independente de cada conjunto de dados. Dessa maneira, o serviço de análise implementado limita a ocorrência de *timeouts* do servidor HTTP e/ou a necessidade de reenvio de uma grande quantidade de dados após uma interrupção da transferência por quedas na rede. Adicionalmente, uma vez que parâmetros de execução podem possuir valores padrões que um usuário do serviço de análise pode decidir manter, a submissão independente de valores para cada parâmetro de execução também deve ser permitida;

RS-3 **Prover suporte à interação assíncrona durante a execução de uma instância de uma atividade de análise.** O tempo de execução de uma instância de uma atividade de análise pode variar consideravelmente de acordo com o objetivo da atividade de análise e/ou os conjuntos de dados de entrada utilizados. Por essa razão, não é adequado induzir um usuário a bloquear-se enquanto espera pelo término de uma execução. O serviço de análise deve permitir interação assíncrona de um usuário com o serviço durante o período de execução de uma instância de atividade de análise, evitando causar um

bloqueio de processamento do lado do usuário. Assim, o usuário deve ser capaz de submeter uma instância de uma atividade de análise para execução e receber uma confirmação dessa submissão de maneira imediata. Em seguida, o usuário deve esperar pelo término da execução da instância submetida, verificando o estado da execução da mesma de maneira periódica, ou ser notificado ao término da execução da atividade de análise de forma assíncrona, de modo a poder realizar outras ações durante esse período;

RS-4 **Prover suporte à indicação de falha.** A execução de uma instância de uma atividade de análise pode terminar em um estado de sucesso ou de falha. O serviço de análise deve prover mecanismos para que os usuários desses serviços sejam capazes de diferenciar facilmente ambos estados de término;

RS-5 **Permitir a recuperação de conjuntos de dados e parâmetros de execução de forma independente.** A execução de uma instância de uma atividade de análise pode gerar diferentes conjuntos de dados de saída. De maneira similar à submissão dos conjuntos de dados de entrada, o serviço de análise deve permitir a recuperação independente de cada conjunto de dados de saída produzido pela execução da atividade de análise;

RS-6 **Prover suporte à remoção de instâncias de uma atividade de análise existentes e seus conjuntos de dados.** O serviço de análise deve prover suporte à remoção de instâncias de atividades de análise existentes e à liberação dos recursos do usuário, armazenados pelo serviço.

3.3.2 Requisitos estruturais e comportamentais para um cliente de um serviço de análise

Parte dos requisitos estruturais e comportamentais de um cliente de um dado serviço de análise é estreitamente dependente da interface e do modelo de interação definidos por esse serviço. Ainda assim, requisitos adicionais podem ser definidos sobre esses clientes. Nesse sentido, os requisitos estruturais e comportamentais abaixo foram definidos para os clientes de serviços de análise desenvolvidos por meio de nossa metodologia:

RC-1 **Prover suporte à interação com um serviço de análise específico.** Embora um mesmo modelo de referência possa ser utilizado para o desenvolvimento de um número de serviços de análise diferentes, especificidades da atividade de análise suportada por cada serviço levam a diferenças na concretização desse modelo de interação. Por exemplo, atividades de análise diferentes podem utilizar ou produzir conjuntos de dados diferentes, bem como ser guiadas por

diferentes parâmetros de execução. Essas diferenças resultam em *endpoints* específicos para cada serviço de análise, assim como no uso de padrões específicos para o tráfego de dados entre um cliente e um serviço. Dessa maneira, a implementação de clientes para serviços de análise em nosso trabalho deve ser realizada de maneira a obter clientes limitados à interação com um serviço específico;

RC-2 **Prover uma interface de usuário específica para uma dada atividade de análise.** Durante ou previamente à execução de um serviço de análise, o cliente deste serviço deve ser capaz de informar ao usuário os parâmetros de execução e os conjuntos de dados a serem utilizados para a atividade de análise. Neste sentido, a interface de usuário do cliente de serviço de análise produzido deve refletir as especificidades desse serviço e da atividade de análise que ele suporta;

RC-3 **Abstrair a interação com o serviço de análise durante a execução da instância de atividade de análise.** Durante a execução do cliente do serviço de análise, as interações entre cliente e serviço necessárias para que essa execução ocorra devem ser abstraídas do usuário do cliente. Dessa maneira, o cliente desenvolvido deve ser capaz de comunicar-se com o serviço de maneira transparente, enviando todos os conjuntos de dados de entrada e parâmetros de execução indicados pelo seu usuário e recuperando os conjuntos de dados de saída automaticamente e sem a necessidade de tomada de ação do usuário durante essa execução;

RC-4 **Ser utilizável em conjunto com outras ferramentas de análise.** O cliente deve ser produzido de maneira a poder ser utilizado em conjunto com outras ferramentas de análise ou integrado em um ambiente de análise. De modo a permitir que o cliente gerado seja integrável com outras ferramentas durante uma análise em bioinformática, decidimos prover clientes executáveis em linha de comando e clientes executáveis por meio de um ambiente integrado de análise. Decidimos por prover suporte ao ambiente Galaxy como representante desse segundo caso, uma vez que trata-se de um ambiente conhecido e bastante utilizado para análises em bioinformática.

3.3.3 Requisitos estruturais para uma descrição de um serviço de análise

Assim como ocorre com os clientes de um serviço de análise, alguns aspectos estruturais de uma descrição de um serviço são dependentes da interface do serviço descrito. Porém, podemos definir um conjunto de requisitos que nossa metodologia deve atender para a criação de descrições de um serviço de análise, de modo a restringir o escopo de nosso trabalho.

RD-1 **Descrição de todos os *endpoints* e operações necessários à execução da atividade de análise e recuperação dos resultados.** A descrição produzida para um serviço de análise deve apresentar todos os *endpoints* e operações necessários à execução da atividade de análise por meio do serviço. Neste sentido, a descrição deve prover o endereço de cada *endpoint* a ser utilizado, as operações (métodos HTTP) aceitos por esse *endpoint*, bem como a representação dos dados esperados e as possíveis respostas que podem ser recebidas. *Endpoints* e operações adicionais, eventualmente providas pelo serviço, não precisam estar representadas nas descrições de serviço desenvolvidas por meio de nossa metodologia;

RD-2 **Representação por meio de uma linguagem padrão de descrição de serviços.** A descrição produzida para um serviço de análise deve ser representada por meio de uma linguagem padrão de descrição de serviços. As linguagens formais WSDL e OpenAPI devem ser suportadas, uma vez que ambas linguagens representam padrões *de facto* para a descrição de serviços *web*.

3.4 Processo de desenvolvimento baseado em modelos para serviços de análise em bioinformática

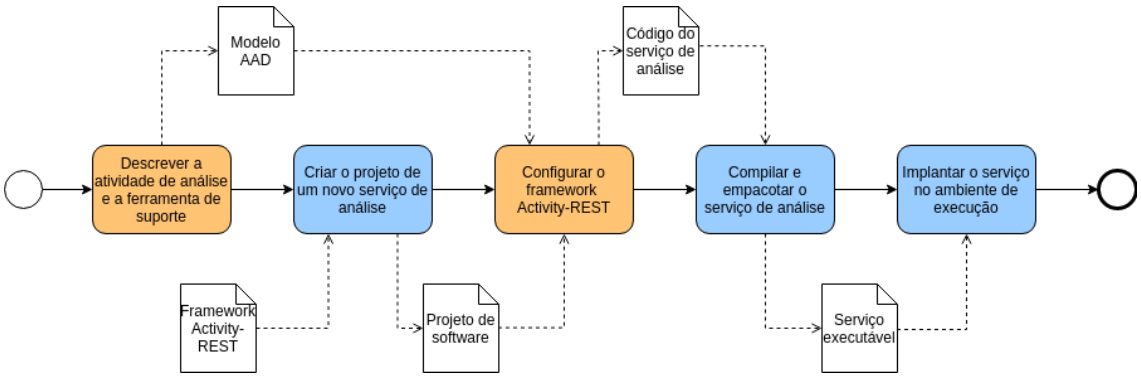
Após a definição dos requisitos para os serviços de análise, seus clientes e suas descrições, definimos um processo de desenvolvimento orientado a modelos desses artefatos. O desenvolvedor desses artefatos faz uso de modelos específicos de domínio para a descrição das características principais da atividade de análise a ser suportada pelo serviço, bem como da ferramenta de análise a ser adaptada para a execução dessa atividade e da implantação do serviço no ambiente de execução final. A partir dos modelos desenvolvidos, um conjunto de interpretações e transformações de modelo providos por bibliotecas e *frameworks* de suporte são utilizadas de modo a obter serviços, clientes e descrições de serviço automaticamente.

3.4.1 Processo de desenvolvimento de um serviço de análise

O processo de desenvolvimento baseado em modelos para serviços de análise em bioinformática consiste de cinco etapas. A Figura 11 apresenta uma visão geral das etapas envolvidas no desenvolvimento de um novo serviço e dos principais artefatos associados a esse desenvolvimento, por meio de um diagrama *Business Process Model and Notation* (BPMN) [29].

A primeira etapa do desenvolvimento de um novo serviço consiste da *descrição da atividade de análise e da ferramenta de suporte*. Nessa etapa, o desenvolvedor

Figura 11 – Processo de desenvolvimento de um serviço de análise.



Fonte: Autoria própria. Um retângulo azul representa um procedimento executado de maneira semi-automática por meio do suporte de artefatos e ferramentas de apoio existentes ou definidos nesse trabalho. Por sua vez, um retângulo laranja representa um procedimento manual executado pelo desenvolvedor do serviço.

do novo serviço descreve a atividade de análise de modo a definir os conjuntos de dados de entrada e os parâmetros de execução necessários para a execução dessa atividade, assim como os conjuntos de dados de saída criados após a sua execução. Adicionalmente, o desenvolvedor também descreve a maneira que a ferramenta de análise deve ser invocada sobre os conjuntos de dados e parâmetros definidos pela atividade de análise, códigos de erro que podem ocorrer e outros aspectos dessa ferramenta. Para realizar essa descrição, o desenvolvedor define um modelo específico de domínio chamado *Analysis Activity Description Model*, ou *Modelo AADM*. O Capítulo 4 apresenta o Modelo AADM e a linguagem específica de domínio definidos para essa atividade.

A segunda etapa do desenvolvimento do novo serviço consiste da *criação de um novo projeto de serviço de análise*. Nessa etapa, o desenvolvedor cria um projeto de *software* que irá conter o código do novo serviço. Esse projeto deve, também, importar todas as dependências necessárias para a compilação e execução do novo serviço de análise. De modo a facilitar o desenvolvimento de um serviço para um serviço de análise, definimos o *framework* Activity-REST. O *framework* Activity-REST implementa um serviço de análise RESTful genérico segundo um modelo de referência adequado para a execução de atividades de análise em bioinformática. Por fim, o *framework* Activity-REST provê facilidades para que a geração de todo o esqueleto do novo projeto seja obtido semi-automaticamente por meio da invocação de um único comando, reduzindo o esforço para a obtenção desse serviço. O capítulo 4 apresenta o modelo de referência para serviços de análise RESTful utilizado no desenvolvimento de *framework* Activity-REST, enquanto o Capítulo 5 apresenta o *framework* em si.

A terceira etapa do desenvolvimento do novo serviço consiste da *configuração*

do framework Activity-REST. Nesta etapa, o desenvolvedor adiciona o modelo AADM previamente definido no projeto do novo serviço de análise, configurando o *framework* por meio desse modelo. O *framework* Activity-REST é capaz de prover a interface e o comportamento esperados para a atividade de análise específica, adaptando a ferramenta de suporte por meio da interpretação do modelo AADM desenvolvido na primeira etapa do processo. Ao final dessa etapa, o desenvolvedor possui um projeto de *software* pronto para ser compilado em um novo serviço *Web*.

A quarta etapa do desenvolvimento do novo serviço consiste da *compilação e empacotamento do serviço de análise*. Nessa etapa, o desenvolvedor invoca as ferramentas que irão compilar o código executável, bem como empacotar esse código compilado para ser implantado em um ambiente de execução.

A última etapa do desenvolvimento do novo serviço consiste da sua *implantação em um ambiente de execução*. A ferramenta de análise alvo da adaptação pelo serviço deve estar instalada na máquina hospedeira na qual o serviço será implantado, podendo ser invocada por esse serviço sem qualquer restrição. Ao final dessa etapa, o desenvolvedor terá obtido um serviço de análise acessível por meio de uma interface RESTful.

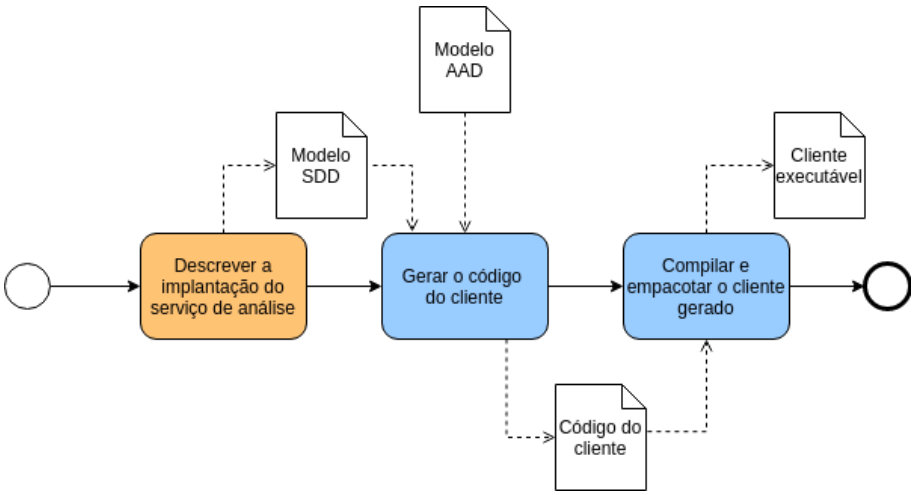
3.4.2 Processo de desenvolvimento de um cliente para um serviço de análise

O processo de desenvolvimento de um cliente para um serviço de análise consiste de três etapas. O serviço de análise de interesse deve ter sido desenvolvido e implantado em um ambiente de execução acessível por meio da Internet. Ao final do processo, o desenvolvedor irá obter um cliente para esse serviço, executável em um dos ambientes de análise suportados. Esse cliente será capaz de interagir com este serviço e executar a atividade de análise remotamente, executando todos os passos para a interação com o serviço de análise automaticamente. A Figura 12 apresenta uma visão geral desse processo por meio de um diagrama BPMN.

A primeira etapa do desenvolvimento do cliente consiste da *descrição da implantação do serviço de análise*. Nessa etapa, o desenvolvedor descreve a implantação do serviço em relação ao endereço base deste serviço, seu nome e outras características relevantes. De modo a descrever a implantação do serviço, o desenvolvedor define um modelo específico de domínio chamado *Service Deployment Description Model*, ou Modelo SDDM. O Capítulo 4 apresenta o Modelo SDDM definido para essa atividade.

A segunda etapa do desenvolvimento do cliente consiste da *transformação dos modelos AADM e SDDM no código do cliente*. Nesta etapa, o desenvolvedor

Figura 12 – Processo de desenvolvimento de um cliente para um serviço de análise.



Fonte: Autoria própria. Um retângulo azul representa um procedimento executado de maneira semi-automática por meio do suporte de artefatos e ferramentas de apoio existentes ou definidos nesse trabalho. Por sua vez, um retângulo laranja representa um procedimento manual executado pelo desenvolvedor do cliente.

realiza um conjunto de transformações modelo-para-texto para obter todo o código fonte do cliente, bem como suas dependências, a partir dos modelos AADM e SDDM. Esta atividade é realizada de maneira automática por meio de uma *transformação executável* implementada para este propósito. Ao final dessa etapa, o desenvolvedor obtém um projeto contendo o código-fonte do novo cliente. O Capítulo 6 apresenta a transformação de modelos que provê suporte a essa etapa.

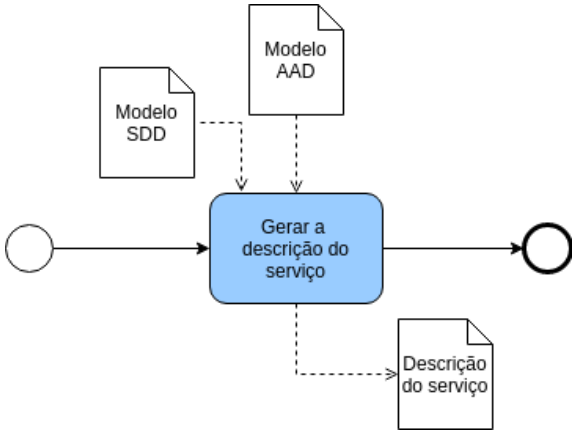
A última etapa do desenvolvimento do cliente consiste da *compilação e empacotamento do cliente gerado*. Nessa etapa, o desenvolvedor invoca as ferramentas que irão produzir código executável, bem como empacotar esse código de modo a facilitar sua distribuição.

3.4.3 Processo de desenvolvimento de uma descrição de um serviço de análise

O processo de desenvolvimento de descrições de um serviço de análise consiste de uma única etapa realizada para cada descrição de serviço desejada. O serviço de interesse deve ter sido desenvolvido e implantado em um ambiente de execução acessível por meio da Internet. Durante essa etapa, o desenvolvedor realiza uma transformação dos modelos AADM e SDDM em uma descrição de serviço WSDL ou OpenAPI. Essa transformação é automatizada por meio de uma ferramenta executável definida de maneira específica para a produção de descrições em cada uma dessas linguagens. O Capítulo 7 apresenta a transformação de modelos que provê suporte a essa etapa. Ao final dessa etapa, o desenvolvedor obtém a descrição

de serviço na linguagem de descrição escolhida. A Figura 13 apresenta uma visão geral desse processo por meio de um diagrama BPMN.

Figura 13 – Processo de desenvolvimento de uma descrição de um serviço de análise.



Fonte: Autoria própria. Um retângulo azul representa um procedimento executado de maneira semi-automática por meio do suporte de artefatos e ferramentas de apoio existentes ou definidos nesse trabalho.

3.5 Considerações finais

O processo de desenvolvimento orientado a modelos proposto define um número de atividades a serem executadas pelo desenvolvedor do serviço de análise. De modo a facilitar a utilização de nossa abordagem por um especialista de domínio, o trabalho define um conjunto de artefatos de suporte que permitem a obtenção de serviços de análise, seus clientes e suas descrições de maneira semi-automática a partir de um conjunto de modelos iniciais. O trabalho também define um modelo de referência para serviços RESTful de análise e provê um *framework* de suporte para facilitar o desenvolvimento de serviços adaptadores que implementam este modelo. Adicionalmente, o trabalho também provê bibliotecas para a criação de clientes para estes serviços e implementa transformações de modelo executáveis para a obtenção automática de clientes e descrições dos serviços de análise criados.

O processo de desenvolvimento proposto faz uso de dois modelos específicos de domínio também definidos neste trabalho: o modelo AADM e o modelo SDDM. O modelo AADM permite a descrição da atividade de análise e da ferramenta de suporte a ser adaptada. Por sua vez, o modelo SDDM permite a descrição da implantação de um serviço de análise. Em conjunto, os modelos AADM e SDDM permitem descrever em alto nível de abstração os conceitos relevantes para a adaptação de uma ferramenta de análise por meio de um serviço *web* segundo a nossa metodologia.

4 Arquitetura de metamodelagem para serviços adaptadores, clientes e descrições de serviço

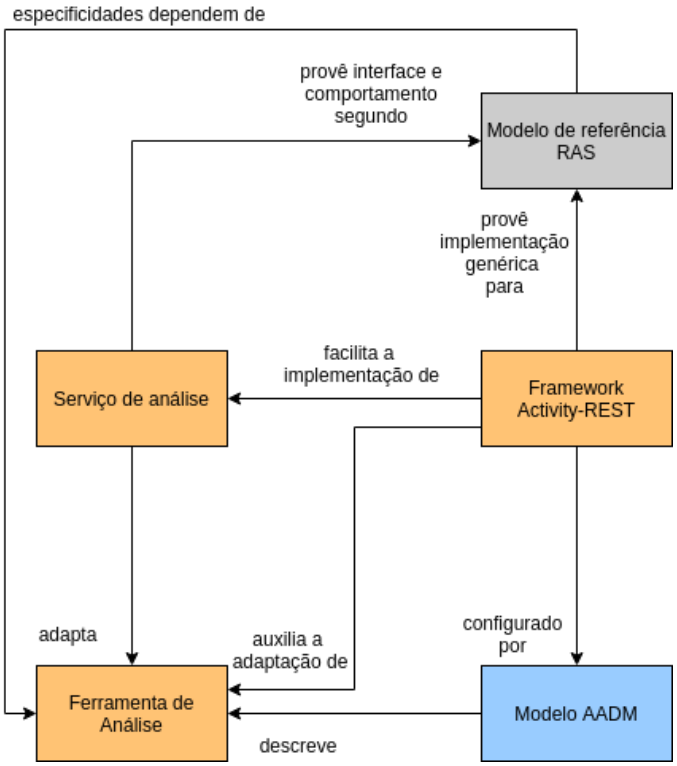
Uma arquitetura de metamodelagem foi definida para prover suporte à trajetória de desenvolvimento baseada em modelos para serviços de análises capazes de adaptar um ferramenta de análise existente. Essa arquitetura faz uso de um modelo de referência definido para especificar a interface e o comportamento de um serviço de análise, bem como de um metamodelo específico de domínio e de um *framework* que provê uma implementação genérica para um serviço adaptador. Uma arquitetura de metamodelagem também foi definida para o suporte à trajetória de desenvolvimento para clientes e descrições para os serviços de análise implementados. Esta arquitetura também faz uso do metamodelo específico de domínio para a descrição de uma atividade de análise, bem como faz uso de um segundo metamodelo específico de domínio para a descrição de implantações de serviço e utiliza transformações de modelo para a obtenção dos artefatos alvo.

Este capítulo está dividido da seguinte maneira: a Seção 4.1 apresenta a arquitetura de metamodelagem para serviços de análise; a Seção 4.2 apresenta a arquitetura de metamodelagem para clientes e descrições para esses serviços; a Seção 4.3 define o modelo de referência para serviços de análise em bioinformática; as Seções 4.4 e 4.5 apresentam metamodelos específicos de domínio definidos para a arquitetura de metamodelagem apresentada; por fim, a Seção 4.6 apresenta algumas considerações finais sobre esta etapa.

4.1 Arquitetura da geração de serviços de análise

A Figura 14 apresenta uma visão geral dos artefatos que compõe a trajetória de desenvolvimento definida para a obtenção de um serviço de análise por meio de uma abordagem de desenvolvimento orientada a modelos. *Ferramenta de análise* representa uma ferramenta com interface de linha de comando capaz de executar uma dada atividade de análise. Essa ferramenta pode tomar a forma de *scripts* para um dado *shell* ou um executável obtido em uma etapa ou pesquisa anterior. Essa ferramenta deve ser capaz de receber um ou mais argumentos em linha de comando e executar a atividade de análise automaticamente, produzindo um número de conjuntos de dados de saída após o término dessa execução.

Figura 14 – Principais artefatos da solução para obtenção de um serviço de análise.



Fonte: Autoria própria. Um retângulo cinza representa um modelo de referência, o qual define abstratamente os principais conceitos e o comportamento esperado parade um serviço de análise. Um retângulo laranja representa um artefato executável. Por fim, um retângulo azul representa um modelo descritivo que deve ser definido durante a criação do serviço.

Modelo de Referência RAS (*RESTful Analysis Service Reference Model* ou *Modelo RAS*) representa um modelo abstrato que define as características gerais de um serviço de análise a ser produzido. Este modelo de referência define a interface *web* exposta por um serviço de análise maduro, bem como as interações entre clientes e serviço necessárias para a completa execução de uma atividade de análise genérica. Especificidades da concretização de cada serviço RAS são definidas de acordo com a atividade e a ferramenta de análise a serem adaptadas.

Modelo AADM (*Analysis Activity Description Model*) representa um modelo descritivo utilizado para especificar formalmente as características mais relevantes de uma ferramenta de análise a ser adaptada. Este modelo define a atividade de análise e a ferramenta adaptada por meio dos conjuntos de dados utilizados ou produzidos durante a execução da ferramenta, bem como dos parâmetros utilizados para guiar essa execução.

Serviço de análise representa o serviço RESTful que irá adaptar a ferramenta de análise existente e prover a interface *web* para a execução da atividade de análise para usuários remotos. Este artefato deve ser produzido ao final da trajetória de

desenvolvimento. O serviço de análise produzido deve ser capaz de receber parâmetros de execução e os conjuntos de dados de entrada necessários para a atividade de análise, executar a ferramenta de análise sobre esses dados e, por fim, disponibilizar ao usuário do serviço os conjuntos de dados de saída criados durante essa execução. Um serviço de análise criado segundo o modelo de referências RAS é dito um serviço RAS.

Framework Activity-REST consiste de um *framework* para a adaptação de ferramentas de análise como serviços *web* RESTful. O *framework* Activity-REST provê uma implementação genérica para um serviço de análise definido segundo o modelo RAS. Este *framework* é configurado por meio do modelo AADM que descreve a ferramenta de análise a ser adaptada, auxiliando a adaptação dessa ferramenta e facilitando a obtenção de um serviço RAS que adapta essa ferramenta. Para isso, o *framework* Activity-REST recebe o modelo AADM e o interpreta, expondo a interface e o comportamento definidos pelo modelo de referência RAS.

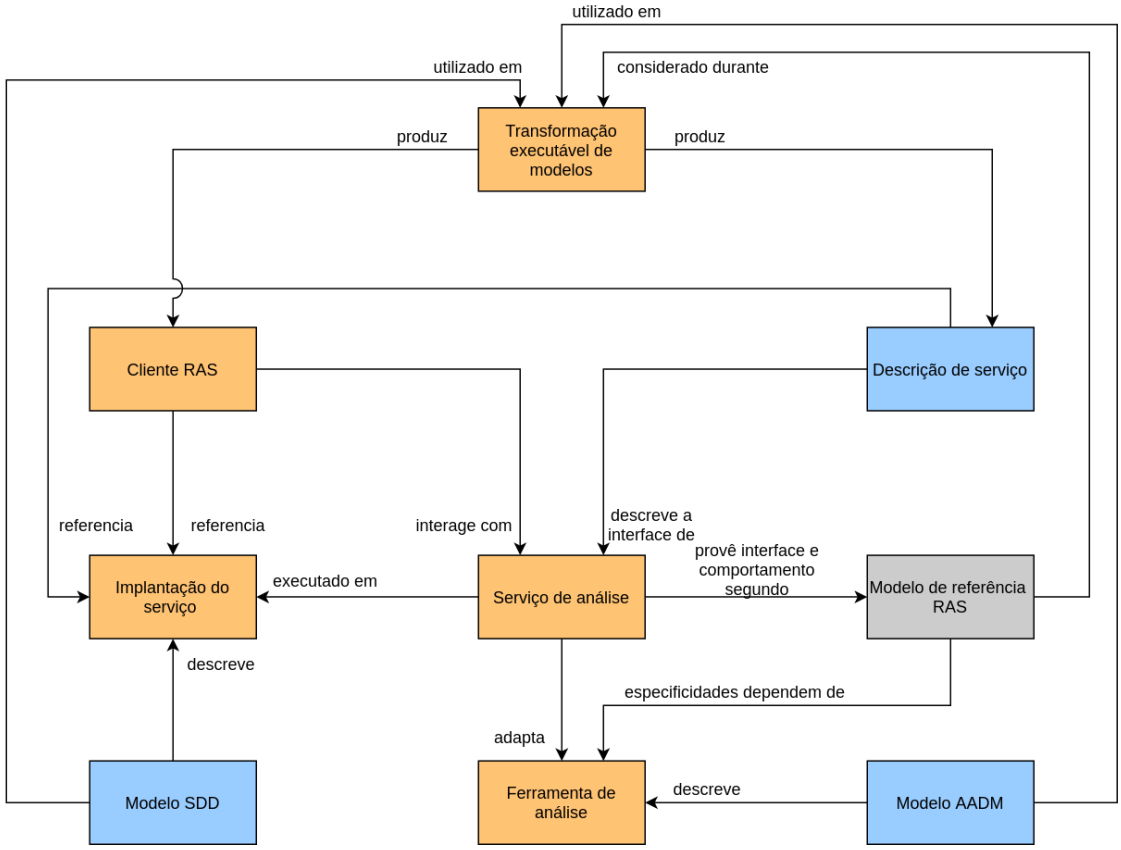
4.2 Arquitetura da geração de clientes e documentação para serviços de análise

A Figura 15 apresenta uma visão geral dos artefatos que compõe a trajetória de desenvolvimento de clientes e de descrições de serviço para serviços RAS. *Implantação do serviço* consiste da localidade lógica na qual o serviço RAS será executado. De modo a poder ser executado, um serviço *web* precisa ser implantado em uma dada máquina hospedeira acessível por meio de um endereço de rede. O endereço de rede de um serviço é dependente dessa implantação e cópias de um mesmo serviço podem existir em diferentes endereços de rede. Por esta razão, clientes e descrições de um serviço referenciam a localidade do serviço para que possam executar a atividade de análise por meio dela.

Um *Modelo SDDM* (*Service Deployment Description Model*) é utilizado para descrever formalmente as características mais relevantes de uma implantação de um serviço RAS. Cada implantação de serviço define um endereço base no qual o serviço pode ser encontrado, bem como a entidade responsável pelo provimento desse serviço e outras características próprias dessa implantação. Um modelo SDDM é utilizado para representar essas características da implantação durante a trajetória de desenvolvimento dos clientes e das descrições de serviço.

Cliente RAS consiste de um executável capaz de interagir com um serviço de análise de modo a executar uma dada atividade de análise por meio deste serviço. De modo a interagir com este serviço, o cliente RAS é definido de maneira específica para este serviço, interagindo com o serviço remoto segundo as definições do modelo

Figura 15 – Principais artefatos para a obtenção de clientes e descrições para serviços de análise.



Fonte: Autoria própria. Um retângulo cinza representa um modelo de referência, o qual define abstratamente os principais conceitos e o comportamento esperado um serviço de análise. Um retângulo laranja representa um artefato executável. Por fim, um retângulo azul representa um modelo descritivo que deve ser definido durante a criação do serviço.

de referência RAS particularizadas para a atividade de análise executada por aquele serviço. Adicionalmente, o cliente RAS deve conhecer o endereço base do serviço de análise definido durante a implantação do serviço RAS.

Descrição de serviço consiste de um modelo descritivo estrutural da interface apresentada por um serviço de análise RAS. Esta descrição de serviço formaliza a interface do serviço RAS definindo os *endpoints* e operações providos por este serviço, bem como das estruturas de dados compartilhadas entre cliente e serviço durante a execução da atividade de análise. A descrição de um dado serviço referencia a implantação deste serviço de modo a descrever sua interface completamente.

Uma vez que a interface e o comportamento de um serviço de análise RAS obedecem ao modelo de referência RAS e que as especifidades de cada serviço RAS podem ser obtidas a partir dos modelos AADM e SDDM, torna-se possível a obtenção de um cliente RAS e uma descrição de um serviço RAS de maneira automatizada.

Assim, *Transformação executável de modelos* consiste de um sistema executável que produz um cliente RAS ou uma descrição de serviço a partir de um modelo SDDM e de um modelo AADM. Esta transformação é definida de modo a concretizar o conhecimento implícito sobre as relações entre os modelos iniciais, o modelo de referência RAS e as estruturas presentes nos artefatos produzidos.

4.3 Modelo de referência para serviços de análise em bioinformática

Um modelo de referência foi definido para guiar a implementação de serviços de análise em bioinformática. Este modelo provê diretrizes para estrutura de recursos presentes em um serviço de análise genérico, bem como provê um modelo de interação entre um usuário e este serviço de modo a executar uma dada atividade de análise genérica e recuperar seus resultados. Adicionalmente, o modelo de referência provê diretrizes sobre a concretização de um serviço de análise quanto ao endereçamento de recursos, formatos de representação e navegabilidade por meio de ligações de hipermídia.

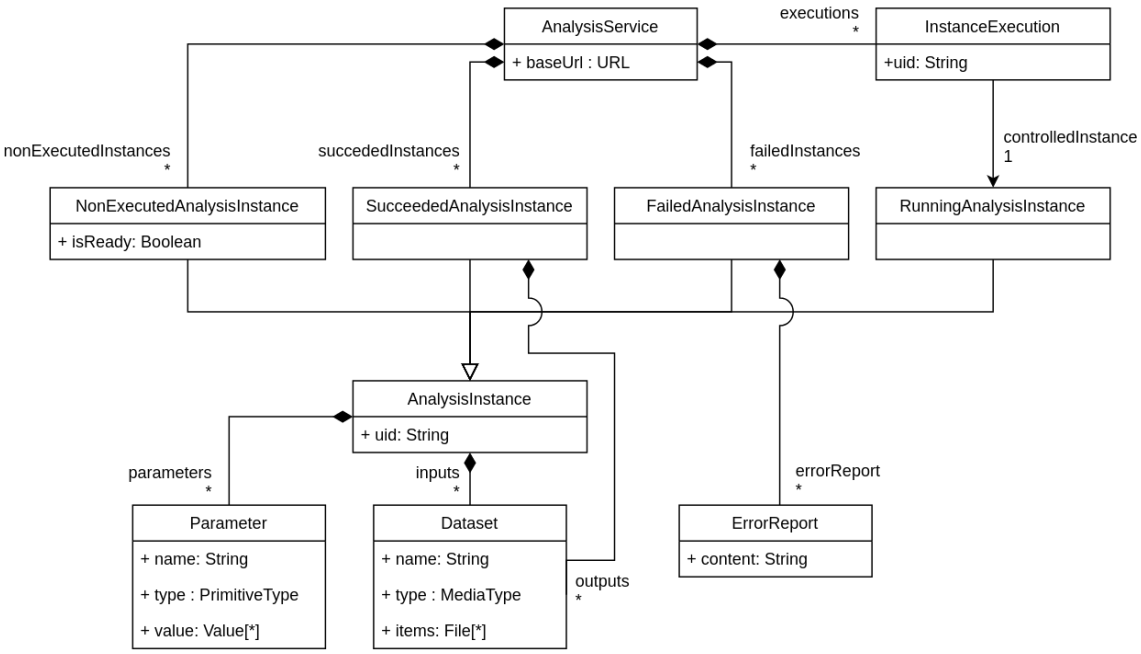
4.3.1 Modelo estrutural de um serviço de análise

Com base o ciclo de vida de uma atividade de análise em um serviço adaptador, um modelo conceitual dos principais recursos existentes nesses serviços foi definido. Este modelo conceitual define um conjunto de características estruturais gerais para os recursos que representam as instâncias da atividade de análise provida por um dado serviço, bem como provê diretrizes para a definição do comportamento geral desses recursos. A Figura 16 ilustra o modelo conceitual de um serviço de análise genérico por meio de um diagrama de classes UML.

`AnalysisService` representa um serviço RESTful capaz de executar uma dada atividade de análise por meio de um conjunto de interações com um usuário. Dessa maneira, o serviço de análise consiste em um recurso que perdura ainda que não haja instâncias de atividades de análise em execução ou previamente executadas. Um dado serviço de análise é identificado por um identificador uniforme de recurso (propriedade `baseUrl`).

`AnalysisInstance` representa uma instância de uma atividade de análise abstrata que evolui por um conjunto de estados. Cada instância é associada a um identificador imutável (propriedade `uid`). Diferentes características estruturais são encontradas em uma instância de atividade de análise de acordo com o estado que essa instância se encontra. Em qualquer estado do seu ciclo de vida, uma `AnalysisInstance` apresenta uma coleção de parâmetros de execução nomeados

Figura 16 – Modelo conceitual de um serviço de análise.



Fonte: Autoria própria.

(referência `parameters`) e uma coleção de conjuntos de dados de entrada (referência `inputs`).

`NonExecutedAnalysisInstance` representa uma instância de uma atividade de análise que foi criada, porém ainda não executada. `NonExecutedAnalysisInstance` possui uma propriedade booleana (propriedade `isReady`) para diferenciar entre instâncias de atividade de análise prontas para o processamento e instâncias de atividades de análise que precisam de dados adicionais (parâmetros ou conjuntos de dados) sejam informados. `RunningAnalysisInstance` representa uma instância de atividade de análise em execução. Enquanto em execução, não é possível interagir diretamente com a instância de atividade de análise, sendo possível apenas observar o estado dessa execução. `SucceededAnalysisInstance` representa uma instância de atividade de análise que foi executada com sucesso e, portanto, possui uma coleção de conjuntos de dados de saída (referência `outputs`) entre suas características estruturais. Finalmente, `FailedAnalysisInstance` representa uma instância de uma atividade de análise que foi executada sem sucesso por causa de um erro de processamento qualquer. `FailedAnalysisInstance` adiciona um relatório do erro (referência `errorReport`) entre suas características estruturais. `AnalysisService` mantém três coleções referentes a cada subclasse de `AnalysisInstance` passível de interação direta pelo usuário de serviço (referências `nonExecutedInstances`, `succeededInstances` e `failedInstances`).

`Parameter` representa um parâmetro de execução. Um parâmetro de execução

consiste de um valor nomeado (propriedade `name`) de um determinado tipo primitivo (propriedade `type`). Por sua vez, `DataSet` representa um conjunto de dados de entrada ou saída de uma análise. Um conjunto de dados de entrada ou saída consiste de um ou mais arquivos (propriedade `items`) que são conjuntamente nomeados (propriedade `name`) e apresentam uma representação de um tipo conhecido (propriedade `type`).

`InstanceExecution` representa um manipulador para o processo que controla a fase de execução de uma dada instância de atividade de análise, atuando como um intermediário entre o usuário do serviço e uma instância de atividade de análise em execução (`RunningAnalysisInstance`). O momento em que um `InstanceExecution` é criado marca o início da fase de execução da instância de atividade de análise associada. `InstanceExecution` indica se o processo de análise encontra-se em execução em um dado momento e, ao final do processamento, indica o estado de término da execução da instância de atividade de análise, bem como a localização final dessa instância entre as coleções mantidas por `AnalysisService`. `AnalysisService` mantém uma referência a cada `InstanceExecution` criado durante a existência do serviço (referência `executions`) e associa cada `InstanceExecution` à `AnalysisInstance` correspondente por meio de um mesmo valor para seus identificadores universais (propriedade `uid`).

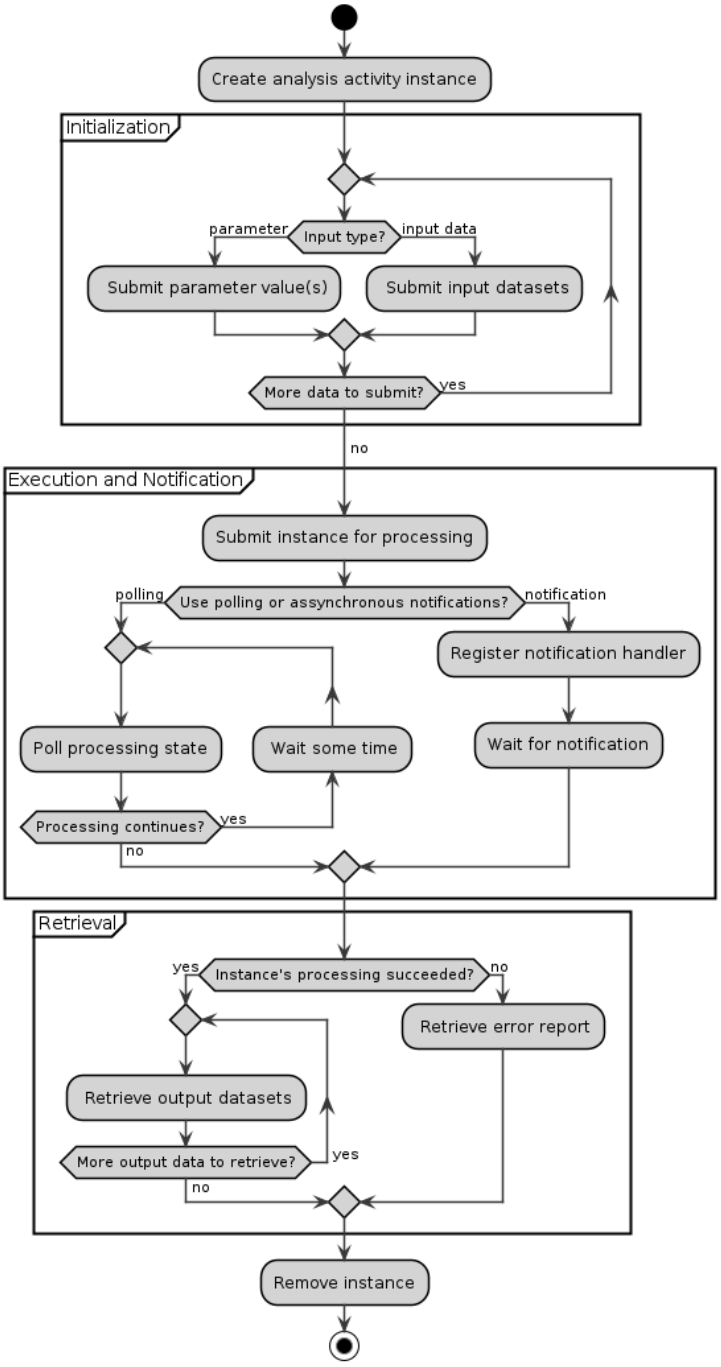
4.3.2 Modelo de interação RESTful

Após a definição do modelo conceitual de um serviço de análise, definimos um conjunto de passos (abstratos) que devem ser realizados para a execução de uma instância de atividade de análise por meio de um serviço RESTful. A Figura 17 apresenta uma visão geral desses passos por meio de um diagrama de atividades UML.

Inicialmente, o usuário do serviço requer a criação de uma instância da atividade de análise provida pelo serviço (`Create analysis activity instance`). Após a criação dessa instância, o usuário informa ao serviço os parâmetros de execução e os conjuntos de dados de entrada necessários para o seu processamento (`Send parameter value` e `Send input data`). A inicialização dos parâmetros e conjuntos de dados de entrada pode ser realizada iterativamente e em qualquer ordem.

Após a inicialização dos parâmetros e dos conjuntos de dados de entrada, o usuário solicita o processamento da instância (`Submit instance for processing`). Em seguida, o usuário verifica periodicamente o estado desse processamento (`Poll processing state`), até que o mesmo termine. Alternativamente, o usuário pode registrar um *handler* para receber notificações assíncronas do serviço (`Register notification handler`) e, então, aguardar até que uma notificação seja recebida por meio desse *handler* (`Wait for notification`)

Figura 17 – Processo de criação e execução de uma atividade de análise.



Fonte: Autoria própria.

O processamento da atividade de análise pode terminar com sucesso ou com falha. Se o processamento da instância terminar com sucesso, o usuário recupera os resultados desse processamento (`Retrieve output data`). Essa recuperação pode ser realizada iterativamente para cada conjunto de dados de saída. Se o processamento da instância terminar com falha, o usuário recupera o relatório contendo um indicativo da causa do erro (`Retrieve error report`). Por fim, após a recuperação dos conjuntos de dados de saída ou do relatório de erro, o usuário remove as informações da instância da atividade de análise do serviço (`Remove instance`).

Os diversos passos do processo de criação e execução de uma instância de atividade de uma análise são associados a transições de estado do ciclo de vida de uma atividade de análise. Por sua vez, cada transição de estado está associada a um conjunto de interações do modelo de interação RESTful. Essas associações são exploradas com maiores detalhes na sequência.

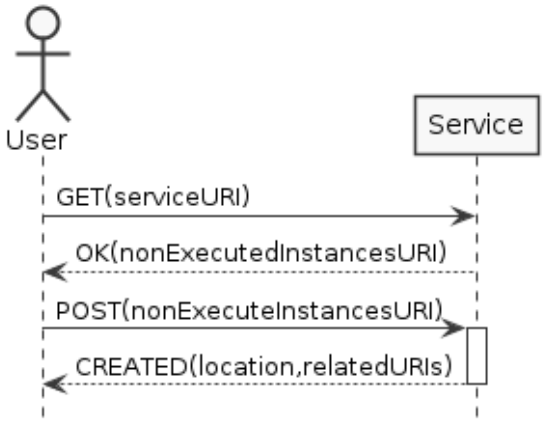
4.3.2.1 Criação de uma instância de uma atividade de análise

Um dado serviço de análise persiste independentemente da existência de alguma instância de atividade de análise. Uma requisição `GET` ao recurso base retorna um URI para a coleção de recursos que representam as instâncias de atividades de análise ainda não executadas (`nonExecutedInstances`). Embora o usuário não deva ser capaz de obter uma listagem (completa) dessa coleção, este usuário deve ser capaz de criar e acessar sub-recursos da coleção individualmente.

Uma nova instância de atividade de análise deve ser criada por meio de uma requisição `POST` à coleção de instâncias não executadas. O serviço valida essa requisição e cria um novo recurso representando uma instância de atividade de análise no estado `CREATED`, bem como representações vazias para cada parâmetro e dado de entrada da análise. Adicionalmente, valores padrões para os parâmetros de execução são aplicados (se definidos). Em seguida, retorna código de status `201 - Created` e *hyperlinks* para todos os sub-recursos da instância de atividade de análise que representam parâmetros e dados de entrada declarados (incluindo aqueles com valores padrões).

A Figura 18 apresenta um diagrama de sequência UML contendo as interações entre um usuário e um serviço com vistas à criação de um contexto de execução e uma instância de atividade de análise. Ao final dessa operação, o contexto de execução e um recurso representando uma instância de atividade de análise são criados e são individualmente endereçáveis. Esse recurso tem seus conjuntos de parâmetros inicializados com valores padrões (se definidos) e os conjuntos de dados de entradas e resultados vazios.

Figura 18 – Criação de uma instância de atividade de análise.



Fonte: Autoria própria.

4.3.2.2 Inicialização de parâmetros e dados de entrada

Uma vez que a instância de atividade de análise foi criada, o usuário do serviço deve informar valores de parâmetros de execução e os conjuntos de dados de entrada que serão utilizados. Essas operações podem ser realizadas tanto no estado `CREATED` quanto no estado `READY` do ciclo de vida de uma instância atividade de análise sem que ocorra a mudança para outro estado. Porém, eventualmente uma transição de estados `CREATED-READY` ocorrerá quando todos os parâmetros de execução e conjunto de dados de entrada tiverem a cardinalidade mínima de valores informados.

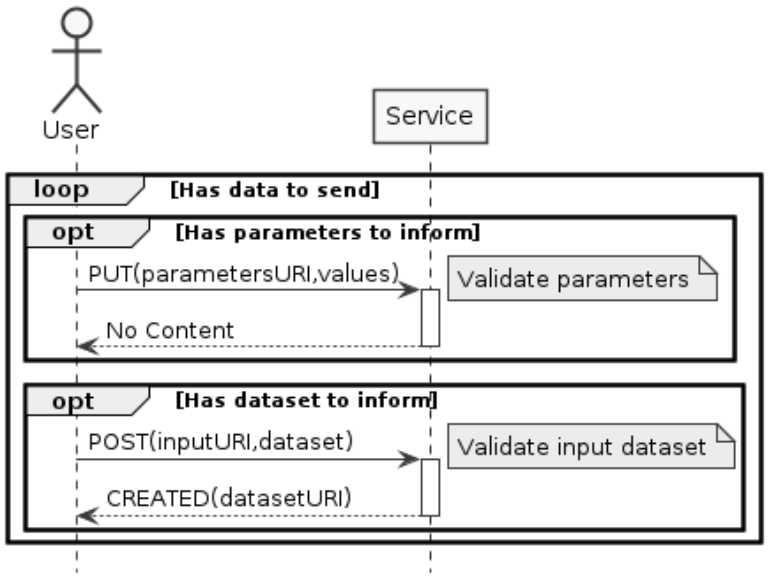
Parâmetros de execução de uma atividade de análise consistem, geralmente, de um conjunto pequeno de dados. Assim, o usuário pode informar os parâmetros para uma instância de modo conveniente por uma única requisição `PUT` enviando uma representação estruturada, como XML ou JSON, ao recurso que representa o conjunto de parâmetros de execução. Como consequência, os valores contidos neste recurso serão substituídos pelos valores enviados junto à requisição. Adicionalmente, o usuário pode definir cada parâmetro separadamente por meio de uma requisição `PUT` a cada recurso independente que representa um parâmetro de entrada. O serviço valida os parâmetros enviados e responde com um código de *status* que indique sucesso caso os valores informados sejam válidos, como, por exemplo, `204 - No Content`. O código de *status* `400 - Bad Request` deve ser retornado caso o usuário envie um conjunto de valores inválidos para os parâmetros definidos.

Por sua vez, os conjuntos de dados de entrada devem ser informados de maneira individual. O usuário do serviço pode informar o conteúdo de cada conjunto de dados de entrada separadamente por meio de requisições `POST` para conjuntos de dados de cardinalidade maior que 1, uma vez que essa operação geralmente

não é idempotente e tem como resultado a criação de novos recursos no serviço (i.e., arquivos para o conjunto de dados que serão independentemente endereçados). Preferencialmente, o usuário do serviço pode utilizar requisições `PUT` para informar o conteúdo de conjuntos de dados de cardinalidade unitária, uma vez que essa operação atualiza o conteúdo do recurso endereçado. Porém, por conveniência, o modelo RAS permite que usuários enviem arquivos para conjuntos de dados unitários por meio de requisições POST quando estes conjuntos de dados ainda não foram inicializados. O serviço valida o conjunto de dados de entrada enviado e responde com código de *status* `201 - Created` e um URI referente ao recurso criado no serviço. Novamente, o código de *status* `400 - Bad Request` deve ser retornado caso o usuário envie um conjunto inválido de dados de entrada. Adicionalmente, o usuário pode substituir um dado conjunto de dados de entrada já informado por meio de requisições `PUT`.

A Figura 19 apresenta um diagrama de sequência UML contendo as interações entre um usuário e um serviço de análise com vistas à inicialização de parâmetros de execução e de conjuntos de dados de entrada. A figura contém apenas as interações executadas com sucesso durante transições de estado `CREATED-CREATED`.

Figura 19 – Inicialização de parâmetros e dados de entrada de uma instância de atividade de análise.



Fonte: Autoria própria.

4.3.2.3 Execução de uma instância de atividade de análise

De modo a tratar atividades de análise com tempo indeterminado de processamento, a interação HTTP que inicia a execução de uma instância de atividade de análise retorna imediatamente. Neste sentido, a instância de atividade de análise é colocada em execução em segundo plano e o usuário do serviço recebe apenas uma

confirmação do início dessa execução. A partir de então, o modelo RAS permite que esse usuário utilize duas abordagens diferentes para acompanhar o estado dessa execução e verificar o seu resultado: espera com verificações periódicas (*polling*) ou espera com o recebimento de notificação assíncrona do término da execução da instância de atividade de análise (*Server-Sent Events*). O usuário do serviço pode escolher livremente entre essas abordagens, bem como mudar a abordagem em uso durante a execução de uma mesma instância de atividade de análise.

4.3.2.3.1 Espera utilizando verificações periódicas

A espera por meio de verificações periódicas é adequada para usuários que possuam à disposição clientes ou bibliotecas que permitam o uso apenas dos métodos HTTP básicos, sem acesso a outros mecanismos da *web* necessários para a notificação assíncrona do estado da atividade de análise. A interação do cliente com o serviço durante esse método de espera segue o padrão de interação *long running request* [56, 91]. Nesse padrão de interação, o cliente envia uma requisição para um recurso gerenciador de tarefas (*job manager*). O recurso gerenciador de tarefas valida essa requisição e imediatamente responde com um URI para um recurso que representa a reificação da nova tarefa (*execution resource*). Por meio desse recurso, o usuário verifica periodicamente o estado do processamento da requisição (*polling*).

Enquanto o recurso existe, i.e., retorna código de status `200 - OK` a uma requisição `GET`, o usuário sabe que a sua requisição inicial ainda está em processamento. Neste sentido, o usuário aguarda por um intervalo de tempo antes de repetir a verificação. Embora o serviço possa sugerir um tamanho para esse intervalo, pragmaticamente cabe ao usuário definir a periodicidade que irá utilizar na verificação do estado do trabalho.

Quando o processamento termina, novas requisições `GET` à reificação da tarefa retornam código de status `303 - See Other` e um URI que referencia os resultados do processamento no campo `Location` do cabeçalho da resposta. Alternativamente, o usuário pode cancelar o processamento por meio do envio de uma requisição `DELETE` ao recurso que reifica o processamento da atividade de análise.

O padrão de interação *long running request* não descreve qual resposta HTTP deve ser retornada para uma verificação do estado da execução da tarefa quando esta execução terminou com falha. Adicionalmente, não há um código de *status* HTTP dedicado para situações semelhantes. O uso do código de status padrão para erro de processamento (`500 - Internal Server Error`) torna a resposta ambígua, dado que esse código pode ser entendido como um erro ou problema temporário no serviço. Portanto, precisamos estender o padrão de interação *long running request*. Neste sentido, adicionamos a esse padrão a possibilidade de uma resposta conter o código

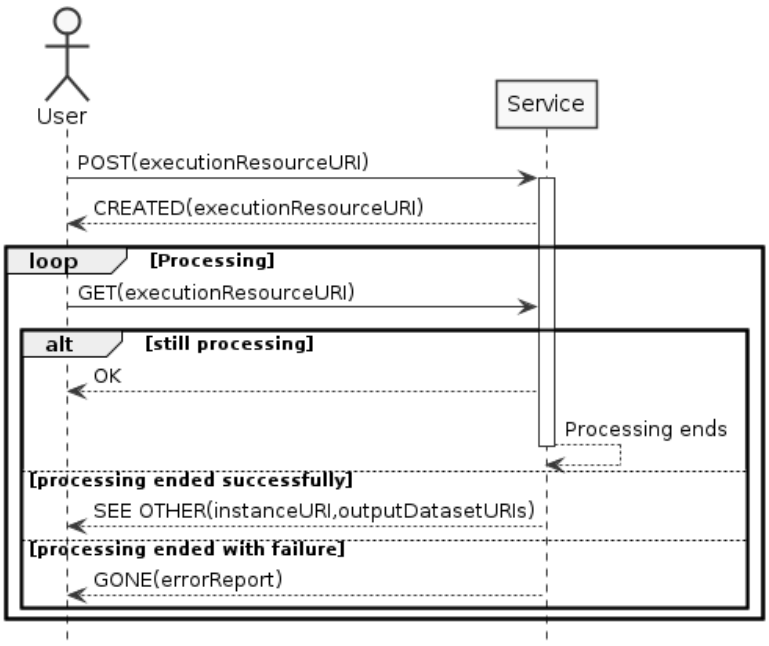
de *status* `410 - Gone` e uma representação de registros de erro da execução no caso da tarefa terminar em falha. Assim, o usuário do serviço é capaz de saber que o processamento acabou, mas que nenhum resultado está disponível.

Em nosso modelo de interação, quando a instância da atividade de análise encontra-se no estado `READY`, respostas às requisições do usuário informando parâmetros de execução ou conjunto de dados de entrada são acompanhadas por um URI de um recurso a ser criado no gerenciador de tarefas. Por meio desse URI, o usuário do serviço envia uma requisição `POST` ao gerenciador de tarefas, requisitando a criação de um novo recurso de execução para a instância da atividade de análise. O recurso gerenciador de tarefas verifica o estado da instância e aceita a requisição, imediatamente respondendo com um URI para um recurso que representa a reificação da nova tarefa (*job resource*). A URI do novo recurso pode ser tanto o próprio URI utilizado para criá-lo quanto um novo URI. Porém, em nosso modelo de iteração utilizamos o identificador da instância da atividade de análise para a criação do URI do recurso que reifica a tarefa que a executa.

O usuário é capaz de verificar o progresso da execução por meio do novo recurso criado. Neste sentido, periodicamente o cliente faz uma requisição `GET` para o recurso até receber como resposta o código de status `303 - See Other`, indicando que a processamento terminou com sucesso. Alternativamente, o usuário pode receber código de *status* `410 - Gone` e uma representação do relatório de erro de execução no caso do processamento ter terminado em falha. A Figura 20 apresenta um diagrama de sequência UML contendo as interações entre um usuário e um serviço de análise com vistas à execução de uma instância de atividade de análise.

Após a execução, a instância de atividade de análise residirá em uma das duas coleções que representam análises finalizadas, `succeededInstances` ou `failedInstances`, de acordo com seu estado ao final do processo.

Figura 20 – Execução de uma instância de uma atividade de análise.



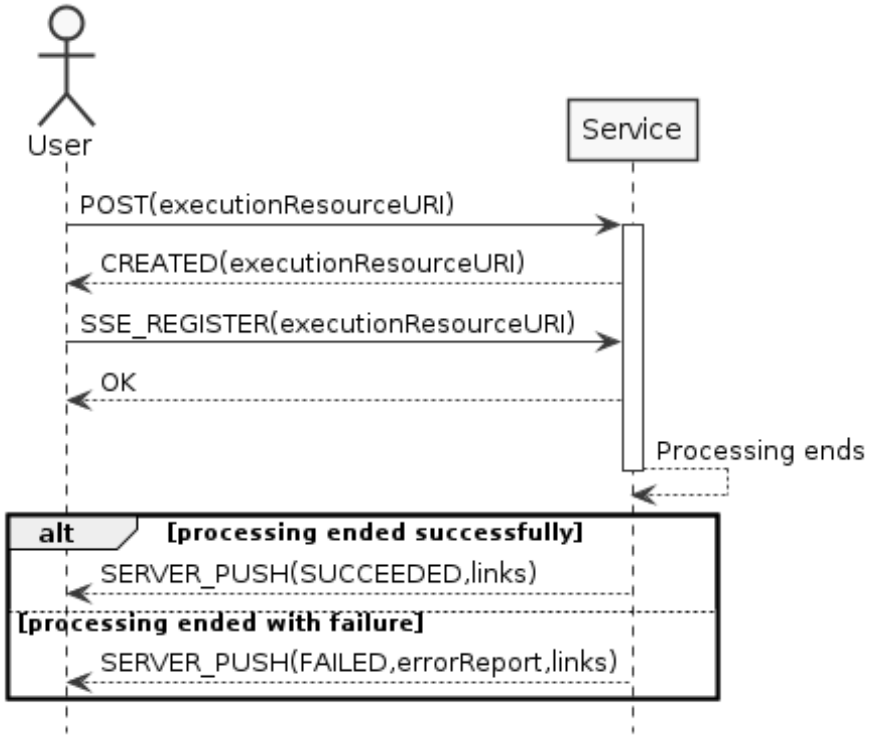
Fonte: Autoria própria.

4.3.2.3.2 Espera por meio de notificações assíncronas

Embora o método de verificação periódica possa ser utilizado por meio de clientes e bibliotecas HTTP mais simples, o número de requisições de *polling* pode tornar-se um gargalo aos recursos da rede conforme o número de usuários simultâneos aumenta. Uma maneira de evitar o consumo desnecessário de recursos por causa das requisições de *polling* é possibilitar que o usuário submeta a instância de atividade de análise para execução e aguarde passivamente até receber uma notificação sobre o término da execução dessa instância. Dessa maneira, não é necessário que requisições de *polling* sejam enviadas durante a execução da atividade de análise. Essa característica reduz o tráfego na rede do serviço e diminui a possibilidade de ocorrer uma negação de serviço acidental. Adicionalmente, durante essa espera o usuário também fica livre para realizar outras tarefas.

A arquitetura cliente-servidor do protocolo HTTP restringe que comunicações sejam iniciadas apenas pelo cliente de um serviço. Nesta arquitetura, o serviço é uma entidade passiva que aguarda as requisições de um cliente e retorna respostas a essas requisições. Porém, a extensão *Server-Sent Events* (SSE) [92] foi definida como um dos protocolos de comunicação da *web*. Por meio de *Server-Sent Events* é possível que um dado serviço notifique ativamente um cliente após uma conexão HTTP inicial ter sido realizada. Uma vez que a conexão é iniciada pelo cliente, o SSE respeita a separação cliente-serviço do modelo RESTful.

Figura 21 – Execução de uma instância de uma atividade de análise utilizando notificação por SSE.



Fonte: Autoria própria.

A Figura 21 apresenta uma visão geral do processo de submissão da atividade de análise para execução e a notificação do término dessa execução utilizando SSE. Até o início do processamento da instância de atividade de análise, o usuário segue o modelo de interação previamente definido. Após o usuário iniciar o processamento da instância de atividade de análise e receber a confirmação que essa instância iniciou sua execução, ao invés de o usuário começar um ciclo de espera e *polling*, o usuário faz uma nova requisição ao serviço para se registrar como destinatário de notificações sobre os eventos que ocorrem com aquela execução. Ao se registrar, o usuário recebe uma confirmação desse registro como resposta. Porém, diferentemente do que ocorre com o protocolo HTTP, a conexão TCP com o serviço é mantida aberta. Dessa maneira, o serviço de análise pode reutilizar essa conexão para notificar ao usuário a ocorrência de eventos, como o fim da execução e seu estado final, sem a necessidade de que novas requisições HTTP sejam ativamente disparadas por esse usuário.

Ao final da execução da instância de atividade de análise, o serviço de análise retorna uma representação desta instância em execução por meio da conexão previamente aberta. Essa representação contém o estado final da instância da atividade de análise, os URIs em que a instância pode ser encontrada após a execução e, no caso de atividade de análise que terminaram em falha, o relatório de erro gerado. A

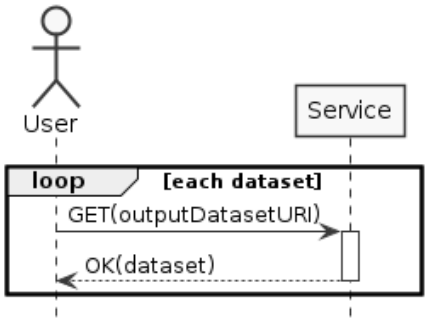
partir dessa notificação, o usuário utiliza os controles de hipermídia retornados para obter os próximos recursos a serem acessados, continuando a interação com o serviço segundo o modelo RAS para recuperar os resultados da execução da instância de atividade de análise.

O uso de SSE para a notificação do estado da execução da atividade de análise implica na existência de um conexão TCP ativa entre cliente e serviço durante essa execução. Essa conexão pode ser derrubada por falhas ou *timeouts* da rede entre cliente e serviço. Porém, o SSE permite que a conexão seja automaticamente reiniciada em caso de queda. O uso de SSE também implica que o usuário do serviço mantenha um *thread* bloqueado à espera das notificações enviadas pelo serviço enquanto o SSE é utilizado Dessa maneira, recursos da máquina do usuário ficam reservados durante essa espera, que pode ser longa para atividades de análise mais computacionalmente intensivas. Porém, nesses casos, o usuário do serviço pode decidir iniciar a espera utilizando notificação assíncrona e descartar essa abordagem em prol do uso de *polling* após algum limiar de tempo de preferência.

4.3.2.4 Recuperação de conjuntos de dados de saída ou do relatório de erro

Após a execução com sucesso de uma instância de atividade de análise, o usuário pode recuperar cada conjunto de dados de saída por meio de requisições `GET` ao sub-recurso que representa esse conjunto de dados. A recuperação de cada conjunto de dados pode ser realizada de maneira independente e em ordem aleatória. Essa recuperação mantém a atividade de análise no estado `SUCCEEDED`. A Figura 22 ilustra essa interação por meio de um diagrama de sequência UML.

Figura 22 – Recuperação de conjuntos de dados de saída.



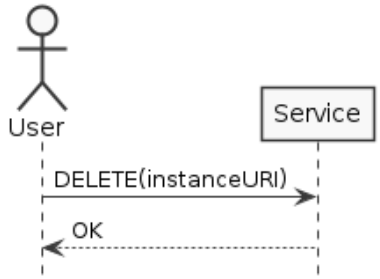
Fonte: Autoria própria.

De maneira semelhante à recuperação de conjuntos de dados de saída, o usuário pode recuperar o relatório de erro de execução por meio do sub-recurso `errorReport` da instância de atividade de análise que terminou em falha. Tal operação pode ser realizada enquanto a instância de atividade de análise executada existir.

4.3.2.5 Remoção de uma instância de atividade de análise

O estado final de uma instância de atividade de análise é o estado `FINISHED`. Uma transição para este estado pode ocorrer a partir de qualquer outro estado do ciclo de vida da atividade de análise, sendo disparada por meio de uma requisição `DELETE` ao recurso que representa a instância. O serviço remove todos os recursos relacionados à instância removida e retorna como resposta um código de status de sucesso, como, por exemplo, `200 - OK`. A Figura 23 ilustra essa interação por meio de um diagrama de sequência UML.

Figura 23 – Remoção de uma instância de atividade de análise.



Fonte: Autoria própria.

4.3.3 Diretrizes para a concretização da interface RESTful

4.3.3.1 Endereçamento de recursos

O modelo conceitual de um serviço de análise foi utilizado como base para a determinação dos identificadores hierárquicos (URI) para cada recurso que pode ser acessado por meio da interface RESTful desse serviço. Neste sentido, partindo de um URI base (*`AnalysisService:uri`*), o caminho para um dado sub-recurso é obtido por meio da concatenação dos nomes de cada referência de composição definida no modelo conceitual (veja Figura 16). Padronizamos os nomes dessas referências utilizando caixa baixa e uso de hífen para separação de palavras (por exemplo, a referência `nonExecutedInstances` é representada no URI por `non-executed-instances`)). Identificadores de instâncias de atividades de análise (atributo `uid`), parâmetros e conjuntos de dados (atributo `name`) são concatenados no URI para diferenciar diferentes instâncias destes recursos.

A Figura 24 apresenta um esquema geral dos URIs acessáveis em um serviço de análise como um diagrama de árvore. Nesta figura, a travessia de cada nível da árvore representa a adição de um segmento ao URI base (*`AnalysisService:uri`*). Recursos apresentados em laranja representam que aquele segmento é preenchido pelo valor da propriedade indicada. Demais identificadores devem ser adicionados literalmente ao caminho. Abreviações são utilizadas para nomes de classes mais

longos. Assim, as classes `NonExecutedAnalysisInstance`, `SucceededAnalysisInstance`, `FailedAnalysisInstance` e `InstanceExecution` são representadas pelas abreviações `NEAI`, `SAI`, `FAI` e `IE`, respectivamente. Por fim, a marcação encontrada ao lado de um retângulo indica a cardinalidade dos sub-recursos existentes naquele nível: `1`, para sub-recursos de cardinalidade unitária, e `*`, para sub-recursos com cardinalidade de 0 ou mais elementos.

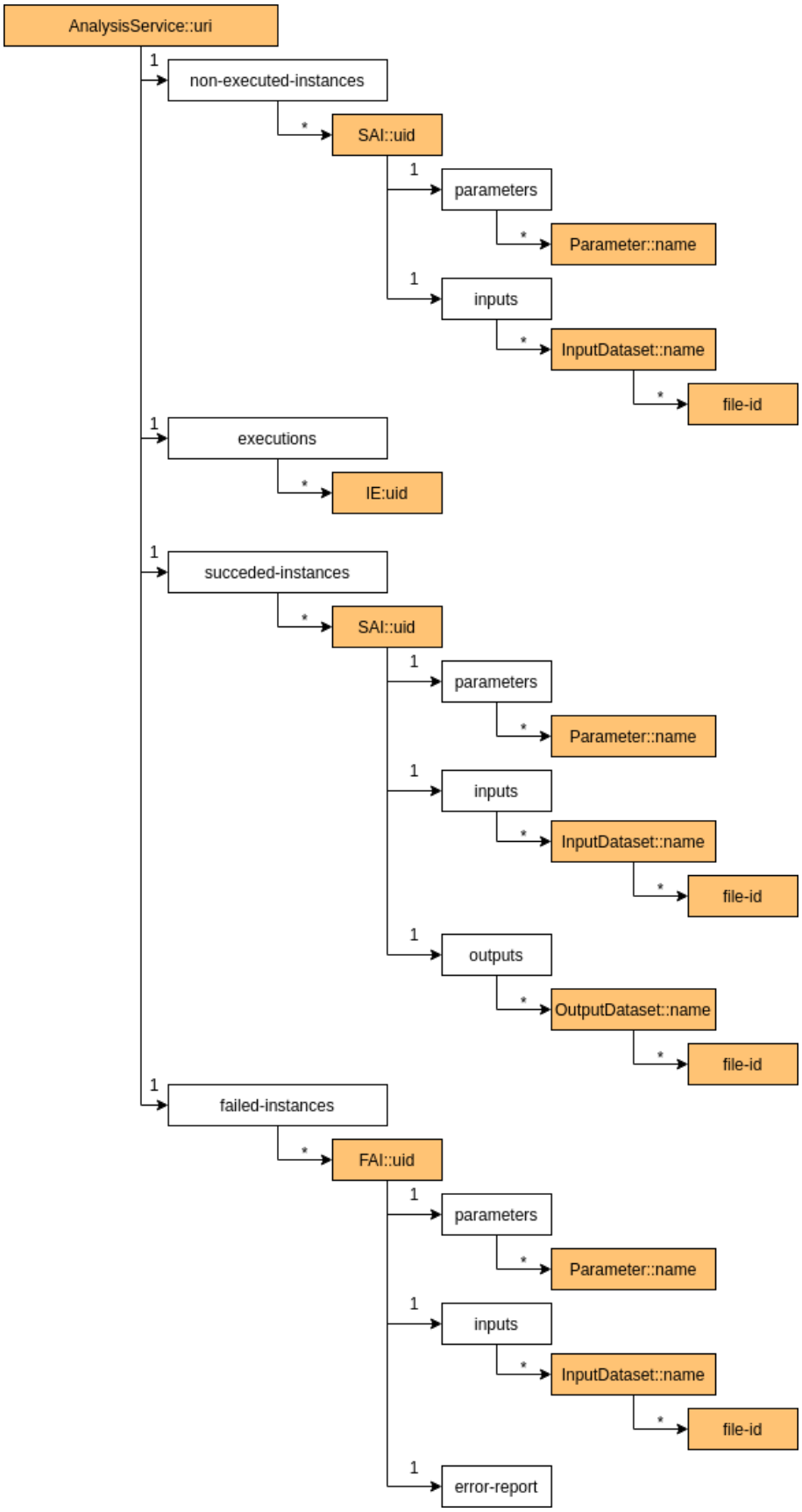
A Figura 25 apresenta um exemplo concreto do endereçamento de recursos de uma instância de atividade de análise no estado `SUCCEEDED`. Utilizamos como exemplo um serviço de análise que executa a atividade *DESeq2 Analysis*. Essa atividade recebe três parâmetros de execução, `threshold`, `cutoff` e `use-false-discovery-rate`. Adicionalmente, essa atividade é executada sobre dois conjuntos de dados de entrada, `reference-samples` e `target-samples`, para os quais um ou mais arquivos são submetidos pelo usuário. Ao final da execução, um único conjunto de dados de saída é produzido, chamado `result-file`, representado como um único arquivo. Suprimimos alguns elementos dessa figura para torná-la mais sucinta. Elementos suprimidos são representados contendo três pontos (`...`).

O serviço de análise possui um URL base (`http://.../deseq2-analysis/`), sob o qual todos os demais recursos são endereçados utilizando o caminho representado na árvore. Nesse URL, os três pontos representa um endereço do hospedeiro do serviço e a porta de utilizada por este serviço como, por exemplo, `kode.ffclrp.usp.br:8081`. A instância de atividade de análise representada na figura recebeu o identificador `93-oz` durante a sua criação. Este identificador é fictício, tendo sido definido como uma *string* pequena nesse exemplo apenas para tornar o exemplo mais sucinto. Com base no esquema de endereçamento proposto, o arquivo identificado como `file1` do conjunto de dados de entrada `reference-samples` da instância identificada por `93-oz` é endereçado como `http://.../deseq2-analysis/succeded-instances/93-oz/inputs/reference-samples/file1`. Por sua vez, o conjunto de parâmetros de execução pode ser acessado por meio do endereço `http://.../deseq2-analysis/succeded-instances/93-oz/parameters`, enquanto cada parâmetros pode ser também individualmente endereçado adicionando o nome desse parâmetro a esse URL. Os caminhos dos demais recursos podem ser obtidos de maneira análoga.

4.3.3.2 Formatos de representação padrões

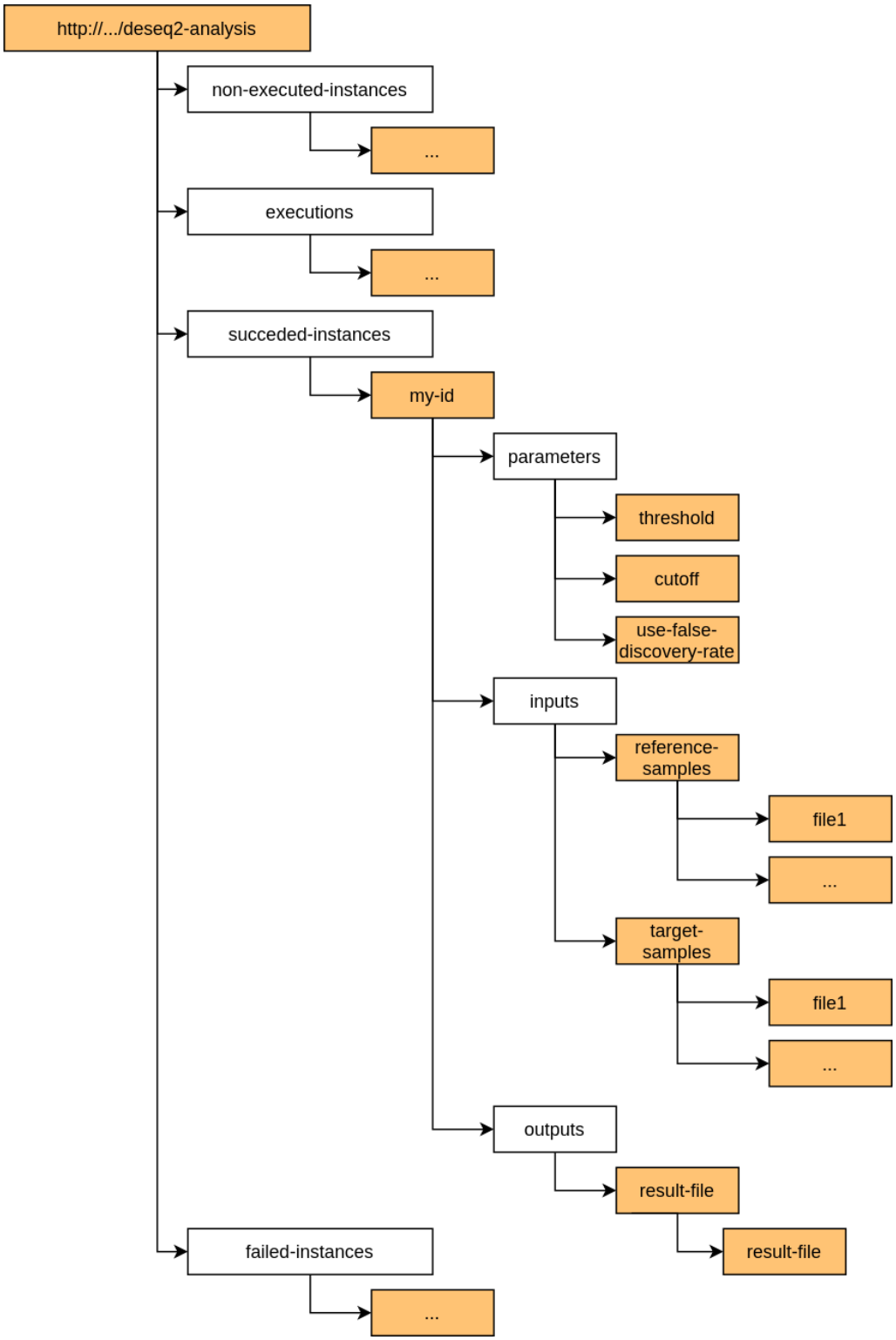
Formatos de serialização padrões devem ser providos para a representação dos recursos RESTful e para a troca de informações entre um usuário e um serviço de análise. O serviço deve prover suporte ao uso de *JavaScript Object Notation* (JSON) e *eXtensible Markup Language* (XML) para a representação dos recursos necessários para a execução da atividade de análise, como as instâncias de `NonExecutedAnaly-`

Figura 24 – Visão geral da atribuição de URIs para os sub-recursos padrões de um serviço de análise segundo o modelo RAS.



Fonte: Autoria própria.

Figura 25 – Exemplo dos segmentos de endereçamento de uma instância de atividade de análise no estado `SUCCEEDED`.



Fonte: Autoria própria.

`sisInstance` e seus parâmetros. Essa diretriz foi definida dado que estes formatos de serialização possuem extensivo suporte tecnológico.

O serviço deve prover a utilização do formato de serialização já reconhecido pela ferramenta subjacente para a representação do conteúdo de conjuntos de dados de entrada e saída (i.e., arquivos). Dessa maneira, um cliente pode fazer uma operação simples de *upload* de um arquivo que já possua para o conjunto de dados, bem como *download* em um formato de serialização conhecido. Adicionalmente, tal pragmatismo evita a necessidade da definição de modelos conceituais para todos os tipos de arquivos que um serviço de análise possa utilizar, assim como evita a realização de conversões entre XML/JSON e o formato de representação conhecido pela ferramenta. Porém, de modo a possibilitar que clientes apenas capazes de receber representações nestes formatos de serialização reconheçam o corpo da mensagem, suporte ao envelopamento do conteúdo de um conjunto de dados em uma representação JSON ou XML deve ser provido. Neste caso, as representações XML e JSON devem possuir ao menos dois campos: `content`, o qual deve apresentar o conteúdo do conjunto de dados representado como texto plano (utilizando, para isso, o algoritmo de codificação `Base64`); e `mediaType`, o qual indica o tipo MIME original do arquivo contido no campo `content`.

4.3.3.3 Relações HATEOAS

Controles de hipermídia retornados por um dado serviço RESTful permitem que usuários naveguem por este serviço sem conhecer de antemão o esquema de endereçamento dos recursos disponíveis. Esses controles são retornados como metadados das representações compartilhadas entre cliente e serviço, seja junto às respostas HTTP providas pelo serviço, seja por meio do cabeçalho HTTP `Link` ou internamente à representação destes recursos.

As relações utilizadas nos controles de hipermídia de um dado serviço RESTful devem ser formalizadas de modo a permitir que usuários desses serviços possam utilizá-las para navegação. Uma vez formalizadas, podemos desenvolver aplicações clientes para esse serviço que sejam capazes de utilizá-lo corretamente apenas navegando pelos controles de hipermídia retornados. Essa aplicação cliente manter-se-á utilizável mesmo que o esquema de endereçamento dos recursos do serviço seja modificado. Por esta razão, definimos um conjunto de relações de hipermídia que devem ser suportadas por serviços RAS e podem ser utilizadas para clientes desses serviços.

Para o modelo RAS, os URIs dos recursos devem ser compartilhados entre serviço e usuário por meio do cabeçalho HTTP (utilizando as chaves `Link` e `Location`). Opcionalmente, esses URIs também podem ser compartilhados por meio de campos de metadados da representação do recurso por meio de Hypertext Application

Language (JSON-HAL e XML-HAL) [93]. A representação preferida de um URI é por meio de um endereço absoluto e não parametrizado, de maneira a reduzir a quantidade de operações executadas pelo usuário para acessar os recursos do serviço.

O formato de um controle de hipermídia em um cabeçalho `Link` em uma resposta HTTP consiste de um URI entre os sinais de maior-que e menor-que (`<` e `>`), seguido de uma lista de itens no formato chave-valor. Para a chave é utilizada a palavra-chave `rel`, enquanto como valor é colocado o identificador da relação entre aspas (`"`). Cada chave é separada de seu valor pelo sinal de igualdade (`=`) e cada item dessa lista é separado dos demais por um ponto-e-vírgula (`;`). O cabeçalho `Link` pode ser utilizado várias vezes em uma resposta HTTP para informar ao usuário do serviço diversas relações existentes entre o recurso retornado e outros recursos de interesse.

A Listagem 1 apresenta um exemplo do retorno de controles de hipermídia por meio do cabeçalho de uma resposta HTTP. Nesta listagem, o usuário realiza uma requisição `GET` ao recurso que representa uma instância de atividade de análise (linha 1). Esta instância é encontrada e uma resposta HTTP é retornada (linhas 3 a 9). Nessa resposta, quatro controles de hipermídia são retornados e apresentados na listagem. O controle contendo a relação `self` (linha 4) retorna o URL padrão do recurso acessado, podendo ser utilizado. O controle contendo a relação `parameters/-cutoff` (linha 5) indica que a atividade de análise possui um parâmetro chamado `cutoff` que pode ser acessado usando o URL associado. Os controles contendo as relações `inputs/sequences/a` e `inputs/sequences/b` (linhas 6 e 7, respectivamente) indicam que a instância de atividade de análise possui um conjunto de dado de entrada chamado `sequences`, para o qual dois arquivos (`a` e `b`) foram submetidos e são acessáveis por meio dos URL associados.

O Apêndice B apresenta em detalhes os controles de hipermídia definidos para os serviços RAS.

4.4 Analysis Activity Description Metamodel

Analysis Activity Description Metamodel (AADMM) consiste de um metamodelo definido para formalizar a descrição de uma atividade de análise em bioinformática. Este metamodelo foi concebido para satisfazer três necessidades: i) representar uma atividade de análise em relação aos parâmetros e conjuntos de dados que devem ser informados para sua execução; ii) representar uma ferramenta utilizada para a execução da atividade descrita; e, iii) definir como essa ferramenta subjacente pode ser invocada considerando-se os valores de parâmetros e conjuntos de dados providos. Dessa maneira, modelos AADM podem ser definidos de modo a

Listagem 1 – Exemplo de um controle de hipermídia representado no cabeçalho `Link` de uma resposta HTTP.

```
1 > GET /non-executed-instances/my-id HTTP/1.1
2
3 < HTTP/1.1 200 OK
4 < Link: <http://service.path/non-executed-instances/my-id>;rel="self"
5 < Link:
      <http://service.path/non-executed-instances/my-id/parameters/cutoff>
      rel="parameters/cutoff"
6 < Link:
      <http://service.path/non-executed-instances/my-id/inputs/sequences/a>;
      rel="inputs/sequences/a"
7 < Link:
      <http://service.path/non-executed-instances/my-id/inputs/sequences/b>;
      rel="inputs/sequences/b"
8 <
9 < ...
```

Fonte: Autoria própria.

representar formalmente uma atividade de análise de interesse. Posteriormente, estes modelos podem ser interpretados e/ou transformados de modo a prover o suporte necessário à execução da atividade de análise descrita. O metamodelo AADMM foi definido por meio do *framework Eclipse Modeling Framework*, provido pela Eclipse Foundation [75]. Os artefatos que formalizam este metamodelo estão disponíveis em um repositório Git próprio[1].

4.4.1 Representação de uma atividade de análise

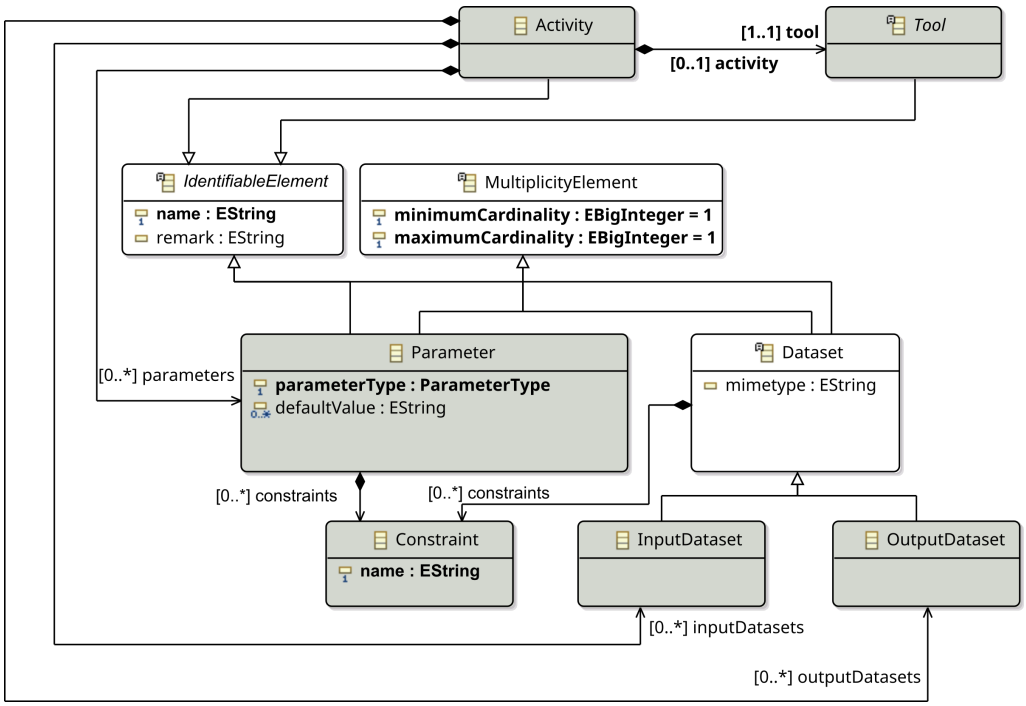
A Figura 26 apresenta as principais metaclasses definidas pelo metamodelo AADMM por meio de um diagrama de classes Ecore [75]. A metaclasse abstrata `IdentifiableElement` representa um elemento que pode ser identificado de forma unívoca (atributo `name`) em um modelo AADM. Adicionalmente, `IdentifiableElement` permite a definição de um comentário textual sobre o elemento (atributo `remark`).

As metaclasses `Activity` e `Tool` representam os principais conceitos do metamodelo. A metaclasse `Activity` estende `IdentifiableElement` de modo a representar uma atividade de análise. `Activity` define os parâmetros e conjuntos de dados de entrada e saída utilizados e produzidos por essa atividade. Esta atividade de análise é realizada por meio da execução de uma ferramenta adequada. Neste sentido, a metaclasse abstrata `Tool` também estende `IdentifiedElement` de modo a representar uma ferramenta de análise.

A metaclasse abstrata `MultiplicityElement` representa um elemento que define o conceito de multiplicidade, isto é, um intervalo, contínuo e inclusivo, de números inteiros positivos que representam cardinalidades válidas para o elemento

[1] `https://purl.org/cawal/lssb/aadmm`

Figura 26 – Principais metaclasses do metamodelo AADMM.



Fonte: Autoria própria. Metaclasses abstratas são representadas como retângulos com fundo branco e com ícone apresentando a letra A como decoração. Metaclasses concretas são representadas como retângulos com fundo cinza.

modelado. A multiplicidade de um elemento define um número de instâncias mínimo e um número de instâncias máximo do elemento modelado para que a atividade de análise possa ser executada corretamente. A cardinalidade máxima um elemento múltiplo pode ser infinita, representada pelo valor especial `-1`.

A metaclasse `Parameter` estende `IdentifiableElement` de modo a representar uma declaração de um parâmetro de execução utilizado pela atividade de análise. O parâmetro declarado será concretizado como uma lista ordenada de valores literais, os quais serão passados à ferramenta que executará a atividade de análise. `Parameter` define um nome (atributo `name`) utilizado para ser referir à lista de valores assumida por aquele parâmetro em uma dada execução da atividade de análise modelada. Esta lista possui uma cardinalidade arbitrária definida pelos atributos `minimumCardinality` e `maximumCardinality` de `MultiplicityElement`. `Parameter` também permite a associação de um comentário textual sobre o significado esperado para o parâmetro modelado.

Um parâmetro de execução pode aceitar apenas valores de um dado tipo primitivo, representado pelo tipo enumerado `ParameterType`. Adicionalmente, `Parameter` possui ainda uma lista de *strings* que permite a definição de valores padrões para um parâmetro (lista `defaultValue`). Uma instância da metaclasse `Activity` possui uma coleção de instâncias de `Parameter` (coleção `parameters`) que mantém

a informação de todos os parâmetros utilizados durante a atividade de análise.

A metaclasse abstrata `Dataset` estende `IdentifiableElement` de modo a representar uma declaração de um conjunto de dados. `Dataset` define um nome para o conjunto de dados (atributo `name`), utilizado como identificador para este conjunto. Adicionalmente, é possível associar um tipo MIME à declaração de um conjunto de dados, como os tipos `text/csv` para um conjunto de dados formados por tabelas com colunas separadas por vírgula e `application/xml` para um conjunto de dados que receba dados estruturados em XML. Dessa maneira, o desenvolvedor pode indicar o formato de representação esperado para aquele conjunto de dados. Conjuntos de dados de entrada e conjuntos de dados de saída são representados por instâncias das metaclasses concretas `InputDataset` e `OutputDataset`, respectivamente, e associados às coleções `inputDatasets` e `outputDatasets` de `Activity`.

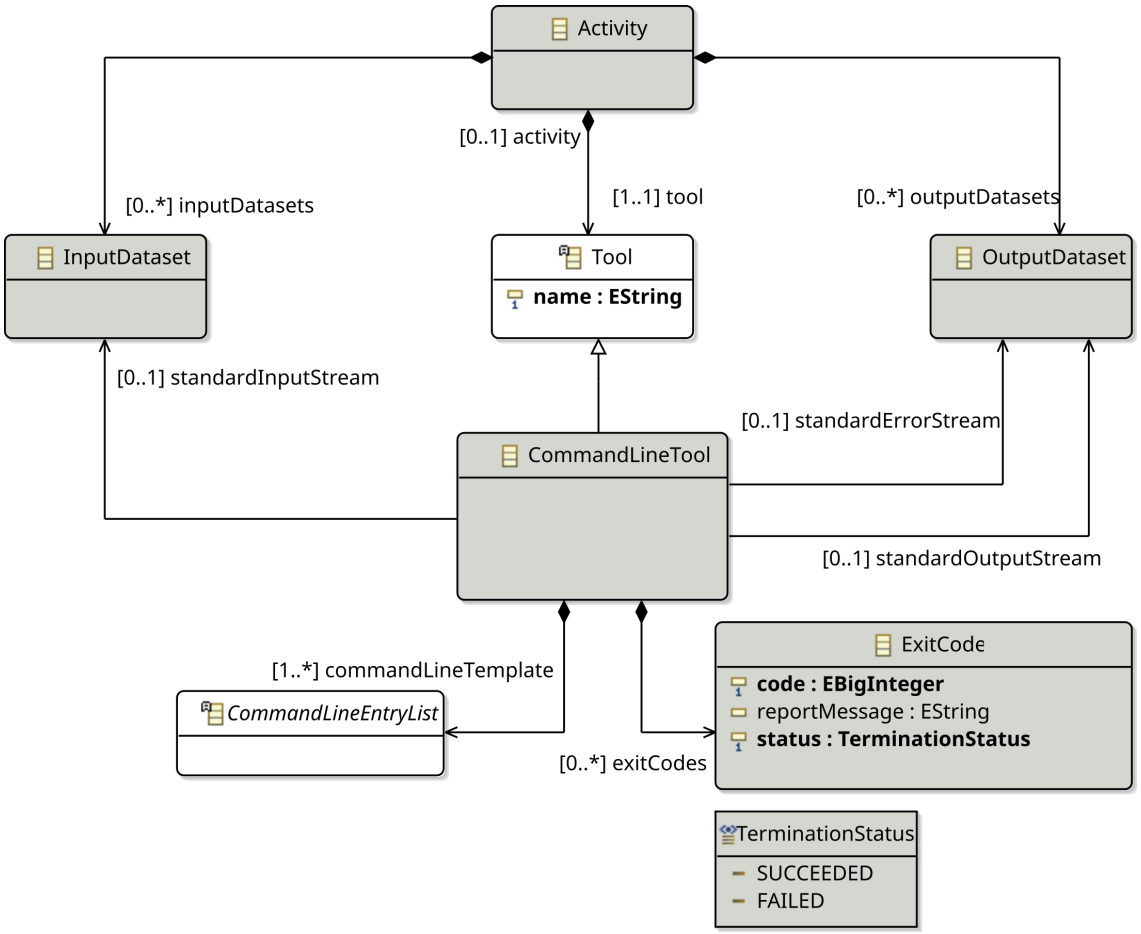
O metamodelo AADMM prevê que restrições sejam associadas a parâmetros e conjuntos de dados. Essas restrições podem ser criadas com o objetivo de validar os valores assumidos por estes elementos. Por exemplo, uma restrição pode ser criada para verificar a formatação do valor indicado para um parâmetro do tipo `STRING` como um endereço de *email* válido, ou garantir que arquivos informados para um conjunto de dados tenham tamanho limitado. Neste sentido, a metaclasse `Constraint` representa uma restrição nomeada. `Constraint` define um identificador (atributo `name`) utilizado para localizar um componente de *software* que realize a validação dessa restrição durante a interpretação e/ou transformação de um modelo AADM. As metaclasses `Parameter` e `Dataset` definem um conjunto de restrições aplicáveis por meio da coleção `contraints`, presente em ambas.

4.4.2 Representação da ferramenta que executa a atividade de análise

A Figura 27 apresenta uma visão geral das principais metaclasses associadas à definição de uma ferramenta utilizada para a execução de uma atividade de análise e invocada por meio linha de comando. `Activity` contém uma referência à metaclasse abstrata `Tool`. Esta metaclasse estende `IdentifiableElement` de modo a representar uma ferramenta capaz de executar uma atividade de análise. O nome de uma instância de `Tool` provê um identificador para essa ferramenta. `Tool` deve ser estendida de modo a capturar as especificidades da invocação dos diferentes tipos de ferramentas que tecnologias associadas ao metamodelo AADMM devem suportar. Atualmente, `Tool` é somente especializada pela metaclasse concreta `CommandLineTool`, de modo a representar uma ferramenta de linha de comando capaz de executar a atividade de análise. Diferentes propriedades de `CommandLineTool` são utilizadas para associar diferentes informações necessárias para a execução correta dessa ferramenta.

Durante a invocação de uma ferramenta por meio da linha de comando, um

Figura 27 – Principais metaclasses associadas à definição de uma ferramenta de linha de comando.



Fonte: Autoria própria. Metaclasses abstratas são representadas como retângulos com fundo branco e com ícone apresentando a letra A como decoração. Metaclasses concretas são representadas como retângulos com fundo cinza.

conjunto de argumentos deve ser informado. Neste sentido, instâncias da metaclasse abstrata `CommandLineEntryList` permitem a definição desses argumentos a partir dos valores informados como parâmetros e conjuntos de dados. Estas instâncias são incluídas na lista ordenada `commandLineTemplate`, detalhada a seguir.

Após a execução de uma atividade de análise, a ferramenta responsável pela execução pode retornar um código de término. Neste sentido, a metaclasse `ExitCode` permite a definição de códigos de terminação retornados pela ferramenta (atributo `code`). `ExitCode` também permite indicar se o código retornado representa um término com sucesso ou falha de execução (atributo `status`, do tipo enumerado `TerminationStatus`). Por fim, é possível associar ao código de terminação uma mensagem textual que indique seu significado (atributo `reportMessage`). Instâncias de `ExitCode` são associadas à coleção `exitCodes` de `CommandLineTool`.

Três fluxos padrões de dados podem ser manipulados por uma ferramenta

linha de comando: a entrada padrão (STDIN), a saída padrão (STDOUT) e a saída padrão de erros (STDERR). Algumas ferramentas de linha de comando são criadas para ler um conjunto de dados passado pela entrada padrão, manipular esses dados e então produzir um conjunto de dados retornado pela saída padrão. Neste sentido, `CommandLineTool` possui as propriedades `standardInputStream`, `standardOutputStream` e `standardErrorStream` que permitem indicar conjuntos de dados de entrada e saída que devem ser associados a esses fluxos padrões de dados. Instâncias de `Dataset` associadas a essas propriedades devem ter cardinalidade mínima e máxima de apenas 1 arquivo.

4.4.3 Definição da invocação da ferramenta de análise subjacente

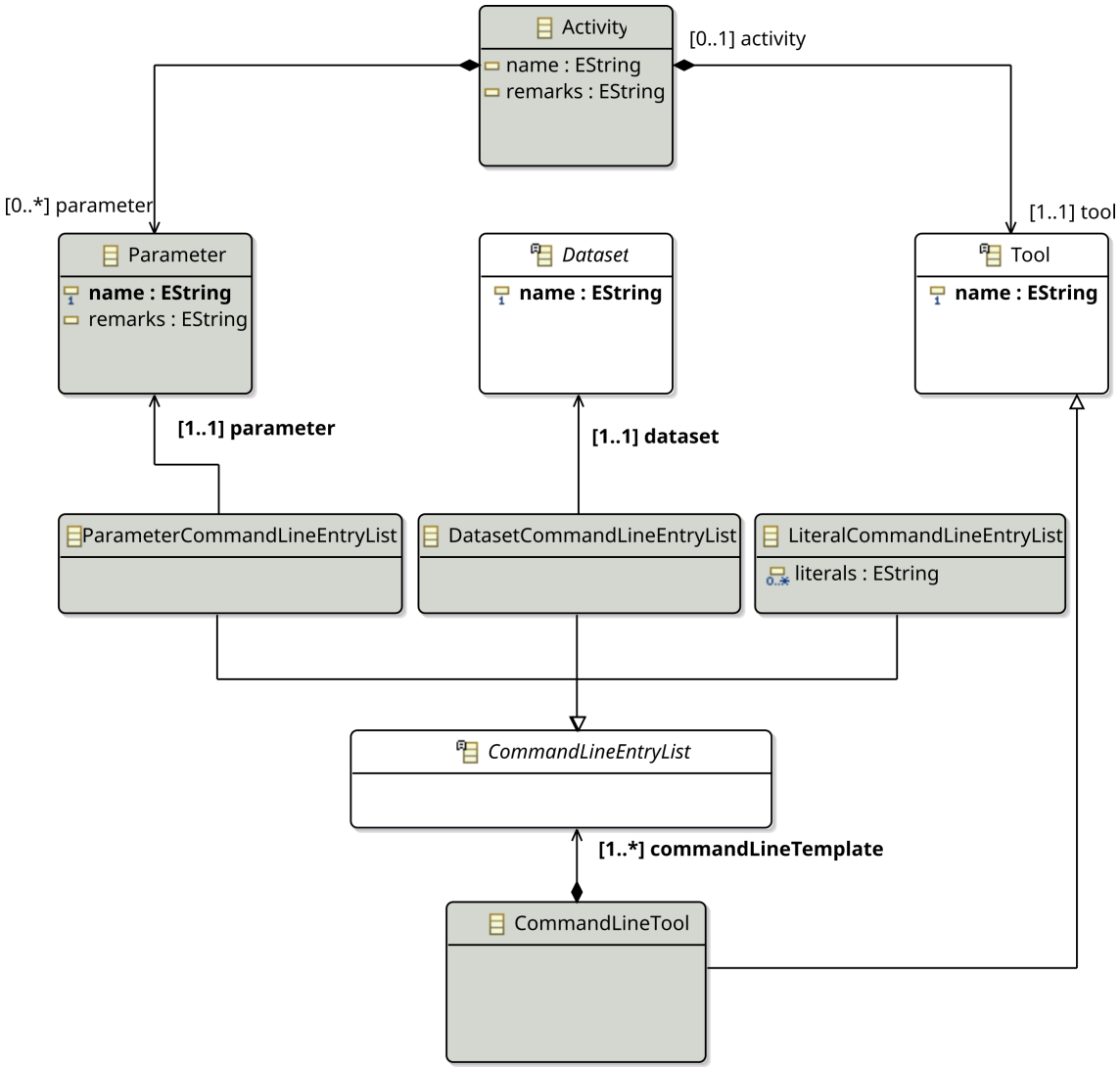
A invocação de uma ferramenta de análise é realizada por meio de uma chamada de função que recebe uma lista de *strings* contendo o nome (ou caminho) do arquivo executável desejado, bem como uma lista de argumentos a serem passados ao processo criado. Parâmetros de execução e conjuntos de dados são incluídos como uma sub-lista estruturada da lista de argumentos utilizada para a invocação da ferramenta de análise. Os valores nestas sub-listas são obtidos, em geral, como *strings* representando valores literais de um parâmetro de execução, ou representando caminhos dos arquivos agrupados em um conjunto de dados.

O metamodelo AADMM permite definir como a ferramenta de análise deve ser invocada a partir dos conjuntos de dados de entrada/saída e parâmetros de execução declarados. Essa invocação é definida utilizando instâncias concretas da metaclasse abstrata `CommandLineEntryList`, referenciadas na lista `commandLineTemplate` de `CommandLineTool`. A Figura 28 representa a metaclasse `CommandLineEntryList` e suas subclasses.

Uma instância de `CommandLineEntryList` representa uma sub-lista ordenada de *strings* a ser utilizada na chamada à ferramenta de análise. Essa sub-lista é incluída como parte da invocação da ferramenta de análise respeitando sua posição relativa às sub-listas criadas a partir das outras instâncias de `CommandLineEntryList` presentes no modelo da atividade de análise.

A metaclasse abstrata `CommandLineEntryList` é especializada pelas metaclasses concretas `ParameterCommandLineEntryList`, `DatasetCommandLineEntryList` e `LiteralCommandLineEntryList`. `ParameterCommandLineEntryList` representa uma sub-lista obtida a partir dos valores informados para um parâmetro de execução. `DatasetCommandLineEntryList` representa uma sub-lista obtida a partir dos caminhos dos arquivos informados para um conjunto de dados. Por sua vez, `LiteralCommandLineEntryList` permite que uma sub-lista de *strings* seja incluída de maneira literal na lista utilizada para a invocação da ferramenta de análise.

Figura 28 – Metaclasses relacionadas à definição da invocação da ferramenta de análise.



Fonte: Autoria própria. Metaclasses abstratas são representadas como retângulos com fundo branco e com ícone apresentando a letra A como decoração. Metaclasses concretas são representadas como retângulos com fundo cinza.

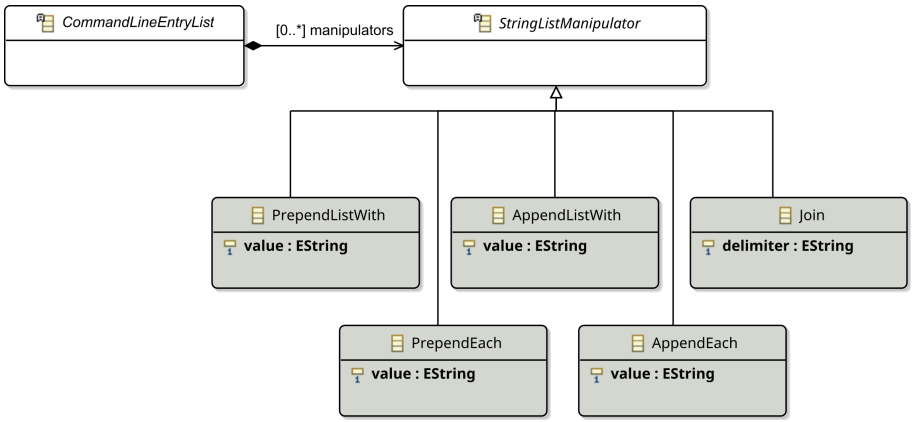
Os valores presentes nas sub-listas derivadas de um parâmetro de execução ou conjunto de dados podem ser modificados e formatados previamente à inclusão dos mesmos na lista utilizada para a execução da atividade de análise. Por exemplo, a ferramenta de análise invocada pode exigir que todos os caminhos de arquivos de um dado conjunto de dados sejam combinados em uma única *string* delimitada por um caractere especial. Neste sentido, é possível utilizar instâncias da metaclasse abstrata `StringListManipulator` de modo a definir regras de transformação a serem executadas sobre essas sub-listas.

Cada instância de `CommandLineEntryList` possui uma lista de instâncias de `StringListManipulator`, as quais definem como a lista de valores referenciada em

um `CommandLineEntryList` deve ser manipulada de modo a obter uma das sub-listas utilizadas como argumentos da chamada à ferramenta de análise.

A Figura 29 apresenta a metaclasse `StringListManipulator` e suas subclasses. A subclasse `PrependListWith` adiciona uma lista de *strings* ao início da lista recebida. A subclasse `AppendListWith` adiciona uma lista de *strings* ao final da lista recebida. A subclasse `PrependEach` prefixa uma string definida a cada string da lista recebida. A subclasse `AppendEach` posfixa uma string definida a cada string da lista recebida. Por fim, a subclasse `Join` une todos os itens da lista recebida, separando-os por um delimitador escolhido pelo usuário, e produz uma lista com um único item.

Figura 29 – Metaclasses relacionadas à transformação de valores para a definição da invocação da ferramenta de análise.



Fonte: Autoria própria. Metaclasses abstratas são representadas como retângulos com fundo branco e com ícone apresentando a letra A como decoração. Metaclasses concretas são representadas como retângulos com fundo cinza.

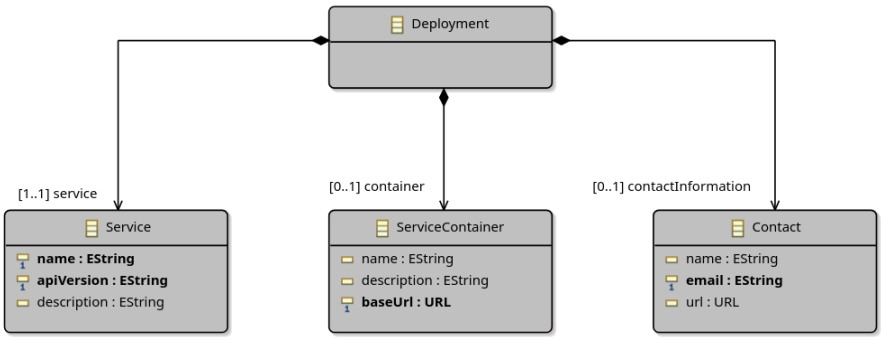
4.5 Service Deployment Description Metamodel

O usuário de um serviço necessita conhecer o URI do recurso base de um serviço de modo a poder utilizá-lo corretamente. Em abordagens de desenvolvimento baseadas em modelo, essas informações devem ser representadas como modelos passíveis de serem manipulados e utilizados durante o desenvolvimento e/ou a execução do sistema em desenvolvimento. Dessa maneira, definimos um novo metamodelo para a representação de informações relativas à implantação de um serviço de análise, chamado *Service Deployment Description Metamodel* (SDDMM).

A Figura 30 apresenta as metaclasses do metamodelo SDDMM como um diagrama de classes UML. A metaclasse `Deployment` representa uma implantação de um serviço de análise RAS. Essa implantação é definida pelo serviço implantado (metaclasse `Service`), o contêiner em que esse serviço é implantado (metaclasse

`ServiceContainer`). Adicionalmente, uma informação de contato do responsável por essa implantação pode ser definida (metaclasse `Contact`).

Figura 30 – Metamodelo SDDMM.



Fonte: Autoria própria.

A metaclasse `Service` representa um serviço RAS. Este serviço possui um nome (atributo `name`) e uma versão (atributo `apiVersion`). Adicionalmente, `Service` pode também definir uma descrição textual para esse serviço. A metaclasse `ServiceContainer` representa um contêiner/hospedeiro para um serviço a ser implantado. Este contêiner é responsável por definir o URI do recurso base do serviço (atributo `baseUrl`). Adicionalmente, esse contêiner possui um nome (atributo `name`) e uma descrição textual (atributo `description`).

Por fim, a metaclasse `Contact` representa uma informação de contato para o responsável pelo serviço de análise implantado. Este contato apresenta o nome do responsável (atributo `name`), um *email* para contato (atributo `email`) e um URI para a página *web* deste responsável.

O metamodelo SDDMM foi definido por meio do *framework Eclipse Modeling Framework* (EMF), provido pela *Eclipse Foundation*. Os artefatos que formalizam este metamodelo estão disponíveis via um repositório Git próprio[2].

4.6 Considerações finais

Durante esta etapa, definimos uma arquitetura de metamodelagem para o desenvolvimento de serviços RESTful por meio da adaptação de ferramentas de análise existentes. Em seguida, definimos um modelo de referência RAS para guiar a implementação de serviços de análise em bioinformática. Por fim, definimos dois metamodelos específicos de domínio, AADMM e SDDMM, para permitir a descrição de uma atividade de análise e da ferramenta que a suporta, e para a descrição da implantação de um serviço de análise, respectivamente.

[2] `https://purl.org/cawal/lssb/sddmm`.

A arquitetura de metamodelagem proposta utiliza um modelo AADM para a descrição abstrata dos aspectos mais importantes da atividade de análise. Essa arquitetura define também que este modelo possa ser interpretado de modo a prover o serviço de análise concreto segundo o modelo de referência RAS. Adicionalmente, um modelo SDDM pode ser definido para especificar a implantação desse serviço, e transformações podem ser aplicadas sobre ambos os modelos AADM e SDDM de modo a produzir clientes e descrições para o serviço de análise de maneira automática.

5 Geração de serviços RAS

Após a definição de um processo de desenvolvimento baseado em modelos e de uma arquitetura de metamodelagem para a obtenção de serviços de análise por meio da adaptação de ferramentas existentes, a próxima etapa deste trabalho consistiu no provimento de suporte ferramental para essa obtenção. Neste sentido, definimos inicialmente uma linguagem chamada ActDL para prover uma sintaxe textual concreta para modelos AADM, de modo a permitir, se necessário, o desenvolvimento desses modelos por meio de editores de texto simples. Em seguida, desenvolvemos um *framework* chamado Activity-REST para facilitar a adaptação de ferramentas de análise como um serviço RESTful segundo o processo de desenvolvimento definido anteriormente. Este *framework* implementa um adaptador que expõe um serviço RESTful para o uso de uma ferramenta de análise em bioinformática, configurado por meio de um modelo AADM interpretado em tempo de execução.

O restante desse capítulo está estruturado da seguinte maneira: a Seção apresenta uma visão geral da linguagem ActDL; a Seção 5.2 apresenta uma visão geral do *framework* Activity-REST; por fim, a Seção 5.3 apresenta um estudo de caso no desenvolvimento e uso de um serviço RESTful que encapsula uma ferramenta de análise existente.

5.1 Uma sintaxe textual para o metamodelo AADMM

O metamodelo AADMM define uma taxonomia dos elementos utilizados para a descrição de atividades de análise, bem como as relações entre esses elementos. Porém, este metamodelo não define como estes modelos devem ser representados concretamente. Deste modo, o metamodelo AADMM provê a sintaxe abstrata de uma linguagem específica de domínio (*Domain Specific Language* ou DSL), mas não a sintaxe concreta dessa linguagem.

Uma sintaxe concreta adequada deve ser provida para a definição de modelos AADM de maneira conveniente. Neste sentido, duas diferentes abordagens podem ser utilizadas para prover esta sintaxe: definir uma sintaxe concreta gráfica ou definir uma sintaxe concreta textual. Uma sintaxe concreta gráfica mapeia os elementos definidos por um metamodelo a um conjunto de elementos de uma notação gráfica. Em geral, estes elementos notacionais tomam a forma de um grafo formado por um conjunto de desenhos geométricos, arestas e rótulos textuais. O formalismo comumente utilizado em sintaxes gráficas pode facilitar a compreensão das relações entre elementos de um modelo qualquer. Porém, a escolha da definição de sintaxes gráficas exige o uso

de editores dedicados para a manipulação dos modelos criados. Esses editores irão interpretar o modelo e produzir uma representação visual adequada ao usuário, em geral na forma de um ou mais diagramas.

Uma sintaxe concreta textual mapeia os elementos definidos por um metamodelo a um conjunto de elementos textuais. Em geral, esses elementos textuais tomam a forma de palavras chaves, identificadores e caracteres delimitadores. Embora a representação textual de um modelo não tenha a intuitividade de uma representação visual, a linguagem definida pode utilizar palavras chaves e estruturas que descrevam o domínio de maneira clara. Adicionalmente, ainda que editores dedicados possam ser providos, modelos representados por meio de uma linguagem textual podem ser criados e manipulados por quaisquer programas de edição de texto existentes. Por essas facilidades, decidimos por utilizar inicialmente uma sintaxe concreta textual para a representação de modelos AADM.

Dado que o metamodelo AADMM foi definido a partir do meta-metamodelo Ecore e dado que Ecore permite a representação dos modelos definidos por meio de *XML Metadata Interchange* (XMI), esta linguagem poderia ser utilizada para a criação e representação de modelos AADM. Porém, a sintaxe XMI é pouco amigável ao desenvolvedor, exigindo conhecimento sobre a linguagem XML, sobre os detalhes estruturais do metamodelo AADMM e sobre a maneira que essas estruturas são mapeadas para elementos XMI. Dessa maneira, decidimos por prover uma sintaxe concreta textual específica de nosso domínio, chamada *Analysis Activity Description Language* (ActDL). Modelos AADM representados usando ActDL são denominados como modelos ActDL neste trabalho.

5.1.1 Visão geral da linguagem ActDL

A Listagem 2 apresenta uma visão geral da estrutura de um modelo ActDL. Um modelo ActDL, assim como um modelo AADM, define um elemento `Activity` como seu elemento raiz. Atributos da atividade de análise, como o nome dessa atividade e uma descrição textual, são definidos logo no início do modelo ActDL. Em seguida, quatro seções são definidas: a seção `on {...}` representa a coleção `Activity::inputDatasets` e agrega as declarações de conjuntos de dados de entrada; a seção `with {...}` representa a coleção `Activity::parameters` e agrega as declarações de parâmetros de execução; a seção `produces {...}` representa a coleção `Activity::outputDatasets` e agrega as declarações de conjuntos de dados de saída; por fim, a seção `using {...}` representa a propriedade `Activity::tool` e inclui a declaração da ferramenta de análise utilizada. A escolha dessas palavras-chaves foi realizada de modo a facilitar a leitura da especificação da atividade de análise na língua inglesa, de modo que, genericamente, o documento pode ser lido como *"The*

ACTIVITY my-activity is performed ON some input datasets WITH some parameters and PRODUCES some output datasets USING a given analysis tool.").

Listagem 2 – Visão geral de um modelo ActDL.

```
activity my-activity {
  remarks 'Some textual description';

  on {
    // input dataset declarations
  }
  with {
    // parameter declarations
  }
  produces {
     // output dataset declarations
  }
  using // ... analysis tool declaration
}
```

Fonte: Autoria própria.

A Listagem 3 apresenta um exemplo da estrutura de declarações de conjuntos de dados e parâmetros de execução. Declarações de parâmetros de execução, conjuntos de dados de entrada e conjuntos de dados de saída seguem um padrão semelhante ao utilizado na linguagem *OCLinEcore* [94]. Na seção adequada, a declaração inicia-se com a palavra-chave `parameter` ou `dataset` seguida de um identificador único. Em seguida, o tipo do parâmetro ou conjunto de dados é indicado, após o sinal de dois pontos. Neste sentido, o tipo de um parâmetro é representado pela enumeração em `Parameter::parameterType`, enquanto o tipo do conjunto de dados é representado pela propriedade `Dataset::mimetype`. A cardinalidade do parâmetro é, então, apresentada entre colchetes. Por exemplo, `[1,1]` para uma cardinalidade mínima e máxima de 1 elemento, e `[1,-1]` para uma cardinalidade mínima de 1 elemento e uma cardinalidade máxima indefinida. Parâmetros podem ainda definir um valor padrão utilizando o sinal de igualdade e declarando uma lista com os valores esperados. Por fim, atributos opcionais, como `Dataset::remark` e a declaração de restrições aplicáveis ao elemento, são incluídos internamente ao escopo demarcado por chaves.

A Listagem 4 apresenta uma visão geral de uma declaração de ferramenta de linha de comando utilizada para executar uma atividade de análise (metaclasse `CommandLineTool`). Esta declaração é iniciada com a palavra-chave `executable` seguida do nome da ferramenta e chaves. Três seções são utilizadas para a definição da ferramenta: a seção opcional `redirecting {...}` é utilizada para redirecionar entradas e saídas padrões para conjuntos de dados definidos anteriormente (atributos `standardInputStream`, `standardErrorStream` e `standardOutputStream`); a seção `commandLineTemplate [...]` é utilizada para definir a representação dos argumentos de linha de comando passados para a ferramenta (apresentado em mais detalhes a seguir); por fim, a seção opcional `returns {...}` é utilizada para definir

Listagem 3 – Visão geral da estrutura das declarações de conjuntos de dados parâmetros de execução.

```
...
  on {
    dataset input : 'text/csv' [1,1] {
      remark 'Some CSV file.';
    };
  }
  with {
    parameter encoding : STRING [1,1] = ['UTF-8'];
    parameter fields : INTEGER [1,-1] {
      remark 'The fields (columns) that will be in the output';
    };
  }
  produces {
    dataset output : 'text/csv' [1,1] {
      remark 'The produced file';
    };
  }
...
```

Fonte: Autoria própria.

instâncias da metaclasse `ExitCode`, associando-as à coleção `CommandLineApplication::exitCodes`. Dessa maneira, essa seção permite declarar os códigos de retorno da ferramenta e associá-los ao sucesso ou falha de execução da atividade de análise, bem como a mensagens de relatório aos códigos retornados.

Listagem 4 – Visão geral da estrutura de uma declaração de ferramenta de análise de linha de comando.

```
...
  using executable cut {
    redirecting {
      stdout to output;
    }
    commandLineTemplate [
      ...
    ]
    returns {
      0 if SUCCEEDED;
      1 if FAILED 'Some problem occurred.';
    }
  }
...
```

Fonte: Autoria própria.

As diferentes seções de um modelo ActDL são delimitadas por colchetes (`[` e `]`) ou chaves (`{` e `}`). Esses marcadores são utilizados de modo a definir se a ordem das declarações contidas no escopo de uma dada seção tem valor semântico ou não no metamodelo AADMM. Escopos delimitados por colchetes definem listas cuja ordem dos elementos é significativa. Por exemplo, os items da lista `commandLineTemplate` da declaração de uma ferramenta de linha de comando precisam ser informados em uma determinada ordem, pois os argumentos definidos devem ser passados corretamente à ferramenta durante sua invocação. Por sua vez, a ordem das declarações em

escopos delimitados por chaves não têm qualquer valor semântico. Neste sentido, essas declarações podem ter um posicionamento fixo ou definir elementos de uma lista na qual ordenação não é importante. Por exemplo, as seções principais da declaração de uma atividade de análise possuem um posicionamento relativo fixo, enquanto as definições de parâmetros e conjuntos de dados incluídas nessas seções definem elementos em uma lista cuja ordenação não é importante.

A sintaxe da linguagem ActDL foi definida utilizando-se do suporte provido pelo *framework* Xtext. O Apêndice C apresenta a gramática da linguagem ActDL por meio da forma de Backus-Naur estendida. Adicionalmente, um editor dedicado para a manipulação de modelos ActDL, utilizável em conjunto com o ambiente de desenvolvimento Eclipse, está disponível no repositório da linguagem[1].

5.1.2 Exemplos

Esta seção apresenta dois exemplos da definição de um modelo AADM por meio de um modelo ActDL para a representação de uma atividade suportada por ferramenta em linha de comando.

Ferramenta Clustal-Omega

A ferramenta Clustal-Omega [95] permite a execução da atividade de análise "alinhamento múltiplo de sequências proteicas". O executável dessa ferramenta (`clustalo`) recebe um conjunto de dados de entrada consistindo de um arquivo de texto contendo um conjunto de sequências proteicas. Ao final do processamento, dois conjuntos de dados de interesse são produzidos: um arquivo contendo o alinhamento obtido a partir das sequências de entrada; e dados de diagnóstico da execução da atividade de análise e impressos na saída padrão do processo. A Listagem 5 apresenta um exemplo de invocação do executável `clustalo`. Neste exemplo, `/home/me/inputFile.fasta` representa o arquivo de entrada e `/home/me/outputFile` representa o arquivo de saída. O argumento `-v` é uma *flag* para que os dados de diagnóstico sejam impressos na saída padrão.

Listagem 5 – Exemplo de invocação da ferramenta *Clustal-Omega*.

```
1 clustalo -i /home/me/inputFile.fasta -o /home/me/outputFile -v
```

Fonte: Autoria própria.

A Listagem 6 apresenta um modelo ActDL para a execução dessa atividade de análise. A linha 1 representa a descrição da atividade de análise e seu identificador. As linhas 2 e 3 apresentam um curto comentário textual sobre essa atividade. As linhas

[1] `https://purl.org/cawal/lssb/actdl`

de 4 a 8 apresentam a definição dos conjuntos de dados de entrada. Nesse sentido, a atividade de análise executa sobre um conjunto de dados chamado `sequences-file`. Esse conjunto de dados consiste de apenas um arquivo do tipo `text/plain`. Um comentário é apresentado sobre esse conjunto de dados na linha 6.

Listagem 6 – Descrição ActDL para a atividade "alinhamento múltiplo de sequências proteicas" utilizando a ferramenta Clustal-Omega.

```
1  activity multiple-alignment-f-protein {
2    remark '''This activity performs the multiple alignment of protein
3    sequences''';
4    on {
5      dataset sequences-file : 'text/plain' [1,1]{
6        remark 'Multiple sequence input file.';
7      };
8    }
9    produces {
10     dataset aligned-sequences: 'text/plain' [1,1] {
11       remark 'Multiple sequence alignment output file.';
12     };
13     dataset verbose-output: 'text/plain' [1,1];
14   }
15   using executable clustalo {
16     redirecting {
17       stdout to verbose-output;
18     }
19     commandLineTemplate [
20       dataset sequences-file
21         | PrependListWith '-i'
22       ,
23       dataset aligned-sequences
24         | PrependListWith '-o'
25       ,
26       literals ['-v']
27     ]
28   }
29 }
```

Fonte: Autoria própria.

As linhas 9 a 14 apresentam a definição dos conjuntos de dados de saída da atividade de análise. Neste sentido, a atividade de análise produz dos conjuntos de dados: `aligned-sequences` e `verbose-output`. Ambos conjuntos de dados são formados por um único arquivo do tipo `text/plain`. Um comentário textual sobre o conjunto de dados `aligned-sequences` é apresentado na linha 11.

As linhas 15 a 28 apresentam a declaração da invocação do executável `clustalo`. As linhas 16 a 18 definem que o conjunto de dados de saída `verbose-output` é produzido a partir da saída padrão do processo executado. As linhas 19 a 27 definem como os argumentos passados à ferramenta devem ser construídos. Neste sentido, o caminho do arquivo informado para o conjunto de dados `sequences-file` deve ser precedido de um argumento com a *string* `-i`. De maneira semelhante, o caminho do arquivo a ser produzido no conjunto de dados `sequences-file` deve ser precedido de um argumento com a *string* `-o`. A *string* `-v` é passada como último argumento da invocação.

Por fim, nenhum código de término é informado. Neste sentido, é subentendido que a atividade de análise terá sido executada com sucesso caso o executável `clustalo` tenha retornado `0` (zero) como código de saída e que uma falha de execução terá ocorrido se um código diferente tenha sido retornado. Esse entendimento deriva das convenções ISO [96].

Ferramenta cut

A ferramenta `cut`, provida por sistemas *Unix-like*, implementa a funcionalidade filtrar colunas de interesse de uma tabela. Esta tabela deve ser representada em texto plano e delimitar os campos de cada linha por meio de um caractere especial. A ferramenta `cut` recebe como parâmetros o caractere delimitador (por exemplo, ponto-e-vírgula), bem como o número (ordinal) das colunas de interesse. A tabela original é recebida, por padrão, por meio da entrada padrão do processo e a tabela filtrada é impressa, por padrão, na saída padrão do processo. Como exemplo, a invocação de `cut` para separar uma tabela com campos separados por ponto-e-vírgula e filtrar apenas as colunas 1, 2, 5 e 8 é apresentada na Listagem 7. Aspas simples são utilizadas para delimitar cada parâmetro passado durante a invocação da ferramenta.

Listagem 7 – Exemplo de invocação da ferramenta `cut` em um sistema *Unix-like*.

```
1 cut '--delimiter=;' '-f1,2,5,8'
```

Fonte: Autoria própria.

A Listagem 8 apresenta um modelo ActDL para a seleção de colunas de um arquivo de um conjunto de dados de entrada por meio da ferramenta `cut` em um sistema *Unix-like*. A linha 1 apresenta a declaração da atividade e de seu identificador. As linhas 2 a 6 apresentam os conjuntos de dados de entrada utilizados na atividade. Nesse sentido, essa atividade é executada sobre um conjunto de dados chamado `input`, o qual possui apenas um arquivo contendo uma tabela cujos campos são separados por vírgula (arquivo CSV). As linhas 7 a 11 apresentam os parâmetros informados para a atividade, de modo que a atividade recebe como parâmetro uma lista de números inteiros chamada `fields`. Esta lista representa as posições das colunas de interesse no arquivo. As linhas 12 a 16 apresentam os conjuntos de dados se saída produzidos pela atividade. A atividade produz um conjunto de dados chamado `output`, o qual tem a forma de um único arquivo CSV.

As linhas 17 a 33 definem como a ferramenta `cut` deve ser invocada para executar a atividade modelada. A ferramenta `cut` lê um arquivo a partir da entrada padrão, processa esse arquivo e imprime na saída padrão apenas as colunas de interesse recebidas por parâmetro. As colunas de interesse devem ser indicadas por meio da *string* `-f` seguida por uma lista de números inteiros separados por vírgula

Listagem 8 – Exemplo da definição da atividade “separar colunas de um arquivo” por meio da ferramenta `cut`.

```
1  activity cut-file-columns {
2    on {
3      dataset input : 'text/csv' [1,1] {
4        remark 'A text file to cut (comma-separated).';
5      };
6    }
7    with {
8      parameter fields : INTEGER [1,-1] {
9        remark 'The fields (columns) that will be in the output';
10     };
11   }
12   produces {
13     dataset output : 'text/csv' [1,1] {
14       remark 'The produced file';
15     };
16   }
17   using executable cut {
18     redirecting {
19       stdin from input;
20       stdout to output;
21     }
22     commandLineTemplate [
23       literals ['--delimiter=,'],
24       parameter fields
25         | Join ','
26         | PrependListWith '-f'
27         | Join ''
28     ]
29     returns {
30       0 if SUCCEEDED;
31       1 if FAILED 'Some problem occurred. Check the selected fields.'
32     }
33   }
34 }
```

Fonte: Autoria própria.

(e.g., “`-f1,2,3,5,8`”). Adicionalmente, o delimitador utilizado (vírgula, neste caso), também deve ser indicado por um argumento de linha de comando (“`-delimiter=,`”).

A linha 17 indica a ferramenta a ser utilizada para a execução da atividade: `cut`. As linhas 17 a 21 indicam que os arquivos que formarem os conjuntos de dados `input` e `output` deverão ser redirecionados para a entrada padrão e criados a partir da saída padrão, respectivamente. As 29 a 33 indicam que `cut` retorna código de saída 0 quando executa com sucesso e 1 quando ocorre um término em falha.

As linhas 22 a 28 indicam como os argumentos da chamada à ferramenta devem ser construídos. O primeiro argumento é o valor literal “`--delimiter=,`”. O segundo argumento é criado a partir da lista de campos de interesse informada para o parâmetro `fields`. A construção desse argumento é um pouco mais complexa e uma série de operações é executada sobre a lista indicada. Suponhamos que uma lista contendo os inteiros `1, 2, 3 e 5` (representada por $[1, 2, 3, 5]$) , seja indicada para o parâmetro `fields`. Essa lista inicial é transformada, primeiramente, em uma lista de *strings* ($[''1'', ''2'', ''3'', ''5'']$). Em seguida, os itens dessa lista são unidos em

uma única *string* utilizando vírgula como separador, e uma lista é criada contendo apenas essa *string* ([″1, 2, 3, 5″]). Em seguida, é adicionado um item (a *string* `-f`) ao início da lista atualizada, obtendo uma nova lista com dois items ([″−f″,″ 1, 2, 3, 5″]). Por fim, os items da lista são unidos novamente, sem a utilização de um delimitador ([″−f1, 2, 3, 5″]). O único item nessa lista final deve ser, então, utilizado como o argumento final da invocação de `cut`.

5.2 Framework para o desenvolvimento de serviços RESTful

O *framework Activity-REST* facilita o desenvolvimento de serviços de análise que seguem o modelo de referência RAS definido anteriormente. Este *framework* foi definido de modo a interpretar um modelo `Ecore Activity Model` que presenta uma atividade de analise em tempo de execução. Por meio dessa interpretação, o *framework Activity-REST* provê os recursos RESTful necessários para a execução de instâncias de uma atividade de análise, bem como para adaptar e invocar uma dada ferramenta de análise subjacente. A Figura 31 apresenta uma visão estrutural dos componentes de um serviço adaptador Activity-REST.

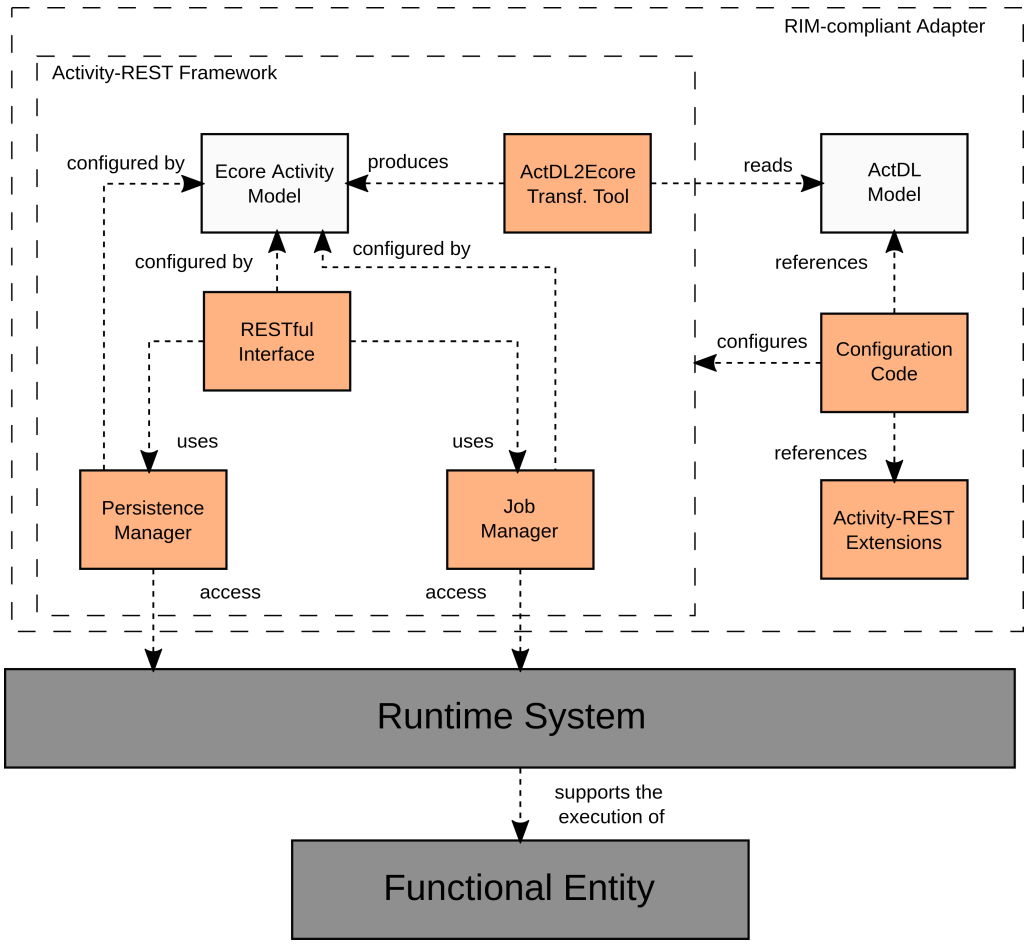
Assumimos que dois componentes já estão implantados na máquina hospedeira do serviço de análise: *Command-line Tool* e *Runtime System*. *Command-line Tool* representa a ferramenta de análise a ser adaptada pelo serviço, como as bibliotecas e outros arquivos utilizados por essa ferramenta durante sua execução (excluindo-se arquivos pertencentes aos conjuntos de dados informados por um usuário).

Runtime System representa os recursos providos pelo sistema operacional da máquina hospedeira em que o serviço será implantado. Tais recursos são utilizados durante a execução de uma instância de atividade de análise, bem como para o provimento de um serviço RESTful. Entre esses recursos estão o sistema de invocação e execução de processos e um servidor de aplicação no qual o adaptador RESTful será implantado.

Um serviço adaptador é composto pelo *framework* Activity-REST e por um conjunto de outros elementos utilizados para configurar e estender o *framework* de modo a prover suporte à execução da atividade de análise específica. O *framework* Activity-REST inclui um conjunto de elementos que proveem o suporte básico para os *endpoints* e as operações de um serviço adaptador. Nomeadamente, o *framework* Activity-REST inclui os componentes `Ecore Activity Model`, `ActDL2Ecore Transformation Tool`, `RESTful Interface`, `Persistence Manager` and `Job Manager`.

Cada instância de atividade de análise em bioinformática requer um ou mais conjuntos de dados de entrada e parâmetros próprios para a sua execução. Ao final dessa execução, a instância produz um ou mais conjuntos de dados de saída também

Figura 31 – Visão geral dos componentes de um serviço adaptador RAS para a execução de instâncias de atividades de análise em bioinformática.



Fonte: Autoria própria. Um retângulo cinza representa um elemento existente na máquina hospedeira. Um retângulo laranja representa um componente de código. Por fim, um retângulo branco representa uma instância uma instância (concreta ou em tempo de execução) de um modelo ActDL.

particulares a essa atividade de análise. Essas especificidades devem ser consideradas durante a execução do serviço de adaptação da atividade de análise, o qual deve expor os recursos RESTful definidos pelo modelo de interação proposto, bem como invocar corretamente a ferramenta de análise encapsulada. O componente `Ecore Activity Model` corresponde à representação em tempo de execução de um modelo ActDL. Este modelo é compartilhado pelos demais componentes do *framework* e interpretado de modo a prover o serviço de análise esperado. Quando um serviço Activity-REST é inicializado, o componente `ActDL2Ecore Transformation Tool` lê o modelo ActDL e produz o modelo `Ecore Activity Model` que é utilizado pelo *framework* Activity-REST.

O componente `RESTful Interface` implementa o suporte às interações definidas pelo modelo de referência RAS para serviços de análise. Este componente interpreta um modelo `Ecore Activity Model` de modo a expor os recursos do ser-

viço por meio de *endpoints* e representações adequadas. Um usuário de um serviço de análise interage apenas com o componente `RESTful Interface` durante a execução da atividade de análise.

O componente `RESTful Interface` utiliza os componentes `Persistence Manager` e `Job Manager` para acessar os recursos da máquina hospedeira indiretamente. O componente `Persistence Manager` é responsável por prover um conjunto de operações para a criação, recuperação, atualização e remoção de instâncias de atividade de análise persistidas no sistema de arquivos da máquina hospedeira. Adicionalmente, este componente provê classes utilizadas, por outros componentes, para a manipulação dessas instâncias de atividades de análise.

Finalmente, o componente `Job Manager` é responsável por prover um ambiente de execução para instâncias de atividades de análise e por invocar a ferramenta de análise adaptada. Adicionalmente, este componente monitora o processamento de cada instância de atividade de análise e provê informações sobre seu estado de execução atual e/ou final.

Além dos componentes do *framework* Activity-REST, um serviço adaptador também inclui um conjunto de elementos específicos para cada atividade de análise, nomeadamente um `modelo ActDL` e os componentes `Activity-REST Extensions` e `Configuration Code`. O componente `Activity-REST Extensions` consiste de um conjunto opcional de classes Java definidas pelo usuário. Estas classes são utilizadas para estender o comportamento do serviço desenvolvido por meio de *hotspots* providos pelos *frameworks* JEE e/ou Activity-REST, permitindo a definição de validadores de restrições impostas a conjuntos de dados/parâmetros de execução e novos *endpoints*. Finalmente, o componente `Configuration Code` consiste de código Java usado para inicializar o *framework* Activity-REST utilizando o modelo ActDL e o componente `Activity-REST Extensions`.

5.2.1 Componente Persistence Manager

O componente *Persistence Manager* representa concretamente uma instância de uma atividade de análise a partir de um modelo `Ecore Activity Model`, de modo a prover suporte à manipulação dessa instância pelos demais componentes do *framework*. A Figura 32 apresenta o modelo conceitual de uma atividade de análise implementado pelo componente *Persistence Manager* e seu relacionamento com o metamodelo AADM. Classes denotadas em cinza são definidas pelo componente *Persistence Manager*. Por sua vez, classes denotadas em branco são referências às metaclasses do metamodelo AADM. Uma vez que tanto o componente *Persistence Manager* quando o metamodelo AADM definem (meta)classes com nomes semelhantes, nesta seção utilizaremos o prefixo `AADM::` para nos referir às metaclasses

definidas pelo metamodelo AADM.

A classe `Activity` representa uma instância de atividade de análise genérica. Cada instância de `Activity` possui um identificador único e uma referência ao modelo `Ecore Activity Model` que define sua estrutura de parâmetros e conjuntos de dados. Adicionalmente, cada instância de `Activity` referencia instâncias das classes `ParameterMap` e `Dataset`.

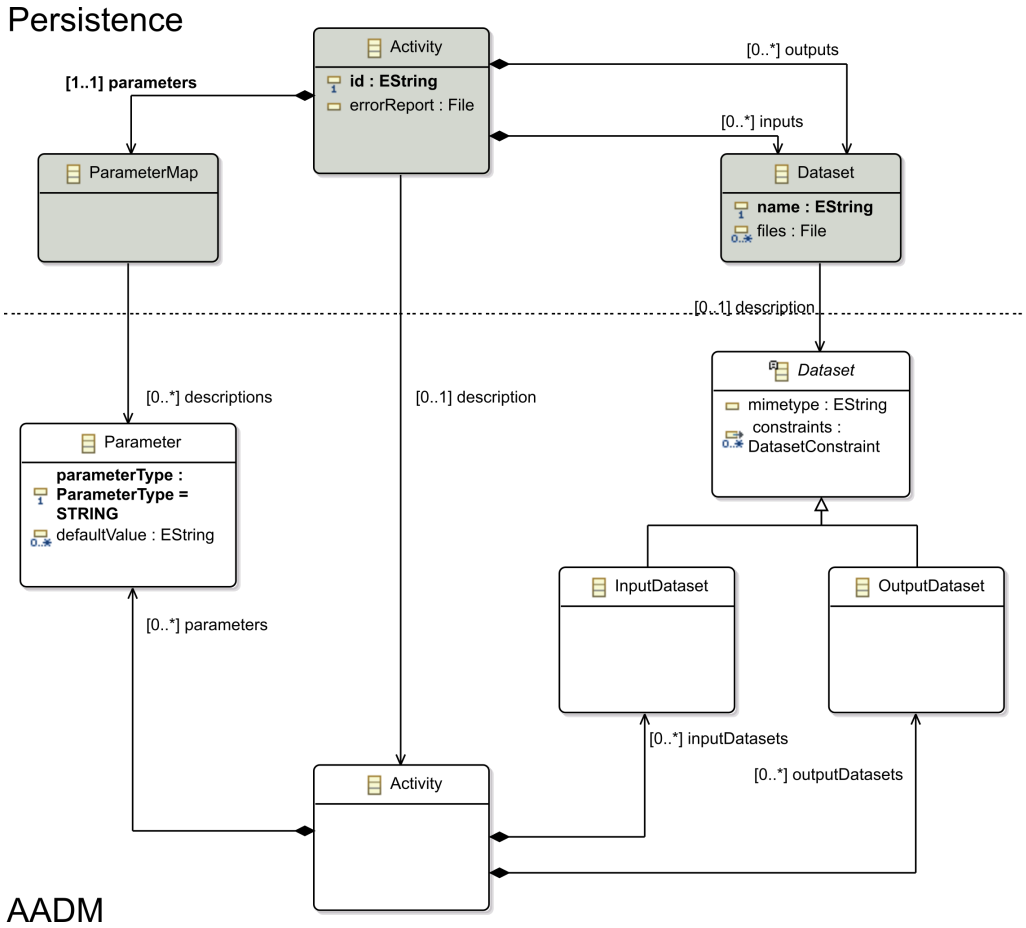
A classe `ParameterMap` representa a lista de valores providas pelo usuário do serviço para os parâmetros de execução. Uma instância de `ParameterMap` contém uma coleção de `AADM::Parameter`, o qual define os valores aceitos pelos parâmetros de execução da atividade de análise. Por sua vez, a classe `Dataset` representa uma lista de arquivos enviados pelo usuário do serviço para um dado conjunto de dados. Uma instância de `Dataset` é associada a um elemento `AADM::Dataset`, o qual define o tipo dos arquivos aceitos por esse conjunto de dados e a cardinalidade máxima da lista que contém esses arquivos. Dessa maneira, embora valores de parâmetros e referências a arquivos de um conjunto de dados sejam armazenados em uma lista, a cardinalidade dos elementos dessa lista é limitada em 1 para parâmetros e conjuntos de dados de valor único.

O componente *Persistence Manager* também é responsável por prover um conjunto de operações para persistir uma instância de uma atividade de análise. Essa responsabilidade é realizada por um conjunto de classes Java que implementam uma interface simples de Objeto de Acesso a Dados (*Data Access Object* ou DAO), denominada `AnalysisActivityRepository`, a qual provê um conjunto de métodos para criar, requisitar, atualizar e remover instâncias de atividades de análise de maneira persistente independentemente do meio em que essa instância será persistida.

Classes concretas podem implementar os métodos de `AnalysisActivity-Repository` de modo a persistir instâncias de atividades de análise em diferentes meios, como arquivos estruturados ou bancos de dados relacionais. Uma vez que as ferramentas adaptadas fazem uso de conjuntos de dados armazenados normalmente em arquivos, inicialmente implementamos a interface `AnalysisActivityRepository` definindo a classe `FileSystemAnalysisActivityRepository`. Esta classe recebe um diretório raiz, o qual representa a coleção de instâncias de atividades de análise, e persiste cada instância de `Activity` como um subdiretório estruturado do diretório raiz. O nome do subdiretório que representa uma instância de uma atividade de análise é obtido diretamente do identificador único da instância de `Activity` (atributo `id`).

A Figura 33 apresenta uma visão geral da estrutura de diretórios utilizada para o armazenamento de uma coleção de instâncias de atividades de análise por meio de `FileSystemAnalysisActivityRepository`. O arquivo `parameters.json`

Figura 32 – Modelo conceitual de uma atividade de análise e sua relação com um modelo ActDL.



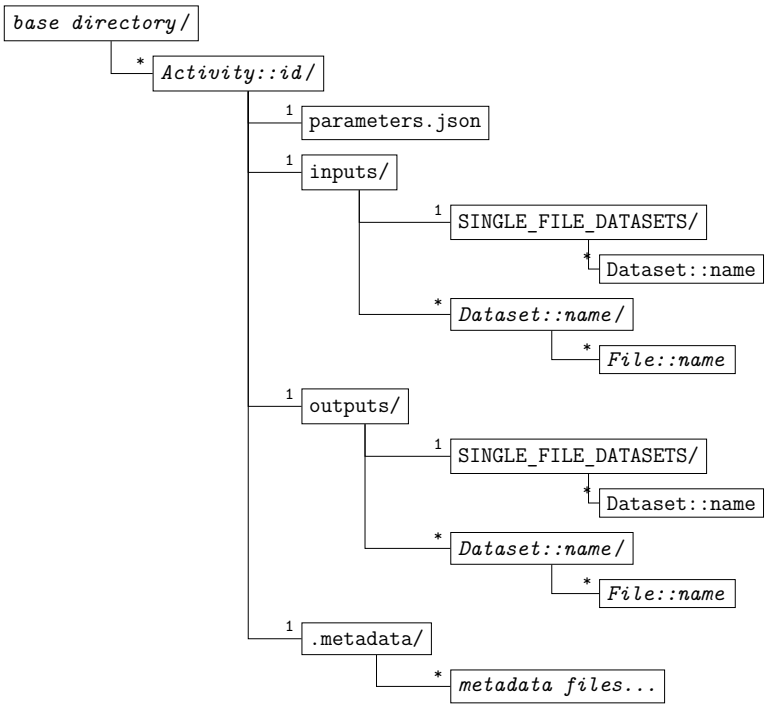
Fonte: Autoria própria.

armazena valores dos parâmetros de execução de uma atividade de análise em uma representação JSON. Os subdiretórios `inputs` e `outputs` armazenam os arquivos pertencentes a conjuntos de dados de entrada e saída, respectivamente. Os diretórios `inputs` e `outputs` são organizados da seguinte maneira: cada conjunto de dados de cardinalidade unitária é representado como um arquivo no subdiretório `SINGLE_FILE_DATASETS`; por sua vez, cada conjunto de dados de cardinalidade múltipla possuem um subdiretório nomeado com o mesmo nome do conjunto de dados. Por fim, o diretório `.metadata` é utilizado para armazenar metadados de interesse para instâncias de `FileSystemAnalysisActivityRepository` que não são representadas completamente pela estrutura do sistema de arquivos, por exemplo, a ordem dos arquivos na coleção `files` de uma instância de `Dataset`.

5.2.2 Componente Job Manager

O componente *Job Manager* é responsável por receber as informações necessárias para a execução de uma instância de atividade de análise e invocar a ferramenta

Figura 33 – Estrutura de diretórios utilizada para o armazenamento das atividades de análise.



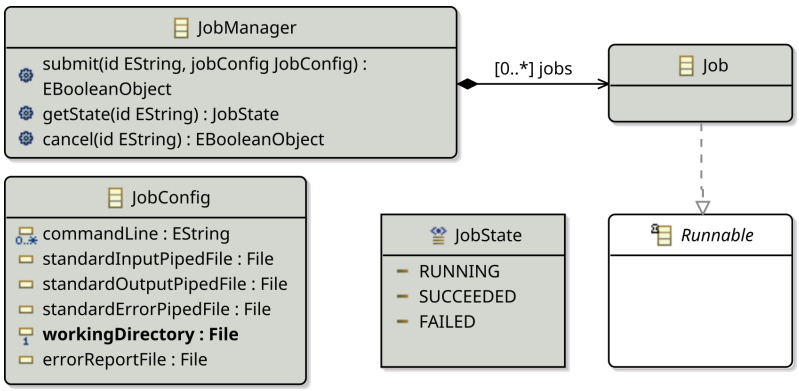
Fonte: Autoria própria.Um retângulo nomeado representa um ou mais arquivos ou diretórios no sistema de arquivos da máquina hospedeira. A cardinalidade de um elemento é apresentada à esquerda deste elemento. Esta cardinalidade é representada como `1`, representando elementos de cardinalidade unitária, e `*`, representado elementos com cardinalidade múltipla não definida. O rótulo em um retângulo representa o nome do arquivo ou diretório. O uso de texto monoespaçado simples representa esse nome literalmente, enquanto texto monoespaçado itálico representa que esse nome é obtido a partir do atributo referenciado em uma instância das classes definidas pelo componente *Persistence*.

de análise adaptada, monitorando sua execução. Este componente recebe instâncias da classe `JobConfig`, a qual representa o conjunto de informações necessárias para executar uma dada ferramenta de análise, e invoca esta ferramenta por meio de uma chamada ao sistema operacional. Neste sentido, a representação de uma instância de atividade de análise provida pelo componente *Persistence Manager* (classe `Activity`) e a representação de um trabalho a ser executado pelo componente *Job Manager* são independentes, possibilitando a evolução de ambos os componentes separadamente.

A Figura 34 apresenta as principais classes contidas no componente *Job Manager*. A classe `JobConfig` representa o conjunto de informações necessárias para a execução de uma instância de atividade de análise. `JobConfig` armazena a lista de *strings* que forma a linha de comando a ser utilizada para a invocação da ferramenta de análise, bem como o diretório de execução desejado e referências a arquivos utilizados para o redirecionamento das entradas e saídas, entre outras

informações. A classe `JobManager` representa o gerenciador de instâncias em execução. `JobManager` recebe instâncias de `JobConfig` e as utiliza para criar, internamente, uma instância de `Job`. A classe `Job` representa um adaptador para o processo executado, permitindo a observação desse processo enquanto executa em segundo plano. Métodos de `JobManager` possibilitam que o usuário do componente verifique o estado ou requisite o cancelamento de um trabalho.

Figura 34 – Classes principais do componente *Job Manager*.



Fonte: Autoria própria.

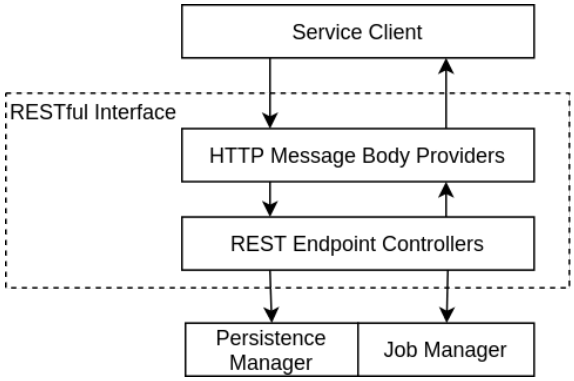
Durante a execução, uma instância de `Job` redireciona entradas e saídas do processo criado enquanto aguarda seu término. Após o fim da execução da atividade de análise, a instância de `Job` sinaliza ao componente *Job Manager* o estado final do processamento (sucesso ou falha), de acordo com o código de saída retornado pelo processo invocado. Dessa maneira, essa informação pode ser retornada a um usuário do componente que requisite o estado da execução da atividade de análise.

5.2.3 Componente RESTful Interface

O componente *RESTful Interface* é responsável por prover suporte ao modelo de interação RESTful definido. *RESTful Interface* possui dois subcomponentes principais, `REST Endpoints` e `HTTP Message Body Providers`. A Figura 35 apresenta uma visão geral do componente *RESTful Interface*, e seus subcomponentes, em relação às suas interações com o cliente e os demais componentes do serviço.

O componente `REST Endpoints` recebe uma descrição da atividade de análise como um modelo `Ecore Activity Model` e expõe ao usuário do serviço os recursos adequados de acordo com o modelo recebido, mapeando URIs aos recursos mantidos pelo serviço, como instâncias de atividades de análise, parâmetros de execução e conjuntos de dados. Esse componente também recebe um conjunto de requisições dos usuários e processa estas requisições, produzindo um conjunto de informações a serem representadas no cabeçalho e no corpo da resposta HTTP. `REST Endpoints`

Figura 35 – Subcomponentes de *RESTful Interface* e serviço RESTful.



Fonte: Autoria própria.

utiliza os componentes *Persistence Manager* e *Job Manager* durante o processamento de uma requisição do usuário.

O componente `HTTP Message Body Providers` é responsável por prover mecanismos para a negociação de formato de representação entre cliente e o serviço de análise, provendo suporte aos formatos de representação JSON e XML definidos pelo modelo de referência RAS. Adicionalmente, o componente `HTTP Message Body Providers` pode ser estendido de modo a prover suporte a novos formatos de representação.

Durante a interação com o serviço, o cliente envia uma requisição ao serviço RAS. O corpo dessa requisição é recebido pelo componente `HTTP Message Body Providers` que transforma o conteúdo dessa mensagem em uma ou mais instâncias de classes Java esperadas pelo componente `REST Endpoints`. O componente `REST Endpoints` recebe essas instâncias e processa a requisição do usuário, retornando as instâncias atualizadas ao final desse processamento. O componente `HTTP Message Body Providers` recebe as instâncias atualizadas e transforma-as em uma resposta HTTP adequada no formato de representação desejado pelo cliente do serviço.

5.2.4 Concretização do framework Activity-REST

O *framework* Activity-REST foi concretizado por meio da linguagem de programação Java. Neste sentido, as classes e interfaces para a representação de uma atividade de análise no componente *Persistence* foram definidas por meio do *Eclipse Modeling Framework* (EMF) [75]. A implementação do componente *RESTful Interface* baseia-se no *framework Java API for RESTful Services* (JAX-RS), o qual provê um conjunto de recursos para a execução de serviços RESTful em um servidor de aplicação Java EE.

As classes contidas no componente `REST Endpoints` recebem a descrição da

atividade de análise como um modelo `Ecore Activity Model` e expõem recursos adequados ao usuário de acordo com o modelo recebido. Operações sobre estes recursos são mapeadas aos métodos dessas classes por meio de um conjunto de anotações providas pelo JAX-RS.

As classes contidas em `REST Endpoints` e `HTTP Message Body Providers` devem ser registradas no *framework* JAX-RS de modo a serem utilizadas para prover um serviço RESTful em um servidor de aplicação com suporte à tecnologia Java EE. Dessa maneira, o desenvolvedor precisa prover uma classe adicional, denominada `Application Config`, contendo um método que registra as classes presentes nesses componentes e as classes presentes nos demais componentes do *framework* desenvolvido.

Por meio de `Application Config`, o desenvolvedor também pode estender o serviço provido, registrando outras classes que não são parte do *framework*. Por exemplo, o desenvolvedor pode registrar novos `HTTP Message Body Providers` de modo a prover suporte a mais formatos de representação dos recursos do serviço, ou novos `REST Endpoints` de modo a prover suporte a recursos adicionais para o serviço de análise. Embora a implementação de `ApplicationConfig` seja simples, provemos uma classe abstrata que pode ser estendida de modo a facilitar o registro dos componentes necessários.

5.3 Estudo de caso

Um exemplo do desenvolvimento de um serviço de análise por meio do *framework Activity-REST* é apresentado a seguir. Nesse exemplo, definimos um serviço para adaptar a ferramenta `one-color-uarray-fold-change-diff-analysis.R`.

5.3.1 Visão geral

A ferramenta `one-color-uarray-fold-change-diff-analysis.R` consiste de um *script* R para suporte à execução da atividade "análise diferencial de dados de expressão gênica obtidos por meio de *microarray one-color*". Esta ferramenta recebe dois arquivos (no formato *tab-separated values*) como conjunto de entrada. Cada arquivo representa uma condição experimental e contém dados normalizados de expressão gênica obtidos por meio de análise de microarray *one color*. A primeira coluna de cada arquivo arquivo deve apresentar uma lista de identificadores de genes. As colunas seguintes apresentam a expressão de cada um dos genes da primeira coluna em diferentes execuções da análise de microarray para aquela mesma condição experimental. Adicionalmente, a ferramenta recebe o parâmetro `--cutoff`. Esse parâmetro é utilizado como um limiar para a identificação de genes diferencialmente

expressos. Nesse sentido, um gene é considerado diferencialmente expresso se os valores de expressão médios nas duas condições experimentais for maior que esse limiar. Após a execução, esta ferramenta produz um conjunto de dados de saída contendo um único arquivo no formato *tab-separated values*. Este arquivo apresenta os genes considerados diferencialmente expressos, bem como a diferença da expressão encontrada nas duas condições experimentais. A Listagem 9 apresenta a invocação desta ferramenta conforme realizada via linha de comando.

Listagem 9 – Exemplo de invocação da ferramenta `one-color-uarray-fold-change-diff-analysis.R`.

```
1 one-color-uarray-fold-change-diff-analysis.R --cutoff <cutoff>
    <input-file-1> <input-file-2> <output-file>
```

Fonte: Autoria própria.

5.3.2 Modelo ActDL

A Listagem 10 apresenta o modelo ActDL para a atividade de análise provida por `one-color-uarray-fold-change-diff-analysis.R`. A linha 1 declara a atividade de análise e define um identificador para essa atividade. As linhas de 2 a 6 definem um comentário textual para a atividade de análise. As linhas de 7 a 14 definem os conjuntos de dados de entrada da atividade de análise. Neste sentido, a atividade de análise executa sobre dois conjuntos de dados contendo, cada um, um único arquivo do tipo *tab-separated values*. Comentários textuais também são apresentados para esses conjuntos de dados.

As linhas de 15 a 19 definem os parâmetros da atividade de análise. Neste sentido, o parâmetro `cutoff` é definido como sendo do tipo `REAL` e possuindo cardinalidade unitária e valor padrão `2`. Novamente, um comentário textual é apresentado para o significado do parâmetro. As linhas de 20 a 22 definem o único conjunto de dados de saída da atividade de análise, chamado `output`, o qual toma a forma de um único arquivo de campos separados por tabulação.

Por fim, as linhas 23 a 32 definem a invocação da ferramenta `one-color-uarray-fold-change-diff-analysis.R`. A linha 23 define o identificador da ferramenta. As linhas 24 a 31 definem os argumentos passados durante a invocação. Neste sentido, as linhas 25 e 26 definem que o primeiro argumento utilizado é a *string* "`--cutoff`", seguida do valor do parâmetro `cutoff` como próximo argumento. As linhas 27 a 29 definem que os demais argumentos são obtidos a partir do caminho onde residem os arquivos informados para os dois conjuntos de dados de entrada, seguido do caminho em que deverá residir o arquivo produzido para o conjunto de dados de saída `output`.

Listagem 10 – Descrição ActDL para a atividade "análise diferencial de *microarray one-color*" utilizando a ferramenta `one-color-uarray-fold-change-diff-analysis.R`.

```
activity one-color-microarray-fold-change {
  remark '''This script receives two files containing
    one-color microarray data and identifies diffential
    expressed genes by fold-change analysis. Each file
    must contains samples collected in a different
    experimental condition.''';
  on {
    dataset first-exp-condition : "text/tsv" [1,1] {
      remark 'Input file for the first comparing condition';
    };
    dataset second-exp-condition : "text/tsv" [1,1]{
      remark 'Input file for the second comparing condition';
    };
  }
  with {
    parameter cutoff : REAL [1,1] = ["2"]{
      remark 'Cutoff (log2-based) for filtering the fold change.';
    };
  }
  produces {
    dataset output : "text/tsv" [1,1];
  }
  using executable 'one-color-uarray-fold-change-diff-analysis.R' {
    commandLineTemplate [
      parameter cutoff
      | PrependListWith '--cutoff',
      dataset first-exp-condition,
      dataset second-exp-condition,
      dataset output
    ]
  }
}
```

Fonte: Autoria própria.

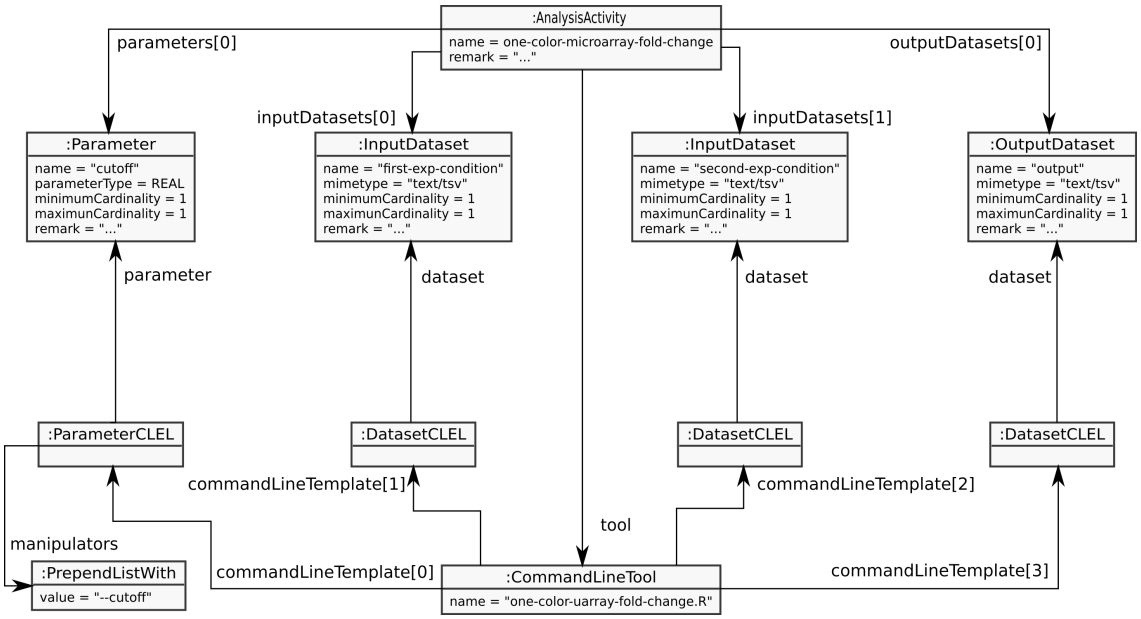
5.3.3 Interpretação do modelo Ecore Activity Model

A Figura 36 apresenta o resultado da transformação do modelo ActDL em um modelo `Ecore Activity Model`. Esse modelo é apresentado por meio de um diagrama de objetos UML. As metaclasses `DatasetCommandLineEntryList` e `ParameterCommandLineEntryList` possuem seus nomes abreviados para melhorar a visualização do modelo.

O modelo `Ecore Activity Model` é então utilizado para configurar o *framework* Activity-REST, de modo a criar um serviço *web* adaptador por meio deste *framework*. O desenvolvedor utiliza um projeto *Java* modelo para facilitar essa criação. Neste sentido, o usuário precisa apenas adicionar o modelo ActDL desenvolvido. O projeto modelo faz a gestão de todas as dependências necessárias para a execução do novo serviço *web*, bem como a compilação e empacotamento do código do serviço, por meio da ferramenta Maven [97]. O código empacotado pode, então, ser implantado em um servidor de aplicação Java, como Glassfish [98] e Tomcat [99].

Por meio da interpretação do modelo ActDL, o *framework* Activity-REST é

Figura 36 – Diagrama de objetos para o modelo ActDL para a atividade “análise diferencial de *microarray one-color*”.



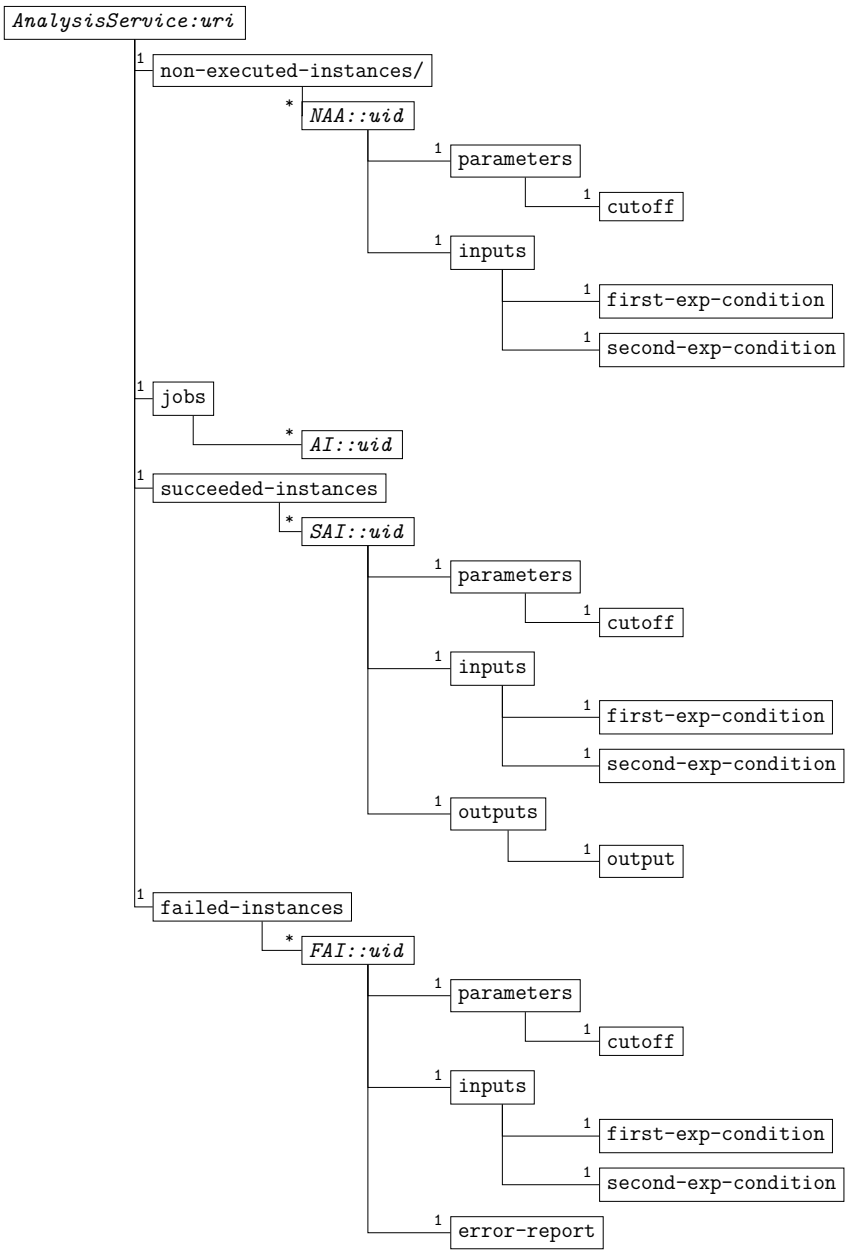
Fonte: Autoria própria. Um retângulo representa um elemento do modelo. A divisão superior do retângulo apresenta a metaclasse da qual esse elemento é instância. A divisão inferior do retângulo apresenta os atributos desses elementos, bem como os valores assumidos por esses atributos. Uma flecha nomeada representa uma referência entre dois elementos. A posição da referência a um elemento em um atributo de cardinalidade múltipla é representada pelo número entre chaves.

capaz de prover os *endpoints* RESTful e as operações necessárias para a execução da atividade de análise segundo o modelo RAS. A Figura 37 apresenta uma visão geral dos recursos disponibilizados para o modelo ActDL provido anteriormente.

5.3.4 Exemplo da interação com o serviço

Como um exemplo da interação de um usuário para a execução da atividade de análise, supomos que o recurso base do serviço seja identificado pelo URI `http://my.address:80/oc-uarray`. Uma requisição `GET` a esse recurso base retorna o URI para a coleção de atividades de análise não executadas. Uma requisição `POST` para o recurso retornado cria um novo contexto de execução para a atividade de análise. A Listagem 11 apresenta a execução dessas operações, enquanto a Figura 38 apresenta a configuração dos *endpoints* do serviço após essa interação. Nesta figura, uma caixa tracejada indica que o usuário conhece o URI do recurso apresentado, porém este recurso ainda não foi inicializado. Dessa maneira, requisições `GET` a este recurso retornam código de *status* `404 - Not Found`. Adicionalmente, a figura abstrai outras instâncias de atividades de análise que por ventura existam nesse mesmo serviço. Após a interação, a atividade de análise encontra-se no estado `CREATED`.

Figura 37 – Visão geral dos recursos expostos pelo *framework* Activity-REST para a atividade de análise "análise diferencial de dados de expressão gênica obtidos por meio de *microarray* (*one-color*)".



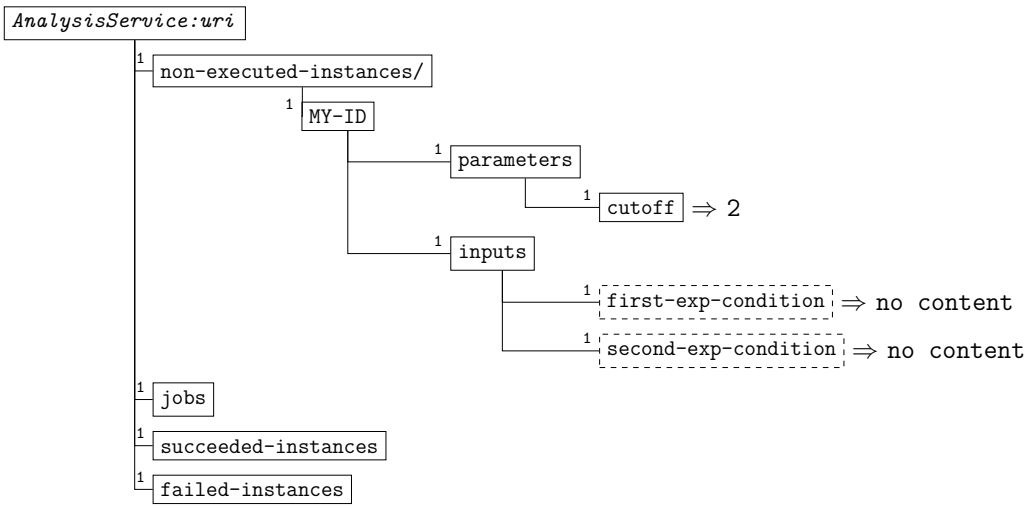
Fonte: Autoria própria.

Listagem 11 – Consulta ao recurso base e criação de um contexto de execução para da atividade de análise.

```
1  > GET /oc-uarray/ HTTP/1.1
2
3  < HTTP/1.1 200 OK
4  < link for posting at /oc-uarray/non-executed-instances
5
6
7  > POST /oc-uarray/non-executed-instances HTTP/1.1
8
9  < HTTP/1.1 201 CREATED
10 < Location: /oc-uarray/non-executed-instances/MY-ID
11 < links for all parameters and input datasets resources
```

Fonte: Autoria própria.

Figura 38 – Estado do serviço após a criação de um novo contexto de execução.



Fonte: Autoria própria.

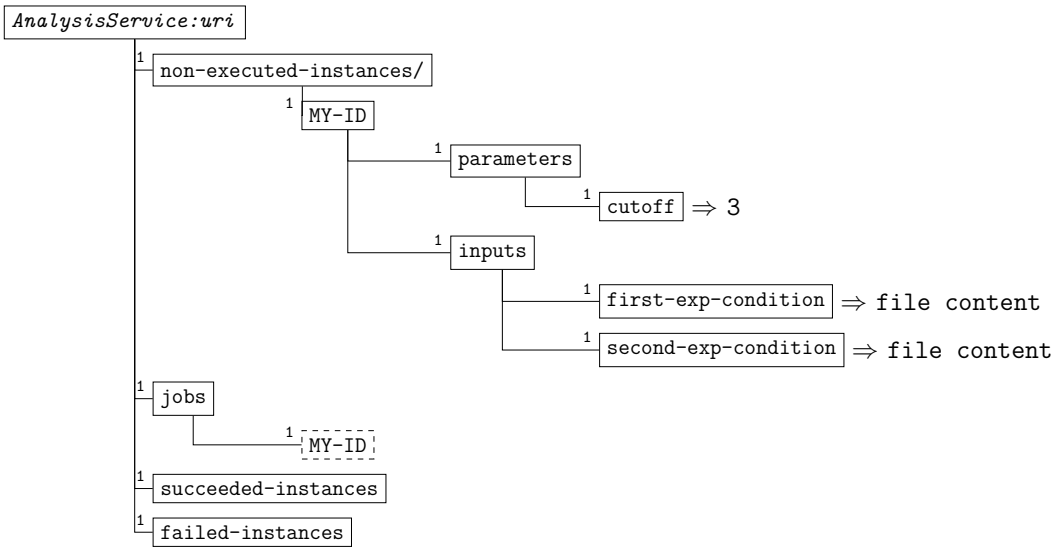
O usuário decide por usar um valor de corte (parâmetro `cutoff`) de `3`. Adicionalmente, ele também inicializa cada conjunto de dados de entrada. O usuário recebe um *link* para a submissão da atividade de análise Essas duas operações são realizadas por meio de requisições `PUT` ao recurso que representa os parâmetros da análise e o recurso que representa cada conjunto de dados de entrada. A Listagem 12 apresenta essas operações. Ao final dessa interação, a atividade de análise encontra-se no estado `READY` e um *link* para a submissão da atividade de análise para processamento é retornado pelo serviço. A Figura 39 apresenta a configuração dos *endpoints* do serviço enquanto a atividade de análise encontra-se no estado `READY`.

Listagem 12 – Inicialização de um parâmetro de uma atividade de análise.

```
> PUT /oc-uarray/non-executed-instances/MY-ID/parameters HTTP/1.1
> Content-type: application/json
> {
>     "cutoff" : 3
> }

< HTTP/1.1 204 No Content


> POST /oc-uarray/non-executed-instances/MY-ID/inputs/first-exp-condition
    HTTP/1.1
> Content-type: text/tsv
> file content

< HTTP/1.1 201 CREATED
< Location:
    /oc-uarray/non-executed-instances/MY-ID/inputs/first-exp-condition


> POST /oc-uarray/non-executed-instances/MY-ID/inputs/second-exp-condition
    HTTP/1.1
> Content-type: text/tsv
> second file content

< HTTP/1.1 201 CREATED
< Location:
    /oc-uarray/non-executed-instances/MY-ID/inputs/first-exp-condition
< link for posting a new job in /oc-uarray/jobs/MY-ID
```

Fonte: Autoria própria.

Figura 39 – Estado do serviço após a inicialização de parâmetros e conjuntos de dados de entrada.

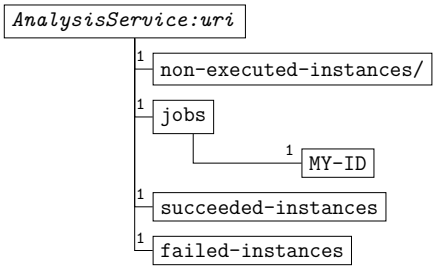


Fonte: Autoria própria.

O usuário dispara a execução da atividade de análise por meio de uma

requisição `PUT` ao URI recebido. Nesse instante, o serviço retira a instância da atividade de análise da coleção `non-executed-instances` e inicia sua execução (estado `RUNNING`). A Figura 40 apresenta a configuração dos *endpoints* do serviço após essa interação.

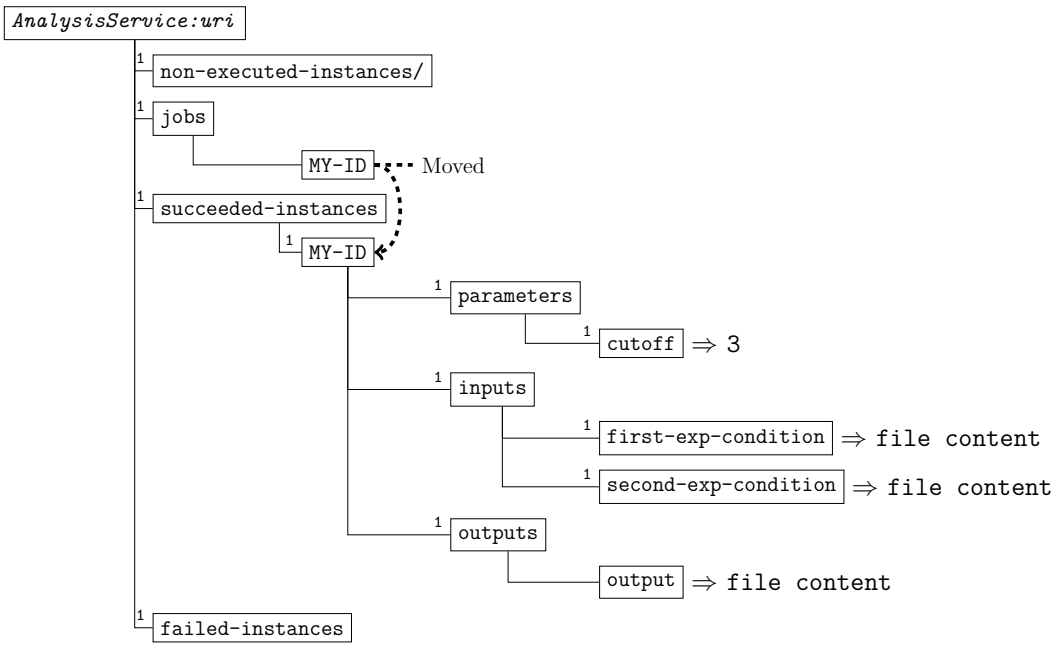
Figura 40 – Estado do serviço durante a execução da atividade de análise.



Fonte: Autoria própria.

Em seguida, o usuário passa a verificar periodicamente o estado dessa execução por meio de requisições `GET` como indicado pelo modelo RAS. Ao final da execução (com sucesso), o serviço cria o novo recurso da atividade de análise na coleção `succeeded-instances`. Novas requisições `GET` ao recurso que representa a execução da atividade de análise (`jobs/MY-ID`) retornam o novo endereço da atividade de análise, indicando que essa execução foi bem sucedida. A Figura 41 apresenta a configuração dos *endpoints* do serviço enquanto a atividade de análise encontra-se no estado `SUCCEEDED`.

Figura 41 – Estado do serviço após a execução da atividade de análise.



Fonte: Autoria própria.

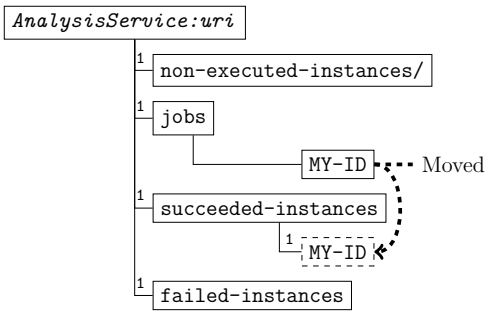
Por fim, o usuário pode recuperar o conjuntos de dados de saída obtido após a execução da atividade de análise, bem como remover os recursos do serviço após essa recuperação. Neste sentido, primeiramente o usuário realiza uma requisição `GET` ao recurso `.../succeeded-instances/MY-ID/outputs/output`, recuperando os resultados da atividade de análise. Em seguida, o usuário realiza uma requisição `DELETE` ao recurso `.../succeeded-instances/MY-ID/` A Listagem 13 ilustra essas interações. A Figura 42 apresenta o estado do serviço após a remoção da atividade de análise.

Listagem 13 – Recuperação de resultados de uma atividade de análise bem sucedida.

```
1 > GET /oc-uarray/succeeded-instances/MY-ID/outputs/output HTTP/1.1
2
3 < HTTP/1.1 200 OK
4 < Content-type: text/csv
5 < result file content
6
7 > DELETE /oc-uarray/succeeded-instances/MY-ID HTTP/1.1
8
9 < HTTP/1.1 200 OK
```

Fonte: Autoria própria.

Figura 42 – Estado do serviço após a remoção de uma atividade de análise.



Fonte: Autoria própria.

5.4 Considerações finais

A linguagem ActDL define uma sintaxe textual sucinta para a representação de modelos AADM, facilitando o desenvolvimento desses modelos por meio de editores de texto comuns. Por sua vez, o *framework* Activity-REST provê a infraestrutura básica para um serviço de análise adaptador segundo o modelo de referência RAS, sendo capaz de adaptar uma ferramenta de análise e prover os *endpoints* e as operações esperadas pelo modelo RAS apenas pela interpretação do modelo ActDL. Por meio de ambos, esperamos facilitar o desenvolvimento de serviços de análise adaptadores em bioinformática.

De modo a utilizar o *framework* Activity-REST para o provimento de um serviço de análise, o desenvolvedor do novo serviço deve criar um novo projeto Java para conter o componente `Configuration Code` e o modelo ActDL utilizado pelo *framework* Activity-REST, bem como para importar todas as dependências desse *framework*. De modo a facilitar esse desenvolvimento, provemos um arquétipo Maven [97] que pode ser utilizado para criar o projeto básico do novo serviço com apenas um comando. De posse desse projeto, o desenvolvedor precisa apenas definir o modelo ActDL utilizado para a adaptação da ferramenta, tendo, então, um serviço adaptador pronto para ser compilado, empacotado e implantado. O arquétipo Maven pode ser encontrado no repositório Git do *framework* Activity-REST[2].

[2] `https://purl.org/cawal/lssb/activity-rest-framework`

6 Geração de clientes para serviços RAS

A solicitação de execução de uma atividade de análise por meio de um serviço RAS demanda o uso de um cliente HTTP próprio para esse serviço. Porém, assim como o desenvolvimento *ad hoc* de um serviço *web*, o desenvolvimento de um cliente para um serviço RAS (cliente RAS-CLI) requer um conjunto de conhecimentos específicos, tais como o domínio das tecnologias de desenvolvimento *web* e do Modelo de referência RAS para os serviços de análise. Adicionalmente, clientes potencialmente distintos podem ser necessários para diferentes ambientes de execução, tais como clientes específicos para a execução via linha de comando e para um dado ambiente integrado de análise de interesse.

Esta capítulo descreve o suporte ao desenvolvimento automático de diferentes tipos de clientes RAS para diferentes plataformas de execução. A Seção 6.1 apresenta uma visão geral da abordagem utilizada para o desenvolvimento de clientes para serviços RAS; a Seção 6.2 apresenta uma sintaxe textual definida para facilitar o desenvolvimento de modelos SDDM; a Seção 6.3 apresenta o desenvolvimento de suporte à obtenção de clientes RAS de linha de comando; a Seção 6.4 apresenta o desenvolvimento de suporte à obtenção de clientes RAS para o ambiente integrado Galaxy; por fim, a Seção 6.5 apresenta algumas considerações finais.

6.1 Visão geral

A disponibilização de um serviço RAS para a execução de uma atividade de análise requer que os usuários desse serviço utilizem um cliente *web* compatível com o modelo de interação definido para esse serviço. Dessa maneira, esse cliente deve interagir com o serviço segundo o modelo RAS, enviando os conjuntos de dados de entrada e parâmetros de execução necessários para a atividade de análise e recuperando os conjuntos de dados de saída produzidos após a execução da instância criada para essa atividade de análise. Adicionalmente, o cliente deve oferecer uma interface adequada para o usuário do serviço de análise, seja esta interface acessível por meio de linha de comando, implementada como uma API para uma dada linhagem de programação ou como uma ferramenta integrada a um ambiente de análise.

Embora seja possível implementar um cliente RAS de maneira *ad hoc*, este desenvolvimento requer um conjunto de conhecimentos e informações específicos, dentre os quais destacam-se:

i) a localização do *endpoint* base do serviço RAS;

ii) a descrição dos conjuntos de dados e parâmetros de execução utilizados/produzidos pela atividade de análise executada por esse serviço;

iii) a localização dos demais *endpoints* do serviço de análise, bem como descrição das operações que devem ser realizadas com esses *endpoints* de modo a executar a instância de atividade de análise;

iv) o domínio de uma tecnologia para a implementação de clientes *web* e a realização das requisições HTTP necessárias, bem como para o tratamento das respostas HTTP recebidas;

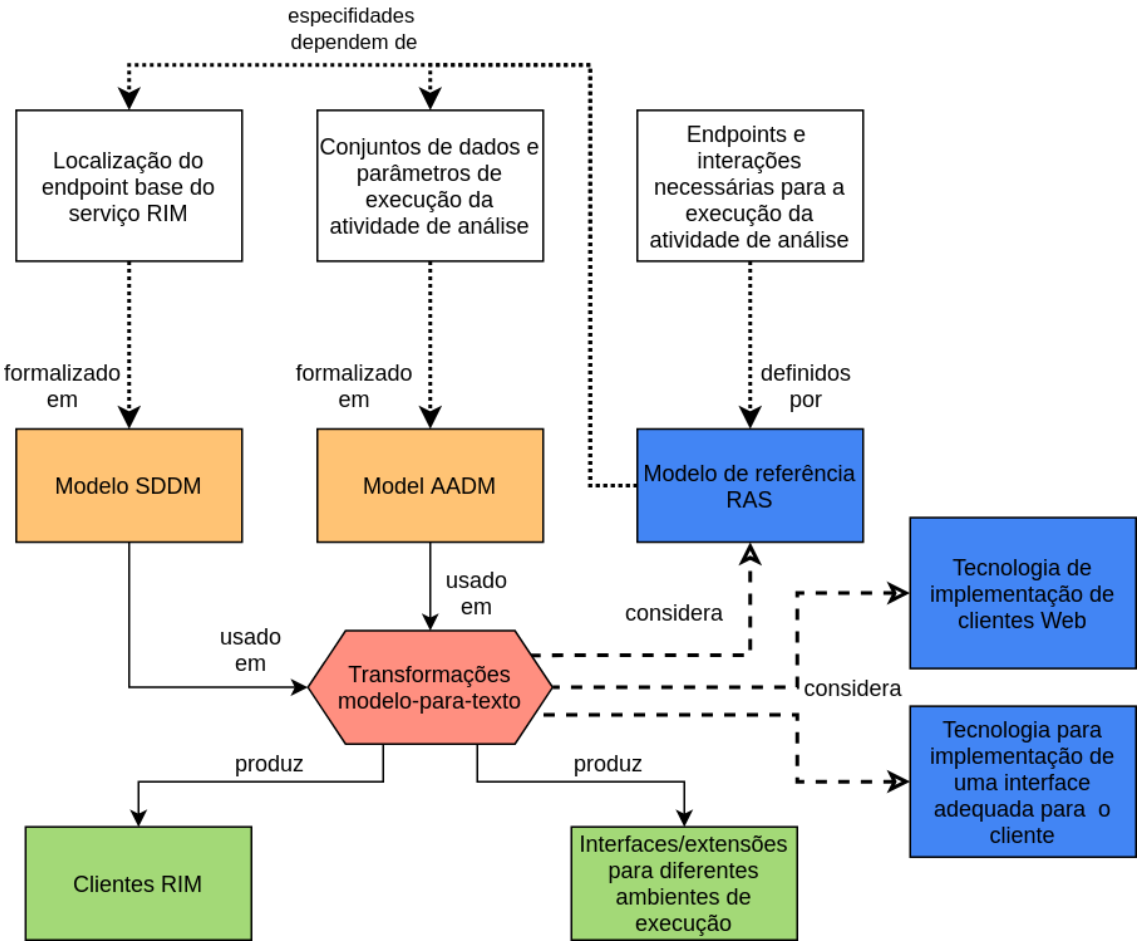
v) o domínio das tecnologias e padrões necessários para prover a interface adequada para os clientes para o ambiente de execução alvo.

A introdução de facilidades para a obtenção de um cliente RAS simplifica também a utilização desse serviço. Assim, de forma análoga à obtenção de um serviço RAS por meio de uma abordagem de desenvolvimento orientada a modelos, também podemos obter clientes para estes serviços segundo uma abordagem pautada nos mesmos modelos utilizados anteriormente.

A Figura 43 apresenta uma visão geral dos principais artefatos da solução para a obtenção automatizada de clientes para serviços RAS, bem como da relação desses artefatos com os modelos e informações necessários para a produção deste cliente. A localização do *endpoint* base do serviço RAS já está formalizado no modelo SDDM definido para este serviço, enquanto informações sobre os conjuntos de dados e parâmetros de execução necessários para a atividade de análise já são definidas pela Descrição ActDL usada por este serviço. Estes dois modelos podem ser usados como modelos fonte para um conjunto de transformações modelo-para-texto para a geração do código fonte (e outros artefatos) que implementa um cliente RAS e/ou as extensões necessárias para um dado ambiente integrado de análise.

Uma transformação modelo-para-texto consiste em um processo que recebe um ou mais modelos e produz um ou mais artefatos textuais [100]. Em uma transformação modelo-para-texto, os modelos iniciais (modelos fonte) são imutáveis durante toda a transformação, de modo que seus elementos podem ser analisados, mas não modificados por essa transformação. Por sua vez, os artefatos criados após a transformação (artefatos alvos) são vistos apenas como sequências de caracteres sobre os quais não é possível realizar leituras que modifiquem o comportamento da transformação e o resultado final. Regras de transformação, definidas em uma linguagem adequada, mapeiam elementos dos modelos fonte para blocos de texto presentes nos artefatos alvos. Estas regras de transformação também são criadas de modo a compor blocos de texto produzidos pela própria regra de transformação com

Figura 43 – Principais componentes da solução para geração de clientes RAS.



Fonte: Autoria própria.Um retângulo branco representa um conhecimento implícito necessário para o desenvolvimento de um cliente RAS. Um hexágono vermelho representa a definição de uma transformação modelo-para-texto. Um retângulo laranja representa um modelo concreto utilizado como modelo fonte para essa transformação. Um retângulo azul representa um modelo abstrato ou tecnologia que é utilizado como base para a definição da transformação modelo-para-texto. Por fim, um retângulo verde representa um artefato obtido por meio da execução dessa transformação.

os blocos de texto produzidos por outras regras de transformação invocadas hierarquicamente maneira adequada. A definição de uma transformação modelo-para-texto em abordagens de desenvolvimentos orientadas a modelos é geralmente realizada de maneira declarativa, de modo que as regras de transformação possam ser (mais) facilmente concretizadas.

A definição de uma transformação modelo-para-texto para a obtenção de um cliente RAS deve considerar o modelo RAS, que define a localização dos demais *endpoints* do serviço RAS e as interações necessárias para a execução da atividade de análise, bem como uma tecnologia para a implementação de clientes *web* e para a implementação da interface adequada ao cliente desejado. Essa transformação pode

ser definida de modo que o código do cliente e outros artefatos produzidos reutilizem bibliotecas auxiliares e outras dependências pré-existentes.

No contexto desse trabalho, definimos transformações para a obtenção automática de um cliente RAS executável como uma ferramenta de linha de comando (Cliente RAS-CLI) e como uma extensão para o ambiente integrado de análise Galaxy (Cliente RAS-Galaxy). Porém, outras transformações modelo-para-texto podem ser (futuramente) definidas de modo a obter outros tipos de clientes RAS, como APIs dedicadas para uma linguagem de programação e/ou como extensões para outros ambientes integrados de análise.

6.2 Uma sintaxe textual para modelos SDDM

Assim como definimos a sintaxe textual ActDL para a representação de modelos AADM, também definimos uma sintaxe textual simples para a representação de modelos de implantação, chamada *Service Deployment Description Language* (SDDL). Dessa maneira, podemos representar e editar esses modelos utilizando qualquer editor de texto convencional.

A Listagem 14 apresenta um exemplo de um modelo SDDM representado por meio da linguagem SDDL. Tal modelo começa com a declaração de um elemento `Deployment` ("`deployment`", linha 1) seguida de um conjunto de declarações entre chaves ("`{`" e "`}`"), as quais delimitam o escopo deste elemento (linhas 1 e 14). Em seguida, elementos `Service`, `ServiceContainer` e `Contact` são declarados nesta ordem, por meio das palavras-chave "`of service`" (linha 2), "`into container`" (linha 6) e "`contact`" (linha 10), respectivamente. O valor para o atributo `name` de cada um desses elementos é definido em seguida.

Outros atributos desses elementos são definidos dentro dos escopos delimitados por chaves (linhas 2 a 5, 6 a 9 e 10 a 13, respectivamente). Nesse sentido, um elemento `Service` contém os atributos `apiVersion` (linha 3) e uma descrição textual (atributo `description`, linha 4). Por sua vez, um elemento `ServiceContainer` contém os atributos `baseUrl` (linha 7) e `description` (linha 8). Por fim, um elemento `Contact` contém os atributos `email` (linha 11) e `url` (linha 12).

A sintaxe da linguagem SDDL foi definida utilizando-se o suporte provido pelo *framework* Xtext. O Apêndice D apresenta a gramática da linguagem SDDL por meio da forma de Backus-Naur estendida. Adicionalmente, um editor dedicado para a manipulação de modelos SDDL, utilizável em conjunto com o ambiente de desenvolvimento Eclipse, está disponível no repositório da linguagem[1].

[1] `https://purl.org/cawal/lssb/sddmm`.

Listagem 14 – Exemplo de um modelo SDDM representado por meio de SDDL.

```
deployment {
    of service 'my-analysis-service' {
        api-version '1.1'
        description 'An analysis service'
    }
    into container 'my-host-machine' {
        base-url 'http://my.service.repository:80/my-analysis-service'
        description 'Main machine of the IT department'
    }
    contact 'John Doe' {
        email 'john.doe@email.com'
        url 'https://john.doe.com/'
    }
}
```

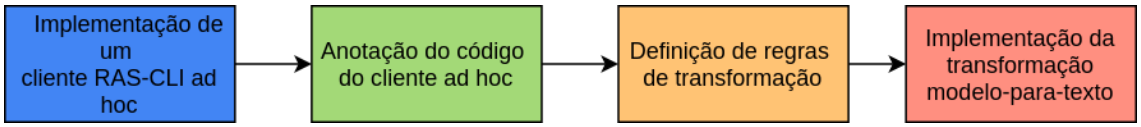
Fonte: Autoria própria.

6.3 Geração de clientes de linha de comando

A execução de um cliente RAS por meio de uma interface de linha de comando (Cliente RAS-CLI) representa a opção mais simples para que os especialistas de domínio utilizem um serviço RAS para a execução de uma atividade de análise. De modo a prover suporte à obtenção de clientes RAS-CLI, definimos e implementamos um conjunto de regras de transformação para obter esses clientes a partir dos mesmos modelos utilizados para a criação um dado serviço RAS.

A Figura 44 apresenta as etapas realizadas de modo a prover suporte à geração de clientes RAS-CLI. Inicialmente, um cliente RAS-CLI foi implementado de maneira *ad hoc* para um serviço RAS disponível. Em seguida, o código desse cliente foi analisado e dividido em seções anotadas como *específicas de serviço* e *independentes de serviço*, isto é, seções do código do cliente RAS cuja implementação é definida de maneira específica para o serviço RAS invocado ou cuja implementação é idêntica entre clientes para diferentes serviços. Regras de transformação foram, então, definidas para as seções específicas de serviço, de modo a descrever como a implementação daquela seção pode ser obtida a partir das informações presentes na descrição ActDL e no modelo de implantação do serviço. Finalmente, uma transformação modelo-para-texto foi implementada. Essa transformação recebe uma descrição ActDL e um modelo de implantação como modelos fontes, produzindo o código cliente específico do serviço de interesse.

Figura 44 – Etapas do desenvolvimento de suporte à geração de clientes RAS de linha de comando.



Fonte: Autoria própria.

6.3.1 Desenvolvimento do cliente *ad hoc*

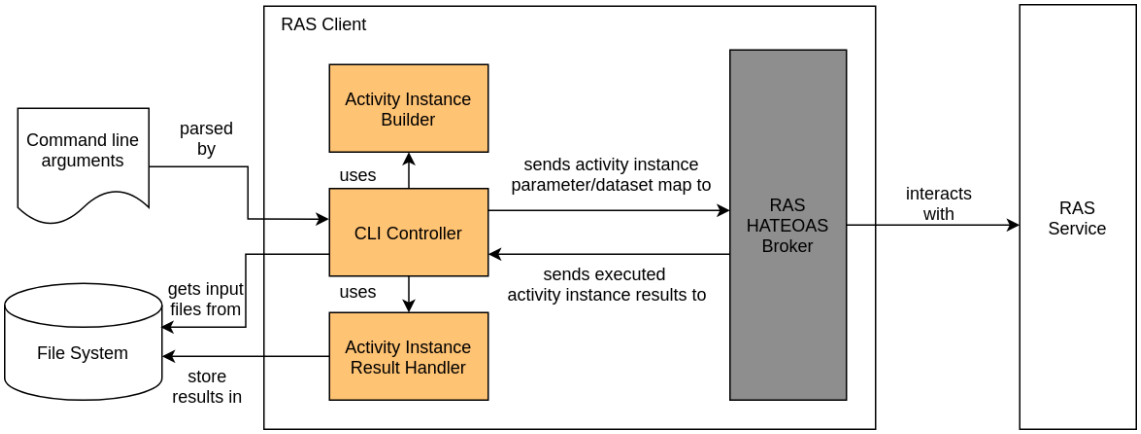
Inicialmente, o serviço `david-chart-report` do repositório GEAS-RAS foi escolhido para o desenvolvimento *ad hoc* de um cliente RAS. Este cliente foi implementado utilizando a linguagem de programação Kotlin [101]. Também foram utilizadas as bibliotecas `picocli` [102] e `JAX-RS Client API` [103] para o auxílio à implementação da interface de linha de comando e das interações RESTful com o serviço, respectivamente. A linguagem de programação Kotlin [101] possui grande expressividade e artefatos implementados nesta linguagem interoperam de forma transparente com outros artefatos definidos utilizando a linguagem Java. Tais características permitiram a reutilização de artefatos já definidos anteriormente para o suporte ao desenvolvimento de serviços RAS, bem como a escrita de código mais sucinto para os clientes RAS.

A implementação do cliente RAS *ad hoc* utilizou a descrição ActDL do serviço `david-chart-report`, bem como o seu modelo de implantação, de modo a definir os detalhes de implementação dependentes de serviço necessários para concretizar o modelo RAS. Neste sentido, os URLs de cada recurso provido pelo serviço foram definidos utilizando o URL base presente no modelo de implantação e os nomes dos conjuntos de dados e parâmetros de execução presentes na descrição ActDL.

A descrição ActDL do serviço também foi utilizada para definir os detalhes da interface de linha de comando do cliente implementado esse cliente. Neste sentido, para cada conjunto de dados e parâmetro de execução da atividade de análise foi criado um argumento de linha de comando utilizado pelo cliente, nomeado segundo o conjunto de dados ou parâmetro de execução associado. A cardinalidade desses argumentos e as *strings* de auxílio ao usuários também foram retiradas dos elementos fonte da descrição ActDL. Adicionalmente, a definição do tipo de dado representado por cada parâmetro de execução foi utilizada para definir o tipo primitivo do argumento de linha de comando análogo utilizado pelo cliente.

Após a implementação inicial do cliente RAS *ad hoc*, o código deste cliente foi reestruturado em unidades responsáveis pelas diferentes atividades do modelo RAS, seguindo a conceitualização de grupos de atividades ilustrada no capítulo 4 (veja Figura 17). Neste sentido, o código do cliente foi estruturado para executar três grupos de atividades em ordem: inicialização do conjuntos de dados e parâmetros de execução da atividade de análise, execução e, por fim, recuperação dos resultados. Em seguida, refatoramos o código de maneira a remover referências explícitas aos URLs dos *endpoints* do serviço de análise (com exceção do URL do recurso base). Os URLs dos *endpoints* necessários à execução são obtidos por meio da inspeção dos controles de hipermídia presentes nas respostas HTTP retornadas pelo serviço. Adicionalmente, apenas o nome dos conjuntos de dados e parâmetros de execução

Figura 45 – Componentes de um cliente RAS-CLI



Fonte: Autoria própria. Um retângulo laranja representa um componente específico de serviço. Um retângulo cinza representa um componente independente de serviço.

são necessários para que os URL corretos sejam encontrados entre os controles de hipermídia retornados.

6.3.2 Anotação e separação de seções de código específicas de serviço e independentes de serviço do cliente RAS

Durante essa etapa, o código do cliente RAS foi refatorado de modo a estruturá-lo em componentes independentes de serviço e componentes específicos de serviço. Componentes específicos de serviço são implementados de maneira diferente em cada cliente de um novo serviço RAS, enquanto componentes independentes de serviço são reutilizados por diferentes clientes RAS. Essa refatoração tornou possível concentrar o esforço de transformação modelo-para-texto apenas nos componentes específicos de serviço, reutilizando os componentes independentes de serviço como dependências no projeto de cada cliente RAS. A Figura 45 apresenta uma visão geral dos diferentes componentes do cliente, bem como aspectos da interação entre esses componentes.

O componente `RAS HATEOAS Broker` foi definido de maneira independente de serviço, sendo, portanto, compartilhado por todos os clientes RAS-CLI. Este componente é responsável por receber de `CLI Controller` um objeto que representa uma instância de atividade de análise e o endereço base de um serviço RAS e executar essa instância remotamente neste serviço. Este componente suporta tanto o modo de espera utilizando *polling* quanto o modo de espera por notificações SSE durante a execução remota da instância de atividade de análise. Assim, o modo de espera a ser utilizado pode também ser informado como argumento previamente à execução da instância de atividade de análise, sendo o modo padrão de espera definido como *polling*. Ao final desta execução, `RAS HATEOAS Broker` recupera os conjuntos de

dados de saída produzidos e retorna a instância de atividade de análise com esses conjuntos de dados para o componente `CLI Controller`.

Os componentes `Activity Instance Builder`, `Activity Instance Result Handler` e `CLI Controller` são específicos de serviço, sendo implementados de acordo com o serviço RAS a ser executado. O componente `CLI Controller` analisa os argumentos de linha de comando informados pelo usuário do cliente e usa tanto `Activity Instance Builder` quanto `Activity Instance Result Handler` durante a execução. Adicionalmente, este componente valida os argumentos informados para a execução da tarefa, verificando se os tipos de dados associados aos parâmetros de execução foram informados corretamente, bem como se todos os arquivos definidos para os conjuntos de dados de entrada existem.

O componente `Activity Instance Builder` recebe do componente `CLI Controller` a informação dos valores dos conjuntos de dados de entrada e parâmetros de execução utilizados pela atividade de análise, produzindo um objeto que representa essa instância. Esse objeto é, então, enviado para o componente `RAS HATEOAS Broker` de modo que a instância seja executada remotamente.

Após a execução da instância de atividade de análise, o componente `Activity Instance Result Handler` recebe de `CLI Controller` o objeto que representa a instância de atividade de análise já executada por `RAS HATEOAS Broker`. `Activity Instance Result Handler` então verifica se a execução foi bem sucedida e, caso tenha sido, armazena os conjuntos de dados de saída no sistema de arquivo segundo os caminhos informados pelo usuário do cliente. Caso a execução da instância não tenha sido bem sucedida, este componente avisa ao usuário da falha ocorrida e termina a execução do cliente RAS-CLI.

O componente `CLI Controller` foi definido por meio do suporte da biblioteca `picocli` para a análise dos argumentos de linha de comando informados pelo usuário. Porém, esta biblioteca precisa ser configurada de maneira específica para os conjuntos de dados e parâmetros de execução utilizados por uma atividade de análise. De modo a definir como esta biblioteca deveria ser configurada para cada novo cliente, definimos que todos os conjuntos de dados e parâmetros de execução devem ser informados precedidos de indicadores desses argumentos. Esses indicadores devem ser formatados segundo o nome desse conjunto de dados ou parâmetro de execução, precedido de dois hifens. Cada valor para um conjunto de dados ou parâmetros de execução deve ser informado em seguida em um argumento de linha de comando próprio (i.e., separados por espaço na linha de comando). A Listagem 15 apresenta um exemplo genérico desse formato para um cliente cuja atividade de análise utilize três parâmetros e um conjunto de dados de entrada, produzindo um conjunto de dados de saída.

Listagem 15 – Formato básico para linha de comando de um cliente RAS-CLI.

```
[rim-client] --dataset1name input_file1 input_file2
             --parameter1name val1 val2 val3
             --parameter2name val
             --parameter3name val
             --outputDataset1Name output_file1
```

Fonte: Autoria própria.

Após a separação dos componentes de um cliente RAS-CLI, o componente `RAS HATEOAS Broker` foi externalizado como um pacote Java próprio, de modo a reutilizá-lo como dependência em todos os clientes RAS. Os demais componentes, específicos de serviço, são criados durante a transformação modelo-para-texto.

Por fim, para que o código do cliente possa ser compilado e empacotado com todas as dependências necessárias, um projeto Maven também precisa ser criado. Poucas diferenças são encontradas entre os projetos Maven de dois clientes RAS. Mais notadamente, é necessário que identificadores de artefatos (atributo `artifactId`), de grupo (atributo `groupId`) e de versão (atributo `version`) sejam criados diferentemente entre dois clientes, de modo que cada cliente seja empacotado com um nome único. Durante a geração do cliente, podemos definir um procedimento padrão para a geração do valor inicial desses identificadores, sendo possível que estes valores sejam modificados posteriormente segundo as necessidades do desenvolvedor. Dessa maneira, podemos utilizar um arquétipo Maven para prover a estrutura geral desse projeto. Posteriormente, estes valores podem ser modificados pelo desenvolvedor do cliente segundo suas necessidades.

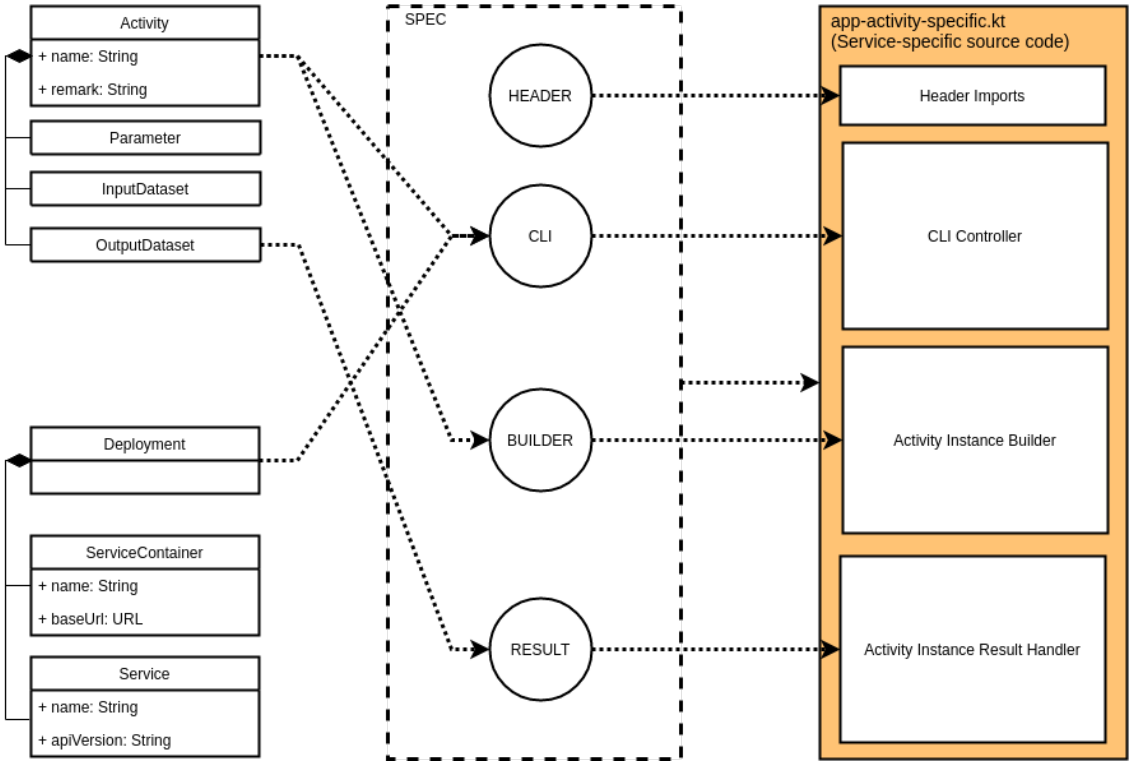
6.3.3 Definição abstrata de regras de transformação

Esta etapa consistiu da definição de um conjunto de regras de transformação abstratas para a obtenção de um cliente RAS a partir de uma descrição ActDL e de um modelo de implantação. Esse conjunto de regras abstratas foi, posteriormente, utilizado como base para a concretização da transformação modelo-para-texto.

Durante a transformação, cada regra de transformação definida recebe um conjunto de elementos dos modelos iniciais e produz um ou mais elementos do projeto de um cliente RAS-CLI. Os elementos produzidos por uma regra podem ser arquivos de interesse do projeto de cliente ou o conteúdo desses arquivos. As regras de transformação são definidas de maneira hierárquica em relação aos elementos do projeto do cliente produzido. Dessa maneira, regras mais abstratas ou responsáveis pela produção de elementos mais gerais invocam regras mais concretas, responsáveis pela criação de elementos específicos do cliente RAS-CLI.

A Figura 46 apresenta uma visão geral de cinco regras de transformação definidas nessa etapa. A regra de transformação SPEC invoca as regras HEADER,

Figura 46 – Visão geral de uma transformação modelo-para-texto para a obtenção de um cliente RAS-CLI.



Fonte: Autoria própria. Regras de transformação são apresentadas ao centro como círculos nomeados. Uma seta associando uma metaclasse a uma regra de transformação indica que elementos daquela metaclasse nos modelos fonte são utilizados para a execução da regra de transformação. Uma seta associando uma regra de transformação a uma estrutura do cliente RAS-CLI representa que essa estrutura é produzida pela execução da regra de transformação.

CLI, BUILDER e RESULT, passando os elementos corretos dos modelos iniciais a essas sub-regras. Após a execução dessas sub-regras, SPEC combina os resultados de modo a produzir o elemento `Service-specific source code` do projeto do cliente RAS-CLI.

A regra HEADER produz os cabeçalhos do arquivo representado pelo elemento `Service-specific source code`. Esta regra independe do conteúdo dos modelos fontes da transformação. Dessa maneira, essa regra não recebe nenhum elemento desses modelos para sua execução.

As demais regras, CLI, BUILDER e RESULT, são responsáveis pela produção dos componentes `CLI Controller`, `Activity Instance Builder` e `Activity Instance Result Handler`, respectivamente. Essas regras, bem como sub-regras relacionadas, produzem elementos do projeto do cliente RAS-CLI de maneira dependente das informações presentes nos modelos iniciais. Por esta razão, as regras CLI e BUILDER recebem elementos dos modelos iniciais necessários para sua execução e

invocam suas sub-regras repassando a elas os elementos desses modelos sobre os quais atuam. Por exemplo, a regra CLI repassa os conjuntos de dados e parâmetros de execução, bem como o URL base do serviço, para sub-regras que irão criar a definição dos argumentos de linha de comando e interagir com o componente `RAS HATEOAS Broker`. Por sua vez, a regra BUILDER irá utilizar a definição dos conjuntos de dados e dos parâmetros de execução para definir o procedimento para a criação de um objeto a partir dos dados providos por `CLI Controller`. Por fim, a regra RESULT recebe conjuntos de dados de saída para definir o procedimento de persistência dos resultados obtidos após a execução da instância de atividade de análise.

O Apêndice E apresenta em maiores detalhes as regras de transformação abstratas definidas nessa etapa.

6.3.4 Implementação da transformação modelo-para-texto

Esta etapa consistiu da implementação das regras de transformação utilizadas para obter um cliente RAS a partir de uma descrição ActDL e de um modelo de implantação. Um arquétipo Maven foi utilizado para prover a estrutura geral do projeto do cliente e a inclusão dos componentes independentes de serviço, bem como de outras dependências necessárias à compilação e execução do cliente gerado. Durante a geração do cliente, os artefatos específicos de serviço de um cliente RAS (componentes `CLI Controller`, `Activity Instance Builder` e `Activity Instance Result Handler`) são adicionados ao projeto gerado a partir desse arquétipo. A linguagem Kotlin também foi utilizada para a implementação da transformação modelo-para-texto que gera esses artefatos.

A Listagem 16 apresenta dois fragmentos da implementação da transformação modelo-para-texto usada para a geração dos clientes RAS. O primeiro fragmento apresenta o método `getCliFileContents` (regra SPEC), utilizado para a geração do código específico de serviço do cliente RAS. Este método recebe uma descrição ActDL e um modelo de implantação e retorna uma *string* contendo o código produzido. O método `getCliFileContents` e os demais métodos invocados para a geração do código específico de serviço são implementados de maneira declarativa por meio de composição de *strings*. Neste sentido, a *string* retornada por este método é definida entre os sinais de três aspas duplas (linhas 80 a 88). Essa *string* tem seu conteúdo composto com os resultados dos métodos auxiliares invocados *in loco* (linhas 81, 83, 85 e 87), os quais também retornam *strings*. A indentação das linhas produzidas no arquivo final é dada pela distância entre o conteúdo das linhas e o caractere de *pipe* ("|"), retirado da *string* retornada por `generateCliFileContents` pela chamada ao método `trimMargin` da classe `String`.

O segundo fragmento apresenta detalhes do método `createCallable`, invo-

Listagem 16 – Fragmentos do código Kotlin implementando a transformação modelo-para-texto para a obtenção de um cliente RAS-CLI.

```
79      fun getCliFileContents(activity: Activity, deployment: Deployment) =
80          """
81          |${cliFileHeader()}
82          |
83          |${createCallable(activity, deployment)}
84          |
85          |${createWriteOutputDatasets(activity)}
86          |
87          |${createGetActivityInstance(activity)}
88          """.trimMargin("|");
```

⋮

```
110     private fun createCallable(activity: Activity, deployment: Deployment)
        =
111             """
112       |@Command(name = "${activity.name}", version = ["${
113             deployment.getService().getApiVersion()}"])
114       |class AppCallable() : Callable<Int> {
115       |
116       |    ${createParameters(activity)}
117       |
118       |    ${createOptions(activity, deployment)}
119       |
120       |    override fun call(): Int {
121       |        return execute(this)
122       |    }
123       |}
124        """.trimMargin("|")
```

Fonte: Autoria própria.

cado por `generateCliFileContents` (linha 83). O método `createCallable` implementa a regra de transformação CLI. Este método define a interface de linha de comando da ferramenta, processada por meio da biblioteca `picocli`. Neste sentido, esse método produz a definição de uma classe chamada `AppCallable`, subclasse da classe `Callable` do *framework* `picocli`, que representa a definição da interface de linha de comando do cliente RAS. A classe `AppCallable` é anotada com `@Command` para representar que uma instância dessa classe deve ser criada contendo os argumentos do usuário já processados. Um nome para a ferramenta de análise e uma versão para essa ferramenta são incluídos como atributos da anotação `@Command`. Outros métodos são chamados para criar a definição dos argumentos obrigatórios (linha 116) e dos argumentos opcionais (linha 118) do cliente RAS. As demais regras de transformação são implementadas de maneira semelhante, usando composição de *strings*.

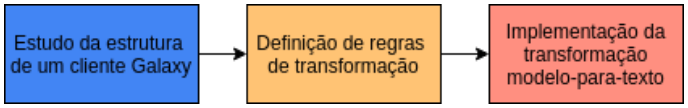
6.4 Geração de clientes para o ambiente Galaxy

O objetivo dessa atividade foi demonstrar que adaptadores de serviços RAS para ambientes integrados de análise podem também ser obtidos facilmente por

meio de uma abordagem de desenvolvimento baseada em modelos. Neste sentido, o ambiente Galaxy foi escolhido como plataforma alvo.

A Figura 47 apresenta as etapas realizadas de modo a prover suporte à obtenção de clientes RAS-Galaxy por meio da transformação de uma descrição ActDL e de um modelo de implantação. Inicialmente, foi realizado um estudo da estrutura de um cliente Galaxy. Em seguida, foi definido um conjunto de regras de transformação abstratas que mapeiam os elementos de uma descrição ActDL e de um modelo de implantação para os elementos do cliente RAS-Galaxy. Por fim, foi realizada a implementação da transformação modelo-para-texto correspondente.

Figura 47 – Etapas do desenvolvimento do suporte à geração de clientes RAS para o ambiente Galaxy.



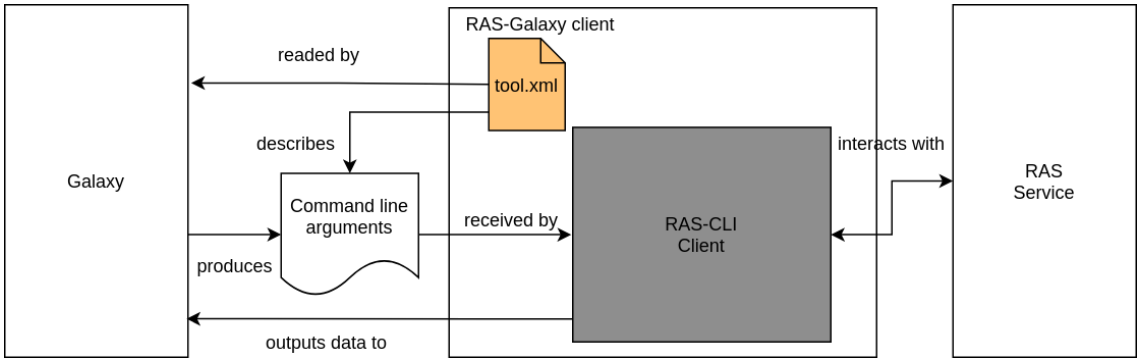
Fonte: Autoria própria.

6.4.1 Estrutura de um cliente RAS-Galaxy

Um cliente Galaxy consiste de dois artefatos: i) uma ferramenta de linha de comando utilizada para executar uma atividade de análise; e ii) um modelo de descrição dessa ferramenta (arquivo `tool.xml`). Para o cliente Galaxy, a ferramenta de linha de comando utilizada é vista como uma caixa preta, sendo apenas necessário descrever como essa ferramenta deve ser invocada pelo ambiente. Dessa maneira, podemos reutilizar nesse ambiente o cliente RAS-CLI gerado automaticamente por meio da transformação definida na seção anterior. O arquivo `tool.xml` que descreve a interface de linha de comando desse cliente, por sua vez, pode ser gerado também por meio de uma transformação modelo-para-texto.

A Figura 48 apresenta uma visão geral dos componentes de um cliente RAS-Galaxy, bem como a relação desses componentes com o ambiente Galaxy e o serviço RAS durante a execução de uma instância de atividade de análise. Em tempo de execução, o ambiente Galaxy lê o modelo de descrição da ferramenta e provê ao usuário uma interface gráfica para a submissão de valores para parâmetros de execução e conjuntos de dados. Após o usuário informar esses valores, o ambiente Galaxy utiliza novamente o modelo de descrição da ferramenta para produzir uma chamada de linha de comando que irá executar a ferramenta subjacente (cliente RAS). Ao final da execução, o ambiente Galaxy recupera os conjuntos de dados de saída produzidos e possibilita que o usuário os utilize em uma nova atividade de análise.

Figura 48 – Componentes de um cliente RAS-Galaxy.



Fonte: Autoria própria.

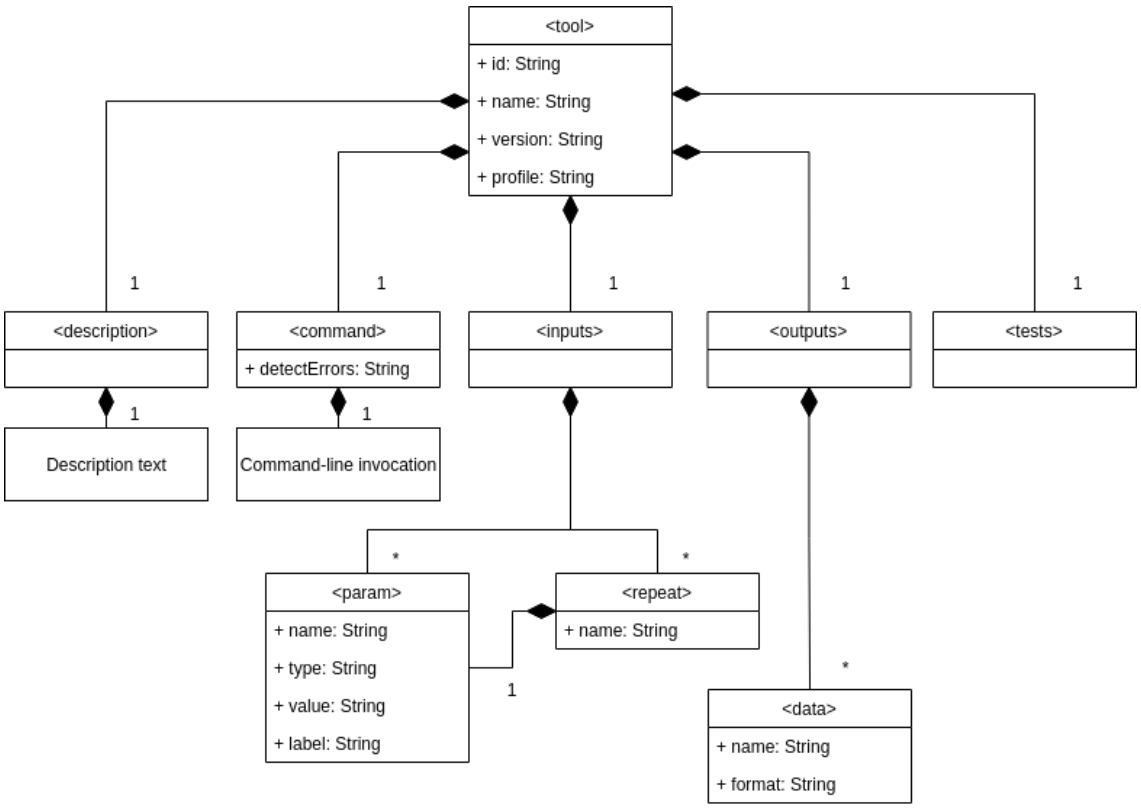
A Figura 49 apresenta uma visão geral do conteúdo do arquivo `tool.xml` como um diagrama de classes UML. Esta figura apresenta apenas os elementos de interesse para o provimento de um cliente RAS-Galaxy. Uma classe UML representa um tipo de elemento XML. Atributos dessa classe representam atributos dos elementos XML daquele tipo. Um objeto (retângulo sem divisões) representa um nó de texto XML, i.e., uma *string* incluída entre o abrir e fechar das *tags* que delimitam o elemento XML. Uma relação de composição entre duas classes ou entre uma classe e um objeto representa que há uma relação de composição entre elementos XML respectivos. A cardinalidade dessa relação é representada como unitária (`1`) ou múltipla (`*`).

O elemento `<tool>` representa a raiz de um arquivo `tool.xml`. Esse elemento define um identificador do cliente Galaxy (atributo `id`), o nome desse cliente (atributo `name`), a versão do cliente (atributo `version`) e versão mínima do ambiente Galaxy no qual o cliente pode ser utilizado (atributo `profile`). Adicionalmente, o elemento `<tool>` precisa conter elementos `<description>`, `<command>`, `<inputs>`, `<outputs>` e `<tests>`.

Um elemento `<description>` define um texto de descrição do cliente Galaxy. Este elemento não tem atributos próprios. O texto de descrição é colocado em um nó de texto XML do elemento `<description>`. Por sua vez, um elemento `<command>` define a invocação da ferramenta de linha de comando que executa a instância de atividade de análise (i.e., o cliente RAS-CLI). Esse comando é incluído em um nó de texto XML do elemento. Substituições de valores de parâmetros de execução e caminhos de arquivos usado para cada conjunto de dados, bem como outras operações necessárias para a execução da ferramenta de análise, são definidas por meio de um motor de manipulação de texto chamado `Cheetah` [104]. Adicionalmente, o elemento `<command>` define qual estratégia deve ser utilizada para verificação de erros de execução da ferramenta de análise encapsulada (atributo `detectErrors`).

O elemento `<inputs>` define todos os parâmetros de execução e conjuntos

Figura 49 – Estrutura do conteúdo do arquivo `tool.xml`



Fonte: Autoria própria.

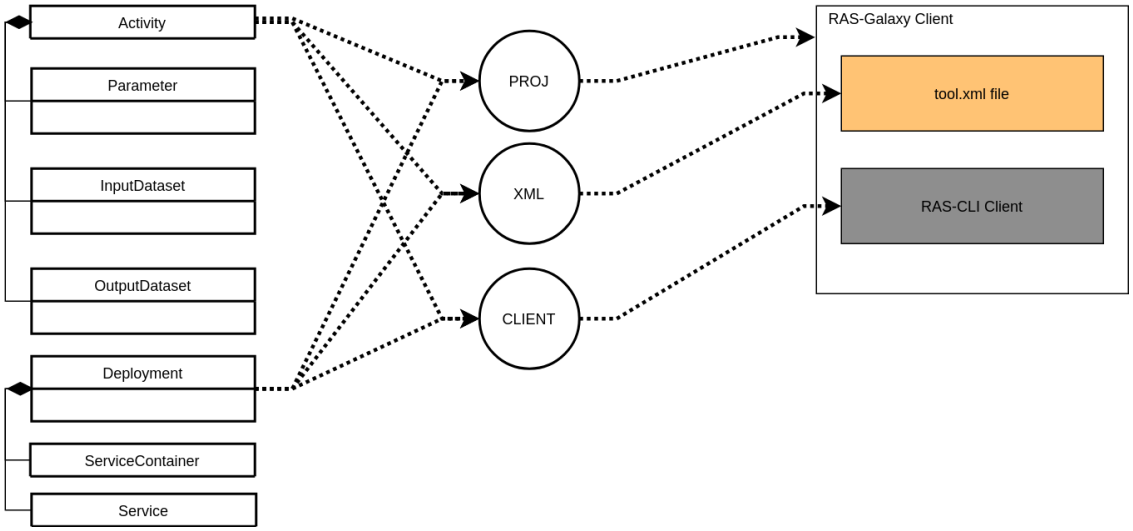
de dados de entrada utilizados pela ferramenta de análise adaptada. Esse elemento pode conter elementos `<param>` e elementos `<repeat>`, que são utilizados em conjunto para definir parâmetros de execução ou conjuntos de dados de entrada de diferentes cardinalidades. Um elemento `<param>`, se definido como elemento filho do elemento `<inputs>`, representa um parâmetro de execução ou conjunto de dados de cardinalidade unitária. Por sua vez, um elemento `<param>` definido como filho de um elemento `<repeat>` representa um conjunto de dados ou parâmetro de execução com cardinalidade múltipla. O elemento `<param>` possui, além do nome (atributo `name` , o tipo de dados que recebe (atributo `type`), um valor padrão (atributo `value`) e um rótulo descritivo (atributo `label`).

O elemento `<outputs>` define os conjuntos de dados de saída produzidos pela ferramenta de análise. Cada conjunto de dados de saída é representado como um elemento `<data>`, o qual define um nome para esse conjunto de dados (atributo `name`) e um tipo do arquivo produzido (atributo `format`). Por fim, um elemento `<tests>` permite definir um conjunto opcional de valores a serem utilizados para parâmetros e conjuntos de dados de modo a realizar um teste da execução do cliente Galaxy.

6.4.2 Definição abstrata de regras de transformação

Esta etapa consistiu da definição de um conjunto de regras de transformação para a obtenção de um arquivo `tool.xml` a partir de uma descrição ActDL e de um modelo de implantação. Essas regras de transformação abstratas foram utilizadas como base para a concretização da transformação na próxima etapa.

Figura 50 – Regras de transformação que definem a estrutura do projeto do cliente RAS-Galaxy.



Fonte: Autoria própria. Regras de transformação são apresentadas ao centro como círculos nomeados. Uma seta associando uma metaclasse a uma regra de transformação indica que elementos daquela metaclasse nos modelos fonte são utilizados para a execução da regra de transformação. Uma seta associando uma regra de transformação a uma estrutura do cliente RAS-Galaxy representa que essa estrutura é produzida pela execução da regra de transformação.

A Figura 50 apresenta as regras de transformação PROJ, CLIENT e XML, as quais são executadas em sequência e definem a estrutura geral do cliente RAS-Galaxy. A regra de transformação PROJ define um projeto de cliente Galaxy. Esta regra cria um diretório e inclui nesse diretório os artefatos produzidos pelas regras XML e CLIENT. A regra CLIENT cria um cliente RAS-CLI para ser incluído no cliente RAS-Galaxy. Essa regra pode ser construída de maneira a invocar as regras definidas previamente para a obtenção de um cliente RAS-CLI. Por sua vez, a regra XML cria um arquivo `tool.xml` para definir a extensão Galaxy corretamente. Esta regra invoca um conjunto de outras regras responsáveis por criar os diferentes elementos XML nesse arquivo.

O Apêndice F apresenta em maiores detalhes as regras de transformação abstratas definidas nessa etapa.

6.4.3 Implementação da transformação

A Listagem 17 apresenta fragmentos da classe `ActDLToGalaxyToolWrapper`, que implementa a transformação modelo-para-texto para a obtenção de um adaptador Galaxy. Esta classe, implementada utilizando a linguagem Kotlin, provê o método `generate` (linha 10), o qual recebe uma descrição ActDL e um modelo de implantação e retorna um arquivo `tool.xml` válido para a adaptação de um cliente RAS de linha de comando. O método `generate` cria um arquivo temporário (linha 12), obtém uma *string* com o resultado da transformação modelo-para-texto (linha 13) e salva essa *string* no arquivo temporário (linha 14). Por fim, esse arquivo é retornado ao usuário do método (linha 16).

A transformação de uma descrição ActDL e de um modelo de implantação no texto do arquivo adaptador Galaxy é implementada pelo método `createXml` (linha 20), bem como outros métodos auxiliares invocados por `createXml`. O método `createXml` cria a estrutura geral do documento XML do adaptador Galaxy e inclui nessa estrutura as informações específicas do cliente RAS. Aqui, novamente utilizamos composições de *strings* criadas por funções auxiliares para produzir o texto do adaptador Galaxy, assim como fizemos anteriormente para a obtenção do cliente RAS. Por exemplo, uma *string* representando cada argumento de linha de comando é definida a partir de cada parâmetro, conjunto de dados de entrada e/ou conjunto de dados de saída presentes na descrição ActDL (linhas 34 a 42).

A Listagem 18 apresenta um fragmento da classe `GalaxyClientGenerator`. Essa classe implementa o método `generateClient`, o qual recebe uma descrição ActDL e um modelo de implantação (linha 11) e retorna um diretório contendo o projeto para o cliente RAS e para o adaptador Galaxy desse cliente. O método `generateClient` obtém o projeto Java de um novo cliente RAS (linha 15) utilizando a classe `JavaProjectGenerator` definida anteriormente para executar a geração de um cliente RAS de linha de comando (linhas 14 a 16). Então, `generateClient` utiliza o método `generate` da classe `ActDLToGalaxyToolWrapper` para obter um adaptador Galaxy para esse cliente (linha 18). Por fim, ambos artefatos são combinados (linha 20) e o diretório base do novo projeto é retornado ao usuário do método (linha 22). Neste diretório, o cliente RAS-CLI deve ser compilado e empacotado. Após o empacotamento, o usuário pode instalar o cliente RAS-CLI no ambiente integrado de análise Galaxy, tornando possível utilizar o serviço de análise por meio deste cliente.

6.5 Considerações finais

Durante esta etapa, definimos uma linguagem específica de domínio para representar um modelo SDDM, facilitando o desenvolvimento desse modelo para

Listagem 17 – Fragmentos do código em `ActDLToGalaxyToolWrapper.kt`

```
8  class ActDLToGalaxyToolWrapper {
9
10     fun generate(activity : Activity, deployment : Deployment) : File {
11
12         val toolWrapper = File.createTempFile("tool",".xml")
13         val xmlWrapper = createXml(activity,deployment)
14         toolWrapper.writeText(xmlWrapper,Charsets.UTF_8)
15
16         return toolWrapper;
17     }
18
19
20     fun createXml(activity : Activity,deployment : Deployment) =
21         """
22         |<?xml version="1.0" encoding="UTF-8"?>
23         |<tool
24         | id="${activity.getName().sanitized()}"
25         | name="${activity.getName()}"
26         | version="${deployment.getService().getApiVersion()}"
27         | profile="16.04"
28         |>
29         | <description>${activity.getRemark()}</description>
30         | <command
31         |   detect_errors="exit_code"
32         | ><![CDATA[
33         | java -jar ${"$"}__tool_directory__/client.jar
34         |        ${activity.getParameters()
35                      .map{it.getCallSyntax()}
36                      .joinToString("\n")}
37         |        ${activity.getInputDatasets()
38                      .map{it.getCallSyntax()}
39                      .joinToString("\n")}
40         |        ${activity.getOutputDatasets()
41                      .map{it.getCallSyntax()}
42                      .joinToString("\n")}
43         | ]]></command>
44         | <inputs>
45         |        ${activity.getParameters()
46                      .map{it.getInputSectionDeclaration()}
47                      .joinToString("\n")}
48         |
49         |        ${activity.getInputDatasets()
50                      .map{it.getInputSectionDeclaration()}
51                      .joinToString("\n")}
52         | </inputs>
53         |
54         | <outputs>
55         |            ${activity.getOutputDatasets()
56                          .map {it.getOutputSectionDeclaration()}
57                          .joinToString("\n")}
58         | </outputs>
59         |
60         | <tests>
61         |        <!-- Include your tests here -->
62         | </tests>
63         |
64         |</tool>
65             """.trimMargin();
```

Fonte: Autoria própria.

Listagem 18 – Fragmentos do código em `GalaxyClientGenerator.kt`

```
8  class GalaxyClientGenerator : ClientGenerator {
9
10
11     override fun generateClient(activity: Activity, deployment: Deployment)
12             : File {
13
14         val javaProject = JavaProject(
15                 JavaProjectGenerator().generate(activity, deployment)
16         )
17
18         val xmlWrapper =
       AadlToGalaxyToolWrapper().generate(activity,deployment)
19
20         javaProject.combineArtifacts(xmlWrapper)
21
22         return javaProject.directory
23     }
24
25
26 }
```

Fonte: Autoria própria.

descrever um conjunto de informações básicas sobre a implantação de um serviço RAS necessárias para a geração tanto de descrições quanto de clientes de um serviço. Em seguida, por meio de metodologias baseadas em modelo, foi possível prover suporte à geração automática de clientes para serviços RAS utilizáveis como ferramentas de linha de comando ou como uma extensão para o ambiente integrado de análise Galaxy. Embora suporte tenha sido provido apenas para a criação de extensões para o ambiente Galaxy, essa mesma metodologia pode ser utilizada para prover suporte para a criação de extensões para outros ambientes de análise, tais como o Taverna [15] e o SemanticSCo [8].

Utilizamos uma abordagem de transformação modelo-para-texto para gerar o código fonte dos clientes RAS-CLI. Uma alternativa à geração de código fonte seria o uso de interpretação de modelos como fizemos para os serviços de análise por meio do *framework* Activity-REST. Para isso, seria necessário implementar uma biblioteca ou *framework* capaz de invocar um serviço RAS corretamente por meio da interpretação dos modelos ActDL e SDDL, bem como incluir as bibliotecas necessárias para a manipulação desses modelos no cliente final. Porém, é interessante que os artefatos que implementam esses clientes sejam pequenos e facilmente executáveis. Adicionalmente, diferente de um serviço RAS cujos *endpoints* podem ser invocados de uma maneira não-sequencial e precisa responder corretamente à essas invocações, um cliente RAS-CLI pode ser mais simples, pois precisa suportar apenas o fluxo base de inicialização, execução e recuperação dos dados de uma atividade de análise. Dessa maneira, a geração de código para esses clientes nos permitiu reduzir o tamanho final do cliente, produzindo apenas o código necessário para suportar esse fluxo base, bem como remover desse cliente as bibliotecas de manipulação dos modelos ActDL e

SDDL.

As transformações modelo-para-texto para a obtenção de clientes RAS foram implementadas utilizando a linguagem Kotlin. Esta linguagem de propósito geral apresenta uma sintaxe declarativa expressiva, particularmente útil para definir composições de blocos de texto. Durante a definição da linguagem para a implementação dessas transformações, consideramos inicialmente linguagens específicas de domínio para a transformação de modelo-para-texto, tais como Acceleo [105] e Xpand [106] e Xtend [107]. Porém, no contexto deste trabalho, Kotlin mostrou-se mais interessante que essas linguagens específicas de domínio para a definição das transformações por apresentar uma curva de aprendizado e uma preparação inicial mais simples que o necessário para o uso das demais linguagens, bem como uma melhor integração com elementos previamente definidos utilizando a linguagem Java.

Em projetos mais complexos e com desenvolvimento iterativo, nos quais seja necessária a reconciliação de uma nova transformação com modificações realizadas nos artefatos produzidos em transformações anteriores, linguagens específicas do domínio de transformação de modelos poderiam ser mais adequadas. Dado que esse caso de uso não foi considerado no nosso trabalho, Kotlin proveu todas as características necessárias para a implementação da transformação modelo-para-texto desenvolvida para a obtenção de artefatos específicos de serviço do cliente RAS.

7 Geração de descrições de serviços RAS

A descrição de um serviço *web* por meio de uma linguagem específica para esta atividade facilita o uso deste serviço, apresentando os *endpoints*, operações e estruturas de dados providos ou utilizados pelo serviço de uma maneira clara. Adicionalmente, aplicações podem fazer uso das informações presentes na descrição de um serviço de modo a filtrar os serviços disponíveis e prover sugestões adequadas de acordo com os interesse de um usuário. Uma descrição de serviço também pode ser utilizada para a obtenção automática de bibliotecas para o auxílio à interação com o serviço para diferentes linguagens de programação [38]. Buscando estender essas vantagens também aos serviços RAS, este capítulo apresenta o suporte à descrição automática de serviços RAS a partir de um modelo AADM e outras informações adicionais.

O presente capítulo está estruturado da seguinte maneira: a Seção 7.1 apresenta uma visão geral dos processos de transformação utilizados para a obtenção de descrições de serviços; a Seção 7.2 apresenta o processo de desenvolvimento de transformações para a obtenção de descrições WSDL para serviços RAS; a Seção 7.3 apresenta o processo de desenvolvimento de transformações para a obtenção de descrições OpenAPI para serviços RAS; a Seção 7.4 apresenta uma plataforma *web* para facilitar a criação de descrições ActDL; modelos de implantação, bem como para a obtenção de serviços e clientes RAS; por fim, a Seção 7.5 apresenta algumas considerações finais sobre essa etapa do trabalho.

7.1 Visão geral

A *interface* de um serviço *web* pode ser descrita por um modelo formal que indica quais os *endpoints* e operações estão disponíveis para serem acessados por meio desse serviço, bem como as requisições e respostas esperadas para esses *endpoints* e operações. Assim, uma descrição formal de um serviço *web* é uma fonte de conhecimento que facilita a compreensão desse um serviço por desenvolvedores interessados em integrá-lo em suas aplicações. Adicionalmente, uma descrição formal de um serviço *web* pode ser interpretada computacionalmente. A interpretação computacional de uma descrição de serviço possibilita o uso de métodos e tecnologias para obter bibliotecas que abstraiam as chamadas e o processamento das respostas a esses serviços, facilitando seu uso em outros programas de uma dada linguagem de programação.

Metodologias e bibliotecas estão disponíveis para a obtenção automática de

descrições de serviços *web* a partir das classes Java que implementa esses serviços. Por exemplo, Guardia *et al.* [7] utilizam o *framework* Axis2/Java [108] para inspecionar o conjunto de classes que compõe o serviço, analisar as anotações JAX-RS, parâmetros e o tipo de retorno dos métodos contidos nessas classes e produzir uma especificação WSDL descrevendo os *endpoints* encontrados. Esta especificação WSDL é, então, anotada semanticamente com termos de uma ontologia biomédica e utilizada como um dos modelo necessários para a integração de um serviço com o ambiente SemanticSCo [8]. Metodologias e bibliotecas semelhantes são disponíveis para a obtenção de descrições OpenAPI de um serviço JAX-RS.

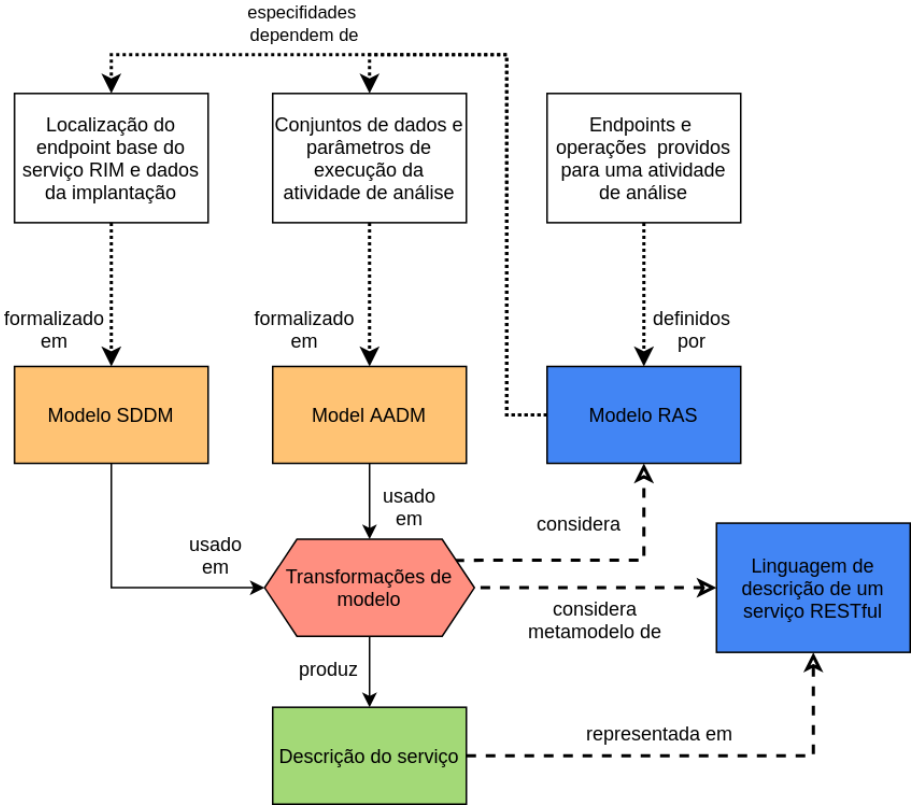
Metodologias baseadas na anotação de classes Java para a obtenção de descrição de um serviço de análise não podem ser aplicadas satisfatoriamente para um serviço implementado por meio do *framework* Activity-REST, uma vez que grande parte da funcionalidade provida por este *framework* é implementado como um conjunto de classes Java genéricas configuradas em tempo de execução. Desse modo, a tentativa de utilizar a mesma abordagem para a obtenção de uma descrição de um serviço RAS resultaria, também, em descrições genéricas e pouco expressivas.

Dado que as classes do *framework* Activity-REST baseiam-se em um modelo AADM interpretado em tempo de execução de modo a modificar automaticamente o comportamento e a interface do serviço segundo o modelo de referência RAS, podemos definir uma metodologia para a obtenção da descrição de um serviço RAS a partir desses modelos. Neste sentido, metodologias de desenvolvimento baseado em modelos também podem ser utilizadas para obter os artefatos que descrevem um serviço de análise a partir de um conjunto de modelos abstratos iniciais. Porém, apenas as informações contidas em um modelo AADM e as diretrizes abstratas do modelo de referência RAS não são capazes de prover todas a informação necessária para a descrição de um serviço de análise. Notadamente, esses modelos não provêem informações sobre a implantação e localização final do serviço. Dessa maneira, é necessário que essas informações sejam providas de forma complementar. Neste trabalho, utilizamos um modelo SDDM para prover tais informações.

A Figura 51 apresenta uma visão geral do processo de transformação de um modelo SDDM e de um modelo AADM em uma descrição do serviço RAS correspondente. O modelo SDDM provê a localização do *endpoint* base do serviço, bem como outros dados sobre a implantação necessários para a descrição do serviço, tais como a sua versão e o contato do mantenedor desse serviço. O modelo AADM define os conjuntos de dados e parâmetros de execução da atividade de análise executada pelo serviço RAS. Ambos modelos são utilizados como fonte da transformação. A transformação é definida com base no modelo de referência RAS e em uma dada linguagem de descrição de serviços. Nesse sentido, o modelo de referência RAS e o modelo AADM definem os *endpoints* e as operações disponíveis no serviço RAS,

enquanto a linguagem de descrição de serviços *web* alvo determina quais elementos devem ser produzidos ao final da transformação.

Figura 51 – Visão geral da transformação de um modelo SDDM e um modelo AADM em uma descrição de um serviço RAS.



Fonte: Autoria própria.

De modo a definir um conjunto de regras para a obtenção de uma dada descrição de serviço, primeiramente realizamos um estudo de caso utilizando um serviço RAS já em funcionamento. Neste sentido, definimos iterativamente a descrição desse serviço na linguagem alvo, obtendo o conhecimento necessário para a definição de regras de transformação abstratas que permitem obter os elementos presentes na descrição do serviço a partir dos elementos dos modelos iniciais.

A cada iteração, selecionamos um novo *endpoint* do serviço e definimos os elementos que o descrevem quanto a suas operações, bem como quanto as representações utilizadas em requisições e respostas a esse *endpoint*. Essa definição foi realizada em duas etapas: a definição das representações utilizadas em requisições e respostas ao *endpoint* e a definição da representação do *endpoint* e das operações providas por ele. Então, definimos a regra de transformação abstrata para obter a descrição daquele *endpoint* em linguagem natural. Em seguida, aplicamos as regras de transformação obtidas manualmente para recursos relacionados. Por exemplo, utilizamos a regras extraída durante a especificação do recurso que manipula um dado parâmetro

da atividade de análise para a especificação dos recursos para a manipulação dos demais parâmetros. Por fim, comparamos a especificação do serviço obtida com o comportamento provido pelo serviço. Repetimos o processo até a obtenção de uma especificação contemplando todos os recursos necessários à execução da atividade de análise.

Após a definição das regras de transformação abstratas, concretizamos essas regras de transformação de modo a poder executá-las automaticamente para um conjunto de modelos iniciais. Inicialmente, escolhemos um conjunto de tecnologias a serem utilizadas durante essa concretização. Neste sentido, usamos diferentes tecnologias para a obtenção de descrições WSDL e OpenAPI. Para o obtenção de descrições WSDL, utilizamos a linguagem de programação Kotlin [101] para implementar transformações modelo-para-texto. Por sua vez, para a obtenção de descrições OpenAPI, aproveitamos a disponibilidade de um metamodelo baseado em Ecore disponível para essa linguagem [109] e concretizamos as regras de transformação utilizando a linguagem de transformação modelo-para-modelo ATL [110]. Em seguida, implementamos as regras de transformação abstratas por meio dessas tecnologias, obtendo uma transformação executável. Por fim, validamos o processo de transformação por meio da geração automática de bibliotecas para a interação com serviços descritos pelos modelos iniciais, bem como pelo uso dessas bibliotecas em um programa que interage com esses serviços.

7.2 Geração de descrições WSDL

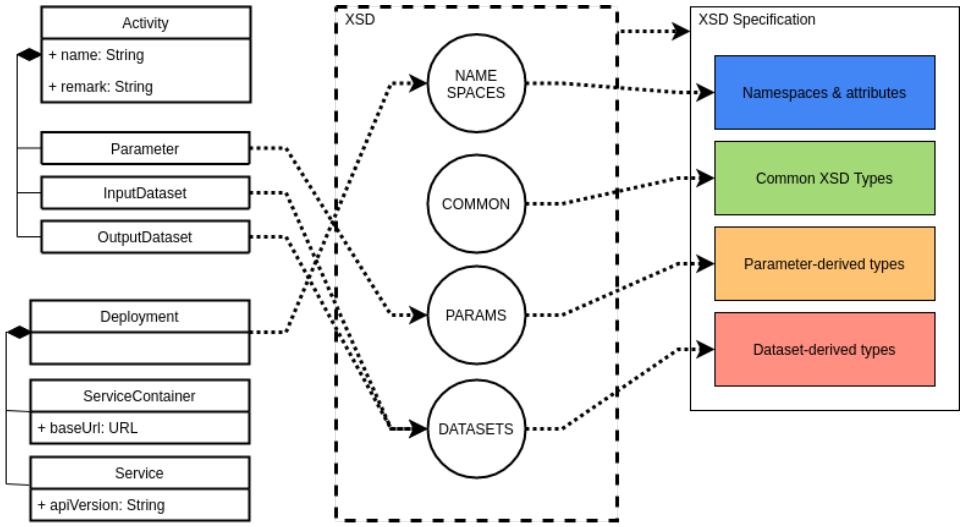
7.2.1 Definição das regras de transformação abstratas

O processo de especificação de um serviço por meio de WSDL pode ser definido em duas fases. Inicialmente, um documento XSD é definido. Este documento contém a estrutura das mensagens trocadas entre cliente e serviço. Em seguida, um documento WSDL é definido. Tal documento importa ou inclui os elementos presentes no documento XSD, referenciando-os nos elementos que definem as operações providas pelo serviço.

A Figura 52 apresenta uma visão geral das regras de transformação para a obtenção de uma especificação XSD. Os diferentes elementos presentes em uma especificação XSD de serviço RAS podem ser agrupados segundo algumas diferenças no processo de definição desses elementos. Essa agregação cria uma divisão abstrata desses elementos que facilita a definição do conjunto de regras de transformação. Primeiramente, há o elemento `<xs:schema>`, os *namespaces* e atributos declarados. Este elemento é produzido utilizando informações derivadas principalmente do modelo SDDM. Em seguida, há um conjunto de elementos que declaram tipos XSD genéricos

para todos os serviços RAS. Esses tipos genéricos são relacionados aos recursos manipulados de maneira semelhante em todos estes serviços ou representam tipos reutilizados para a definição de elementos específicos de um serviço. Por fim, há dois conjuntos de elementos relacionados às definições de parâmetros e de conjuntos de dados do serviço. Estes elementos precisam ser definidos de maneira específica para cada serviço por meio do uso ou extensão dos elementos do conjunto comum.

Figura 52 – Visão geral das regras de transformação para obtenção de uma especificação XSD.



Fonte: Autoria própria. Os elementos de um modelo AADM e de um modelo SDDM à esquerda e a visão abstrata de uma especificação XSD para um serviço RAS à direita. Regras de transformação são representadas como círculos nomeados. Uma flecha conectando um elemento a uma regra de transformação representa que esse elemento é usado como entrada para a regra de transformação. Uma flecha conectando uma regra de transformação a um elemento representa que esse elemento é criado durante a execução da regra de transformação.

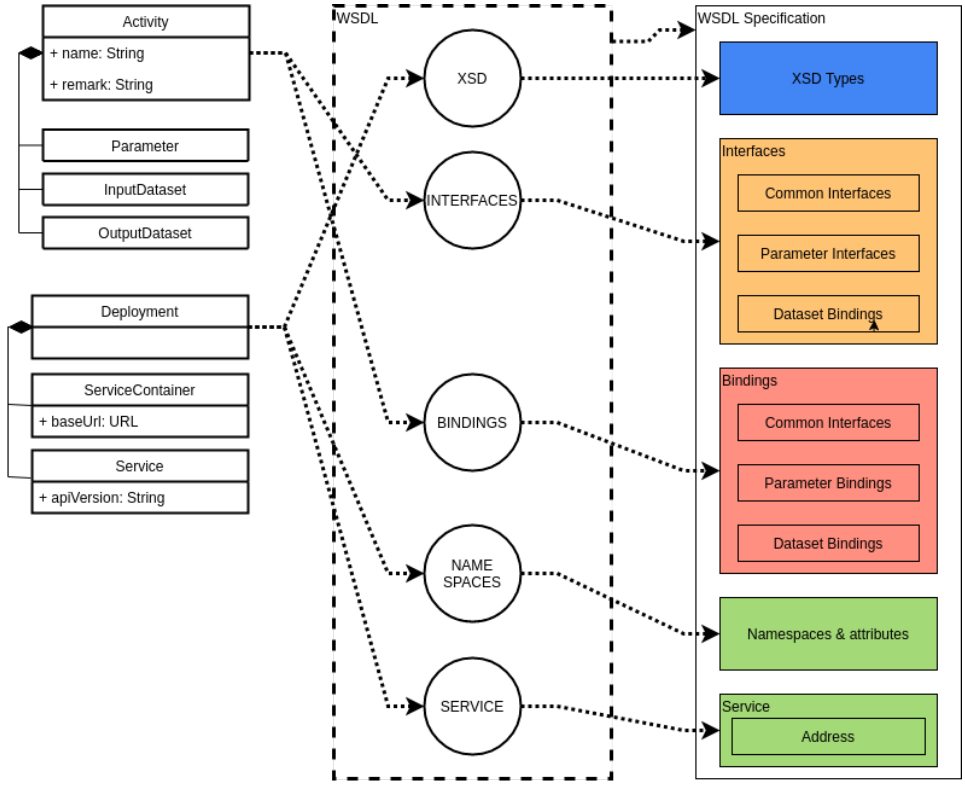
A regra XSD é a regra mais geral dessa transformação. XSD é responsável por criar o documento XSD e seu elemento raiz (elemento `<xsd:schema>`). XSD invoca as demais regras da transformação e combina seus resultados. A regra NAMESPACES define os *namespaces* utilizados no documento XSD e incluídos como atributos do elemento `<xsd:schema>`. A regra COMMON adiciona o conjunto de elementos do documento XSD compartilhado entre todas as descrições de serviços RAS. As regras PARAMS e DATASETS definem os elementos que especificam a representação de parâmetros de execução e conjunto de dados, respectivamente, nas requisições e respostas HTTP ao serviço RAS.

O Apêndice G apresenta em maiores detalhes as regras de transformação abstratas para a obtenção de uma descrição XSD.

Assim como os elementos da especificação XSD, os diferentes elementos

presentes em uma especificação WSDL de um serviço RAS também podem ser agrupados segundo algumas diferenças no processo de definição desses elementos. A Figura 53 apresenta uma visão geral de um conjunto de regras de transformação para a obtenção de uma especificação WSDL.

Figura 53 – Visão geral das regras de transformação para obtenção de uma especificação WSDL.



Fonte: Autoria própria. Metaclasses de interesse do modelo AADM e do modelo SDDM são apresentadas à esquerda. Elementos de uma especificação WSDL para um serviço RAS são apresentados à direita. Regras de transformação são representadas como círculos nomeados. O retângulo pontilhado ao centro representa uma regra de transformação mais geral, a qual invoca as demais regras de transformação, combinando seus resultados. Uma flecha conectando um elemento a uma regra de transformação representa que esse elemento é usado como entrada para a regra de transformação, enquanto uma flecha conectando uma regra de transformação a um elemento representa que esse elemento é criado durante a execução da regra de transformação.

Primeiramente, há o elemento `<wsdl:definition>`, raiz dos demais elementos da especificação. Os *namespaces* e atributos que `<wsdl:definition>` define são derivados principalmente do modelo SDDM. Em seguida, há a importação dos tipos XSD definidos anteriormente. Elementos que representam operações abstratas e *bindings* podem ser agrupados em três sub-conjuntos: elementos comuns para todos os serviços RAS, elementos derivados da definição dos parâmetros da atividade de análise e elementos derivados da definição dos conjuntos de dados da atividade de

análise. Por fim, a definição do elemento `<wsdl:service>` e de seus elementos é derivada principalmente do modelo SDDM.

A regra de transformação WSDL invoca as regras XSD, INTERFACES, BINDINGS, NAMESPACES e SERVICE. Durante essa invocação, WSDL repassa a essas regras os elementos dos modelos iniciais necessários à sua execução, combinando os resultados obtidos de modo a obter todo o conteúdo da descrição WSDL do serviço.

A regra XSD produz uma declaração de importação dos elementos definidos pela especificação XSD produzida na transformação anterior. Esta regra necessita apenas do modelo SDDM como entrada, uma vez que esse modelo provê os dados necessários para definir o *namespace* da descrição XSD importada. O modelo SDDM também é o único modelo fonte para as regras NAMESPACES e SERVICE, que definem os *namespaces* utilizados na descrição WSDL produzida e um elemento `<wsdl:service>` contendo o endereço do serviço, respectivamente.

As regras INTERFACES e BINDINGS definem, respectivamente, os elementos `<wsdl:interface>` e `<wsdl:bindings>` da descrição WSDL, respectivamente. Esses elementos definem as operações realizáveis no serviço, bem como os *endpoints* que provêm essas operações. Dessa maneira, ambas as regras dependem dos elementos do modelo AADM que, em conjunto com o modelo de referência RAS, definem as características estruturais do serviço. INTERFACES e BINDINGS utilizam regras específicas para a criação de elementos mais específicos de `<wsdl:interface>` e `<wsdl:bindings>`.

O Apêndice I apresenta em maiores detalhes as regras de transformação abstratas para a obtenção de uma descrição WSDL.

7.2.2 Implementação da transformação

Esta etapa consistiu da implementação, utilizando a linguagem de programação Kotlin, das regras abstratas de transformação para a obtenção de uma descrição WSDL a partir de um modelo SDDM e um modelo AADM. A Listagem 19 apresenta um fragmento do código que implementa a transformação para a obtenção da descrição XSD do tipos de dados utilizados pelo serviço RAS. Esta listagem apresenta a regra XSD, modelada como uma função que recebe um modelo AADM e um modelo SDDM e que retorna o conteúdo da descrição XSD dos tipos de dados do serviço RAS (linha 17). A função cria um elemento `<xs:schema>` (linhas 19 a 37) e invoca outras regras de transformação para criar detalhes e elementos dessa descrição.

Quatro regras são invocadas em sequência. Inicialmente, a regra NAMESPACES é invocada para definir os namespaces da descrição XSD. Esses namespaces são incluídos como atributos do elemento `<xs:schema>` (linha 20). Então, é invocada

Listagem 19 – Fragmento da transformação para a obtenção de uma descrição XSD.

```
17 fun toXsd(activity: Activity, deploymentModel: Deployment): String {
18   return """<?xml version="1.0" encoding="UTF-8"?>
19       <xs:schema
20                 ${getNamespaces(deploymentModel)}
21          >
22          ${xsdCommonElements()}
23
24          ${getParams(activity)}
25
26          ${getDatasets(activity.getInputDatasets())}
27
28          ${getDatasets(activity.getOutputDatasets())}
29
30       </xs:schema>
31       """
32 }
```

Fonte: Autoria própria.

a regra COMMON para a inclusão dos elementos comuns a todas as descrições XSD de serviços RAS (linha 22). Em seguida, a regra PARAMS é invocada para cada parâmetro de execução definido no modelo AADM (linha 24). Por fim, a regra DATASETS é invocada para os conjuntos de dados de entrada e para os conjuntos de dados de saída (linhas 26 e 28, respectivamente). A regra XSD compõe o resultado dessa regras e retorna o conteúdo da descrição XSD como uma *string*.

A Listagem 20 apresenta a implementação da regra WSDL. Essa regra é modelada como uma função que recebe um modelo AADM e um modelo SDDM e retorna o conteúdo da descrição WSDL para o serviço RAS associado (linha 17). A função cria um elemento `<wsdl:description>` e invoca outras regras de transformação para criar detalhes e elementos dessa descrição. Nesse sentido, os namespaces da descrição WSDL são criados a partir do modelo SDDM pela regra NAMESPACES (linha 20) e incluídos como atributos XML em `<wsdl:description>`. Então, as regras XSD, INTERFACES, BINDINGS e SERVICE são invocadas em sequência para a obtenção dos elementos de importação da descrição XSD dos tipos de dados utilizados pelo serviço, o elemento `<wsdl:interface>`, o elemento `<wsdl:bindings>` e o elemento `<wsdl:service>` (linhas 24, 26, 28 e 30, respectivamente). A regra WSDL compõe os resultados parciais das demais regras e retorna o conteúdo da descrição WSDL como uma *string*.

Após a implementação da transformação para a obtenção das descrições XSD e WSDL para o serviço RAS, incluímos essa transformação como parte do *framework* Activity-REST. Dessa maneira, um serviço RAS implementado com o *framework* Activity-REST pode prover descrições XSD e WSDL que descrevem a estrutura desse serviço em relação aos *endpoints* necessários para a execução de uma instância de atividade de análise. Para isso, torna-se apenas necessário que o desenvolvedor desse serviço também defina um modelo SDDM para o serviço e associe esse modelo

Listagem 20 – Fragmento da transformação para a obtenção de uma descrição WSDL.

```
17 fun toWsdl(activity : Activity, deploymentModel : Deployment) : String =
18    """<?xml version="1.0" encoding="UTF-8"?>
19     <wsdl:description
20             ${getNamespaces(deploymentModel)}
21         >
22
23
24         ${getXsdImport(deploymentModel)}
25
26         ${getWsdlInterface(activity)}
27
28         ${getWsdlBindings(activity)}
29
30         ${getWsdlService(deploymentModel)}
31
32     </wsdl:description>
33    """
```

Fonte: Autoria própria.

SDDM durante a criação do componente `Configuration Code` do novo serviço RAS. As descrições XSD e WSDL serão, então, disponibilizadas nos *endpoints* `/xsd` e `/wsdl` do novo serviço, respectivamente.

7.2.3 Validação da descrição WSDL obtida por meio da transformação

De modo a validar a especificação WSDL definida, utilizamos o *framework* Axis2 para a verificar sintaticamente essa especificação e produzir automaticamente código *stub* para clientes do serviço. Axis2 consiste de um *framework* Java para o suporte ao desenvolvimento e uso de serviços *Web* documentados por meio de descrições WSDL [108]. Utilizando o código *stub* produzido pelo Axis2, desenvolvemos testes automatizados para a execução do serviço implementado, bem como para a verificação da estrutura das mensagens trocadas entre cliente e serviço. Por meio desses testes, verificamos a conformidade entre o serviço implementado e a especificação definida.

Durante o processo de definição e validação da especificação WSDL do serviço, encontramos algumas limitações relacionadas ao uso de WSDL para a anotação de serviços RAS e ao uso do *framework* Axis2 para a sua validação. Embora a especificação da linguagem WSDL permita a descrição de *endpoints* RESTful que consumam ou produzam representações em linguagens diferentes do XML, o *framework* Axis2 produz automaticamente apenas clientes capazes de lidar com requisições e respostas HTTP contendo representações em XML dos recursos manipulados. Dessa maneira, foi necessário adicionar um conjunto de *MessageBodyProviders* de modo a prover suporte à representação em XML para os recursos disponibilizados pelo *framework* Activity-REST previamente à validação da descrição do serviço utilizando o *framework* Axis2. Adicionalmente, foi preciso

encapsular o conteúdo dos conjuntos de dados em mensagens XML quando do envio ou recepção de arquivos contendo esses conjuntos de dados.

7.3 Geração de descrições OpenAPI para serviços RAS

7.3.1 Definição abstrata das regras de transformação

Esta etapa consistiu da definição de um conjunto de regras abstratas para a transformação de um modelo AADM e um modelo SDDM em uma descrição de serviço OpenAPI. A Figura 54 apresenta uma visão geral de um subconjunto das regras de transformação definidas nessa etapa. A regra de transformação API é responsável por criar um elemento `API` do metamodelo OpenAPI. Este elemento receberá todos os demais elementos da descrição OpenAPI produzidos pelas demais regras.

A regra INFO recebe uma instância de `Activity` e uma instância de `Deployment` de modo a definir o elemento `INFO` da descrição do serviço. Essa regra é a única que precisa das informações do modelo SDDM para a sua execução, necessitando de uma instância de `Deployment` para definir informações como o endereço base do serviço, sua versão e o contato da pessoa responsável pela manutenção do mesmo.

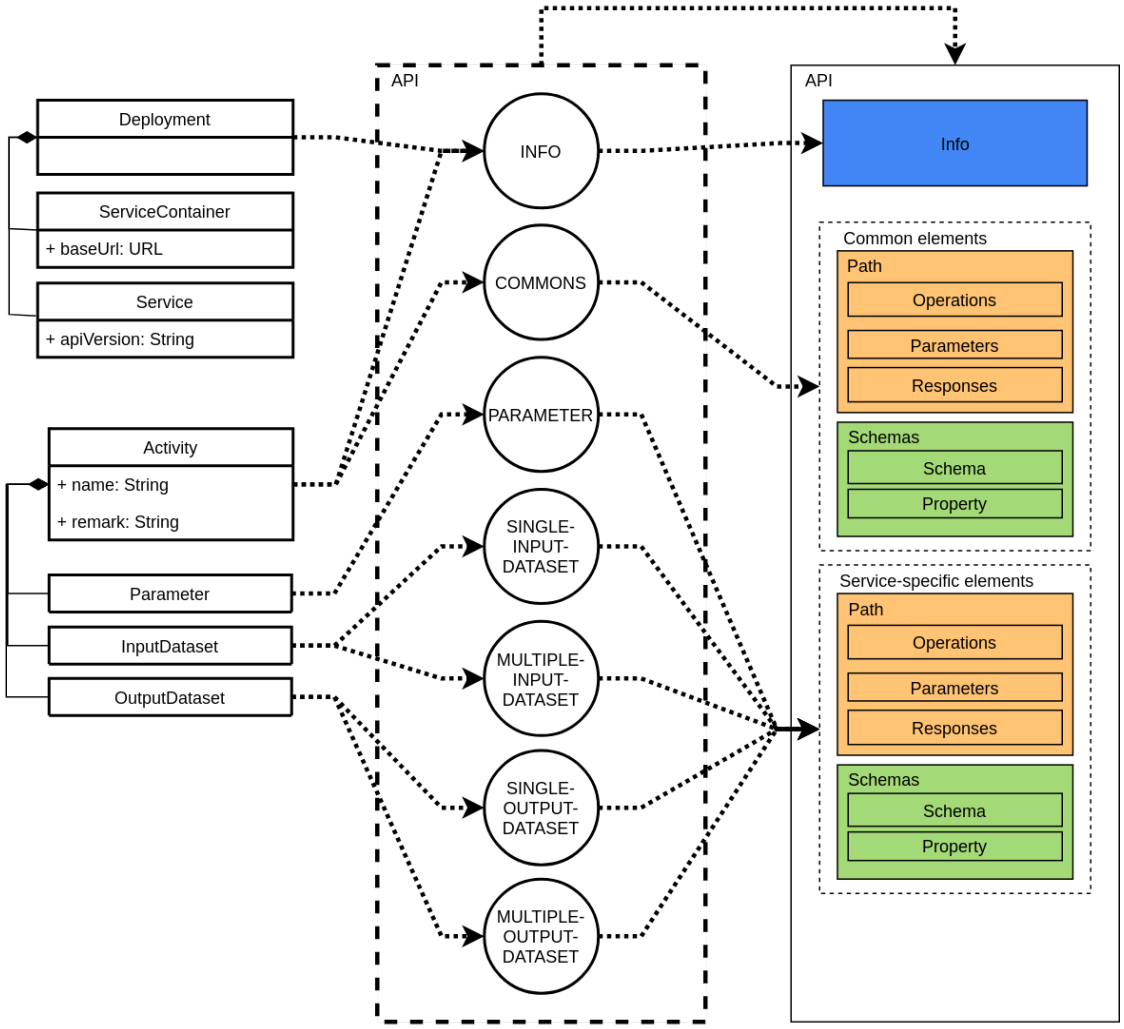
Os elementos presentes nos demais componentes de uma descrição OpenAPI são responsáveis pelo endereçamento relativo dos recursos, pela definição de operações e dos tipos de dados utilizados pelo serviço RAS. Assim como os *endpoints* e operações de um serviço RAS, estes elementos também podem ser divididos entre elementos comuns a todos os serviços RAS e elementos específicos de um dado serviço. Cada um desses conjuntos de elementos inclui instâncias de `Path`, `Operation`, `Responses` e `Schemas`, bem como elementos relacionados, necessários para representar os *endpoints* e operações do serviço para uma dada instância de atividade de análise em todos os estados de seu ciclo de vida. Dessa maneira, a regra de transformação COMMONS é responsável por criar os elementos do conjunto comum a todos os serviços RAS. Por sua vez, as regras restantes recebem elementos que definem os detalhes de uma atividade de análise e conjuntamente produzem os elementos específicos do serviço RAS associado ao modelo AADM usado como fonte para a transformação.

O Apêndice K apresenta as regras abstratas de transformação definidas nessa etapa.

7.3.2 Implementação da transformação modelo-para-modelo

Nesta etapa, implementamos as regras de transformação abstratas definidas anteriormente de maneira a permitir sua execução automática. Durante essa imple-

Figura 54 – Visão geral da transformação modelo-para-modelo para a obtenção de uma descrição OpenAPI.



Fonte: Autoria própria. Metaclasses de interesse do modelo AADM e do modelo SDDM são apresentadas à esquerda. Elementos de uma especificação OpenAPI para um serviço RAS são apresentados à direita. Regras de transformação são representadas como círculos nomeados. O retângulo pontilhado ao centro representa uma regra de transformação mais geral, a qual invoca as demais regras de transformação, combinando seus resultados. Uma flecha conectando um elemento a uma regra de transformação representa que esse elemento é usado como entrada para a regra de transformação, enquanto uma flecha conectando uma regra de transformação a um elemento representa que esse elemento é criado durante a execução da regra de transformação.

mentação, utilizamos o suporte do OpenAPI-Metamodel [109] para a representação de uma descrição OpenAPI versão 2 como um modelo Ecore e para a serialização desse modelo Ecore em uma especificação OpenAPI em JSON. Uma vez que os metamodelos de um modelo AADM e de um modelo SDDM também são definidos utilizando o meta-metamodelo Ecore como base, decidimos utilizar um *framework* de transformação de modelos para a concretização das regras de transformação definidas.

Neste sentido, utilizamos o *framework* de transformação de modelos ATL [110] para essa concretização. ATL permite a definição de transformações de modelos Ecore utilizando uma linguagem específica de domínio que busca permitir a concretização das regras de transformação de maneira declarativa.

A Listagem 21 apresenta um fragmento da implementação da transformação em ATL. Esta listagem apresenta a regra ATL `OutputDatasetSingleFile`, que implementa a regra SINGLE-OUTPUT-DATASET, responsável por criar os elementos necessários para a recuperação de um arquivo em um conjunto de dados de saída de cardinalidade unitária.

Listagem 21 – Fragmento da transformação para a obtenção de uma descrição OpenAPI de um serviço RAS

```
1002 rule OutputDatasetSingleFile {
1003   from
1004     dataset: aadl!OutputDataset ( 1 = dataset.maximumCardinality)
1005
1006   to
1007
1008     succeededAnalysisPath: openapi!Path (
1009       relativePath <- '/succeeded-instances/{id}/outputs/' + dataset.name,
1010       parameters <- thisModule.resolveTemp(
1011             dataset.refImmediateComposite(),'activityIdentifierParameter'
1012       ),
1013       get <- getInSuccededActivity
1014     ),
1015     getInSuccededActivity : openapi!Operation (
1016       operationId <- 'get-' + dataset.name + '-from-succeded-analysis',
1017       description <- dataset.remark,
1018       parameters <- thisModule.resolveTemp(
1019             dataset.refImmediateComposite(),'activityIdentifierParameter'
1020       ),
1021       responses <- okResponse
1022     ),
1023     okResponse: openapi!Response (
1024       code <- '200',
1025       description <- 'Dataset value is retrieved',
1026       headers <- thisModule.getLocationHeader(),
1027       headers <- thisModule.getLinkHeader(),
1028       schema <- thisModule.resolveTemp(
1029             dataset.refImmediateComposite(),'fileDefinition'
1030       )
1031     )
1032 }
```

Fonte: Autoria própria.

A declaração da regra é realizada na linha 1002. Duas seções da declaração de uma regra ATL são apresentadas, `from` (linhas 1003 a 1004) e `to` (linhas 1006 a 1031). A seção `from` define o subconjunto dos elementos dos modelos iniciais no qual a regra deve ser aplicada. Neste sentido, a regra `OutputDatasetSingleFile` é aplicável a elementos `OutputDataset` do modelo AADM, porém limitada aos elementos que definem a cardinalidade máxima do conjunto de dados tendo o valor de 1 arquivo (linha 1004).

A seção `to` declara os elementos a serem produzidos no modelo final (a

descrição OpenAPI) após a transformação, assim como a maneira que os atributos desses elementos devem ser definidos. O fragmento apresenta que a regra cria três elementos: um elemento `Path` nomeado `succedAnalysisPath` (linha 1008), um elemento `Operation` nomeado `getInSucceededActivity` (linha 1015) e um elemento `Response` nomeado `okResponse` (linha 1023). Valores para atributos desses elementos são definidos nos blocos delimitados por parênteses que seguem as suas declarações por meio dos valores do elemento `OutputDataset` inicial, referências a outros elementos criados por essa mesma regra, ou elementos criados por outras regras. Por exemplo, o elemento `Path` deve ter seu parâmetro `relativePath` definido como uma *string* que contém o valor do atributo `name` do elemento `OutputDataset` (linha 1009). O atributo `parameters` desse mesmo elemento deve ser definido por meio da coleta do elemento nomeado `activityIdentifierParameter` produzido na regra aplicada ao elemento que contém o `OutputDataset` em questão (linhas 1010 a 1012). Por fim, o valor do atributo `get` do elemento `Path` produzido deve ser definido como o elemento `getInSucceededActivity` definido nessa mesma regra (linha 1013). Os demais elementos são definidos de maneira semelhante.

Após a definição da transformação utilizando ATL, utilizamos este *framework* para implementar uma ferramenta de linha de comando, chamada `ras-to-openapi` capaz de executar a transformação automaticamente sobre modelos iniciais informados. O código dessa ferramenta e a definição da transformação na linguagem ATL estão disponíveis no repositório GIT do *framework* Activity-REST[1].

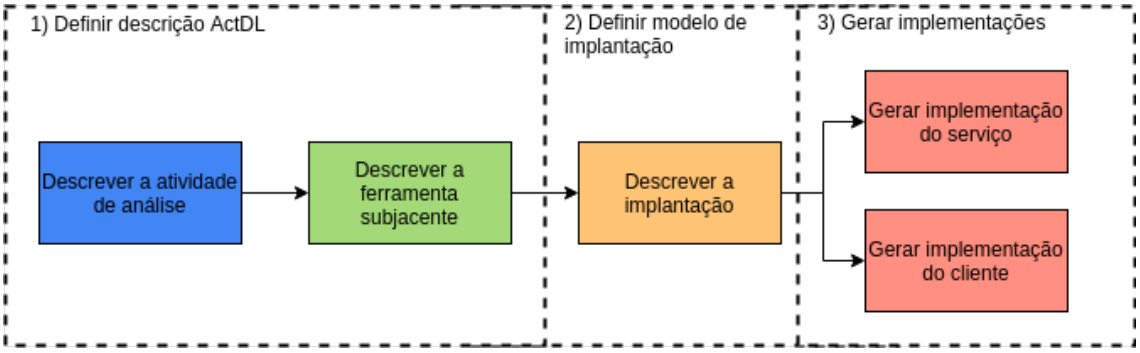
7.4 Activity-REST Boot

De modo a facilitar uso do *framework* Activity-REST para a criação de serviços e clientes de análise RAS, desenvolvemos a aplicação *web* Activity-REST Boot[2]. Essa aplicação integra todas as ferramentas desenvolvidas no trabalho, provê um *wizard* que possibilita a um especialista de domínio descrever uma ferramenta de análise de maneira facilitada, guiando-o durante a definição da descrição ActDL e do modelo de implantação. De maneira transparente ao usuário, projetos Maven para um serviço ou cliente RAS são criados no servidor desta aplicação *web*, utilizando tanto os arquétipos Maven definidos quanto executando as transformações de modelo-para-texto necessárias. Ao final do processo, o especialista de domínio pode obter tanto um serviço RAS quanto um cliente RAS implementados e prontos para o empacotamento e distribuição, bem como descrições para o novo serviço de análise.

A Figura 55 apresenta os passos para a obtenção de clientes e serviços RAS por meio do sistema Activity-REST Boot. Inicialmente, o usuário define a descrição

[1] `https://purl.org/cawal/lssb/activity-rest-framework`
[2] `https://purl.org/cawal/lssb/activity-rest-bootstrap`

Figura 55 – Passos para a obtenção de clientes e serviços RAS.



Fonte: Autoria própria.

ActDL por meio de duas visões: uma visão focada na definição da atividade de análise e uma visão focada na definição da ferramenta de análise subjacente ao serviço. Após definir a descrição ActDL, o usuário define o modelo de implantação informando, entre outras coisas, o URL base do serviço e informações de contato do seu mantenedor. Por fim, o usuário submete ambos os modelos para processamento e recebe como resposta os arquivos compactados do projeto, prontos para serem compilados e executados.

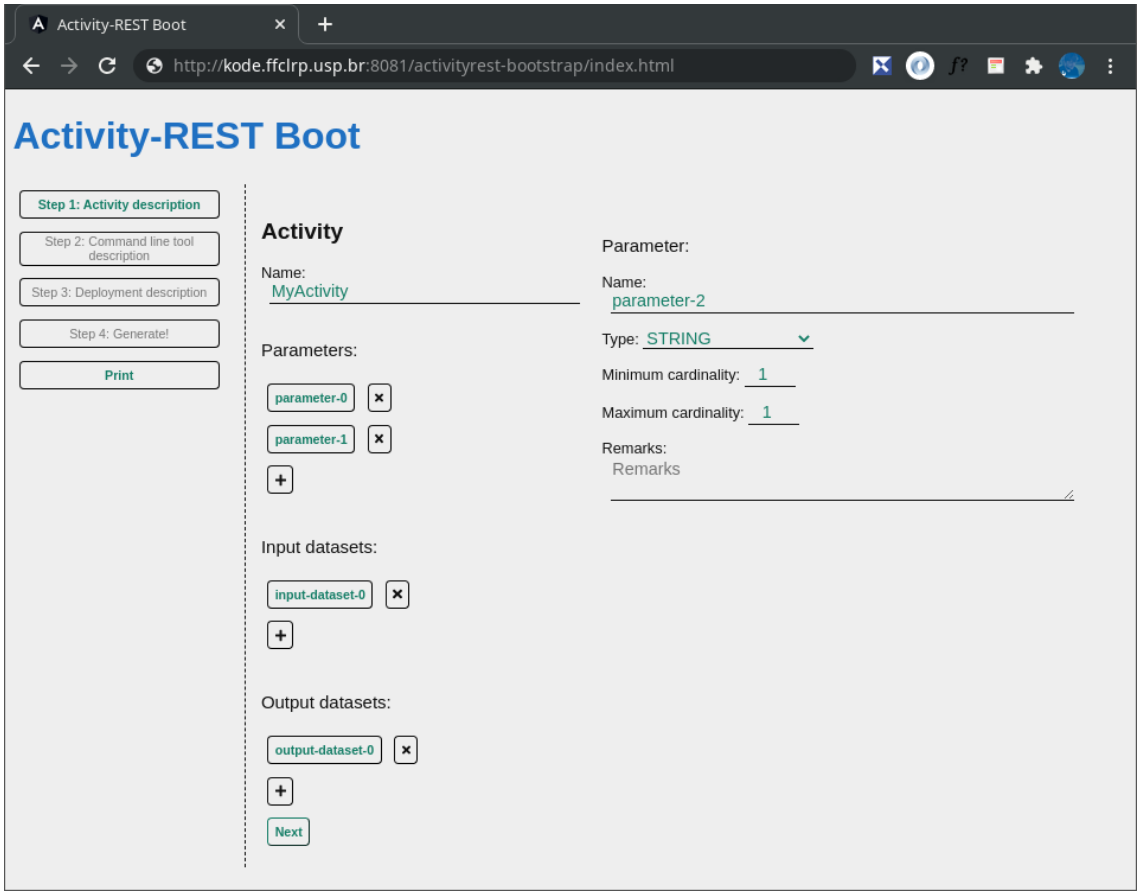
A Figura 56 apresenta a interface *web* para a definição de uma atividade de análise. De acordo com essa interface, o especialista de domínio descreve os parâmetros de execução e os conjuntos de dados de entrada e saída da atividade de análise. Após realizar a descrição dos parâmetros de execução e conjuntos de dados, o usuário pode iniciar a etapa de descrição da ferramenta de análise.

7.5 Considerações finais

Nesta etapa, implementamos transformações de modelos que nos permitiram obter descrições WSDL e OpenAPI para um serviço RAS a partir de um modelo AADM e um modelo SDDM. Utilizamos duas diferentes abordagens durante a implementação das transformações utilizadas para obter descrições de serviço, nomeadamente transformações modelo-para-texto e modelo-para-modelo.

A transformação que produz uma descrição WSDL foi implementada em Kotlin como uma transformação modelo-para-texto que utiliza interpolação de *strings* para produzir o documento WSDL representado em XML. Essa transformação foi desenvolvida apenas usando o suporte das bibliotecas padrões das linguagens Kotlin e Java, bem como o suporte para a leitura e manipulação de modelos ActDL e SDDL desenvolvidos em nosso trabalho. O objetivo final da transformação foi obter um documento WSDL sintaticamente válido contendo todos os seus elementos corretamente referenciados.

Figura 56 – Interface *web* para a definição de uma atividade de análise.



Fonte: Autoria própria.

Por sua vez, a transformação que produz uma descrição OpenAPI foi implementada como uma transformação modelo-para-modelo utilizando o *framework* ATL. Previamente à transformação em si, o suporte para leitura e manipulação de modelos ActDL e SDDL é utilizado para carregar esses modelos em memória. O *framework* ATL é, então, invocado para executar a transformação de modelos implementada por meio da linguagem específica sobre esses modelos em memória, produzindo um modelo OpenAPI, representado por meio de um metamodelo Ecore existente [109]. Finalmente, o modelo OpenAPI é serializado em JSON por meio do suporte a essa serialização provido em conjunto com o metamodelo Ecore.

Inicialmente, ambas transformações para obtenção de descrições WSDL e OpenAPI seriam realizadas usando o *framework* ATL. Porém, este *framework* necessita que tanto os modelos fontes e os modelos alvos da transformação sejam representados utilizando o meta-metamodelo Ecore [75]. Embora o metamodelo OpenAPI Metamodel tenha sido implementado utilizando Ecore e estivesse disponível, não havia um metamodelo WSDL 2.0 em Ecore já implementado. De modo a contornar essa limitação, poderíamos implementar um metamodelo WSDL 2.0 mínimo, procurar outros *frameworks* de transformação de modelos ou utilizar diretamente

uma transformação modelo-para-texto. Concluímos que, embora a estrutura geral de uma descrição WSDL seja simples, o metamodelo inerente a ela é complexo e teria uma implementação custosa. Neste sentido, o uso de uma transformação modelo-para-texto mostrou-se uma solução pragmática no contexto deste trabalho.

Em seguida, consideramos um conjunto de opções de *frameworks* e linguagens existentes com suporte à transformação modelo-para-texto com modelos baseados em Ecore como fontes da transformação. Ao final, percebemos que todas as operações que seriam necessárias para esta transformação poderiam ser representadas em Kotlin com semelhante nível de abstração e declaratividade que seriam obtidas por meio das linguagens específicas disponíveis, dado que operações mais complexas, como a reconciliação de mudanças realizadas no artefato de texto criado em novas re-gerações desse artefato, não seriam necessárias para nosso uso.

A escolha de Kotlin para a implementação da transformação modelo para texto permitiu não depender de *frameworks* dedicados apenas para essa tarefa, removendo dificuldades inerentes à inicialização e o uso desses *frameworks*, os quais por vezes são mal documentados. Adicionalmente, o uso de uma linguagem de propósito geral possibilitou reutilizar o conhecimento e o suporte disponível para o desenvolvimento usando as linguagens Java e Kotlin, os quais são superiores ao suporte existente para outras linguagens consideradas durante essa tarefa. Assim, o uso de linguagens de propósito geral para a implementação de transformações modelo-para-texto revelou-se uma solução pragmática e facilmente acessível.

8 Reengenharia dos serviços do repositório GEAS

O repositório GEAS disponibiliza um conjunto de serviços para a execução de atividades de análise de expressão gênica, como a normalização de conjuntos de dados e a identificação de genes diferencialmente expressos. Os serviços GEAS foram implementados como adaptadores para um conjunto de ferramentas de análise de linha de comando. A implementação destes serviços foi realizada de maneira tradicional e não foi baseada em um modelo de interação bem definido, de modo que cada serviço GEAS apresenta pequenas diferenças no endereçamento de *endpoints* e na ordem de realização das operações sobre esses *endpoints*. Adicionalmente, a implementação dos serviços GEAS não apresenta separação clara entre o componente da interface RESTful, o formatador da chamada de linha de comando e o gerenciador de trabalhos.

Este capítulo apresenta uma visão geral da reengenharia dos serviços de análise disponibilizados pelo repositório GEAS usando o *framework* Activity-REST. O restante do capítulo está estruturado como segue: a seção 8.1 apresenta uma visão geral do repositório GEAS e dos serviços contidos nesse repositório; a seção 8.2 apresenta uma visão geral do processo de reengenharia dos serviços GEAS utilizando o *framework* Activity-REST e a criação de um novo repositório de serviços chamado GEAS-RAS; por fim, a seção 8.3 descreve em detalhes o processo de reengenharia de um serviço GEAS.

8.1 Visão geral

O repositório *Gene Expression Analysis Services* (GEAS)[1] disponibiliza um conjunto de serviços *Web* para a análise de dados de expressão gênica provenientes de plataformas microarray e RNA-Seq. O repositório GEAS provê, para cada serviço, uma descrição textual das operações providas pelo serviço, bem como uma descrição WSDL detalhando sua interface e bibliotecas clientes que podem ser utilizadas para o acesso programático aos diferentes *endpoints* providos.

Os serviços presentes no repositório GEAS podem ser divididos em três grupos: i) suporte à análise de dados de microarray; ii) suporte à análise de dados de RNA-Seq; e iii) suporte à análise de enriquecimento e visualização de dados de expressão gênica. A Tabela 2 apresenta uma visão geral dos serviços disponibilizados

[1] `https://purl.org/cawal/lssb/geas`.

pelo repositório GEAS, o grupo ao qual esse serviço pertence e uma breve descrição da atividade de análise associada.

Tabela 2 – Serviços de análise do repositório GEAS.

Serviço GEAS	Grupo	Atividade de análise
MicroAffyNorm	I	Normalização de dados de microarray da plataforma Affymetrix
MicroAgilentNorm	I	Normalização de dados de microarray (*one-color*) da plataforma Agilent
MicroGenepixNorm	I	Normalização de dados de microarray da plataforma Genepix
MicroOneDifferentialAnalysis	I	Identificação de genes diferencialmente expressos em dados de microarray *one-color*
MicroTwoDifferentialAnalysis	I	Identificação de genes diferencialmente expressos em dados de microarray *two-color*
MicroHCluster	I	Agrupamento hierárquico de dados de microarray
MicroKCluster	I	Agrupamento de dados de microarray pelo método de *k-means*
MicroHClusterViewer	I	Visualização de dados de microarray hierarquicamente agrupados
RnaSeqDifferentialAnalysis	II	Identificação de genes diferencialmente expressos em dados de RNA-Seq
EnrichmentAnalysis	III	Análise de enriquecimento de dados de expressão gênica
DAVID-REST	III	Análise de enriquecimento de dados de expressão gênica
GeneSetEnrichmentAnalysis	III	Análise de enriquecimento de grupos de genes em dados de expressão gênica
KeggPathwayViewer	III	Visualização de dados de expressão gênica em vias KEGG

Fonte: Autoria própria.

Cada serviço do repositório GEAS adapta uma ou mais ferramentas de análise de linha de comando, a qual provê o suporte necessário à execução da atividade de análise disponibilizada pelo serviço. A Tabela 3 provê uma visão geral das ferramentas adaptadas por cada serviço GEAS. Algumas destas ferramentas foram desenvolvidas pelo nosso grupo de pesquisa (*in house*), enquanto outras ferramentas foram desenvolvidas por terceiros e utilizadas sem quaisquer modificações (*off-the-shelf*). As ferramentas *in house* adaptadas utilizam principalmente de tecnologias Java e pacotes R para análise estatística e de bioinformática.

Tabela 3 – Ferramentas de análise adaptadas pelos serviços do repositório GEAS.

Serviço GEAS	Ferramenta	Tipo
MicroAffyNorm	AffyNorm	*in house*
MicroAgilentNorm	AgilentOneColorNorm	*in house*
MicroGenepixNorm	GenepixNorm	*in house*
MicroOneDifferentialAnalysis	TwoSampleFoldChange TwoSampleTestT	*in house*
MicroTwoDifferentialAnalysis	OneSampleFoldChange OneSampleTestT	*in house*
MicroHCluster	Cluster 3.0	*off-the-shelf*
MicroKCluster	Cluster 3.0	*off-the-shelf*
MicroHClusterViewer	TreeView	*off-the-shelf*
RnaSeqDifferentialAnalysis	DESeqAnalysisR	*in house*
EnrichmentAnalysis	EnrichmentAnalysisR	*in house*
DAVID-REST	DAVIDClient	*off-the-shelf*
GeneSetEnrichmentAnalysis	GeneSetEnrichmentAnalysisR	*in house*
KeggPathwayViewer	KeggPathwayViewerApplication	*in house*

Fonte: Autoria própria.

8.2 Processo de reengenharia do repositório GEAS

A reengenharia dos serviços do repositório GEAS foi realizada segundo as seguintes etapas: i) refatoração de serviços que, segundo o modelo RAS, realizam mais de uma atividade de análise; ii) refatoração das ferramentas de análise *in house* usadas pelos serviços do repositório GEAS; iii) criação de um projeto Activity-REST para cada novo serviço; iv) compilação, teste e implantação dos novos serviços.

8.2.1 Refatoração de serviços que realizam diferentes atividades de análise segundo o modelo RAS

De modo a facilitar a criação e o uso de serviços de análise, o modelo RAS define que uma atividade de análise deve possuir um número fixo e determinado de conjuntos de dados de entrada e saída, bem como de parâmetros de execução. Ademais, o modelo RAS define que um serviço de análise deve prover a execução de uma única atividade de análise. Assim, um mesmo serviço RAS não deve permitir usos nos quais diferentes cardinalidades para algum conjunto de dados ou parâmetro de execução sejam requeridas. Segundo o modelo RAS, essa característica denota

que cada caso de uso representa efetivamente uma atividade de análise distinta e, portanto, cada caso de uso deveria ser separado em um serviço dedicado. Por esta razão, durante essa etapa da reengenharia dos serviços GEAS, inicialmente avaliamos cada serviço de análise do repositório GEAS de modo a identificar as diferentes atividades de análise providas por cada um desses serviços.

Inicialmente, identificamos todos os *endpoints* de cada serviço e mapeamos os conjuntos de dados e os parâmetros de execução associados a esses *endpoints*. Durante esse mapeamento, utilizamos as descrições WSDL dos serviços GEAS para obter informações sobre esses *endpoints* e as mensagens trocadas entre cliente e serviço.

Em seguida, identificamos conjuntos de *endpoints* que são utilizados de maneira alternativa pelo usuário do serviço, i.e., conjuntos de *endpoints* que, se utilizados, excluem a possibilidade de outros *endpoints* do serviço também serem utilizados para a mesma instância da atividade de análise. Essa atividade foi facilitada pela disponibilidade de modelos BPMN [29] que descrevem as interações entre cliente e serviço durante a execução da(s) atividade(s) de análise providas. Por meio desses modelos, identificamos divisões no fluxo de comunicação entre cliente e serviço e mapeamos os diferentes parâmetros de execução e conjuntos de dados submetidos/recebidos em cada um dos caminhos dessa comunicação.

Cada conjunto de *endpoints* mutualmente excludentes foi considerado um caso de uso/atividade de análise diferente e, portanto, conjuntos de dados e parâmetros de execução manipulados por esses *endpoints* foram mapeados para serviços independentes. Conjuntos de dados e parâmetros de execução manipulados por *endpoints* presentes em todos os possíveis fluxos de execução foram, por sua vez, mapeados para todos os casos de uso/atividades de análise definidas, de modo que estes estejam presentes em todos os diferentes serviços derivados após a reengenharia.

A Tabela 4 apresenta o mapeamento entre os serviços GEAS e os serviços GEAS-RAS após a atividade de reengenharia. Nomeamos os serviços GEAS-RAS utilizando o caminho do recurso base do serviço. Dois serviços GEAS, *MicroOneDifferentialAnalysis* e *MicroTwoDifferentialAnalysis*, possibilitavam a execução de mais de uma atividade de análise e foram divididos em serviços mais simples.

8.2.2 Refatoração de ferramentas de análise utilizadas pelo GEAS

As ferramentas desenvolvidas *in house* e adaptadas pelos serviços GEAS originalmente apresentavam algumas características que dificultavam sua maior distribuição, implantação e uso pela comunidade em geral. Por exemplo, as ferramentas desenvolvidas *in house* careciam de uma interface de linha de comando que provesse argumentos descritivos, bem como dependiam da existência de um processo auxilar

Tabela 4 – Mapeamento de serviços GEAS para serviços GEAS-RAS.

Serviço GEAS	Serviço(s) GEAS-RAS
MicroAffyNorm	one-color-affymetrix-microarray-normalization
MicroAgilentNorm	agilent-one-color-microarray-normalization
MicroGenepixNorm	genepix-microarray-normalization
MicroOneDifferentialAnalysis	one-color-microarray-fold-change one-color-microarray-t-test
MicroTwoDifferentialAnalysis	two-color-microarray-fold-change two-color-microarray-t-test
MicroHCluster	hierarquical-clustering-of-microarray-data
MicroKCluster	k-means-clustering-of-microarray-data
MicroHClusterViewer	hierarquical-cluster-viewer
RnaSeqDifferentialAnalysis	deseq2-analysis
EnrichmentAnalysis	enrichment-analysis
DAVID-REST	david-chart-report-by-dataset
GeneSetEnrichmentAnalysis	gene-set-enrichment
KeggPathwayViewer	kegg-pathway-viewer

Fonte: Autoria própria.

atuando como um *daemon* para a sua execução. Embora o *framework* Activity-REST pudesse ser usado para adaptar essas ferramentas sem a necessidade de modificá-las, aproveitamos o período de reengenharia dos serviços para refatorar as ferramentas desenvolvidas *in house* com objetivo de i) reduzir o esforço para a instalação e execução dessas ferramentas em uma dada máquina hospedeira, e ii) melhorar a interface de linha de comando provida por essas ferramentas, facilitando sua utilização por novos usuários.

Durante essa etapa, inicialmente instalamos e utilizamos as ferramentas desenvolvidas *in house* enquanto realizamos uma atualização periódica da máquina hospedeira dos serviços GEAS. No decurso dessa instalação, descrevemos o processo utilizado e anotamos os pontos de dificuldade encontrados, bem como as dificuldades encontradas durante o uso das ferramentas durante o teste realizado posteriormente. Em seguida, analisamos o código-fonte dessas ferramentas de modo a identificar as decisões de implementação que causaram as dificuldades encontradas e compreender como refatorar cada ferramenta de modo a tratar as dificuldades encontradas. Para todas as ferramentas analisadas, percebemos que decisões de implementação semelhantes foram utilizadas e definimos diretrizes para a refatoração dessas ferramentas.

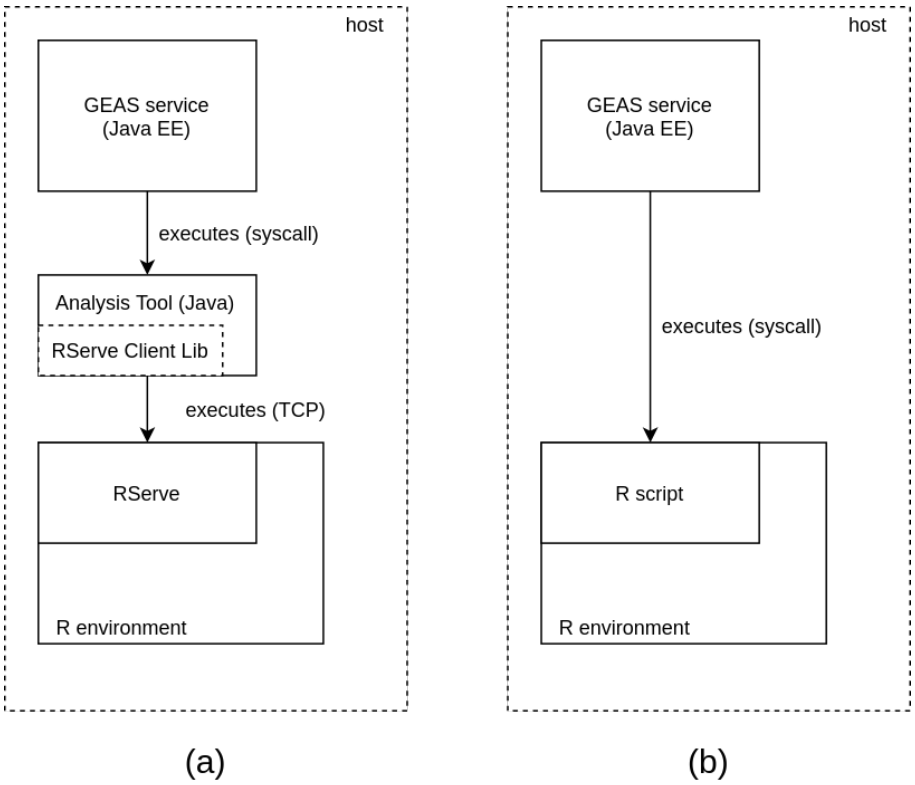
Posteriormente, refatoramos as ferramentas desenvolvidas *in house* segundo estas diretrizes. Por fim, definimos novos meios de implantação para as ferramentas após a refatoração.

As ferramentas atualizadas durante esta etapa definiam o método de execução da atividade de análise por meio da linguagem R, utilizando o ambiente de execução R para o processamento dos conjuntos de dados e parâmetros de execução enviados pelo usuário. Porém, a execução do código R definido para atividade de análise era realizada de maneira indireta: ao invés de invocar diretamente um programa (*script*) R, um serviço GEAS invocava um executável escrito em Java. Esse programa Java recebia do serviço GEAS os parâmetros e os conjuntos de dados de entrada enviados pelo usuário e, por meio de concatenações de *strings* substituindo variáveis, criava um código R para a atividade de análise pretendida. O código R resultante, bem como arquivos dos conjuntos de dados, eram, então, enviados para um processo *daemon*, o qual finalmente executaria a atividade de análise. O processo *daemon* consistia de uma instância do ambiente de execução R sempre ativa, na qual um servidor `RServe` [111] era mantido em execução. `RServe` consiste de um pequeno servidor que espera por conexões TCP, avalia os dados recebidos por meio dessa conexão e executa as declarações presentes nesses dados no ambiente R subjacente, retornando o resultado dessa execução ao processo que a iniciou. A Figura 57a apresenta uma visão geral desses componentes durante a execução de uma atividade de análise.

Após analisar o uso do `RServe` para a execução de atividades de análise, constatamos que este representava uma complexidade desnecessária para os serviços GEAS. O uso desse processo *daemon* também representava um problema à estabilidade desses serviços, uma vez que os serviços GEAS dependiam transitivamente desse *daemon* que, em diferentes ocasiões, tornara-se irresponsivo durante a operação. Assim, decidimos reimplementar as ferramentas diretamente na linguagem R, removendo assim o executável em Java intermediário e a necessidade do *daemon* `RServe`. Dessa maneira, um dado serviço GEAS pode chamar diretamente o *script* R associado, iniciando o ambiente R sob demanda. Tal arquitetura contribuiu para evitar falhas que poderiam ocorrer durante a conexão com o *daemon* `RServe`, bem como facilitou a implantação dessas ferramentas em novos hospedeiros. A Figura 57b apresenta uma visão geral da arquitetura refatorada de uma ferramenta de análise *in house*.

Durante a atualização das ferramentas de análise do GEAS, também adicionamos aos novos *scripts* uma interface de linha de comando mais descritiva utilizando a biblioteca `argparser` [112]. Esta biblioteca provê auxílio à definição da interface de linha de comando de um programa, facilitando a definição, validação e documentação dos argumentos esperados durante a invocação desse programa, bem como o provimento de valores padrões para alguns parâmetros opcionais e de documentação

Figura 57 – Visão geral da execução de uma ferramenta de análise desenvolvida *in house* adaptada pelos serviços GEAS. a) arquitetura original; b) arquitetura refatorada.



Fonte: Autoria própria.

de ajuda ao usuário. Dessa maneira, buscamos auxiliar novos usuário durante o uso das ferramentas *in house* já existentes.

Após a atualização, as ferramentas foram testadas utilizando conjuntos de dados de entrada e parâmetros de execução da atividade de análise. Nessa atividade, comparamos os conjuntos de dados de saída obtidos por meio da sua versão anterior com conjuntos de dados de saída obtidos pela execução da nova versão e constatamos que não houve alteração nos valores presentes nesses conjuntos de dados.

Por fim, empacotamos cada ferramentas *in house* em pacotes DEB, os quais podem ser instalados em sistemas Linux baseados na distribuição Debian. Dessa maneira, buscamos facilitar a distribuição e instalação das ferramentas de análise do repositório GEAS neste sistema operacional. Adicionalmente, também provemos uma imagem Docker[2] na qual as ferramentas subjacentes aos serviços GEAS estão pré-instaladas. Tal facilidade permite que usuários utilizam estas ferramentas por meio de um ambiente isolado e portável, sem a necessidade de instalação direta no sistema operacional local. O código-fonte das ferramentas de análise GEAS atualizadas foi,

[2] `https://hub.docker.com/repository/docker/cawal/geas-scripts`

então, disponibilizado em um repositório Git[3] dedicado.

8.2.3 Desenvolvimento dos serviços GEAS-RAS

Após o mapeamento dos serviços GEAS para serviços GEAS-RAS, realizamos o desenvolvimento de cada um dos novos serviços segundo nossa abordagem orientada a modelos. O desenvolvimento de um serviço GEAS-RAS ocorreu em quatro etapas: i) criação de um projeto Maven para o novo serviço; ii) descrição da atividade de análise por meio da linguagem ActDL; iii) descrição dos aspectos da implantação; e iv) compilação, empacotamento e implantação do novo serviço.

A criação do projeto Activity-REST para cada serviço de análise fez uso do suporte automático para a criação desses projetos provido pelo artefato Maven distribuído em conjunto com o *framework* Activity-REST. Dessa maneira, um projeto com todos os arquivos necessários para um serviço Activity-REST padrão foi obtido semi-automaticamente, necessitando apenas da descrição da atividade de análise a ser executada e dos aspectos da implantação na máquina hospedeira.

Em seguida, uma descrição ActDL foi desenvolvida para cada novo serviço RAS. Durante essa etapa, descrevemos os conjuntos de dados de entrada e saída e os parâmetros de execução da atividade de análise por meio da linguagem ActDL. Também descrevemos a lista de argumentos utilizada para a execução da atividade de análise por meio da ferramenta subjacente. Essa descrição foi realizada utilizando o suporte provido pelo *plugin* para o ambiente de desenvolvimento Eclipse que definimos para a linguagem ActDL.

Após a descrição da atividade de análise a ser executada por cada serviço e da ferramenta que suporta essa execução por meio de uma descrição ActDL, descrevemos os aspectos mais importantes da implantação desse serviço por meio de um modelo SDDL. Nesse sentido, descrevemos nesses modelos o nome do serviço, um número de versão da API provida, bem como dados do contêiner que o receberia e o URL base do serviço.

As descrições ActDL de cada serviço e os modelos de implantação foram, então, incluídos no projeto Maven do serviço. Estes artefatos substituíram os arquivos *templates* criados durante a inicialização do projeto pelo arquétipo Maven.

8.2.4 Compilação, teste e implantação dos serviços GEAS-RAS

Após o desenvolvimento dos modelos ActDL e de implantação, compilamos os novos serviços e os executamos localmente em um computador contendo as ferramentas subjacentes já instaladas. Testamos estes serviços localmente por meio da

[3] `https://purl.org/cawal/lssb/geas-scripts`

implementação de testes automatizados criados utilizando o suporte dos *frameworks* Java JUnit [113] e RestAssured [114]. JUnit consiste de um *framework* para o desenvolvimento de testes automatizados em Java, os quais podem ser automaticamente executados durante a fase de compilação e empacotamento do código. RestAssured consiste em um *framework* para o desenvolvimento de testes para serviços RESTful que provê uma API para a descrição de requisições HTTP, bem como das respostas esperadas, no formato de uma linguagem específica de domínio.

Para cada serviço foi implementado ao menos um teste automatizado compreendendo a execução completa de uma instância de atividade de análise. Finalmente, após o teste dos novos serviços, estes foram implantados como um novo repositório, denominado GEAS-RAS[4]. Os projetos Java criados para a implementação desses serviços, bem como os modelos ActDL definidos nessa implementação, então disponíveis no repositório Git do projeto GEAS-RAS[5].

8.3 Reengenharia do serviço MicroOneDifferentialAnalysis

Esta seção ilustra o processo de reengenharia do serviço GEAS *MicroOneDifferentialAnalysis*. Este serviço foi selecionado para o estudo de caso pois sua reengenharia demandou ações em todas as etapas do processo de reengenharia definidas anteriormente.

8.3.1 Análise estrutural do serviço MicroOneDifferentialAnalysis

O serviço *MicroOneDifferentialAnalysis* provê suporte à atividade de análise *identificação de genes diferencialmente expressos em dados de microarray one-color*. Esta atividade recebe dois conjuntos de dados de entrada, cada qual representando uma condição experimental diferente. Cada conjunto de dados de entrada recebe, para execução, um arquivo tabular contendo uma ou mais amostras de mRNA já normalizadas.

O serviço *MicroOneDifferentialAnalysis* provê dois métodos diferentes para a identificação dos genes diferencialmente expressos entre condições experimentais: por meio de análise de *fold change* ou pelo método de análise *t de Student*. O usuário do serviço faz a escolha do método utilizado quando submete a atividade de análise para execução, por meio do acesso a diferentes *endpoints* disponibilizados pelo serviço.

Diferentes de parâmetros de execução são necessário para a utilização de cada método de identificação de genes diferencialmente expressos. O método de análise *fold change* requer um valor inteiro usado como limiar de corte para a seleção de genes.

[4] `https://purl.org/cawal/lssb/geas-ras`
[5] `https://purl.org/cawal/lssb/geas-ras-git`

Por sua vez, o método de análise *t de Student* requer um valor real representando o nível de significância do teste, um valor inteiro representando o limiar de corte para a seleção de genes e um identificador (*string*) representando o método de correção a ser utilizado.

Como resultado, o serviço produz um conjunto de dados contendo uma lista de genes diferencialmente expressos entre condições experimentais. Assim como os parâmetros de execução, as informações presentes no conjunto de dados produzido é diferente para cada método de análise: enquanto para o método *fold change* apenas uma lista de identificadores de gene é provida, o método *t de Student* também associa a cada gene dessa lista o valor da estatística de teste t e o valor-p encontrado (assim como o valor-p ajustado, caso tenha sido utilizado um método de correção). Adicionalmente, cada um dos métodos de análise é suportado por uma ferramenta de análise diferente.

A Tabela 5 apresenta os *endpoints* providos pelo serviço *MicroOneDifferentialAnalysis* no repositório GEAS. Esta tabela apresenta o URL de cada *endpoint* relativa ao recurso raiz do serviço. Removemos parte do caminho absoluto dos *endpoints* para manter a tabela mais concisa. A tabela também apresenta os métodos HTTP aceitos por esses *endpoints* e uma breve descrição do efeito de cada operação provida. A Figura 58 apresenta uma visão geral da execução de uma instância de atividade de análise no serviço *MicroOneDifferentialAnalysis* por meio de um diagrama BPMN.

O serviço *MicroOneDifferentialAnalysis* foi analisado e constatou-se que este provia dois diferentes métodos de execução para a atividade de análise, cada qual utilizando diferentes parâmetros. A existência de dois diferentes métodos de execução é explicitada também pelo uso de diferentes ferramentas subjacentes de acordo com o método estatístico utilizado. Nesse sentido, encontramos duas atividades de análise diferentes providas pelo serviço *MicroOneDifferentialAnalysis*: i) identificação de genes diferencialmente expressos em dados de microarray *one-color* por meio de *fold change*, e; ii) identificação de genes diferencialmente expressos em dados de microarray *one-color* por meio de teste *t de Student*.

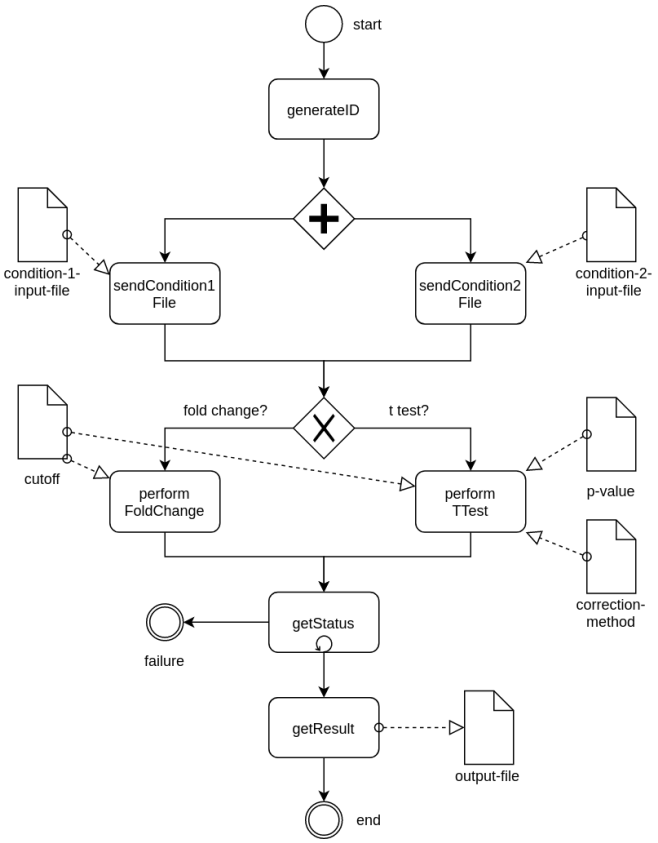
A Tabela 6 apresenta uma visão geral dos conjuntos de dados e parâmetros de execução de cada atividade de análise do serviço *MicroOneDifferentialAnalysis*. Embora os conjuntos de dados de entrada de ambas atividades sejam semelhantes, diferentes parâmetros de execução são necessários para cada atividade de análise. Adicionalmente, os tipos de dados representados nos conjuntos de dados de saída produzidos por cada atividade de análise diferem ligeiramente. Por esta razão, este serviço foi dividido em diferentes serviços RAS.

Tabela 5 – *Endpoints* do serviço GEAS *MicroOneDifferentialAnalysis*.

Endpoint	Métodos HTTP aceitos	Descrição
`/generateID`	GET	Cria uma instância da atividade de análise e retorna um identificador dessa instância no formato textual (*plain text*). Este identificador é utilizado nos demais *endpoints* do serviço (representado por `{instance-id}` no URL desses *endpoints*).
`/getCorrectionMethods`	GET	Retorna uma lista de identificadores de métodos de correção para testes múltiplos em formato XML. (*Endpoint informacional*)
`/sendCondition1File/ {instance-id}/ {file-name}`	POST	Submete um arquivo de dados de réplicas de microarray *one-color* normalizada para o conjunto de dados de entrada da condição 1 (referência). O arquivo é enviado no corpo da requisição HTTP.
`/sendCondition2File/ {instance-id}/ {file-name}`	POST	Submete um arquivo de dados de réplicas de microarray *one-color* normalizada para o conjunto de dados de entrada da condição 2 (alvo). O arquivo é enviado no corpo da requisição HTTP.
`/performFoldChange/ {instance-id}/ {cutoff}`	POST	Inicia a execução da atividade de análise utilizando o método *fold change* para a identificação de genes diferencialmente expressos. O valor de `{cutoff}` deve ser um número interior que será utilizado como limiar de corte para a seleção de genes diferencialmente expressos.
`/performTTest/ {instance-id}/ {p-value}/ {cutoff}/ {correction-method}`	POST	Inicia a execução da atividade de análise utilizando o método *teste T* para a identificação de genes diferencialmente expressos. O valor de `{p-value}` deve ser um número real que será utilizado para o nível de significância esperada do teste. O valor de `{cutoff}` deve ser um número interior que será utilizado como limiar de corte para a seleção de genes diferencialmente expressos. O valor de `{correction-method}` deve ser uma *string* que será utilizada para a seleção do método de correção de testes múltiplos (informada entre as opções providas pelo *endpoint* `/getCorrectionMethods`.
`/getStatus/ {instance-id}`	GET	Retorna o estado de execução da atividade de análise em texto no corpo da resposta HTTP.
`/getResult/ {instance-id}`	GET	Retorna o conjunto de dados de saída produzido pela execução com sucesso da instância de atividade de análise.

Fonte: Autoria própria.

Figura 58 – Execução de uma instância de atividade de análise no serviço *MicroOneDifferentialAnalysis*.



Fonte: Autoria própria.

8.3.2 Refatoração das ferramentas de análise utilizadas pelo serviço MicroOneDifferentialAnalysis

Após a divisão do serviço *MicroOneDifferentialAnalysis* em serviços específicos para cada atividade de análise executada, avaliamos e refatoramos as ferramentas de análise utilizadas pelo serviço original. Durante esta etapa, retiramos o uso do serviço *daemon* `RServe` e reescrevemos as ferramentas, originalmente desenvolvidas em Java e R, utilizando apenas a linguagem R.

A Listagem 22 apresenta alguns detalhes do código da ferramenta *TwoSampleFoldChange* antes da refatoração. Dois fragmentos são apresentados nessa listagem, ambos contendo código escrito utilizando a linguagem Java. O primeiro fragmento apresenta a extração dos argumentos de linha de comando passados para o executável Java (linhas 9 a 13) e o estabelecimento de conexão com o serviço `RServe` (linha 16). A ferramenta, antes da refatoração, recebia todos os argumentos necessários para sua execução em uma ordem fixa e não apresentava documentação destes ao usuário. Dessa maneira, o usuário da ferramenta precisava verificar a ordem destes argumentos no código-fonte caso houvesse dúvidas sobre o uso da ferramenta.

Tabela 6 – Resultado da análise estrutural do serviço *MicroOneDifferentialAnalysis*.

	A Identificação de genes diferencialmente expressos em dados de microarray one-color por meio de *fold change*	B Identificação de genes diferencialmente expressos em dados de microarray one-color por meio de teste *t de Student*
Conjuntos de dados de entrada	**condition-1-input-file:** Valores de expressão de mRNA para amostras na condição de referência **condition-2-input-file:** Valores de expressão de mRNA para amostras na condição alvo	**condition-1-input-file:** Valores de expressão de mRNA para amostras na condição de referência **condition-2-input-file:** Valores de expressão de mRNA para amostras na condição alvo
Parâmetros de execução	**cutoff:** Valor de corte para seleção de genes	**cutoff:** Valor de corte para seleção de genes **p-value:** Nível de significância do teste (valor-p) **correction-method:** Método de correção a ser utilizado
Conjuntos de dados de saída	**output-file:** Lista de genes diferencialmente expressos entre as amostras em cada condição	**output-file:** Lista de genes diferencialmente expressos entre as amostras em cada condição associados ao valor da estatística de teste t, valor-p encontrado e valor-p ajustado para testes múltiplos

Fonte: Autoria própria.

O segundo fragmento apresentado na Listagem 22 ilustra como o código R era executado por meio do serviço `RServe`. Duas principais operações eram utilizadas: atribuição (`RConnection::assign()` linhas 21 e 25) e execução (`RConnection::eval()`, linhas 22 e 26). A operação de atribuição recebe uma *string* representado o nome de uma variável (no ambiente R) e um objeto a ser atribuído a essa variável pelo ambiente `RServe`. Por sua vez, a operação de execução recebe uma *string* contendo código R a ser executado diretamente pelo serviço `RServe`. A *string* enviada para a avaliação era, em muitos pontos, gerada por meio de concatenação de *strings* no processo Java original.

A Listagem 23 apresenta alguns detalhes do código da ferramenta após a refatoração. Dois fragmentos também são apresentados nessa listagem, ambos contendo código escrito utilizando a linguagem R. O primeiro fragmento apresenta a utilização da biblioteca `argparser` para a análise dos argumentos de linha de comando, bem como para o provimento de textos de ajuda para o usuário da

ferramenta. Neste sentido, a biblioteca `argparser` é incluída (linha 15) e um objeto próprio para a análise automática dos argumentos recebidos pela ferramenta é criado (linha 18). Em seguida, os argumentos aceitos pela ferramenta são definidos utilizando esse objeto. A listagem apresenta a definição de um desses argumentos (linha 21), nomeado `condition-1-input-file`, sua descrição (linha 22), a cardinalidade aceita para este argumento (linha 23, apenas um único valor) e seu tipo (linha 24, uma *string* de caracteres que representam o caminho do arquivo).

O segundo fragmento apresenta o código equivalente ao apresentado na Listagem 22. Neste fragmento, as declarações R são utilizadas diretamente, sem a necessidade de serem criadas como *strings* pelo código Java. Assim, as variáveis `cond1` e `cond2` são inicializadas diretamente com os dados da contidos nos arquivos `condition1InputFile` e `condition2InputFile` (linhas 53 e 54).

8.3.3 Desenvolvimento dos serviços RAS

Após a refatoração das ferramentas de análise utilizadas pelo serviço original, criamos projetos Maven para cada um dos dois novos serviços RAS a serem desenvolvidos utilizando o arquétipo Maven provido pelo *framework* Activity-REST. Em seguida, desenvolvemos as descrições ActDL das atividades de análise a serem executadas por cada um desses serviços.

A Listagem 24 apresenta a descrição ActDL da atividade de análise *identificação de genes diferencialmente expressos em dados de microarray one-color por meio de fold-change*. Dois conjuntos de dados de entrada (`condition-1-input-file` e `condition-2-input-file`) estão definidos nessa descrição, representando respectivamente os valores de expressão de mRNA para amostras nas condições de referência e alvo (linhas 2 e 3). Estas amostras são submetidas como um único arquivo no formato de texto tabular (tipo MIME `text/tsv`). Em seguida, o valor de corte para a seleção de genes é definido como um parâmetro de execução (`cutoff`), representado como um número real de cardinalidade unitária (linha 7). Como resultado da execução da atividade de análise, um único conjunto de dados de saída (`output`) é produzido (linha 12). Este conjunto de dados consiste de único arquivo tabular (tipo MIME `text/tsv`).

Após a descrição dos conjuntos de dados de entrada e saída, bem como do parâmetro de execução, a ferramenta de análise subjacente (`geas-one-color-microarray-fold-change.R`) foi, então, descrita. Primeiramente, a ferramenta recebe como argumento o valor de corte informado pelo usuário precedido da *string* `--cutoff` (linhas 16 e 17) Nesta linhas é utilizado o manipulador `PrependListWith` para adicionar a *string* `--cutoff` como argumento antes do valor de corte informado pelo usuário. Em seguida, são passados à ferramenta os nomes dos arquivos que

Listagem 22 – Detalhes do código da ferramenta *TwoSampleFoldChange* antes da refatoração.

```
7     public static void main(String[] args) throws RserveException {
8
9         //Get parameters
10        String condition1InputFile = args[0];
11        String condition2InputFile = args[1];
12        String outputFileName = args[2];
13        String cutoff = args[3];
14
15        //Make a new local connection on default port (6311)
16        RConnection c = new RConnection();
```

⋮

```
20        //Read condition 1 file
21        c.assign("inputFile", condition1InputFile);
22        c.eval("cond1 = read.table(file=inputFile, header = TRUE, sep =
      \"\t\", na.strings = \"NA\", comment.char = \"&\")");
23
24        //Read condition 2 file
25        c.assign("inputFile", condition2InputFile);
26        c.eval("cond2 = read.table(file=inputFile, header = TRUE, sep =
      \"\t\", na.strings = \"NA\", comment.char = \"&\")");
```

Fonte: Autoria própria.

Listagem 23 – Detalhes do código da ferramenta *TwoSampleFoldChange* após da refatoração.

```
15 library("argparser");
16
17 # Create a command line parser
18 p <- arg_parser(PURPOSE);
19
20
21 p <- add_argument(p, "condition1-input-file",
22   help="Input file for the first comparing condition",
23   nargs=1,
24   type="character");
```

⋮

```
53 cond1 <- read.table(file=condition1InputFile, header = TRUE, sep = "\t",
       na.strings = "NA", comment.char = "&"));
54 cond2 <- read.table(file=condition2InputFile, header = TRUE, sep = "\t",
       na.strings = "NA", comment.char = "&"));
```

Fonte: Autoria própria.

representam os conjuntos de dados `condition-1-input-file` (linha 18), `condition-2-input-file` (linha 19) e `output-file` (linha 20). O nome desses arquivos em tempo de execução é desconhecido para o criador da descrição ActDL, sendo transparentemente substituído pelo *framework* Activity-REST.

Após a descrição da chamada utilizada para executar a ferramenta, são descritos os códigos de saída retornados após a execução da ferramenta. Dois códigos de saída são descritos: o código de saída `0` representa que essa execução foi bem sucedida (linha 23), enquanto o código de saída `2` representa que a execução terminou com falha (linha 24). Para facilitar a depuração da falha, adicionamos uma pequena descrição da falha associada a esse código (`Bad invocation parameters`). Essa descrição é retornada junto ao relatório de erros produzido para a instância de atividade de análise caso a falha venha a ocorrer.

Listagem 24 – Descrição ActDL da atividade de análise *identificação de genes diferencialmente expressos em dados de microarray one-color por meio de fold-change*

```
1  activity one-color-microarray-fold-change {
2    on {
3      dataset condition1-input-file : 'text/tsv' [1,1];
4      dataset condition2-input-file : 'text/tsv' [1,1];
5    }
6    with {
7      parameter cutoff : REAL [1,1] {
8        remark 'Cutoff (log2-based) for filtering the fold change.';
9      };
10   }
11   produces {
12     dataset output-file : 'text/tsv' [1,1];
13   }
14   using executable 'geas-one-color-microarray-fold-change.R' {
15     commandLineTemplate [
16       parameter cutoff
17         | PrependListWith '--cutoff',
18       dataset condition1-input-file,
19       dataset condition2-input-file,
20       dataset output-file
21     ]
22     returns {
23       0 if SUCCEEDED;
24       2 if FAILED 'Bad invocation parameters.';
25     }
26   }
27 }
```

Fonte: Autoria própria.

A Listagem 25 apresenta a descrição ActDL da atividade de análise *identificação de genes diferencialmente expressos em dados de microarray one-color por meio do teste* t de Student. Esta descrição é semelhante à descrição da Listagem 24, diferindo apenas nos parâmetros de execução (linas 6 a 12) e a ferramenta de análise descritas (linhas 16 a 31). Nesse sentido, dois outros parâmetros de execução são descritos: o método de correção de testes múltiplos a ser utilizado (`correction-method`, linha 10), que é representado por uma *string*, e o valor-p esperado para o teste

(`p-value`, linha 11), representado como um número real cujo valor padrão é 0.05. A ferramenta de análise utilizada, `geas-one-color-microarray-t-test.R`, recebe como argumento o valor de corte precedido da *string* `--cutoff`, o nome do método de correção precedido da *string* `--correction-method`, o valor-p esperado precedido da *string* `--p-value`, e, por fim, os nomes dos arquivos para os três conjuntos de dados (na mesma ordem da ferramenta anterior).

Listagem 25 – Descrição ActDL da atividade de análise *identificação de genes diferencialmente expressos em dados de microarray one-color por meio do teste* t de Student

```
1  activity one-color-microarray-t-test {
2    on {
3      dataset condition1-input-file : 'text/tsv' [1,1];
4      dataset condition2-input-file : 'text/tsv' [1,1];
5    }
6    with {
7      parameter cutoff : REAL [1,1] {
8        remark 'Cutoff (log2-based) for filtering the fold change.';
9      };
10     parameter correction-method : STRING [1,1];
11     parameter p-value : REAL [1,1] = ['0.05'];
12   }
13   produces {
14     dataset output-file : 'text/tsv' [1,1];
15   }
16   using executable 'geas-one-color-microarray-t-test.R' {
17     commandLineTemplate [
18       parameter cutoff
19         | PrependListWith '--cutoff',
20       parameter correction-method
21         | PrependListWith '--correction-method',
22       parameter p-value
23         | PrependListWith '--p-value',
24       dataset condition1-input-file,
25       dataset condition2-input-file,
26       dataset output-file
27     ]
28     returns {
29       0 if SUCCEEDED;
30       2 if FAILED 'Bad invocation parameters.';
31     }
32   }
33 }
```

Fonte: Autoria própria.

A Listagem 26 apresenta o modelo SDDL para o serviço `one-color-microarray-fold-change`. O modelo SDDL para o serviço `one-color-microarray-t-test` é semelhante e, portanto, não é apresentado aqui. Para ambos os serviços, foi associado o número de versão como 1.1 (linha 2). Também ambos os serviços foram implantados em um contêiner hospedeiro nomeado como `kode` (linha 3), seus URLs base foram definidos (linha 4) e um comentário sobre o contêiner foi provido (linha 5).

Listagem 26 – Modelo SDDL para o serviço de execução da atividade de análise *identificação de genes diferencialmente expressos em dados de microarray one-color por meio de fold-change*

```
1 deployment {
2         of service "one-color-microarray-fold-change" { api-version "1.1" }
3         into container 'kode' {
4                 base-url
    'http://kode.ffcrp.usp.br:8081/one-color-microarray-fold-change'
5                 description 'IT department main host'
6             }
7 }
```

Fonte: Autoria própria.

8.3.4 Compilação, teste e implantação dos serviços RAS

Após o desenvolvimento dos modelos de implantação, compilamos e testamos os novos serviços. Para cada serviço, definimos um conjunto de testes automatizados executados com o auxílio do *framework* de testes JUnit.

A Listagem 27 apresenta um fragmento do código desenvolvido para o teste do serviço `one-color-microarray-fold-change`. Inicialmente é definida a classe `ExecuteAnalysisTests` (linha 36), a qual terá seus métodos executados automaticamente pelo *framework* JUnit. Esta classe estende a classe `TestBase`, de modo a reutilizar um conjunto de métodos auxiliares definidos para a execução de testes de serviços RAS.

O método `setUpBeforeClass()` (linhas 35 a 40) define a inicialização da biblioteca RestAssured. Este método é marcado com a anotação `@BeforeClass` de modo a ser executado antes dos demais métodos de teste.

Cada método de teste é marcado com a anotação `@Test`. A Listagem 27 apresenta um fragmento do teste `executeAnalysis()` (linhas 42 até o final da figura). O método `executeAnalysis()` foi definido para executar completamente uma instância de atividade de análise no serviço `one-color-microarray-fold-change` utilizando os controles de hipermídia providos pelo serviço. Dessa maneira, este método testa se o princípio HATEOAS do REST foi corretamente implementado, ou seja, se o serviço pode ser totalmente utilizado por um clientes que possua apenas conhecimento do endereço do recurso raiz e das relações utilizadas nos controles de hipermídia providos.

O teste utiliza a biblioteca RestAssured para executar as requisições HTTP utilizadas para comunicar-se com o serviço, pois essa biblioteca provê uma interface fluente para a definição de requisições HTTP, bem como de teste para as respostas HTTP recebidas. O teste inicia-se com a requisição ao recurso raiz do serviço (linhas 45 a 49) e a verificação dos controles de hipermídia retornados pela resposta a essa requisição (linhas 51 a 58). Então, o teste cria uma nova instância de atividade

de análise no serviço por meio de uma requisição `POST` a um dos URLs recebidos anteriormente (linhas 61 a 65) e verifica se essa requisição retornou o código de *status* HTTP correto (linha 67).

Em seguida, o teste armazena a localização da instância de atividade de análise criada (linha 70) e requisita uma representação dessa instância (linhas 74 a 77). O teste explora, então, os controles de hipermídia retornados na resposta dessa requisição em busca do endereço para a submissão dos parâmetros de execução da instância da atividade de análise (linhas 80 a 88) e verifica a existência desse endereço (linha 90). A partir de então, o teste utiliza esse endereço para o envio dos parâmetros de execução e continua a execução da atividade de análise utilizando apenas os controles de HATEOAS retornados nas respostas HTTP do serviço (não apresentado na listagem).

Por fim, após os testes dos novos serviços, estes foram empacotados como arquivos WAR e implantados no repositório GEAS-RAS.

8.4 Considerações finais

Durante essa etapa de nosso trabalho, utilizamos o processo de desenvolvimento orientado a modelos proposto por este trabalho na reengenharia do repositório GEAS, um repositório de serviços de análise adaptadores para a execução de atividades de análise no domínio da genômica funcional. Para essa reengenharia, definimos um processo em quatro etapas que nos permitiu obter um novo repositório de serviços adaptadores que seguem o modelo de referência RAS, chamado GEAS-RAS.

O processo de reengenharia definido utiliza o modelo conceitual de um serviço adaptador, proposto em nosso trabalho, para identificar serviços de análise que proveem suporte a mais de uma atividade de análise. As responsabilidades desses serviços devem ser divididas de modo a obter um conjunto de serviços finais que executem, cada um, apenas uma atividade de análise. Dessa maneira, podemos criar modelos ActDL para descrever cada uma dessas atividades, utilizando, então, o *framework* Activity-REST para interpretar esses modelos e prover o serviço adaptador.

Durante o processo de reengenharia dos serviços GEAS, refatoramos também as ferramentas de análise adaptadas pelos serviços iniciais. Neste sentido, provemos uma interface de linha de comando capaz de prover informações sobre os argumentos que a ferramenta deve receber para sua execução, bem como removemos a dependência da existência de um serviço local para executar o código R presente em algumas dessas ferramentas. Essa etapa de refatoração das ferramentas de análise não é necessária para o uso de modelos ActDL e do *framework* Activity-REST para

Listagem 27 – Fragmento do código desenvolvido para teste do serviço `one-color=microarray-fold-change`

```
32 public class ExecuteAnalysisTests extends TestBase {
33
34
35   @BeforeClass
36   public static void setUpBeforeClass() throws Exception {
37
38     TestBase.setUpRestAssured(ExecuteAnalysisTests.class
39         .getResourceAsStream("local_tests_config.properties"));
40   }
41
42   @Test
43   public void executeAnalysis() throws IOException, URISyntaxException {
44
45     // explore the root resource
46     Response r = given()
47       .accept("application/json")
48       .get()
49       .andReturn();
50
51     List<Link> headerList = processHeadersForLinks(r.getHeaders());
52
53     Link newAnalysisLink = headerList.stream()
54       .filter(h ->
55           h.getRel().equalsIgnoreCase(
56             ResourceRelations.ROOT_2_NEW_ANALYSES_COLLECTION
57             )
58           ).findFirst().get();
59
60     // create new instance
61     Response createResponse =
62       given()
63       .accept("application/json")
64       .post(newAnalysisLink.getUri())
65       .andReturn();
66
67     assertEquals(HttpStatus.SC_CREATED, createResponse.getStatusCode());
68
69     // get instance location URL
70     this.locationUri = URI.create(createResponse.header("Location"));
71
72
73     // explore analysis activity instance resource
74     Response getRequest = given()
75       .accept(MediaType.APPLICATION_JSON)
76       .get(locationUri)
77       .andReturn();
78
79     // retrieve parameters link
80     List<Link> headerList =
       processHeadersForLinks(getRequest.getHeaders());
81
82     Link parametersLink = headerList.stream()
83       .filter(h ->
84           h.getRel()
85           .equalsIgnoreCase(
86             ResourceRelations.ANALYSYS_PARAMETERS_COLLECTION)
87           )
88       .findFirst().get();
89
90     assertNotNull(parametersLink);
91
92     ...
```

Fonte: Autoria própria.

obter os novos serviços de análise. Porém, realizamos esse esforço adicional para, principalmente, facilitar a distribuição e o uso dessas ferramentas.

9 Avaliação do modelo RAS e do *framework* Activity-REST

Este capítulo avalia o modelo RAS usado como base para o desenvolvimento do *framework* Activity-REST, bem como o esforço de desenvolvimento de um serviço RAS. Este capítulo também avalia o desempenho de serviços e clientes RAS. As seguintes questões foram investigadas:

QA1: Qual o nível de aderência das diferentes soluções existentes para a criação de serviços (RESTful) frente ao Modelo de Maturidade RESTful de Richardson?

QA2: Quanto esforço de desenvolvimento é economizado por usar o *framework* Activity-REST, se comparado ao desenvolvimento *ad hoc* do mesmo serviço?

QA3: Qual a perda de desempenho imposta pelo uso do *framework* Activity-REST na execução de uma instância de atividade de análise, se comparado à execução de uma instância de atividade de análise usando um serviço desenvolvido de forma *ad hoc*?

O restante do capítulo está estruturado como segue: a seção 9.1 compara diferentes modelos de interação usados (implicitamente) como base para o desenvolvimento de diferentes conjuntos de serviços de análise no domínio biomédico ao Modelo de Maturidade de Richardson, bem como ao próprio modelo de interação RESTful proposto nesse trabalho; a seção 9.2 compara o esforço para a implementação *ad hoc* de um serviço de análise em relação ao esforço de implementação desse mesmo serviço utilizando o *framework* Activity-REST; a seção 9.3 avalia o desempenho dos serviços gerados usando o *framework* Activity-REST e dos clientes RAS gerados para esses serviços durante a execução de uma atividade de análise; a seção 9.4 apresenta um conjunto de críticas à validade de nossas avaliações; por fim, a seção 9.5 apresenta algumas conclusões obtidas durante as atividades de avaliação.

9.1 Avaliação da maturidade dos serviços de análise

9.1.1 Repositórios de serviços em bioinformática

Diversos serviços *web* foram desenvolvidos para facilitar a execução de análises em bioinformática [115–124]. Historicamente, a maior parte dos serviços desenvolvidos na área foram desenvolvidos de modo a prover apenas uma interface de acesso a

bancos de dados biológicos, não havendo suporte à execução remota de atividades de análise [5]. Estes serviços de análise são costumeiramente providos pelos mesmos grupos de pesquisa que criaram a ferramenta de análise subjacente ao serviço, de maneira a promover o uso dessa ferramenta por outros pesquisadores.

Serviços de análise similares ou relacionados costumam ser agrupados em repositórios de serviços de análise. Porém, apenas poucos repositórios permitem a execução de atividades de análise. Entre estes repositórios estão o repositório de serviços mantido pelo *European Bioinformatics Institute* (EBI) [9], o repositório mantido pelo *National Center for Biotechnology Information* (NCBI) [10] e o repositório *Gene Expression Analysis Services* (GEAS) [7]. EBI é um dos principais provedores de serviços de análise [90,125,126], tendo coordenado o projeto EMBRACE [127,128]. O projeto EMBRACE visava formar uma rede de laboratórios europeus para o provimento de serviços *web* para o acesso a bancos de dados biológicos e à execução de ferramentas de análise. Inicialmente os serviços de análise do EBI foram implementados utilizando SOAP [128]. Porém, posteriormente serviços RESTful também foram providos [90].

NCBI é outro repositório de serviços de análise largamente conhecido entre bioinformatas [10, 89]. O repositório de serviços do NCBI provê serviços *web* para a execução de análises de alinhamento de sequências utilizando BLAST [129]. Adicionalmente, NCBI também provê serviços para acesso a um conjunto de bandos de dados para o domínio biomédico, incluindo serviços para acesso à literatura publicada, nem como a dados de genes, proteínas e compostos químicos [89, 130].

Por fim, o repositório GEAS provê um conjunto de serviços para a realização de atividades de análise de expressão gênica. GEAS também provê um conjunto de serviços de adaptação, os quais são utilizados para manipular os dados compartilhados entre serviços de análise. Por fim, os serviços presentes no repositório GEAS são descritos pode meio de descrições WSDL anotadas com termos de ontologias do domínio. A anotação semântica da descrição do serviço com termos de uma ontologia possibilita o uso de técnicas para a exploração e integração baseadas nos termos desse vocabulário estruturado [8].

9.1.2 Modelos de interação utilizados nos diferentes repositórios

Diferentes modelos de interação para a execução de serviços e atividades de análise são utilizados em diferentes repositórios de serviços de análise em bioinformática. Neste sentido, esta atividade de avaliação posicionou o modelo RAS frente aos modelos de interação (implicitamente) usados em outros repositórios de serviços de análise, bom como às próprias restrições definidas pelo Modelo de Maturidade RESTful de Richardson.

De modo a realizar esta avaliação, inicialmente estudamos a estrutura e o modelo de interação de serviços de análise presentes em diferentes repositórios de serviços em bioinformática. Neste sentido, selecionamos um serviço representativo para cada um dos diferentes repositórios e listamos os principais *endpoints* de cada serviço selecionado. Para cada *endpoint*, também listamos os métodos HTTP utilizados sobre esses *endpoints*, os parâmetros recebidos (via URL ou o corpo da requisição HTTP) e as respostas providas por esse *endpoint*. Por fim, posicionamos cada serviço em relação ao Modelo de Maturidade de Richardson.

Os repositórios de serviços de bioinformática *National Center for Biotechnology Information* (NCBI) [89], *European Bioinformatics Institute* (EBI) [9] e *Gene Expression Analysis Services* (GEAS) [7] foram considerados para análise. Na sequência, selecionamos os serviços BLAST como representantes dos repositórios NCBI e EBI. Ambos os serviços, utilizados para a atividade de análise *Alinhamento de sequências biológicas*, requerem um conjunto de dados de entrada composto de um único arquivo e produzem um conjunto de dados de saída de um único arquivo. Porém, uma vez que o repositório GEAS não provê um serviço de alinhamento de sequências biológicas, escolhemos como representante desse repositório um serviço com características similares. Dessa maneira, selecionamos o serviço *Enrichment Analysis* como representante do repositório GEAS. Este serviço, utilizado para a análise *Enriquecimento funcional de informação de expressão gênica*, também requer um conjunto de dados de entrada composto de um único arquivo e produz um conjunto de dados de saída de um único arquivo.

9.1.3 Avaliação do modelo de interação do serviço BLAST do NCBI

A Tabela 7 apresenta o único *endpoint* provido pelo serviço BLAST do repositório do NCBI. Omitimos o URL base dos *endpoints* dos serviços para tornar a sua apresentação mais sucinta.

Tabela 7 – *Endpoint* do serviço BLAST do repositório do NCBI

Endpoint	Métodos HTTP aceitos	Descrição
`/Blast.cgi`	`POST`, `GET`	Único *endpoint* do serviço BLAST do repositório do NCBI. Diferentes operações são realizadas por meio de modificadores no URL e/ou conteúdo do corpo da requisição HTTP.

Fonte: Autoria própria.

Quatro principais operações são disponibilizadas por este único *endpoint*. Cada operação é acessada por meio de uma combinação do método HTTP usado na requisição (`POST` ou `GET`) e um conjunto de parâmetros submetidos no corpo ou nos

modificadores (*query string*) da requisição. Uma vez que o serviço BLAST do NCBI provê apenas um *endpoint* para o acesso e a manipulação de todos os seus recursos, este serviço cumpre apenas os requisitos para o nível 0 do Modelo de Maturidade de Richardson.

9.1.4 Avaliação do modelo de interação do serviço BLAST do EBI

A Tabela 8 apresenta os principais *endpoints* do serviço BLAST do repositório do EBI. Uma *string* delimitada por chaves representa uma variável usada para parametrização do URL. Cada *endpoint* do serviço manipula apenas um método HTTP.

Tabela 8 – *Endpoints* do serviço BLAST do repositório do EBI

Endpoint	Métodos HTTP aceitos	Descrição
`/parameters`	`GET`	Lista parâmetros disponíveis. Retorna uma representação em XML contendo o nome dos parâmetros.
`/parameterdetails/{parameterName}`	`GET`	Lista detalhes dos parâmetros. Retorna uma representação e XML descrevendo o parâmetro.
`/run`	`POST`	Submete uma instância da atividade de análise para processamento.
`/status/{id}`	`GET`	Retorna uma representação em texto plano do estado da execução.
`/resulttypes/{id}`	`GET`	Lista os resultados (i.e., arquivos dos conjuntos de dados de saída) da atividade de análise. Retorna uma representação em XML contendo os identificadores desses resultados.
`/result/{id}/{type}`	`GET`	Recupera um arquivo produzido durante a execução da instância da atividade de análise.

Fonte: Autoria própria.

Os serviços do repositório do EBI proveem um único URL para cada recurso subjacente. Adicionalmente, esses serviços utilizam-se da semântica intrínseca dos métodos HTTP para executar operações nesses recursos. Porém, esses serviços não proveem controles de *hypermedia* que guiem a navegação do usuário pelos diferentes recursos acessíveis. Portanto, o serviço BLAST do repositório do EBI cumpre apenas os requisitos do nível 2 do Modelo de Maturidade de Richardson.

9.1.5 Avaliação do modelo de interação do serviço Enrichment Analysis do GEAS

A Tabela 9 apresenta os principais *endpoints* do serviço *Enrichment Analysis* do repositório GEAS. De maneira similar à descrição do serviço BLAST do repositório

do EBI, uma *string* delimitada por chaves representa uma variável utilizada para parametrização do URL.

Tabela 9 – *Endpoints* do serviço *Enrichment Analysis* do repositório GEAS

Endpoint	Métodos HTTP aceitos	Descrição
`/generateId`	`GET`	Cria uma instância de atividade de análise e retorna o identificador dessa instância em uma representação `text/plain` no corpo da resposta HTTP.
`/sendFile/{id}/{filename}`	`POST`	Submete um arquivo chamado `{filename}` para o único conjunto de dados de entrada da instância da atividade de análise identificada por `{id}`.
`/getCorrectionMethods`	`GET`	Retorna uma lista de identificadores de métodos de correção estatística, bem como uma descrição textual desses métodos, em uma representação XML.
`/getSpecies`	`GET`	Retorna uma lista de identificadores e nomes de espécies.
`/performEnrichmentAnalysis/{id}/{speciesId}/{correctionMethod}`	`POST`	Submete os parâmetros de execução `{speciesId}` e `{correctionMethod}` para a instância de atividade de análise, bem como inicia o processamento dessa instância.
`/getStatus/{id}`	`GET`	Recupera o estado da execução em texto plano.
`/getResult/{id}`	`GET`	Recupera o arquivo produzido para o conjunto de dados de saída da instância de atividade de análise identificada. A atividade de análise executada por este serviço produz apenas um arquivo para este conjunto de dados.

Fonte: Autoria própria.

O serviço *Enrichment Analysis* do repositório GEAS provê um único URL para cada recurso subjacente. Adicionalmente, o serviço *Enrichment Analysis* utiliza-se da semântica intrínseca aos métodos HTTP para a execução de operações sobre esses recursos. Porém, este serviço também não provê controles de *hypermedia* que guiem a navegação dos usuários pelos diferentes recursos acessíveis. Portanto, o serviço *Enrichment Analysis* cumpre apenas os requisitos do nível 2 do Modelo de Maturidade de Richardson.

9.1.6 Análise comparativa com os serviços RAS

Ao compararmos os serviços disponibilizados nos repositórios NCBI, EBI e GEAS, identificamos um conjunto de similaridades na estrutura e no modelo de interação desses serviços. Primeiro, nenhum dos serviços provê suporte completo

ao modelo de URL hierarquicamente definidos para recursos *web*. Ainda assim, os repositórios EBI e GEAS utilizam melhor os URLs como identificadores de recursos. Segundo, esses serviços usam diferentes formatos de representação para a serialização das respostas enviadas aos seus usuários. Neste sentido, o estado da execução de uma instância de atividade de análise é frequentemente retornado em uma representação em texto plano, enquanto informações relacionadas aos parâmetros e seus valores são frequentemente providas em uma representação em XML. Dessa maneira, o usuário de um serviço precisa estar ciente das representações esperadas para cada recurso subjacente e tratar cada representação corretamente, o que dificulta a utilização destes serviços. Finalmente, os serviços não permitem a listagem de seus recursos ou a recuperação dos conjuntos de dados submetidos por um usuário. Assim, o usuário fica obrigado a manter informações locais sobre os conjuntos de dados e parâmetros de execução submetidos a cada instância de atividade de análise até recuperar os resultados do processamento.

O modelo RAS, implementado pelo repositório GEAS-RAS com base no *framework* Activity-REST, foi definido de modo a prover suporte ao desenvolvimento de serviços de análise alinhados ao nível 3 do Modelo de Maturidade de Richardson. Diferentemente das soluções consideradas nesta avaliação, serviços RAS proveem suporte ao modelo hierárquico de identificação para todos os recursos de uma instância de atividade de análise, endereçando cada sub-recurso por meio de caminhos derivados do caminho do recurso que conceitualmente o engloba. Adicionalmente, um serviço RAS provê a capacidade de negociação da representação dos recursos da instância da atividade de análise, permitindo que os usuários do serviço escolham o formato de representação para o corpo de requisições/respostas. Finalmente, os usuários de serviços RAS são livres para listar, recuperar e substituir os dados submetidos para uma dada instância de atividade de análise. Dessa maneira, serviços GEAS-RAS cumprem os requisitos para o nível 3 do Modelo de Maturidade de Richardson.

A Tabela 10 resume os resultados do estudo da aderência dos repositórios de serviços de análise NCBI, EBI, GEAS e GEAS-RAS comparados ao Modelo de Maturidade RESTful de Richardson.

9.2 Avaliação do esforço de desenvolvimento

A abordagem *ad hoc* para a adaptação de uma ferramenta de análise como um serviço *web* exige que o desenvolvedor desse serviço possua um conhecimento adequado de plataformas de execução e *frameworks* dedicados para esse tipo de solução, bem como um domínio significativo das restrições desse estilo arquitetural. Nossa abordagem de desenvolvimento de serviços baseada em modelos busca reduzir essa barreira inicial por meio do uso de modelos desenvolvidos usando uma linguagem

Tabela 10 – Aderência dos repositórios de serviços de análise comparados ao Modelo de Maturidade de Richardson

Repositório	Orientado a recursos? (Nível 1)	Utiliza semântica dos métodos HTTP? (Nível 2)	Apresenta controles de *hypermedia*? (Nível 3)
NCBI	Não	Não	Não
EBI	Sim	Sim, porém apresenta apenas um método por *endpoint*	Não
GEAS	Sim	Sim, porém apresenta apenas um método por *endpoint*	Não
GEAS-RAS	Sim	Sim	Sim

Fonte: Autoria própria.

específica de domínio e um *framework* para a adaptação de ferramentas de análise configurado por meio desses modelos.

Com o intuito de avaliar o esforço de desenvolvimento de um serviço *web* segundo nossa abordagem, inicialmente comparamos o número de *endpoints* e operações providas por serviços RAS frente a serviços *ad hoc* equivalentes. Em seguida, comparamos o esforço necessário para o desenvolvimento de um serviço de análise utilizando o *framework* Activity-REST em relação ao esforço necessário para o desenvolvimento de um mesmo serviço utilizando uma solução *ad hoc*. Essas medidas nos apresentam uma visão geral das características de cada serviço frente ao esforço de desenvolvimento dispendido em sua implementação.

9.2.1 Comparação do número de *endpoints*

O uso do *framework* Activity-REST permite a provisão de um serviço de análise com um amplo conjunto de *endpoints* e operações, ao mesmo tempo que busca reduzir o esforço necessário para o desenvolvimento de um serviço se comparado à implementação *ad hoc* de um serviço equivalente. A Tabela 11 apresenta o número de *endpoints* e de operações acessíveis para cada instância de atividade de análise criada nos serviços dos repositórios GEAS e GEAS-RAS. Os números apresentados representam os *endpoints* e operações hierarquicamente relacionadas a uma única instância de atividade de análise no serviço, incluindo todos os *endpoints* associados a cada estado possível de um instância de atividade de análise.

Tabela 11 – Número de *endpoints* e operações de uma instância de atividade de análise suportada.

Serviço	GEAS		GEAS-RAS	
	Endpoints	Operações	Endpoints	Operações
DAVID Chart Report	9	9	34	49
Enrichment Analysis	6	6	25	37
Gene-set enrichment Analysis	10	10	33	48
Kegg Pathway Viewer	8	8	28*	41*
One-color Affymetrix Microarray Normalization	7	7	16*	25*
One-color Agilent Microarray Normalization	6	6	16	25
Genepix Microarray Normalization	7	7	19	29
K-Means Clustering of Microarray Data	9	9	28*	41*
Hierarchical Clustering of Microarray Data	11	11	22*	33*
Hierarchical Microarray Cluster Viewer	9	9	25	39
RNA-Seq Differential Analysis	11	11	28*	42*
One-Color Microarray T-Test	9	9	28	42
One-Color Microarray Fold-Change	9	9	22	35
Two-Color Microarray T-Test	7	7	19	29
Two-Color Microarray Fold-Change	7	7	25	37

Fonte: Autoria própria. Os números apresentados representam a quantidade mínima de *endpoints* e operações providos pelo serviço para uma instância de atividade de análise executada, considerando todos os possíveis estados dessa instância. Um asterisco indica que a quantidade real de *endpoints* para a instância de atividade de análise pode ser maior que a apresentada, uma vez que a atividade de análise lida com conjuntos de dados que podem conter múltiplos arquivos e o modelo RAS define que cada arquivo enviado deve ser acessível por meio de um *endpoint* próprio.

Uma vez que os serviços GEAS-RAS concretizam completamente o modelo RAS para atividades de análise, estes proveem mais *endpoints* e, por consequência, mais operações para cada instância de atividade de análise que os respectivos serviços presentes no repositório GEAS. Serviços GEAS-RAS proveem controle e uso mais detalhado da instância de atividade de análise que a implementação *ad hoc* utilizada para os serviços GEAS. Por exemplo, arquivos de conjuntos de dados de entrada costumam ser inacessíveis após sua submissão em serviços GEAS. Estes mesmos conjuntos de dados se mantém acessíveis e podem ser posteriormente removidos nos serviços GEAS-RAS.

Alguns serviços GEAS proveem *endpoints* para a recuperação de valores

aceitos para um determinado parâmetro da atividade de análise. Estes *endpoints* não são contemplados pelo modelo RAS. Porém, serviços baseados no *framework* Activity-REST podem ser estendidos de maneira a incluir *endpoints* adicionais utilizando o suporte provido para a manipulação de atividades de análise. Dessa maneira, tais *endpoints* podem ser implementados (manualmente) para esses serviços.

9.2.2 Comparação do esforço de desenvolvimento

Nossa abordagem de desenvolvimento busca reduzir o esforço necessário para o desenvolvimento de um serviço se comparado à implementação *ad hoc* de um serviço equivalente. Nesta atividade de avaliação, utilizamos os serviços GEAS e GEAS-RAS como base para a comparação do esforço de desenvolvimento *ad hoc* de um serviço de análise com o esforço de desenvolvimento de um serviço por meio do suporte provido pelo *framework* Activity-REST.

De modo a avaliar o esforço de desenvolvimento de cada serviço, utilizamos como medida o número de linhas de código (*Lines Of Code*, LOC) implementadas para este serviço. O número de linhas de código é uma medida conhecida para a análise de esforço de desenvolvimento de um *software* qualquer. Estendemos o conceito de linhas de código para também considerar o número de linhas de uma descrição ActDL escritas para a definição de cada serviço. Porém, dado que a quantidade de linhas de código Java de cada serviço no repositório GEAS-RAS é constante para todos os serviços implementados e este código é obtido automaticamente por meio do suporte *framework* Activity-REST, decidimos desconsiderar o código Java de um serviço RAS durante essa comparação. Assim, estimamos a razão entre os esforços de desenvolvimento por meio da comparação do número de linhas de código implementadas para um serviço GEAS com o número de linhas da descrição ActDL que define o serviço RAS equivalente, uma vez que estes valores refletem mais precisamente o esforço necessário para a implementação de cada serviço. O valor estimado para a redução do esforço de desenvolvimento foi calculado de acordo com a equação 9.1.

$$Redução\ do\ esforço\% = \frac{(GEAS\ LOC) - (ActDL\ LOC)}{(GEAS\ LOC)} \times 100 \qquad (9.1)$$

A Tabela 12 compara o número de linhas de código implementadas em cada serviço GEAS (*Java LOC*) com o número de linhas da descrição ActDL correspondente (*ActDL LOC*). Esta tabela também apresenta um valor estimado para a redução do esforço de desenvolvimento. Em média, o uso do *framework* Activity-REST reduziu em 93% o esforço de desenvolvimento de um serviço se comparado ao uso de uma abordagem *ad hoc* de desenvolvimento.

Tabela 12 – Esforço de desenvolvimento nos repositórios GEAS e GEAS-RAS

Serviço	GEAS Java LOC	GEAS-RAS ActDL LOC	Redução do esforço
DAVID Chart Report	643	40	93.8%
Enrichment Analysis	377	28	92.6%
Gene-set enrichment Analysis	407	40	90.2%
Kegg Pathway Viewer	467	36	92.3%
One-color Affymetrix Microarray Normalization	285	14	95.1%
One-color Agilent Microarray Normalization	285	14	95.1%
Genepix Microarray Normalization	288	25	91.3%
K-Means Clustering of Microarray Data	376	56	85.1%
Hierarchical Clustering of Microarray Data	591	21	96.4%
Hierarchical Microarray Cluster Viewer	591	21	96.4%
RNA-Seq Differential Analysis	436	32	92.7%
One-Color Microarray T-Test	603	33	94.5%
One-Color Microarray Fold-Change	603	27	95.5%
Two-Color Microarray T-Test	399	31	92.2%
Two-Color Microarray Fold-Change	399	25	93.7%

Fonte: Autoria própria.

9.3 Avaliação de desempenho do *framework* Activity-REST

O encapsulamento de uma ferramenta de análise de linha de comando por meio de um adaptador *web* impõe um conjunto de penalidades ao desempenho da execução da atividade de análise provida por meio dessa ferramenta. Além do custo normal de execução da atividade de análise sobre os conjuntos de dados de interesse, o uso de um adaptador adiciona um custo referente ao conjunto de operações realizadas por esse adaptador antes e depois da execução da ferramenta adaptada. Adicionalmente, há também um custo associado à transferência dos conjuntos de dados entre cliente e serviço por meio de uma rede, bem como referente à execução desse cliente. Dado este contexto, esta atividade avalia o desempenho dos serviços e dos clientes RAS na execução de atividades de análise por meio da comparação do tempo da execução de uma atividade de análise por meio do uso de um serviço RAS em relação ao tempo de execução desta atividade de análise utilizando diretamente a ferramenta subjacente a esse serviço, bem como em relação ao tempo de execução desta atividade de análise por meio de um serviço *ad hoc*.

De modo a mensurar o tempo de execução de uma atividade de análise, inicialmente definimos um conjunto de cenários de teste para a execução desta atividade. Cada cenário de teste define um método de execução de uma dada atividade de análise, i.e., diretamente utilizando a ferramenta de análise (caso base), por meio de um serviço *ad hoc* GEAS ou por meio de um serviço GEAS-RAS. Esses cenários foram definidos de maneira mensurar os custos adicionais associados à execução da atividade de análise pelos diferentes tipos de serviços e/ou clientes.

Em seguida, definimos um ambiente para a execução desses cenários de teste com o objetivo de remover influências externas ao tempo de execução de cada atividade de análise. Posteriormente, escolhemos um conjunto de atividades de análise com diferentes características para serem executadas segundo os cenários de teste definidos. Nessa seleção, procuramos escolher atividades com diferentes tamanhos de conjuntos de dados e diferentes tempos de execução entre as atividades de análise disponíveis nos repositórios GEAS e GEAS-RAS.

Após a seleção das atividades de análise, realizamos a execução das mesmas em cada um dos cenários de teste definidos. Essa execução foi realizada de maneira iterativa para cada atividade de análise em cada cenário, registrando marcadores de tempo para o início e o fim de cada iteração. Por fim, comparamos os tempos de execução de uma mesma atividade de análise nos diferentes cenários de teste, obtendo uma razão de desempenho entre os diferentes métodos de execução da atividade de análise.

9.3.1 Definição dos cenários de teste de desempenho

Como primeira etapa desta avaliação, definimos um conjunto de cenários de teste de desempenho. Cada cenário de teste define um modo para a execução da atividade de análise e para o registro do tempo necessário para essa execução. De modo a obter um valor consistente para o tempo de execução, em todos os cenários realizamos iterativamente um número de execuções de cada atividade de análise em teste, registrando marcadores de tempo no início e no final de cada iteração. A partir desses marcadores de tempo, obtivemos valores médios e totais para a execução de cada atividade de análise em teste em cada cenário.

Os seguintes cenários de teste foram considerados:

CTF1: **Execução direta da ferramenta de análise (*benchmark*)**. Todas iterações são realizadas externamente à ferramenta de análise invocada. O registro do tempo de cada iteração ocorre antes e depois da ferramenta de análise executar;

CTF2: **Execução do serviço GEAS por meio de um cliente *ad hoc***. Este cliente utiliza a estratégia de *polling* durante a execução da atividade de análise. Todas as iterações são realizadas internamente ao processo do cliente. O registro do tempo de cada iteração ocorre internamente ao cliente, antes e depois de todas as interações cliente-serviço necessárias à execução de uma instância da atividade de análise serem executadas;

CTF3: **Execução do serviço RAS por meio de um cliente *ad hoc***. Este cliente utiliza a estratégia de *polling* durante a execução da atividade de análise. Todas as iterações são realizadas internamente ao processo do cliente. O registro do tempo de cada iteração ocorre internamente ao cliente, antes e depois de todas as interações cliente-serviço para a execução de uma instância da atividade de análise serem executadas.

O método de execução das atividades de análise e obtenção dos marcadores de tempo possui especificidades inerentes a cada cenário de avaliação. No cenário CTF1, as ferramentas de análise são executadas localmente por meio de um *script*. Este *script* invoca iterativamente cada ferramenta por meio de sua interface CLI original, registrando marcadores de tempo antes de cada invocação e após o término da execução da ferramenta. Esses marcadores de tempo são gerados por meio do comando `date` do *shell* `bash`. Dessa maneira, obtivemos os valores base para a comparação do tempo de execução nos demais cenários.

Os clientes *ad hoc* utilizados para os cenários CTF2 e CTF3 possuem construção semelhante. Nesse sentido, ambos foram desenvolvidos como testes de unidade JUnit e executados por meio da ferramenta Maven. Durante a execução desses cenários, a ferramenta Maven é invocada e executa um método marcado como teste de unidade. Esse método itera internamente até o limite definido, executando a atividade de análise uma vez a cada iteração. Durante a execução da atividade de análise, ambos clientes realizam *polling* do estado dessa execução a cada $1000ms$. No início e no término de cada iteração, um marcador de tempo é registrado utilizando a API padrão para a representação de tempo da linguagem Java. Por fim, após todas as iterações serem executadas, o processo Maven termina.

Como parte dessa atividade de avaliação, consideramos também o desempenho dos clientes RAS. Neste sentido, definimos um conjunto de cenários de teste adicionais:

CTC1: **Execução do serviço RAS por meio de um cliente *ad hoc***. Este cenário é semelhante ao cenário de teste CTF3;

CTC2: **Execução do serviço RAS por meio de um cliente RAS-*Pooling***. Este cliente foi gerado automaticamente a partir do *framework* Activity-REST e

utiliza a estratégia de *polling* durante a execução da atividade de análise. Todas as iterações são realizadas externamente ao processo do cliente. O registro de tempo ocorre antes e depois do processo cliente executar; e

CTC3: **Execução do serviço RAS por meio de um cliente RAS-SSE**. Este cliente foi gerado automaticamente a partir do *framework* Activity-REST e utiliza a estratégia de notificação por SSE durante a execução da atividade de análise. Todas as iterações são realizadas externamente ao processo do cliente. O registro de tempo ocorre antes e depois do processo cliente executar.

A execução dos testes para os cenários CTC2 e CTC3 ocorre de maneira um pouco diferente do cenário CTC1. Em todos os cenários são utilizados os clientes RAS. Porém, enquanto os clientes *ad hoc* usados no cenário CTC1 foram definidos para executar diversas iterações a cada invocação, os clientes RAS usados nos cenários CTC2 e CTC3 são definidos de modo a executar apenas uma instância de atividade de análise. Para contornar essa limitação sem modificar esses clientes, a iteração para a execução sequencial das atividades de análise foi definida por meio de um *script* que invoca cada cliente RAS repetidas vezes, como ocorre com a ferramenta de análise no cenário CTF1. Antes e depois de cada invocação, o *script* registra os marcadores de tempo de início e de final da iteração utilizando o comando `date`. Adicionalmente, utiliza-se a estratégia de espera utilizando *polling* no cenário CTC2, enquanto utiliza-se a estratégia de espera até o recebimento de um evento via SSE no cenário CTC3.

Por construção, o método de execução para os cenários CTC2 e CTC3 inclui o tempo necessário para inicializar a máquina virtual Java (*Java Virtual Machine*, ou JVM) entre dois marcadores de tempo da iteração, o que não ocorre no cenário CTC1, usado como base de comparação. Porém, o custo de inicialização da JVM, ainda que presente atualmente em nosso teste, pode ser reduzido futuramente pela evolução desta, bem como pelo uso de outras máquinas virtuais compatíveis e/ou compilação para um executável nativo, possibilitado por projetos como o GraalVM [131]. Frente a essa possibilidade, e no intuito de descobrir o desempenho de cada cliente desconsiderando o custo de inicialização da JVM, dois cenários adicionais foram definidos:

CTC2-*b*: **Execução do serviço RAS por meio de um cliente RAS-*Pooling* modificado**. Este cliente foi gerado automaticamente a partir do *framework* Activity-REST e utiliza a estratégia de *polling* durante a execução da atividade de análise. Todas as iterações são realizadas internamente ao processo do cliente. O registro do tempo de cada iteração ocorre internamente ao cliente, antes e depois de todas as interações cliente-serviço para a execução de uma instância

da atividade de análise serem executadas; O registro de tempo ocorre antes e depois do processo cliente executar;

CTC3-*b*: **Execução do serviço RAS por meio de um cliente RAS-SSE modificado**. Este cliente foi gerado automaticamente a partir do *framework* Activity-REST e utiliza a estratégia de notificação por SSE durante a execução da atividade de análise. Todas as iterações são realizadas internamente ao processo do cliente. O registro do tempo de cada iteração ocorre internamente ao cliente, antes e depois de todas as interações cliente-serviço para a execução de uma instância da atividade de análise serem executadas.

Os cenários de teste CTC2-*b* e CTC3-*b* são análogos aos cenários CTC2 e CTC3, respectivamente. Porém, estes novos cenários realizam a iteração e o registro de tempo internamente ao cliente RAS. Para isso, os clientes gerados para esses cenários tiveram seus códigos-fontes modificados para incluir essa iteração.

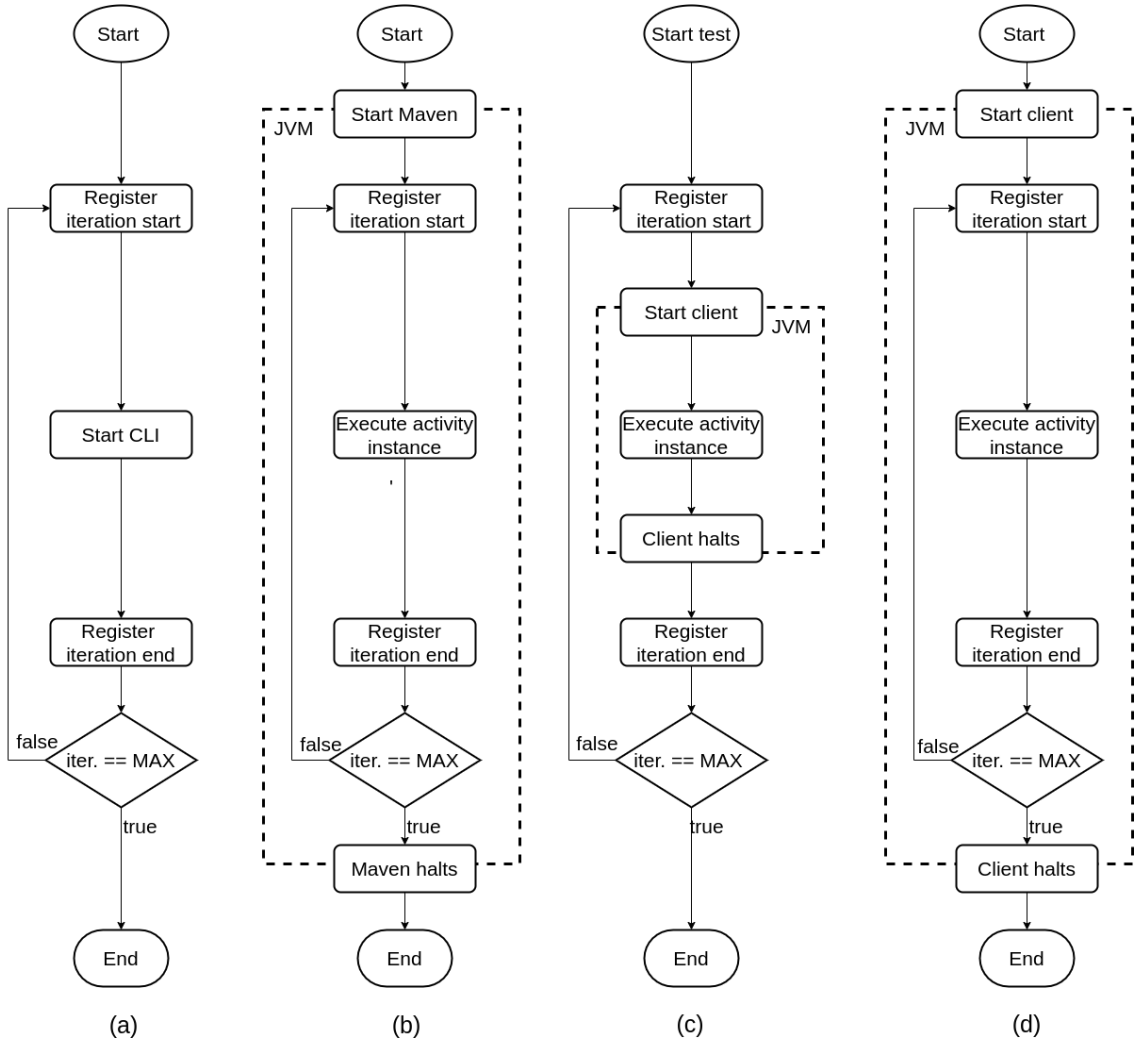
A Figura 59 apresenta um fluxograma em alto nível da execução dos cenários de análise. Um retângulo tracejado indica os eventos que ocorrem durante a execução da JVM (i.e., o processo do cliente *ad hoc* ou RAS) nos diferentes cenários. O registro de tempo da iteração ocorre fora desse processo nos cenários representados pelos fluxogramas (b) e (d), enquanto ocorre dentro do processo da JVM nos cenários representados pelo fluxograma (c).

9.3.2 Definição do ambiente de execução

O ambiente de execução consistiu de um *cluster* virtual gerenciado por meio do sistema de orquestração Kubernetes [132]. Nesse *cluster*, um conjunto de contêineres, i.e. máquinas virtuais isoladas, foram implantadas para a execução dos testes. O sistema de orquestração Kubernetes foi escolhido como sistema de gerenciamento dos contêineres desse cluster por ser uma tecnologia *open source*, de fácil aplicação e disponível para uso em diferentes ambientes de programação em nuvem disponíveis atualmente.

A configuração dos softwares instalados nos contêineres de avaliação foi definida por meio de um conjunto de imagens Docker. Uma imagem Docker consiste de uma imagem de disco estática e compartilhável, definida por meio de arquivos de configuração que indicam o sistema operacional e demais *softwares* que devem estar presentes nos contêineres derivados daquela imagem. Por fim, o *cluster* Kubernetes do ambiente de execução foi alojado utilizando a infraestrutura do Google Cloud Computing Services (Google Cloud) [133]. Este ambiente de execução foi utilizado de modo a facilitar a reprodução dos testes realizados.

Figura 59 – Fluxograma de um teste nos diferentes cenários



Fonte: Autoria própria. a) Fluxograma de um teste no cenário CTF1; b) Fluxograma de um teste nos cenários CTF2 e CTF3; c) Fluxograma de um teste nos cenários CTC2 e CTC3; e d) Fluxograma de um teste nos cenários CTC2-*b* e CTC3-*b*.

A Tabela 13 apresenta as principais características técnicas do *cluster* Kubernetes criado. O tipo de máquina alocado para o sistema, `n1-standard-4`, consiste de uma máquina de propósito geral com quatro CPUs virtuais e 15GB de memória. Três dessas máquinas foram disponibilizadas para a execução dos testes (campo "Número de nós"). O tipo de imagem utilizado para a execução do *cluster* Kubernetes (COS) consiste de uma imagem de sistema operacional otimizada para a execução de contêineres Docker. O tipo e o tamanho do disco, `pd-standard` e 20GB, respectivamente, indicam que a máquina física em que o cluster é executado aloca esse espaço para a execução do *cluster* em discos rígidos comuns (não-SSD).

Três contêineres foram implantados no *cluster* de avaliação: o contêiner *Benchmark*, no qual os *scripts* e os clientes são executados, e os contêineres *GEAS* e *GEAS-RAS*, nos quais os serviços presentes nesses repositórios são executados.

Tabela 13 – Características técnicas do cluster Kubernetes utilizado na avaliação

Atributo	Valor
Fornecedor	Google Cloud (Kubernetes Engine)
Versão do cluster	1.15.12-gke.2
Zona	us-central1-a
Tipo de máquina	n1-standard-4
Número de nós	3
Memória	15GB
Tipo de imagem	Container-optmized OS (COS)
Tipo de disco	pd-standard
Tamanho do disco	20GB

Fonte: Autoria própria.

A Figura 60 apresenta uma visão geral da organização do cluster de teste. Uma linha pontilhada representa os limites de um contêiner. Componentes de interesse nesses contêineres são representados como retângulos nomeados. Relações entre dois componentes são representadas como setas. Dentro do cluster, componentes em diferentes contêineres comunicam-se apenas por meio da rede privada local.

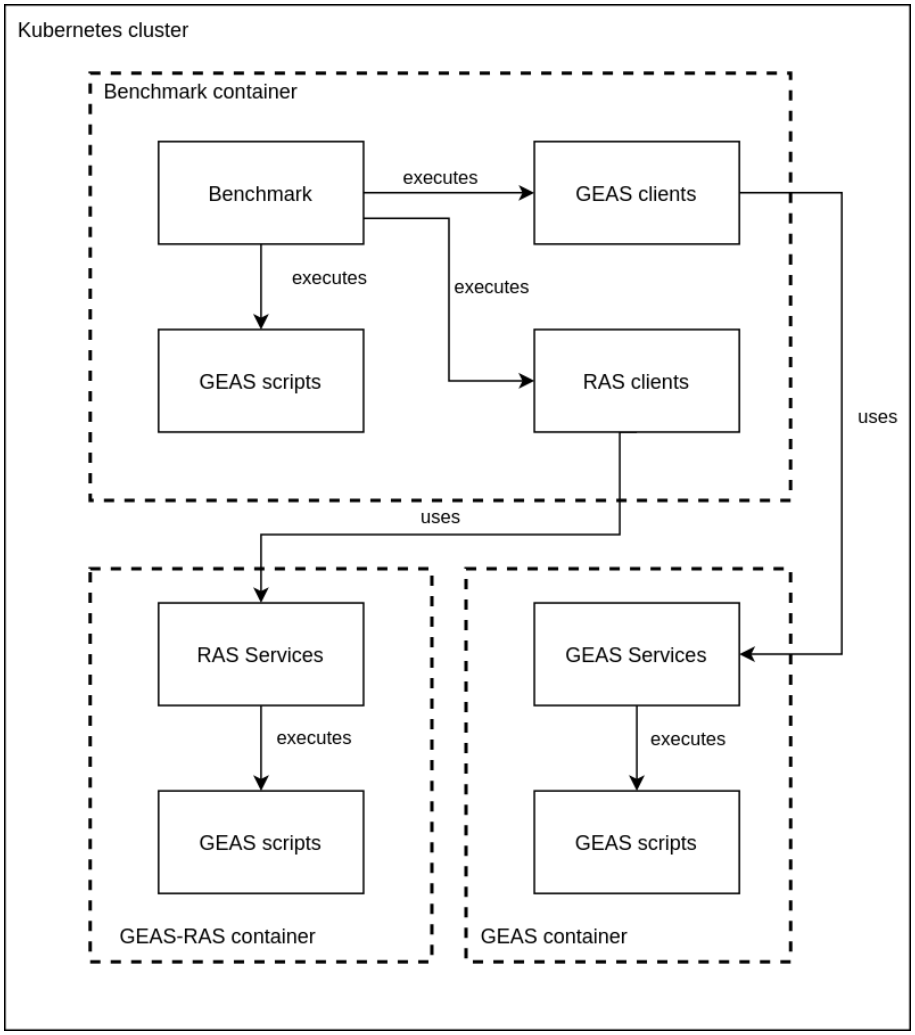
O contêiner *Benchmark* contém uma cópia das ferramentas de análise de expressão gênica, uma cópia de cada cliente utilizado em um cenário de avaliação, scripts para a execução automatizada dessas avaliações e outras dependências e/ou programas de suporte. Os contêineres *GEAS* e *GEAS-RAS* contém também uma cópia das ferramentas de análise de expressão gênica, bem como servidores de aplicação Tomcat (versão 9) e os artefatos que implementam os adaptadores *web* para as ferramentas de análise dos respectivos repositórios. As imagens Docker utilizadas durante esta avaliação estão disponíveis no repositório Docker Hub [134].

9.3.3 Definição das atividades de análise a serem executadas

As atividades de análise *Gene-set enrichment analysis*, *Agilent's one-color microarray normalization* e *Differential analysis of RNA-Seq data* foram escolhidas para avaliar o desempenho dos serviços e clientes RAS. Estas atividades apresentam diferentes tamanhos de conjuntos de dados de entrada e saída, bem como diferentes tempos de processamento. Todas as ferramentas de análise que suportam essas atividades foram executadas utilizando o ambiente de execução R (versão 3.6.3-2), presente em todos os contêineres de teste.

A atividade *Gene-set enrichment analysis* provê suporte à análise de enri-

Figura 60 – Visão geral da arquitetura do cluster de avaliação



Fonte: Autoria própria.

quecimento de conjuntos de genes em dados de expressão gênica. A ferramenta de análise adaptada, `geas-gene-set-enrichment.R`, recebe um conjunto de dados de expressão gênica já normalizados representando amostras em diferentes condições experimentais. Adicionalmente, esta ferramenta também recebe um conjunto de dados que descreve o mapeamento de grupamentos de genes para termos de um vocabulário normalizado, bem como um conjunto de parâmetros de execução. Após a execução, a atividade de análise produz um arquivo de saída contendo uma lista de termos do vocabulário normalizado associados ao genes diferencialmente expressos nas amostras do conjunto de dados de entrada.

A atividade *Agilent's one-color microarray normalization* pré-processa um conjunto de arquivos gerados na plataforma *Agilent One-color Microarray*. A ferramenta de análise adaptada, `geas-agilent-1-color-microarray-norm.R`, recebe ao menos dois arquivos de dados de expressão gênica gerados pela plataforma Agilent. Após sua execução, a atividade de análise produz um único conjunto de dados de

saída contendo os dados de expressão gênica normalizados.

Por fim, a atividade *Differential analysis of RNA-Seq data* identifica genes diferencialmente expressos em dados de RNA-Seq por meio do uso de um modelo de distribuição binomial negativa. A ferramenta de análise adaptada, `geas-deseq2-analysis.R`, recebe um conjunto de dados de entrada contendo dados de expressão gênica produzidos por RNA-Seq já normalizados e produz um arquivo contendo os resultados da análise diferencial.

9.3.4 Resultados da execução das atividades de análise

Após a execução das atividades de análise nos diferentes cenários de teste, compilamos os resultados obtidos e calculamos o tempo médio para a execução das instâncias da atividade de análise em cada cenário, bem como o tempo total para essa execução e o custo frente a um caso base.

9.3.4.1 Testes de desempenho dos serviços RAS

A Tabela 14 apresenta o tempo necessário para realizar 100 execuções de cada atividade de análise nos cenários CTF1, CTF2 e CTF3. Esta tabela apresenta a atividade de análise, o tamanho do conjunto de dados utilizado no teste, o cenário de avaliação, o número de iterações realizado em cada cenário, o tempo médio de uma iteração, o tempo total para executar todas as iterações e, por fim, o custo de desempenho do cenário frente ao *benchmark* (cenário CTF1).

Cada execução da atividade *Gene-set enrichment analysis* fez uso de dois conjuntos de dados de entrada totalizando 3.1MB e resultou em um conjunto de dados de saída de 2.4MB, totalizando 5.5MB. Tanto o serviço *ad hoc* disponibilizado pelo repositório GEAS quanto o serviço GEAS-RAS apresentaram um custo de cerca de 20% sobre o caso base (execução local). Essa diferença frente ao caso base pode ser atribuída ao baixo tempo necessário para o processamento da atividade de análise, cerca de 3 segundos, quando comparado ao tempo de transmissão dos conjuntos de dados por meio da rede privada do *cluster*.

Cada execução da atividade de análise *Agilent's one color microarray normalization* fez uso de 18 arquivos de cerca de 6,6MB cada em seu conjunto de dados de entrada, num total de 118MB, e resultou em um conjunto de dados de saída contendo um arquivo de cerca de 10MB, totalizando 128MB em arquivos de conjuntos de dados trafegados a cada execução. O serviço GEAS apresentou um custo de cerca de 35% frente ao caso base, enquanto o serviço GEAS-RAS apresentou um custo de cerca de 25% frente a esse mesmo cenário. Essa diferença média de pouco mais de 30% frente ao caso base pode ser atribuída ao tempo necessário para transferir os conjuntos de dados da análise por meio da rede privada do *cluster*.

Finalmente, cada execução da atividade de análise *Differential analysis of RNA-Seq data* fez uso de seis arquivos em seu conjunto de dados de entrada totalizando cerca de 4.7MB e resultou em um arquivo de 1.1MB em seu conjunto de dados de saída, totalizando cerca de 5.8MB em arquivos de conjuntos de dados trafegados a cada execução. Neste cenário, o serviço GEAS e o serviço GEAS-RAS apresentaram um custo adicional sobre o caso base de cerca de 3% e 8%, respectivamente.

A diferença de custo em desempenho para a execução da atividade de análise *Differential analysis of RNA-Seq data* nos serviços GEAS e GEAS-RAS, menor que a encontrada nas atividades de análise anteriores, revela um ponto em que o tamanho dos conjuntos de dados e a vazão da rede tornam-se menos relevantes conforme o tempo de execução da atividade de análise aumenta. Neste sentido, ambas as atividades *Gene-set enrichment analysis* e *Differential analysis of RNA-Seq data* utilizaram conjuntos de dados de tamanho similares, porém apresentam tempos de execução no cenário CTF1 diferentes. *Differential analysis of RNA-Seq data*, que apresentou um tempo médio de execução maior que *Gene-set enrichment analysis* no cenário base, apresentou um custo adicional menor nos cenários CTF2 e CTF3.

Tabela 14 – Tempos para a execução das atividades de análise nos cenários CTF1, CTF2 e CTF3

Atividade	Conj. de dados (MB)	Cenário	Iterações	Δt médio	t_{total}	Custo frente CTF1 (%)
Gene-set enrichment	5.5MB	CTF1	100	03.42s	05min 43s	-
		CTF2	100	04.12s	06min 52s	20%
		CTF3	100	04.12s	06min 52s	20%
Agilent one color μarray normalization	128MB	CTF1	100	06.05s	10min 05s	-
		CTF2	100	08.17s	13min 37s	35%
		CTF3	100	07.60s	12min 40s	25%
Differential analysis of RNA-Seq	5.8MB	CTF1	100	19.51s	32min 31s	-
		CTF2	100	20.27s	33min 47s	3%
		CTF3	100	21.22s	35min 22s	8%

Fonte: Autoria própria.

9.3.4.2 Testes de desempenho dos clientes RAS

A Tabela 15 apresenta o tempo necessário para realizar 100 execuções de cada atividade de análise nos cenários CTC1, CTC2 e CTC3. Esta tabela apresenta a mesma estrutura da Tabela 14, representando a atividade de análise, o tamanho do

conjunto de dados utilizado no teste, o cenário de avaliação, o número de iterações realizado em cada cenário, o tempo médio de uma iteração, o tempo total para executar todas as iterações e, por fim, o custo adicional frente a cenário base (CTC1). Os mesmos conjuntos de dados de entrada e saída foram utilizados nesses cenários.

Tabela 15 – Tempos para a execução das atividades de análise nos cenários CTC1, CTC2 e CTC3

Atividade	Conj. de dados (MB)	Cenário	Iterações	Δt médio	t_{total}	Custo frente CTC1 (%)
Gene-set enrichment	5.5MB	CTC1	100	04.12s	06min 52s	-
		CTC2	100	05.12s	08min 32s	24%
		CTC3	100	05.11s	08min 31s	24%
Agilent one color μarray normalization	128MB	CTC1	100	07.60s	12min 40s	-
		CTC2	100	11.19s	18min 86s	47%
		CTC3	100	11.86s	19min 46s	56%
Differential analysis of RNA-Seq	5.8MB	CTC1	100	21.22s	35min 22s	-
		CTC2	100	23.11s	38min 31s	9%
		CTC3	100	22.82s	37min 52s	7%

Fonte: Autoria própria.

Frente ao cenário CTC1, a execução dos cenários CTC2 e CTC3 apresentou um custo mais relevante conforme a quantidade e tamanho dos arquivos para os conjuntos de dados submetidos para a execução da atividade de análise aumenta. Esse custo também mostra-se menos relevante conforme o tempo de execução da atividade de análise aumenta. Neste sentido, a atividade *Agilent One-color Microarray Normalization*, que transfere 19 arquivos durante a execução, totalizando 128MB de dados, apresentou a pior custo em desempenho, de cerca de 50% ao cenário CTC1. As atividades *Gene-set enrichment* e *Differential analysis of RNA-Seq*, que submetem conjuntos de dados menores, apresentaram um custo em desempenho menor que *Agilent one-color microarray normalization* quando executadas por meio de serviços GEAS e GEAS-RAS. *Gene-set enrichment* apresenta o segundo maior custo de desempenho para o serviços de análise em relação ao cenário CTC1, o qual representa um aumento do tempo médio de execução da atividade de análise em 24% em ambos cenários. Por fim, a atividade *Differential analysis of RNA-Seq* apresenta o menor custo em desempenho para os serviços de análise em relação ao cenário CTC1, de 7 a 9% em cada caso. Comparando *Gene-set enrichment* e *Differential analysis of RNA-Seq*, vemos que ambas as atividades transferem conjuntos de dados

de tamanho similar. Porém, *Differential analysis of RNA-Seq* apresenta um tempo de execução maior no cenário base (CTF1, a execução direta da ferramenta de análise) que *Gene-set enrichment*. Isso demonstra que tempos de execução do caso base maiores tornam menos relevante o custo em desempenho decorrente da utilização serviços de análise.

A Tabela 15 apresenta o tempo necessário para realizar 100 execuções de cada atividade de análise nos cenários CTC1, CTC2-*b* e CTC3-*b*. Novamente, a tabela apresenta a atividade de análise, o tamanho do conjunto de dados utilizado no teste, o cenário de avaliação, o número de iterações realizado em cada cenário, o tempo médio de uma iteração, o tempo total para executar todas as iterações e, por fim, o custo adicional frente a cenário base (CTC1). Os mesmos conjuntos de dados de entrada e saída usados anteriormente foram utilizados nesses cenários.

Tabela 16 – Tempos para a execução das atividades de análise nos cenários CTC1, CTC2-*b* e CTC3-*b*

Atividade	Conj. de dados (MB)	Cenário	Iterações	Δt médio	t_{total}	Custo frente CTC1 (%)
Gene-set enrichment	5.5MB	CTC1	100	04.12s	06min 52s	-
		CTC2-*b*	100	03.67s	06min 07s	-11%
		CTC3-*b*	100	03.63s	06min 07s	-11%
Agilent one color μarray normalization	128MB	CTC1	100	07.60s	12min 40s	-
		CTC2-*b*	100	08.14s	13min 34s	7%
		CTC3-*b*	100	08.01s	13min 21s	5%
Differential analysis of RNA-Seq	5.8MB	CTC1	100	21.22s	35min 22s	-
		CTC2-*b*	100	20.96s	34min 56s	-1%
		CTC3-*b*	100	20.69s	34min 29s	-2%

Fonte: Autoria própria.

Nos cenários CTC2-*b* e CTC3-*b* modificamos os clientes RAS de forma a realizar internamente as 100 execuções da atividade de análise e o registro do tempo de seu tempo execução. Nestes cenários, os clientes utilizados aproximam-se (e por vezes superam) do desempenho do cenário CTC1. Comparando os resultados obtidos nestes cenários ao resultados obtidos nos cenário CTC2 e CTC3, nos quais as iterações e o registro de tempo são realizados extarnamente aos clientes RAS, revelamos o custo em desempenho causado pelo tempo de inicialização da JVM. A eventual melhora dos tempos das execuções nos cenários CTC2-*b* e CTC3-*b* frente ao cenário base (CTC1) pode ser decorrente das diferentes bibliotecas HTTP utilizadas

nesses cenários (REST-Assured para o cenário CTC1 e JAX-RS para os clientes utilizados nos demais cenários).

9.3.5 Avaliação dos resultados

Após a compilação dos resultados das execuções das atividades de análise nos diferentes cenários de teste, realizamos uma avaliação dos resultados obtidos.

9.3.5.1 Desempenho dos serviços RAS

Comparando-se os resultados obtidos nos cenários CTF2 e CTF3 aos resultados obtidos no cenário CTF1, obtivemos um entendimento sobre o custo para executar a atividade de análise por meio de serviços Web. A diferença de desempenho observada quando comparamos tanto serviços GEAS (cenário CTF2) quando serviços GEAS-RAS (cenário CTF3) frente ao caso base (cenário CTF1) representa os custos adicionais inerentes à execução de processamento remoto por meio de uma rede de computadores. Estes custos envolvem o tempo de transferência dos dados/arquivos por meio da rede, o manuseio desses dados pelo adaptador e o intervalo de espera utilizado entre duas requisições de *polling* enquanto a atividade de análise está executando. Neste sentido, os testes mostraram que o custo para a transferência de arquivos dos conjuntos de dados das atividades de análise executadas por meio de uma rede é um dos fatores de maior impacto no custo do uso de serviços *web* para a análise. Porém, esse custo adicional torna-se proporcionalmente menos relevante conforme o tempo de execução de uma atividade de análise cresce, como pôde ser observado na execução da atividade de análise *Differential analysis of RNA-Seq data*.

Comparando-se os cenários CTF2 e CTF3 entre si, mensuramos o custo de nossa solução para o desenvolvimentos de adaptadores por meio da interpretação de modelos ActDL frente ao uso de serviços adaptadores construídos de maneira *ad hoc*. Os testes mostraram que o uso do *framework* Activity-REST para a adaptação de interfaces de ferramentas de análise não apresenta uma diferença de desempenho significante para o usuário quanto comparado com o uso de uma implementação *ad hoc* mais simples. Por consequência, a abordagem proposta neste trabalho para a implementação de adaptadores de ferramentas de análise baseada na interpretação de um modelo ActDL apresenta um desempenho semelhante à solução de implementação *ad hoc* de um adaptador equivalente. Essa percepção nos leva a acreditar que, caso houvéssemos escolhido uma abordagem de geração de código do serviço adaptador a partir de um modelo ActDL ao invés da abordagem de interpretação de modelos do *framework* Activity-REST, o eventual ganho de desempenho não seria significativo.

9.3.5.2 Desempenho dos clientes RAS

Comparando-se os resultados obtidos nos cenários CTC2 e CTC3 com os resultados obtidos no cenário CTC1 mensuramos o custo adicional de inicializar o cliente do serviço de análise. Neste sentido, esses resultados parecem indicar que o custo para a inicialização da JVM e para a abertura dos arquivos dos conjuntos de dados de entrada é mais proeminente quando a execução da atividade de análise é de curta duração ou conforme o número de arquivos de entrada cresce.

Comparando-se os resultados obtidos nos cenários CTC2 e CTC3 mensuramos as diferenças de desempenho entre o uso da estratégia de *polling* e a estratégia de notificação via SSE dos clientes gerados. Observamos, nessa comparação, que a diferença entre os tempos médios de iteração é menor que 1s. Esse tempo é equivalente ao tempo utilizado entre duas chamadas de *polling* em nosso teste. Estes resultados não apresentaram vantagem em desempenho entre o uso da estratégia de *polling* e o uso de notificações assíncrona por meio de *Server-Side Events (SSE)* durante o período de execução da atividade de análise. Porém, durante a avaliação, utilizamos cenários em que as requisições de *polling* são frequentes (cerca de uma requisição por segundo).

Em um cenário de uso dos serviços de análise no qual uma grande quantidade de atividades de análise estejam executando simultaneamente, uma frequência alta de requisições de *pooling* pode consumir rapidamente os recursos disponíveis ao serviço e gerar uma situação de baixo desempenho e/ou negação de serviço. Dessa forma, em um cenário de uso normal destes serviços, pode ser desejável que clientes realizem requisições de *polling* menos frequentemente. Assim, a vantagem em desempenho no uso de SSE se mostraria mais evidente. Nestes cenários, a extensão do modelo RAS por meio de notificação assíncrona do estado final da análise utilizando SSE ajuda a evitar esse gargalo, reduzindo a quantidade de requisições entre cliente e serviço.

9.3.5.3 Considerações finais

Durante a avaliação do *framework* Activity-REST utilizamos para comparação o tempo médio de 100 iterações da execução de cada atividade de análise em cada diferente cenário de estudo. Este número de iterações foi suficiente para a comparação do tempo médio em cada cenário, uma vez que o isolamento do ambiente de teste proporcionou tempos de execução estáveis para todas as iterações de uma atividade de análise em cada cenário. Porém, caso quiséssemos realizar mais iterações em cada cenário, estaríamos limitados pelos serviços do repositório GEAS quanto ao número máximo de iterações possíveis. Isso ocorre pois, por construção, os serviços do repositório GEAS utilizam um mecanismo de transferência de arquivos que mantém uma cópia temporária dos arquivos enviados para os conjuntos de dados de entrada,

armazenada no diretório de arquivos temporários do servidor de aplicação Tomcat.

O armazenamento da cópia temporária dos arquivos transferidos não seria um problema em uma situação de menos intensa dos serviços GEAS, pois o espaço em disco ocupado por esses arquivos seria recuperado por um sistema coletor de lixo após algum tempo. Porém, dados a grande quantidade de execuções em um pequeno espaço de tempo e o tamanho dos conjuntos de dados para a atividade de análise *Agilent One-color Microarray Normalization* (128MB), todo o espaço disponibilizado pelo *cluster* para o contêiner GEAS seria rapidamente tomado durante o cenário CTF2. O uso excessivo do espaço de armazenamento do *cluster* é identificado pelo Kubernetes como potencial problema para a estabilidade dos demais contêineres. Em resposta, o Kubernetes destrói o contêiner "hostil" e o substitui por um novo, recém criado. Durante essa operação, de modo que os serviços GEAS se tornam inoperantes e, por consequência, os tempos de execução da atividade de análise tornam-se pouco confiáveis.

Esta limitação poderia ser contornada modificando-se os serviços GEAS de modo a remover ativamente esses arquivos do diretório temporário do servidor de aplicação. Porém, realizar essa modificação não seria adequado para nossa avaliação, pois modificaria a implementação desses serviços e, por consequência, do modelo de interação subjacente a eles, bem como dos custos em desempenho impostos por essa implementação. Por esta razão, decidimos manter o número de iterações em 100, o qual foi o número máximo de iterações que pudemos realizar em todos os cenários para todas as atividades de análise.

9.4 Críticas à validade das avaliações

Críticas podem ser feitas quanto a validade das avaliações apresentadas nesse capítulo. Quando à avaliação da maturidade dos serviços de análise disponíveis no domínio da bioinformática, podemos discutir ameaças à conclusão obtida pelo estudo. Neste sentido, não avaliamos exaustivamente todos os serviços de análise presentes na literatura da área, de modo que um estudo mais extenso dos serviços de análise existentes na área de bioinformática poderia revelar a existência de serviços RESTful tão maduros quanto os providos por nossa abordagem de desenvolvimento. Porém, nossa abordagem buscou prover uma visão geral do estado da arte dos serviços de análise em bioinformática ao avaliar serviços de análise representativos de um conjunto de repositórios de serviços de análise conhecidos.

Quanto à avaliação da redução do esforço de desenvolvimento de serviços de análise provida pelo *framework* Activity-REST frente ao desenvolvimento de um serviço de análise *ad hoc*, podemos discutir ameaças à validade externa do estudo.

Neste sentido, a avaliação de esforço foi realizada utilizando como aproximação desse esforço uma comparação entre a quantidade de linhas de código de um conjunto de serviços de análise existente e a quantidade de linhas da descrição ActDL definidas para obter um serviço análogo por meio do *framework* Activity-REST. Essa medida não captura diretamente o esforço necessário para aprender os conceitos de um modelo ActDL e para utilizar a abordagem de desenvolvimento proposta neste trabalho. Dessa maneira, seria interessante a realização de uma avaliação de esforço e usabilidade mais extensa e em conjunto com especialistas da área. Adicionalmente, embora tenhamos comparado serviços *ad hoc* com serviços RAS equivalentes, os serviços comparados possuem conjuntos diferentes de *endpoints* e operações, o que pode causar alguma confusão à primeira vista. Neste sentido, os serviços obtidos com nossa abordagem de desenvolvimento são mais complexos que os serviços *ad hoc* originais e, por esta razão, a medida baseada em linhas de código/modelo pode estar subestimando a redução real de esforço de desenvolvimento obtida pelo uso do *framework* Activity-REST,

Por fim, quanto à avaliação de desempenho de um serviço de análise RAS e dos diferentes clientes RAS desenvolvidos por meio do suporte do *framework* Activity-REST, podemos discutir ameaças à validade das conclusões pela construção do experimento. Nesse sentido, embora tenhamos executado todos os cenários de teste em um mesmo *cluster* Kubernetes, mantendo a mesma topologia dos serviços desse *cluster* durante todas as execuções, todas as iterações para cada cenário de teste foram executadas de maneira contígua para cada atividade de análise escolhida. Assim, quaisquer flutuações no nível de serviço disponibilizado pelo *Google Cloud Provider* que tenham ocorrido durante o período do experimento podem ter afetado de maneira assimétrica o tempo total de diferentes cenários dessa análise.

9.5 Considerações finais

Neste capítulo demonstramos que serviços de análise construídos por meio do *framework* Activity-REST são mais aderentes ao modelo RESTful, pontuando melhor frente ao Modelo de Maturidade de Richardson que outras soluções encontradas em repositórios de serviços de análise presentes no domínio biomédico. Também demonstramos que o desenvolvimento baseado em modelos para adaptadores de ferramentas de análise em bioinformática reduz consideravelmente o esforço de desenvolvimento quando comparado à implementação *ad hoc* de um serviço de análise equivalente. Por fim, demonstramos que a abordagem utilizada na implementação do *framework* Activity-REST, baseada na interpretação de modelos em tempo de execução, tem desempenho comparável ao encontrado durante a execução de um serviço de análise *ad hoc*. Dessa maneira, acreditamos que nossa solução baseada em

modelos facilita o desenvolvimento de serviços de análise maduros para o domínio biomédico sem penalizar o tempo de execução da atividade de análise, quando comparada a outras soluções semelhantes para alcançar o mesmo objetivo.

Em nossos testes dos clientes RAS-CLI, percebemos que estes apresentam um custo extra de cerca de dois segundos frente ao caso base em que o serviço é executado a partir de um processo em execução. Esse custo, relacionado ao carregamento e inicialização da máquina virtual Java utilizada para executar o código do cliente, é pequeno se comparado ao tempo total de execução de uma atividade de análise mais completa. Adicionalmente, esse custo pode ser reduzido futuramente de diferentes maneiras. Por exemplo, por meio da nossa abordagem de geração de código para esses clientes, é possível prover a geração de clientes compilados para serem executando diretamente sobre o sistema operacional hospedeiro.

10 Conclusão

Este trabalho teve o objetivo de investigar e propor uma abordagem baseada em modelos para o desenvolvimento de serviços *web* para a execução de análises em bioinformática por meio da adaptação de ferramentas de linha de comando existentes. Este capítulo apresenta as principais contribuições do trabalho e suas limitações, bem como compara os resultados obtidos com trabalhos relacionados. Por fim, o capítulo apresenta direções futuras para a pesquisa.

O restante desse capítulo está estruturado da seguinte maneira: a Seção 10.1 apresenta as principais contribuições do trabalho; a Seção 10.2 posiciona o trabalho frente a soluções existentes e abordagens alternativas, bem como discute suas limitações; por fim, a Seção 10.3 apresenta perspectivas para o desenvolvimento de trabalhos futuros.

10.1 Principais contribuições

Há um grande número de ferramentas de linha de comando desenvolvidas para o suporte à execução de atividades de análise em bioinformática. Adicionalmente, novas ferramentas são criadas diariamente por pesquisadores do domínio. Ao mesmo tempo, há um interesse crescente na criação e uso de serviços *web* para a execução de atividades de análise em bioinformática. Dessa maneira, um esforço para facilitar a obtenção de serviços *web* adaptadores para ferramentas de análise de linha de comando pode difundir as ferramentas de análise existentes, facilitando o acesso e uso das mesmas.

Neste sentido, as principais contribuições deste trabalho são: i) a definição de um modelo de referência para o desenvolvimento de serviços de análise em bioinformática; ii) a definição de um processo de desenvolvimento baseado em modelos e uma arquitetura de metamodelagem para o desenvolvimento de serviços de análise a partir de ferramentas de linha de comando existentes; iii) a implementação de uma infraestrutura de suporte ao desenvolvimento de serviços de análise aderentes ao modelo de referência provido, bem como de clientes e descrições para esses serviços; e, por fim, iv) a aplicação da metodologia e suporte definidos na reengenharia do repositório de serviços para análise de expressão gênica GEAS.

10.1.1 Modelo de referência para serviços de análise

O interesse crescente na criação e no uso de serviços de análise requer uma maior conceitualização da interface desses serviços e da interação entre serviço e cliente para a execução de uma atividade de análise. A padronização das interfaces dos serviços de análise facilita não apenas a composição destes serviços, bem como a geração automática da implementação de novos serviços, seus clientes e suas descrições.

Novos serviços *web* são providos diariamente para o acesso a bancos de dados biológicos e para a execução de atividades de análise de maneira remota. Enquanto a interface RESTful de um serviço *web* para acesso a banco de dados biológicos possui, em geral, grande dependência da conceitualização e organização do domínio representado por esses bancos de dados, a interface RESTful para a execução de atividades de análise possui uma menor dependência da conceitualização dos dados a serem utilizados nessa análise e maior acoplamento ao processamento desses dados.

A existência de um modelo de referência para a interface desses serviços facilita o entendimento e a integração de serviços de análise, bem como possibilita o suporte ao seu desenvolvimento, descrição e uso. Nesse sentido, este trabalho propôs o Modelo de Referência para Serviços de Análise RESTful (*RESTful Analysis Services Reference Model*, ou modelo RAS). O modelo RAS define a interface de um serviço de análise para bioinformática, bem como as interações entre cliente e serviço durante a execução de uma atividade de análise, de modo que esse serviço apresente uma interface RESTful madura e bem definida. Adicionalmente, o modelo de referência RAS guia a construção e a interação com um serviço de análise genérico e provê diretrizes para a sua concretização segundo as necessidades específicas de uma dada atividade de análise.

Por meio do modelo de referência RAS, podemos compreender a estrutura de um dado serviço de análise apenas conhecendo os nomes/identificadores e os tipos dos parâmetros de execução e dos conjuntos de dados de entrada e saída utilizados/criados pela atividade de análise a ser executada por meio deste serviço. Adicionalmente, um serviço construído por meio do modelo RAS provê facilidades para a navegação e descoberta do serviço usando controles de hipermídia retornados junto às respostas às requisições HTTP realizadas pelo usuário do serviço. Por fim, acreditamos que o modelo RAS possa ser aplicado com sucesso não apenas ao desenvolvimento de serviços de análise adaptadores, construídos a partir de ferramentas de análise de linhas de comando existentes, mas também ao desenvolvimento de serviços de análise monolíticos.

10.1.2 Abordagem de desenvolvimento baseada em modelos e arquitetura de metamodelagem para serviços adaptadores

Uma dificuldade recorrente associada ao desenvolvimento e ao uso de serviços de análise em bioinformática é a necessidade de conhecimento técnico relativo aos padrões e às tecnologias da *web*. Tal conhecimento não é tipicamente comum aos bioinformatas e biologistas que produzem ferramentas de análise durante suas pesquisas e que gostariam de disponibilizar essas ferramentas como serviços de análise completos. Por outro lado, estes especialistas conhecem o domínio de aplicação das ferramentas que constroem, bem como os parâmetros e os conjuntos de dados que estas ferramentas utilizam ou produzem durante sua execução. Dessa maneira, torna-se necessário a existência de metodologias e processos de desenvolvimento para serviços de análise que permitam a construção de um serviço de análise a partir de uma ferramenta de análise existente ao mesmo tempo que abstraiam os detalhes técnicos necessários para a construção desses serviços.

Frente a esse cenário, este trabalho propôs um processo de desenvolvimento baseado em modelos para a obtenção de serviços de análise adaptadores para ferramentas de análise de linha de comando existentes. O processo de desenvolvimento proposto utiliza modelos específicos de domínio para descrever atividades e ferramentas de análise em alto nível de abstração, assim como para descrever a implantação do serviço após sua criação. Adicionalmente, o processo de desenvolvimento proposto faz uso de transformações e interpretações dos modelos iniciais para obter serviços adaptadores, bem como clientes e descrições para esses serviços, a partir dos modelos iniciais.

Este trabalho também propôs uma arquitetura de metamodelagem associada ao processo de desenvolvimento proposto. Essa arquitetura provê a conceitualização base os modelos utilizados no processo de desenvolvimento, definindo os elementos do domínio que são essenciais para a obtenção dos serviços, clientes e descrições. Adicionalmente, também foram providas sintaxes abstratas e concretas para linguagens específicas de domínio utilizadas nesse processo, nomeadas *Activity Definition Language* (ActDL) e *Service Deployment Description Language* (SDDL).

10.1.3 Infraestrutura de suporte ao desenvolvimento de serviços adaptadores, seus clientes e descrições

De modo a facilitar a aplicação do processo de desenvolvimento baseado em modelos proposto, desenvolvemos um conjunto de artefatos de suporte a esse processo. Nomeadamente, implementamos o *framework* Activity-REST para a geração de um serviço adaptador por meio da interpretação dos modelos ActDL. Um arquétipo

Maven também foi desenvolvido para facilitar a criação, compilação e empacotamento do projeto Java que resultará no serviço adaptador. *Plugins* de suporte à criação e edição dos modelos utilizados no processo de desenvolvimento proposto foram implementados e disponibilizados para o ambiente de desenvolvimento Eclipse. Ferramentas para a transformação de modelos foram, também, providas, permitindo a geração de clientes e descrições dos serviço implementados. Finalmente, foi desenvolvido suporte para a obtenção de clientes de linha de comando e clientes para o ambiente integrado de análise Galaxy, bem como para a geração de descrições de serviços definidas nas linguagens WSDL e OpenAPI.

Com base na infraestrutura de suporte provida, um bioinformata pode obter um serviço adaptador para uma ferramenta de análise existente com um mínimo de esforço. Este serviço pode, então, ser implantado em um ambiente de execução Java Enterprise, tal como Tomcat ou Glassfish, de modo a adaptar a ferramenta de análise (previamente instalada na máquina hospedeira), e prover a execução remota da atividade de análise definida. Adicionalmente, as transformações de modelo implementadas automatizam a obtenção dos clientes e descrições para esses serviços. Por fim, para facilitar ainda mais o desenvolvimento de serviços de análise adaptadores, seus clientes e suas descrições, uma aplicação Web que integra nossa infraestrutura de suporte foi provida. Por meio dessa aplicação Web, um bioinformata pode obter serviços, clientes e descrições por meio de uma interface simplificada, acessível por meio de qualquer navegador *web*.

10.1.4 Reengenharia do repositório GEAS

Após o provimento de suporte ao uso do processo de desenvolvimento orientado a modelos para serviços de análise adaptadores, de seus clientes e suas descrições, aplicamos este processo na reengenharia dos serviços para análise de expressão gênica disponíveis no repositório GEAS. Os serviços GEAS haviam sido definidos de maneira *ad hoc*, sem uma conceitualização forte da interface HTTP destes serviços. Dessa maneira, os serviços GEAS podiam ser posicionados apenas no nível 2 do Modelo de Maturidade RESTful de Richardson. Adicionalmente, diferentes serviços GEAS apresentavam pequenos detalhes divergentes na organização de seus recursos.

Ao aplicar a metodologia e o processo de desenvolvimento orientado a modelos descrito por este trabalho, modernizamos os serviços GEAS por meio de uma interface RESTful mais madura desenvolvida segundo o modelo de referência RAS. Também identificamos serviços GEAS que, segundo o modelo conceitual de uma atividade de análise proposto, realizavam mais de uma atividade de análise. Em seguida, dividimos as responsabilidades pelas diferentes atividades de análise em serviços independentes. Durante esse processo, um novo repositório de serviços de análise de expressão gênica

foi obtido, o repositório GEAS-RAS.

Como etapa final do processo de reengenharia dos serviços GEAS, comparamos os serviços obtidos frente aos serviços iniciais quanto ao esforço de implementação, adesão aos princípios RESTful e eficiência. Durante essa avalização, mostramos que os serviços GEAS-RAS provêem mais *endpoints* e operações para a manipulação da instância de atividade de análise que os serviços GEAS, bem como são implementados de maneira mais padronizada e utilizando menor esforço. Também mostramos que os serviços GEAS-RAS provêem maior adesão aos princípios RESTful e possuem um desempenho comparável aos serviços GEAS originais.

10.2 Discussão

Embora a literatura apresente trabalhos relacionados ao uso de metodologias baseadas em modelos para o desenvolvimento de serviços *web*, essas metodologias apresentam algumas características que as tornam inadequadas para o uso direto no domínio de bioinformática. Neste sentido, as metodologias existentes para a obtenção de serviços *web* não abstraem completamente as especifidades desses serviços, não geram serviços completos ou apenas produzem serviços capazes de executar operações CRUD de forma simplificada

A arquitetura orientada a modelos com suporte ontológico para a modelagem e transformação de serviços proposta por Pahl [77] provê um conjunto de visões arquiteturais e metamodelos de referência para guiar o desenvolvimento de um dado serviço. Porém, o uso dessa metodologia demanda a criação de diversos modelos interrelacionados usando a linguagem *Web Ontology Language* (OWL) [135]. OWL consiste de uma linguagem para a representação de conhecimento utilizada principalmente na representação de conhecimento por meio de ontologias formais. Embora apresente uma base formal forte, a linguagem OWL não foi concebida para a modelagem de serviços e, portanto, não abstrai do desenvolvedor do serviço as complexidades inerentes de uma linguagem usada para um domínio tão amplo quanto a representação de conhecimento. Uma vez que o desenvolvimento de um novo serviço segundo a metodologia proposta por este trabalho requer a modelagem estrutural e comportamental do novo serviço por meio de OWL, o desenvolvedor interessado em utilizar a metodologia precisa tanto conhecer esta linguagem, pouco dominada fora de seu domínio de aplicação, quanto conhecer o subconjunto de elementos passíveis de serem utilizados nesses modelos. Isso torna essa abordagem pouco prática para um bioinformata interessado em rapidamente obter um serviço de análise para uma ferramenta existente.

O trabalho de Schreier [78] provê um metamodelo que pode ser utilizado para

a modelagem de um serviço RESTful em seus diversos aspectos estruturais e comportamentais. Dessa maneira, esse trabalho provê um passo além do que é provido pelas linguagens de descrição de serviços *web* atuais, focadas principalmente na descrição estrutural de um dado serviço. De maneira contrária à abordagem de Schreier, nosso trabalho buscou abstrair esses aspectos do processo de desenvolvimento, de modo a criar serviços padronizados segundo o modelo de referência RAS. Tal escolha foi realizada de modo a reduzir o esforço de modelagem e implementação do serviço, bem como para prover serviços mais adequados ao domínio da execução de uma atividade de análise.

Haupt [79] propõe uma abordagem orientada a modelos para o desenvolvimento de serviços RESTful. O usuário dessa metodologia precisa definir um conjunto de modelos interrelacionados, os quais são sucessivamente refinados até obter um modelo dependente de plataforma que pode ser transformado no código fonte base para um novo serviço. Esses modelos vão desde um modelo de domínio que descreve os tipos de dados manipulados pelo serviço, até modelos para a descrição de recursos da API RESTful e modelos para associar tais recursos aos URLs que devem ser encontrados no serviço obtido. A metodologia proposta busca obter serviços RESTful consistentes e corretos, porém demanda a modelagem de aspectos técnicos de um serviço REST, tais como a estrutura dos recursos e os relacionamentos entre esses recursos e seus URLs. Adicionalmente, esse trabalho não provê mecanismos para a adaptação de uma ferramenta de análise para a execução da atividade de análise, nem guia o usuário na conceitualização do domínio e interface do serviço necessária para manipular instâncias dessas atividades.

Os trabalhos de Bender *et al.* [21] e Ed-douibi *et al.* [24] propõe abordagens com objetivos similares. Estes trabalhos proveem métodos para a obtenção de serviços capazes de prover operações CRUD simples sobre seus dados. Tal característica demanda que operações mais complexas, tais como invocar uma ferramenta de análise existente e monitorar sua execução, sejam implementadas diretamente no código do serviço produzido. Por sua vez, Sferruzza [81] provê uma abordagem para obter um serviço com operações mais complexas que o CRUD, desde que essas operações estejam implementadas separadamente em uma dada linguagem de programação e sejam associadas/compostas por meio de anotações na descrição OpenAPI do novo serviço. Porém, esse trabalho é inadequado para o domínio da bioinformática o serviço produzido não segue um dado modelo de referência, nem abstrai do usuário os detalhes da modelagem RESTful, uma vez que utiliza uma descrição OpenAPI como modelo fonte.

O desenvolvimento de um serviço RAS requer a criação de um modelo AADM para descrever os aspectos principais da atividade de análise executada por este serviço e da adaptação da ferramenta de análise que executa essa atividade.

Adicionalmente, um modelo SDDM é necessário para a execução das transformações que permitem obter clientes e descrições para os serviços RAS produzido. De modo a facilitar a criação desses modelos fontes, este trabalho propôs sintaxes concretas textuais para a representação desses modelos, nomeadamente as linguagens ActDL e SDDL.

Uma alternativa ao uso de sintaxes textuais para os modelos AADM e SDDM seria o provimento de sintaxes gráficas para a representação desses modelos. O uso de uma sintaxe gráfica para a representação de um modelo pode facilitar o entendimento dos elementos descritos por um modelo mesmo entre especialistas de diferentes domínios. Adicionalmente, notações gráficas permitem o uso de diferentes formas, cores e ícones para a representação de diferentes elementos de um modelo, característica que pode facilitar a compreensão de modelos mais complexos.

Embora uma sintaxe gráfica possa facilitar a compreensão do modelo representado, o uso desse tipo de sintaxe tem suas limitações. Em geral, o uso de uma sintaxe gráfica demanda suporte de um editor dedicado, o qual deve interpretar a representação serializada de um modelo e apresentar uma visualização conveniente ao usuário na forma de um ou mais diagramas. Neste sentido, um usuário de nossa abordagem estaria limitado a manipular os modelos AADM e SDDM utilizando tais editores dedicados, restrição que não ocorre com o uso de sintaxes textuais manipuláveis por meio de qualquer editor de texto padrão. Adicionalmente, dado que modelos ActDL e SDDL possuem uma estrutura hierárquica simples com poucos elementos, o uso de uma sintaxe gráfica como forma padrão para a criação desses modelos não seria justificado.

O *framework* Activity-REST interpreta um modelo AADM e expõe os *endpoints* e operações RESTful adequados para a execução da atividade de análise modelada, bem como usa esse modelo para executar a ferramenta de análise subjacente. Neste sentido, o desenvolvedor de um dado serviço de análise deve produzir ou obter apenas um código-fonte mínimo para configurar o *framework* desenvolvido por meio dos modelos AADM e de implantação criados.

Uma abordagem alternativa à interpretação de um modelo AADM seria a geração de código do serviço de análise diretamente, a exemplo da abordagem utilizada para a geração de um cliente de um serviço RAS. A geração de código a partir do modelo geralmente é percebida como uma solução mais simples, sendo capaz de produzir um código diretamente inspecionável e personalizável. Adicionalmente, o código obtido por meio de uma transformação de modelos pode ser gerado sob medida para as necessidades apresentadas pelos modelos iniciais, o que pode permitir a obtenção de artefatos mais simples e que potencialmente apresentem melhor desempenho durante sua execução.

A interpretação de um modelo, por sua vez, tem como vantagem a não necessidade da regeração do código do serviço após alterações do modelo ActDL inicial, reduzindo esforços para a manutenção do serviço RAS caso mudanças sejam realizadas neste modelo (ou no serviço). Adicionalmente, pontos de extensão são providos pelo *framework* Activity-REST para a personalização de alguns aspectos do serviço RAS, tais como para a validação dos dados submetidos para as instâncias de atividade de análise. Dessa maneira, personalizações do serviço de análise podem ainda ser realizadas e não precisam ser reconciliadas após mudanças dos modelos inciais, como ocorreria caso utilizássemos uma abordagem baseada em geração de código para serviços RAS. Finalmente, a comparação de desempenho dos serviços GEAS-RAS frente aos serviços GEAS mostrou que boa parte do tempo e dos recursos utilizados durante a execução do serviço estão relacionados à transferência dos conjuntos de dados via rede e à execução da ferramenta de análise subjacente ao serviço. Assim, o uso de transformações de modelo para a criação de um serviço dedicado não proporcionaria ganhos de desempenho relevantes frente à nossa abordagem, baseada em interpretação.

Uma vez que o foco deste trabalho foi a adaptação de ferramentas de análise para expor uma interface RESTful para a execução das atividades de análise suportadas, não nos preocupamos acerca de aspectos de autenticação e privacidade durante o desenvolvimento do modelo de referência RAS e do *framework* Activity-REST. Neste sentido, caso o serviço RAS seja provido de maneira isolada, o usuário deste serviço precisa apenas conhecer o URL de uma dada instância de atividade de análise para ter acesso a essa instância e a todos os seus recursos relacionados.

Embora um usuário que deseje uma solução completa para o provimento de um serviço *web* possa considerar o não provimento de mecanismos de autenticação como uma limitação do trabalho, acreditamos que a definição e o uso desses mecanismos são responsabilidades complementares ao nosso problema e, por tanto, devem ser definidas separadamente. Neste sentido, diferentes estratégias podem ser utilizadas para prover mecanismos de autenticação e autorização a serviços RESTful não seguros. Por exemplo, um serviço de borda pode ser usado, impedindo o acesso direto ao serviço RAS e provendo mecanismos para autenticar usuários e autorizar suas requisições antes de retransmiti-las ao serviço RAS protegido. O serviço de borda poderia, ainda, inspecionar os controles de hipermídia retornados pelo serviço RAS junto às respostas ao usuário, de modo a recuperar os URLs de novos recursos criados e acessíveis pelo usuário e autorizar requisições futuras a esses recursos.

Outra limitação aparente de nosso trabalho está na falta de uma avaliação estruturada sobre o esforço necessário para o aprendizado e uso de nossa abordagem. Tal esforço poderia ser mensurado por meio de uma avaliação de usabilidade, envolvendo tanto o desenvolvimento de modelos ActDL e SDDL quanto do uso da

infraestrutura de suporte. Porém, tal limitação não invalida nosso trabalho pois acreditamos que essa análise de usabilidade não modificaria a essência do modelo de referência RAS para serviços de análise, tampouco o processo e a infraestrutura de desenvolvimento propostos.

No limite de nosso conhecimento, não há na literatura trabalhos que proponham metodologias de desenvolvimento baseado em modelos para serviços RESTful adaptadores para ferramentas de análise em bioinformática. Também não encontramos na literatura trabalhos que definam um modelo de referência para padronizar o desenvolvimento desses serviços. Assim, acreditamos que nosso processo de desenvolvimento e infraestrutura de suporte poderão fomentar o desenvolvimento de novos serviços neste domínio, permitindo que um número maior de pesquisadores e grupos de pesquisa criem e disponibilizem seus próprios serviços de análise. Também acreditamos que o modelo de referência RAS representa um esforço para a padronização de serviços de análise em bioinformática que facilitará não apenas a criação, mas também o reúso dos serviços desenvolvidos. Finalmente, embora nosso foco tenha sido a bioinformática, acreditamos que este trabalho possa ser utilizado para facilitar o desenvolvimento de serviços RESTful adaptadores em outros domínios do conhecimento com requisitos similares.

10.3 Trabalhos futuros

10.3.1 Análise de usabilidade

Como primeiro trabalho futuro, uma análise de usabilidade poderia ser realizada sobre as linguagens ActDL e SDDL, bem como sobre todo o processo de desenvolvimento proposto. Dessa maneira, seria possível obter uma visão clara sobre quão familiares os conceitos representados nos modelos utilizados são para bioinformatas provindos de diferentes áreas, como as ciências exatas e as ciências biológicas.

Essa análise poderia ser realizada em duas partes. Inicialmente, poderíamos calcular métricas de usabilidade para linguagens específicas de domínio, tais como viscosidade e visibilidade, como apresentado em Albuquerque *et al.* [136]. Essas métricas poderiam indicar uma heurística inicial para refinar as linguagens ActDL e SDDL. Em seguida, sessões acompanhadas em que bioinformatas utilizassem nossa abordagem para produzir serviços de análise para a adaptação de ferramentas conhecidas poderiam ser realizadas, colhendo as opiniões desses profissionais, bem como o tempo necessário para obter um conjunto de serviços.

A partir dos resultados dessa análise, poderíamos, se necessário, ajustar conceitos e processos da abordagem de desenvolvimento proposta, tornando-os mais

simples e compreensíveis para seus usuários. Outras alternativas também poderiam ser exploradas para facilitar a modelagem dos aspectos de interesse da atividade de análise e da ferramenta subjacente ao serviço. Por exemplo, poderíamos substituir o suporte provido pelos `StringListManipulators` para a construção da chamada utilizada para a invocação da ferramenta, utilizando em seu lugar uma linguagem de propósito geral como os *Cheetah Templates* [104], utilizados pelo ambiente integrado de análise Galaxy, ou o Javascript, utilizado pela *Common Workflow Language* (CWL) [137] para essa tarefa.

10.3.2 Uso de transformações modelo-para-texto para a obtenção de um serviço RAS com código dedicado

Como segundo trabalho futuro, consideramos o desenvolvimento de uma abordagem para a obtenção de serviços RAS com código dedicado, bem como suporte para a obtenção desse serviço para diferentes plataformas e linguagens de programação. Embora nossa avaliação dos serviço RAS produzidos com o suporte do *framework* adaptador Activity-REST não demonstrou uma diferença significativa entre o desempenho dos serviços do repositório GEAS-RAS frente ao desempenho dos serviços dedicados presentes no repositório GEAS, a existência desse suporte poderia prover melhores comparações entre o uso de interpretação de modelo *versus* a geração de código, bem como promover o uso de serviços RAS em laboratórios que apresentem restrições ao uso e à manutenção a algum ambiente de execução destes serviços.

De forma alternativa, seria possível também considerar a definição de transformações de modelos ActDL e SDDL para metamodelos "RESTful" adicionais, como, por exemplo, os apresentados nos trabalhos de Schreier [78] e Haupt [79]. Essas transformações produziriam um modelo "RESTful" do novo serviço de análise segundo as diretrizes do modelo de referência RAS. Em seguida, transformações modelo-para-texto poderiam ser utilizadas para obter o código do serviço em uma dada linguagem de programação a partir do "modelo RESTful" do serviço. Diversas linguagens de programação para a implementação de um serviço RAS poderiam ser, então, suportadas apenas pela definição de diferentes transformações para o passo final dessa abordagem de desenvolvimento. Esse estrutura de modelos e transformações seria semelhante à promovida pela Arquitetura Orientada a Modelos [70]. Modelos ActDL e SDDL representariam modelos independentes de computação, enquanto o modelo "RESTful" do serviço representaria um modelo específico de plataforma.

10.3.3 Extensões à arquitetura e aos metamodelos definidos

Como um terceiro trabalho futuro, a arquitetura de metamodelagem definida pode ser estendida por meio da adição de diferentes sintaxes para os modelos AADM e SDDM. Por exemplo, seria possível prover uma sintaxe gráfica para modelos AADM, de modo a permitir representar os conceitos relacionados desses modelos de uma maneira facilmente identificável em ambientes integrados de análise. Também poderia ser possível utilizar tanto a representação textual (ActDL) quanto a representa gráfica de maneira simultânea, permitindo representar esses modelos conforme a necessidade e o interesse do desenvolvedor.

O uso de anotações semânticas em um modelo ActDL também poderia ser investigado futuramente. Essa anotação poderia ser realizada por meio da extensão da linguagem ActDL de modo a permitir associar termos de ontologias biomédicas aos elementos modelados, como a própria atividade análise descrita, bem como aos parâmetros de execução e conjuntos de dados utilizados ou produzidos pela sua execução. Essa informação semântica poderia, então, ser utilizada para obter automaticamente descrições de serviço semanticamente anotadas, bem como ser retornada nos controles de hipermídia de um serviço RAS, de modo a facilitar a descoberta de serviços de análise de interesse por meio do uso de serviços de diretório.

O modelo de referência RAS também poderia ser estendido de modo a incluir, junto aos controles de hipermídia retornados, informações sobre os tipos de dados que devem ser utilizados ou esperados para cada *endpoint* do serviço de análise. Isso permitiria a construção automática de interfaces gráficas de usuário para a execução de atividades de análise por meio de um serviço RAS apenas dependentes da interação com o serviço de análise em si. Tais interfaces podem, atualmente, ser obtidas diretamente da descrição do serviço de análise gerada pela infraestrutura de suporte provida. Adicionalmente, as ligações de hipermídia retornadas também poderiam incluir as anotações semânticas incluídas no modelo ActDL, provendo essa informação sobre os dados trafegados ao usuário do serviço. Isso possibilitaria que usuários utilizassem mecanismos de descoberta semântica de serviços de análise, facilitando a composição destes serviços durante o processo de análise, como, por exemplo, em [8].

Finalmente, podemos investigar o desenvolvimento de suporte à adaptação não apenas de ferramentas de linha de comando, mas também de outros artefatos como, por exemplo, funções e módulos escritos em diferentes linguagens de programação. Por exemplo, um desenvolvedor poderia desenvolver um serviço RAS no qual um módulo em uma dada linguagem de programação, como Python e R, seria carregado pelo *framework* Activity-REST e uma das funções de análise desse módulo fosse invocada diretamente. O provimento desse suporte não é necessário para a criação

de serviços adaptadores, pois tais funções e módulos podem ser facilmente providas como ferramentas de linha de comando por meio do suporte nativo presente em qualquer linguagem de programação de propósito geral. Porém, a existência de interoperabilidade mais estreita com diferentes linguagens de programação poderia facilitar o desenvolvimento de novos serviços RAS, bem como prover um melhor desempenho para a execução desses serviços.

10.3.4 Suporte ao uso de serviços RAS a partir de diferentes ambientes de análise

Entre os ambientes integrados de análise existentes, provemos suporte à obtenção de clientes para o ambiente Galaxy apenas. Porém, o processo de desenvolvimento orientado a modelos proposto é igualmente aplicável para a obtenção de clientes de serviço para outros ambientes. Dessa maneira, como último trabalho futuro, podemos investigar o desenvolvimento de suporte à obtenção de clientes para novos ambientes integrados de análise, tais como Taverna [11, 15] e SemanticSCo [8].

Referências

[1] BARE, J. C.; BALIGA, N. S. Architecture for interoperable software in biology. *Briefings in Bioinformatics*, Oxford University Press, v. 15, n. 4, p. 626, 2014.

[2] ELIXIR. *bio.tools · Bioinformatics Tools and Services Discovery Portal*. 2018. Disponível em: https://bio.tools/.

[3] ISON, J. et al. Tools and data services registry: a community effort to document bioinformatics resources. *Nucleic Acids Research*, Oxford University Press, v. 44, n. D1, p. D38–D47, jan 2016. ISSN 0305-1048. Disponível em: http://nar.oxfordjournals.org/lookup/doi/10.1093/nar/gkv1116.

[4] NEERINCX, P. B. T.; LEUNISSEN, J. A. M. Evolution of web services in bioinformatics. *Briefings in Bioinformatics*, Oxford University Press, v. 6, n. 2, p. 178–188, jan 2005. ISSN 1467-5463. Disponível em: http://bib.oxfordjournals.org/cgi/doi/10.1093/bib/6.2.178.

[5] ŐSZ, Á. et al. A snapshot of 3649 Web-based services published between 1994 and 2017 shows a decrease in availability after 2 years. *Briefings in Bioinformatics*, Narnia, v. 20, n. 3, p. 1004–1010, may 2019. ISSN 1477-4054. Disponível em: https://academic.oup.com/bib/article/20/3/1004/4710326.

[6] NUCLEIC ACIDS RESEARCH. Editorial: Nucleic Acids Research annual Web Server Issue in 2016. *Nucleic Acids Research*, Oxford University Press, v. 43, n. W1, p. W1—-W2, jul 2015. Disponível em: http://nar.oxfordjournals.org/lookup/doi/10.1093/nar/gkv581http://nar.oxfordjournals.org/lookup/doi/10.1093/nar/gkw460.

[7] GUARDIA, G. D. A. et al. A Methodology for the Development of RESTful Semantic Web Services for Gene Expression Analysis. *PLOS ONE*, Public Library of Science, v. 10, n. 7, p. e0134011, jul 2015. ISSN 1932-6203.

[8] GUARDIA, G. D. et al. SemanticSCo: a Platform to Support the Semantic Composition of Services for Gene Expression Analysis. *Journal of Biomedical Informatics*, v. 66, p. 116–128, 2017. ISSN 15320464. Disponível em: http://linkinghub.elsevier.com/retrieve/pii/S1532046416301885.

[9] European Molecular Biology Laboratory. *The European Bioinformatics Institute*. 2021. Disponível em: https://www.ebi.ac.uk/.

[10] U.S. National Library of Medicine. *National Center for Biotechnology Information*. 2021. Disponível em: https://www.ncbi.nlm.nih.gov/.

[11] OINN, T. et al. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics (Oxford, England)*, v. 20, n. 17, p. 3045–54, nov 2004. ISSN 1367-4803. Disponível em: http://www.ncbi.nlm.nih.gov/pubmed/15201187.

[12] SHANNON, P. T. et al. The Gaggle: an open-source software system for integrating bioinformatics software and data sources. *BMC Bioinformatics*, v. 7, 2006. Disponível em: http://dx.doi.org/10.1186/1471-2105-7-176.

[13] BARTOCCI, E. et al. BioWMS: a web-based Workflow Management System for bioinformatics. *BMC Bioinformatics*, BioMed Central, v. 8, n. Suppl 1, p. S2, 2007. ISSN 14712105. Disponível em: http://bmcbioinformatics.biomedcentral.com/articles/10.1186/1471-2105-8-S1-S2.

[14] GOECKS, J. et al. Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome biology*, v. 11, n. 8, p. R86, 2010. ISSN 1474-760X. Disponível em: http://www.ncbi.nlm.nih.gov/pubmed/20738864http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC2945788.

[15] WOLSTENCROFT, K. et al. The Taverna workflow suite: designing and executing workflows of Web Services on the desktop, web or in the cloud. *Nucleic Acids Research*, Oxford University Press, v. 41, n. W1, p. W557—-W561, jul 2013. ISSN 0305-1048. Disponível em: http://nar.oxfordjournals.org/lookup/doi/10.1093/nar/gkt328.

[16] ARANGUREN, M.; GONZÁLEZ, A.; WILKINSON, M. D. Executing SADI services in Galaxy. *Journal of Biomedical Semantics*, BioMed Central, v. 5, n. 1, p. 42, 2014. ISSN 2041-1480. Disponível em: http://jbiomedsem.biomedcentral.com/articles/10.1186/2041-1480-5-42.

[17] PIREDDU, L. et al. A Hadoop-Galaxy adapter for user-friendly and scalable data-intensive bioinformatics in Galaxy. In: *Proceedings of the 5th ACM Conference on Bioinformatics, Computational Biology, and Health Informatics - BCB '14*. New York, New York, USA: ACM Press, 2014. p. 184–191. ISBN 9781450328944. Disponível em: http://dl.acm.org/citation.cfm?doid=2649387.2649429.

[18] MELLOR, S. J.; CLARK, A. N.; FUTAGAMI, T. Model Driven Development. *IEEE Software*, v. 20, n. 5, p. 14–18, 2003.

[19] SELIC, B. The pragmatics of model-driven development. *IEEE Software*, v. 20, n. 5, p. 19–25, 2003. ISSN 0740-7459.

[20] SENDALL, S.; KOZACZYNSKI, W. Model transformation: the heart and soul of model-driven software development. *IEEE Software*, v. 20, n. 5, p. 42–45, 2003. ISSN 0740-7459. Disponível em: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1231150.

[21] BENDER, A.; BOZIC, S.; KONDOV, I. An EMF-based toolkit for creation of domain-specific data services. In: *Model-Driven Engineering and Software Development (MODELSWARD), 2014 2nd International Conference on*. [S.l.: s.n.], 2014. p. 30–40.

[22] ZOLOTAS, C.; SYMEONIDIS, A. L. Towards an MDA Mechanism for RESTful Services Development. In: *Proceedings of the 3rd International Workshop on Model-Driven Engineering on and for the Cloud 18th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2015)*. [S.l.: s.n.], 2015. p. 31–36.

[23] LI, L.; CHOU, W. Designing Large Scale REST APIs Based on REST Chart. In: *2015 IEEE International Conference on Web Services*. IEEE, 2015. p. 631–638. ISBN 978-1-4673-7272-5. Disponível em: http://ieeexplore.ieee.org/document/7195624/.

[24] ED-DOUIBI, H. et al. EMF-REST: generation of RESTful APIs from models. In: *Proceedings of the 31st Annual ACM Symposium on Applied Computing (SAC '16)*. New York, New York, USA: ACM Press, 2016. p. 1446–1453. ISBN 9781450337397. Disponível em: http://dl.acm.org/citation.cfm?doid=2851613.2851782.

[25] HERMIDA, J. M. et al. Applying model-driven engineering to the development of Rich Internet Applications for Business Intelligence. *Information Systems Frontiers*, Springer, v. 15, n. 3, p. 411–431, jul 2013. ISSN 13873326. Disponível em: http://www.dlsi.ua.es.

[26] MARTÍNEZ-GARCÍA, A. et al. Working with the HL7 metamodel in a Model Driven Engineering context. *Journal of Biomedical Informatics*, v. 57, p. 415–424, sep 2015. ISSN 1532-0480. Disponível em: http://www.sciencedirect.com/science/article/pii/S1532046415001938.

[27] DEMSKI, H.; GARDE, S.; HILDEBRAND, C. Open data models for smart health interconnected applications: The example of openEHR Standards, technology, machine learning, and modeling. *BMC Medical Informatics and Decision Making*, BioMed Central Ltd, v. 16, n. 1, p. 1–9, oct 2016. ISSN 14726947. Disponível em: https://link.springer.com/articles/10.1186/s12911-016-0376-2https://link.springer.com/article/10.1186/s12911-016-0376-2.

[28] NTANOS, E. et al. A model-driven software engineering workflow and tool architecture for servitised manufacturing. *Information Systems and e-Business Management*, Springer Verlag, v. 16, n. 3, p. 683–720, aug 2018. ISSN 16179854. Disponível em: https://doi.org/10.1007/s10257-018-0371-5.

[29] OBJECT MANAGEMENT GROUP. *Business Process Model and Notation (BPMN) Version 2.0*. 2011.

[30] HOLLINGSWORTH, D. *Workflow management coalition: The workflow reference model*. [S.l.], 1993. v. 59, n. 10, 904–13 p. Disponível em: http://www.ncbi.nlm.nih.gov/pubmed/21787529{\\%}5Cnhttp://www.avicos.ru/images/photo/1/7.

[31] GARZA, L. de la et al. From the desktop to the grid: scalable bioinformatics via workflow conversion. *BMC Bioinformatics*, BioMed Central, v. 17, n. 1, p. 127, 2016. ISSN 1471-2105. Disponível em: http://www.biomedcentral.com/1471-2105/17/127.

[32] AMELLER, D. et al. Development of service-oriented architectures using model-driven development: A mapping study. *Information and Software Technology*, v. 62, n. 1, 2015. ISSN 09505849.

[33] TANENBAUM, A. S.; STEEN, M. V. *Distributed Systems: principles and paradigms*. 2nd. editi. ed. Upper Saddle River: Pearson Education Inc., 2007. 686 p. ISBN 0-13-239227-5.

[34] AFGAN, E. et al. The Galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2016 update. *Nucleic Acids Research*, Oxford University Press, v. 44, n. W1, p. W3—-W10, jul 2016. ISSN 0305-1048. Disponível em: http://nar.oxfordjournals.org/lookup/doi/10.1093/nar/gkw343.

[35] BØRNICH, C. et al. Galaxy Portal: interacting with the galaxy platform through mobile devices. *Bioinformatics*, Oxford University Press, v. 32, n. 11, p. 1743–1745, jun 2016. ISSN 1367-4803. Disponível em: http://bioinformatics.oxfordjournals.org/lookup/doi/10.1093/bioinformatics/btw042.

[36] MACKENZIE, C. M. et al. Reference Model for Service Oriented Architecture 1.0. 2006. Disponível em: http://docs.oasis-open.org/soa-rm/v1.0/.

[37] WORLD WIDE WEB CONSORTIUM. *Web Services Glossary*. 2004. Disponível em: http://www.w3.org/TR/ws-gloss/.

[38] World Wide Web Consortium. *Web Services Architecture*. 2004. Disponível em: https://www.w3.org/TR/2004/NOTE-ws-arch-20040211/.

[39] WORLD WIDE WEB CONSORTIUM. *Hypertext Transfer Protocol - HTTP/1.1*. [S.l.], 1999. Disponível em: https://tools.ietf.org/pdf/rfc2616.pdf.

[40] WORLD WIDE WEB CONSORTIUM. *Extensible Markup Language (XML)*. Disponível em: http://www.w3.org/XML/.

[41] WORLD WIDE WEB CONSORTIUM. SOAP Version 1.2 Part 0: Primer (Second Edition). 2007. Disponível em: https://www.w3.org/TR/2007/REC-soap12-part0-20070427/.

[42] WORLD WIDE WEB CONSORTIUM. SOAP Version 1.2 Part 1: Messaging Framework (Second Edition). 2007. Disponível em: https://www.w3.org/TR/2007/REC-soap12-part1-20070427/.

[43] WORLD WIDE WEB CONSORTIUM. SOAP Version 1.2 Part 2: Adjuncts (Second Edition). 2007. Disponível em: https://www.w3.org/TR/2007/REC-soap12-part2-20070427/.

[44] WORLD WIDE WEB CONSORTIUM. SOAP Version 1.2 Specification Assertions and Test Collection (Second Edition). 2007. Disponível em: https://www.w3.org/TR/2007/REC-soap12-testcollection-20070427/.

[45] WORLD WIDE WEB CONSORTIUM. *RFC2821 - Simple Mail Transfer Protocol*. [S.l.], 2001. Disponível em: https://www.ietf.org/rfc/rfc2821.txt.

[46] DEFENSE ADVANCED RESEARCH PROJECTS AGENCY. *RFC793 - Transmission Control Protocol*. [S.l.], 1981. Disponível em: https://www.ietf.org/rfc/rfc793.txt.

[47] WORLD WIDE WEB CONSORTIUM. *Web Services Description Language (WSDL) Version 2.0 Part 0: Primer*. 2007. Disponível em: https://www.w3.org/TR/wsdl20-primer/.

[48] WORLD WIDE WEB CONSORTIUM. *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language*. 2007. Disponível em: https://www.w3.org/TR/wsdl20/.

[49] WORLD WIDE WEB CONSORTIUM. *Web Services Description Language (WSDL) Version 2.0 Part 2: Adjuncts*. 2007. Disponível em: https://www.w3.org/TR/wsdl20-adjuncts/.

[50] OASIS. *UDDI Version 3.0.2*. [S.l.], 2004. Disponível em: https://www.oasis-open.org/committees/uddi-spec/doc/spec/v3/uddi-v3.0.2-20041019.htm.

[51] OASIS. *OASIS - Advancing open standards for the information society*. 2017. Disponível em: https://www.oasis-open.org/.

[52] FIELDING, R. T. *Architectural Styles and the Design of Network-based Software Architectures*. Tese (Doutorado) — University of California, Irvine, 2000. Disponível em: https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf.

[53] FIELDING, R. T.; TAYLOR, R. N. Principled design of the modern Web architecture. *ACM Transactions on Internet Technology*, ACM, v. 2, n. 2, p. 115–150, 2002. ISSN 15335399. Disponível em: http://portal.acm.org/citation.cfm?doid=514183.514185.

[54] ECMA INTERNATIONAL. ECMA-404: The JSON Data Interchange Format. 2013.

[55] BEN-KIKI, O.; EVANS, C.; NET, I. döt. *YAML Ain't Markup Language (YAML™) Version 1.2*. 2017. Disponível em: http://yaml.org/spec/1.2/spec.html.

[56] HAUPT, F.; LEYMANN, F.; PAUTASSO, C. A Conversation Based Approach for Modeling REST APIs. In: *Proceedings - 12th Working IEEE/IFIP Conference on Software Architecture, WICSA 2015*. [S.l.]: IEEE, 2015. p. 165–174.

[57] World Wide Web Consortium. *W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes*. 2012. Disponível em: https://www.w3.org/TR/xmlschema11-1/https://www.w3.org/TR/xmlschema11-2/.

[58] World Wide Web Consortium. *W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures*. 2012. Disponível em: https://www.w3.org/TR/xmlschema11-1/.

[59] THE LINUX FOUNDATION. *Open API Initiative*. 2017. Disponível em: https://www.openapis.org/.

[60] SMARTBEAR SOFTWARE. *Swagger – The World's Most Popular Framework for APIs*. 2017. Disponível em: http://swagger.io/.

[61] SUN MICROSYSTEMS. Web Application Description Language. 2009. Disponível em: https://www.w3.org/Submission/wadl/.

[62] WRIGHT, A. et al. *JSON Schema: A Media Type for Describing JSON Documents*. [S.l.], 2020. Work in Progress. Disponível em: https://datatracker.ietf.org/doc/html/draft-bhutton-json-schema-00.

[63] FARIAS, C. R. G. de. *Architectural Design of Groupware Systems: a Component-Based Approach*. Tese (Doutorado) — University of Twente, The Netherlands, 2002. Disponível em: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.81.2109{\&}rep=rep1{\&}type=pdfhttp://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.81.2109{\\%}7B{\&}{\\%}7Drep=rep1.

[64] ATKINSON, C.; KUHNE, T. Model-driven development: a metamodeling foundation. *IEEE Software*, v. 20, n. 5, p. 36–41, sep 2003. ISSN 0740-7459.

[65] OBJECT MANAGEMENT GROUP. OMG Unified Modeling Language (OMG UML), version 2.5. 2015. Disponível em: http://www.omg.org/spec/UML/2.5/.

[66] GROUP, O. M. *Meta Object Facility (MOF) Core Specification*. [S.l.], 2006.

[67] OBJECT MANAGEMENT GROUP. *OMG Unified Modeling Language (TM) Infrastructure Version 2.4.1*. 2011.

[68] BARIŠIĆ, A.; AMARAL, V.; GOULÃO, M. Usability driven DSL development with USE-ME. *Computer Languages, Systems & Structures*, Pergamon, v. 51, p. 118–157, jan 2018. ISSN 1477-8424. Disponível em: https://www.sciencedirect.com/science/article/pii/S1477842417300477.

[69] OBJECT MANAGEMENT GROUP. *MDA Guide Version 1.0.1*. 2003. Disponível em: http://www.omg.org/cgi-bin/doc?omg/03-06-01.pdf.

[70] OBJECT MANAGEMENT GROUP. Model Driven Architecture. 2000.

[71] MELLOR, S. J. et al. Model-Driven Architecture. In: BRUEL, J.-M.; BELLAHSENE, Z. (Ed.). *Advances in Object-Oriented Information Systems: OOIS 2002 Workshops Montpellier*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002. cap. Model-Driv, p. 290–297. ISBN 978-3-540-46105-0. Disponível em: http://dx.doi.org/10.1007/3-540-46105-1_33.

[72] OBJECT MANAGEMENT GROUP. *MDA Guide rev.2.0*. 2014. Disponível em: http://www.omg.org/cgi-bin/doc?ormsc/14-06-01.

[73] ECLIPSE FOUNDATION. *Eclipse Modeling Project*. Disponível em: http://www.eclipse.org/modeling/.

[74] ECLIPSE FOUNDATION. *Eclipse IDE*. Http://www.eclipse.org/. Disponível em: http://www.eclipse.org/.

[75] ECLIPSE FOUNDATION. *Eclipse Modeling - EMF - Home*. Disponível em: http://www.eclipse.org/modeling/emf/.

[76] Object Management Group et al. *Meta Object Facility (MOF) Core Specification*. [S.l.], 2006.

[77] PAHL, C. Semantic model-driven architecting of service-based software systems. *Information and Software Technology*, v. 49, n. 8, p. 838–850, 2007. ISSN 09505849.

[78] SCHREIER, S. Modeling RESTful applications. In: *Proceedings of the Second International Workshop on RESTful Design (WSREST)*. New York, New York, USA: ACM Press, 2011. p. 15–21. ISBN 9781450306232. Disponível em: http://www.ws-rest.org/2011/proc/a4-schreier.pdf.

[79] HAUPT, F. et al. A Model-Driven Approach for REST Compliant Services. In: *2014 IEEE International Conference on Web Services*. IEEE, 2014. p. 129–136. ISBN 978-1-4799-5054-6. Disponível em: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6928890.

[80] MORDANI, R. JSR-000154: Java Servlet Specification Version 2.5 MR6. 2007. Disponível em: https://jcp.org/en/jsr/detail?id=154.

[81] SFERRUZZA, D. Top-down model-driven engineering of web services from extended openapi models. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. New York, NY, USA: ACM, 2018. p. 940–943.

[82] MOHSENI, M.; SOHRABI, M. K.; DORRIGIV, M. A model-driven approach for semantic web service modeling using web service modeling languages. *Journal of Software: Evolution and Process*, John Wiley & Sons, Ltd, v. 33, n. 7, p. e2364, jul 2021. ISSN 2047-7473. Disponível em: https://onlinelibrary.wiley.com/doi/10.1002/smr.2364.

[83] OBJECT MANAGEMENT GROUP. *OMG Object Constraint Language (OCL) Version 2.3.1*. 2012.

[84] BRUIJN, J. de et al. *Web Service Modeling Language (WSML) - W3C Submission*. 2005. Disponível em: https://www.w3.org/Submission/WSML.

[85] FENSEL, D. et al. Web Service Modeling Ontology. In: *Semantic Web Services*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. v. 1, p. 107–129.

[86] FEIER, C. et al. Towards intelligent web services: the web service modeling ontology (WSMO). In: *2005 International Conference on Intelligent Computing (ICIC'05)*. [S.l.: s.n.], 2005.

[87] DOBIN, A. et al. STAR: ultrafast universal RNA-seq aligner. *Bioinformatics (Oxford, England)*, Oxford University Press, v. 29, n. 1, p. 15–21, jan 2013. ISSN 1367-4811. Disponível em: http://www.ncbi.nlm.nih.gov/pubmed/23104886http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC3530905.

[88] LANGMEAD, B.; SALZBERG, S. L. Fast gapped-read alignment with Bowtie 2. *Nature methods*, v. 9, n. 4, p. 357–9, 2012. ISSN 1548-7105. Disponível em: http://dx.doi.org/10.1038/nmeth.1923.

[89] AGARWALA, R. et al. Database resources of the National Center for Biotechnology Information. *Nucleic Acids Research*, Oxford University Press, v. 46, n. D1, p. D8–D13, jan 2018. ISSN 13624962. Disponível em: http://www.ncbi.nlm.nih.gov/pubmed/29140470http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC5753372.

[90] CHOJNACKI, S. et al. Programmatic access to bioinformatics tools from EMBL-EBI update: 2017. *Nucleic Acids Research*, Oxford University Press, v. 45, n. W1, p. W550–W553, jul 2017. ISSN 0305-1048. Disponível em: https://academic.oup.com/nar/article-lookup/doi/10.1093/nar/gkx273.

[91] PAUTASSO, C. RESTful Web Services: Principles, Patterns, Emerging Technologies. In: *Web Services Foundations*. New York, NY: Springer New York, 2014. p. 31–51. Disponível em: http://link.springer.com/10.1007/978-1-4614-7518-7_2.

[92] World Wide Web Consortium. *Server-Sent Events*. [S.l.], 2015. Disponível em: https://www.w3.org/TR/eventsource/.

[93] KELLY, M. *JSON Hypertext Application Language (draft-kelly-json-hal-08)*. [S.l.], 2016. Disponível em: https://tools.ietf.org/html/draft-kelly-json-hal-08.

[94] ECLIPSE FOUNDATION. *Eclipse Modeling - MDT - Home*. Disponível em: http://www.eclipse.org/modeling/mdt/.

[95] SIEVERS, F. et al. Fast, scalable generation of high-quality protein multiple sequence alignments using Clustal Omega. *Molecular systems biology*, v. 7, n. 1, p. 539, oct 2011. ISSN 1744-4292. Disponível em: http://msb.embopress.org/cgi/doi/10.1038/msb.2011.75http://www.ncbi.nlm.nih.gov/pubmed/21988835http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC3261699.

[96] International Organization for Standardization. *ISO/IEC 9899:2018 - Information technology – Programming languages – C*. [S.l.], 2018. Disponível em: https://www.iso.org/standard/74528.html.

[97] THE APACHE SOFTWARE FOUNDATION. *Maven*. 2018. Disponível em: https://maven.apache.org/.

[98] ORACLE CO. *GlassFish*. 2018. Disponível em: https://javaee.github.io/glassfish/.

[99] THE APACHE SOFTWARE FOUNDATION. *Apache Tomcat*. 2018. Disponível em: http://tomcat.apache.org/.

[100]Object Management Group. *MOF Model to Text Transformation Language , v1.0*. [S.l.], 2008. Disponível em: https://www.omg.org/spec/MOFM2T/1.0/PDF.

[101]Kotlin Foundation. *Kotlin Programming Language*. 2021. Disponível em: https://kotlinlang.org/.

[102]DEVELOPERS, P. *picocli - a mighty tiny command line interface*. 2021. Disponível em: https://picocli.info/.

[103]SUN MICROSYSTEMS. *JSR-000311 JAX-RS: The JavaTM API for RESTful Web Services 1.0 Final Release*. Santa Clara, California, 2008.

[104]The Cheetah Development Team. *Cheetah3, the Python-Powered Template Engine*. 2020. Disponível em: https://cheetahtemplate.org/index.html.

[105]OBEO. *Acceleo*. 2020. Disponível em: https://www.eclipse.org/acceleo/.

[106]Eclipse Foundation. *Eclipse Xpand*. 2020. Disponível em: https://projects.eclipse.org/projects/modeling.m2t.xpand.

[107]Eclipse Foundation. *Xtend - Modernized Java*. 2020. Disponível em: https://www.eclipse.org/xtend/.

[108]APACHE FOUNDATION. *Axis 2*. 2010. Disponível em: http://axis.apache.org/.

[109]ED-DOUIBI, H. *The OpenAPI metamodel*. 2019. Disponível em: https://github.com/opendata-for-all/openapi-metamodel/.

[110]JOUAULT, F. et al. ATL: A model transformation tool. *Science of Computer Programming*, v. 72, n. 1-2, p. 31–39, jun 2008. ISSN 01676423.

[111]URBANEK, S. *Package Rserve*. Comprehensive R Archive Network (CRAN), 2019. Disponível em: https://cran.r-project.org/package=Rserve.

[112]DAVIS, T. L. *Command Line Optional and Positional Argument Parser [R package argparse version 2.0.1]*. Comprehensive R Archive Network (CRAN), 2019. Disponível em: https://cran.r-project.org/package=argparse.

[113]The JUnit Team. *JUnit 5*. Disponível em: https://junit.org/junit5/.

[114]HALEBY, J. *REST-assured*. 2019. Disponível em: http://rest-assured.io/.

[115]WINTER, A. G.; WILDENHAIN, J.; TYERS, M. BioGRID REST Service, BiogridPlugin2 and BioGRID WebGraph: new tools for access to interaction data at BioGRID. *Bioinformatics*, Oxford University Press, v. 27, n. 7, p. 1043–1044, apr 2011. ISSN 1460-2059. Disponível em: https://academic.oup.com/bioinformatics/article-lookup/doi/10.1093/bioinformatics/btr062.

[116]BLEDA, M. et al. CellBase, a comprehensive collection of RESTful web services for retrieving relevant biological information from heterogeneous sources. *Nucleic Acids Research*, Oxford University Press, v. 40, n. W1, p. W609–W614, jul 2012. ISSN 0305-1048. Disponível em: https://academic.oup.com/nar/article-lookup/doi/10.1093/nar/gks575.

[117]REISINGER, F. et al. Introducing the PRIDE Archive RESTful web services. *Nucleic acids research*, Oxford University Press, v. 43, n. W1, p. W599–604, jul 2015. ISSN 1362-4962. Disponível em: http://www.ncbi.nlm.nih.gov/pubmed/25904633http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC4489246.

[118]YATES, A. et al. The Ensembl REST API: Ensembl Data for Any Language. *Bioinformatics*, Oxford University Press, v. 31, n. 1, p. 143–145, jan 2015. ISSN 1460-2059. Disponível em: https://academic.oup.com/bioinformatics/article-lookup/doi/10.1093/bioinformatics/btu613.

[119]FENYÖ, D.; BEAVIS, R. C. The GPMDB REST interface. *Bioinformatics*, Oxford University Press, v. 31, n. 12, p. 2056–2058, jun 2015. ISSN 1367-4803. Disponível em: https://academic.oup.com/bioinformatics/article-lookup/doi/10.1093/bioinformatics/btv107.

[120]OTEGUI, J.; GURALNICK, R. P. The geospatial data quality REST API for primary biodiversity data. *Bioinformatics*, Oxford University Press, v. 32, n. 11, p. 1755–1757, jun 2016. ISSN 1367-4803. Disponível em: https://academic.oup.com/bioinformatics/article-lookup/doi/10.1093/bioinformatics/btw057.

[121]KHAN, A.; MATHELIER, A. JASPAR RESTful API: accessing JASPAR data from any programming language. *Bioinformatics*, p. 1–3, dec 2017. ISSN 1367-4803. Disponível em: http://academic.oup.com/bioinformatics/advance-article/doi/10.1093/bioinformatics/btx804/4747882.

[122]SZKLARCZYK, D. et al. The STRING database in 2017: quality-controlled protein–protein association networks, made broadly accessible. *Nucleic Acids Research*, v. 45, n. D1, p. D362–D368, jan 2017. Disponível em: http://www.ncbi.nlm.nih.gov/pubmed/27924014.

[123]JIAO, X. et al. DAVID-WS: a stateful web service to facilitate gene/protein list analysis. *Bioinformatics*, Oxford University Press, v. 28, n. 13, p. 1805–1806, jul 2012. ISSN 1367-4803. Disponível em: https://academic.oup.com/bioinformatics/article-lookup/doi/10.1093/bioinformatics/bts251.

[124]FINN, R. D. et al. HMMER web server: 2015 update. *Nucleic Acids Research*, Oxford University Press, v. 43, n. W1, p. W30–W38, jul 2015. ISSN 0305-1048. Disponível em: https://academic.oup.com/nar/article-lookup/doi/10.1093/nar/gkv397.

[125]PILLAI, S. et al. SOAP-based services provided by the European Bioinformatics Institute. *Nucleic acids research*, Oxford University Press, v. 33, n. Web Server issue, p. W25–8, jul 2005. ISSN 1362-4962. Disponível em: http://www.ncbi.nlm.nih.gov/pubmed/15980463http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC1160251.

[126]GOUJON, M. et al. A new bioinformatics analysis tools framework at EMBL-EBI. *Nucleic Acids Research*, Oxford University Press, v. 38, n. Web Server, p. W695–W699, jul 2010. ISSN 0305-1048. Disponível em: https://academic.oup.com/nar/article-lookup/doi/10.1093/nar/gkq313.

[127]RICE, P. M. et al. EMBRACE: Bioinformatics Data and Analysis Tool Services for e-Science. In: *2006 Second IEEE International Conference on e-Science and Grid Computing (e-Science'06)*. IEEE, 2006. p. 146–146. ISBN 0-7695-2734-5. Disponível em: http://ieeexplore.ieee.org/document/4031119/.

[128]PETTIFER, S. et al. The EMBRACE web service collection. *Nucleic Acids Research*, Oxford University Press, v. 38, n. Web Server issue, p. W683–8, jul 2010. ISSN 1362-4962. Disponível em: http://www.ncbi.nlm.nih.gov/pubmed/20462862http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC2896104.

[129]ALTSCHUL, S. F. et al. Basic local alignment search tool. *Journal of Molecular Biology*, Academic Press, v. 215, n. 3, p. 403–410, oct 1990. ISSN 0022-2836. Disponível em: https://www.sciencedirect.com/science/article/pii/S0022283605803602?via{\\%}3Di.

[130]KIM, S. et al. An update on PUG-REST: RESTful interface for programmatic access to PubChem. *Nucleic acids research*, Oxford University Press, v. 46, n. W1, p. W563–W570, jul 2018. ISSN 1362-4962. Disponível em: http://www.ncbi.nlm.nih.gov/pubmed/29718389http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC6030920.

[131]Oracle Inc. *GraalVM Community*. 2020. Disponível em: https://www.graalvm.org/.

[132]The Kubernetes Authors; The Linux Foundation. *Kubernetes*. 2020. Disponível em: https://kubernetes.io/.

[133]Google LLC. *Cloud Computing Services | Google Cloud*. 2020. Disponível em: https://cloud.google.com/.

[134]Docker Inc. *Docker Hub*. 2020. Disponível em: https://hub.docker.com/.

[135]World Wide Web Consortium. *OWL 2 Web Ontology Language Document Overview (Second Edition)*. 2012. Disponível em: http://www.w3.org/TR/owl2-overview/.

[136]ALBUQUERQUE, D. et al. Quantifying usability of domain-specific languages: An empirical study on software maintenance. *Journal of Systems and Software*, Elsevier, v. 101, p. 245–259, mar 2015. ISSN 0164-1212. Disponível em: https://www.sciencedirect.com/science/article/pii/S0164121214002799?via{\\%}3Di.

[137]CRUSOE, M. R. et al. Methods included: Standardizing computational reuse and portability with the common workflow language. *arXiv preprint arXiv:2105.07028*, 2021.

A Estados de uma atividade de análise

O ciclo de vida de uma atividade de análise define um conjunto de estados possíveis para as instâncias de atividade de análise, bem como os eventos que levam à transição entre esses estados. Em cada estado, um conjunto de operações pode ser executado sobre a instância de atividade de análise. A Tabela 17 apresenta uma visão geral desses estados.

Tabela 17 – Estados de uma instância de atividade de análise.

Estado	**Descrição**
START	A instância da atividade de análise ainda não existe.
CREATED	A instância da atividade de análise foi criada. Porém, uma vez que há parâmetros ou dados de entrada ainda não definidos, essa instância ainda não está pronta para ser executada.
READY	Uma instância criada está pronta para a execução. Todos os parâmetros e dados de entrada necessários foram providos.
RUNNING	A instância está sendo executada. Resultados parciais ou intermediários podem existir, embora não sejam recuperáveis.
SUCCEEDED	A instância foi executada com sucesso. Resultados da execução dessa instância já podem ser recuperados.
FAILED	A instância foi executada. Porém, essa execução resultou em falha. Um relatório de erro pode estar disponível.

Continua.

Tabela 17 – Estados de uma instância de atividade de análise (Continuação).

Estado	**Descrição**
FINISHED	Uma instância foi removida pelo usuário ou por um sistema de liberação de recursos. O acesso aos conjuntos de dados e parâmetros dessa instância não está mais disponível ao usuário.

Fonte: Autoria própria.

O estado de uma instância de atividade de análise muda em decorrência de eventos disparados por ações do usuário de um serviço de análise, bem como por causa de eventos internos a esse serviço. A Tabela 18 apresenta as transições definidas para o ciclo de vida de uma instância de atividade de análise.

Tabela 18 – Transições de estados de uma instância de atividade de análise.

Est. inicial	**Est. final**	**Descrição**
`START`	`CREATED`	Transição disparada após a criação de uma instância da atividade de análise.

Continua.

Tabela 18 – Transições de estados de uma instância de atividade de análise (Continuação).

Est. inicial	**Est. final**	**Descrição**
`CREATED`	`CREATED`	Transição disparada em resposta aos seguintes eventos: 1. O usuário informa parâmetros de execução ou conjuntos de dados de entradas. Porém, o conjunto total de parâmetros e dados de entrada necessários para a execução da atividade de análise ainda está incompleto; 2. O usuário recupera conjuntos de dados de entrada ou parâmetros de execução.
`CREATED`	`READY`	Transição obrigatória, disparada quando os parâmetros de execução e dados de entrada mínimos necessários são informados.
`READY`	`READY`	Transição disparada em resposta aos seguintes eventos: 1. O usuário provê parâmetros de execução ou conjuntos de dados de entradas; 2. O usuário recupera conjuntos de dados de entrada ou parâmetros de execução.
`READY`	`RUNNING`	Transição disparada quando o usuário inicia execução da instância de atividade de análise.

Continua.

Tabela 18 – Transições de estados de uma instância de atividade de análise (Continuação).

Est. inicial	**Est. final**	**Descrição**
RUNNING	READY	Transição disparada quando o usuário cancela a execução da instância de atividade de análise.
RUNNING	SUCCEEDED	Transição disparada pela conclusão com sucesso da execução da instância de atividade de análise. Resultados estão disponíveis ao usuário.
SUCCEEDED	SUCEEDED	Transição disparada quando o usuário recupera conjuntos de dados de entrada, conjuntos de dados de saída e parâmetros de execução de uma atividade de análise terminada com sucesso.
RUNNING	FAILED	Transição disparada quando a execução da instância de atividade de análise é interrompida em virtude de uma falha.
FAILED	FAILED	Transição disparada quando o usuário recupera o relatório contendo a descrição provável da falha ocorrida durante a execução de uma atividade de análise, conjuntos de dados de entrada e parâmetros de execução.

Continua.

Tabela 18 – Transições de estados de uma instância de atividade de análise (Continuação).

Est. inicial	**Est. final**	**Descrição**
Qualquer estado (exceto `START`)	`FINISHED`	Transição disparada em resposta aos seguintes eventos: 1. Usuário removeu uma instância de atividade de análise que se encontra nos estados `CREATED`, `READY`, `RUNNING`, `SUCCEEDED` ou `FAILED`; 2. O tempo máximo (se definido) para a a execução da instância de atividade de análise expirou; 3. O tempo máximo (se definido) para a recuperação de conjuntos de dados de saída ou relatórios de erro de uma instância de atividade de análise expirou.

Fonte: Autoria própria.

B Controles de hipermídia definidos para serviços RAS

Controles de hipermídia permitem que um usuário utilize um serviço de análise sem necessitar de conhecer, de antemão, os esquemas de endereçamento utilizados. Cada um desses controles representa uma relação semântica entre os recursos do serviço que, se navegada, torna transparente o endereço concreto de cada recurso.

A Tabela 19 descreve as relações utilizadas nos controles de hipermídia definidos para serviços RAS. Esta tabela contém o tipo de relação definida entre os recursos, os recursos de um serviço RAS que podem retornar esse controle e uma descrição do significado esperado para essa relação. Elementos entre chaves em uma relação devem ser substituídos pelos valores descritos.

Tabela 19 – Relações utilizadas nos controles de hipermídia.

Relação	Retornado junto a	Descrição
`self`	Todos os recursos	Informa o URI pelo qual o recurso está disponível.
`create-instance`	Recurso raiz do serviço	Informa o recurso que permite criar novas instâncias de atividade de análise (coleção `new-instances`).
`parameters`	Instância em qualquer estado, exceto `RUNNING`	Localização do recurso que representa os parâmetros de execução da instância de atividade de análise.

Continua.

Tabela 19 – Relações utilizadas nos controles de hipermídia (Continuação).

Relação	Retornado junto a	Descrição
`parameters/{parameter-name}`	Instância em qualquer estado, exceto `RUNNING`	Localização do recurso que representa o parâmetro de execução `{parameter-name}`.
`inputs`	Instância em qualquer estado, exceto `RUNNING`	Localização do recurso que representa os conjuntos de dados de entrada da instância de atividade de análise.
`inputs/{input-name}`	Instância em qualquer estado, exceto `RUNNING`	Localização do recurso que representa o conjunto de dados de entrada `{input-name}`.
`inputs/{input-name}/{file-name}`	Instância em qualquer estado, exceto `RUNNING`	Localização do arquivo `{file-name}` do conjunto de dados de entrada `{input-name}`
`outputs`	Instância no estado `SUCCEEDED`	Localização do recurso que representa os conjuntos de dados de saída da instância de atividade de análise.
`outputs/{output-name}`	Instância no estado `SUCCEEDED`	Localização do recurso que representa o conjunto de dados de saída `{output-name}`.

Continua.

Tabela 19 – Relações utilizadas nos controles de hipermídia (Continuação).

Relação	Retornado junto a	Descrição
`outputs/{output-name}/{file-name}`	Instância no estado `SUCCEEDED`	Localização do arquivo `{file-name}` do conjunto de dados de saída `{output-name}`
`error-report`	Instância no estado `FAILED`	Localização do relatório de erros produzido pela execução da instância de atividade de análise.
`start-execution`	Instância no estado `READY`	Localização do recurso que permite iniciar a execução da instância de atividade de análise.

Fonte: Autoria própria.

C Linguagem ActDL

A linguagem ActDL é representada na Listagem 28 por meio de Forma de Backus-Naur Estendida (EBNF). Símbolos do alfabeto não-terminal da linguagem são representados entre os sinais de < e >. Símbolos do alfabeto terminal são representados entre aspas. A regra de produção para um símbolo não-terminal é apresentada após o símbolo ::=. O símbolo | separa diferentes alternativas para uma dada regra de produção. O símbolo & entre dois elementos representa que o elemento anterior e o elemento posterior formam um grupo não-ordenado, de modo que os elementos desse grupo podem estar presentes em uma ordem qualquer.

Por padrão, um elemento apresentado em uma regra de produção possui cardinalidade unitária, devendo ser encontrado uma vez e apenas uma vez na posição em que reside. Os símbolos ?, * e + são modificadores de cardinalidade que são aplicados ao elemento imediatamente anterior. O símbolo ? representa que o elemento anterior pode ser encontrado zero ou uma vez. O símbolo * representa que o elemento pode ser encontrado repetido zero ou mais vezes. O símbolo + representa que o elemento pode ser repetido zero ou mais vezes. Elementos podem ser agrupados em e expressões por meio de parênteses, de modo que os diversos modificadores possam ser aplicados a toda a expressão.

Os símbolos !, .., . e -> auxiliam na definição de conjuntos de elementos terminais. O símbolo ! é um prefixo de negação que, quando aplicado a um conjunto de elementos terminais, representa que qualquer caractere terminal que não esteja definido pelo conjunto ao qual o símbolo é aplicado pode ser encontrado na posição indicada. O símbolo .. entre dois caracteres terminais define um intervalo (inclusivo) de caracteres válidos para aquela posição. O símbolo . representa que qualquer caractere é válido na posição indicada. O símbolo -> indica que quaisquer caracteres podem ser aparecer entre a sequência de caracteres anterior e e a sequência de caracteres posterior.

Listagem 28 – Gramática da linguagem ActDL em EBNF.

```
1 <Activity> ::=
2   "activity" <EString> "{"
3     ( "remark" <EString> ";")?
4     ( "on" "{" <InputDataset>+ "}" )?
5     ( "with" "{" <Parameter>+ "}" )?
6     ( "produces" "{" <OutputDataset>+ "}" )?
7     "using" <Tool>
8   "}";
9
```

```
<Tool> ::= <CommandLineTool>;

<Dataset> ::= <InputDataset> | <OutputDataset>;

<CommandLineEntryList> ::=
  <LiteralCommandLineEntryList>
  | <DatasetCommandLineEntryList>
  | <ParameterCommandLineEntryList>;

<StringListManipulator> ::=
  <Join>
  | <PrependEach>
  | <AppendEach>
  | <AppendListWith>
  | <PrependListWith>;

<EString> ::= <MULTILINE_STRING> | <STRING> | <ID> ;

<Parameter> ::=
  "parameter" <EString> ":"
  <ParameterType> "[" <EBigInteger> "," <EBigInteger> "]"
  ( "=" [ <EString> ( "," <EString> )* "]" )?
  ( "{"
          ( "remark"  <EString> ";" )?
          ( "constraints" "["
      <Constraint> ( "," <Constraint>)*
    "]" )?
  "}" )?
  ";" ;

<InputDataset> ::=
  'dataset' <EString> ':'
  (<EString>)? '[' <EBigInteger> ',' <EBigInteger> ']'
  ( '{'
    ('remark' <EString> ';')?
    ('constraints' '['
          <Constraint>
      (',' <Constraint>)*
    ']' ';')?
      '}' )?
      ';' ;

<OutputDataset> ::=
  'dataset' <EString> ':'
  (<EString>)? '[' <EBigInteger> ',' <EBigInteger> ']'
  ( '{'
    ('remark' <EString> ';')?
    ('constraints' '['
          <Constraint>
      (',' <Constraint>)*
    ']' ';')?
      '}' )?
      ';' ;

<EBigInteger> ::= '-'? <INT>;

<ParameterType> ::= 'STRING' | 'INTEGER' | 'REAL';
```

```
67
68
69
70 <Constraint> ::=
71   'Constraint' <EString>;
72
73
74
75 <CommandLineTool> ::=
76   'executable' <EString> '{'
77     ('redirecting' '{'
78     (
79       ('stdin' 'from' <EString> ';')?
80       & ('stdout' 'to' <EString> ';')?
81       & ('stderr' 'to' <EString> ';')?
82     )
83     '}'
84           )?
85         'commandLineTemplate' '['
86     <CommandLineEntryList>
87     ( ',' <CommandLineEntryList>)*
88   ']'
89   ('returns' '{' <ExitCode>* '}' )?
90   '}';
91
92
93 <ExitCode> ::=
94   <EBigInteger> 'if' <TerminationStatus> (<EString>)? ';'
95   ;
96
97
98 <LiteralCommandLineEntryList> ::=
99   'literals' '[' <EString> ( ',' <EString> )* ']'
100  ('|' <StringListManipulator>)*
101  ;
102
103 <DatasetCommandLineEntryList> ::=
104  'dataset' <EString>
105  ('|' <StringListManipulator>)*
106  ;
107
108 <ParameterCommandLineEntryList> ::=
109  'parameter' <EString>
110  ('|' <StringListManipulator>)*
111  ;
112
113
114
115 <Join> ::= 'Join' <EString>;
116
117 <PrependEach> ::= 'PrependEach' <EString>;
118
119 <AppendEach> ::= 'AppendEach' <EString>;
120
121 <AppendListWith> ::= 'AppendListWith' <EString>;
122
123 <PrependListWith> ::= 'PrependListWith' <EString>;
```

```
124
125 <TerminationStatus> ::= 'SUCCEEDED' | 'FAILED';
126
127 <MULTILINE_STRING> ::= "'''" -> "'''";
128
129 <ID> ::= '^'? ('a'..'z'|'A'..'Z'|'_')
130           (("-")?('a'..'z'|'A'..'Z'|'_'|'0'..'9'))*;
131
132 <INT> ::= ('0'..'9')+;
133
134 <STRING> ::=
135   '"' ( '\\' . | !('\\'|'"') )* '"'
136   | "'" ( '\\' .  | !('\\'|"'") )* "'"
137   ;
```

Fonte: Autoria própria.

D Gramática da linguagem SDDL

A Listagem 29 apresenta a gramática da linguagem SDDL por meio de Forma de Backus-Naur Estendida (EBNF). Símbolos do alfabeto não-terminal da linguagem são representados entre os sinais de `<` e `>`. Símbolos do alfabeto terminal são representados entre aspas. A regra de produção para um símbolo não-terminal é apresentada após o símbolo `::=`. O símbolo `|` separa diferentes alternativas para uma dada regra de produção. O símbolo `&` entre dois elementos representa que o elemento anterior e o elemento posterior formam um grupo não-ordenado, de modo que os elementos desse grupo podem estar presentes em uma ordem qualquer.

Por padrão, um elemento apresentado em uma regra de produção possui cardinalidade unitária, devendo ser encontrado uma vez e apenas uma vez na posição em que reside. Os símbolos `?`, `*` e `+` são modificadores de cardinalidade que são aplicados ao elemento imediatamente anterior. O símbolo `?` representa que o elemento anterior pode ser encontrado zero ou uma vez. O símbolo `*` representa que o elemento pode ser encontrado repetido zero ou mais vezes. O símbolo `+` representa que o elemento pode ser repetido zero ou mais vezes. Elementos podem ser agrupados em e expressões por meio de parênteses, de modo que os diversos modificadores possam ser aplicados a toda a expressão.

Os símbolos `!`, `..`, `.` e `->` auxiliam na definição de conjuntos de elementos terminais. O símbolo `!` é um prefixo de negação que, quando aplicado a um conjunto de elementos terminais, representa que qualquer caractere terminal que não esteja definido pelo conjunto ao qual o símbolo é aplicado pode ser encontrado na posição indicada. O símbolo `..` entre dois caracteres terminais define um intervalo (inclusivo) de caracteres válidos para aquela posição. O símbolo `.` representa que qualquer caractere é válido na posição indicada. O símbolo `->` indica que quaisquer caracteres podem ser aparecer entre a sequência de caracteres anterior e e a sequência de caracteres posterior.

Listagem 29 – Gramática da linguagem SDDL em EBNF.

```
grammar br.usp.ffclrp.dcm.lssb.activityrest.deploymentmodel.DSLSyntax with
    org.eclipse.xtext.common.Terminals

<Deployment>::=
  "deployment"
  "{"
    "of" <Service>
    "into" <ServiceContainer>
    (<Contact>)?
  "}";

<ServiceContainer>::=
  "container"
  <EString>
  "{"
    "base-url" <URL>
    ("description" <EString>)?

  "}"
;


<Service>::=
  "service"
  name=EString
  "{"
    "api-version" <EString>
    ("description" <EString>)?
  "}";

<Contact>::=
  "contact" <EString>
  "{"
    "email" <EString>
    ("url" <URL>)?
  "}";

<EString>::=
  STRING;

<URL>::=
  STRING;


<STRING> ::=
  '"' ( '\\' . | !('\\'|'"') )* '"'
  | "'" ( '\\' .  | !('\\'|"'") )* "'"
  ;
```

Fonte: Autoria própria.

E Regras de transformação para a obtenção de um cliente RAS-*CLI*

Um conjunto de regras de transformação abstratas foi definido para a obtenção de um cliente RAS a partir de uma descrição ActDL e de um modelo de implantação foi definido. Essas regras são executadas de maneira hierárquica, com regras mais gerais invocando e combinando os resultados de regras mais específicas.

O restante desse apêndice está organizado da seguinte maneira: a Seção E.1 apresenta as regras de transformação que definem a estrutura do projeto de um cliente RAS-*CLI*; a Seção E.2 apresenta as regras de transformação que definem os componentes específicos de serviço de um cliente RAS-*CLI*; por fim, a Seção E.3 apresenta as regras de transformação que definem o componente `CLI Controller` deste cliente.

A implementação dessas transformações pode ser encontrada no repositório Git do *framework* Activity-REST[1].

E.1 Regras de transformação que definem a estrutura do projeto do cliente RAS-*CLI*

A Figura 61 apresenta uma visão geral das regras de transformação PROJ e SPEC. Estas regras definem a estrutura geral do projeto Kotlin que implementa o cliente RAS-*CLI* e estão descritas a seguir:

PROJ: Criar projeto Maven base

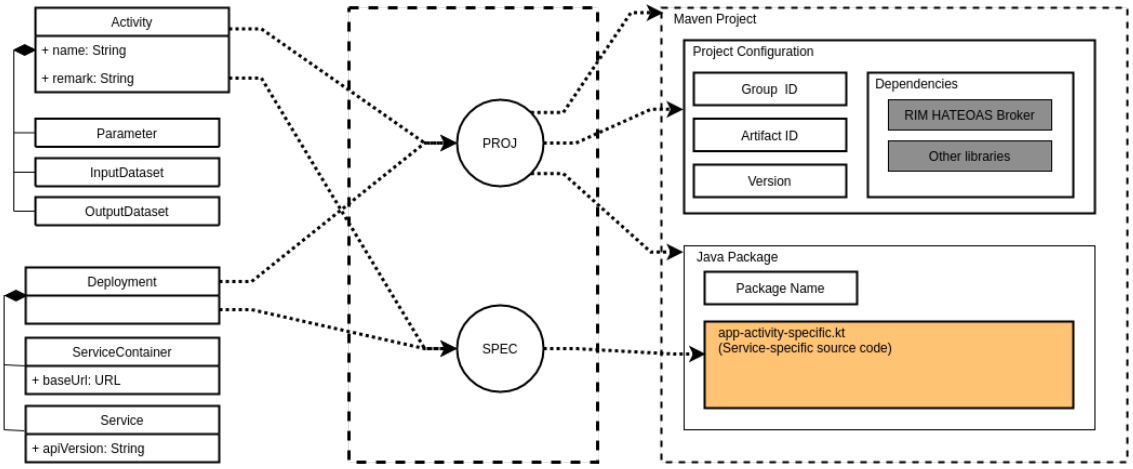
Esta regra tem por objetivo gerar um projeto Maven base a partir de produzido para um par descrição ActDL e modelo de implantação. Esse projeto deverá apresentar um pacote Java nomeado `{activity.name}`, bem como toda a configuração necessária para compilar e empacotar a aplicação cliente, incluindo todas as dependências para o novo executável. A Tabela 20 apresenta os valores padrões para os identificadores do projeto Maven criado.

SPEC: Criar código fonte específico do serviço

Esta regra tem por objetivo gerar um arquivo para conter o código fonte específico do serviço a partir de um par contendo uma descrição ActDL e

[1] `https://purl.org/cawal/lssb/activity-rest-framework`

Figura 61 – Visão geral das regras de transformação PROJ e SPEC na transformação modelo-para-texto para a obtenção de um cliente RAS-*CLI*.



Metaclasses da descrição ActDL e de um modelo de implantação são apresentadas à esquerda. Estruturas de interesse do cliente RAS-*CLI* são apresentadas à direita. Regras de transformação são apresentadas ao centro como círculos nomeados. Uma seta associando uma metaclasse a uma regra de transformação indica que elementos daquela metaclasse nos modelos fonte são utilizados para a execução da regra de transformação. Uma seta associando uma regra de transformação a uma estrutura do cliente RAS-*CLI* representa que essa estrutura é produzida pela execução da regra de transformação. Um retângulo tracejado circundando as regras de transformação representa o contexto mais geral no qual as regras apresentadas são invocadas. Fonte: Autoria própria.

Tabela 20 – Valores padrões para o projeto Maven criado.

Atributo	Valor padrão
Id do artefato Maven	`{activity.name}-client`
Grupo do artefato Maven	`{activity.name}`
Versão do Artefato Maven	`{deployment.service.version}`

Fonte: Autoria própria.

um modelo de implantação. Esse arquivo conterá o código fonte específico de serviço, definindo os argumentos da linha de comando do cliente, a criação dos objetos de dados da instância da atividade de análise, a passagem desse objeto de dados para o cliente RAS HATEOAS e o armazenamento dos conjuntos de dados de saída da instância de atividade de análise após execução. Adicionalmente, o arquivo criado por essa regra deve ser incluído no

caminho `src/main/java/{activity.name}/app-activity-specific.kt` do projeto Maven criado em PROJ.

A regra SPEC invoca as regras HEADER, CLI, BUILDER e RESULT e combina os resultados retornados por essas regras.

E.2 Regras de transformação que definem a os componentes específicos de serviço do cliente RAS-*CLI*

A Figura 62 apresenta uma visão geral das regras de transformação HEADER, CLI, BUILDER e RESULT. Essas regras definem a estrutura geral dos componentes específicos de serviço de um cliente RAS-*CLI* e estão descritas a seguir:

HEADER: Incluir cabeçalho

Esta regra tem por objetivo definir o cabeçalho do arquivo `app-activity-specific.kt`. Este cabeçalho deve importar as classes necessárias para a definição da interface de linha de comando e para a execução da interação com o serviço de análise para a execução da instância de atividade de análise.

CLI: Criar componente `CLI Controller`

Essa regra tem por objetivo definir o componente `CLI Controller` a partir de uma descrição ActDL e de um modelo de implantação. Esse componente deve ser capaz de compreender os argumentos de linha de comando e inicializar um conjunto de variáveis locais com os valores informados pelo usuário.

A regra CLI invoca as regras PARAM-MAND-ARG, PARAM-OPT-ARG, INPUT-ARG, OUTPUT-ARG, EXTRA-ARGS, PARAM-VAR, e DATASET-VAR, combinando o resultado retornado por essas regras (veja Figura 63).

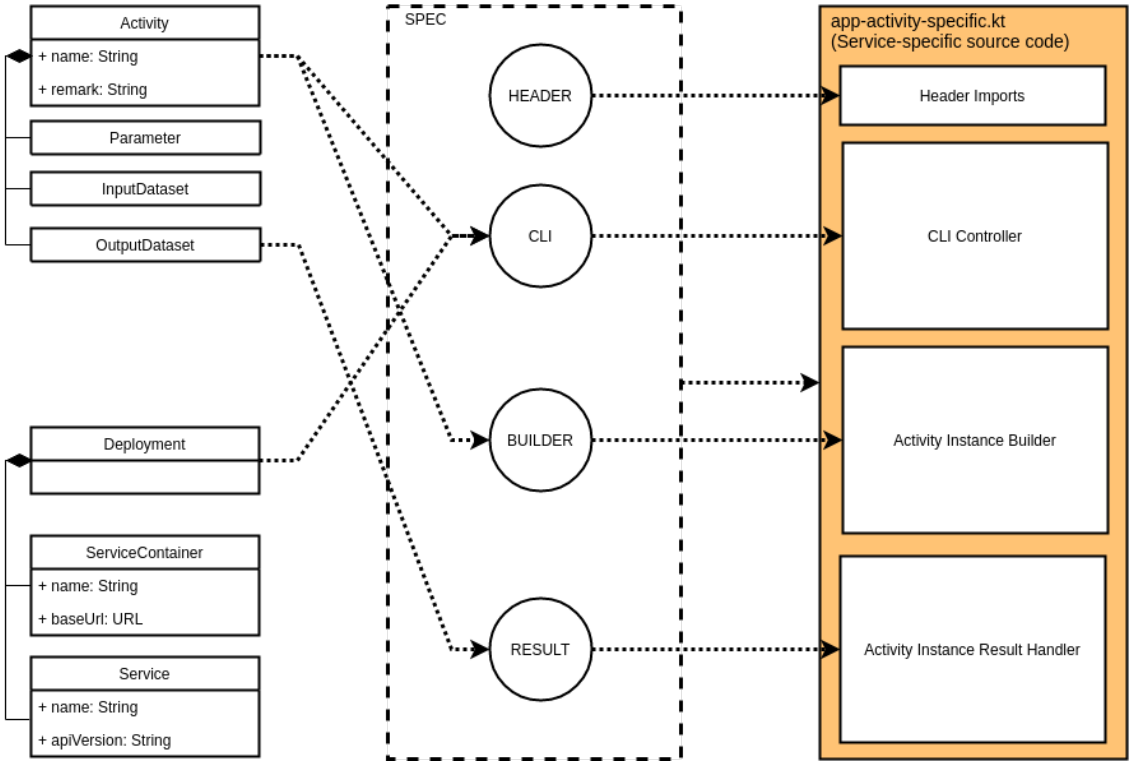
BUILDER: Criar componente `Activity Instance Builder`

Esta regra tem por objetivo criar o componente `Activity Instance Builder` a partir de uma descrição ActDL. Esse componente deve possuir o código necessário para transformar os valores dos argumentos definidos no componente `CLI Controller` em um objeto que representa uma instância de atividade de análise. Tal objeto será, posteriormente, submetido ao componente `RAS HATEOAS Broker` para a execução da atividade de análise.

RESULT: Criar componente `Activity Instance Result Handler`

Esta regra tem por objetivo criar o componente `Activity Instance Result Handler` a partir de uma descrição ActDL. Este componente deve possuir o código necessário para receber os resultados da execução da atividade de análise,

Figura 62 – Visão geral das regras de transformação HEADER , CLI, BUILDER e RESULT.



Metaclasses da descrição ActDL e de um modelo de implantação são apresentadas à esquerda. Estruturas de interesse do cliente RAS-*CLI* são apresentadas à direita. Regras de transformação são apresentadas ao centro como círculos nomeados. Uma seta associando uma metaclasse a uma regra de transformação indica que elementos daquela metaclasse nos modelos fonte são utilizados para a execução da regra de transformação. Uma seta associando uma regra de transformação a uma estrutura do cliente RAS-*CLI* representa que essa estrutura é produzida pela execução da regra de transformação. Um retângulo tracejado circundando as regras de transformação representa o contexto mais geral no qual as regras apresentadas são invocadas. Fonte: Autoria própria.

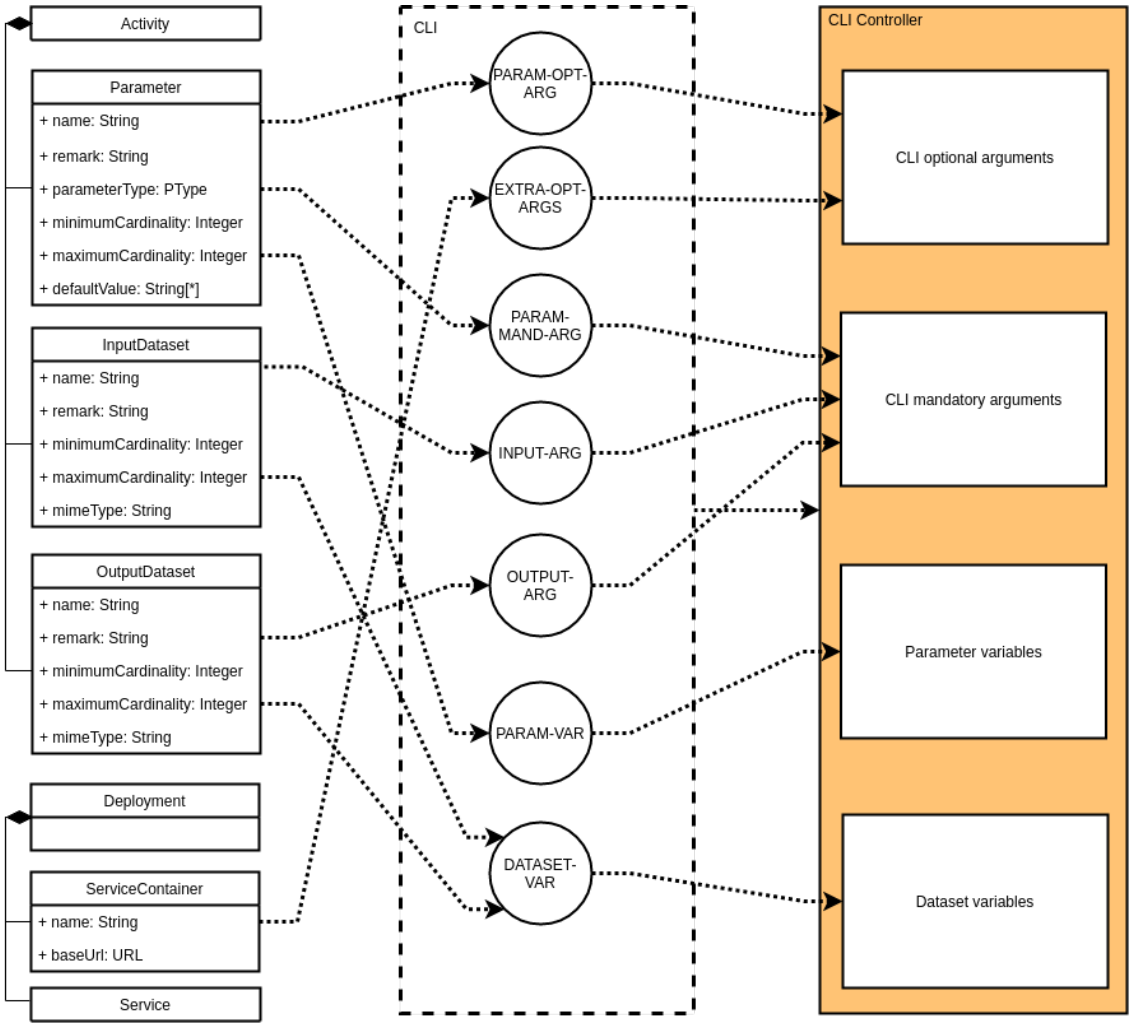
retornado pelo componente `RAS HATEOAS Broker`, e armazená-los no caminho passado como argumento para cada conjunto de dados de saída conforme definido na linha de comanda interpretada pelo componente `CLI Controller`.

E.3 Regras de transformação que definem detalhes do componentes `CLI Controller` do cliente RAS-*CLI*

A Figura 63 apresenta uma visão geral das regras de transformação PARAM-MAND-ARG, PARAM-OPT-ARG, INPUT-ARG, OUTPUT-ARG, EXTRA-ARGS, PARAM-VAR, e DATASET-VAR. Essas regras definem detalhes do componente

`CLI Controller` do cliente RAS-*CLI* e estão descritas a seguir:

Figura 63 – Visão geral das regras de transformação PARAM-MAND-ARG, PARAM-OPT-ARG, INPUT-ARG, OUTPUT-ARG, EXTRA-ARGS, PARAM-VAR, e DATASET-VAR.



Metaclasses da descrição ActDL e de um modelo de implantação são apresentadas à esquerda. Estruturas de interesse do cliente RAS-*CLI* são apresentadas à direita. Regras de transformação são apresentadas ao centro como círculos nomeados. Uma seta associando uma metaclasse a uma regra de transformação indica que elementos daquela metaclasse nos modelos fonte são utilizados para a execução da regra de transformação. Uma seta associando uma regra de transformação a uma estrutura do cliente RAS-*CLI* representa que essa estrutura é produzida pela execução da regra de transformação. Um retângulo tracejado circundando as regras de transformação representa o contexto mais geral no qual as regras apresentadas são invocadas. Fonte: Autoria própria.

PARAM-MAND-ARG: Criação de argumentos de linha de comando para parâmetros sem valor padrão

Esta regra tem por objetivo criar o código necessário para representar um argumento de linha de comando obrigatório a partir de uma instância de `Parameter` que não defina um valor padrão. Tal argumento deve ter as mesmas cardinalidades mínima e máxima que o parâmetro definido na descrição ActDL. A descrição do parâmetro deve ser utilizada na descrição do argumento quando a *string* de ajuda da aplicação for requerida pelo usuário. O argumento criado deve ser precedido por `--{parameter.name}`, quando usado na linha de comando, cada valor passado para o parâmetro deve ser indicado em seguida, separadamente.

PARAM-OPT-ARG: Criação de argumentos de linha de comando para parâmetros com valores padrão

Esta regra tem por objetivo criar o código necessário para representar um argumento de linha de comando a partir de uma instância de `Parameter` que defina um valor padrão. Tal argumento deve ter as mesmas cardinalidades mínima e máxima que o parâmetro definido na descrição ActDL. A descrição do parâmetro deve ser utilizada na descrição do argumento quando a *string* de ajuda da aplicação for requerida pelo usuário.

O argumento criado deve ser precedido por `--{parameter.name}`, quando usado na linha de comando, cada valor passado para o parâmetro deve ser indicado em seguida, separadamente. Tal argumento deve ter as mesmas cardinalidades mínima e máxima que o parâmetro definido na descrição ActDL. Cada valor passado para o parâmetro deve ser indicado em seguida, separadamente. Porém, se o usuário não utilizar esse argumento durante a chamada ao cliente RAS-*CLI*, deve ser utilizado o valor padrão definido na descrição ActDL.

INPUT-ARG: Criação de argumentos de linha de comando para conjuntos de dados de entrada

Esta regra tem por objetivo criar o código necessário para representar um argumento de linha de comando obrigatório a partir de uma instância de `InputDataset`. Tal argumento deve ter as mesmas cardinalidades mínima e máxima que o conjunto de dados de entrada definido na descrição ActDL. A descrição do conjunto de dados de entrada deve ser utilizada na descrição do argumento quando a *string* de ajuda da aplicação for requerida pelo usuário.

O argumento criado deve ser precedido por `--{inputDataset.name}`, quando usado na linha de comando. Cada arquivo passado para o conjunto de dados deve ser indicado em seguida por meio dos caminhos desses arquivos, separadamente.

OUTPUT-ARG: Criação de argumentos de linha de comando para conjuntos de dados de saída

Esta regra tem por objetivo criar o código necessário para representar um argumento de linha de comando obrigatório a partir de uma instância de `OutputDataset`. Tal argumento deve ter as mesmas cardinalidades mínima e máxima que o conjunto de dados de entrada definido na descrição ActDL. A descrição do conjunto de dados de entrada deve ser utilizada na descrição do argumento quando a *string* de ajuda da aplicação for requerida pelo usuário.

O argumento criado deve ser precedido por `--{inputDataset.name}`, quando usado na linha de comando. Cada arquivo passado para o conjunto de dados deve ser indicado em seguida por meio dos caminhos desses arquivos, separadamente.

EXTRA-ARGS: Criação de argumentos opcionais extras

Esta regra tem por objetivo criar o código necessário para incluir dois argumentos de linha de comando opcionais no cliente RAS-*CLI*.

O argumento `--use-sse` indica o uso da estratégia de notificação de término da execução da atividade de análise por *Server-Side Events*. Por sua vez, o argumento `--delete-after-completion` indica que a instância de atividade de análise deve ser removida do serviço após os conjuntos de dados de saída serem recuperados. Caso o usuário não informe valores para esses argumentos, são utilizados os valores padrões para "notificação por *polling*" e "não remover a instância de atividade de análise do serviço após sua execução", respectivamente.

PARAM-VAR: Criação de variável para armazenar o valor de um parâmetro de execução

Esta regra tem por objetivo criar o código necessário para armazenar os valores recebidos para um parâmetro de execução durante a execução do cliente RAS-*CLI* a partir de uma instância de `Parameter`. O nome dessa variável deve ser um nome de variável válido para a linguagem Kotlin. Nesse sentido, caracteres não permitidos para o nome de uma variável no Kotlin presentes no nome do parâmetro de execução na descrição ActDL, tais como o caractere *hífen* ("`-`"), devem ser substituídos pelo caractere de sublinhado ("`_`"). O tipo da variável que irá receber os valores do argumento gerado a partir de um parâmetro de execução deve ser um tipo primitivo ou classe da linguagem Java. A Tabela 21 apresenta o mapeamento dos tipos de parâmetros de uma descrição ActDL para os tipos de variáveis usados para esses parâmetros no cliente RAS-*CLI*.

Tabela 21 – Mapeamento dos tipos de parâmetros de uma descrição ActDL para o tipo da variável no cliente RAS.

Tipo do parâmetro	Card. máx.	Classe Java
STRING	1	java.lang.String
INTEGER	1	int
REAL	1	double
BOOLEAN	1	boolean
STRING	> 1	java.util.List<java.lang.String>
INTEGER	> 1	java.util.List<java.lang.Integer>
REAL	> 1	java.util.List<java.lang.Double>
BOOLEAN	> 1	java.util.List<java.lang.Boolean>

Fonte: Autoria própria.

DATASET-VAR: Criação de uma variável para armazenar os caminhos de arquivos informados para um conjunto de dados de entrada ou saída

Esta regra tem por objetivo criar o código necessário para armazenar os caminhos de arquivos informados para um conjunto de dados durante a execução do cliente RAS-*CLI* a partir de uma instância de `InputDataset` ou de `OutputDataset`. O nome dessa variável deve ser um nome de variável válido para a linguagem Kotlin. Nesse sentido, caracteres não permitidos para o nome de uma variável no Kotlin presentes no nome do parâmetro de execução na descrição ActDL, tais como o caractere *hífen* ("`-`"), devem ser substituídos pelo caractere de sublinhado ("`_`"). A Tabela 22 apresenta o mapeamento da cardinalidade de um conjunto de dados descrito em uma descrição ActDL para o tipo da variável que irá receber esses conjuntos de dados no cliente RAS-*CLI*.

Tabela 22 – Mapeamento da cardinalidade de um conjunto de dados em uma descrição ActDL para a classe que o representa no cliente RAS.

Card. máx.	Classe Java
1	java.io.File
> 1	java.util.List<java.io.File>

Fonte: Autoria própria.

F Regras de transformação para a obtenção de um cliente RAS-Galaxy

Um conjunto de regras de transformação abstratas foi definido para a obtenção de um cliente RAS-Galaxy a partir de uma descrição ActDL e de um modelo de implantação foi definido. Essas regras são executadas de maneira hierárquica, de modo que regras mais gerais invocam regras mais específicas e compõe os resultados destas de modo a obter os artefatos que compõe o cliente.

O restante desse apêndice está organizado da seguinte maneira: a Seção F.1 apresenta as regras de transformação que definem a estrutura do projeto de um cliente RAS-*Galaxy*; a Seção F.2 apresenta as regras de transformação que definem o conteúdo de um arquivo `tool.xml` desse cliente; por fim, a Seção F.3 apresenta as regras de transformação que definem os elementos que representam entradas e saídas de uma atividade de análise presentes no arquivo `tool.xml`.

A implementação dessas transformações pode ser encontrada no repositório Git do *framework* Activity-REST[1].

F.1 Regras de transformação que definem a estrutura do projeto do cliente RAS-Galaxy

A Figura 64 apresenta as regras de transformação PROJ, CLIENT e XML. Essas regras definem a estrutura geral do cliente RAS-Galaxy.

PROJ: Definição de um projeto para o cliente RAS-GALAXY

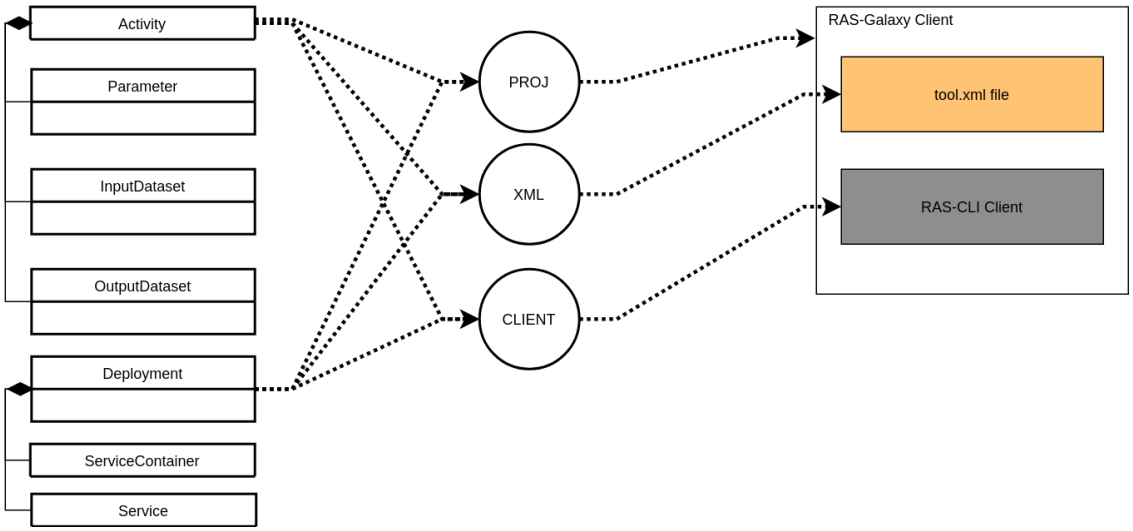
Esta regra tem por objetivo definir um diretório deve ser criado para conter os artefatos do cliente RAS-Galaxy a partir de uma descrição ActDL e de um modelo de implantação. A regra PROJ invoca as regras CLIENT e XML para a criação dos artefatos a serem incluídos nesse diretório.

CLIENT: Inclusão de um cliente RAS-*CLI*

Esta regra tem por objetivo produzir um cliente RAS-*CLI* a partir de uma descrição ActDL e de um modelo de implantação. A regra CLIENT invoca a transformação para a obtenção de um cliente RAS-*CLI* definida no Apêndice E.

[1] `https://purl.org/cawal/lssb/activity-rest-framework`

Figura 64 – Regras de transformação que definem a estrutura do projeto do cliente RAS-Galaxy.



Metaclasses da descrição ActDL e de um modelo de implantação são apresentadas à esquerda. Estruturas de interesse do cliente RAS-*Galaxy* são apresentadas à direita. Regras de transformação são apresentadas ao centro como círculos nomeados. Uma seta associando uma metaclasse a uma regra de transformação indica que elementos daquela metaclasse nos modelos fonte são utilizados para a execução da regra de transformação. Uma seta associando uma regra de transformação a uma estrutura do cliente RAS-*Galaxy* representa que essa estrutura é produzida pela execução da regra de transformação. Um retângulo tracejado circundando as regras de transformação representa o contexto mais geral no qual as regras apresentadas são invocadas. Fonte: Autoria própria.

XML: Criação do arquivo `tool.xml`

Esta regra tem por objetivo criar um arquivo adaptador para o ambiente Galaxy (arquivo `tool.xml`) a partir de uma descrição ActDL e de um modelo de implantação. A regra XML invoca a regra TOOL para definir o conteúdo do arquivo produzido.

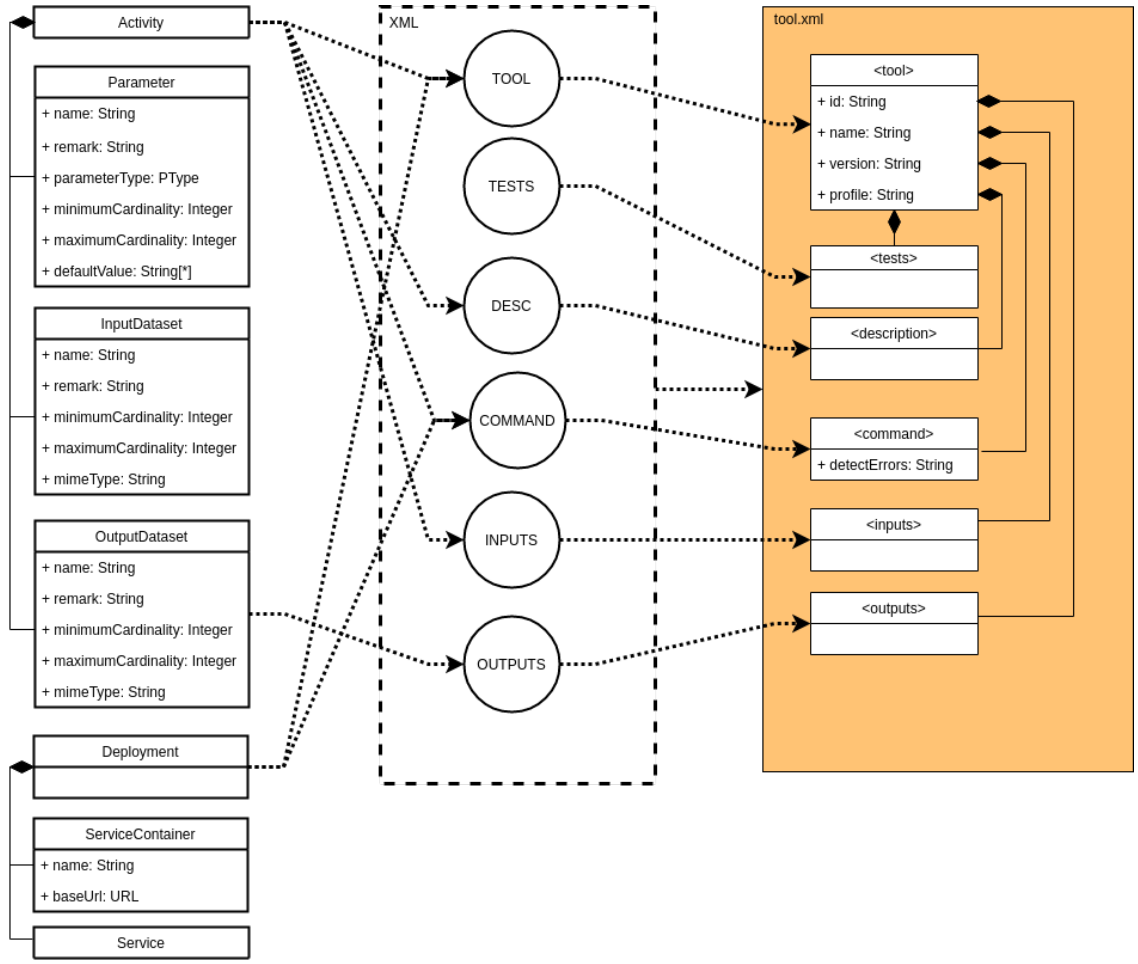
F.2 Regras que definem os elementos no arquivo `tool.xml`

A regra XML cria o arquivo `tool.xml`, invocando um conjunto de regras mais específicas utilizadas para criar o conteúdo desse aquivo. A Figura 65 apresenta as regras de transformação TOOL, DESC,INPUTS, OUTPUTS, TESTS, e COMMAND, que definem os elementos contidos no arquivo `tool.xml`.

TOOL: Criação do elemento `<tool>`

Esta regra tem por objetivo definir o element `<tool>` a partir de uma descrição ActDL e um modelo de implantação. Os atributos do elemento `<tool>` devem

Figura 65 – Regras de transformação que definem o conteúdo de um arquivo `tool.xml` do projeto do cliente RAS-Galaxy.



Metaclasses da descrição ActDL e de um modelo de implantação são apresentadas à esquerda. Estruturas de interesse do cliente RAS-*Galaxy* são apresentadas à direita. Regras de transformação são apresentadas ao centro como círculos nomeados. Uma seta associando uma metaclasse a uma regra de transformação indica que elementos daquela metaclasse nos modelos fonte são utilizados para a execução da regra de transformação. Uma seta associando uma regra de transformação a uma estrutura do cliente RAS-*Galaxy* representa que essa estrutura é produzida pela execução da regra de transformação. Um retângulo tracejado circundando as regras de transformação representa o contexto mais geral no qual as regras apresentadas são invocadas. Fonte: Autoria própria.

ser obtidos segundo as definições contidas na Tabela 23. A regra TOOL invoca as regras DESC, COMMAND, INPUTS,OUTPUTS e TESTS para obter os elementos-filhos de `<tool>`, compondo os resultados dessas regras.

Tabela 23 – Atributos do elemento `<tool>`.

Atributo	Descrição
`id`	Deve ser obtido do atributo `name` do elemento `Activity` da descrição ActDL. Caracteres inválidos devem ser substituídos por *underscore* (“_”).
`name`	Deve ser obtido a partir do atributo `name` do elemento `Activity` da descrição ActDL.
`version`	Deve ser obtido a partir do atributo `version` do elemento `Service` do modelo de implantação.
`profile`	Versão mínima do ambiente Galaxy capaz de utilizar a extensão definida. Valor fixo: “`16.04`”

Fonte: Autoria própria.

DESC: Criação do elemento `<description>`

Esta regra tem por objetivo produzir um elemento `<description>` a partir de uma descrição ActDL. Este elemento `<description>` deve conter como texto interno a descrição da atividade de análise obtida a partir do atributo `remark` do elemento `Activity` da descrição ActDL.

INPUTS: Criação do elemento `<inputs>`

Esta regra tem por objetivo produzir um elemento `<inputs>` a partir de uma descrição ActDL. Esta regra invoca a regra PARAMS sobre cada elemento `Parameter` e `InputDataset` da atividade de análise, de modo a obter os elementos-filhos de `<inputs>`

OUTPUTS: Criação do elemento `<outputs>`

Esta regra tem por objetivo produzir um elemento `<outputs>` a partir de uma descrição ActDL. Esta regra invoca a regra DATA sobre cada elemento `OutputDataset` da atividade de análise, de modo a obter os elementos-filhos de `<output>`.

TESTS: Criação do elemento `<tests>`

Esta regra tem por objetivo produzir um elemento `<tests>` deve ser incluído no elemento `<tool>`. Este elemento pode estar vazio.

COMMAND: Criação do elemento `<command>`

Esta regra tem por objetivo produzir um elemento `<command>` a partir de uma descrição ActDL e de um modelo de implantação. Este elemento deve possuir um atributo `detect_errors` com valor `exit_code`. Este elemento deve conter um nó de texto do tipo `CDATA` com a invocação da ferramenta definida usando a linguagem de definição de *templates* Cheetah.

F.3 Regras que definem os elementos `<param>`, `<repeat>` e `<data>` no arquivo `tool.xml`

As regra INPUTS e OUTPUTS criam os elementos base para a definição dos parâmetros de entrada e conjuntos de dados de saída no arquivo `tool.xml`, respectivamente. A Figura 65 apresenta as regras de transformação PARAMS, DATA e TYPE, que definem os elementos que definem esses parâmetros e conjuntos de dados.

PARAMS: Descrição de parâmetros de execução e conjuntos de dados de entrada

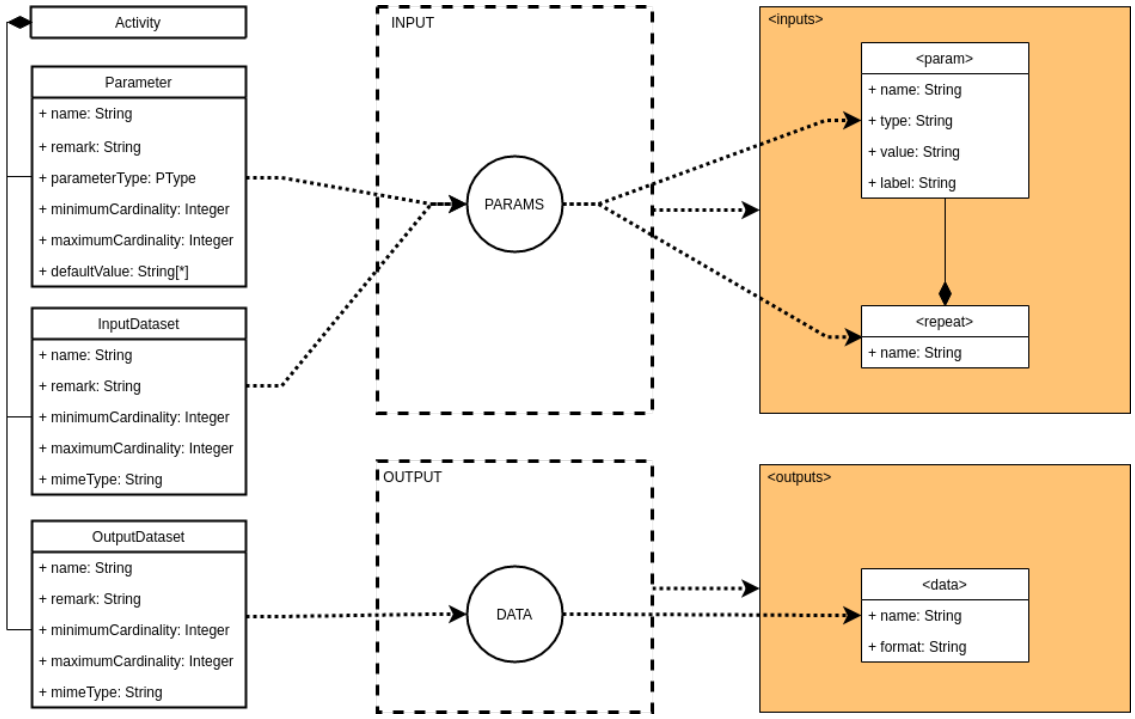
Esta regra tem por objetivo definir um elemento `<param>` ou `<repeat>` para representar um parâmetro de execução ou conjunto de dados de entrada. Essa regra recebe como entrada uma instância de `Parameter` ou de `InputDataset`.

A regra PARAMS deve produzir um elemento do tipo `<param>` ou do tipo `<repeat>` de acordo com a cardinalidade máxima desse parâmetro de execução ou conjunto de dados. Parâmetros de execução ou conjuntos de dados com cardinalidade máxima de 1 deve ser representado por um elemento `<param>` contendo os atributos apresentados na Tabela 24. Por sua vez, parâmetros de execução ou conjuntos de dados com cardinalidade máxima maior que 1 devem ser representados por um elemento `<repeat>` contendo os atributos apresentados na Tabela 25. Este elemento `<repeat>` deve possuir um elemento `<param>` contendo os atributos apresentados na Tabela 26.

DATA: Descrição dos conjuntos de dados de saída

Esta regra tem por objetivo pruduzir um elemento `<data>` a partir de um elemento `OutpuDataset`. O elemento `<data>` produzido deve possuir os atributos apresentados na Tabela 23.

Figura 66 – Regras que definem os elementos `<param>`, `<repeat>` e `<data>` no arquivo `tool.xml`.



Metaclasses da descrição ActDL e de um modelo de implantação são apresentadas à esquerda. Estruturas de interesse do cliente RAS-*Galaxy* são apresentadas à direita. Regras de transformação são apresentadas ao centro como círculos nomeados. Uma seta associando uma metaclasse a uma regra de transformação indica que elementos daquela metaclasse nos modelos fonte são utilizados para a execução da regra de transformação. Uma seta associando uma regra de transformação a uma estrutura do cliente RAS-*Galaxy* representa que essa estrutura é produzida pela execução da regra de transformação. Um retângulo tracejado circundando as regras de transformação representa o contexto mais geral no qual as regras apresentadas são invocadas. Fonte: Autoria própria.

TYPE: Definição do tipo Galaxy de um parâmetro ou conjunto de dados

Esta regra tem por objetivo definir o tipo de dados Galaxy de um parâmetro de execução ou conjunto de dados. O tipo de dados Galaxy deve ser especificado segundo as seguintes regras: Caso for derivado de um parâmetro de execução, o tipo de dado é dado pelo atributo `parameterType` da instância de `Parameter` da descrição ActDL segundo a Tabela 28. Caso for derivado de um conjunto de dados de entrada ou saída, o tipo de dados deve ser "`tabular`", se este conjunto de dados possuir *MIME Types* `text/tsv` ou `text/csv`. Caso contrário, o tipo de dados deve ser definido como "`data`".

Tabela 24 – Atributos do elemento `<param>` criado a partir de um conjunto de dados ou parâmetro de execução com cardinalidade máxima igual a 1 da descrição ActDL.

Atributo	Descrição
`name`	Deve ser obtido a partir do atributo `name` do elemento `Parameter` ou `InputDataset` da descrição ActDL. Este valor deve ser tratado de modo a substituir caracteres inválidos por *underscore* ("_").
`type`	Deve ser obtido do atributo `mimeType` do elemento `Parameter` ou `InputDataset` da descrição ActDL segundo a regra TYPE.
`value` (apenas para `Parameter`)	Deve ser obtido a partir do atributo `defaultValues` do elemento `Parameter`. Deve ser representado como valores separados por vírgula.
`label`	Deve ser obtido a partir do atributo `remark` do elemento `Parameter` ou `InputDataset` da descrição ActDL.

Fonte: Autoria própria.

Tabela 25 – Atributos do elemento `<repeat>` produzido a partir de um conjunto de dados ou parâmetro de execução da descrição ActDL.

Atributo	Descrição
`name`	Deve ser obtido a partir do atributo `name` do elemento `Parameter` ou `InputDataset` da descrição ActDL. Este valor deve ser tratado de modo a substituir caracteres inválidos por *underscore* ("_").
`title`	Deve ser obtido a partir do atributo `name` do elemento `Parameter` ou `InputDataset` da descrição ActDL.

Fonte: Autoria própria.

Tabela 26 – Atributos do elemento `<param>` interno ao elemento `<repeat>` produzido a partir de um conjunto de dados ou parâmetro de execução com cardinalidade máxima maior que 1 da descrição ActDL.

Atributo	Descrição
`name`	Valor: "`value`"
`type`	Deve ser obtido do atributo `mimeType` do elemento `Parameter` ou `InputDataset` da descrição ActDL segundo a regra TYPE.
`value` (apenas para `Parameter`)	Deve ser obtido a partir do atributo `defaultValues` do elemento `Parameter`. Deve ser representado como valores separados por vírgula.
`label`	Deve ser obtido a partir do atributo `remark` do elemento `Parameter` ou `InputDataset` da descrição ActDL.

Fonte: Autoria própria.

Tabela 27 – Atributos do elemento `<data>` produzido a partir de um conjunto de dados de saída da descrição ActDL.

Atributo	Descrição
`name`	Deve ser obtido a partir do atributo `name` do elemento `OutputDataset` da descrição ActDL. Este valor deve ser tratado de modo a substituir caracteres inválidos por *underscore* ("_").
`type`	Deve ser obtido do atributo `mimeType` do elemento `OutputDataset` da descrição ActDL segundo a regra TYPE.
`format`	Deve ser obtido do atributo `mimeType` do elemento `OutputDataset` da descrição ActDL. Este valor é obtido do subtipo do *MIME type*.

Fonte: Autoria própria.

Tabela 28 – Mapeamento do tipo de um parâmetro presente na descrição ActDL e o tipo do parâmetro análogo no arquivo `tool.xml`.

Parameter Type	Tipo Galaxy
`BOOLEAN`	`boolean`
`INTEGER`	`integer`
`REAL`	`float`
`STRING`	`text`

Fonte: Autoria própria.

G Regras de transformação para a geração de um documento XSD

Um conjunto de regras de transformação abstratas foi definido para guiar a implementação das transformações modelo-para-texto para obter um documento XSD que descreve os tipos de dados utilizados em um serviço RAS a partir de uma descrição ActDL e um modelo SDDM. Essas regras são executadas de maneira hierárquica, com regras mais gerais invocando e combinando os resultados de regras mais específicas. Neste sentido, esta transformação é realizada em dois níveis hierárquicos. Inicialmente, uma regra mais geral define a estrutura geral do documento XSD. Essa regra geral invoca um conjunto de regras mais específicas para definir os elementos desse documento.

G.1 Regra de transformação que define a estrutura do documento XSD

XSD: Criar especificação XSD base

Uma especificação XSD deve ser criada a partir de um modelo AADM e de um modelo SDDM. A especificação base deve conter:

- uma declaração `<?xml version="1.0"encoding="UTF-8"?>` na primeira linha do arquivo;
- um único elemento `<xs:schema>` como elemento raiz XML.

A regra XSD invoca as regras NAMESPACES, COMMON, PARAMS e DATASETS e combina os resultados retornados por estas.

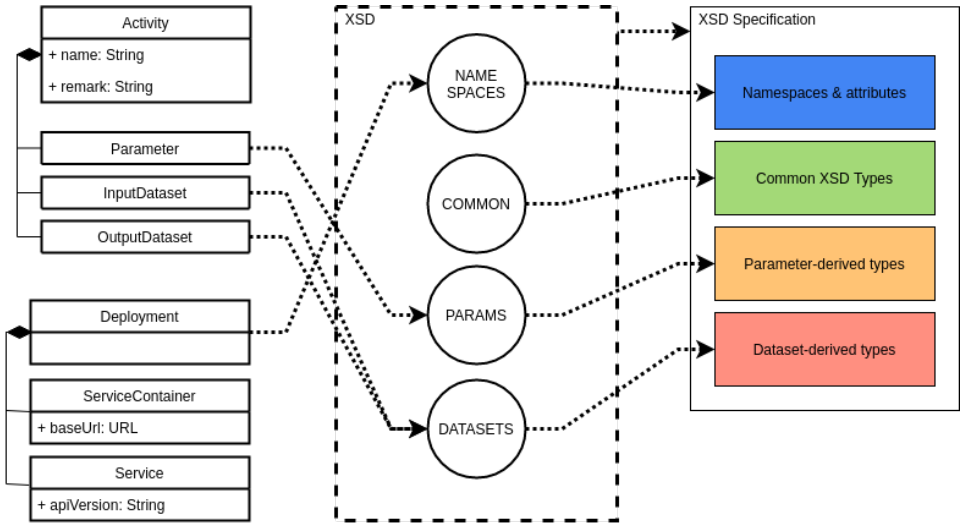
G.2 Regras de transformação que definem os elementos do documento XSL

A Figura 67 apresenta uma visão geral das regras de transformação invocadas pela regra XSD.

NAMESPACES: Definir os namespaces e atributos da descrição XSD

O elemento `<xs:schema>` deve definir um atributo `@targetNamespace`. Este atributo assume o valor `http://`*`{endereço}`*`:`*`{porta}`*` /`*`{recursoBase}`*`/xsd`,

Figura 67 – Visão geral das regras de transformação para obtenção de uma especificação XSD.



Fonte: Autoria própria.

Tabela 29 – *Namespaces* e atributos do elemento `<xs:schema>`.

Atributo / Namespace	Valor
`xmlns`	`http://www.w3.org/2001/XMLSchema`
`xmlns:xs`	`http://www.w3.org/2001/XMLSchema`
`xmlns:aa`	*Namespace alvo da especificação XSD.*
`targetNamespace`	*Namespace alvo da especificação XSD.*
`elementFormDefault`	`unqualified`

Fonte: Autoria própria.

no qual *{endereço}* representa o endereço de rede do servidor, *{porta}* representa a porta em que o serviço estará disponível, *{recursoBase}* representa o caminho do recurso base do serviço. Estes valores são derivados do modelo SDDM. O segmento de caminho `/xsd` é um endereço relativo padrão utilizado pelo *framework* Activity-REST para a obtenção da especificação XSD.

O elemento `<xs:schema>` criado pela regra XSD deve declarar também um conjunto de *namespaces* e atributos. A Tabela 29 apresenta esses elementos.

A escolha dos identificadores locais `xs` e `aa` é uma decisão padrão de projeto. Caso outra escolha seja feita, o mesmo identificador local deve ser usado nas

demais regras de transformação.

COMMON: Incluir elementos comuns

Um conjunto de definições de tipos e elementos comuns a todos os serviços deve ser incluído no elemento `<xs:schema>`. O Apêndice H apresenta esses elementos.

PARAMS: Definir as representações de parâmetros de execução

Para cada elemento `Parameter` presente no modelo AADM, deve ser criado, internamente à definição `<xml:schema>`, um elemento `<xs:element>` contendo:

1. **Nome.** O elemento produzido deve possuir o atributo `@name` com valor igual ao nome do parâmetro;
2. **Definição de um parâmetro.** O elemento produzido deve conter uma declaração `<xs:complexType>`. Essa declaração, por sua vez, deve conter uma declaração `<xs:sequence>`. A declaração `<xs:sequence>` deve incluir, como seus subelementos, duas declarações `<xs:element>` definidas como segue:
   - a primeira declaração `<xs:element>` deve ter a forma: `<xs:element name="analysis-id" type="aa:analysis-id" maxOccurs="1" minOccurs="0" />`;
   - a segunda declaração `<xs:element>` deve ter seu atributo `@name` com valor `value`, seu atributo `@type` uma referência ao tipo XSD `aa:Value_Type`. A declaração deve apresentar atributo `@minOccurs` com valor `0`, de modo a ser possível representar um parâmetro ainda não informado. A declaração deve também apresentar o atributo `@maxOccurs` com valor igual ao da cardinalidade máxima definida pelo elemento `Parameter`. Se `Parameter` define cardinalidade máxima infinita, `@maxOccurs` deve possuir o valor `unbounded`.

DATASETS: Definir as representações de conjuntos de dados

Para cada elemento `Dataset` presente no modelo AADM, deve ser criado, internamente à definição `<xml:schema>`, um elemento `<xs:element>` contendo:

1. **Nome.** O atributo `@name` desse elemento deve receber o mesmo valor do atributo `name` do elemento `Dataset` de que foi derivado;
2. **Tipo.** O atributo `@type` desse elemento deve possuir como valor uma referência ao tipo XSD `aa:File_Type`.

H Elementos de uma definição XSD comuns às descrições de todos os serviços RAS

Listagem 30 – Elementos XSD comuns a todos os serviços RAS.

```
<xs:simpleType name="AnalysisActivityState_Type">
  <xs:restriction base="xs:string">
    <xs:enumeration value="CREATED" />
    <xs:enumeration value="READY" />
    <xs:enumeration value="RUNNING" />
    <xs:enumeration value="SUCCEEDED" />
    <xs:enumeration value="FAILED" />
    <xs:enumeration value="REMOVED" />
  </xs:restriction>
</xs:simpleType>


<xs:simpleType name="HTTPMethod_Type">
  <xs:restriction base="xs:string">
    <xs:enumeration value="GET" />
    <xs:enumeration value="POST" />
    <xs:enumeration value="PUT" />
    <xs:enumeration value="DELETE" />
  </xs:restriction>
</xs:simpleType>


<xs:simpleType name="Email_Type">
  <xs:restriction base="xs:string">
    <xs:pattern value="[^@]+@[^\.]+\..+" />
  </xs:restriction>
</xs:simpleType>


<!-- HATEOAS controls -->
<xs:complexType name="Link_Type">
  <xs:attribute
    name="rel"
    use="required"
    type="xs:string" />
  <xs:attribute
    name="href"
    use="required"
    type="xs:anyURI" />
  <xs:attribute
    name="method"
    use="optional"
    type="aa:HTTPMethod_Type" />
</xs:complexType>

```

```
46
47 <xs:element
48   name="link"
49   type="aa:Link_Type" />
50
51
52 <xs:complexType name="LinkList_Type">
53   <xs:choice maxOccurs="unbounded">
54     <xs:element ref="aa:link" />
55   </xs:choice>
56 </xs:complexType>
57
58
59 <xs:element
60   name="links"
61   type="aa:LinkList_Type" />
62
63
64 <!-- Activity Representation -->
65 <xs:simpleType name="analysis-id">
66   <xs:restriction base="xs:string">
67   </xs:restriction>
68 </xs:simpleType>
69
70
71 <xs:complexType name="AnalysisActivity_Type">
72   <xs:all>
73     <xs:element
74       ref="aa:links"
75       minOccurs="0"
76       maxOccurs="1" />
77   </xs:all>
78   <xs:attribute
79     name="id"
80     type="aa:analysis-id"
81     use="required" />
82   <xs:attribute
83     name="state"
84     type="aa:AnalysisActivityState_Type"
85     use="required" />
86 </xs:complexType>
87
88
89 <xs:element
90   name="AnalysisActivity"
91   type="aa:AnalysisActivity_Type" />
92
93
94 <xs:complexType name="JobInstance_Type">
95   <xs:all>
96     <xs:element
97       ref="aa:links"
98       minOccurs="0"
99       maxOccurs="1" />
100    <xs:element
101      name="error-report"
102      minOccurs="0"
```

```
      maxOccurs="1" />
  </xs:all>
  <xs:attribute
    name="id"
    type="aa:analysis-id"
    use="required" />
  <xs:attribute
    name="state"
    type="aa:AnalysisActivityState_Type"
    use="required" />
</xs:complexType>


<xs:element
  name="job-instance"
  type="aa:JobInstance_Type" />


<xs:simpleType name="ParameterValue_Type">
  <xs:restriction base="xs:string">
    <xs:enumeration value="STRING" />
    <xs:enumeration value="INTEGER" />
    <xs:enumeration value="REAL" />
  </xs:restriction>
</xs:simpleType>


<xs:complexType name="Value_Type">
  <xs:simpleContent>
    <xs:extension base="xs:string">
      <xs:attribute
        name="type"
        type="aa:ParameterValue_Type"
        use="optional" />
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>


<xs:element
  name="value"
  type="aa:Value_Type" />


<xs:complexType name="ValueList_Type">
  <xs:sequence>
    <xs:element
      name="id"
      type="aa:analysis-id"
      maxOccurs="1"
      minOccurs="0" />
    <xs:element
      ref="aa:value"
      maxOccurs="unbounded" />
  </xs:sequence>
</xs:complexType>

```

```
160
161 <xs:element
162   name="List"
163   type="aa:ValueList_Type" />
164
165
166 <xs:complexType name="ActivityIdBasedRequest_Type">
167   <xs:all>
168     <xs:element
169       name="id"
170       type="aa:analysis-id" />
171   </xs:all>
172 </xs:complexType>
173
174
175 <xs:element
176   name="ActivityIdBasedRequest"
177   type="aa:ActivityIdBasedRequest_Type" />
178
179
180 <xs:complexType name="File_Type">
181   <xs:all>
182     <xs:element
183       name="name"
184       type="xs:string"
185       minOccurs="0"
186       maxOccurs="1" />
187     <xs:element
188       name="content-type"
189       type="xs:string"
190       minOccurs="0"
191       maxOccurs="1" />
192     <xs:element
193       name="content"
194       type="xs:string"
195       minOccurs="1"
196       maxOccurs="1" />
197   </xs:all>
198 </xs:complexType>
199
200
201 <xs:complexType name="InputDatasetsResponse_Type">
202   <xs:all>
203     <xs:element ref="aa:links" />
204   </xs:all>
205 </xs:complexType>
206
207
208 <xs:complexType name="OutputDatasetsResponse_Type">
209   <xs:all>
210     <xs:element ref="aa:links" />
211   </xs:all>
212 </xs:complexType>
213
214
215 <xs:element
216   name="output-dataset"
```

```
217   type="OutputDatasetsResponse_Type" />
```

Fonte: Autoria própria.

I Regras de transformação para a geração de um documento WSDL

Um conjunto de regras de transformação abstratas foi definido para guiar a implementação das transformações modelo-para-texto para obter um documento WSDL que descreve a interface de um serviço RAS a partir de uma descrição ActDL e de um modelo SDDM. Essas regras são executadas de maneira hierárquica, com regras mais gerais invocando e combinando os resultados de regras mais específicas. Inicialmente, uma regra mais geral define a estrutura geral do documento WSDL. Essa regra geral invoca um conjunto de regras mais específicas para definir os elementos desse documento.

I.1 Regra de transformação que define a estrutura do documento WSDL

WSDL: Criar o documento WSDL base

A especificação base deve conter:

- uma declaração `<?xml version="1.0"encoding="UTF-8"?>` na primeira linha do arquivo;
- um único elemento `<wsdl:description>`, raiz da especificação WSDL.

Esta regra invoca as regras NAMESPACES, XSD, INTERFACES, BINDINGS e SERVICE de modo a definir os elementos presentes em `<wsdl:description>`, combinando os resultados dessas regras.

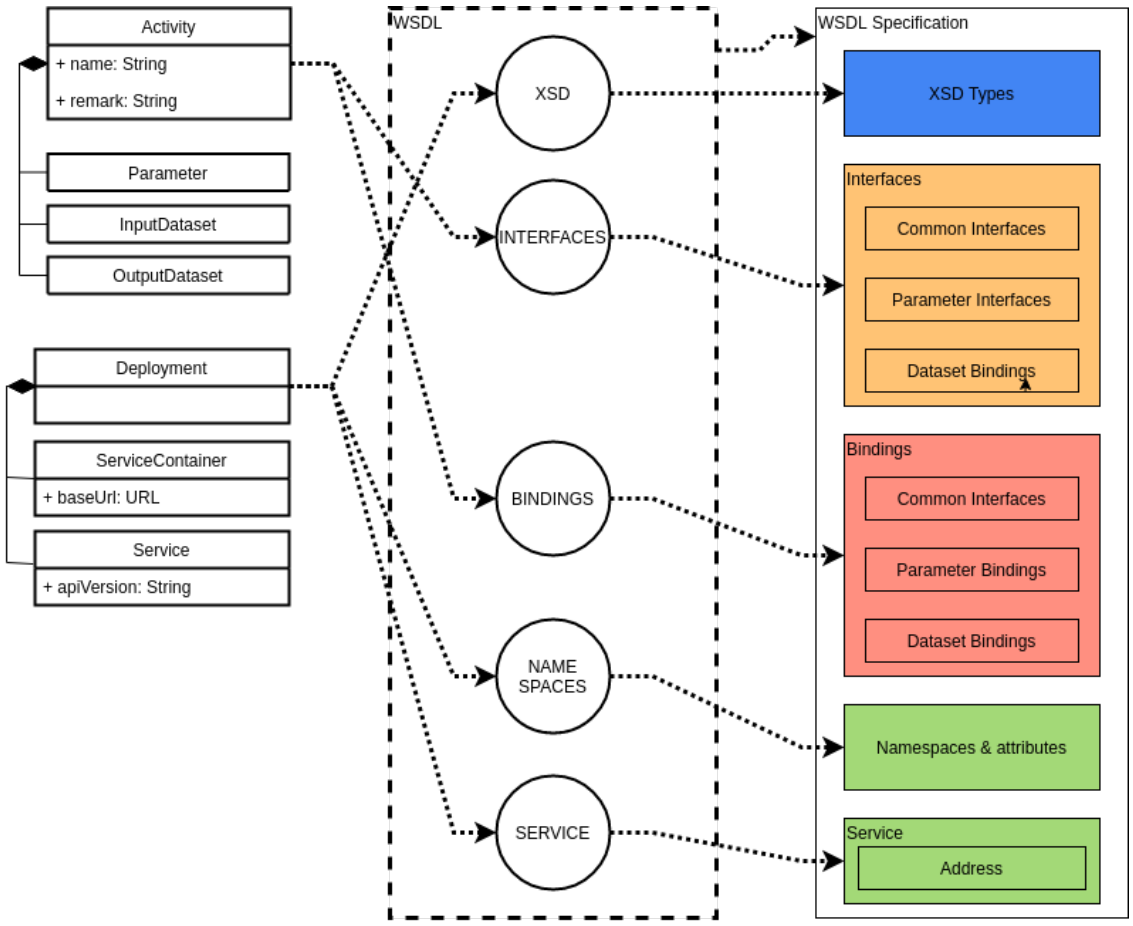
I.2 Regras de transformação que definem subelementos do elemento raiz

A Figura 68 apresenta uma visão geral das regras de transformação invocadas por WSDL. Estas regras definem atributos e elementos contidos pelo elemento `<wsdl:description>`.

NAMESPACES: Definir os *namespaces* do documento WSDL

O elemento `<wsdl:description>` deve definir um atributo `@targetNamespace`. Este atributo assume o valor `http://`*`{endereço}`*`:`*`{porta}`*`/`*`{recursoBase}`*

Figura 68 – Regras de transformação dos elementos de `<wsdl:descrption>`.



Fonte: Autoria própria. Metaclasses de interesse do modelo AADM e do modelo SDDM são apresentadas à esquerda. Elementos de uma descrição WSDL para um serviço RAS são apresentados à direita. Regras de transformação são representadas como círculos nomeados. O retângulo pontilhado ao centro representa uma regra de transformação mais geral, a qual invoca as demais regras de transformação, combinando seus resultados. Uma flecha conectando um elemento a uma regra de transformação representa que esse elemento é usado como entrada para a regra de transformação, enquanto uma flecha conectando uma regra de transformação a um elemento representa que esse elemento é criado durante a execução da regra de transformação.

`/wsdl`, no qual *`{endereço}`* representa o endereço de rede do servidor, *`{porta}`* representa a porta em que o serviço estará disponível, *`{recursoBase}`* representa o caminho do recurso base do serviço. Estes valores são derivados do modelo SDDM. O segmento de caminho `/wsdl` é um endereço relativo padrão utilizado pelo *framework* Activity-REST para a obtenção da especificação WSDL.

Adicionalmente, o elemento `<wsdl:description>` deve declarar um conjunto de *namespaces* necessários para a definição dos elementos que o compõe. A Tabela 30 apresenta os valores padrões para esses namespaces.

Tabela 30 – *Namespaces* do elemento `<wsdl:description>`.

Atributo / Namespace	Valor
`xmlns:wsdl`	`http://www.w3.org/ns/wsdl`
`xmlns:xs`	`http://www.w3.org/2001/XMLSchema`
`xmlns:whttp`	`http://www.w3.org/ns/wsdl/http`
`xmlns:wsdlx`	`http://www.w3.org/ns/wsdl-extension`
`xmlns:aa`	*Namespace alvo da especificação XSD segundo a regra NAMESPACES.*
`xmlns:tns`	*Namespace alvo da especificação WSDL.*

Fonte: Autoria própria.

XSD: Importar os tipos XSD definidos para o serviço de análise

O elemento `<wsdl:description>` deve incluir um elemento `<wsdl:types>` para definir os tipos de dados utilizados pela descrição WSDL. O elemento `<wsdl:types>`, por sua vez, deve incluir um elemento `<xs:import>` com dois atributos: `@namespace` e `@schemaLocation`. O atributo `@namespace` deve ter como valor o *namespace* alvo da definição XSD definido segundo a regra NAMESPACES. O atributo `@schemaLocation` deve ter como valor o endereço relativo padrão `./xsd`.

INTERFACES: Definir a interface do serviço

O elemento `<wsdl:description>` deve incluir um elemento `<wsdl:interface>`. Este elemento deve possuir um atributo `@name` cujo valor padrão é `interface`. Esta regra invoca as regras COMMON-INTERF, PARAMS-INTERF, INPUT-DATASET-INTERF, SINGLE-OUTPUT-DATASET-INTERF e MULTIPLE-OUTPUT-DATASET-INTERF de modo a criar os sub-elementos que compõe o elemento `<wsdl:interface>`.

BINDINGS: Definir os *bindings* do serviço

O elemento `<wsdl:description>` deve conter um elemento `<wsdl:bindings>`. A Tabela 31 apresenta a definição dos atributos deste elemento.

Esta regra invoca as regras COMMON-BINDINGS, PARAMS-BINDINGS, INPUT-DATASET-BINDINGS, SINGLE-OUTPUT-DATASET-BINDINGS

Tabela 31 – Atributos de um elemento <wsdl:bindings>.

Atributo	Valor
name	Um nome para o elemento. Valor padrão: bindings.
interface	O nome qualificado do elemento <wsdl:interface>.
type	http://www.w3.org/ns/wsdl/http
whttp:methodDefault	GET
wsdlx:safe	true

Fonte: Autoria própria.

e MULTIPLE-OUTPUT-DATASET-BINDINGS de modo a criar os subelementos que compõe o elemento <wsdl:bindings>.

SERVICE: Definir o endereçamento base do serviço

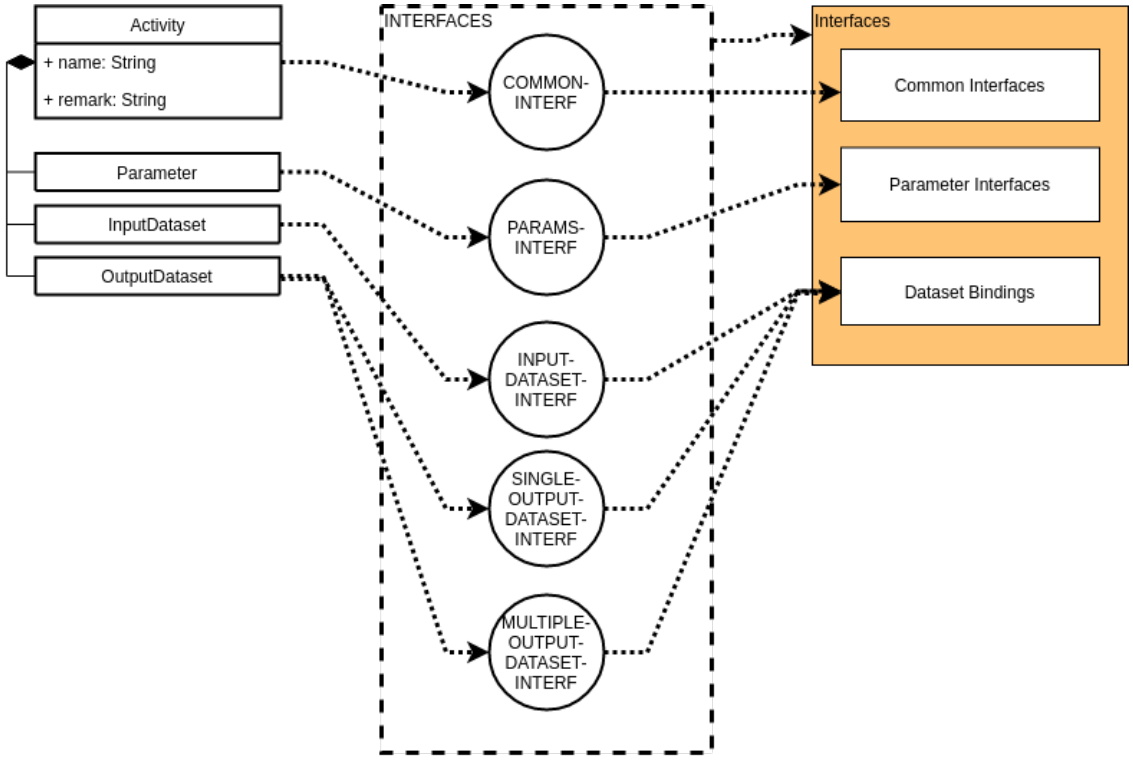
O elemento <wsdl:description> deve incluir um elemento wsdl:service. Este elemento deve possuir dois atributos: @name e @interface. O atributo @name deve ter como valor um nome para o serviço. O atributo @interface deve referenciar o elemento <wsdl:interface> definido anteriormente por meio do seu nome qualificado (tns:*valor_do_atributo_nome*).

O elemento <wsdl:service> deve incluir um elemento <wsdl:endpoint> que deve possuir três atributos: name, binding e address. O atributo name deve ter como valor um nome para o endpoint (valor padrão: RootResourceHTTPEndpoint). O atributo binding deve referenciar o elemento <wsdl:bindings> definido anteriormente por meio do seu nome qualificado (tns:*valor_do_atributo_nome*). Por fim, o atributo address deve conter o endereço do recurso raiz do serviço. Este atributo assume o valor http://*{endereço}*:*{porta}*/*{recursoBase}*, no qual *{endereço}* representa o endereço de rede do servidor, *{porta}* representa a porta em que o serviço estará disponível e *{recursoBase}* representa o caminho do recurso base do serviço. Estes valores são derivados do modelo SDDM.

I.3 Regras de transformação para definição da interface do serviço

A Figura 69 apresenta uma visão geral das regras de transformação invocadas pela regra INTERFACES. Estas regras definem elementos contidos pelo elemento `<wsdl:interface>`.

Figura 69 – Visão geral das regras de transformação para a definição da interface do serviço na descrição WSDL.



Fonte: Autoria própria. Metaclasses de interesse do modelo AADM e do modelo SDDM são apresentadas à esquerda. Elementos de uma descrição WSDL para um serviço RAS são apresentados à direita. Regras de transformação são representadas como círculos nomeados. O retângulo pontilhado ao centro representa uma regra de transformação mais geral, a qual invoca as demais regras de transformação, combinando seus resultados. Uma flecha conectando um elemento a uma regra de transformação representa que esse elemento é usado como entrada para a regra de transformação, enquanto uma flecha conectando uma regra de transformação a um elemento representa que esse elemento é criado durante a execução da regra de transformação.

COMMON-INTERF: Definir as interfaces comuns a todos os serviços RAS

Um conjunto de elementos `<wsdl:operation>`, comuns às descrições WSDL de todos os serviços RAS, deve ser incluído no elemento `<wsdl:interface>`. O Apêndice J apresenta esses elementos;

Tabela 32 – Atributos do elemento `<wsdl:operation>` que representa a operação de recuperação do valor de um parâmetro de execução.

Atributo	Valor
`name`	`Get`*`NomeDoParametro`*
`pattern`	`http://www.w3.org/ns/wsdl/in-out`
`wsdlx:safe`	`true`

Fonte: Autoria própria.

Tabela 33 – Atributos do elemento `<wsdl:operation>` que representa a operação de atualização do valor de um parâmetro de execução.

Atributo	Valor
`name`	`Put`*`NomeDoParametro`*
`pattern`	`http://www.w3.org/ns/wsdl/in-out`
`wsdlx:safe`	`true`

Fonte: Autoria própria.

PARAMS-INTERF: Definir as interfaces derivadas de parâmetros de execução

Para cada elemento `Parameter`, dois elementos elementos `<wsdl:operation>` devem ser produzidos e incluídos em `<wsdl:interface>`. O primeiro elemento `<wsdl:operation>` descreverá a operação de recuperação do valor do parâmetro. A Tabela 32 apresenta a definição dos atributos desse elemento. Este elemento deve conter também um elemento `<wsdl:input>` e um elemento `<wsdl:output>`. O elemento `<wsdl:input>` deve possuir o atributo `@element` contendo uma referência ao tipo XML `aa:ActivityIdBasedRequest`. O elemento `<wsdl:output` deve possuir o atributo `@element` contendo uma referência ao tipo XML criado na definição XSD a partir desse mesmo `Parameter`.

O segundo elemento `<wsdl:operation>` descreverá a operação de atualização do valor do parâmetro. A Tabela 33 apresenta a definição dos atributos desse elemento. O elemento `<swdl:operation>` deve conter também um elemento `<wsdl:input>`, o qual possui o atributo `@element` contendo uma referência ao tipo XML criado na definição XSD a partir desse mesmo `Parameter`.

Tabela 34 – Atributos do elemento `<wsdl:operation>` que descreve a operação de envio de um arquivo para um conjunto de dados de entrada.

Atributo	Valor
`name`	`Post`*`NomeDoConjuntoDeDados`* para um conjunto de dados com cardinalidade máxima diferente de `1`. `Put`*`NomeDoConjuntoDeDados`* para um conjunto de dados com cardinalidade máxima `1`.
`pattern`	`http://www.w3.org/ns/wsdl/in-out`
`wsdlx:safe`	`true`

Fonte: Autoria própria.

INPUT-DATASET-INTERF: Definir interfaces derivadas de conjuntos de dados de entrada

Para cada elemento `InputDataset` deve ser criado um elemento `<wsdl:operation>` que descreverá a operação de envio de um arquivo para o conjunto de dados. Este elemento deve ser incluído em `<wsdl:interface>`. A Tabela 34 apresenta a definição dos atributos desse elemento. Este elemento deve conter também um elemento `<wsdl:input>`, o qual possui o atributo `@element` contendo uma referência ao tipo XML criado na definição XSD a partir desse mesmo `InputDataset`.

SINGLE-OUTPUT-DATASET-INTERF: Definir interfaces derivadas de conjuntos de dados de saída de cardinalidade unitária

Para cada elemento `OutputDataset` com cardinalidade máxima unitária, deve ser produzido um elemento `<wsdl:operation>` que descreverá a operação de recuperação de um arquivo para o conjunto de dados de saída, o qual deve ser incluído em `<wsdl:interface>`. A Tabela 35 apresenta a definição dos atributos de `<wsdl:operation>`. O elemento `<wsdl:operation>` também deve conter um elemento `<wsdl:input>` e um elemento `<wsdl:output>`. O elemento `<wsdl:input>` deve conter um atributo `@element` referenciando ao elemento `aa:ActivityIdBasedRequest`. O elemento `<wsdl:output>` deve conter um atributo `@element` referenciando ao tipo XML criado na definição XSD a partir desse mesmo `OutputDataset`.

MULTIPLE-OUTPUT-DATASET-INTERF: Definir interfaces derivadas de conjuntos de dados de saída de cardinalidade múltipla

Tabela 35 – Atributos do elemento `<wsdl:operation>` que descreve a operação de recuperação de um arquivo de um conjunto de dados de saída com cardinalidade unitária.

Atributo	Valor
`name`	`Get`*`NomeDoConjuntoDeDados`*
`pattern`	`http://www.w3.org/ns/wsdl/in-out`
`wsdlx:safe`	`true`

Fonte: Autoria própria.

Tabela 36 – Atributos do elemento `<wsdl:operation>` que descreve a operação de recuperação dos URLs dos arquivos em um conjunto de dados de saída com cardinalidade múltipla.

Atributo	Valor
`name`	`Get`*`NomeDoConjuntoDeDados`*`Links`
`pattern`	`http://www.w3.org/ns/wsdl/in-out`
`wsdlx:safe`	`true`

Fonte: Autoria própria.

Para cada elemento `OutputDataset` que descreve um conjunto de dados de saída com cardinalidade múltipla, dois elementos elementos `<wsdl:operation>` devem ser produzidos e incluídos em `<wsdl:interface>`. O primeiro elemento `<wsdl:operation>` descreverá a operação de recuperação dos URIs de acesso aos arquivos do conjunto de dados de saída. A Tabela 36 apresenta a definição dos atributos de `<wsdl:operation>`. Este elemento deve conter também um elemento `<wsdl:input>` e um elemento `<wsdl:output>`. O elemento `<wsdl:input>` deve possuir o atributo `@element`, o qual contém uma referência ao tipo XML `aa:ActivityIdBasedRequest`. O elemento `<wsdl:output` deve possuir o atributo `@element`, o qual contém uma referência ao tipo XML criado na definição XSD a partir desse mesmo `OutputDataset`.

O segundo elemento a ser produzido descreverá a operação de recuperação de um arquivo do conjunto de dados de saída. Este elemento deve ser incluído em `<wsdl:interface>` e conter um conjunto de atributos definidos segunda a Ta-

Tabela 37 – Atributos do elemento `<wsdl:operation>` que descreve a operação de recuperação de um arquivo de um conjunto de dados de saída com cardinalidade múltipla.

Atributo	Valor
`name`	`Get`*`NomeDoConjuntoDeDados`*`File`
`pattern`	`http://www.w3.org/ns/wsdl/in-out`
`wsdlx:safe`	`true`

Fonte: Autoria própria.

bela 37. A Tabela 37 apresenta a definição dos atributos de `<wsdl:operation>`. Este elemento também deve conter um elemento `<wsdl:input>` e um elemento `<wsdl:output>`. O elemento `<wsdl:input>` deve conter um atributo `@element`, o qual referencia o elemento `aa:ActivityIdBasedRequest`. O elemento `<wsdl:output>` deve conter um atributo `@element`, o qual referencia o tipo XML criado na definição XSD a partir desse mesmo `OutputDataset`.

I.4 Regras de transformação para a definição dos *bindings* do serviço

A Figura 70 apresenta uma visão geral das regras de transformação invocadas por BINDINGS. Estas regras definem elementos contidos pelo elemento `<wsdl:bindings>`.

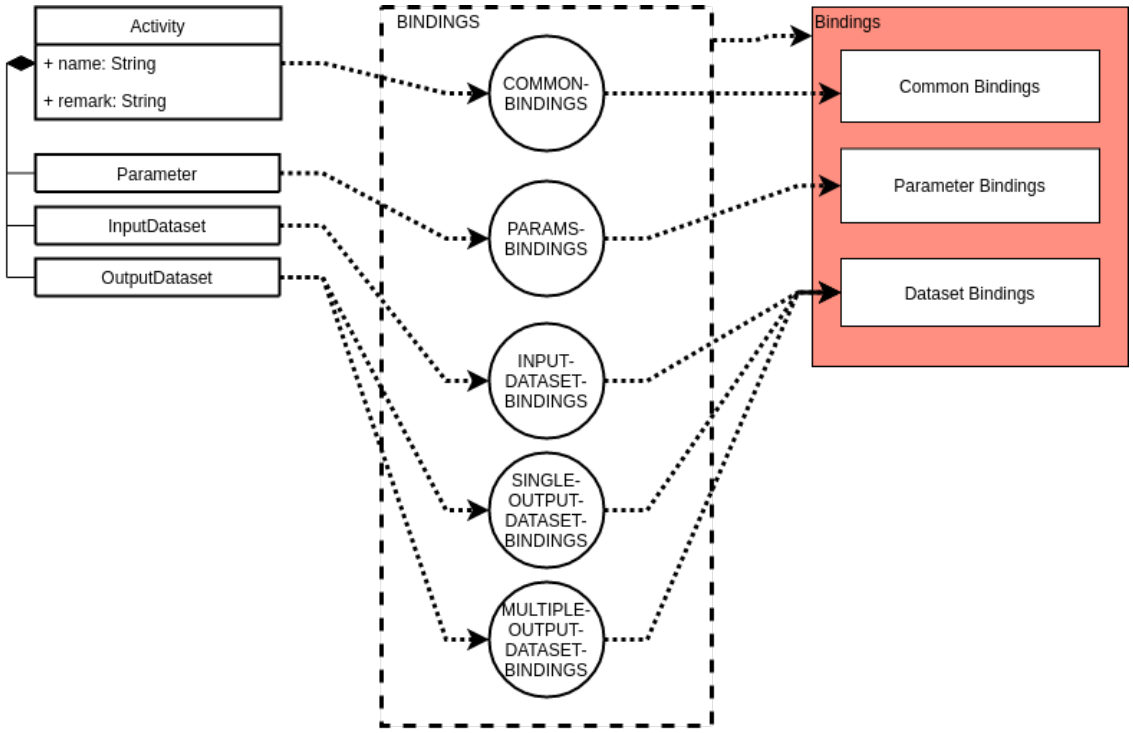
COMMON-BINDINGS: Definir *bindings* comuns a todos os serviços RAS

Um conjunto de elementos `<wsdl:operation>`, comuns às descrições WSDL de todos os serviços RAS, deve ser incluído no elemento `<wsdl:bindings>`. O Apêndice J apresenta esses elementos;

PARAMS-BINDINGS: Definir os *bindings* das operações relacionadas a parâmetros de execução

Para cada elemento `Parameter`, dois elementos elementos `<wsdl:operation>` devem ser produzidos e incluídos em `<wsdl:bindings>`. O primeiro elemento `<wsdl:operation>` descreverá a concretização da operação de recuperação do valor do parâmetro definida pela regra PARAMS-INTERF. A tabela 38 apresenta como esses atributos devem ser definidos.

Figura 70 – Visão geral das regras de transformação para a definição dos *bindings* do serviço na descrição WSDL.



Fonte: Autoria própria. Metaclasses de interesse do modelo AADM e do modelo SDDM são apresentadas à esquerda. Elementos de uma descrição WSDL para um serviço RAS são apresentados à direita. Regras de transformação são representadas como círculos nomeados. O retângulo pontilhado ao centro representa uma regra de transformação mais geral, a qual invoca as demais regras de transformação, combinando seus resultados. Uma flecha conectando um elemento a uma regra de transformação representa que esse elemento é usado como entrada para a regra de transformação, enquanto uma flecha conectando uma regra de transformação a um elemento representa que esse elemento é criado durante a execução da regra de transformação.

O segundo elemento `<wsdl:operation>` descreverá a concretização da operação de atualização do valor do parâmetro definida pela regra PARAMS-INTERF. A Tabela 39 apresenta a definição dos atributos desse elemento.

INPUT-DATASET-BINDINGS: Definir os *bindings* derivados de um conjunto de dados de entrada Para cada elemento `InputDataset` deve ser criado um elemento `<wsdl:operation>` que descreverá a concretização da operação de envio de um arquivo para o conjunto de dados. Este elemento deve ser incluído em `<wsdl:bindings>`. A Tabela 40 apresenta a definição dos atributos desse elemento.

SINGLE-OUTPUT-DATASET-BINDINGS: Definir *bindings* derivados de conjuntos de dados de saída de cardinalidade unitária

Tabela 38 – Atributos do elemento `<wsdl:operation>` que representa a operação de recuperação do valor de um parâmetro de execução.

Atributo	Valor
`ref`	O nome qualificado do elemento que descreve a operação e recuperação do valor do parâmetro criado pela regra PARAMS-INTERF
`whttp:method`	`GET`
`whttp:location`	`non-executed-instances/{id}/parameters/`*`nome-do-parametro`*
`inputSerialization`	`application/xml`
`outputSerialization`	`application/xml`

Fonte: Autoria própria.

Tabela 39 – Atributos do elemento `<wsdl:operation>` que representa a operação de atualização do valor de um parâmetro de execução.

Atributo	Valor
`ref`	O nome qualificado do elemento que descreve a operação de atualização do valor do parâmetro criado pela regra PARAMS-INTERF
`whttp:method`	`PUT`
`whttp:location`	`non-executed-instances/{id}/parameters/`*`nome-do-parametro`*
`inputSerialization`	`application/xml`
`outputSerialization`	`application/xml`

Fonte: Autoria própria.

Tabela 40 – Atributos do elemento `<wsdl:operation>` que representa a concretização da operação de envio de um arquivo para um conjunto de dados de entrada.

Atributo	Valor
`ref`	O nome qualificado do elemento criado pela regra INPUT-DATASET-INTERF
`whttp:method`	`POST` para um conjunto de dados com cardinalidade máxima diferente de `1`. `PUT` para um conjunto de dados com cardinalidade máxima `1`.
`whttp:location`	`non-executed-instances/{id}/inputs/`*`nome-do-conjunto-de-dados`*
`inputSerialization`	`application/xml`
`outputSerialization`	`application/xml`

Fonte: Autoria própria.

Para cada elemento `OutputDataset` com cardinalidade máxima unitária, deve ser produzido um elemento `<wsdl:operation>` que descreverá a concretização da operação de recuperação de um arquivo para o conjunto de dados de saída. Este elemento deve ser incluído em `<wsdl:bindings>`. A Tabela 41 apresenta a definição dos atributos de `<wsdl:operation>`.

MULTIPLE-OUTPUT-DATASET-BINDINGS: Definir *bindings* derivados de conjuntos de dados de saída de cardinalidade múltipla

Para cada elemento `OutputDataset` que descreva um conjunto de dados de saída com cardinalidade múltipla, dois elementos elementos `<wsdl:operation>` devem ser produzidos e incluídos em `<wsdl:bindings>`. O primeiro elemento `<wsdl:operation>` descreverá a concretização da operação de recuperação dos URIs de acesso aos arquivos do conjunto de dados de saída. A Tabela 42 apresenta a definição dos atributos do elemento `<wsdl:operation>`.

O segundo elemento a ser produzido descreverá a concretização da operação de recuperação de um arquivo do conjunto de dados de saída. Este elemento deve ser incluído em `<wsdl:interface>`. A Tabela 43 apresenta a definição dos atributos do elemento `<wsdl:operation>`.

Tabela 41 – Atributos do elemento `<wsdl:operation>` que descreve a concretização da operação de recuperação de um arquivo de um conjunto de dados de saída com cardinalidade unitária.

Atributo	Valor
`ref`	O nome qualificado do elemento criado pela regra SINGLE-OUTPUT-DATASET-INTERF
`whttp:method`	`GET`
`whttp:location`	`succeded-instances/{id}/outputs` *`/nome-do-conjunto-de-dados`*
`inputSerialization`	`application/xml`
`outputSerialization`	`application/xml`

Fonte: Autoria própria.

Tabela 42 – Atributos do elemento `<wsdl:operation>` que descreve a concretização da operação de recuperação dos URLs dos arquivos em um conjunto de dados de saída com cardinalidade múltipla.

Atributo	Valor
`ref`	O nome qualificado do elemento criado pela regra MULTIPLE-OUTPUT-DATASET-BINDINGS
`whttp:method`	`GET`
`whttp:location`	`succeded-instances/{id}/outputs/`*`nome-do-conjunto-de-dados`*
`inputSerialization`	`application/xml`
`outputSerialization`	`application/xml`

Tabela 43 – Atributos do elemento `<wsdl:operation>` que descreve a concretização da operação de recuperação de um arquivo de um conjunto de dados de saída com cardinalidade múltipla.

Atributo	Valor
`ref`	O nome qualificado do elemento criado pela regra MULTIPLE-OUTPUT-DATASET-INTERF
`whttp:method`	`GET`
`whttp:location`	`succeded-instances/{id}/outputs/`*`nome-do-conjunto-de-dados`*`/{filename}`
`inputSerialization`	`application/xml`
`outputSerialization`	`application/xml`

J Elementos de uma descrição WSDL comuns às descrições de todos os serviços RAS

J.1 Elementos a serem incluídos em <wsdl:interface>

Listagem 31 – Elementos WSDL presentes no elemento <wsdl:interface> de um serviço RAS.

```
<wsdl:operation name="PostNewAnalysisActivity"
  pattern="http://www.w3.org/ns/wsdl/in-out" wsdlx:safe="true">
  <wsdl:documentation>
    This operation creates a new analysis activity
    resource.
  </wsdl:documentation>
  <wsdl:output messageLabel="Out"
    element="aa:AnalysisActivity" />
</wsdl:operation>


<!-- NEW ANALYSIS RESOURCE -->
<wsdl:operation name="GetNewAnalysisActivity"
  pattern="http://www.w3.org/ns/wsdl/in-out" wsdlx:safe="true"
  style="http://www.w3.org/ns/wsdl/style/iri">
  <wsdl:documentation>
    This operation retrieves an analysis activity
    representation.
  </wsdl:documentation>
  <wsdl:input element="aa:ActivityIdBasedRequest" />
  <wsdl:output element="aa:AnalysisActivity" />
</wsdl:operation>


<wsdl:operation name="DeleteNewAnalysisActivity"
  pattern="http://www.w3.org/ns/wsdl/in-out" wsdlx:safe="false">
  <wsdl:documentation>
    This operation retrieves an analysis activity
    representation.
  </wsdl:documentation>
  <wsdl:input element="aa:ActivityIdBasedRequest" />
</wsdl:operation>


<!-- PARAMETER SET RESOURCE -->
<wsdl:operation name="GetParameterSet"
  pattern="http://www.w3.org/ns/wsdl/in-out" wsdlx:safe="true">
  <wsdl:documentation>
    This operation retrieves a map with the values
    for all
    the parameters
```

```
42      of the
43      analysis activity. There are two kinds
44      of parameters:
45      single
46      -valued an list-valued. Additionally,
47      parameters
48      can have values
49      (or
50      list-items) of a number of primitive types:
51      Strings, integers,
52      doubles.
53    </wsdl:documentation>
54    <wsdl:input element="aa:ActivityIdBasedRequest" />
55    <wsdl:output element="aa:parameters" />
56  </wsdl:operation>
57
58
59  <!-- JOB COLLECTION -->
60  <wsdl:operation name="PostStartProcessing"
61    pattern="http://www.w3.org/ns/wsdl/in-out" wsdlx:safe="true">
62    <wsdl:input element="aa:ActivityIdBasedRequest" />
63    <wsdl:output element="aa:AnalysisActivity" />
64  </wsdl:operation>
65
66
67  <wsdl:operation name="GetPoolProcessing"
68    pattern="http://www.w3.org/ns/wsdl/in-out" wsdlx:safe="true">
69    <wsdl:input element="aa:ActivityIdBasedRequest" />
70    <wsdl:output element="aa:AnalysisActivity" />
71  </wsdl:operation>
```

Fonte: Autoria própria.

J.2 Elementos a serem incluídos em <wsdl:bindings>

Listagem 32 – Elementos WSDL presentes no elemento <wsdl:bindings> de um serviço RAS.

```
1  <wsdl:operation ref="tns:PostNewAnalysisActivity"
2    whttp:method="POST" whttp:location="new-analyses"
3    whttp:inputSerialization="application/xml"
4    whttp:outputSerialization="application/xml">
5  </wsdl:operation>
6
7
8  <wsdl:operation ref="tns:GetNewAnalysisActivity"
9    whttp:method="GET" whttp:location="new-analyses/{id}"
10   whttp:inputSerialization="application/xml"
11   whttp:outputSerialization="application/xml">
12 </wsdl:operation>
13
14
15 <wsdl:operation ref="tns:DeleteNewAnalysisActivity"
16   whttp:method="DELETE" whttp:location="new-analyses/{id}"
17   whttp:inputSerialization="application/xml"
```

```
18    whttp:outputSerialization="application/xml">
19  </wsdl:operation>
20
21
22  <!-- job manager -->
23  <wsdl:operation ref="tns:PostStartProcessing"
24    whttp:method="POST" whttp:location="instances/{id}"
25    whttp:inputSerialization="application/xml"
26    whttp:outputSerialization="application/xml">
27  </wsdl:operation>
28
29
30  <wsdl:operation ref="tns:GetPoolProcessing"
31    whttp:method="GET" whttp:location="instances/{id}"
32    whttp:inputSerialization="application/xml"
33    whttp:outputSerialization="application/xml">
34
35  </wsdl:operation>
```

Fonte: Autoria própria.

K Regras de transformação para a geração de um documento OpenAPI

Um conjunto de regras de transformação foi definido para a obtenção de um documento OpenAPI a partir de uma descrição ActDL e um modelo SDDL. O documento OpenAPI produzido descreve um serviço RAS em relação a todos os *endpoints* necessários para a execução de uma instância de uma atividade de análise, bem como os tipos de dados trafegados em operações sobre esses *endpoints*. A Figura 71 apresenta uma visão geral das regras abstratas de transformação definidas para a obtenção de uma descrição OpenAPI para um serviço RAS.

API: Criar um documento OpenAPI

Um documento OpenAPI deve ser criado a partir de um par composto de uma descrição ActDL e de um modelo SDDL. Esse documento deve conter um elemento `API` contendo as definições básicas da interface do serviço RAS. A Tabela 44 apresenta a definição do valor dos atributos desse elemento.

INFO: Definir do elemento `Info`

Um elemento `Info` deve ser criado a partir de uma descrição ActDL e de um modelo SDDL. Este elemento `Info` deve definir descrições textuais gerais sobre este serviço. A Tabela 45 apresenta a definição dos valores para os atributos desse elemento.

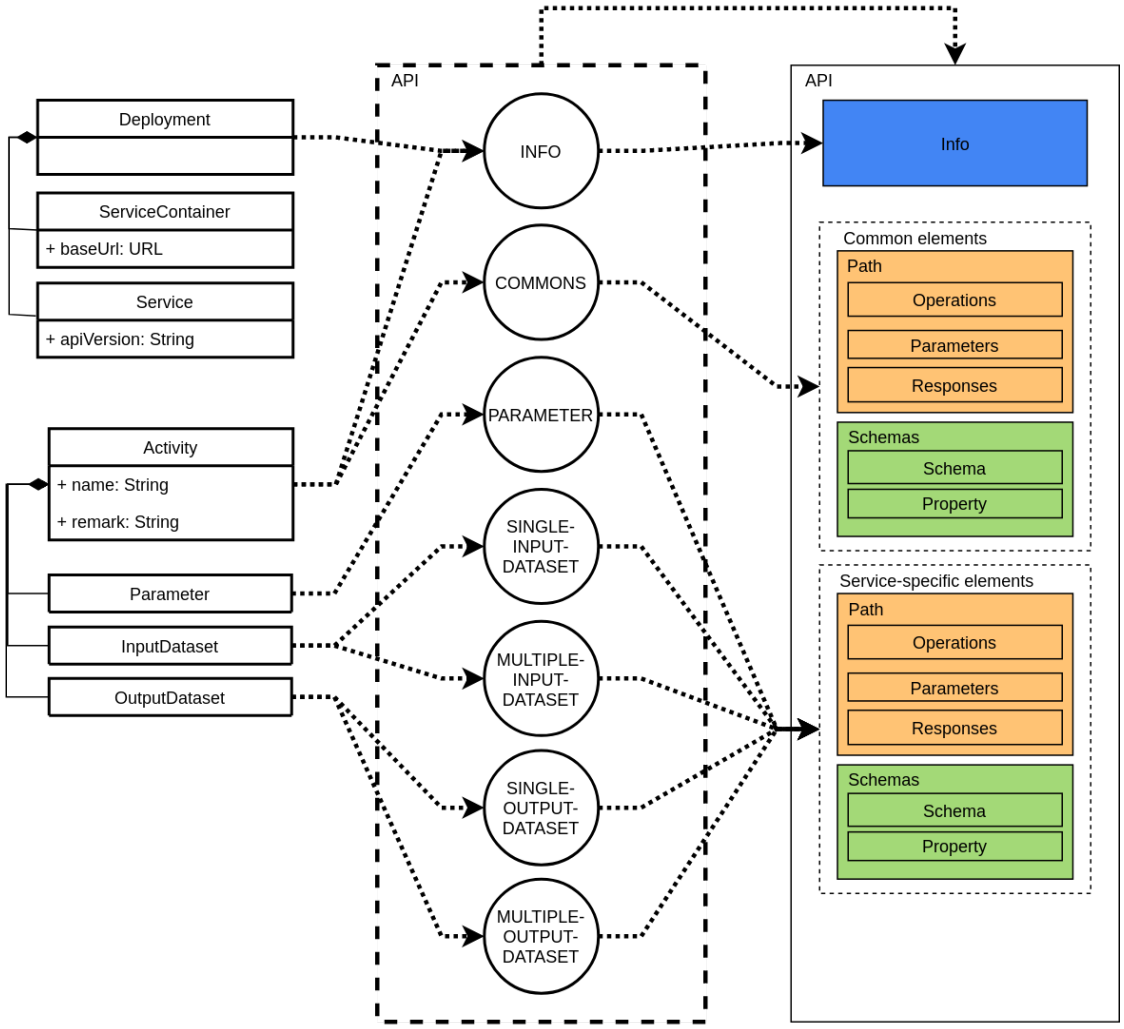
COMMONS: Definir os elementos padrões de uma descrição OpenAPI para um serviço RAS

Um conjunto de elementos padrões presentes em todas as descrições OpenAPI para um serviço RAS deve ser produzido para uma descrição ActDL. Os elementos desse conjunto padrão incluem elementos `Path`, `Parameter`, `Operation` e `Response` necessários para descrever todas os *endpoints* e operações comuns a todos os serviços RAS, bem como elementos `Schema` que descrevem todos os tipos de dados trafegados nessas operações.

PARAMETER: Definir os elementos derivados de um parâmetro de execução

Um conjunto de elementos deve ser criado a partir de um elemento `Parameter` de uma descrição ActDL. A Tabela 46 apresenta uma visão geral desses elementos.

Figura 71 – Visão geral da transformação modelo-para-modelo para a obtenção de uma descrição OpenAPI.



Fonte: Autoria própria. Metaclasses de interesse do modelo AADM e do modelo SDDM são apresentadas à esquerda. Elementos de uma especificação OpenAPI para um serviço RAS são apresentados à direita. Regras de transformação são representadas como círculos nomeados. O retângulo pontilhado ao centro representa uma regra de transformação mais geral, a qual invoca as demais regras de transformação, combinando seus resultados. Uma flecha conectando um elemento a uma regra de transformação representa que esse elemento é usado como entrada para a regra de transformação, enquanto uma flecha conectando uma regra de transformação a um elemento representa que esse elemento é criado durante a execução da regra de transformação.

SINGLE-INPUT-DATASET: Definir os elementos derivados de um conjunto de dados de entrada de cardinalidade unitária

Um conjunto de elementos deve ser criado a partir de um elemento `InputDataset` que representa um conjunto de dados de cardinalidade unitária de uma descrição ActDL. A Tabela 47 apresenta uma visão geral desses elementos.

Tabela 44 – Atributos de um elemento `API`.

Atributo	Valor
`host`	Valor do atributo `baseUrl` do elemento `ServiceContainer` do modelo SDDL. Apenas o valor do endereçamento do container do serviço e o endereço da porta utilizada, sem o *path* do recurso base do serviço.
`basePath`	Valor do atributo `baseUrl` do elemento `ServiceContainer` do modelo SDDL. Apenas o *path* absoluto do recurso base do serviço.
`schemes`	Valor: `http`.
`consumes`	Valores: `application/json` e `application/xml`.
`produces`	Valores: `application/json` e `application/xml`.
`info`	Elemento criado pela regra INFO.
`paths`	Referências a todos elementos `Path` criados pelas outras regras de transformação.
`definitions`	Referencias a todos elementos `Schema` criados pelas outras regras de transformação.

Fonte: Autoria própria.

MULTIPLE-INPUT-DATASET: Definir os elementos derivados de um conjunto de dados de entrada de cardinalidade múltipla

Um conjunto de elementos deve ser criado a partir de um elemento `InputDataset` que representa um conjunto de dados de cardinalidade múltipla de uma descrição ActDL. A Tabela 48 apresenta uma visão geral desses elementos.

Tabela 45 – Atributos de um elemento `Info`.

Atributo	Valor
`title`	Valor do atributo `name` do elemento `Service` do modelo SDDL.
`version`	Valor do atributo `apiVersion` do elemento `Service` do modelo de implantação.
`description`	Valor do atributo `description` do elemento `Service` do modelo SDDL. Alternativamente, pode ser utilizado o valor do atributo `remark` do elemento `Activity` da descrição ActDL.

Fonte: Autoria própria.

Tabela 48 – Elementos na descrição OpenAPI criados a partir de um elemento `InputDataset` que representa um conjunto de dados de cardinalidade múltipla da descrição ActDL.

Nome	Tipo	Visão geral
`nonExecutedInstancePath`	`Path`	Elemento `Path` que representa o caminho para acesso à coleção e arquivos representado por esse conjunto de dados para uma instância de atividade de análise nos estados `CREATED` e `READY`. Deve incluir também um elemento `Operation` descrevendo a operação de `GET` nessa coleção.

Continua.

Tabela 48 – Elementos na descrição OpenAPI criados a partir de um elemento `InputDataset` que representa um conjunto de dados de cardinalidade múltipla da descrição ActDL (Continuação).

Nome	Tipo	Visão geral
`nonExecutedInstanceFilePath`	`Path`	Elemento `Path` que representa o caminho para acesso a um arquivo independente da coleção desse conjunto de dados para uma instância de atividade de análise nos estados `CREATED` e `READY`. Esse `Path` deve ser parametrizado para que o nome do arquivo possa ser definido em tempo de execução. Deve incluir também elementos `Operation` descrevendo as operações de `GET`, `POST`, `PUT` e `DELETE` sobre esse *endpoint*.
`succeededInstancePath`	`Path`	Elemento `Path` que representa o caminho para acesso à coleção e arquivos representado por esse conjunto de dados para uma instância de atividade de análise no estado `SUCCEEDED`. Deve incluir também um elemento `Operation` descrevendo a operação de `GET` nessa coleção.

Continua.

Tabela 48 – Elementos na descrição OpenAPI criados a partir de um elemento `InputDataset` que representa um conjunto de dados de cardinalidade múltipla da descrição ActDL (Continuação).

Nome	Tipo	Visão geral
`succeededInstance-FilePath`	`Path`	Elemento `Path` que representa o caminho para acesso a um arquivo independente da coleção desse conjunto de dados para uma instância de atividade de análise no estado `SUCCEEDED`. Esse `Path` deve ser parametrizado para que o nome do arquivo possa ser definido em tempo de execução. Deve incluir também um elemento `Operation` descrevendo a operação de `GET` nesse arquivo.
`failedInstancePath`	`Path`	Elemento `Path` que representa o caminho para acesso à coleção e arquivos representado por esse conjunto de dados para uma instância de atividade de análise no estado `FAILED`. Deve incluir também um elemento `Operation` descrevendo a operação de `GET` nessa coleção.

Continua.

Tabela 48 – Elementos na descrição OpenAPI criados a partir de um elemento `InputDataset` que representa um conjunto de dados de cardinalidade múltipla da descrição ActDL (Continuação).

Nome	Tipo	Visão geral
`failedInstanceFilePath`	`Path`	Elemento `Path` que representa o caminho para acesso a um arquivo independente da coleção desse conjunto de dados para uma instância de atividade de análise no estado `FAILED`. Esse `Path` deve ser parametrizado para que o nome do arquivo possa ser definido em tempo de execução. Deve incluir também um elemento `Operation` descrevendo a operação de `GET` nesse arquivo.
`failedInstancePath`	`Path`	Elemento `Path` que representa o caminho para acesso a esse conjunto de dados para uma instância de atividade de análise no estado `FAILED`. Deve incluir também um elemento `Operation` descrevendo a operação de `GET`.

Fonte: Autoria própria.

SINGLE-OUTPUT-DATASET: Definir os elementos derivados de um conjunto de dados de saída de cardinalidade unitária

Um conjunto de elementos deve ser criado a partir de um elemento `OutputDataset` que representa um conjunto de dados de cardinalidade unitária de uma descrição ActDL. A Tabela 49 apresenta uma visão geral desses elementos.

MULTIPLE-OUTPUT-DATASET: Definir os elementos derivados de um conjunto de dados de saída de cardinalidade múltipla

Um conjunto de elementos deve ser criado a partir de um elemento `OutputDataset` que representa um conjunto de dados de cardinalidade múltipla de uma descrição ActDL. A Tabela 50 apresenta uma visão geral desses elementos.

Tabela 46 – Elementos na descrição OpenAPI criados a partir de um elemento `Parameter` da descrição ActDL.

Nome	Tipo	Visão geral
`property`	`Property`	Elemento `Property` para representar a propriedade relacionada a este parâmetro nas requisições e respostas do *endpoints* que lidam com a coleção de parâmetros de execução completa.
`localProperty`	`Property`	Elemento `Property` usado para representar a propriedade relacionada a este parâmetro nas requisições e resposta dos *endpoints* que lidam com esse parâmetro independentemente.
`schema`	`Schema`	Elemento `Schema` representando um objeto JSON que contém apenas uma propriedade representada pelo elemento `localProperty`.
`nonExecutedInstancePath`	`Path`	Elemento `Path` que representa o caminho para acesso a esse parâmetro de execução para uma instância de atividade de análise nos estados `CREATED` e `RUNNING`. Deve incluir também elementos `Operation` descrevendo as operações de `GET` e `PUT` sobre esse *endpoint*.
`succeededInstancePath`	`Path`	Elemento `Path` que representa o caminho para acesso a esse parâmetro de execução para uma instância de atividade de análise no estado `SUCCEEDED`. Deve incluir também um elemento `Operation` descrevendo a operação de `GET`.
`failedInstancePath`	`Path`	Elemento `Path` que representa o caminho para acesso a esse parâmetro de execução para uma instância de atividade de análise no estado `FAILED`. Deve incluir também um elemento `Operation` descrevendo a operação de `GET`.

Fonte: Autoria própria.

Tabela 47 – Elementos na descrição OpenAPI criados a partir de um elemento `InputDataset` que representa um conjunto de dados de cardinalidade unitária da descrição ActDL.

Nome	Tipo	Visão geral
`nonExecutedInstancePath`	`Path`	Elemento `Path` que representa o caminho para acesso a esse conjunto de dados para uma instância de atividade de análise nos estados `CREATED` e `RUNNING`. Deve incluir também elementos `Operation` descrevendo as operações de `GET`, `POST`, `PUT` e `DELETE` sobre esse *endpoint*.
`succeededInstancePath`	`Path`	Elemento `Path` que representa o caminho para acesso a esse conjunto de dados para uma instância de atividade de análise no estado `SUCCEEDED`. Deve incluir também um elemento `Operation` descrevendo a operação de `GET`.
`failedInstancePath`	`Path`	Elemento `Path` que representa o caminho para acesso a esse conjunto de dados para uma instância de atividade de análise no estado `FAILED`. Deve incluir também um elemento `Operation` descrevendo a operação de `GET`.

Fonte: Autoria própria.

Tabela 49 – Elementos na descrição OpenAPI criados a partir de um elemento `OutputDataset` que representa um conjunto de dados de cardinalidade unitária da descrição ActDL.

Nome	Tipo	Visão geral
`succeededInstancePath`	`Path`	Elemento `Path` que representa o caminho para acesso a esse conjunto de dados para uma instância de atividade de análise no estado `SUCCEEDED`. Deve incluir também um elemento `Operation` descrevendo a operação de `GET`.

Fonte: Autoria própria.

Tabela 50 – Elementos na descrição OpenAPI criados a partir de um elemento `OutputDataset` que representa um conjunto de dados de cardinalidade múltipla da descrição ActDL.

Nome	Tipo	Visão geral
`succeededInstancePath`	`Path`	Elemento `Path` que representa o caminho para acesso à coleção e arquivos representado por esse conjunto de dados para uma instância de atividade de análise no estado `SUCCEEDED`. Deve incluir também um elemento `Operation` descrevendo a operação de `GET` nessa coleção.
`succeededInstance-FilePath`	`Path`	Elemento `Path` que representa o caminho para acesso a um arquivo independente da coleção desse conjunto de dados para uma instância de atividade de análise no estado `SUCCEEDED`. Esse `Path` deve ser parametrizado para que o nome do arquivo possa ser definido em tempo de execução. Deve incluir também um elemento `Operation` descrevendo a operação de `GET` nesse arquivo.

Fonte: Autoria própria.