Pontifícia Universidade Católica
do Rio de Janeiro

Thiago Manhente de Carvalho Marques

Reengenharia de uma aplicação científica para inclusão de conceitos de workflow

Dissertação de Mestrado

Dissertação apresentada como requisito parcial para obtenção do título de Mestre pelo Programa de Pós-Graduação em Informática da PUC-Rio.

Orientador: Prof. Carlos José Pereira de Lucena

Rio de Janeiro

Dezembro de 2014

Pontifícia Universidade Católica
do Rio de Janeiro

Thiago Manhente de Carvalho Marques

Reengenharia de uma aplicação científica para inclusão de conceitos de workflow

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre pelo Programa de Pós-graduação em Informática do Departamento de Informática do Centro Técnico e Científico da PUC-Rio. Aprovada pela Comissão Examinadora abaixo assinada.

Prof. Carlos José Pereira de Lucena
Orientador
Departamento de Informática – PUC-Rio

Prof. Hélio Côrtes Vieira Lopes
Departamento de Informática – PUC-Rio

Prof. Alessandro Fabricio Garcia
Departamento de Informática – PUC-Rio

Prof. José Eugenio Leal
Coordenador(a) Setorial do
Centro Técnico Científico - PUC-Rio

Rio de Janeiro, 18 de dezembro de 2014

Thiago Manhente de Carvalho Marques

Bacharel em sistemas de informação pela Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio).

Ficha Catalográfica

Marques, Thiago Manhente de Carvalho

Reengenharia de uma aplicação científica para inclusão de conceitos de workflow / Thiago Manhente de Carvalho Marques ; orientador: Carlos José Pereira de Lucena. – 2014.
92 f. : il. (color.) ; 30 cm

Dissertação (mestrado)–Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2014.
Inclui bibliografia

1. Informática – Teses. 2. Workflows científicos. 3. Reengenharia de aplicações. I. Lucena, Carlos José Pereira de. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Agradecimentos

Ao meu orientador, professor Carlos José Pereira de Lucena, por aceitar me orientar neste trabalho e por todo o apoio dado durante a sua elaboração.

Aos professores Alessandro Fabricio Garcia, Hélio Côrtes Vieira Lopes e Simone Diniz Junqueira Barbosa, por aceitarem participar da comissão examinadora deste trabalho e pelo auxílio na revisão da proposta preliminar de dissertação.

A Edmundo Marassi Cianni e Gustavo Robichez de Carvalho, por me permitirem a oportunidade de cursar este mestrado em meio às minhas atividades profissionais e por todo o apoio dado durante esse período e na elaboração e revisão desta dissertação.

A todos os demais companheiros de mestrado e de trabalho, amigos e familiares que me acompanharam durante estes anos de mestrado, contribuindo direta ou indiretamente para a sua realização.

Resumo

Marques, Thiago Manhente de Carvalho. **Reengenharia de uma aplicação científica para inclusão de conceitos de workflow.** Rio de Janeiro, 2014. 92p. Dissertação de Mestrado - Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

A aplicação de técnicas de workflows na área de computação científica é bastante explorada para a condução de experimentos e construção de modelos *in silico*. Ao analisarmos alguns desafios enfrentados por uma aplicação científica na área de geociências, percebemos que workflows podem ser usados para representar os modelos gerados na aplicação e facilitar o desenvolvimento de funcionalidades que supram as necessidades identificadas. A maioria dos trabalhos e ferramentas na área de workflows científicos, porém, são voltados para uso em ambientes de computação distribuída, como serviços web e computação em grade, sendo de difícil uso ou integração dentro de aplicações científicas mais simples. Nesta dissertação, discutimos como viabilizar a composição e representação de workflows dentro de uma aplicação científica existente. Descrevemos uma arquitetura conceitual de motor de workflows voltado para o uso dentro de uma aplicação *stand-alone*. Descrevemos também um modelo de implantação em uma aplicação C++ usando redes de Petri para modelar um workflow e funções C++ para representar as tarefas. Como prova de conceito, implantamos esse modelo de workflows em uma aplicação existente e analisamos o impacto do seu uso na aplicação.

Palavras-chave

Workflows científicos; Reengenharia de aplicações

Abstract

Marques, Thiago Manhente de Carvalho. **Scientific application: reengineering to add workflow concepts.** Rio de Janeiro, 2014. 92p. Dissertação de Mestrado - Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

The use of workflow techniques in scientific computing is widely adopted in the execution of experiments and building in silico models. By analysing some challenges faced by a scientific application in the geosciences domain, we noticed that workflows could be used to represent the geological models created using the application so as to ease the development of features to meet those challenges. Most works and tools on the scientific workflows domain, however, are designed for use in distributed computing contexts like web services and grid computing, which makes them unsuitable for integration or use within simpler scientific applications. In this dissertation, we discuss how to make viable the composition and representation of workflows within an existing scientific application. We describe a conceptual architecture of a workflow engine designed to be used within a stand-alone application. We also describe an implementation model of this architecture in a C++ application using Petri nets to model a workflow and C++ functions to represent tasks. As proof of concept, we implement this workflow model in an existing application and studied its impact on the application.

Keywords

Scientific Workflow, Application Reengineering

Sumário

1 Introdução

Nesta seção, discutiremos os desafios enfrentados por uma aplicação de apoio ao processo de caracterização geológica que motivaram este trabalho e a nossa proposta de uso de fluxos de trabalho (*workflows*) para atendê-los. Primeiramente, apresentaremos de forma breve o processo de caracterização geológica. Discutiremos, então, os desafios de falta de informações de proveniência e de automação enfrentados por uma aplicação de apoio a esse processo que nos levaram à nossa proposta de entender um modelo geológico como o resultado da composição de um fluxo de trabalho científico. Por fim, apresentamos a nossa proposta de estudar a adaptação de uma aplicação científica para introduzir conceitos de fluxos de trabalho científicos.

1.1. O processo de caracterização geológica

Geocientistas trabalham para investigar e caracterizar a subsuperfície terrestre. Isso é feito através da construção de **modelos geológicos** que representam uma determinada região de interesse da subsuperfície, tal como uma jazida mineral ou uma bacia sedimentar. Esses modelos são constituídos de interpretações geológicas, predições geoestatísticas e visualizações gráficas de elementos geológicos daquela região e suas **propriedades**. O objetivo final de um modelo geológico é a avaliação da variação dos valores de propriedades geológicas de interesse econômico ou social na região sob estudo. (HOULDING, 1994)

Exemplos de propriedades geológicas são:

- O grau de um determinado metal (cobre, ferro, níquel etc.) em relação ao total de minério em uma jazida;
- O nível de contaminação do solo e da água de um rio ou aquífero em uma região de efluentes industriais ou aterro sanitário;
- Propriedades geomecânicas de rochas (tensões, pressões, elasticidade, dentre outras) para a perfuração e escavação de túneis, poços ou minas.

A figura 1 mostra um exemplo de modelo geológico estimando a contaminação do solo por chorume em uma região adjacente a um aterro sanitário.

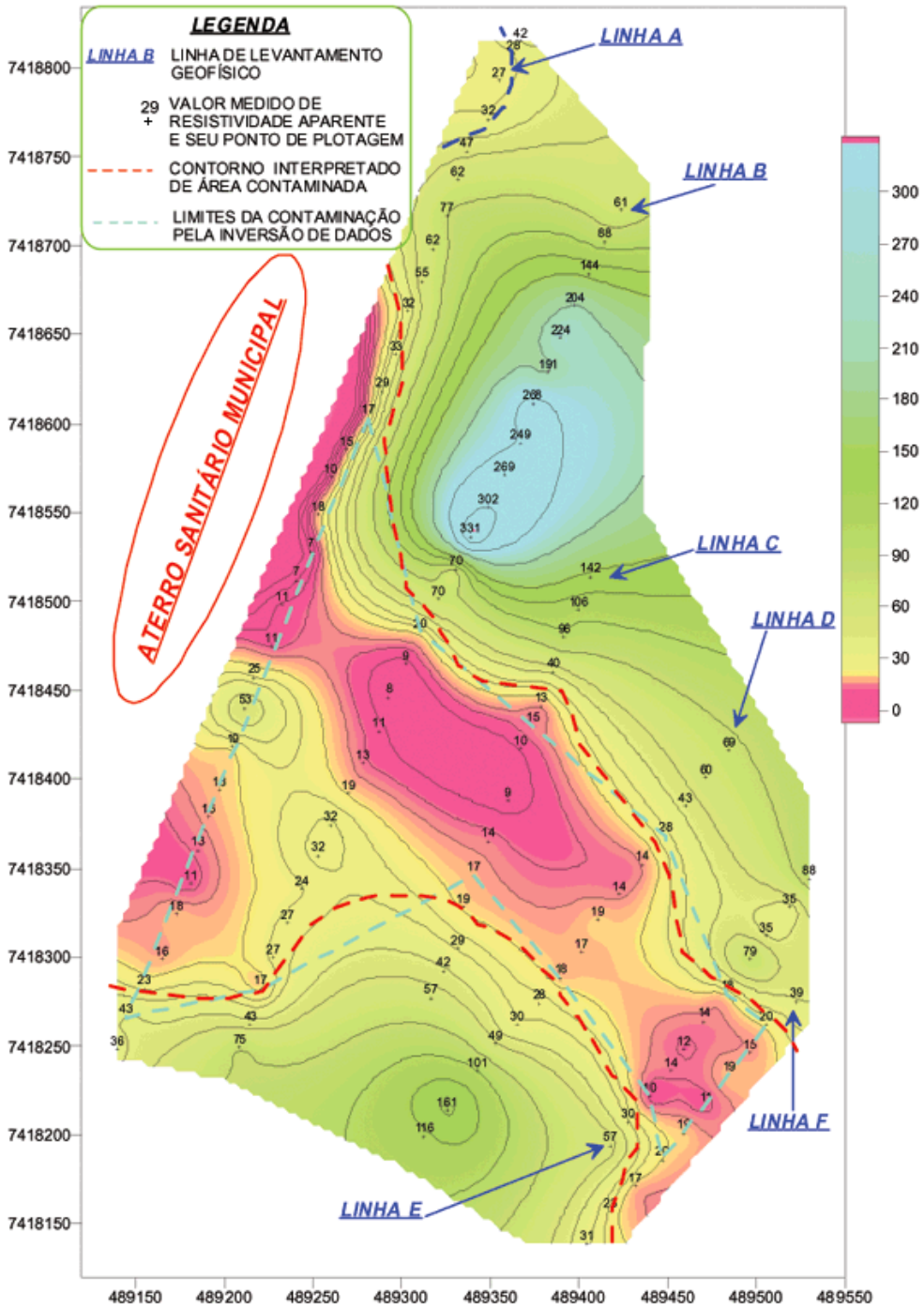

Figura 1 - Exemplo de modelo geológico: mapa da resistividade medida em terreno adjacente a um aterro sanitário para estimativa de contaminação do solo por chorume.[1]

A contaminação é estimada a partir da medição da resistividade do solo em diversos pontos da região. Usa-se a resistividade como base visto que a condução elétrica, em condições naturais, ocorre principalmente através de íons, que são abundantes no chorume.

[1] GALLAS, José Domingos Faraco et al. Contaminação por chorume e sua detecção por resistividade. **Revista Brasileira de Geofísica**, São Paulo, v. 23, nº. 1, Março 2005. Disponível em http://ref.scielo.org/74g3kb. Acessado em 25 de Janeiro de 2015.

O processo de construção desses modelos geológicos, chamado **processo de caracterização geológica** consiste, de modo geral, nas seguintes etapas (HOULDING, 1994):

- **Prospecção** da área de interesse através de perfuração de poços, coleta de amostras de rocha, sensoriamento sísmico, observações de campo, dentre outros. Da prospecção são gerados os dados brutos usados como entrada para a construção do modelo geológico.
- **Interpretação** de feições geológicas, tal como estratos rochosos, falhas e fraturas, criando uma divisão e qualificação do espaço geológico em conjuntos de volumes irregulares de acordo com as suas características geológicas específicas. Essa interpretação é baseada na análise dos dados medidos na prospecção e no conhecimento e experiência do geólogo sobre a região.
- **Predição** da variação espacial dos valores das propriedades geológicas a partir dos valores medidos na prospecção. Essa predição consiste na aplicação de métodos matemáticos e estatísticos sobre os dados da prospecção, usando como parâmetros de controle as feições interpretadas pelo geólogo.
- **Análise espacial** das feições e propriedades geológicas resultantes da interpretação e predição, para validação do modelo gerado e seu uso no processo fim relevante (extração de petróleo, planejamento da escavação de um túnel, etc.). Essa análise se baseia em análises volumétricas e procedimentos de visualização.

A figura abaixo resume esquematicamente essas etapas do processo de caracterização geológica:

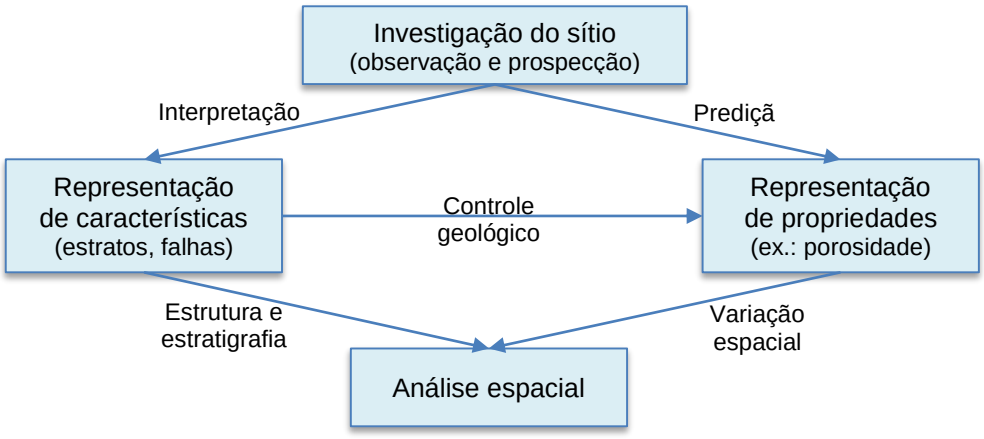

Figura 2 - Resumo do processo geral de caracterização geológica. Adaptado de (HOULDING, 1994).

1.2. Sistemas de informação aplicados à caracterização geológica

Diversos sistemas computacionais foram desenvolvidos para auxiliar no processo de caracterização geológica, a maioria sendo sistemas comerciais ou internos de companhias de mineração, petróleo etc. (HOULDING, 1994) e (TURNER, 2005) apresentam algumas das características e desafios dessas aplicações. Destacaremos no restante desta sessão as características mais relevantes para o nosso trabalho.

Sistemas de apoio à caracterização geológica proveem recursos de projeto auxiliado por computador (*CAD*, na sigla em inglês), sistemas de informações geográficas (*GIS*) e visualização 3D, além de ferramentas estatísticas e matemáticas para manipulação de dados e predição de valores de propriedades.

O modelo geológico, nesses sistemas, é representado como um conjunto de objetos geométricos (volumes, malhas triangulares, polígonos de contorno etc.) e georreferenciados[2]. Esses objetos representam feições e valores de propriedades geológicas e são gerados pelo usuário através das ferramentas de desenho do sistema, na etapa de interpretação, ou como resultado da aplicação das operações estatísticas e matemáticas providas pelo sistema, na etapa de predição.

Todas essas ferramentas são dispostas para o usuário de forma que ele possa usá-las livremente, em qualquer ordem que ele queira. A escolha de quais dessas ferramentas serão usadas, e em qual ordem, depende das características de cada região e dos dados de prospecção que o usuário tem à disposição e do tipo ou metodologia de análise que ele deseja aplicar sobre essa região.

1.3. Motivação: Desafios de uma aplicação geocientífica

Nosso trabalho foi inspirado por dois desafios que identificamos em uma aplicação de apoio ao processo de caracterização geológica desenvolvida por uma companhia de petróleo: A falta de informações de proveniência sobre um modelo geológico gerado pela aplicação e a automação limitada que ela provê para a atualização ou reprocessamento desse modelo em alguns cenários de uso.

[2] Um objeto georreferenciado é um objeto que possui uma localização definida em um espaço físico através de um sistema de coordenadas de referência. Em aplicações geocientíficas e sistemas de informações geográficas, o georreferenciamento de objetos é feito de forma definir a sua localização ao longo do globo terrestre.

1.3.1. Falta de informações de proveniência do modelo geológico

Frequentemente os usuários precisam trabalhar com modelos geológicos feitos por outras pessoas. O caso mais comum é o uso desse modelo como insumo em diferentes etapas do processo de negócio da organização, tal como por gestores para avaliar a viabilidade econômica da exploração de uma determinada região, ou por engenheiros para selecionar e projetar as estruturas e equipamentos ideais para realiza-la. Também é comum um geocientista trabalhar com modelos geológicos catalogados em acervos da organização, seja para atualizá-los ou confrontá-los com dados de novas prospecções na região representada pelo modelo, ou para usar como base para a criação de novos modelos de regiões próximas ou que tenham características geológicas similares.

Quando um novo usuário recebe um modelo existente, ele tem apenas uma visão do resultado final da análise feita pelo autor do modelo. Isso se dá porque a aplicação armazena apenas os objetos resultantes das operações feitas pelo autor do modelo, sem guardar nenhum registro ou histórico dessas operações em si. Assim, para qualquer objeto do modelo final, o novo usuário não tem como saber a partir de quais dados de entrada ele foi gerado, por quais cálculos e transformações ele passou e que parâmetros foram usados nessas operações.

Um exemplo dessa situação ocorre quando o usuário cria um objeto de malha regular no modelo a partir de um conjunto de pontos para estimar a variação espacial de uma propriedade medida nesses pontos sobre toda a região. Hoje, a aplicação disponibiliza três algoritmos diferentes de criação de malha (chamados métodos de gridagem) diferentes. Para cada método, há um conjunto de parâmetros que o usuário pode definir, como o raio de pesquisa entre celas e a definição da malha (número de linhas e colunas, tamanho das celas, orientação da malha etc.). Quando um novo usuário analisa um objeto de malha em um modelo geológico, ele não tem como saber qual dos três métodos de gridagem foi usado nem sua parametrização.

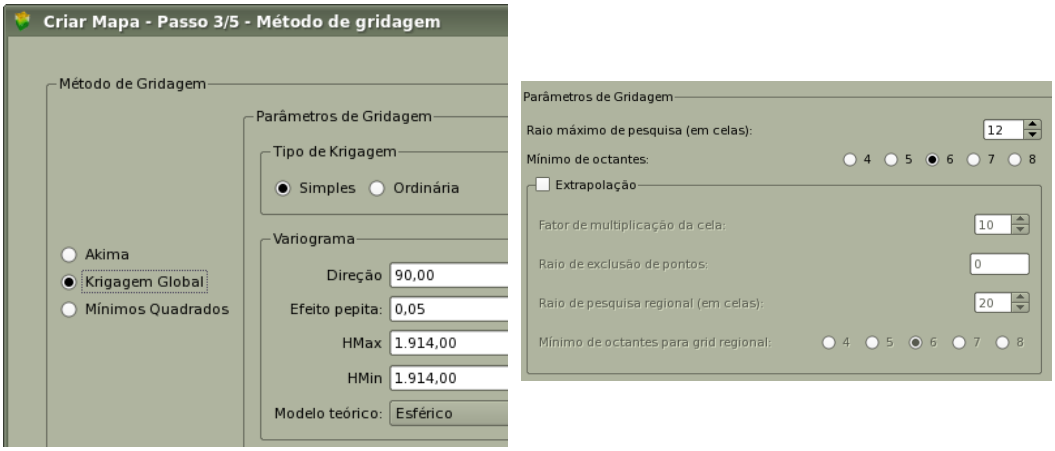

Figura 3 - Exemplos de métodos de gridagem e parâmetros disponíveis na aplicação.

A todo esse conjunto de informações sobre o histórico de como um objeto foi gerado, descrevendo todas as derivações feitas desde os dados originais de entrada, é dado o nome de **proveniência**. (SIMMHAN; PLALE; GANNON, 2005) A falta do registro dessas informações sobre o modelo geológico gerado pela aplicação dificulta e, em alguns casos, inviabiliza a validação, reprodução ou mesmo o entendimento de um modelo geológico por outros usuários e cientistas.

1.3.2. Automação limitada no reprocessamento de modelos geológicos

O fato de a aplicação não guardar nenhuma informação sobre as operações feitas pelo usuário para a criação de um modelo geológico também impacta nos cenários de uso da aplicação em que é necessário reprocessar esse modelo geológico. Isso ocorre quando há novos dados de entrada disponíveis, em geral devido a novas prospecções na região do projeto (por exemplo: novos poços perfurados), ou quando os dados de entrada usados sofrem alterações, geralmente por revisões nas etapas de interpretação do processo.

Sempre que há uma alteração nos dados de entrada de um modelo, todas as operações feitas pelo usuário a partir desses dados precisam ser refeitas para que o modelo final reflita essas alterações. Hoje, o usuário precisa disparar manualmente cada operação que ele executou originalmente no modelo, na mesma ordem e com os mesmos parâmetros, passo a passo, para atualizar todos os objetos do modelo. Esse processo, além de muito trabalhoso e demorado, é altamente propenso a erros, em especial se o usuário se esquecer de uma operação no meio do processo ou se errar na entrada de algum parâmetro.

Outro cenário de falta de automação ocorre quando o usuário deseja repetir várias vezes um mesmo conjunto de operações que representam uma determinada análise. Isso ocorre quando ele deseja fazer uma mesma análise de uma propriedade em diferentes sub-regiões do modelo (por exemplo: em diferentes intervalos de profundidade de um poço ou para diferentes tipos de rocha). Também é comum repetir uma mesma sequência de operações com pequenas variações na sua parametrização para fazer simulações dos resultados e analisar a incerteza do modelo. Em todos esses casos, ele novamente tem que refazer todas as operações manualmente, uma vez para cada região ou variação de parametrização de entrada com que ele deseje trabalhar.

1.4. Representação de modelos geológicos como *workflows* científicos

Estudando os problemas descritos acima, percebemos que o registro de um modelo geológico apenas como um conjunto de objetos geológicos, tal como é feito hoje pela aplicação estudada, não é suficiente para que o sistema possa atender as necessidades dos usuários. É necessário que uma aplicação geocientífica registre também a forma como esses objetos do modelo foram gerados.

Nesse sentido, começamos a entender o processo de caracterização geológica como um processo de composição de um fluxo de trabalho científico. Em linhas gerais, um **fluxo de trabalho científico** (*workflow* científico) é uma definição de uma sequência de tarefas que representa um processo ou experimento científico, incluindo os canais de dados e dependências entre essas tarefas, de forma a automatizar a sua computação e análise (DEELMAN et al., 2009). Cada **tarefa** do fluxo pode representar um procedimento ou atividade realizada pelo cientista, seja alguma atividade manual ou um procedimento computacional automatizado.

O conceito de fluxos de trabalho científico é bastante explorado na literatura, em especial no ramo de *e-science*[3], e aplicado em diversas áreas como astronomia e bioinformática. (DEELMAN; GANNON; SHIELDS, 2007). Em especial, como fluxos de trabalho registram explicitamente todos os passos executados pelo cientista os dados usados como entrada em cada um, o seu

[3] *e-Science*: Ciência em larga escala realizada através de colaborações distribuídas globalmente pela internet, usando enormes coleções de dados e recursos computacionais em larga escala. [definição adaptada do *National e-Science Centre* do Reino Unido, disponível em www.nesc.ac.uk/nesc/define.html. Último acesso em 06/11/2014]

uso como forma de registro de proveniência do experimento científico e como facilitador do seu compartilhamento e reprodução é amplamente explorado. (DEELMAN et al., 2009)(GARIJO; GIL; REY, 2011)

Dadas essas características, vemos que representar um modelo geológico como um resultado de um fluxo de trabalho pode auxiliar no desenvolvimento de funcionalidades em uma aplicação de apoio ao processo de caracterização, em especial para atender às necessidades de exibir informações de proveniência do modelo geológico e automatizar o seu reprocessamento nos cenários que levantamos na seção anterior. Passamos, assim, a ver um modelo geológico como o conjunto de dados usado como entrada (provenientes da prospecção) e o conjunto de fluxos de trabalho que, a partir desses dados, geraram os demais objetos geológicos do modelo final.

Esses fluxos de trabalho que compõem um modelo geológico, hoje, existem apenas implicitamente, no momento em que um geólogo aplica uma operação no modelo e usa algum outro objeto ou dado desse modelo como entrada. O fluxo, assim, permanece um conhecimento tácito do usuário. Torna-se necessário que a aplicação geocientífica passe a registrar explicitamente esses fluxos de trabalho realizados pelo usuário como um objeto de primeira classe no sistema.

1.5. Objetivo do trabalho

Partindo da ideia de representar um modelo geológico como o resultado de um fluxo de trabalho, o objetivo desta dissertação é estudar **como podemos viabilizar a composição e o registro de fluxos de trabalho em uma aplicação científica existente**, em especial no contexto de uma arquitetura de aplicação científica *stand-alone* como a que motivou este trabalho.

Nosso foco é identificar quais estruturas e componentes precisam ser inseridos em uma aplicação (tal como classes de objetos para representar um fluxo de trabalho e um motor para executá-los) para que esta possa registrar os fluxos de trabalho feitos pelo usuário. Também analisamos como os demais componentes da aplicação e o próprio método de desenvolvimento são impactados com a introdução desses componentes de fluxo de trabalho. Além disso, estudamos como esse registro de fluxos de trabalho facilita o desenvolvimento de novas funcionalidades na aplicação, em especial voltadas às questões de proveniência e automação que descrevemos nas seções anteriores.

O resultado desse estudo é exposto na forma de um modelo conceitual de um motor de execução de fluxos de trabalho voltado para a representação de fluxos de trabalho em uma aplicação científica *stand-alone*. Também descrevemos uma implementação desse modelo como uma biblioteca de classes C++, que usamos para representar fluxos de trabalho na aplicação que usamos como base neste estudo. Nesta implementação, usamos redes de Petri para representar os fluxos de trabalho e funções C++ para representar as tarefas desses fluxos.

Com este trabalho, esperamos contribuir para facilitar o uso de fluxos de trabalhos em aplicações científicas existentes. Acreditamos que o modelo conceitual e a implementação C++ de um motor de execução voltado para aplicações científicas *stand-alone* que propomos, assim como a experiência de implantação em uma aplicação existente que relatamos, podem ser usadas como base no projeto ou na reengenharia de outras aplicações científicas similares, tanto no domínio de geologia quanto de outras áreas do conhecimento. Outra contribuição importante que esperamos neste trabalho é mostrar alguns exemplos de como o uso de fluxos de trabalho nessas aplicações pode facilitar o desenvolvimento de novas funcionalidades dentro da aplicação.

1.6. Organização da dissertação

Inicialmente, estudamos alguns trabalhos existentes na área de fluxos de trabalho científicos, em especial sistemas gerenciadores de fluxos de trabalho (seção 2). Os trabalhos que encontramos, porém, são voltados para uso de fluxos de trabalho em ambientes de computação distribuída, o que acaba dificultando seu uso em uma arquitetura de aplicação mais simples, como a que estamos tomando como base.

Nós, assim, propomos um modelo conceitual de motor de execução de fluxos de trabalho que contenha apenas os elementos que consideramos essenciais para a composição e representação de fluxos de trabalho em uma aplicação científica (seção 3). Nós também propomos uma implementação desse modelo de motor de execução de fluxos em C++ usando redes de Petri para representar os fluxos e funções C++ para representar as tarefas (seção 3.3).

Uma vez que temos esses modelos conceitual e de implementação, investigamos em mais detalhes o processo de implantação de fluxos de trabalho em uma aplicação existente. Detalhamos a modificação da estrutura de um modelo geológico (seção 4.1) e a mudança no método de desenvolver a aplicação para fazerem o registro dos fluxos de trabalho do usuário (seção 4.2).

Também descrevemos adaptações necessárias em algumas operações da aplicação para representa-las como tarefas de um fluxo (seção 4.3), em especial a necessidade de refatorar algumas delas para decompô-las em funções menores (seção 4.4). Por último, mostramos os critérios que usamos para organizar e planejar a implantação gradual e iterativa de todas essas mudanças na aplicação durante o processo de desenvolvimento (seção 4.5).

Para exercitar essa abordagem, criamos protótipos de prova de conceito de composição de dois fluxos de trabalho de exemplo na aplicação que usamos como motivação desse trabalho usando essa estrutura (seção 5.1). Por último, discutimos como essas representações de *workflow* facilitam o desenvolvimento de funcionalidades que aprimorem o registro de informações de proveniência e automação de processamento do modelo geológico (seção 5.2), atendendo aos desafios que motivaram esse trabalho.

Para concluir, recapitulamos as principais contribuições do nosso trabalho (seção 6.1) e descrevemos trabalhos futuros que podem ser derivados desta dissertação (seção 6.2).

2 Trabalhos relacionados

Nesta seção, apresentaremos alguns trabalhos existentes na área de fluxos de trabalho científicos. Mostraremos alguns sistemas gerenciadores de fluxos de trabalho científicos que podem ser usados como base para desenvolver aplicações científicas ou integrados a aplicações existentes. Por fim, discutiremos a dificuldade de usar esses sistemas em uma aplicação *stand-alone*, dado que a maioria desses sistemas gerenciadores de fluxos de trabalho é voltada para ambientes de computação distribuída.

2.1. *Workflows* científicos

Os primeiros trabalhos ligados à área de fluxos de trabalho (*workflows*) surgiram no contexto de automação de processos de negócios. Com o crescimento de sistemas computacionais apoiando experimentos científicos, conceitos de fluxos de trabalho começaram a ser aplicados para representar e automatizar a execução desses experimentos *in silico*, cunhando-se então o termo fluxos de trabalho científicos (*workflows* científicos).

O ciclo de vida típico de um fluxo de trabalho científico pode ser classificado em quatro fases (DEELMAN et al., 2009), descritas abaixo e esquematizadas na figura 4:

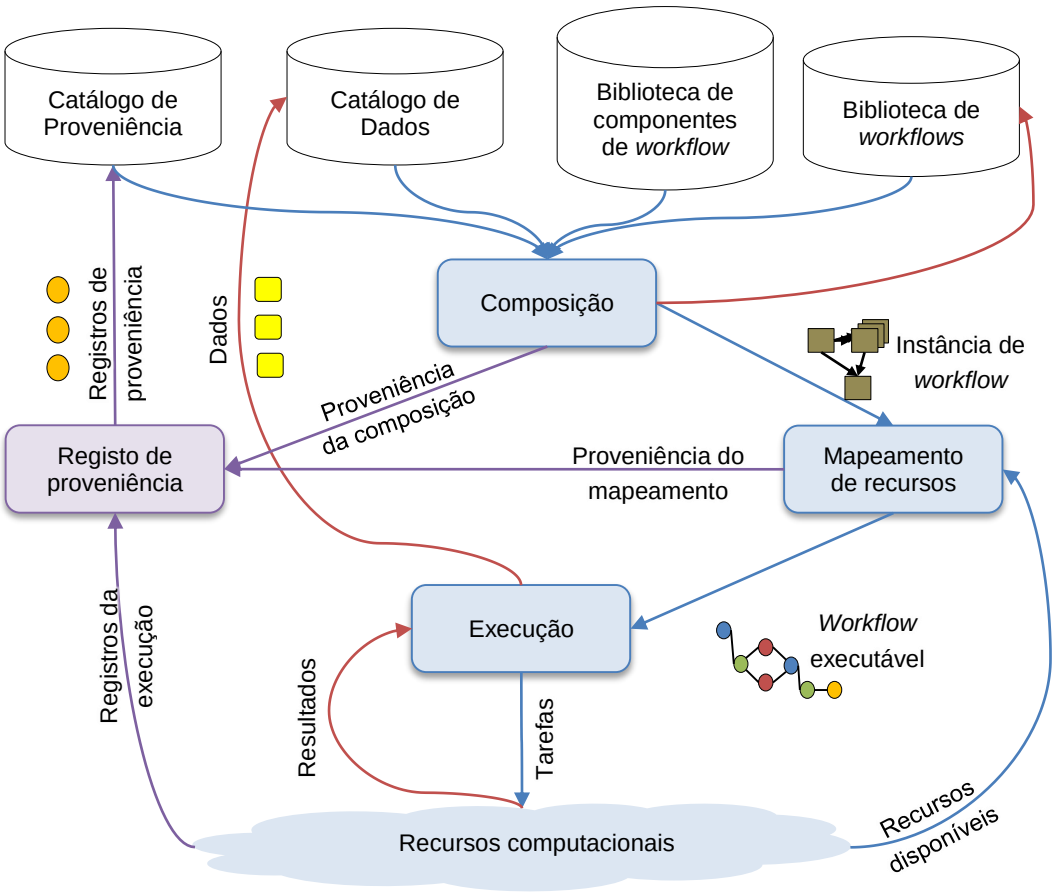

Figura 4 - Ciclo de vida de um fluxo de trabalho científico, adaptado de (DEELMAN et al., 2009).

1. **Composição**: Criação de um fluxo de trabalho para descrever uma análise ou experimento científico. Pode-se criar um fluxo completamente novo ou a partir da modificação de um fluxo já existente. O usuário compõe o fluxo utilizando itens de dados, tarefas e outros componentes disponibilizados em catálogos, bibliotecas ou repositórios de dados e de componentes de fluxos. Essa composição em geral é feita de forma textual ou gráfica.
2. **Mapeamento**: A maioria dos trabalhos de fluxos de trabalho científicos assume ambientes de execução abertos, como infraestruturas de computação em grade (*grid*) e serviços disponibilizados em redes de computadores (*web services*). Nesses casos, o fluxo definido na etapa de composição é um fluxo de trabalho abstrato[4], especificando tarefas ou serviços genéricos ou

[4] Essa nomenclatura não é padronizada. Usamos o termo fluxo de trabalho abstrato (*abstract workflow*) seguindo o trabalho de (GARIJO; GIL; REY, 2011). Por outro lado, (DEELMAN et al., 2009) usa o termo instância de fluxo de trabalho (*workflow instance*) para descrever esse tipo de fluxo.

de alto nível. Essas tarefas genéricas devem então ser mapeadas para recursos computacionais que ofereçam esses serviços e que estejam disponíveis no momento da execução. Assim, durante o mapeamento, ocorrem etapas de descoberta e negociação de serviços. O resultado do mapeamento é chamado fluxo de trabalho executável.

3. **Execução**: Uma vez tendo um fluxo executável, este é enviado para um motor de execução. O motor cuida do disparo de cada tarefa de acordo com a ordem descrita no fluxo. Nesta etapa, pode ocorrer o agendamento (*scheduling*) de tarefas nos servidores, movimentação de dados, além de diversas otimizações do fluxo. O motor monitora o estado da execução do fluxo e coletando os resultados gerados. Motores de execução também devem monitorar a ocorrência de falhas e fazer o seu tratamento.
4. **Proveniência**: Durante todas as etapas anteriores, diversas informações de proveniência devem ser coletadas para contextualizar tanto o fluxo de trabalho composto como os resultados que a sua execução geraram.

Nosso foco, neste trabalho, estará na etapa de **composição**: como expressar, em uma aplicação científica, a sequência de operações e tarefas realizadas pelo usuário e o fluxo de dados entre elas. Nós também trataremos de forma breve as etapas de **execução** e **proveniência**. Já a etapa de mapeamento, no nosso trabalho, não será necessária, visto que estamos considerando uma aplicação *stand-alone*, ou seja, um ambiente de execução fechado. Assim, todos os recursos computacionais já são conhecidos no momento da construção do sistema e permanecem fixos durante toda a sua execução. Todos os fluxos de operações gerados na nossa aplicação são, assim, sempre fluxos concretos e executáveis.

2.2. Sistemas Gerenciadores de *Workflows* Científicos

A maioria dos trabalhos sobre sistemas de fluxos de trabalho científicos que encontramos são relativos a Sistemas Gerenciadores de *Workflows* Científicos (SGWfC[5]). Um SGWfC é um sistema que apoia todos os passos do ciclo de vida de um *workflow*, desde a sua composição e execução ao registro de proveniência e dos resultados gerados. Exemplos de SGWfCs são: Kepler

[5] Em inglês: *SWfMS – Scientific Workflow Management System*.

(LUDÄSCHER et al., 2006), Taverna (WOLSTENCROFT et al., 2013) e VisTrails (CALLAHAN et al., 2006).

SGWfC são projetados para serem usados diretamente pelos cientistas para compor fluxos de trabalho graficamente, provendo uma interface voltada para facilitar o uso por pessoas com pouca ou nenhuma experiência com programação. A interface de uso de um SGWfC consiste, assim, em uma área de composição gráfica de um fluxo de trabalho e uma paleta de componentes (tarefas, conectores etc.). O cientista, então, compõe um fluxo que representa um experimento ou análise científica usando os itens da paleta. Ao final, o usuário dispara a execução do fluxo e analisa os resultados.

Para exemplificar essa estrutura dos SGWfCs, a figura abaixo mostra a interface de uso do Kepler.

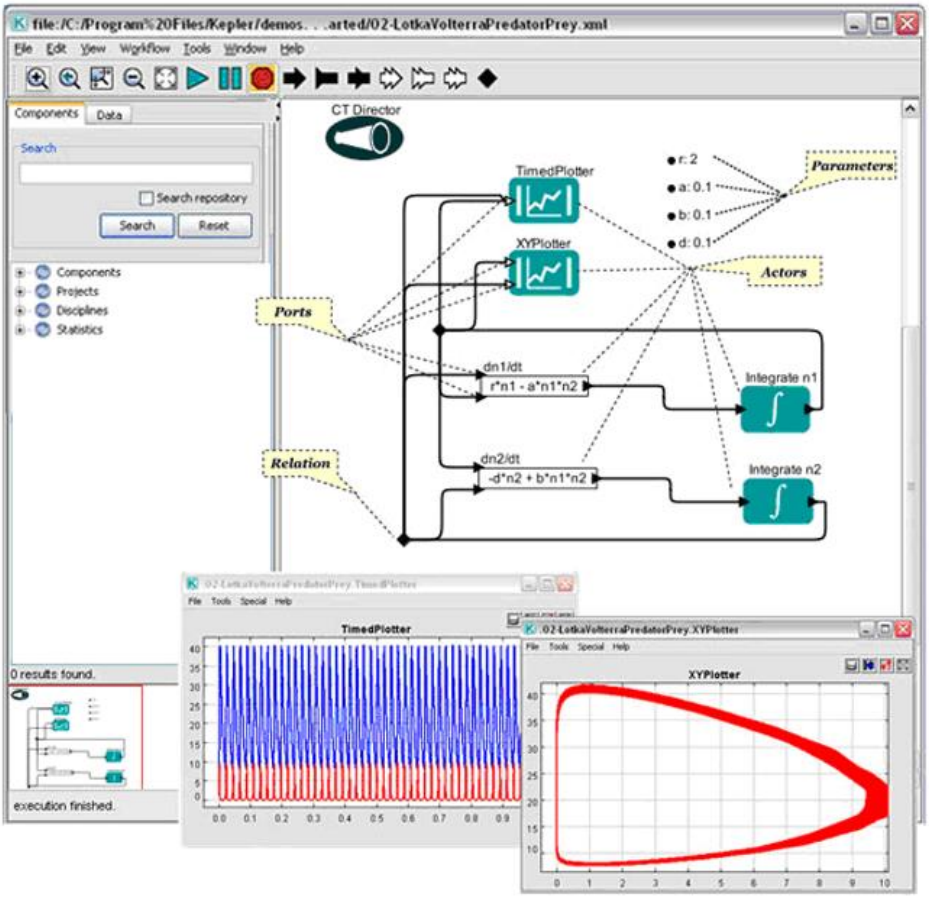

Figura 5 - Interface do Kepler mostrando um *workflow* e os seus resultados exibidos em gráficos. Retirado de https://kepler-project.org/users/sample-workflows.

Essa interface é composta de uma área central aonde o usuário compõe ou visualiza um fluxo de trabalho. A barra lateral à esquerda lista os componentes de fluxo disponíveis no sistema. O usuário pode, então, selecionar esses componentes para inseri-los no fluxo sendo composto, ligando-os aos demais componentes para estipular a sua ordem de execução. Por fim, a barra

de ferramentas possui controles para disparar e controlar a execução do fluxo de trabalho. Uma vez executado o fluxo, os resultados são exibidos em janelas e diálogos separados.

A maioria dos SGWfCs surgiu como uma ferramenta de integração de serviços distribuídos e controle de sua execução em infraestruturas de computação em grade ou serviços *web* específica dentro de um grupo ou laboratório de pesquisa. Isso se reflete na arquitetura de SGWfCs em geral assumir um ambiente de execução aberto e tem um grande enfoque na descoberta de serviços e no mapeamento de fluxos em recursos computacionais. (LIN et al., 2009; PELLEGRINI; GIACOMINI; GHISELLI, 2007) Assim, os componentes de dados e tarefas usados nos fluxos são definidos nesses sistemas como referências para serviços externos através de protocolos de comunicação como serviços *web* ou chamadas a procedimento remotas. (CURCIN; GHANEM, 2008)

2.3. Integração de um SGWfC em uma aplicação existente

Nosso principal interesse em estudar SGWfCs é analisar se algum deles poderia ser integrado em uma aplicação existente para prover as funcionalidades de composição e execução de fluxos de trabalho.

Tipicamente SGWfCs permitem que aplicações externas os usem como motores de execução de fluxos. Isso é feito disponibilizado o SGWfC em um servidor ou embutido na instalação da aplicação e acessando seus serviços via protocolos de comunicação em rede, no caso de uma instalação em servidor, ou via uma interface de linha de comando.

Aplicações externas dispararam a execução de fluxos de trabalho no motor do SGWfC através dessas interfaces, informando o fluxo a ser executado e os dados de entrada a serem usados. O SGWfC, então, dispara os serviços que representam os passos ou as tarefas do fluxo, armazenando os resultados gerados. A aplicação, por fim, recupera esses resultados do SGWfC e os exibe ao usuário.

Esse modelo de integração de SGWfC com uma aplicação existente, no contexto de uma aplicação *stand-alone* que estamos tratando nesse trabalho, impõe uma série de dificuldades que podem torna o custo de integração de um SGWfC maior do que as vantagens que essa integração proporciona.

O principal ponto de dificuldade que vemos é o registro no SGWfC dos componentes da aplicação que representam as tarefas do fluxo. Como nosso objetivo no uso de fluxos de trabalho é representar as operações que o usuário

faz hoje na aplicação, as tarefas contidas nesses fluxos representam a execução de componentes da aplicação (funções ou métodos de classes, por exemplo) que realizam os cálculos e transformações realizados pelo usuário. Todos esses componentes da aplicação precisariam ser adaptados de forma a proverem uma interface que possa ser invocada externamente pelo SGWfC.

Essa adaptação, além de trabalhosa para sistemas com um grande número de operações, pode não ser trivial dependendo da diferença entre a arquitetura da aplicação com as interfaces providas pelo SGWfC para a representação de tarefas no fluxo de trabalho.

A principal interface provida pelos SGWfC para representar tarefas de um fluxo é na forma de serviços *web* ou de computação em grade, usando protocolos como WSDL e SOAP[6], além de alguns mais específicos como BioMoby[7]. Em uma aplicação *stand-alone*, para prover acesso aos seus componentes através desses protocolos, haveria o custo de treinamento e aprendizagem dessas tecnologias pela equipe de desenvolvimento e um grande impacto na arquitetura da aplicação, por demandar a implantação de uma infraestrutura rede e de servidores, trazendo preocupações com questões de disponibilidade, desempenho da rede (latência, balanceamento de carga etc.) e segurança (na transmissão de dados sigilosos, por exemplo) que hoje não são necessárias.

A maioria dos SGWfC também suporta fluxos com tarefas que representem a execução de um comando local na máquina. Embora essas interfaces de linha de comando sejam mais simples e flexíveis para representar os componentes de uma aplicação *stand-alone*, elas geralmente impõem o uso de arquivos para representar as entradas e saídas das tarefas. Isso gera a necessidade de constantemente exportar (serializar) os objetos manipulados pela aplicação da memória para um arquivo, de forma a usá-lo como entrada em um fluxo disparado no SGWfC, e depois importar os resultados gerados na forma de arquivo de volta para a aplicação. Além de trabalhoso, isso gera um custo a mais de processamento durante a execução da aplicação que hoje não existe na aplicação, que trabalha com os objetos diretamente na memória.

[6] Conjunto de padrões de comunicação mantidos pela W3C – *World Wide Web Consortium*. Ambos são baseados na linguagem XML – *Extensive Markup Language*, e são usados para descrever e coordenar a execução de serviços disponibilizados em uma rede de computadores (*web services*).
WSDL: *Web Service Description Language*.
SOAP: *Simple Object Access Protocol*.

[7] Biblioteca de *web services* na área de bioinformática que define um conjunto adicional de ontologias para descrever tipos de dados e análises neste domínio.

Alguns SGWfC suportam fluxos com tarefas que representem um acesso a APIs em algumas linguagens de programação, geralmente Java ou R. Essa é a abordagem mais interessante para representar os componentes de uma aplicação *stand-alone*, desde que estes sejam desenvolvidos em uma das linguagens suportadas pelo SGWfC. Ainda assim, como os SGWfC são projetados de forma a trabalhar com tarefas genéricas nos fluxos de trabalho, eles acabam forçando restrições às APIs usadas que podem dificultar a integração. No Taverna, por exemplo, todos os tipos de classe usados como parâmetros de entrada ou saída de uma API Java devem ser serializáveis, o que não necessariamente acontece na aplicação em que queremos integrar o SGWfC.

Nesta dissertação, como estamos interessados especialmente em usar fluxos de trabalho no contexto de uma aplicação *stand-alone*, acreditamos que o esforço necessário para a integração de um SGWfC na aplicação seria muito grande e não representaria um ganho significativo, uma vez que usaríamos *a priori* poucas das funcionalidades proporcionadas pelo SGWfC. Isso nos levou a propor um modelo simplificado de motor de execução de fluxos de trabalho, voltado especialmente para uso em aplicações científicas *stand-alone*.

3 Arquitetura da solução

Nesta seção, descreveremos a arquitetura de um motor de fluxos de trabalho que projetamos para ser facilmente embutida em uma aplicação científica existente. Primeiramente, daremos uma visão geral dos componentes de um motor de fluxos de trabalho (3.1). Detalharemos, então, o modelo conceitual dos componentes principais da camada de domínio (3.2). Por último, mostraremos um modelo de implantação desses conceitos em uma aplicação C++ usando funções para representar as tarefas e redes de Petri para representar um fluxo de trabalho e o motor de execuções (3.3).

3.1. Visão geral

Para sermos capazes de representar e executar fluxos de trabalho dentro de uma aplicação *stand-alone*, projetamos um modelo de motor para execução que contenha apenas os elementos essenciais para a representação desses fluxos, de forma que possa ser implantada diretamente na aplicação ou na forma de uma biblioteca de classes simples e que se integre facilmente a aplicações existentes.

Os principais componentes necessários para representar e manipular fluxos de trabalho em uma aplicação são:

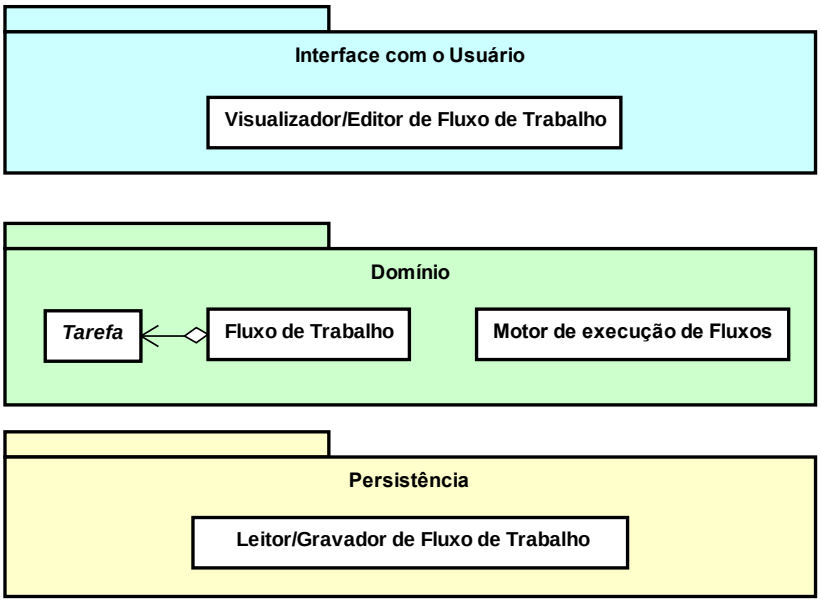

Figura 6 - Modelo conceitual dos principais componentes para representação de fluxos de trabalho, divididos em camadas.

- Uma interface de **Tarefa**, que possa ser usada pela aplicação para representar as operações executadas pelo o usuário de forma que elas possam ser compostas em um fluxo de trabalho.
- Uma classe de **Fluxo de trabalho**, que represente uma sequência de Tarefas encadeada através de ligações entre os parâmetros de saída de uma Tarefa com os parâmetros de entrada da Tarefa seguinte.
- Um ou mais **Motores de execução de fluxos** que, dado um **Fluxo de trabalho**, dispara a execução das suas tarefas seguindo a sequência determinada pelo fluxo. Mais de um Motor de execução podem ser disponibilizados com diferentes estratégias e recursos de execução de acordo com as necessidades da aplicação.
- Componentes de **Visualizador/Editor de fluxos de trabalho** para na interface com o usuário de forma que ele possa ver o fluxo (as tarefas e os dados de entrada) usado para gerar um objeto na aplicação ou para o próprio usuário poder compor ou editar fluxos de trabalho diretamente.
- Componentes de **Leitor/Gravador de fluxos de trabalho**, de forma que a aplicação possa persistir os fluxos feitos pelo usuário entre seções de uso da aplicação.

A figura 6 na página anterior ilustra a divisão desses componentes em uma arquitetura em camadas. Nosso foco, neste trabalho, será nos componentes da camada de domínio, em especial *Tarefa* e *Fluxo de trabalho*, que detalharemos nas seções seguintes.

3.2. Modelo conceitual

Descreveremos abaixo em mais detalhes o modelo conceitual dos componentes da camada de domínio – Tarefa, Fluxo de trabalho e Motor de execução de fluxos – que são o foco do nosso trabalho.

3.2.1. Tarefas

Consideramos uma **Tarefa** como um processamento feito pela aplicação em cima de um conjunto de dados de entrada (os parâmetros da tarefa) e que gera um conjunto de dados de saída (resultados). Uma tarefa, nesse sentido, assemelha-se ao conceito matemático de função, de modo que os resultados de uma operação são definidos univocamente a partir dos dados de entrada. Assim,

repetidas execuções de uma mesma tarefa usando os mesmo valores de parâmetros de entrada devem sempre gerar os mesmos resultados.

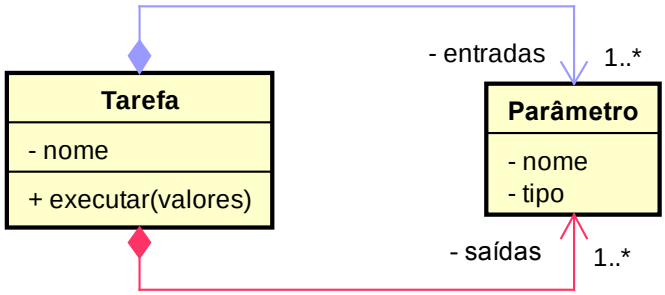

Figura 7 - Diagrama de classes conceitual de uma tarefa.

Todo dado recebido ou gerado por uma operação possui um *tipo* associado, representando a natureza desse dado. Em uma linguagem de programação fortemente tipificada ou orientada a objeto, como C++ e Java, o tipo do parâmetro equivale ao tipo ou à classe de objeto recebido como parâmetro ou gerado como saída da tarefa. Esses tipos dos parâmetros são usados para validar as ligações entre as entradas e saídas das tarefas durante a composição de um fluxo de trabalho, como veremos mais adiante.

No diagrama abaixo mostramos um exemplo de tarefa que representa a multiplicação de uma matriz por um escalar, cujos parâmetros de entrada são uma matriz (representada pela classe `Matriz`) e um número real (tipo `double`), e o parâmetro de saída é também uma matriz.

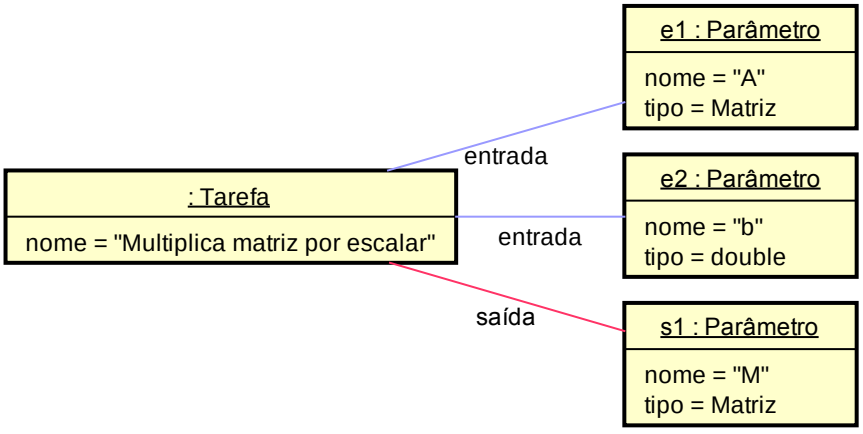

Figura 8 – Diagrama de objetos mostrando um exemplo de instância de tarefa para representar a multiplicação de uma matriz por um escalar ($M = A \cdot b$).

Toda tarefa também deve possuir uma interface que permita que o motor de execução de fluxos dispare a tarefa passando os valores dos parâmetros de entrada e recupere os valores gerados como saída. Nós representamos essa interface através do método *executar()*. Uma tarefa, assim, pode ser vista como uma instância do padrão de projeto Comando (*Command*)(GAMMA et al., 2000).

3.2.2. Fluxos de trabalho

Um **fluxo de trabalho** é uma sequência de tarefas encadeadas, onde os resultados de uma tarefa são usados como parâmetros de entrada de outra operação subsequente. A figura 9 mostra os elementos de um fluxo de trabalho (em verde) e sua relação com os componentes de tarefa (em amarelo).

Todo fluxo de trabalho também possui um conjunto de **parâmetros de entrada do fluxo**. Eles correspondem aos parâmetros de entrada das tarefas do fluxo que não estão conectados a nenhum parâmetro de saída de outra tarefa. Toda vez que um fluxo for executado, os valores desses parâmetros devem ser informados.

Uma vez especificados os parâmetros de entrada de um fluxo e as suas tarefas, um fluxo de trabalho deve definir a **parametrização** de cada uma dessas tarefas. Em outras palavras: Para cada parâmetro de entrada de cada tarefa do fluxo, deve-se definir a *origem* dos valores que serão usados para preenchê-lo. Na maioria dos casos, esses valores são provenientes do resultado de outra tarefa do fluxo (ou seja, de um *resultado intermediário* da execução do fluxo). Já no caso das tarefas iniciais do fluxo, os seus parâmetros são preenchidos a partir dos valores dos *parâmetros de entrada do fluxo* descritos no parágrafo anterior. Em alguns cenários de uso, pode ser interessante também que a própria definição do fluxo de trabalho contenha a definição de alguns *valores fixos* (constantes) para uso em algumas tarefas.

As conexões entre os parâmetros de entrada e saída das tarefas no fluxo definidas na parametrização das tarefas devem levar em conta a **compatibilidade entre os tipos dos parâmetros**. Em outras palavras: O tipo do resultado de uma operação deve ser compatível com o tipo esperado pelo parâmetro de entrada da operação seguinte em que o resultado será usado. A definição de compatibilidade entre tipos pode variar de acordo com a necessidade de cada aplicação. Em uma aplicação desenvolvida usando uma abordagem orientada a objeto, por exemplo, a compatibilidade entre tipos significa verificar se a classe do objeto gerado por uma tarefa é uma subclasse da classe esperada como parâmetro de entrada da tarefa seguinte.

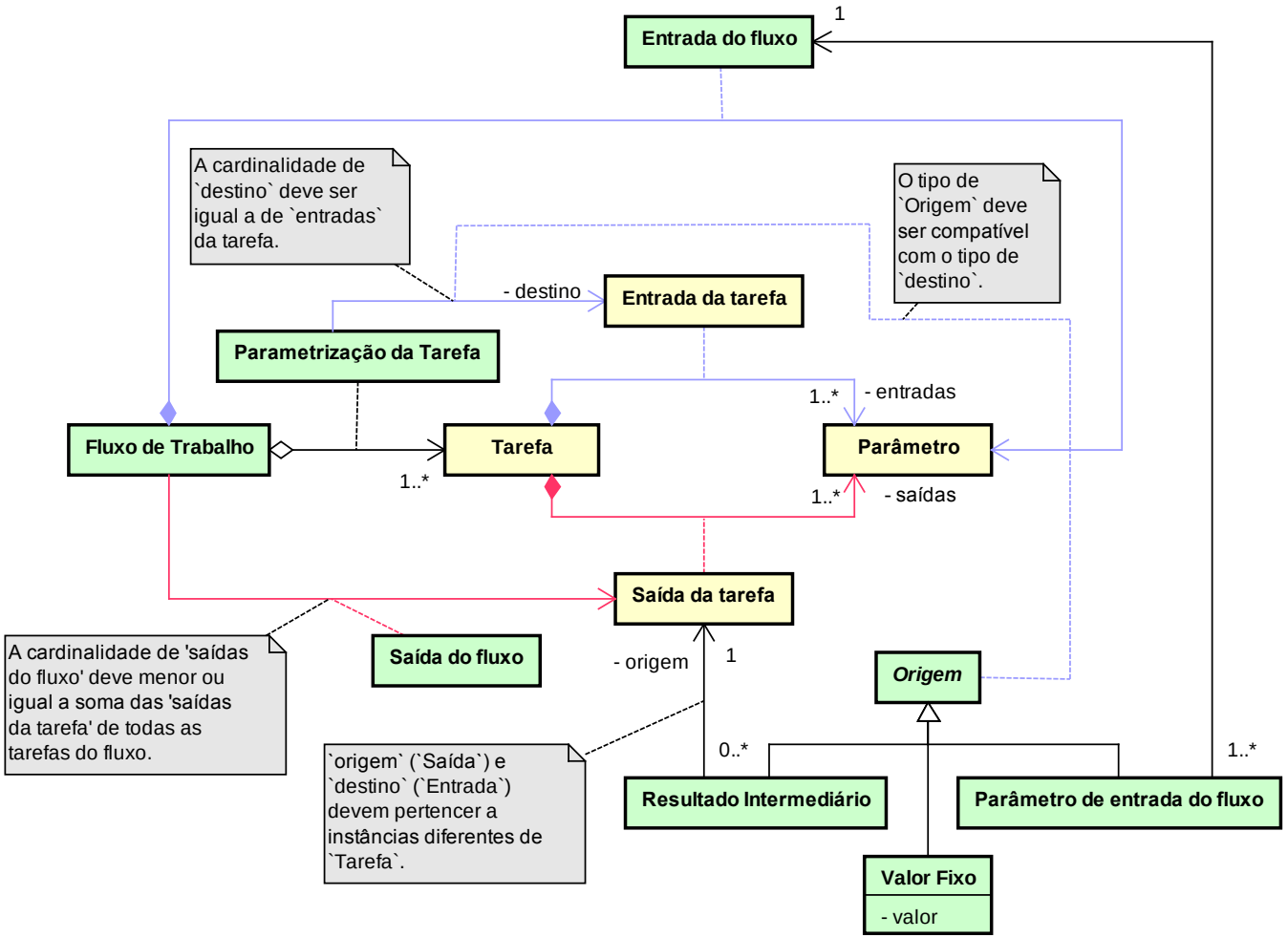

Figura 9 - Diagrama de classes conceitual de um fluxo de trabalho.

Por fim, todo fluxo de trabalho também possui um conjunto de **parâmetros de saída do fluxo**. Eles correspondem aos parâmetros de saída das tarefas do fluxo que tem um significado para o usuário. Em um caso extremo, todo parâmetro de saída de todas as tarefas do fluxo podem ser considerados significativos. Em geral, porém, é interessante identificar quais resultados de um fluxo são mais relevantes para a análise do usuário e quais são usados apenas como resultados intermediários do fluxo, por exemplo. Uma heurística simples que pode ser usada é considerar todos os parâmetros de saída de tarefas que não são usados como parâmetros de entrada de nenhuma outra tarefa do fluxo como parâmetros de saída do fluxo, e todos os demais como parâmetros intermediários.

3.2.3. Motor de execução de fluxos

A principal função de um **motor de execução de fluxos** é prover uma interface para que a execução de um fluxo de trabalho seja disparada. Essa interface deve receber os valores dos parâmetros de entrada do fluxo a serem usados naquela execução, assim como deve retornar os valores gerados pela execução das tarefas do fluxo. O motor de execução deve, então, disparar as tarefas do fluxo conforme a sequência especificada pela ligação de parâmetros entre elas.

Uma forma simples de representar um motor de execução de fluxos é como um método *executa()* na própria interface da classe de fluxo de trabalho, de forma similar à interface de tarefa. Essa abordagem é indicada quando não há a necessidade de uma lógica mais rebuscada de execução das tarefas. A figura abaixo ilustra a diferença entre uma implementação de motor de execução de fluxos diretamente na interface de um fluxo de trabalho (A) ou como um componente separado (B).

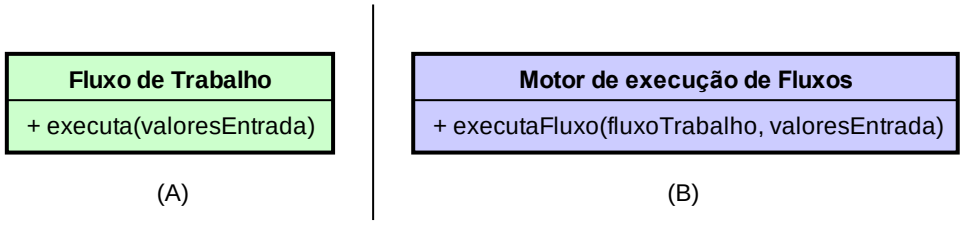

Figura 10 – Diagrama de classes conceitual de um motor de execução de fluxos modelado como um método da classe de fluxo (A) ou como uma classe separada (B).

Um importante aspecto de um motor de execução é a tolerância a falhas e tratamento de erros na execução de um fluxo de trabalho. Como estamos trabalhando com tarefas que representam procedimentos da aplicação, em geral

erros na execução das tarefas são sinalizados via exceções, o que significa que o motor de execução de fluxos deve capturar as exceções levantadas, interromper a execução do fluxo e reporta-las de forma adequada ao restante da aplicação.

Motores de execução também podem prover outras funcionalidades de acordo com as necessidades de cada aplicação, tal como mecanismos de acompanhamento da execução do fluxo (geralmente usando o padrão Observador) ou a opção de suspender e posteriormente retomar a execução de fluxos muito longos. Múltiplos motores de execução também podem ser disponibilizados, cada um fornecendo um conjunto de recursos, de forma que a aplicação possa usar aquele que for mais adequado para cada situação, tal como um motor mais leve para a execução de um fluxo mais simples e um motor mais completo para a execução de fluxos mais rebuscados.

3.2.4. Execuções de fluxos de trabalho

Uma vez que um fluxo de trabalho é executado em um motor de execução, os resultados especificados pelas saídas do fluxo são retornados para serem manipulados pela aplicação. Na maioria dos casos, porém, além desses resultados, a própria execução do fluxo deve ser registrada e armazenada pelo próprio motor de execução ou pela aplicação. Esse registro é usado para prover informações de proveniência dos resultados gerados pela execução.

O registro de uma execução de fluxo, de forma simples, é o registro do fluxo de trabalho executado, dos valores usados como parâmetros de entrada do fluxo (*parametrização do fluxo*) e os *valores de saída* gerados ao final da execução do fluxo. Dependendo das necessidades da aplicação, outras informações adicionais podem ser registradas, como o ambiente onde a execução foi realizada (equipamentos, versões de *software* etc.), o usuário responsável pelo acompanhamento da execução, a data e hora da execução, dentre outros.

A figura 11 mostra os elementos que representam uma execução de um fluxo de trabalho (em azul) e a relação destes com os componentes de fluxo de trabalho (figura 9) e tarefa (figura 7) mostrados anteriormente.

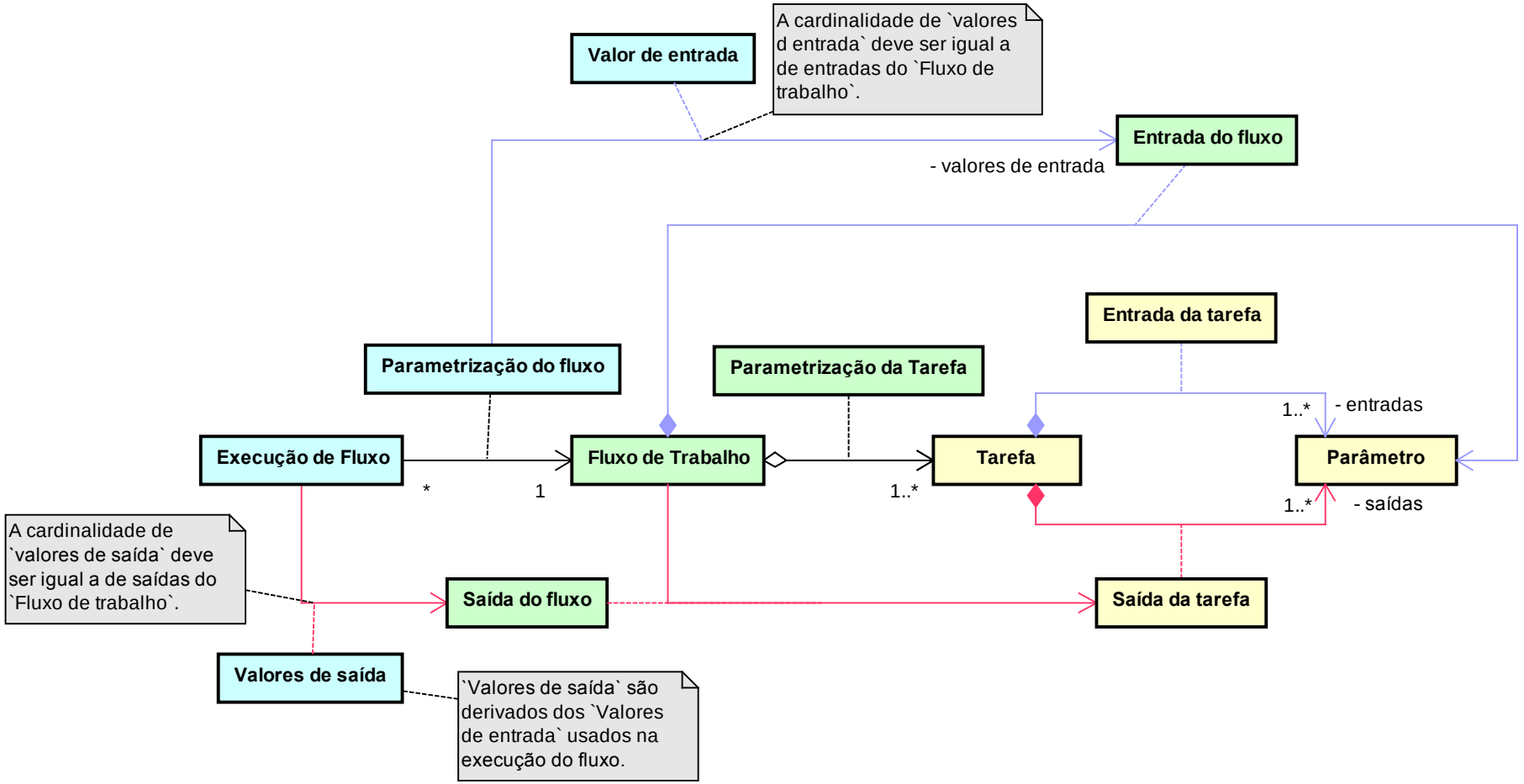

Figura 11 - Diagrama de classes conceitual do registro de uma execução de fluxo de trabalho.

3.3. Modelo de implantação

A partir desse modelo conceitual de motor de fluxos de trabalho, estudamos uma forma de implantar dos componentes do motor de fluxos em uma aplicação C++. Nós estruturamos, assim, um modelo de implantação usando funções C++ para representar as tarefas de um fluxo e redes de Petri para representar os fluxos de trabalho em si.

3.3.1. Tarefas como funções C++

Em uma aplicação C++, a forma mais natural de representar uma tarefa é como uma **função**, pois este recurso da linguagem já contém a maioria dos elementos de uma tarefa: a definição dos parâmetros de entrada e saída e seus tipos, além de uma semântica de invocação e execução bem definida.

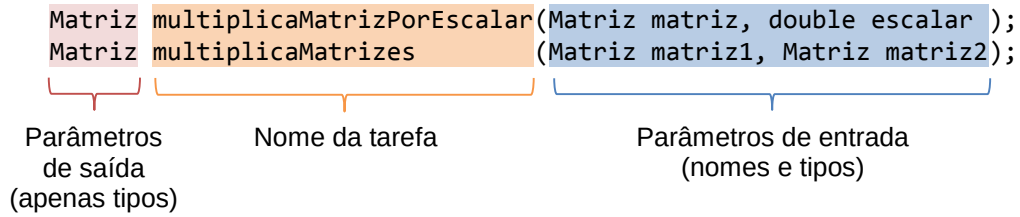

Figura 12 – Exemplos de assinaturas de funções em C++.

O uso de funções para representar as tarefas em uma aplicação C++ existente, além de ser familiar para os desenvolvedores dessa aplicação, permite reaproveitar a estrutura das operações da aplicação que serão usadas como tarefas nos fluxo de trabalho e que hoje já são representadas no código como funções.

É importante que uma função C++ usada como uma tarefa seja de fato uma função no sentido matemático da palavra (ou uma **função pura**, no jargão usado na área de programação). Isso não é forçado pela linguagem, e exige que o programador tenha o cuidado de não acessar nenhum estado mutável da aplicação (variáveis globais, por exemplo) que possa ser alterado entre duas chamadas a essa função e de não causar nenhum efeito colateral (*side effect*) no restante da aplicação durante a sua execução. Isso é necessário para garantirmos que diferentes execuções de um fluxo de trabalho com os mesmos parâmetros de entrada gerem sempre os mesmos resultados.

Para usarmos funções C++ em um fluxo de trabalho, precisamos de uma interface genérica que permita a composição e invocação de funções de diferentes números e tipos de parâmetros. Para isso, criamos uma classe de interface de tarefa, similar ao modelo conceitual da figura 7, e uma classe

adaptadora[8] de uma função C++ para esta interface, na forma de um gabarito de classe (*class template*). A estrutura dessas classes é mostrada na figura abaixo:

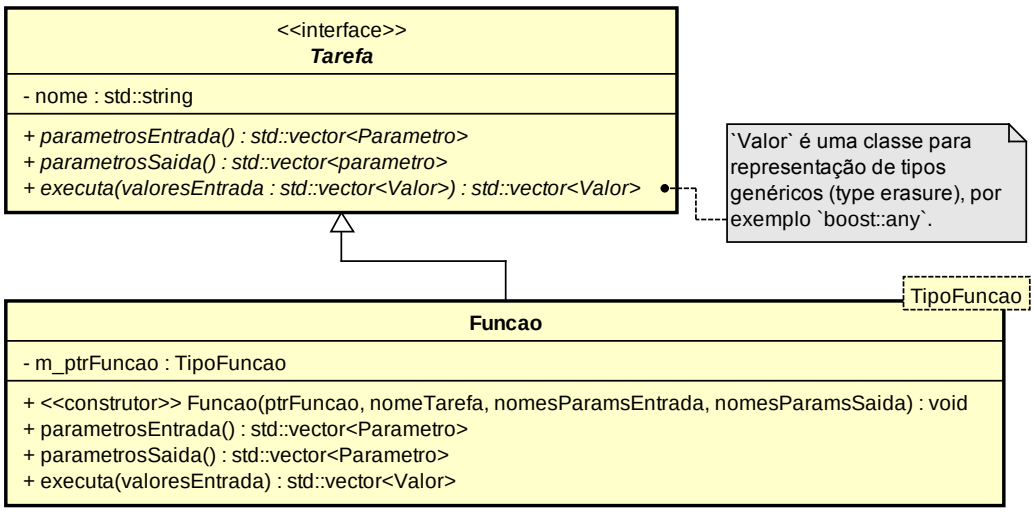

Figura 13 - Classes `Tarefa` e `Funcao`.

As principais adaptações feitas por essa classe adaptadora são:

- Popular as informações sobre os parâmetros de entrada e saída da tarefa (métodos `parametroEntrada()` e `parametroSaida()`) a partir dos tipos de parâmetros e retorno da assinatura da função C++.
- Fazer a chamada à função C++ quando a tarefa for disparada (método `executa()`), extraindo os valores dos parâmetros de entrada recebidos para repassá-los como parâmetros da função e empacotando o valor retornado pela função na forma de parâmetros de saída da tarefa para repassá-los às tarefas seguintes.

A figura abaixo mostra um exemplo de uso dessa classe adaptadora para uma função que represente uma multiplicação de uma matriz por escalar como a descrita na seção 3.2.1.

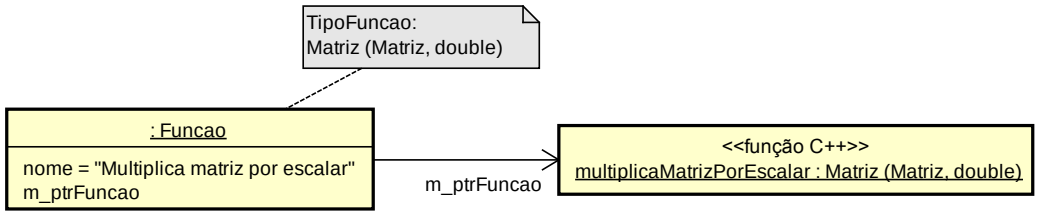

Figura 14 - Diagrama de objetos de exemplo de uma instância do adaptador `Funcao` para uma função de multiplicação de matriz por escalar.

[8] Padrão de projeto Adaptador (*Adapter*)(GAMMA et al., 2000)

Como o número e tipos de parâmetros, tanto de entrada como de saída, de uma tarefa são variáveis, usamos uma coleção (`std::vector`) de objetos genéricos. Quando uma tarefa é disparada e recebe os valores passados como parâmetros de entrada na forma desses objetos genéricos, é feita, então, uma conversão de tipo (*type cast*) para o tipo esperado daquele parâmetro. Em linguagens de programação orientadas a objeto que possuem uma classe-base da todas as demais classes, tal como `Object` em Java, essas coleções podem ser definidas em termo dessa classe-base (por exemplo: `java.util.Collection<Object>`). Já em C++, precisamos usar uma classe de *type erasure*[9], tal como `boost::any`[10].

Para representar múltiplos retornos de uma tarefa em uma linguagem em que funções podem ter apenas um retorno, como C++, precisamos usar tipos de objetos especiais para representar múltiplos parâmetros de saída. No nosso caso, escolhemos as classes `std::pair` e `std::tuple` da biblioteca padrão C++. O adaptador `Funcao`, quando instanciado para uma função, analisa o tipo retornado por ela e, sendo um desses tipos, faz a separação dos elementos do par ou da tupla[11] retornados pela função antes de empacota-los no `std::vector<Valor>` retornado pelo método `executa()`.

Essa estrutura de adaptador como gabaritos parametrizados com a assinatura da função C++ diminui o trabalho do desenvolvedor para usar uma operação da aplicação em um fluxo de trabalho, uma vez que a estrutura do adaptador é gerada pelo compilador através do gabarito. Esse adaptador também permite que o desenvolvedor especifique os nomes da operação e dos parâmetros, já que C++ não possui mecanismos de reflexão que nos permita extraí-los da própria assinatura da função (e, mesmo que houvesse esses mecanismos, os nomes usados na assinatura da função possivelmente não seriam adequados para exibição pra o usuário).

---

[9] *Type erasure* consiste em técnicas ou processos realizados em tempo de compilação em linguagens fortemente tipificadas para remover especificações de tipos de um objeto. O principal uso de *type erasure* em C++ consiste em permitir a manipulação de objetos de tipos arbitrários (genéricos), uma vez que C++ não possui uma interface geral (tal como a classe `Object` em Java) que permita nativamente a manipulação de objetos de diferentes tipos de uma forma genérica.

[10] http://www.boost.org/doc/libs/release/libs/any/, baseada em (HENNEY, 2000).

[11] Uma **tupla** (também chamado *enupla*) é uma sequência ordenada de *n* elementos. Em linguagens de programação, tuplas são usadas para representar coleções de tamanho fixo compostas por valores ou objetos de tipos heterogêneos. A biblioteca padrão de C++ provê a classe `std::tuple` para representar uma tupla.

3.3.2. Fluxos de trabalho como redes de Petri

Uma forma bastante usada de representar fluxos de trabalho é através de **redes de Petri**. Uma rede de Petri, em linhas gerais, é um formalismo matemático usado principalmente para modelar sistemas distribuídos. Uma rede de Petri consiste em um grafo orientado bipartido com dois tipos de vértice: *transições* e *posições*, que representam, respectivamente, os eventos ou ações que podem ocorrer em um sistema e as condições para a sua ocorrência ou disparo.

As arestas que ligam posições e transições são chamadas arcos. Os arcos que ligam uma posição a uma transição são chamados arcos de entrada da transição. De forma análoga, arcos que ligam transição a uma posição são chamados arcos de saída da transição. As posições nas extremidades desses arcos, assim, são chamadas posições de entrada e saída, respectivamente, da transição.

Cada posição da rede pode conter um conjunto de marcadores, que representam o estado atual do sistema. Quando todas as posições de entrada de uma transição possuem ao menos um marcador, aquela transição é dita habilitada.

A figura 15 mostra um exemplo de rede de Petri. Para representar graficamente uma rede de Petri, usa-se *círculos* para representar as posições, *barras* para representar as transições e *setas* para representar os arcos. Os marcadores são representados por *pontos*, que na figura destacamos em laranja. Nessa rede de exemplo, as transições (A) e (B) estão habilitadas.

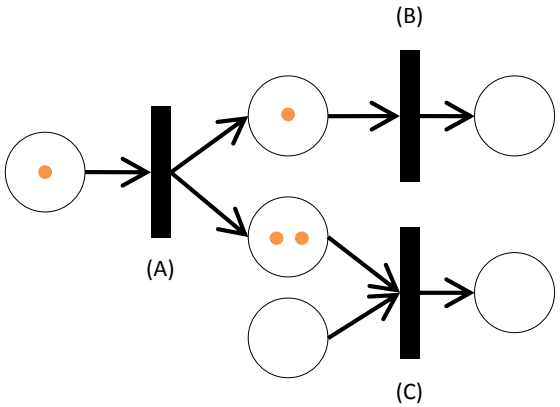

Figura 15 - Exemplo de uma rede de Petri. Nesta rede, as transições (A) e (B) estão habilitadas.

Uma transição habilitada pode ser disparada (isto é, a ação ou evento que ela representa é processado). Quando isso ocorre, os marcadores das posições

de entrada da transição são removidos e novos marcadores são colocados em cada posição de saída da transição. Essa movimentação de marcadores representa a mudança de estado do sistema ou a movimentação de dados entre as partes de um sistema. A figura 16 mostra dois exemplos de redes de Petri e seus estados antes do disparo das transições habilitadas (1) e após o seu disparo (2).

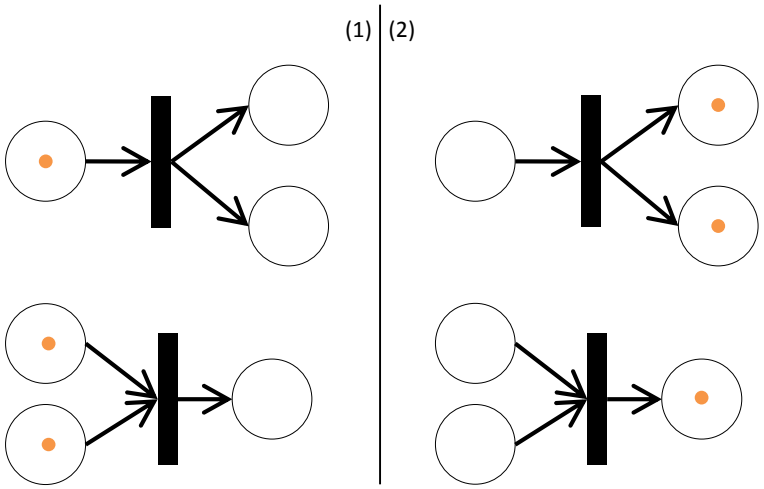

Figura 16 – (1) Exemplo de duas transições habilitadas. (2) Estado da rede após o disparo das transições.

Além do modelo tradicional de redes de Petri, diversas extensões foram propostas para representação de elementos adicionais tais como hierarquias de redes e manipulação de tempo. Uma extensão de redes de Petri especialmente útil para fluxos de trabalho científicos é a adição de cor aos marcadores e posições da rede para representar tipos de dados. (VAN DER AALST, 1998)

A principal vantagem de usarmos redes de Petri e suas extensões para representar fluxos de trabalho é a facilidade em mapear os elementos de um fluxo de trabalho em elementos de uma rede de Petri, resumido na tabela abaixo:

Fluxo de trabalho	Rede de Petri (com extensão de cor)
Tarefa	Transição
Parâmetro (de uma tarefa ou fluxo)	Posição
Valor de um parâmetro	Marcador (*token*)
Tipo de um parâmetro	Cor da posição
Tipo de um valor	Cor do marcador
Execução da tarefa	Disparo da transição

Tabela 1 - Mapeamento entre os elementos de fluxos de trabalho e de redes de Petri.

Além disso, redes de Petri possuem uma semântica formal bastante clara e precisa que facilitam o desenvolvimento de motores de execução de fluxos e a

construção de ferramentas automáticas de verificação para detectar erros na composição do fluxo. (VAN DER AALST, 1998)

O uso de redes de Petri para representar fluxos de trabalho também foi motivado por termos encontrado um motor de fluxos de trabalho em C++, CppWfms[12], baseado em redes de Petri e de código aberto, que pudemos usar como referência para a implementação de um motor para o nosso trabalho.

Uma questão importante a que devemos nos atentar ao usar redes de Petri é que a execução de um fluxo de trabalho deve ser determinística, no sentido que se um mesmo fluxo for executado repetidas vezes com os mesmos valores de entrada, os resultados gerados devem ser os mesmos. Em uma rede de Petri, porém, quando múltiplas transições estiverem habilitadas ao mesmo tempo, a ordem de disparo delas não é determinada. Isso pode causar dois tipos de problema:

1. Se as transições habilitadas possuírem efeitos colaterais, o estado final do sistema pode variar conforme a ordem de disparo dessas transições.
2. Se duas transições habilitadas compartilham têm uma mesma posição de entrada em comum, a ordem de consumo dos marcadores pelas transições não é determinada. É possível, inclusive, que todos os marcadores sejam consumidos por uma única transição e a outra transição nunca seja disparada.

A figura 17 ilustra essas duas situações de não-determinismo em uma rede de Petri. Em (1), duas ordens de disparo diferentes são possíveis: Primeiro a transição A e depois a transição B ou vice-versa. Já em (2), apenas uma das transições habilitadas C ou D será disparada, consumindo o único marcador na posição de entrada, o que torna a outra transição desabilitada. Neste caso temos duas marcações (estados) finais possíveis: Um marcador na posição S1 (se a transição C for disparada) ou um marcador na posição S2 (se a transição D for disparada).

[12] (PELLEGRINI; GIACOMINI, 2008), disponível em http://cppwfms.sourceforge.net/.

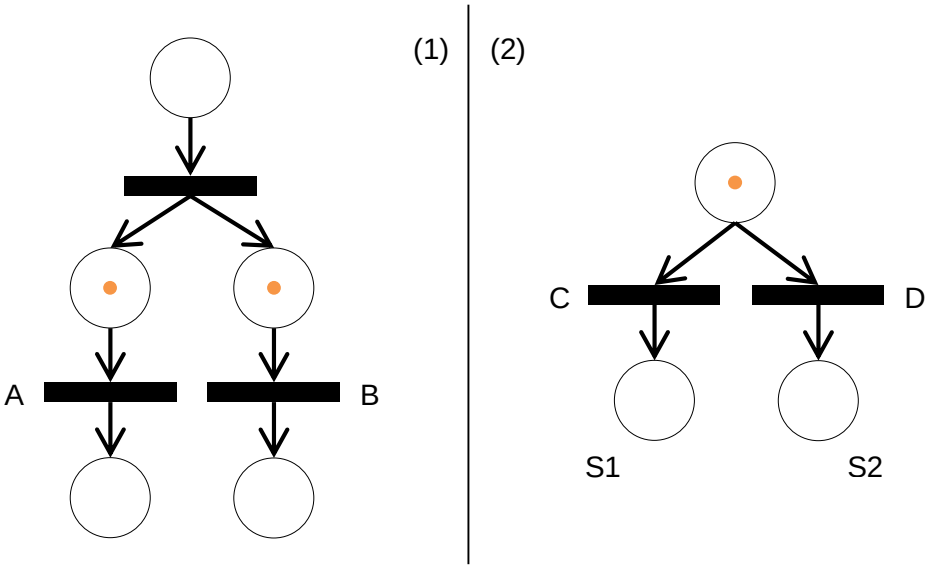

Figura 17 - Exemplos de situações de não-determinismo em redes de Petri.

Como estamos considerando tarefas como funções puras, sem efeitos colaterais, uma configuração como em (1) não representa um problema. Já uma configuração como em (2) pode surgir por um descuido em tentar representar o uso de um resultado de uma tarefa como parâmetro de entrada de duas tarefas subsequente. Nesses casos, a semântica desejada, em termos de uma rede de Petri, equivale a uma “transição de cópia” do marcador que representa o resultado da tarefa, de forma a passar uma cópia do marcador para cada uma das transições seguintes.

A figura abaixo mostra um exemplo de fluxo de trabalho em que um mesmo parâmetro (E1) é usado como entrada para duas tarefas diferentes (A) e (B). Se fizermos uma transformação simples desse fluxo em uma rede de Petri aplicando apenas as equivalências listadas na tabela 1, teríamos uma rede similar à rede (2) da figura 17. Isso levaria a um comportamento incorreto, pois apenas uma das tarefas seria realizada, e não as duas. A figura abaixo mostra, assim, a visão correta do fluxo como uma rede de Petri, introduzindo uma transição de cópia de marcadores para representar a passagem do valor do parâmetro de entrada (E1) para ambas as tarefas.

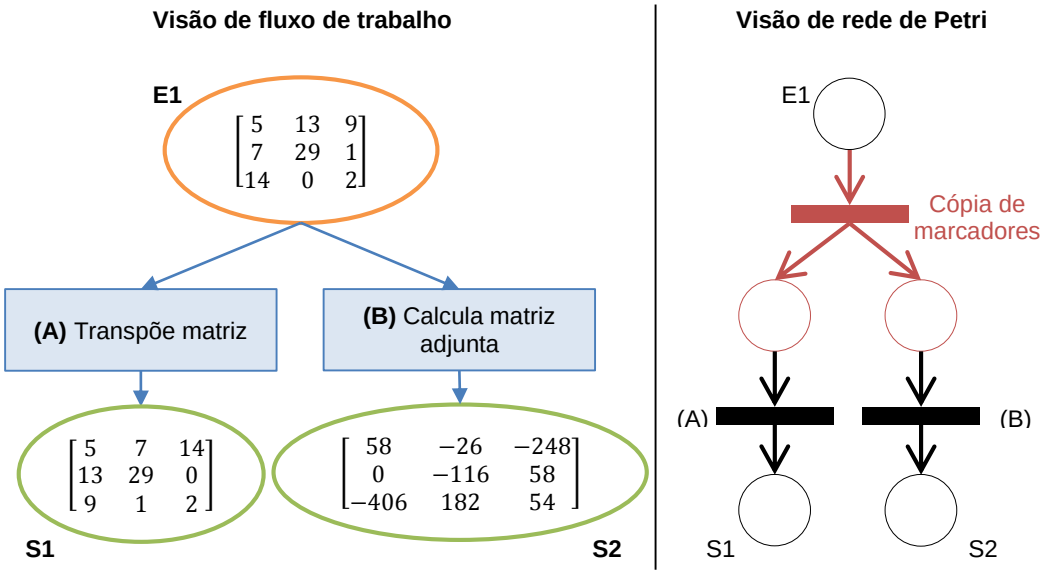

Figura 18 - Exemplo de uso de um parâmetro como entrada de duas tarefas, que poderia levar a uma representação errada como na figura 17 (2), e a sua representação correta com a cópia dos marcadores.

Também é importante evitar em uma rede de Petri que represente um fluxo de trabalho quaisquer construções que levem a execução a um impasse (*deadlock*) ou a laços infinitos.

Para que as redes de Petri montadas na aplicação para representar fluxos de trabalho respeitem todas essas restrições, ao invés de manipularmos as redes de Petri diretamente, usamos uma classe `FluxoTrabalho`, ilustrada na figura 19. Essa classe guarda internamente a rede de Petri usada para representar o fluxo e provê métodos para a adição de tarefas e parâmetros de entrada do fluxo. Assim, podemos controlar e verificar as composições de redes pelo desenvolvedor de forma a garantir que a rede de Petri represente um fluxo de trabalho válido.

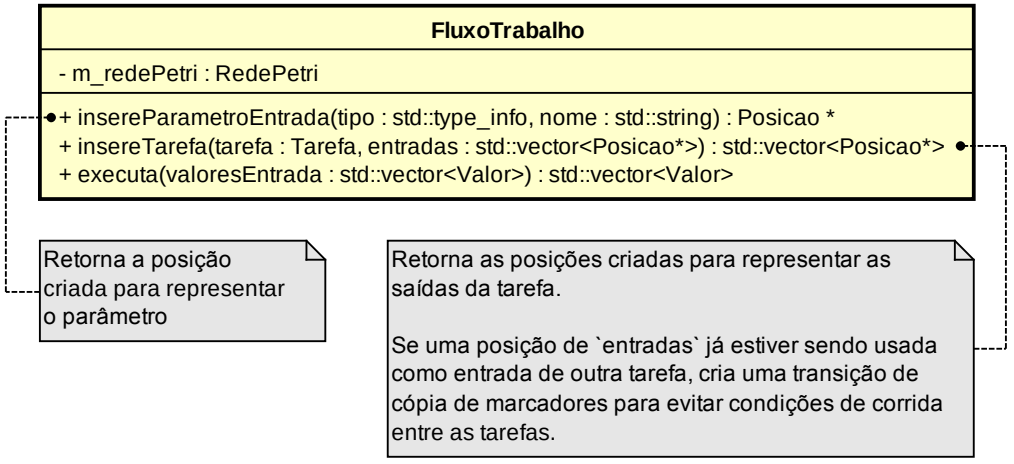

Figura 19 - Classe `FluxoTrabalho`.

4 Implantação

Nesta seção, descreveremos alguns detalhes da implantação dos conceitos de fluxos de trabalho na nossa aplicação. Comentaremos primeiro as principais mudanças na estrutura do modelo geológico e no método de desenvolvimento da aplicação para passar a registrar fluxos de trabalho. Nós discutiremos mais detalhadamente, então, o processo de adaptar as funções da aplicação para a interface de tarefa do fluxo e como isso levou à reengenharia de uma série de pontos, em especial quanto à decomposição de funções muito grandes (*god classes*) em funções menores. Por último, apresentaremos uma estratégia para viabilizar a adaptação gradual dos diversos fluxos de trabalho e operações da aplicação ao longo das iterações de desenvolvimento da aplicação.

4.1. Mudanças na estrutura de um modelo geológico

Tradicionalmente, um modelo geológico armazena uma coleção de instâncias de objetos geológicos, tais como poços, falhas etc. Essa estrutura simples é representada na figura abaixo:

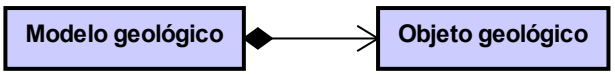

Figura 20 - Diagrama de classes da estrutura tradicional de um modelo geológico como uma simples coleção de objetos.

Com a introdução de fluxos de trabalho na aplicação, parte desses objetos geológicos passará a ser armazenados como resultados da execução de um fluxo, e não mais como instâncias explícitas dos objetos. Um modelo geológico passa assim a armazenar também objetos que representam execuções de fluxos de trabalho. Esses novos objetos registram as operações executadas pelo usuário na aplicação durante a geração desse modelo e os parâmetros usados nessas operações. Os dados brutos de prospecção, que o usuário busca de bases de dados ou importa de arquivos e usa como entrada para começar a criar o modelo, continuam sendo armazenados como instâncias dos objetos geológicos diretamente pelo modelo. Já os demais objetos que são gerados pela aplicação a partir das operações realizadas pelo usuário passam a ser

registrados no modelo como referências para a saída da execução de fluxo que os geraram. A figura 21 ilustra a nova estrutura de um modelo geológico com o registro de execuções de fluxos de trabalho.

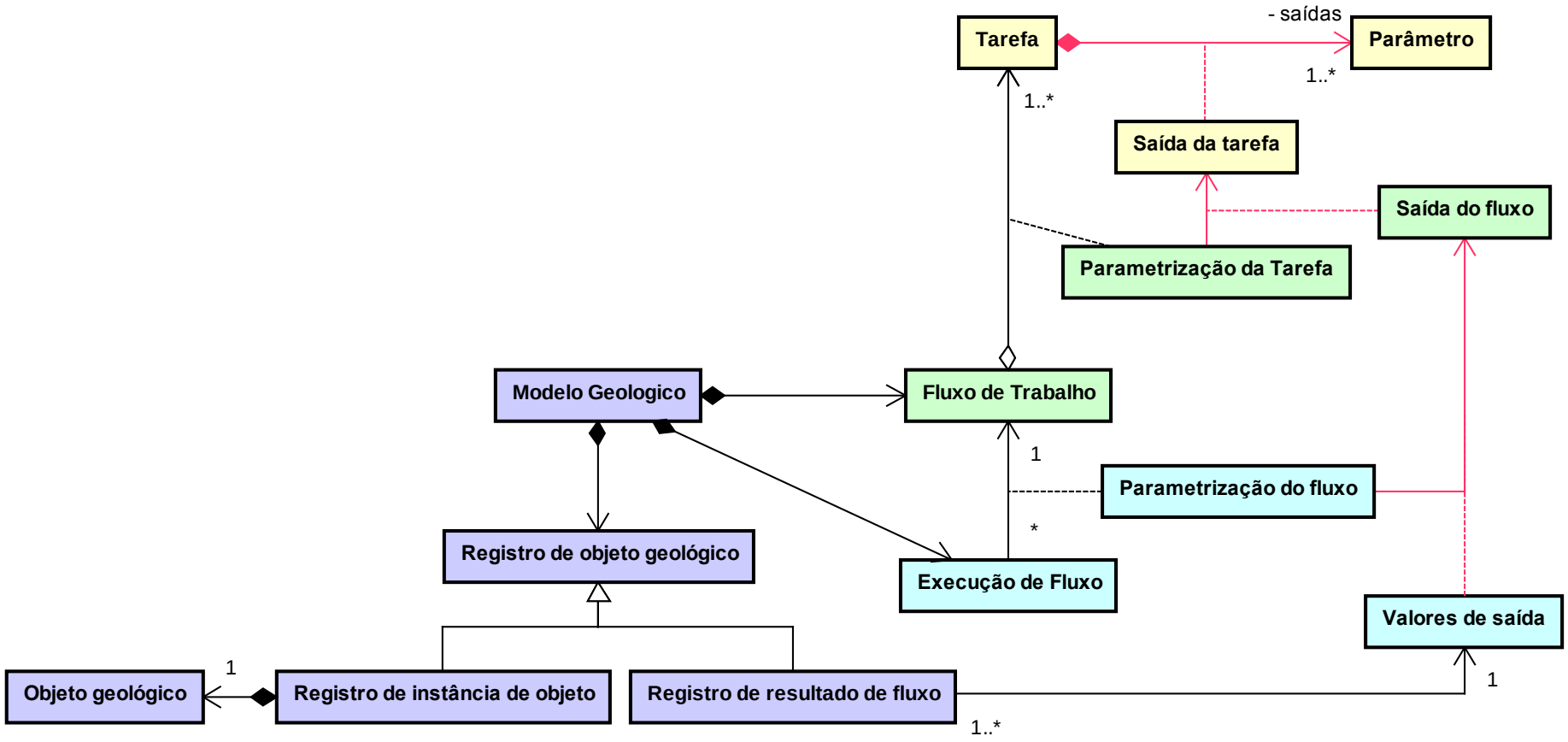

Figura 21 - Diagrama de classes da estrutura de um modelo geológico com fluxos de trabalho.

Além da mudança na estrutura interna do modelo geológico, novos métodos devem ser adicionados à classe de modelo geológico para permitir o registro de objetos gerados por fluxos na aplicação e a recuperação posterior da execução de fluxo que gerou um determinado objeto do modelo, tal como ilustrado na figura abaixo:

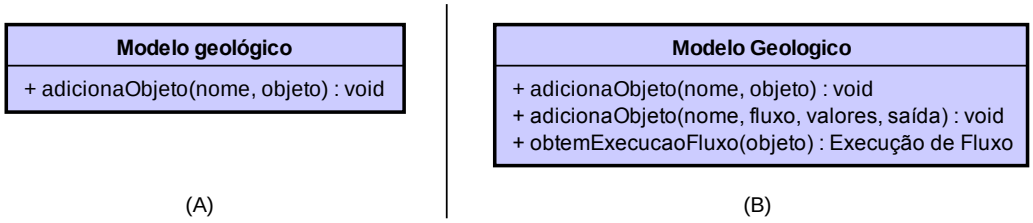

Figura 22 - Diagrama de classes com os métodos originais para registro de objetos em um modelo geológico (A) e os novos métodos para registro e recuperação de execuções de fluxos de trabalho (B).

4.2. Mudanças no método de desenvolvimento

A principal mudança na aplicação com a adição de fluxos de trabalho se dá na forma de desenvolver o código dos componentes da aplicação que representam a lógica de controle das operações disparadas pelo usuário. Esses componentes de controle, que tradicionalmente invocam os demais componentes de cálculo e transformações de dados da aplicação que compõem a operação (no nosso caso, funções), devem passar a montar um fluxo de trabalho, transformando as chamadas diretas às funções de cálculo em especificações de tarefas no fluxo.

Para exemplificar essa mudança, tomemos uma situação em que o usuário deseja analisar a média da densidade de um determinado tipo de rocha (argila) ao longo de uma região. Para isso, ele usa os perfis[13] de densidade (DENS) dos poços perfurados na região, filtrando-os para manter apenas os valores medidos nas rochas de argila (A). Por fim, o usuário calcula a média desses valores em cada poço (B) e faz uma extrapolação desses valores ao longo de toda a região através de uma triangulação Akima (C). O resultado é um mapa da média da densidade nas rochas de argila naquela região. A figura 22 mostra uma representação gráfica desse fluxo.

[13] Um **perfil de poço** (*well log*) é um registro de propriedades físicas medidas ou formações geológicas encontradas ao longo da perfuração de um poço.

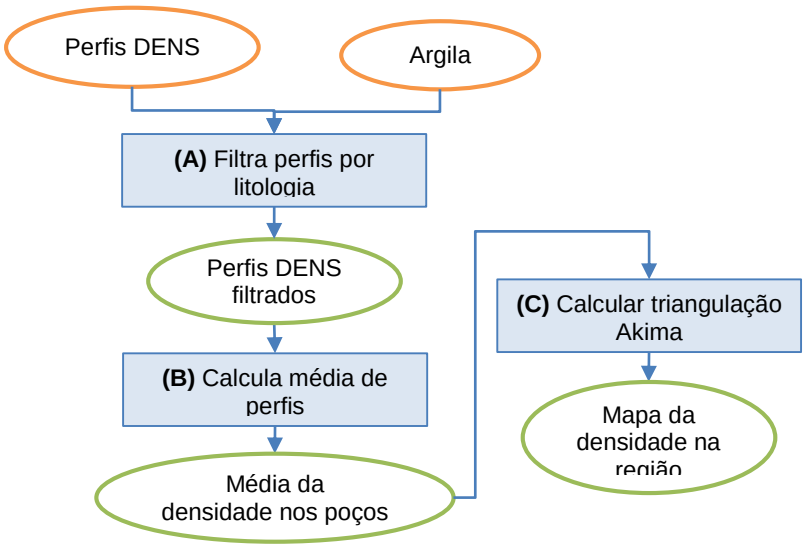

Figura 23 - Fluxo de exemplo: cálculo da média da densidade nas rochas de argila.

Tradicionalmente, quando o usuário dispara uma operação na interface da aplicação, o controlador acionado para executar a operação invoca diretamente cada uma das funções para realizar esses cálculos e processamentos (o filtro de perfil, o cálculo de média e a triangulação). Para cada função invocada, o controlador guarda o objeto retornado (os perfis filtrados, os valores da média em cada poço e por fim o mapa), repassando-o como argumento da função seguinte. Ao concluir as chamadas, o controlador adiciona o objeto final do processamento (o mapa de densidades) no modelo geológico. Esse comportamento é descrito no diagrama de sequência da figura 24.

Com fluxos de trabalho, os controladores devem representar essa sequência de chamadas de função um fluxo. O controlador, assim, deve criar um objeto de fluxo de trabalho, inserir nele as funções como tarefas, indicando as conexões dos parâmetros de entrada e saída dessas tarefas. Ao final, o controlador deve registrar o próprio fluxo no modelo geológico, especificando os valores dos parâmetros de entrada a serem usados na sua execução. O modelo geológico, então, armazena essa especificação de execução de fluxo, disparando-a sempre que for necessário obter os objetos do modelo que o fluxo representa (no caso, o mapa de densidades). Esse novo comportamento é descrito no diagrama de sequência da figura 25.

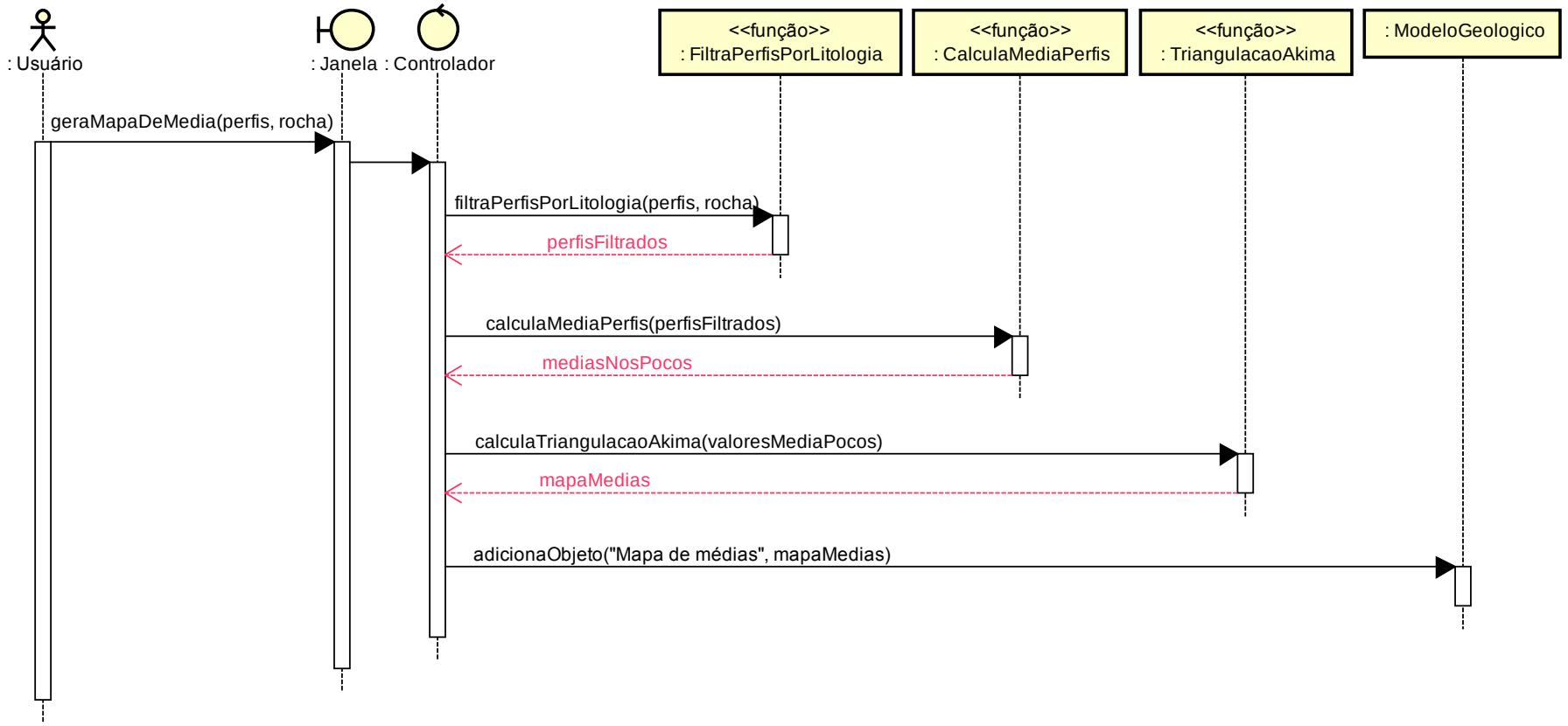

Figura 24 - Diagrama de sequência conceitual representando o comportamento atual da aplicação ao ser disparada uma operação: os cálculos são feitos imediatamente e o objeto resultante é adicionado ao modelo.

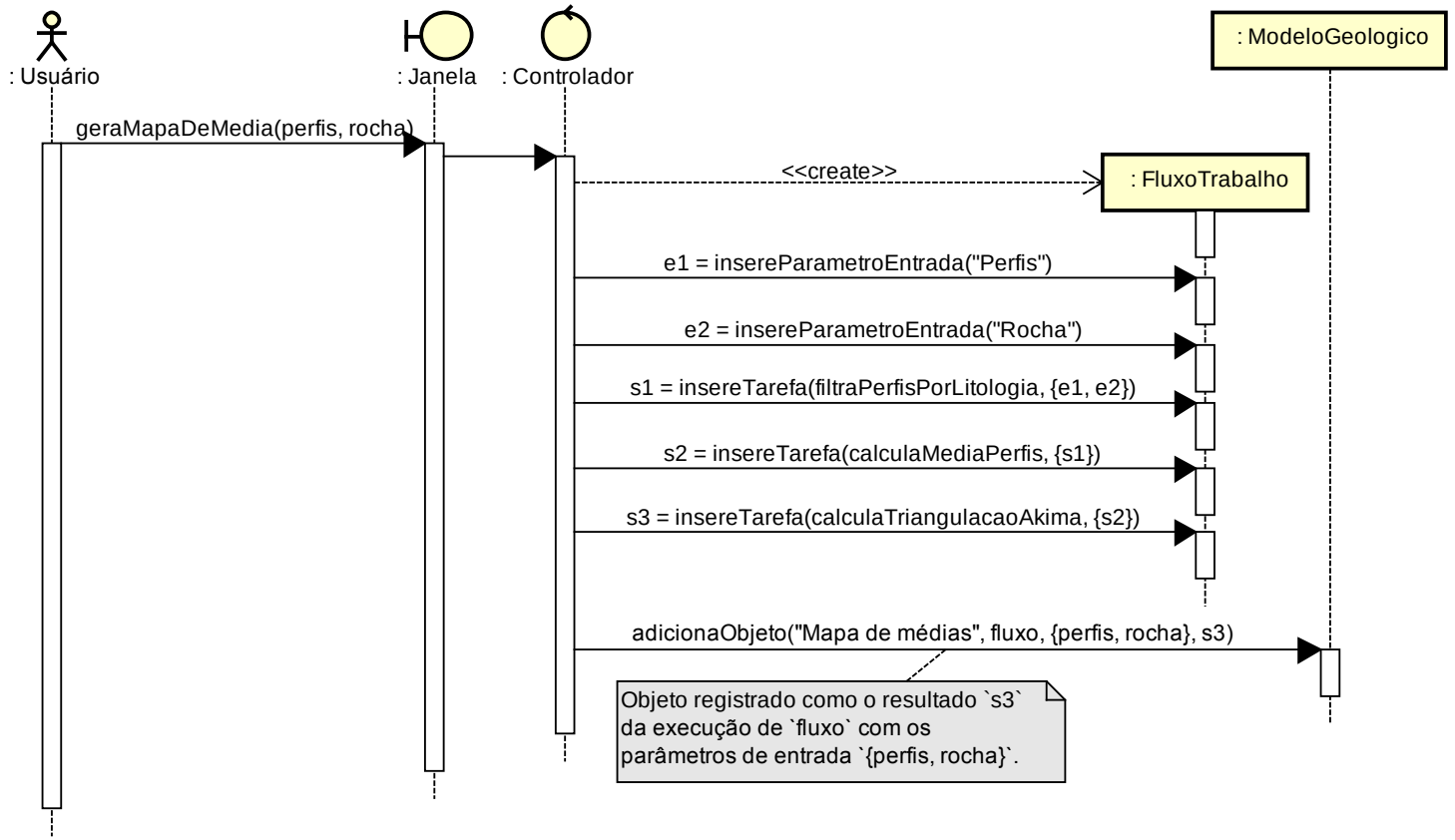

Figura 25 - Diagrama de sequência conceitual representando o novo comportamento da aplicação ao ser disparada uma operação: um fluxo de trabalho é composto representando os cálculos a serem realizados, e o novo elemento a ser adicionado no modelo é registrado como o resultado da execução desse fluxo com os parâmetros de entrada informados pelo usuário. Os parâmetros `tipo` de `insereParametroEntrada()` foram omitidos para simplificar o diagrama.

4.3. Adaptação de funções em tarefas

Como dissemos na 3.3.1, para usarmos uma função C++ da nossa aplicação como uma tarefa no fluxo de trabalho, nós usamos o gabarito de classe `Funcao` como adaptador.

Na maioria dos casos, essa adaptação é bem simples: Tornamos a função original privada (removendo a sua declaração do arquivo de cabeçalho e movendo a sua definição para um *namespace* anônimo no arquivo fonte) e declaramos uma variável do tipo do `Funcao` que representa uma chamada à função original. O corpo da função original, que contém a lógica da operação realizada, é mantido inalterado. O quadro abaixo ilustra esse processo de adaptação:

Código original	Código adaptado
Cabeçalho: Akima.h	
`Grid akima(const Property &);`	`extern const Funcao<Grid (const Property &)> akima;`
Fonte: Akima.cpp	

`Grid akima(const Property & p)`
`{`
`  // Lógica da triangulação`
`  ...`
`}`
	`namespace {`
	
	`Grid akimaImpl(const Property & p)`
	`{`
	`  // Lógica da triangulação`
	`  ...`
	`}`
	
	`} // Fim do namespace`
	
	`const auto akima = criaTarefa(`
	`  akimaImpl,`
	`  "Calcula triangulação Akima",`
	`  { "Propriedade" },`
	`  { "Mapa" }`
	`);`

Quadro 1 - Exemplo de adaptação de uma função para a interface de `Tarefa`. Neste exemplo, `criaTarefa()` é uma função de fábrica de objetos `Funcao`.

Em alguns casos, precisamos rever e fazer pequenas alterações na assinatura das funções para que pudessem ser usadas como tarefas no fluxo, em especial em relação às seguintes práticas tradicionais de programação em C e C++:

Parâmetros usados para retorno de valores: Usado para retornarem múltiplos objetos, é ruim porque a distinção entre parâmetro de entrada, saída (ou ambos) é feita exclusivamente via documentação do código. Para poder ser

usada com a classe `Funcao`, é necessário transformar os parâmetros de saída em retornos propriamente ditos.

Código original

```
void calculaMetricas(
    const Property & p,
    double & mediaAritmetica,
    double & mediaGeometrica,
    double & mediaHarmonica,
    double & mediaQuadratica )
{
      // Lógica do cálculo
      ...
}
```

Código adaptado

```
std::tuple<double, double,
           double, double>
calculaMetricas(
    const Property & p)
{
      double mediaAritmetica,
             mediaGeometrica,
             mediaHarmonica,
             mediaQuadratica;

      // Lógica do cálculo
      ...

      return std::make_tuple(
          mediaAritmetica,
          mediaGeometrica,
          mediaHarmonica,
          mediaQuadratica);
}
```

Quadro 2 – Exemplo de adaptação da assinatura de uma função para trocar parâmetros de saída por retornos da função.

Modificações nos parâmetros de entrada: Usado para evitar cópias desnecessárias de objetos temporários, é um caso especial de uso de parâmetro para retorno de valores. Com a introdução de *move-semantics*[14] no C++11[15], esse padrão é desencorajado. A adaptação de funções para uso com a classe `Funcao`, nesse caso, é feita de duas formas:

1. Se o objeto de entrada é uma classe concreta, adicionamos um construtor de *move* na classe (ou usamos o gerado automaticamente pelo compilador) e alteramos a assinatura da função para receber e retornar o objeto por valor.
2. Se o objeto de entrada é uma classe abstrata (ou usada polimorficamente), alteramos a assinatura da função para receber e retornar o objeto por `std::unique_ptr`.

[14] *Move semantics* são um conjunto de recursos adicionados no C++11 para representar e manipular objetos temporários ou no fim do seu ciclo de vida (chamados rvalues, *prvalues* ou *xvalues*). O principal uso de *move semantics* é evitar cópias desnecessárias de objetos temporários ao serem passados como parâmetro ou retornados de uma função.

[15] C++11 é o nome popular da terceira edição do padrão da linguagem de programação C++ (ISO/IEC 14882:2011), de 12 de agosto de 2011.

Código original

Classe concreta: `PerfilRegular`

```
class PerfilRegular :
    public Perfil
{
public:
      PerfilRegular();
      ...
};



void filtraPerfilPorLitologia(
    PerfilRegular & p,
    const Rocha & rocha )
{
    // Lógica da função
    // (altera diretamente os
    // valores de `p`):
    ...
}


void filtraPerfilPorZona(
    PerfilRegular & p,
    const Zona & zona );
```

Classe abstrata: `Perfil`

```


void filtraPerfilPorLitologia(
    Perfil & p,
    const Rocha & rocha )
{
    // Lógica da função
    // (altera diretamente os
    // valores de `p`):
    ...
}









void filtraPerfilPorZona(
    Perfil & p,
    const Zona & zona );
```

Código adaptado

Classe concreta: `PerfilRegular`

```
class PerfilRegular :
    public Perfil
{
public:
      PerfilRegular();

      // Construtor de move:
      PerfilRegular(
          PerfilRegular&&);

      ...
};

PerfilRegular
filtraPerfilPorLitologia(
    PerfilRegular p,
    const Rocha & rocha )
{
    // Lógica da função
    // (igual à original):
    ...

    return p;
}

PerfilRegular
filtraPerfilPorZona(
    PerfilRegular p,
    const Zona & zona );
```

Classe abstrata: `Perfil`

```
std::unique_ptr<Perfil>
filtraPerfilPorLitologia(
    std::unique_ptr<Perfil> _p,
    const Rocha & rocha )
{
    assert(_p);

    Perfil & p = *_p;

    // Lógica da função:
    // Igual à original.
    ...

    return _p;
}

std::unique_ptr<Perfil>
filtraPerfilPorZona(
    std::unique_ptr<Perfil> _p,
    const Zona & zona );
```

Quadro 3 - Exemplo de adaptação de funções para remover modificações nos parâmetros de entrada.

Funções retornando ponteiros gerenciados pelo invocador: Embora essa situação não tenha acontecido na nossa aplicação, um terceiro fator que

identificamos que pode demandar uma adaptação na assinatura da função é o retorno de ponteiros para objetos alocados dinamicamente na função (alocação no *heap*) e deleguem ao invocador a gerência e liberação desse objeto da memória. Esse padrão é comum em aplicações mais antigas e principalmente em interfaces de bibliotecas externas desenvolvidas em C. A responsabilidade de gerência de memória, nesses casos, está definida apenas na forma de documentação, o que impede que o motor de fluxo saiba quando ele deve fazer esse tipo de gerência. Assim, a assinatura da função deve ser alterada para que ela retorne um objeto ponteiro inteligente (*smart pointer*)[16] tal como `std::unique_ptr` que cuide da gerência de memória.

As adaptações mostradas acima têm a vantagem de incorporar no código da aplicação padrões de programação modernos e novos recursos da linguagem de programação. Pode-se, assim, aproveitar esse processo de adaptação das operações no código para também fazer uma revisão e refatoração do código.

É importante lembrar, contudo, que todas as adaptações que fizemos acima pressupõem que as funções e objetos modificados fazem parte da nossa aplicação e que temos livre acesso para modificá-los. Pode haver, entretanto, casos em que as funções ou objetos que desejamos usar em um fluxo de trabalho pertençam a bibliotecas ou componentes externos e que não tenhamos como alterá-los. Nesses casos, precisaremos criar adaptadores específicos para ajustá-los à interface de tarefa do fluxo. Isso pode ser feito de duas formas:

1. Criando uma função na nossa aplicação que encapsule a chamada ao componente externo, fazendo todas as adaptações necessárias para a interface de tarefa, e registrando essa função através da classe `Funcao`. Essa abordagem é indicada para adaptar funções e componentes individuais.
2. Criando uma nova classe de adaptador que estenda diretamente a interface de Tarefa. Essa abordagem é indicada para adaptar um conjunto de funções ou componentes de uma mesma família (por exemplo, de uma mesma biblioteca externa) e para os quais seja possível criar uma única classe de adaptador que sirva para todos eles.

[16] Um *ponteiro inteligente* é uma estrutura de dados que simula um ponteiro enquanto provê funcionalidades adicionais de gerência de memória. Ponteiros inteligentes ajudam principalmente a reduzir erros de vazamento de memória. A biblioteca padrão C++11 provê os ponteiros inteligentes `std::shared_ptr` e `std::unique_ptr`.

4.4. Decomposição de operações

Além das mudanças nas assinaturas das funções discutidas acima, o processo de adaptar as operações existentes em fluxos de operações também nos levou a identificar oportunidades de refatoração de algumas operações muito longas e que continham mais responsabilidades do que deveriam (*God Classes*[17]).

Um exemplo que encontramos foi uma operação de cálculo de tensões em uma superfície. Essa operação recebe como entrada uma superfície e um conjunto de valores de propriedades mecânicas dessa superfície que são usadas como base para o cálculo. O componente que representa esse cálculo de tensões no código, além de realizar o cálculo propriamente dito, faz:

1. **Conversão de unidades de medidas**: Os cálculos de tensão são realizados usando unidades de medida padrões, tal como megapascal (MPa) para representar tensões e pressões. Os valores das propriedades medidas da superfície, porém, podem estar em uma unidade de medida diferente da esperada. O usuário também pode desejar que as tensões resultantes sejam geradas em uma outra unidade de medida específica. A operação de cálculo de tensões foi, assim, originalmente projetada para aceita um parâmetro a mais indicando a unidade de medida de saída desejada e, internamente, além do cálculo das tensões, faz as conversões necessárias.
2. **Uso de fórmulas para calcular a propriedade**: Em alguns casos, ao invés de usar os valores das propriedades mecânicas medidos na superfície, o usuário deseja usar uma fórmula matemática para definir esses valores. Para suportar esses casos, a operação de cálculo de tensão aceita as propriedades mecânicas de entrada ou como um objeto do modelo que contém os seus valores ou como uma fórmula que é executada para cada ponto da geometria da superfície para gerar esses valores.

Num contexto de fluxo de trabalho, a conversão de unidades de medida e a geração de valores a partir de uma fórmula seriam mais bem representadas como tarefas separadas do fluxo. Essa abordagem permite que simplifiquemos

[17] *God Class* é um anti-padrão de projeto em que uma classe que centraliza uma grande quantidade de trabalho em um sistema, violando o princípio de que cada classe deve ter apenas uma única responsabilidade. Uma g*od class* apresenta uma alta complexidade, baixa coesão interna e dificulta o entendimento e a manutenção do sistema.

tanto a interface da operação de cálculo de tensões, que passa a receber apenas a superfície e os valores das suas propriedades mecânicas, quanto a sua implementação, movendo todo o código que não estava diretamente ligado ao cálculo para fora. A estruturação das operações de conversão de unidades e aplicação de fórmulas como tarefas independentes também permitem que elas sejam usadas na composição de diferentes fluxos de trabalho além do cálculo de tensões.

A figura 25 ilustra três casos de uso da execução do cálculo de tensões: no cenário (1), realiza-se apenas o cálculo de tensões em si; no cenário (2), realiza-se o cálculo e duas conversões de unidades de medida; no cenário (3), realiza-se o cálculo usando uma fórmula matemática para representar os valores de propriedades mecânicas de entrada. Para cada cenário, mostramos como essa execução é representada hoje na aplicação, com o cálculo de tensões sendo uma *god class*, e como ela poderia ser decomposta em um fluxo de tarefas menores.

Os diagramas de sequência das figuras Figura 27 e Figura 28 mostram a mudança realizada no código da aplicação para fazer essa decomposição. A figura 27 mostra a estrutura atual do cálculo de tensões, destacando em vermelho os trechos não ligados à lógica do cálculo de tensões propriamente dito. Esses trechos são os que envolvem a aplicação de fórmulas para derivar as propriedades mecânicas de entrada e as conversões de unidades de medida. Já a figura 28 mostra a migração desses trechos para o controlador da aplicação, na forma de uma composição de fluxo de trabalho em que as tarefas de aplicação de fórmula e conversões de unidades são inseridas no fluxo conforme a seleção do usuário.

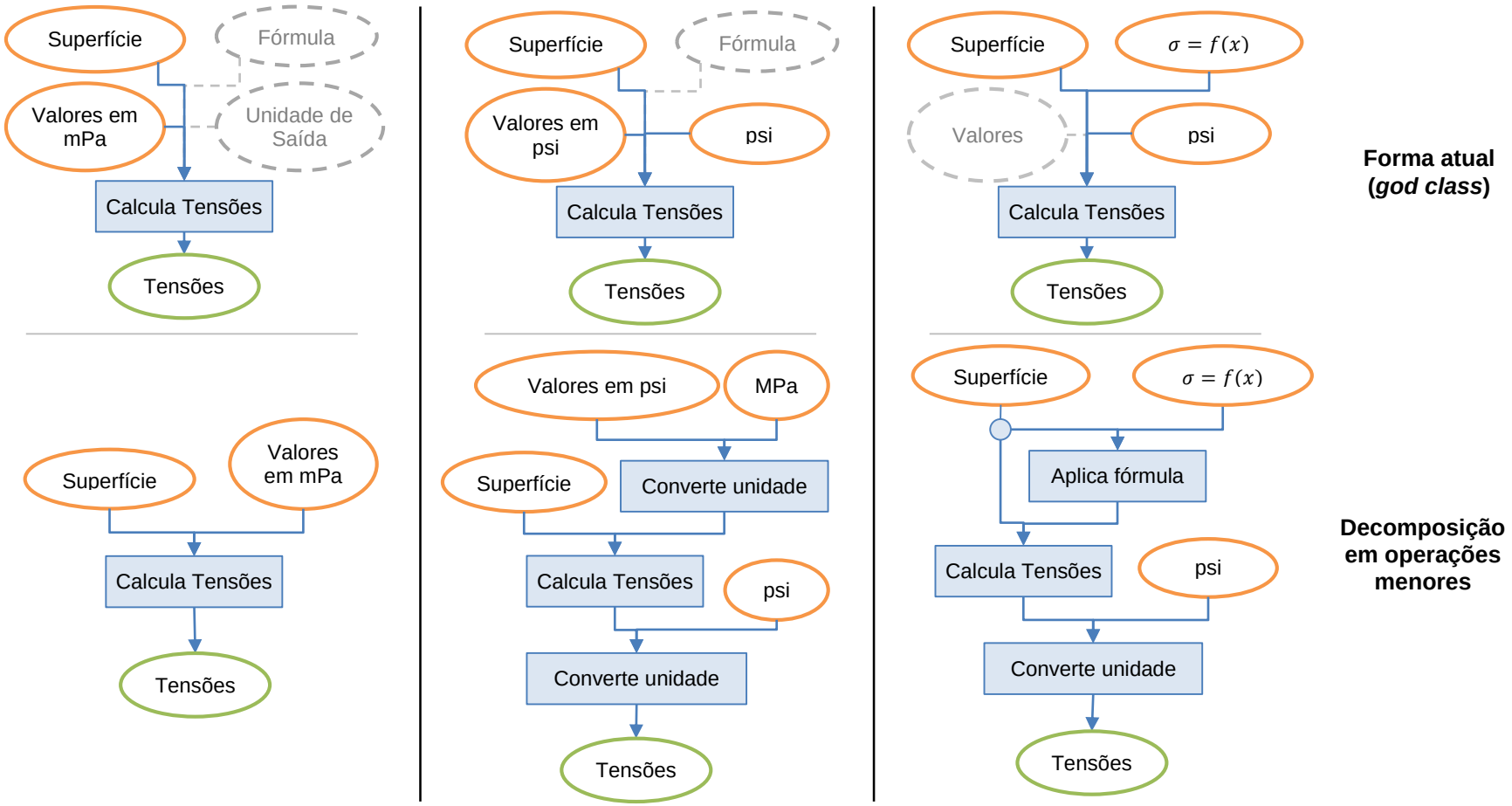

Figura 26 – Exemplos de fluxos de diferentes combinações do cálculo de tensões com conversões de unidade e aplicação de fórmulas na forma atual (acima) e com as operações decompostas em tarefas separadas (abaixo). As elipses com borda tracejada representam parâmetros de entrada opcionais que não foram usados na execução do cálculo de tensões.

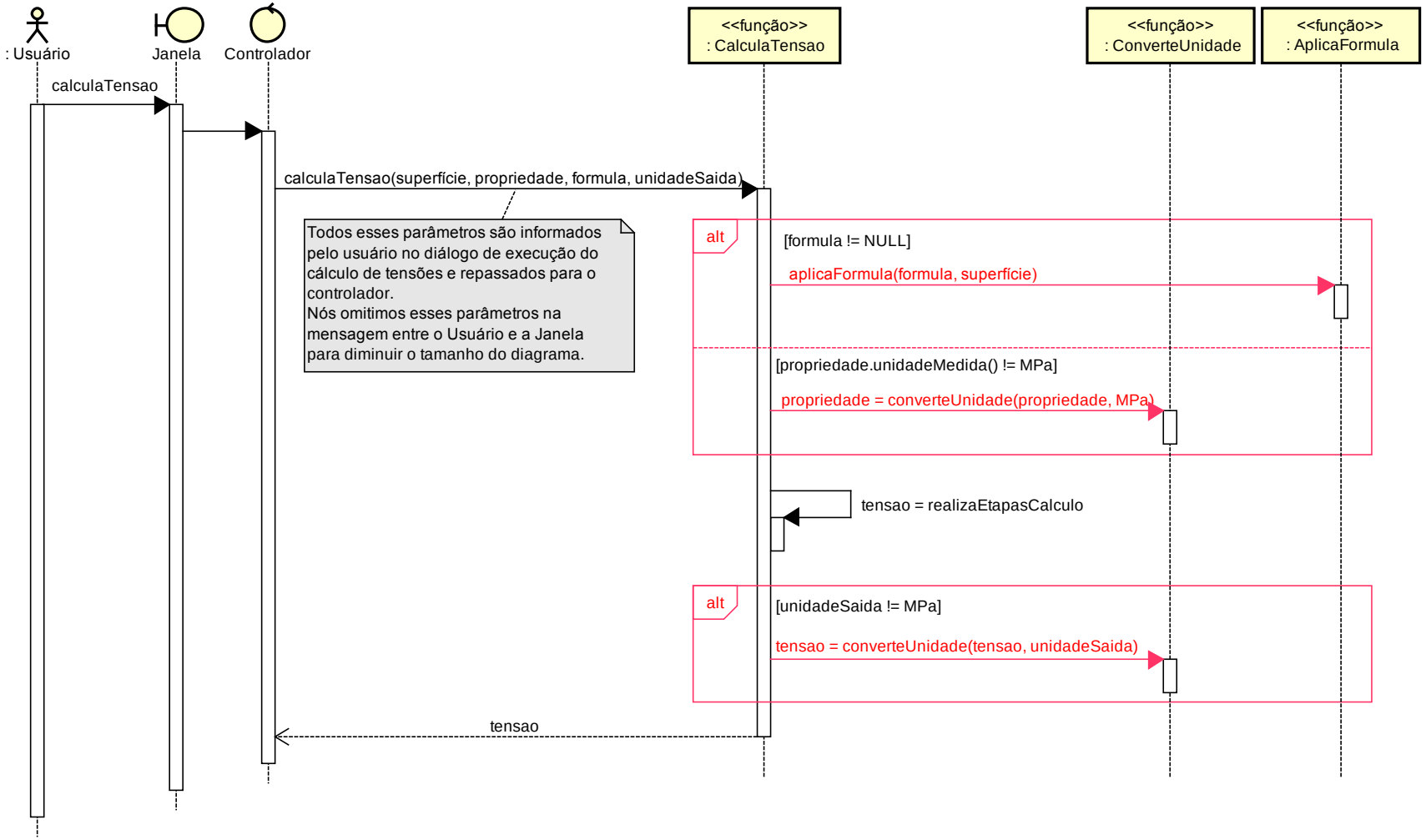

Figura 27 - Diagrama de sequência conceitual da estrutura atual do cálculo de tensões.

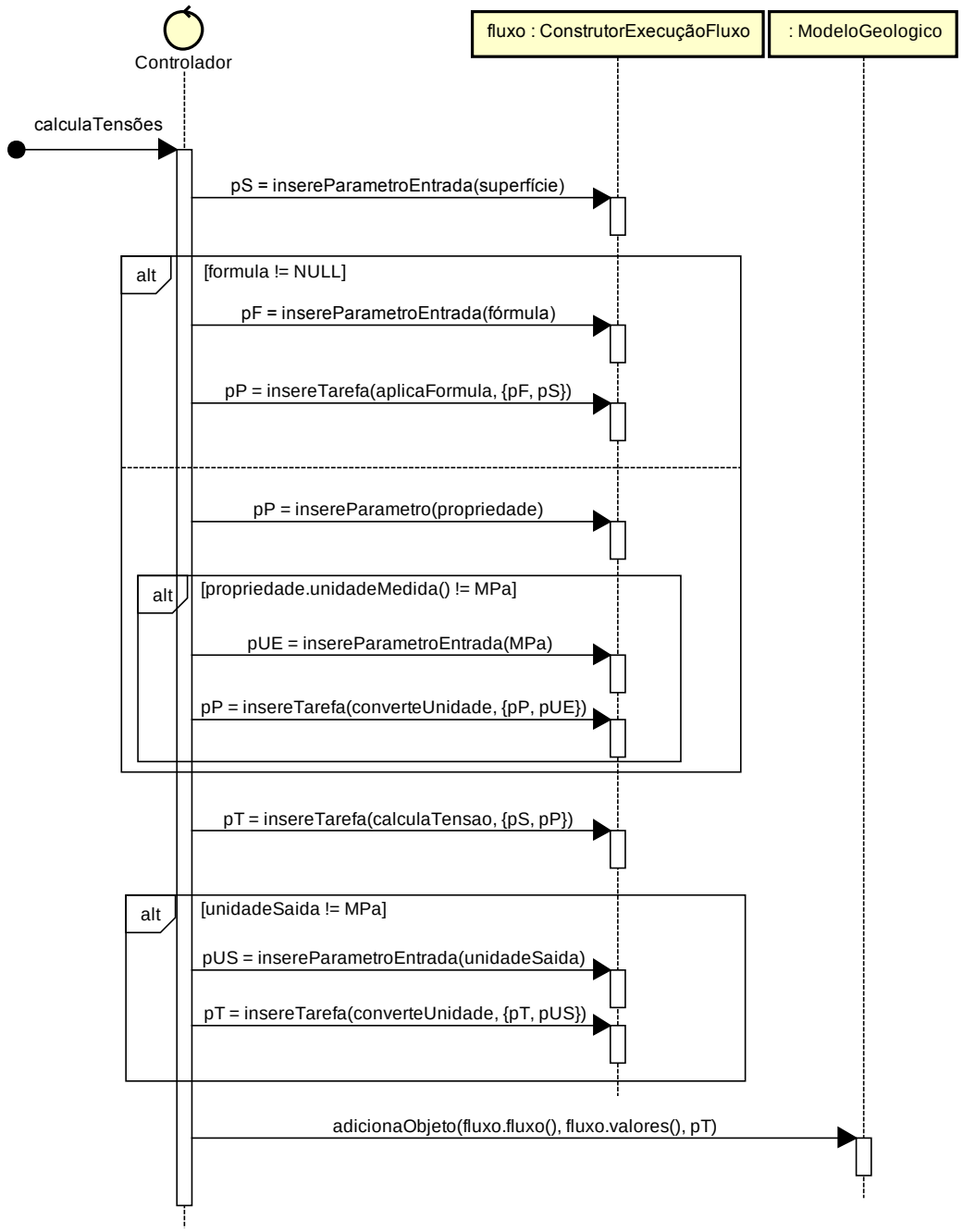

Figura 28 - Diagrama de sequência conceitual da criação de um fluxo de trabalho com as operações de conversão de unidade e execução de fórmula separadas do cálculo de tensões.

4.5. Priorização das operações e fluxos de trabalho a serem adaptados

Como a aplicação é muito grande e adota uma prática de entrega contínua em que é preciso balancear o desenvolvimento dos recursos de fluxo de trabalho com outras demandas dos usuários que podem surgir, pareceu-nos inviável, num cenário como esse, que todas as operações existentes na aplicação pudessem ser adaptadas de uma vez só, em uma única iteração de

desenvolvimento. Assim, propomos uma política de adaptação gradual da aplicação baseada nas seguintes diretrizes:

- Toda nova operação adicionada na aplicação já será inicialmente desenvolvida usando a estrutura de fluxos de trabalho.
- A adaptação das operações já existentes na aplicação será gradual e de acordo com a sua prioridade.

Definimos a prioridade de adaptação de cada operação com base nos fluxos de trabalho realizados pelo usuário, seguindo a ordem de execução do fluxo: adaptando primeiro as operações iniciais de um fluxo, depois seguindo a ordem das ligações entre as tarefas, até chegar às suas as operações finais. Essa abordagem garante que o esforço de desenvolvimento esteja focado de forma que cada operação adaptada possa ser imediatamente conectada às demais operações existentes, de forma que não haja “buracos” no fluxo de trabalho. Isso não seria verdade, por exemplo, se planejássemos a adaptação de operações por módulo da aplicação, uma vez que um fluxo de trabalho geralmente cruza as fronteiras dos módulos, encadeando operações de diversos módulos em um único fluxo.

Um exemplo de adaptação parcial das operações que poderia levar a “buracos” em fluxos de trabalho que identificamos na aplicação estudada ocorre no fluxo de cálculo de média da densidade mostrado anteriormente na figura 23. Esse fluxo envolve duas operações de poço (o filtro de perfil por litologia e o cálculo de média dos valores de perfis) e uma operação de *grid* (a triangulação *Akima*). Essas operações estão hoje divididas em dois módulos separados, como ilustrado no diagrama de pacotes (A) da figura 29. Se as operações do módulo de *Grid* forem adaptadas para tarefas, mas as operações do módulo de *Poço* não o forem, a aplicação representará o mapa da densidade como o resultado de um fluxo de um único passo (a triangulação Akima) e considerará a média da densidade nos poços uma entrada de dados bruta. Essa situação é ilustrada no fluxo de trabalho (B) da figura 29.

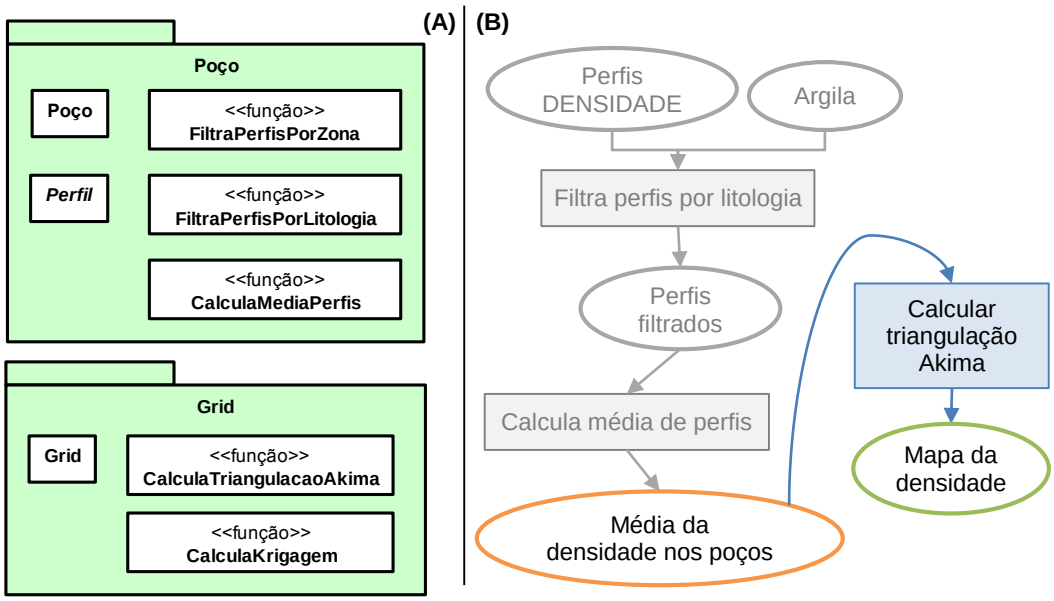

Figura 29 – Exemplo de adaptação com “buracos” de um fluxo de trabalho (em cinza).

Para escolhermos quais fluxos de trabalho abordaríamos primeiro, nos reunimos com os usuários da aplicação e pedimos que eles nos mostrassem os principais cenários de uso que eles executam na aplicação, tentando identificar em quais eles sentiam, por exemplo, a maior necessidade de automação. Com estas informações dadas pelo usuário, pudemos priorizar e planejar quais fluxos de trabalho serão tratados em cada iteração de desenvolvimento.

5 Prova de conceito

Nesta seção, mostraremos dois cenários de uso da nossa aplicação que usamos como prova de conceito da viabilidade de representar as operações feitas pelo usuário como fluxos de trabalho usando a abordagem proposta nesta dissertação. Também discutiremos como essa estrutura de fluxos apoia a criação de três novas funcionalidades na aplicação para atender aos desafios que identificamos na 1.3.

5.1. Fluxos de trabalho implantados

Nós implantamos com sucesso dois fluxos de trabalho que representam dois cenários de uso da aplicação priorizados pelos usuários, ambos no domínio de análise de dados da perfuração poços. O primeiro cenário consiste na geração de um mapa do valor médio dos perfis de um grupo de poços. O segundo consiste no cálculo de um novo perfil em um conjunto de poços a partir da correlação de outros dois perfis nestes mesmos poços.

Nos esquemas dos fluxos que mostraremos a seguir, usaremos a seguinte notação:

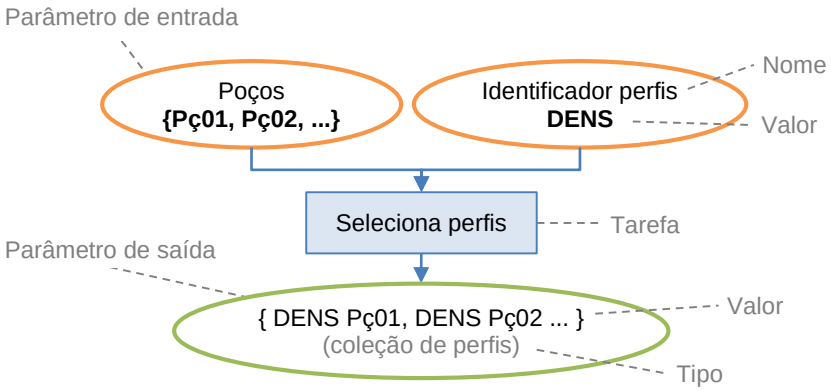

Figura 30 - Notação usada nos esquemas de fluxo de trabalho.

5.1.1. Cenário 1 – Geração de mapa de valor médio de perfis

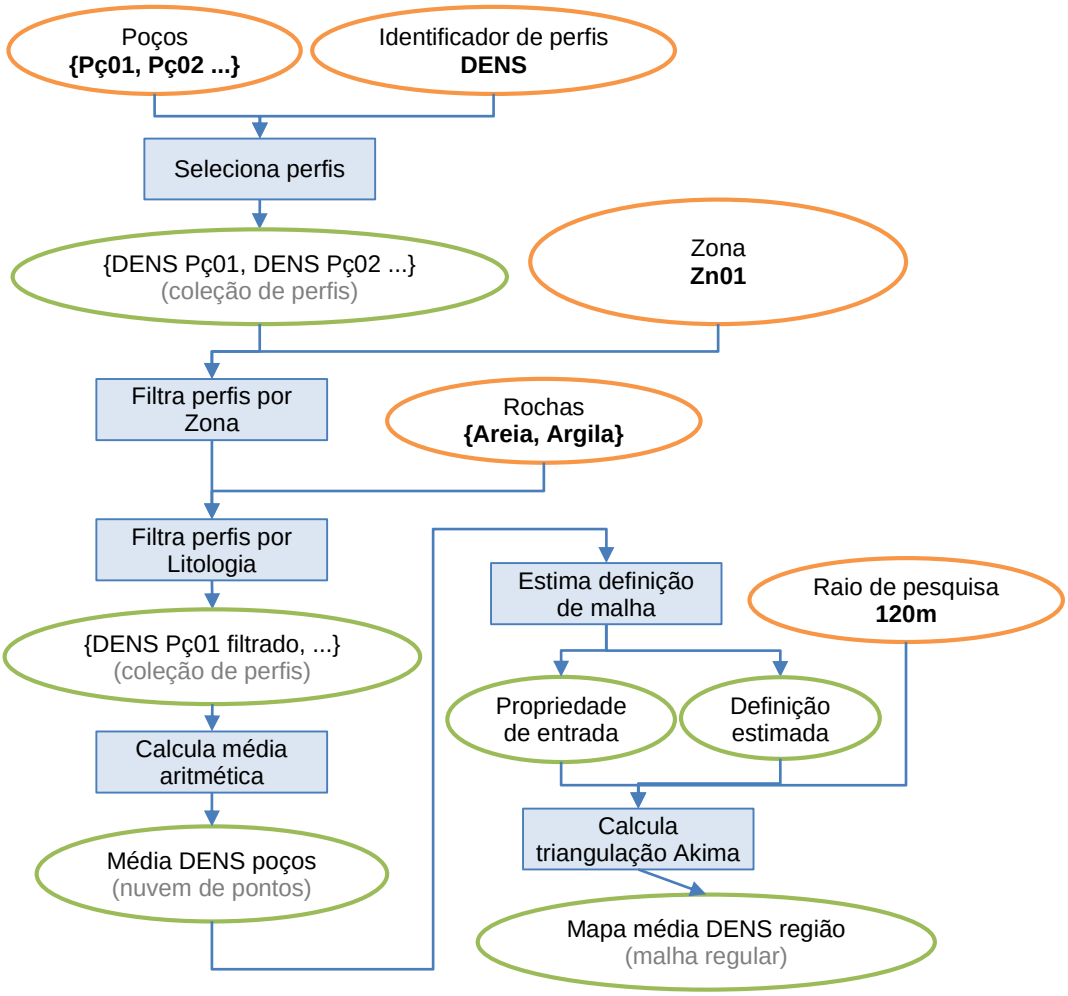

Figura 31 - Esquema do fluxo de criação de um mapa de valor médio de uma propriedade em uma região a partir dos valores medidos na perfuração de poços.

Neste cenário, o usuário deseja estimar a média da densidade das rochas de areia e argila ao longo da região sob estudo. Para isso, ele usa como entrada os valores de densidade medidos na perfuração de poços na região (os perfis DENS desses poços).

Em geral, esse tipo de análise não é feita no perfil completo do poço, mas apenas em alguns intervalos de interesse nesse perfil, definidos por zonas no poço. Essas zonas são delimitadas pelo usuário em cada poço a partir da interpretação dos perfis desses poços. Assim, o usuário filtra os perfis dos poços para manter apenas os valores medidos na zona e no tipo de rocha sobre os quais ele deseja trabalhar. Ele então faz o cálculo da média desses valores em cada poço.

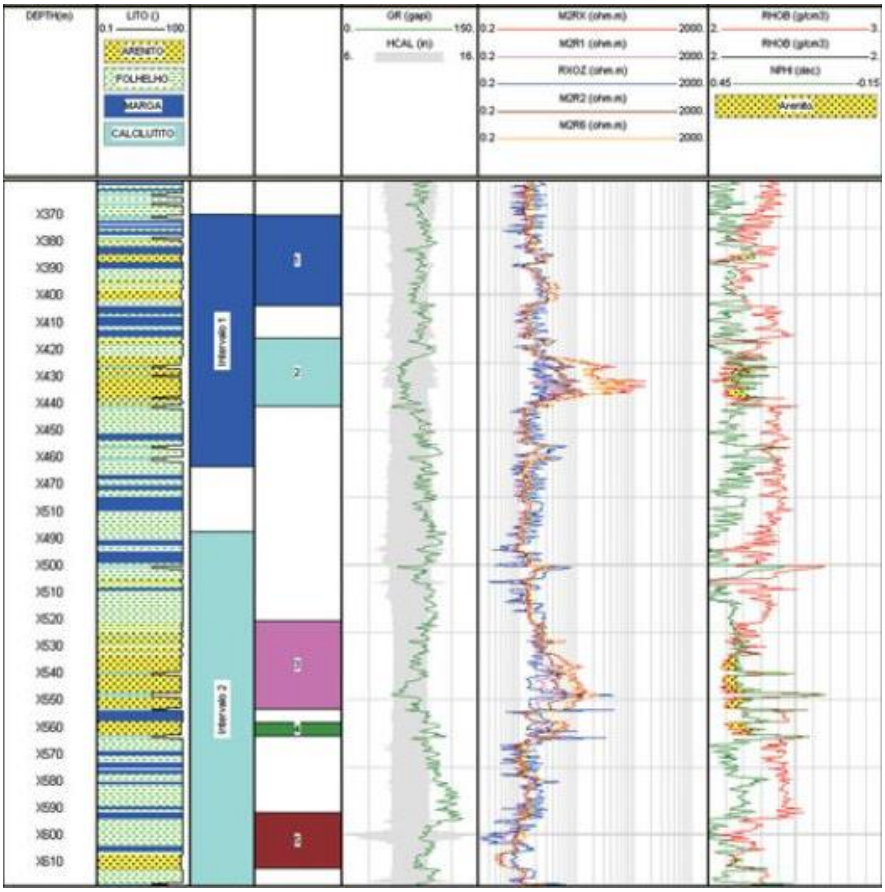

Figura 32 - Exemplo de perfis (trilhas 5 a 7) e zonas (trilha 3 e 4) de um poço. A trilha 1 indica a profundidade em que esses valores foram medidos.[18]

Para estimar a variação desses valores ao longo de toda a região, o usuário faz uma interpolação dos valores da densidade em cada poço, gerando uma malha regular através de uma triangulação Akima. Ao invés de informar explicitamente a definição da malha (o tamanho e quantidade das celas e a orientação da malha), o usuário deixa a própria aplicação estima-la a partir da geometria do objeto de entrada (o conjunto de pontos representando os valores em cada poço).

[18] Retirado de: GUIMARAES, Margareth da Silva Brasil, DENICOL, Paulo Sergio y GOMES, Ricardo Manhães Ribeiro. **Avaliação e caracterização de reservatórios laminados: comparação entre as ferramentas convencionais e o perfil de indução multicomponente.** Rev. bras. geociênc., mar. 2008, vol.38, no.1, supl.1, p.188-206. ISSN 0375-7536.

As tarefas desse fluxo são representadas no código da seguinte forma:

```
std::vector<Perfil>
selecionaPerfis(const std::vector<Poco> &, const std::string &);

std::vector<Perfil>
filtraPerfisPorZona(std::vector<Perfil>, const Zona &);

std::vector<Perfil>
filtraPerfisPorLitologia(std::vector<Perfil>, const std::vector<Rocha> &);

NuvemPontos
calculaMediaAritmetica(const std::vector<Perfil> &);

std::pair<const NuvemPontos &, DefinicaoMalha>
estimaDefinicaoMalha(const NuvemPontos &);

MalhaRegular
calculaTriangulacaoAkima(const NuvemPontos &,
                         const DefinicaoMalha &,
                         double raioPesquisa);
```

Quadro 4 - Assinaturas das tarefas do fluxo de criação de mapa de propriedade.

A tarefa `selecionaPerfis()` recebe um identificador do tipo de perfil desejado (por exemplo, o nome do perfil) e retorna os perfis desse tipo encontrados nos poços de um grupo de poços.

A tarefa `calculaMediaAritmetica()` recebe um conjunto de perfis de poços e calcula, para cada perfil, a média dos seus valores. O resultado dessa tarefa é uma nuvem de pontos onde cada ponto corresponde às coordenadas do poço de cada perfil recebido como entrada. Cada ponto tem associado o valor da média calculado no perfil daquele poço (no nosso caso, a média da densidade).

As tarefas de geração de malhas, dentre elas `calculaTriangulacaoAkima()`, recebem como entrada a definição da malha a ser gerada. Essa definição pode ser informada explicitamente pelo usuário, lida de outro objeto de malha ou estimada automaticamente pela aplicação. Neste último caso, usamos a tarefa `estimaDefinicaoMalha()`, que recebe a nuvem de pontos que será usada na geração da malha e que retorna a definição de malha estimada. Como essa tarefa é sempre usada para fornecer as entradas para a tarefa de geração de malha, a tarefa de estimação também retorna a própria nuvem de pontos usada como entrada, de forma a simplificar a composição das duas tarefas.

A composição desse fluxo é feita de seguinte forma:

```
FluxoTrabalho fluxo;
std::vector<FluxoTrabalho::Posicao> saidasTarefa;

// Composição de perfis:
auto posPocos = fluxo.insereEntrada(typeid(std::vector<Poco>), "Poços");
auto posGrupoPerfis = fluxo.insereEntrada(typeid(std::string),
                                          "Identificador de perfis");
saidasTarefa = fluxo.insereTarefa(selecionaPerfis,
                                  {posPocos, posGrupoPerfis});


// Filtros:
auto posZona = fluxo.insereEntrada(typeid(Zona), "Zona");
saidasTarefa = fluxo.insereTarefa(filtraPerfisPorZona,
                                  {saidasTarefa[0], posZona});

auto posRochas = fluxo.insereEntrada(typeid(std::vector<Rocha>), "Rochas");
saidasTarefa = fluxo.insereTarefa(filtraPerfisPorLitologia,
                                  {saidasTarefa[0], posRochas});


// Cálculo da média:
saidasTarefa = fluxo.insereTarefa(calculaMediaAritmetica, {saidasTarefa[0]});


// Gridagem:
saidasTarefa = fluxo.insereTarefa(estimaDefinicaoMalha, {saidasTarefa[0]});

auto posRaioPesquisa = fluxo.insereEntrada(typeid(double), "Raio de pesquisa");
saidasTarefa = fluxo.insereTarefa(calculaTriangulacaoAkima,
                                  {saidasTarefa[0],
                                   saidasTarefa[1],
                                   posRaioPesquisa});


// Valores para os parâmetros de entrada:
std::vector<Valor> valores;
valores.push_back( {pc01, pc02, ...} );
valores.push_back( "DENS" );
valores.push_back( zn01 );
valores.push_back( std::vector<Rocha>{areia, argila} );
valores.push_back( 120.0 );


// Adição no modelo:
modeloGeologico.criaObjetoFluxo("Mapa média DENS",
                                fluxo,
                                std::move(valores),
                                saidasTarefa[0]);
```

Quadro 5 - Pseudocódigo da composição do fluxo de criação do mapa de média da densidade de uma região a partir dos valores medidos na perfuração de poços.

5.1.2. Cenário 2 – Cálculo de perfis a partir de uma regressão

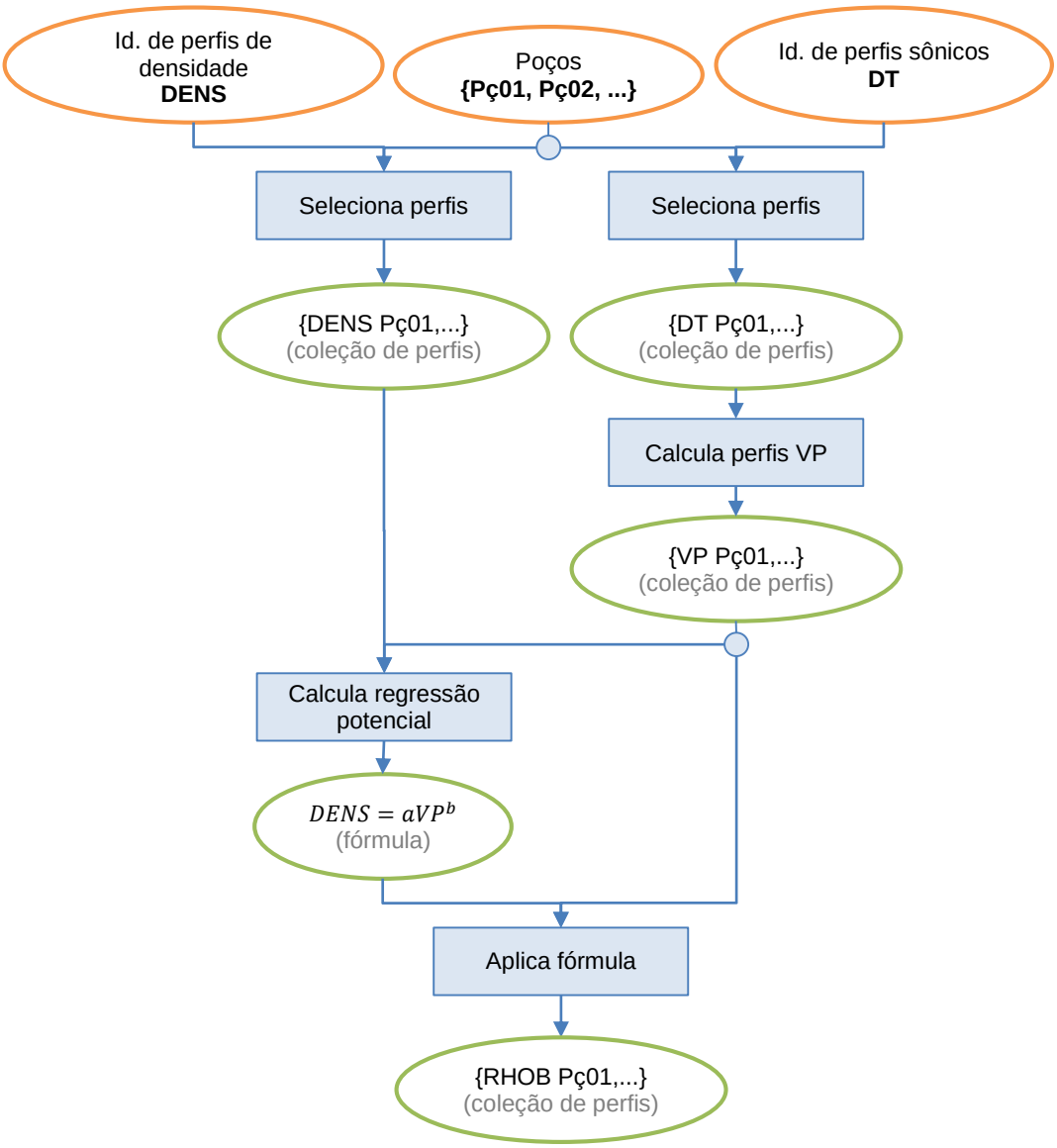

Figura 33 - Esquema do fluxo de cálculo de perfis em poços a partir de uma regressão entre outros dois perfis.

Neste cenário, o usuário deseja estimar a densidade das rochas ao longo da trajetória de um poço a partir dos valores do perfil sônico desse poço. Os perfis sônicos (DT ou Δt) registram o intervalo de tempo de uma onda sísmica na rocha. Desse intervalo de tempo, é possível calcular a velocidade da onda sísmica na rocha. Dessa velocidade da onda, por sua vez, é possível estimar a densidade da rocha.

Assim, o usuário seleciona os demais poços da região em que os perfis de densidade e sônico sejam conhecidos, calcula os perfis de velocidade da onda sísmica (VP) e correlaciona esses perfis de velocidade com os perfis de densidade desses poços através de uma regressão potencial. Essa regressão

determina os coeficientes da correlação que são usados, então, para calcular a densidade a partir dos perfis de velocidade (VP) dos poços.

As tarefas desse fluxo são representadas no código da seguinte forma:

```
std::vector<Perfil>
selecionaPerfis(const std::vector<Poco> &, const std::string &);

std::vector<Perfil>
calculaPerfisVp(std::vector<Perfil>);

Regressao
calculaRegressaoPotencial(const std::vector<Perfil> &,
                          const std::vector<Perfil> &);

std::vector<Perfil>
aplicaFormula(const Formula &, std::vector<Perfil>);
```

Quadro 6 - Assinaturas das tarefas do fluxo do cálculo de perfis a partir de uma regressão.

A composição desse fluxo é feita de seguinte forma:

```
FluxoTrabalho fluxo;
std::vector<FluxoTrabalho::Posicao> saidasTarefa;

// Composição de perfis:
auto posPocos = fluxo.insereEntrada(typeid(std::vector<Poco>), "Poços");

auto posGrupoPerfisDens = fluxo.insereEntrada(typeid(std::string),
                                              "Id. de perfis de densidade");
saidasTarefa = fluxo.insereTarefa(selecionaPerfis,
                                  {posPocos, posGrupoPerfisDens});
const auto posPerfisDens = saidasTarefa[0];

auto posGrupoPerfisDt = fluxo.insereEntrada(typeid(std::string),
                                            "Id. de perfis sônicos");
saidasTarefa = fluxo.insereTarefa(selecionaPerfis,
                                  {posPocos, posGrupoPerfisDt});


// Cálculo VP:
saidasTarefa = fluxo.insereTarefa(calculaPerfisVp, {saidasTarefa[0]});
const auto posPerfisVp = saidasTarefa[0];


// Cálculo regressões:
saidasTarefa = fluxo.insereTarefa(calculaRegressaoPotencial,
                                  {saidasTarefa[0] , posPerfisDens});


// Cálculo do RHOB:
saidasTarefa = fluxo.insereTarefa(aplicaFuncao,
                                  {saidasTarefa[0], posPerfisVp});


// Valores para os parâmetros de entrada:
std::vector<Valor> valores;
valores.push_back( {pc01, pc02, ...} );
valores.push_back( dens );
valores.push_back( dt );


// Adição no modelo:
modeloGeologico.criaObjetoFluxo("Perfis RHOB",
                                fluxo,
                                std::move(valores),
                                saidasTarefa[0]);
```

Quadro 7 – Pseudocódigo da composição do fluxo de cálculo de perfis a partir de uma regressão.

Como dito na 0, ao adicionarmos a segunda tarefa de `selecionarPerfis` para os perfis DT e a tarefa de `aplicaFuncao`, que usam como entrada uma posição que já é usada como entrada de outra tarefa, o método `insereTarefa()` adiciona automaticamente transições internas na rede de Petri que representa o fluxo para fazer a cópia dos marcadores dessas posições de entrada.

5.2. Suporte a funcionalidades de histórico e automação

A seguir, mostraremos três novas funcionalidades que podem ser desenvolvidas baseadas nos fluxos de trabalho que geram os objetos do modelo geológico para atender às demandas de maior automação e registro de proveniência do modelo: A visualização do histórico de operações que gerou cada objeto, a notificação de objetos desatualizados e automação do seu reprocessamento e, por último, o suporte a múltiplas parametrizações para a execução de um fluxo.

5.2.1. Visualização de histórico

O armazenamento dos fluxos de trabalho que geram os objetos do modelo geológico permite que adicionemos uma funcionalidade para o usuário ver o histórico de cada objeto do modelo. Assim, quando o usuário escolhe ver o histórico de um objeto do modelo, a aplicação recupera a execução do fluxo de trabalho registrada no modelo que gerou aquele objeto, monta uma representação gráfica do fluxo de trabalho (similar às figuras Figura 31 e Figura 33) e os valores dos parâmetros usados como entrada e a exibe para o usuário.

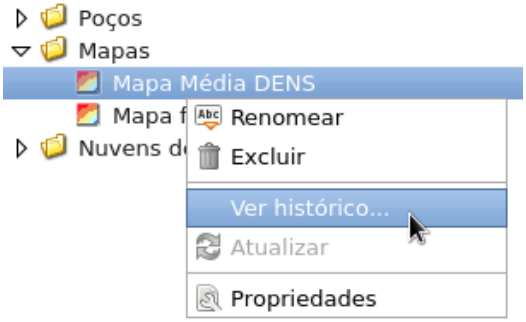

Figura 34 - Visualização do histórico de operações aplicadas em um objeto do modelo geológico.

Na nossa aplicação, usamos a biblioteca de grafos da Boost[19] para montar a rede de Petri usada internamente na classe de fluxo de trabalho. Para criar uma imagem a partir desse fluxo para exibirmos ao usuário, usamos o transformador para o formato Graphviz[20] disponibilizado pela biblioteca da Boost e, então, usando a biblioteca Graphviz, gerar uma imagem SVG[21] do fluxo, que exibimos para o usuário.

[19] BGL – Boost Graph Library (http://www.boost.org/doc/libs/release/libs/graph/).
[20] Biblioteca de código aberto para visualização de grafos (http://www.graphviz.org/).
[21] *Scalable Vector Graphics*: Formato de imagens vetoriais desenvolvido pela W3C – *World Wide Web Consortium* (http://www.graphviz.org/).

5.2.2. Notificação de resultado desatualizado e automação do reprocessamento

Guardar o fluxo de trabalho que gerou cada objeto do modelo também nos permite monitorar os objetos usados como entrada do fluxo para, quando houver alguma modificação neles, notificar o usuário de que o resultado do fluxo precisa ser reprocessado para que os seus resultados sejam atualizados e reflitam as mudanças nos objetos de entrada.

Para isso, toda vez que um objeto do modelo é modificado, podemos buscar no modelo geológico todas as execuções de fluxo registradas que usam esse objeto como entrada, marcando aquela entrada do fluxo como “suja” e mostramos um ícone ao lado do objeto na interface da aplicação para indicar que ele está desatualizado. O usuário pode, então, ver quais objetos de entrada foram modificados e disparar a atualização do objeto resultante do fluxo. A aplicação, por fim, reexecuta o fluxo de trabalho e atualiza os objetos resultantes no modelo.

Abaixo mostraremos um exemplo de como a aplicação se comportaria em relação ao mapa de média de densidade gerado pelo fluxo da figura 31 caso o usuário faça uma alteração na definição da zona usada na filtragem dos perfis (zona `Zn01`):

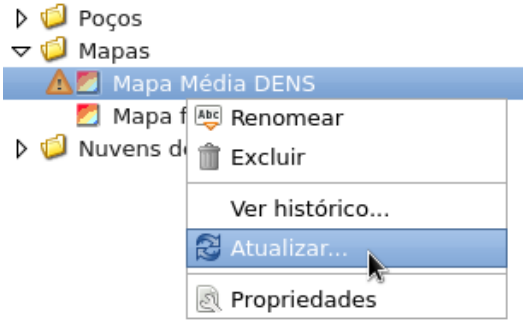

Figura 35 - Exemplo de objeto do modelo geológico com sinalização indicando que ele está desatualizado.

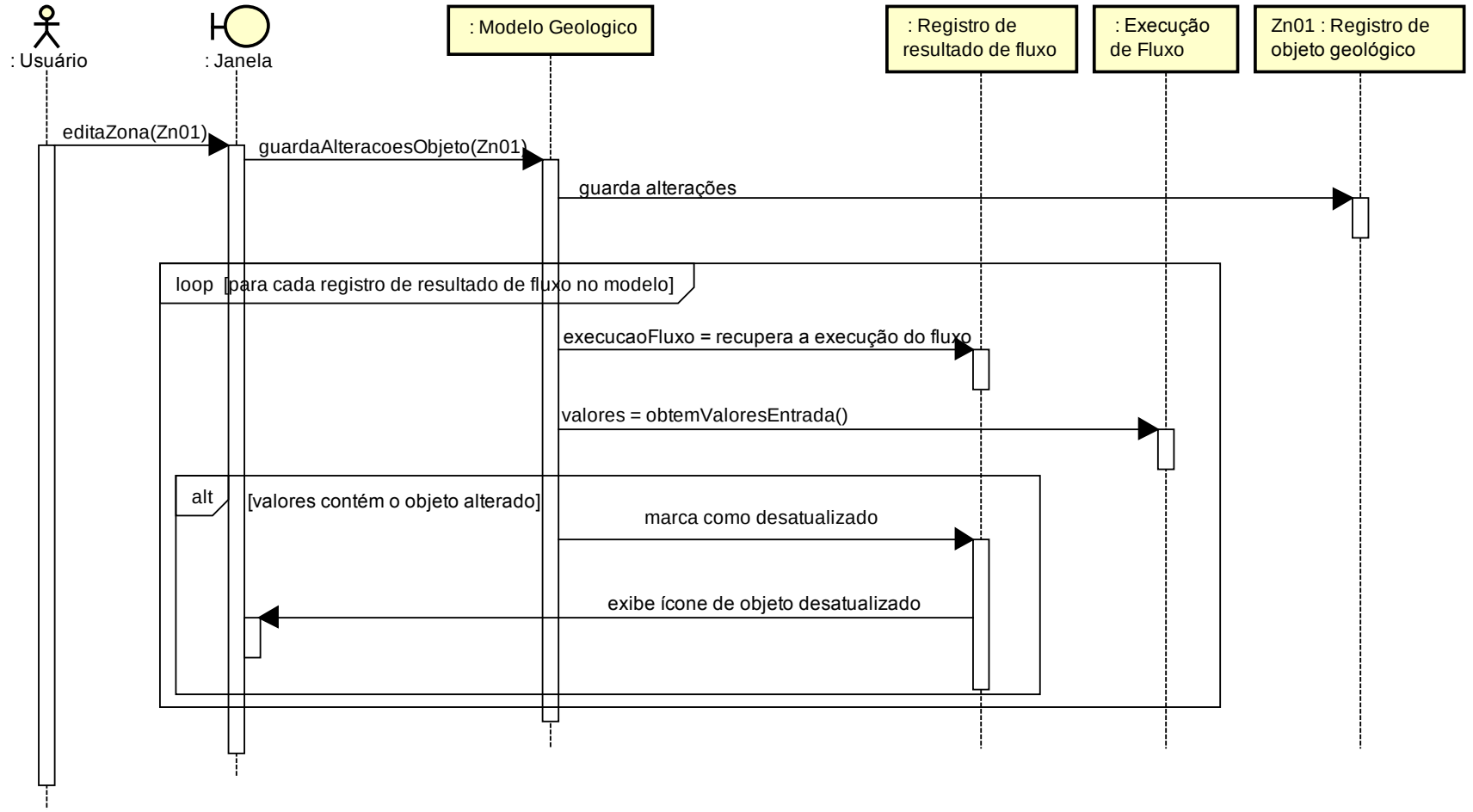

Figura 36 – Diagrama de sequência de exemplo da lógica de marcação de objetos do modelo geológico como desatualizados quando o valor de um objeto usado como parâmetro de entrada do fluxo de trabalho que o gerou é alterado.

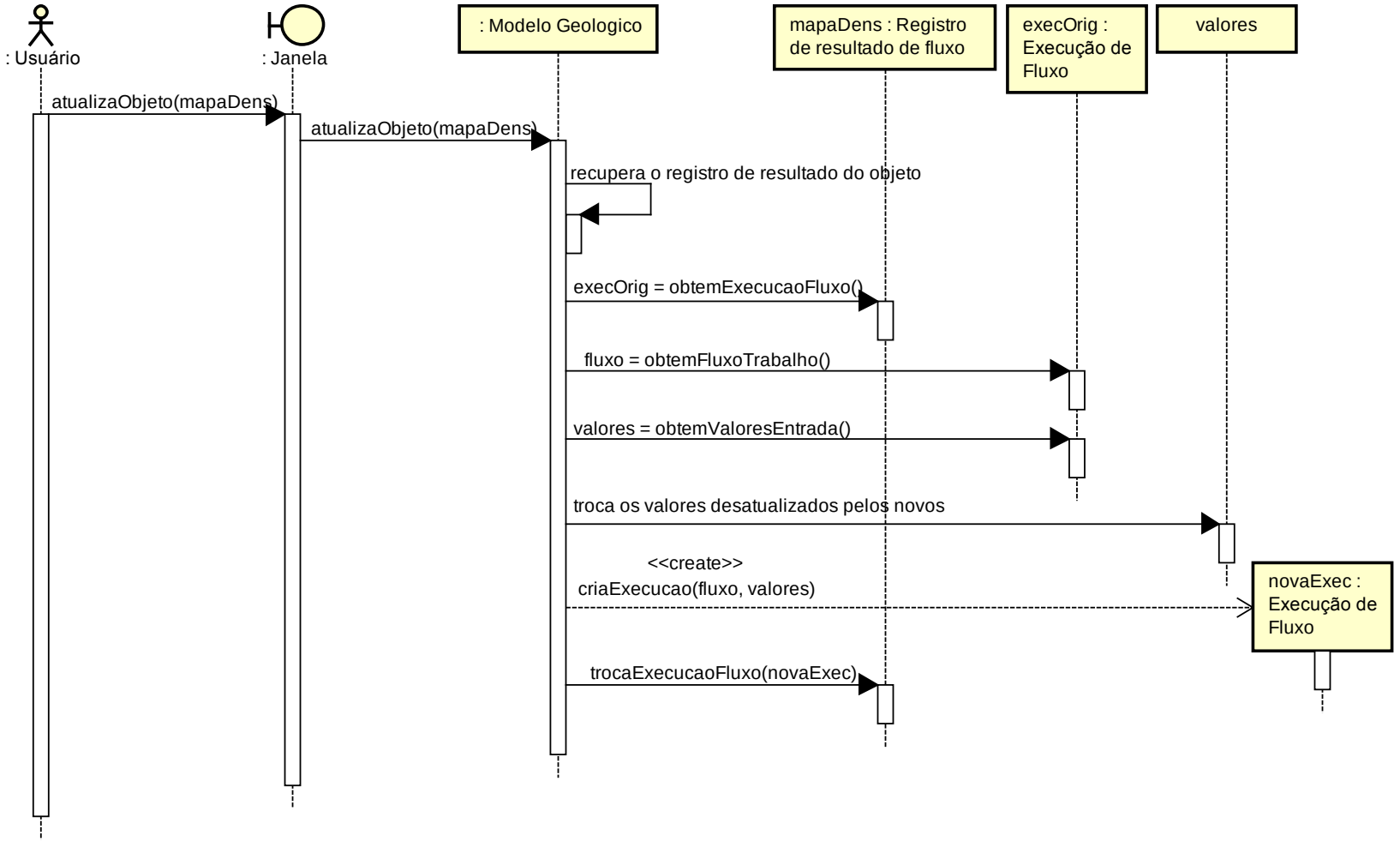

Figura 37 - Diagrama de sequência de exemplo da lógica de atualização de um objeto resultado de uma execução de fluxo de trabalho.

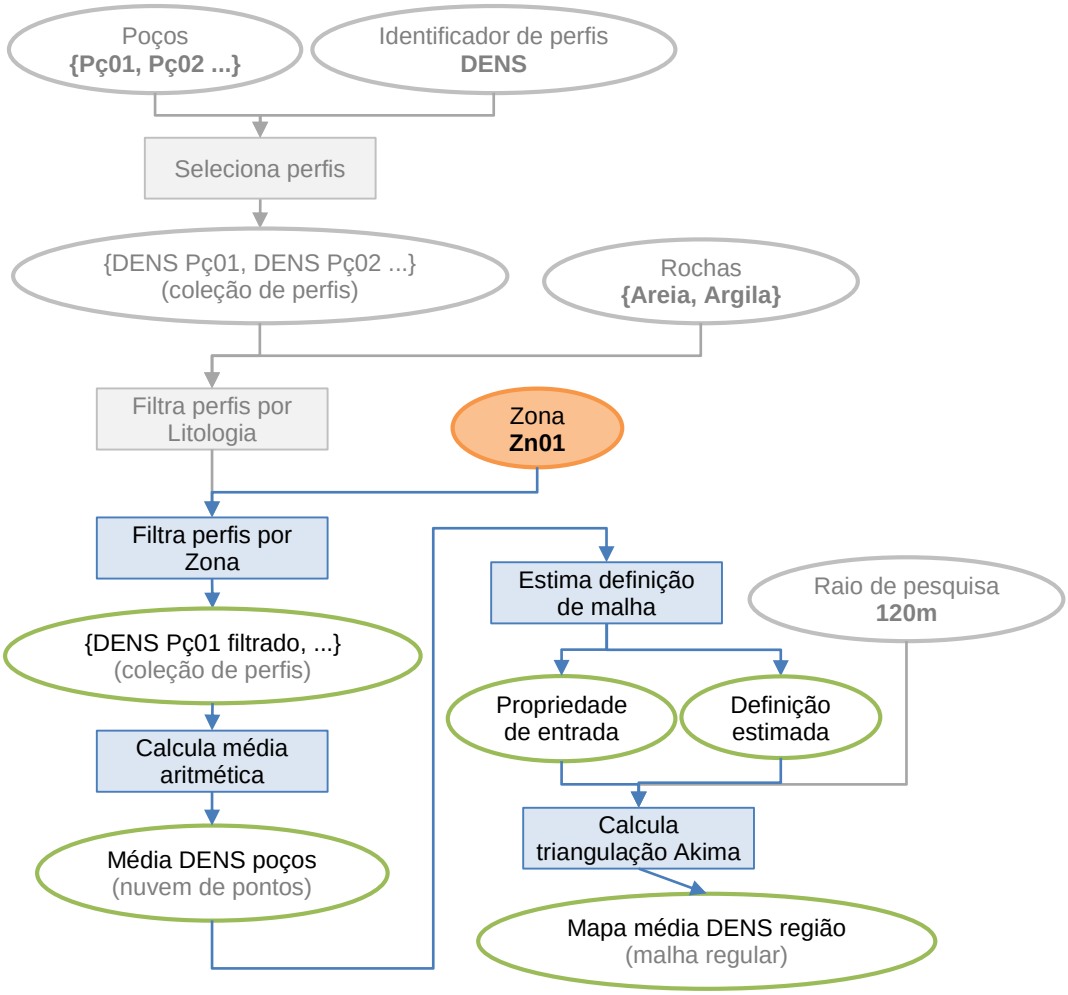

Figura 38 - Exemplo de visualização de uma execução de fluxo de trabalho desatualizada. O parâmetro de entrada cujo valor foi alterado é destacado, assim como quais tarefas serão reeecutadas para atualizar os resultados. As tarefas de filtro foram reordenadas em relação à figura 31 apenas para destacar os parâmetros de entrada não alterados e as tarefas que não precisariam ser refeitas (em cinza).

5.2.3. Execução de um fluxo com múltiplas configurações de parâmetros

Armazenar os fluxos de trabalho também viabiliza a criação de uma funcionalidade para permitir ao usuário executar um mesmo fluxo de trabalho usando diferentes combinações de valores para os seus parâmetros de entrada. Em uma aplicação geocientífica, isso é especialmente útil quando o usuário deseja repetir uma mesma sequência de operações em diferentes regiões do modelo, tal como em diferentes zonas de um poço ou para diferentes tipos de rocha. A aplicação, então, executa automaticamente o fluxo de trabalho para cada combinação de parâmetros definida pelo usuário, evitando que ele tenha que fazê-lo manualmente.

Assim, podemos adicionar no diálogo de visualização de histórico de um objeto do modelo uma opção para o usuário reexecutar aquele fluxo de trabalho para diferentes dados de entrada. Essa opção abre um novo diálogo que permite ao usuário especificar conjuntos de valores diferentes para cada parâmetro de entrada do fluxo. Conforme o usuário seleciona os conjuntos de valores para cada parâmetro de entrada, a aplicação gera todas as diferentes combinações desses parâmetros e lista para o usuário.

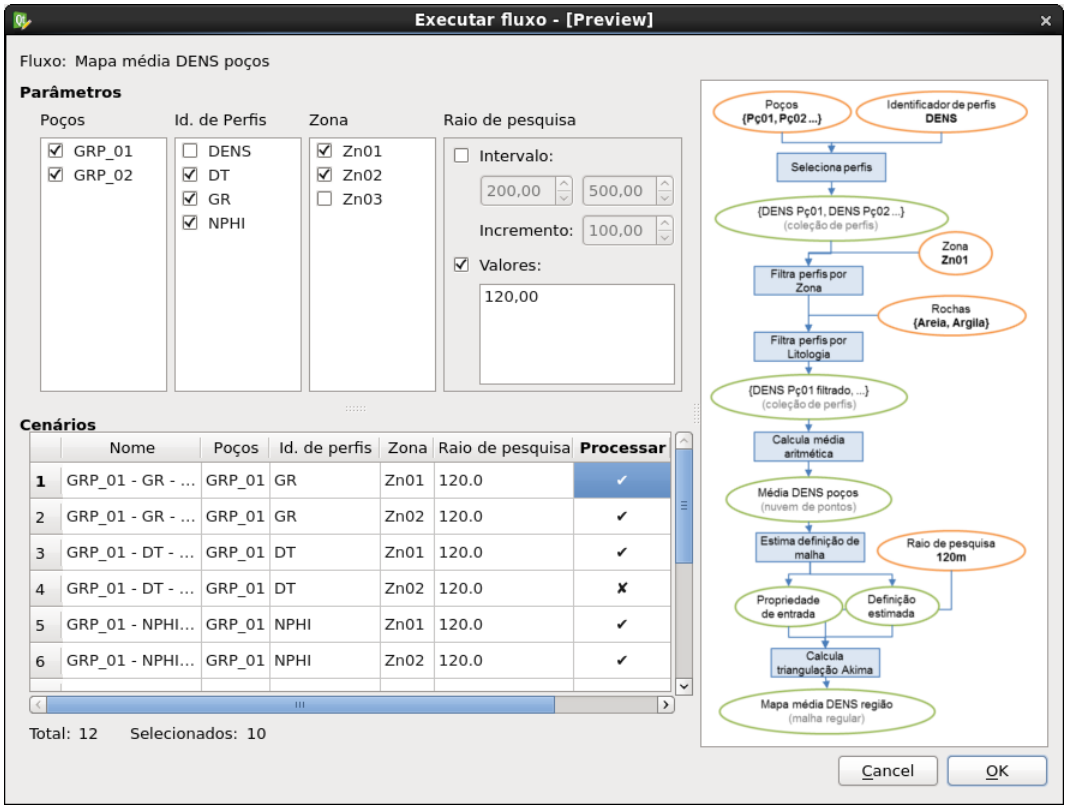

Figura 39 - Diálogo de execução de um fluxo de trabalho com múltiplos parâmetros de entrada.

Esse diálogo de execução com múltiplas configurações de parâmetros é criado pela aplicação dinamicamente a partir da execução de fluxo que o usuário selecionou: a aplicação varre os parâmetros de entrada do fluxo e adiciona componentes de interface para a entrada de seus valores. O componente de interface adicionado varia com o tipo do parâmetro de entrada. A figura 40 ilustra dois exemplos:

1. Para entradas que representam objetos do modelo, é exibida a lista de objetos daquele tipo (zonas, tipos de perfis etc.) para o usuário selecionar quais ele deseja usar;
2. Para tipos numéricos, são exibidos campos para a definição de intervalos de valores (valor mínimo, valor máximo e incremento).

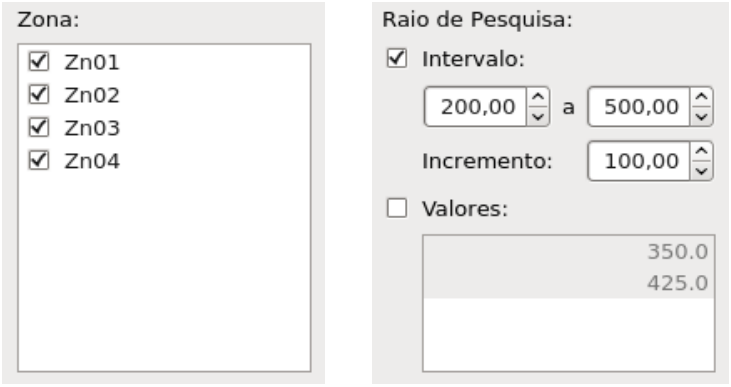

Figura 40 - Exemplos de componentes para entrada de dados de diferentes tipos: uma lista para seleção de objetos do tipo `Zona` registrados no modelo e campos para entrada de intervalos valores numéricos do tipo `double`.

Uma vez que o usuário termine de configurar as combinações de parâmetros que ele deseja e confirma o diálogo, a aplicação cria no modelo geológico um novo objeto para cada combinação expressa pelo usuário, registrando-o no modelo como o resultado do fluxo de trabalho selecionado pelo usuário e a combinação de valores dos parâmetros de entrada gerada.

A estrutura de um diálogo de execução de um fluxo de trabalho com múltiplas configurações de parâmetros é resumida no diagrama de classes da figura 42. Também mostramos dois diagramas de sequência que resumem a lógica de construção de uma instância desse diálogo para um dado fluxo (figura 42) e a lógica da geração das múltiplas configurações de execuções desse fluxo a partir da combinação dos valores de entrada especificados pelo usuário para cada parâmetro (figura 43).

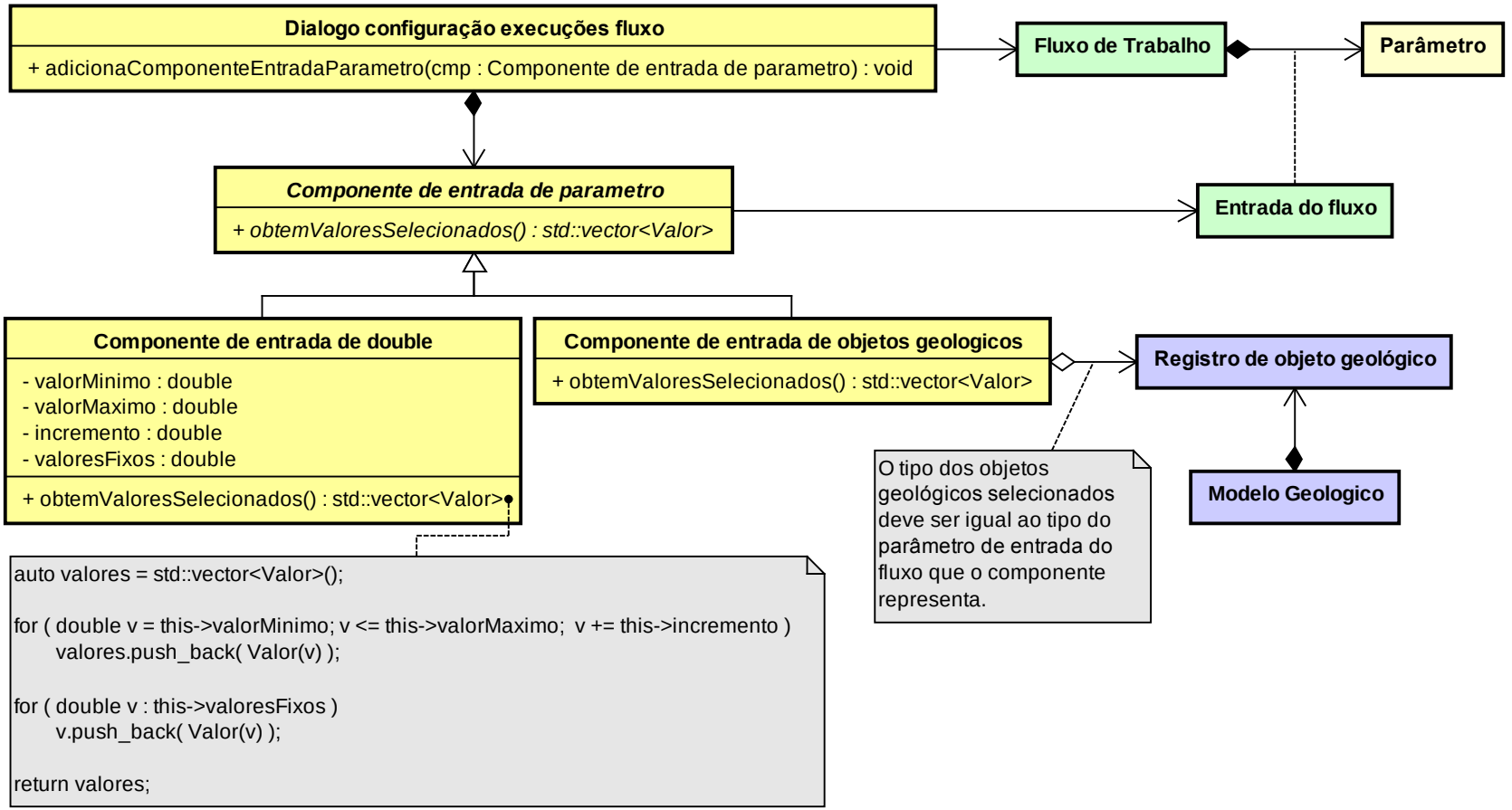

Figura 41 - Diagrama de classes das classes de diálogo de configuração de execuções de um fluxo de trabalhos usando múltiplas combinações de valores para seus parâmetros de entrada. A quantidade de classes de componente de entrada de parâmetros pode variar de acordo com os tipos parâmetros nos fluxos de trabalho de cada aplicação.

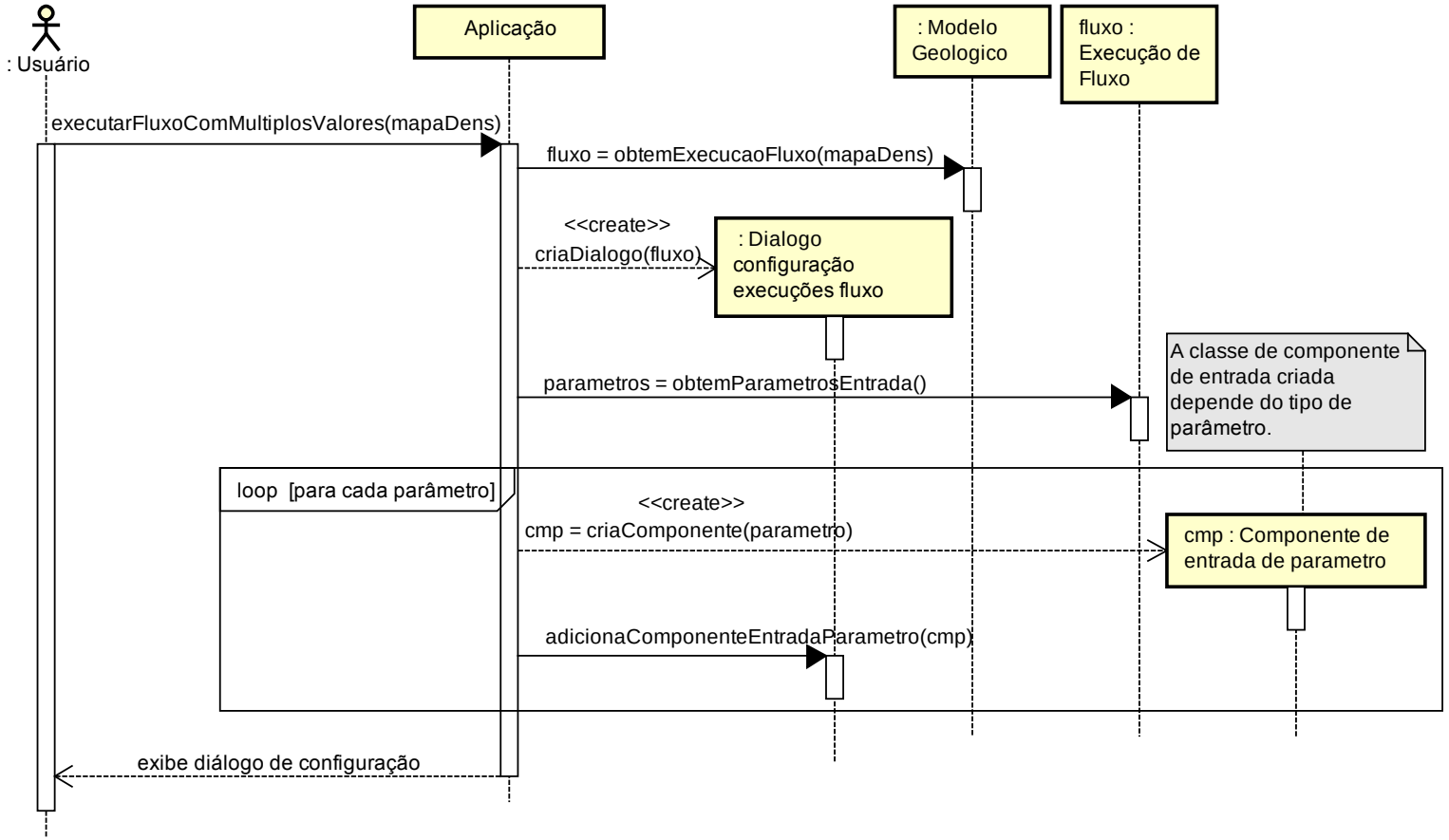

Figura 42 - Diagrama de sequência de exemplo da criação do diálogo de configuração de execuções de um fluxo de trabalho.

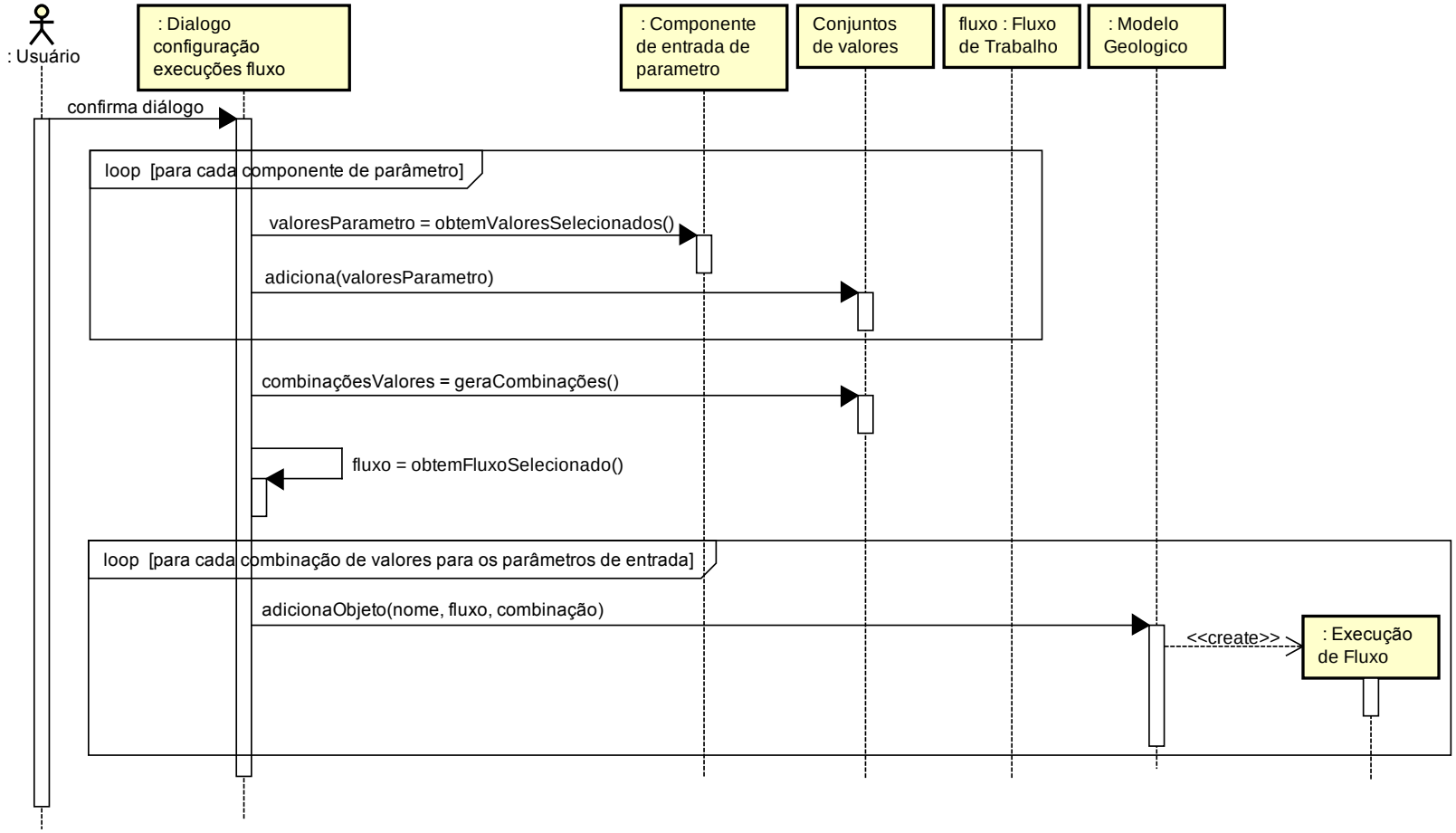

Figura 43 - Diagrama de sequência de exemplo da montagem das execuções do fluxo de trabalho para cada combinação de valores de parâmetros selecionados pelo usuário.

6 Conclusões

Nesta seção, sintetizaremos as contribuições desta dissertação para a área de fluxos de trabalho científicos e apresentaremos trabalhos futuros que podem ser derivados a partir desta dissertação.

6.1. Contribuições

Neste trabalho, mostramos que é viável usar fluxos de trabalho em uma aplicação científica existentes da área de geologia para representar os modelos geológicos gerados pelos usuários. Em especial, mostramos que essa abordagem facilita o desenvolvimento de novas funcionalidades na aplicação para atender as demandas dos usuários por maior registro de proveniência e automação na aplicação, tais como a visualização do histórico de objetos e a execução automática de fluxos em casos de alterações nos parâmetros de entrada ou com diferentes combinações de parâmetros.

Acreditamos que o uso de fluxos de trabalho como elemento para apoiar a reengenharia e o desenvolvimento de novas funcionalidades em aplicações científicas, tal como fizemos nesse trabalho, ainda é pouco explorado em trabalhos na área de *workflows* científicos. A maior parte dos trabalhos e sistemas de *workflows* científicos que encontramos, sendo voltada para integração de serviços computacionais científicos distribuídos, acaba sendo de difícil aplicação em sistemas científicos mais simples, tal como em aplicações *stand-alone* como a que estudamos durante este trabalho. Nesse sentido, nós mostramos um novo cenário de uso de fluxos de trabalho científicos que pode vir a ser mais explorada em trabalhos futuros.

Nós também identificamos os componentes fundamentais de um motor de fluxos de trabalho que são necessários para o uso de fluxos de trabalho em aplicações científicas mais simples. Mostramos, então, como esses componentes podem ser implantados em uma aplicação C++ usando redes de Petri para representar os fluxos de trabalho e funções C++ para representar suas tarefas. Esses modelos poderão ser usados como guia para a implantação de fluxos de trabalho em outras aplicações científicas similares.

Por fim, analisamos os principais pontos de mudança em uma aplicação existente com a introdução de fluxos de trabalho, tais como os controladores da

aplicação, que devem passar a compor e registrar fluxos de trabalho, e as funções da aplicação, que podem precisar de mudanças nas assinaturas ou mesmo ser decompostas em funções menores para ser usadas como tarefas de um fluxo de trabalho. Essa nossa experiência pode ser usada como base no projeto ou na reengenharia de outras aplicações científicas, ajudando a identificar os componentes da aplicação que podem precisar de revisão para que a aplicação passe a registrar os fluxos de trabalho do usuário.

6.2. Trabalhos futuros

A seguir, apresentaremos alguns trabalhos futuros que propomos a partir da introdução de fluxos de trabalho em uma aplicação científica.

6.2.1. Proveniência dos dados interpretados

O modelo de fluxos de trabalho que abordamos nesta dissertação facilita o registro de sequências de cálculos e transformações de dados, tornando-os ideais, por exemplo, para a representação das etapas de predição em aplicações de apoio ao processo de caracterização geológica. Em aplicações científicas que possuem recursos de desenho (*CAD*), porém, fluxos de trabalho não são capazes de registrar por conta própria a origem dos objetos criados a partir de ferramentas de desenho. No caso de uma aplicação de apoio ao processo de caracterização geológica (descrito na seção 1.1), fluxos de trabalho não registrariam, assim, os objetos gerados na etapa de interpretação de feições geológicas, mas apenas o seu uso como entrada para controle geológico nas operações de cálculo da etapa de predição. A figura abaixo destaca essa situação:

Figura 44 – Suporte de fluxo de trabalhos para registro de proveniência nas etapas do processo geológico.

Um exemplo dessa limitação pode ser visto no cenário de geração de mapa de uma propriedade que apresentamos na seção 5.1.1: na tarefa de filtrar perfis por uma zona, o fluxo de trabalho enxerga a zona delimitada nos poços na etapa de interpretação apenas como um dado bruto de entrada. Uma zona em um poço, porém, é delimitada manualmente pelo usuário a partir da análise dos perfis e demais dados desses poços em uma visualização gráfica (similar à mostrada na figura 32). Para registrar a proveniência da zona, a aplicação precisaria, assim, registrar todo o contexto de interpretação do usuário no momento em que ele fez o desenho da zona, o que incluiria os dados de poço que o usuário estava visualizando no momento da marcação e que podem o ter levado a escolher aquela posição do poço para delimitar aquela zona (tal como os perfis exibidos na figura 32).

Assim, enquanto fluxos de trabalho são suficientes para o registro de proveniência dos objetos gerados a partir de cálculos, é necessário estudar mecanismos complementares para o registro da proveniência dos dados interpretados (desenhados) pelo usuário em aplicações que possuam suporte a esse tipo de entrada livre de dado. Esses mecanismos podem variar desde campos de anotação preenchidos manualmente pelo usuário até a captura automática do contexto da aplicação no momento em que a interpretação ou o desenho foi criado (tal como citamos no exemplo da zona anteriormente).

Note que para termos as informações completas de proveniência de um modelo geológico é necessário também o registro da proveniência dos dados de prospecção usados como entrada para a construção do modelo. Essas informações consistem em detalhes de como a prospecção foi feita (ferramentas usadas, condições do ambiente durante a realização das medições etc.) e como elas foram integradas e armazenadas em bases de dados corporativas ou externas. Isso, porém, está além da responsabilidade de uma aplicação geocientífica, que é apenas consumidora dos dados disponibilizados nessas bases.

6.2.2. Tratamento de etapas manuais

Algumas aplicações científicas podem ter cenários em que, durante a execução de uma sequência de operações, o usuário precisa fazer alguma decisão ou intervenção manual para determinar como prosseguir a análise que ele está fazendo. Um exemplo comum na área de geociências são casos em que o usuário precisa estimar uma propriedade (por exemplo, o perfil de pressão em um grupo de poço) para usar em sua análise, e essa estimativa pode ser feita

usando dois métodos de cálculo diferentes. Nessas situações, em geral o usuário faz a estimativa usando os dois métodos diferentes, faz uma comparação qualitativa dos resultados gerados por cada método e por fim escolhe qual deles ele considera mais adequado para ser usado no restante da análise. Esse cenário é ilustrado na figura abaixo:

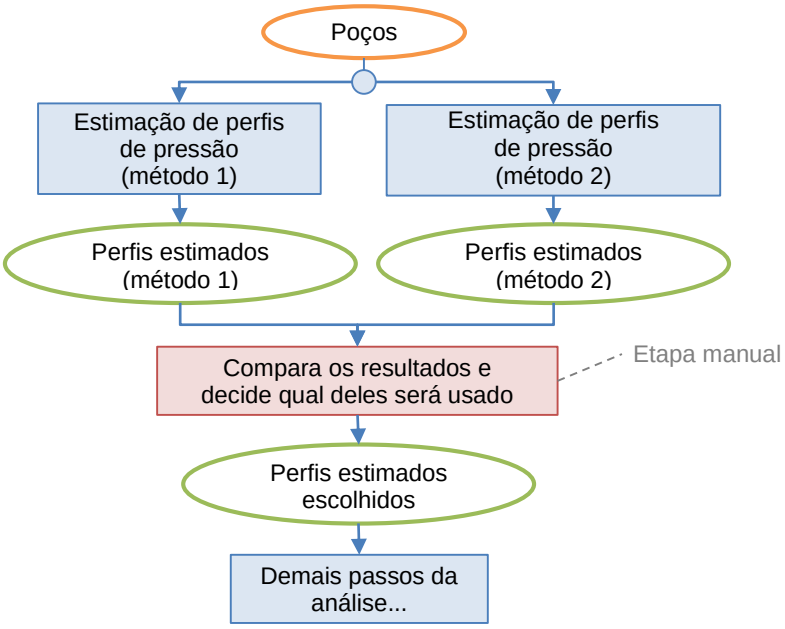

Figura 45 - Exemplo de fluxo de trabalho com uma etapa de escolha manual feita pelo usuário.

Nesta dissertação, tratamos apenas fluxos de trabalho com tarefas que representam cálculos ou processamentos feitos automaticamente pela aplicação. A introdução de tarefas manuais exige a extensão do modelo de motor de fluxo de trabalho que propomos para incluir mecanismos para especificar tarefas que possuem interação com o usuário. Também é necessário definir mecanismos para que durante a execução de um fluxo, quando uma tarefa manual for disparada, o motor de fluxo possa notificar ao usuário a ação que ele tem que tomar para que a execução do fluxo possa continuar.

6.2.3. Impacto em outras disciplinas do processo de desenvolvimento

Neste trabalho, analisamos o impacto do uso de fluxos de trabalho no nível de implementação da aplicação. Acreditamos, porém, que o uso de fluxos de trabalho pode trazer vantagens em outras disciplinas do processo de desenvolvimento.

Na disciplina de **requisitos**, muitos usuários da nossa aplicação já costumam fazer espontaneamente desenhos ou esquemas similares a fluxos

para representar o trabalho que eles gostariam de realizar na aplicação. Esses esquemas, porém, são feitos de forma livre, sem haver uma notação ou um método padronizado, o que faz com que nem sempre eles contenham todas as informações necessárias para o levantamento dos requisitos. Assim, acreditamos que fluxos de trabalho poderiam ser explorados como uma ferramenta de comunicação ou mesmo de especificação de requisitos para a aplicação e dos processos científicos (no nosso caso, de análise geológica) que ela deve suportar. Alguns trabalhos de geociências já exploram informalmente o uso de notações similares a fluxos de trabalho para comunicação, tal como (PYRCZ; DEUTSCH, 2014). Explorar mais a fundo o uso desses conceitos de fluxos de trabalho como ferramenta de comunicação é interessante, inclusive, pelo fato de o usuário aplicar a mesma linguagem no levantamento de requisitos que ele já estará usando na aplicação (por exemplo, ao ver o histórico de um objeto).

Já na disciplina de **testes**, acreditamos que as ferramentas de composição de fluxos de trabalho poderiam ser usadas pela equipe de desenvolvimento como forma de especificar e executar testes automatizados da aplicação de forma mais simples, inclusive via uma interface gráfica de composição de fluxos. Isso pode facilitar e acelerar tanto a construção quanto a manutenção desses testes, que hoje é feita via código.

Fluxos de trabalho também podem ser explorados como ferramenta de auxílio para a **paralelização** da aplicação. Diversos trabalhos, tais como (CHEN; JOHNSON, 2013) e bibliotecas como Intel Threading Building Blocks[22], propõem o uso de estruturas de fluxos de dados ou grafos de dependências para representar paralelismo de trechos de código. Da mesma forma, motores de fluxos de trabalho podem ser projetados para executar as tarefas de um fluxo de forma paralela, fazendo as sincronizações necessárias entre elas de acordo com as ligações entre elas. Um motor de fluxos também pode permitir a execução simultânea de múltiplos fluxos de trabalho. Isso reduz a necessidade de os desenvolvedores fazerem esse controle de paralelismo entre tarefas manualmente no restante do código.

6.2.4. Transferência de conhecimento e recomendações

O registro das análises ou experimentos realizados pelos usuários de uma aplicação científica na forma de fluxos de trabalho, além de possibilitar que essas análises ou experimentos sejam facilmente compartilhados com outros

[22] http://www.threadingbuildingblocks.org/

usuários e cientistas, também pode ser explorado para criação de acervos de fluxos de trabalho que podem ser consultados e usados como base por outros usuários na construção de novos modelos geológicos.

Uma vez que a aplicação científica tem à disposição um acerto de fluxos de trabalho, podemos utilizar técnicas de mineração e descoberta de dados[23] para identificar tendências e padrões nos fluxos de trabalho criados pelos usuários de acordo. No caso de uma aplicação da área de geologia, poderíamos detectar padrões de fluxos de trabalho em relação à região sendo estudada ou a características dos dados usados como entrada.

Com base nos padrões e tendências identificados no acervo de fluxos da aplicação, podemos implantar funcionalidades de recomendação para auxiliar os usuários na criação de novos modelos geológicos. Conforme o usuário for selecionando os dados de entrada que ele deseja usar e disparando as operações na aplicação para fazer a análise ou experimento que ele deseja, a aplicação pode recomendar dados adicionais de entrada, valores a serem usados como parâmetros em uma operação ou mesmo fluxos de trabalho inteiros que outros usuários usaram em modelos geológicos similares.

A implantação de recursos de recomendação é de especial interesse em aplicações na área de geologia justamente devido ao grande número de técnicas de análise e composição de modelos geológicos que o usuário tem a disposição. Alguns sistemas de recomendação baseados em agentes de *software* (WOOLDRIDGE, 1997) já foram propostos para auxiliar em etapas do processo de caracterização geológica, como o mapeamento de superfícies em um volume. (GALLIMORE et al., 1999) Nesses sistemas, por exemplo, as informações de acervo de fluxos de trabalho poderiam ser usadas como insumos pelos agentes para determinar a estratégia de recomendação que eles adotarão.

[23] Mineração e descoberta de dados são técnicas usadas para extrair significados, informações e conhecimento a partir de um conjunto de dados. Em geral, consistem na identificação de padrões nos dados de uma ou mais bases de dados. Para mais informações, veja (FAYYAD; PIATETSKY-SHAPIRO; SMYTH, 1996).

7 Referências

CALLAHAN, S. P. et al. **VisTrails: visualization meets data management**Proceedings of the 2006 ACM SIGMOD international conference on Management of data - SIGMOD '06. **Anais**...New York, New York, USA: ACM Press, 2006Disponível em: <http://portal.acm.org/citation.cfm?doid=1142473.1142574>. Acesso em: 17 nov. 2014

CHEN, N.; JOHNSON, R. E. **JFlow: Practical refactorings for flow-based parallelism**2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE). **Anais**...IEEE, nov. 2013Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6693080>

CURCIN, V.; GHANEM, M. **Scientific workflow systems - can one size fit all?**2008 Cairo International Biomedical Engineering Conference. **Anais**...IEEE, dez. 2008Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4786077>

DEELMAN, E. et al. Workflows and e-Science: An overview of workflow system features and capabilities. **Future Generation Computer Systems**, v. 25, n. 5, p. 528–540, maio 2009.

DEELMAN, E.; GANNON, D.; SHIELDS, M. S. **Workflows for e-Science**. London: Springer London, 2007.

FAYYAD, U.; PIATETSKY-SHAPIRO, G.; SMYTH, P. From Data Mining to Knowledge Discovery in databases. **AI MAGAZINE**, v. 17, n. 3, p. 37–54, 1996.

GALLIMORE, R. J. et al. Cooperating agents for 3-D scientific data interpretation. **IEEE Transactions on Systems, Man and Cybernetics, Part C (Applications and Reviews)**, v. 29, n. 1, p. 110–126, 1999.

GAMMA, E. et al. **Padroes de projeto: soluções reutilizáveis de software orientado a objetos**. Porto Alegre: Bookman, 2000.

GARIJO, D.; GIL, Y.; REY, M. **A new approach for publishing workflows: abstractions, standards, and linked data**Proceedings of the 6th workshop on Workflows in support of large-scale science - WORKS '11. **Anais**...New York, New York, USA: ACM Press, 2011Disponível em: <http://dl.acm.org/citation.cfm?doid=2110497.2110504>. Acesso em: 2 jun. 2014

HENNEY, K. Valued Conversions. **C++ Report**, v. 12, n. 7, p. 37–40, 2000.

HOULDING, S. W. **3D Geoscience Modeling**. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994.

LIN, C. et al. A Reference Architecture for Scientific Workflow Management Systems and the VIEW SOA Solution. **IEEE Transactions on Services Computing**, v. 2, n. 1, p. 79–92, jan. 2009.

LUDÄSCHER, B. et al. Scientific workflow management and the Kepler system. **Concurrency and Computation: Practice and Experience**, v. 18, n. 10, p. 1039–1065, 25 ago. 2006.

PELLEGRINI, S.; GIACOMINI, F. **Design of a Petri Net-Based Workflow Engine**Grid and Pervasive Computing Workshops, 2008. GPC Workshops '08. The 3rd International Conference on. **Anais**...IEEE, maio 2008Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4539329>. Acesso em: 5 dez. 2013

PELLEGRINI, S.; GIACOMINI, F.; GHISELLI, A. **A Practical Approach for a Workflow Management System**Proceedings of the CoreGRID Workshop 2007. **Anais**...Dresden: 2007

PYRCZ, M. J.; DEUTSCH, C. V. **Geostatistical Reservoir Modeling**. 2. ed. [s.l.] Oxford University Press, 2014.

SIMMHAN, Y. L.; PLALE, B.; GANNON, D. A survey of data provenance in e-science. **ACM SIGMOD Record**, v. 34, n. 3, p. 31, 1 set. 2005.

TURNER, A. K. Challenges and trends for geological modelling and visualisation. **Bulletin of Engineering Geology and the Environment**, v. 65, n. 2, p. 109–127, 7 dez. 2005.

VAN DER AALST, W. M. P. The Application of Petri Nets To Workflow Management. **Journal of Circuits, Systems and Computers**, v. 08, n. 01, p. 21–66, fev. 1998.

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**, v. 41, n. Web Server issue, p. W557–61, jul. 2013.

WOOLDRIDGE, M. Agent-based software engineering. **IEE Proceedings - Software Engineering**, v. 144, n. 1, p. 26, 1997.