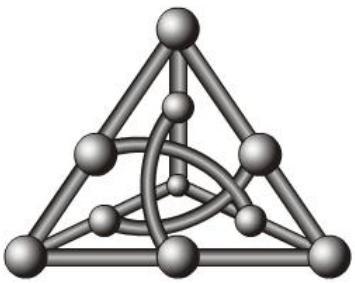
UNIVERSIDADE FEDERAL DE MATO GROSSO DO SUL
FACULDADE DE COMPUTAÇÃO

Workflows Paramétricos para Aplicações do Método dos Elementos Finitos em Ambientes Paralelos Heterogêneos

Vinícius Bueno da Silva

Dissertação apresentada à Faculdade de Computação da Universidade Federal de Mato Grosso do Sul, como parte dos requisitos para obtenção do título de Mestre em Ciência da Computação

ORIENTADOR: Dr. Paulo Aristarco Pagliosa



Campo Grande, MS
2013

Aos meus pais Teresinha e Luiz Antônio e à minha esposa Michelle.

Agradecimentos

Agradeço primeiramente a Deus por estar sempre guiando meus passos pelo melhor caminho, pela força e paciência que me deu para suportar as dificuldades que tive nesta jornada e pelas conquistas alcançadas dentro e fora deste trabalho.

Aos meus pais Teresinha e Luiz Antônio pela criação que me deram, pelo amor que sentem por mim, pelo apoio e pela confiança que sempre tiveram e por todo o esforço e toda a ajuda que me deram, estando sempre presentes mesmo com a distância física que tivemos durante minha vida acadêmica. Graças aos esforços de ambos pude chegar à conclusão de mais esta importante etapa de minha vida.

Aos meus irmão Laís e André pelo incentivo, pelo carinho e pela compreensão que tiveram, sempre torcendo por mim e estando ao meu lado nos momentos difíceis.

Agradeço à minha amiga, companheira e hoje esposa Michelle, pelo companheirismo, apoio e incentivo dados desde que nos conhecemos no primeiro ano da faculdade. Agradeço também pela confiança que teve em mim, deixando sua própria família para me acompanhar nesta nova etapa que estamos vivendo para construir nossa família.

Aos meus amigos Geraldo Landre, Leandro Magalhães e Marcel Tolentino, por participarem ativamente das conquistas realizadas durante o mestrado, participando dos grupos de estudos, sendo parte dos grupos de trabalho para que juntos pudéssemos concluir o curso.

Agradeço ao meu orientador Paulo Aristarco Pagliosa pela oportunidade, pela disposição, pelo incentivo à realização deste projeto e pela compreensão que teve em todos os momentos no decorrer deste trabalho, sendo uma referência em minha vida acadêmica e um verdadeiro amigo.

Por fim, agradeço à CAPES pela oportunidade de desenvolvimento profissional que me proporcionou através da bolsa de estudos oferecida para a realização deste trabalho.

Resumo

Silva, V.B. *Workflows Paramétricos para Aplicações do Método dos Elementos Finitos em Ambientes Paralelos Heterogêneos*. Dissertação (Mestrado em Ciência da Computação), Universidade Federal de Mato Grosso do Sul, 2013.

O objetivo geral deste trabalho é o desenvolvimento de um sistema de *workflows paramétricos* para aplicações do método dos elementos finitos (MEF) em ambientes paralelos heterogêneos. Um workflow é um processo definido por um conjunto de *atividades* que executam sequencialmente e/ou em paralelo e que podem produzir, transformar ou consumir dados. O fluxo de execução de um workflow é definido por *canais* que ligam uma *porta* de saída de uma atividade de origem a uma porta de entrada de outra atividade de destino. Por um canal podem trafegar dados ou um sinal de controle da atividade de origem à de destino, indicando que a última pode iniciar sua execução. Um workflow paramétrico é um modelo no qual uma ou mais atividades são argumentos de tipo do workflow. No sistema proposto, workflows podem ser gerados a partir de um workflow paramétrico definindo-se quais são os tipos de atividades correspondentes a cada um dos dos argumento de tipo do modelo. O sistema é escrito em C++ e constituído de três componentes principais: uma interface gráfica através da qual o usuário pode interativamente criar, modificar, armazenar e executar workflows; um motor que atua como uma máquina virtual paralela responsável pela execução de workflows; e uma biblioteca de atividades primitivas que representam os principais blocos básicos de construção de um programa, tais como sentenças de seleção, repetição, desvio e expressões. Embora possa ser destinado a outros tipos de aplicação, o projeto do sistema foi voltado para especificação de programas de análise numérica via MEF baseados em um arcabouço cujos componentes de software foram desenvolvidos pelo Grupo de Visualização, Simulação e Jogos Digitais da FACOM–UFMS. Tal arcabouço, em conjunto com o sistema de workflows paramétricos proposto neste trabalho, permitem a geração interativa e visual de aplicações de simulação via MEF e sua execução em ambientes paralelos formados por CPUs de vários núcleos e uma ou mais unidades de processamento gráfico (GPUs).

Palavras-chave: *workflows científicos*, *programação paralela*, *GPGPU*, *método dos elementos finitos*.

Abstract

Silva, V.B. *Workflows Paramétricos para Aplicações do Método dos Elementos Finitos em Ambientes Paralelos Heterogêneos*. Dissertação (Mestrado em Ciência da Computação), Universidade Federal de Mato Grosso do Sul, 2013.

The aim of this work is the development of a *parametric workflow* system for finite element method (FEM) applications in parallel heterogeneous environments. Workflow is a process defined by a set of *activities* which execute sequentially and/or in parallel and can produce, transform or consume data. The execution flow of a workflow is defined by *channels* connecting an output *port* of an source activity to an input port of a target activity. By a channel can travel data or control signals from the source acticity to the target activity, indicating that the latter can be executed. A parametric workflow is a model in which one or more activities are workflow type arguments. In the proposed system, workflows can be generated from a parametric workflow defining which activity type corresponds to each workflow type argument. The system is developed in C++ and consists of three main components: a graphical interface from which users can interactively create, modify, store and execute workflows; an engine that works as a parallel virtual machine responsible for executing workflows; and an API with a set of primitive activities representing the main language programming structures such as selection, repetition, jump, and expression statements. Although it can be used for other purposes, the system was designed for developing programs via FEM numerical analysis based on a framework whose software components were developed by the Group of Visualization, Simulation and Games at FACOM–UFMS. This framework, in conjunction with the parametric workflow system proposed in this work, allow the interactive and visual generation of simulation applications via MEF and their implementation on parallel environments composed of multi-core CPUs and one or more graphics processing units (GPUs).

Keywords: *scientific workflows*, *parallel programming*, *GPGPU*, *finite element method*.

Conteúdo

Lista de Figuras **iii**

1 Introdução **1**

1.1 Motivação e Justificativas 1

1.2 Objetivos e Contribuições 3

1.3 Organização do Texto 3

2 Workflows Científicos **5**

2.1 Introdução 5

2.2 Aspectos Gerais 6

2.3 Caracterização de Sistemas de Workflow 10

2.4 Sistemas de Workflows Científicos 12

2.4.1 Ptolemy 12

2.4.2 Pegasus 12

2.4.3 Kepler 13

2.4.4 Sistemas de Workflows para Visualização 14

2.5 Comentários Finais 17

3 Análise de Sólidos Elásticos pelo Método dos Elementos Finitos **18**

3.1 Introdução 18

3.2 Análise Elastostática Sequencial em CPU 18

3.3 Análise Elastostática em GPU 22

3.4 Análise por Decomposição de Domínio 24

3.5 Comentários Finais 26

4 Descrição do Sistema **28**

4.1 Introdução 28

4.2 Modelo de Workflows Paramétricos 28

4.3 Estrutura de Classes do Modelo 34

4.3.1 Conectores 35
4.3.2 Conexões 35
4.3.3 Componentes Conectáveis 36
4.4 Atividades Primitivas 37
4.4.1 Atividades Básicas 38
4.4.2 Estendendo a API 41
4.5 Interface Gráfica 42
4.5.1 Desenvolvimento de Atividades Paramétricas 44
4.5.2 Desenvolvimento de Workflows 47
4.5.3 Execução de Workflows 48
4.6 Comentários Finais 49

5 Exemplos **51**
5.1 Introdução 51
5.2 Gradientes Conjugados 51
5.3 Análise Estática via MEF 58
5.4 Análise por Decomposição de Domínio 63
5.5 Comentários Finais 65

6 Conclusão **67**
6.1 Discussão dos Resultados Obtidos 67
6.2 Trabalhos Futuros 69

Referências **74**

Lista de Figuras

2.1 Compatibilidade de tipos para a realização de conexões. 8
2.2 Abordagens para envio de resultados. 9
2.3 Abordagens para análise de execução de workflows. 10
2.4 Produto cartesiano entre entradas de dados. 11
2.5 Interface do sistema Ptolemy. 13
2.6 Workflow desenvolvido no sistema Pegasus. 14
2.7 Interface do sistema Kepler. 15
2.8 Interface do sistema Vistrails. 16
2.9 Interface do VTK Designer. 16

3.1 Uma malha triangular 2D com elementos coloridos. 24

4.1 Representação de atividade básica. 28
4.2 Fluxo de controle entre duas atividades. 29
4.3 Atividade `MatrixReader`. 29
4.4 Atividade `MatrixWriter`. 30
4.5 Atividade `MulMatrix`. 30
4.6 Conexões entre portas de tipos de dados compatíveis. 30
4.7 Paralelismo entre atividades. 31
4.8 Sincronismo em fluxos de controle. 31
4.9 Sincronismos em fluxos de dados. 32
4.10 Proxies de porta de um bloco. 32
4.11 Atividade paramétrica `StaticAnalysis`. 33
4.12 Tipo de atividade `SimpleDofNumberer`. 33
4.13 Instanciação de atividade paramétrica. 34
4.14 Proxy de argumento de tipo. 34
4.15 Diagrama de classes do modelo proposto. 37
4.16 Composição de um bloco. 39

4.17 Representação da atividade `StaticAnalysis`. 42
4.18 Interface Gráfica. 43
4.19 Diagrama de classes dos itens gráficos. 44
4.20 Criação de atividade paramétrica. 44
4.21 Caracterização de atividades paramétricas. 45
4.22 Criação de portas. 46
4.23 Criação de argumentos de tipos. 46
4.24 Criação da atividade `Block`. 47
4.25 Criação de fluxo de execução através de canais. 47
4.26 Instanciação de atividade paramétrica. 48
4.27 Diagrama de classes do motor de execução. 48
4.28 Execução do MEF. 49

5.1 Workflow `CGSolver`. 54
5.2 Atividade `For 1..maxIter`. 55
5.3 Bloco da atividade `statement`. 56
5.4 Instanciação de `CGSolver` para CPU. 57
5.5 Instanciação de `CGSolver` para GPU. 57
5.6 Atividade `FEMMain`. 58
5.7 Atividade do tipo `FemStaticAnalysis`. 60
5.8 Atividade `FemLinearAlgorithm`. 61
5.9 Bloco de formação do sistema na atividade `FemLinearAlgorithm`. 62
5.10 `FEMMain` para CPU. 62
5.11 `FEMMain` para GPU. 63
5.12 Aplicação do MEF para subdomínios. 64
5.13 Atividade `FemSubdomainAnalysis`. 65
5.14 Bloco de `FemSubdomainAnalysis` que computa as contribuições do subdomínio. 66
5.15 Análise com decomposição de domínio em GPU. 66

CAPÍTULO 1

Introdução

1.1 Motivação e Justificativas

Um problema importante em aplicações de simulação em ciências e engenharia, e que motiva o desenvolvimento deste trabalho, é a determinação dos deslocamentos de todos os pontos de um sólido deformável provocados por um conjunto de forças aplicadas em seu domínio e/ou contorno. Uma vez que o problema é computacionalmente intensivo, pretende-se solucioná-lo através da utilização de ambientes paralelos heterogêneos, que são ambientes formados por CPUs de vários núcleos e uma ou mais unidades de processamento gráfico (GPUs).

Matematicamente, o problema é descrito por um modelo definido por um conjunto de equações diferenciais parciais, as quais relacionam os deslocamentos e forças em termos da geometria e propriedades materiais do sólido, e um conjunto de condições de contorno essenciais à unicidade da solução. Em geral, o modelo matemático admite somente soluções aproximadas, as quais podem ser obtidas com o uso de métodos numéricos tais como o método dos elementos finitos (MEF).

Utilizamos neste trabalho uma API (interface de programação de aplicação) desenvolvida em [34] para criação de programas de análise numérica baseados no MEF em CPU e/ou GPU. Tal API disponibiliza classes implementadas em C++ que representam objetos tais como domínios, elementos finitos e sistemas lineares, entre outros componentes que modelam dados e/ou processos relacionados aos passos do pipeline de análise numérica via MEF.

A codificação manual de uma aplicação de análise numérica com elementos finitos envolve a instanciação, dentre os componentes disponíveis na API, daqueles responsáveis pelo processamento de cada passo do pipeline. Feito isso, deve-se estabelecer o fluxo de controle e/ou de dados entre os mesmos, associando entradas e saídas de componentes distintos de acordo com a compatibilidade entre ambos. Visto que mais de um tipo de componente pode ser empregado para a mesma etapa do pipeline de análise como, por exemplo, a utilização de diferentes tipos de montadores e solucionadores de sistemas lineares, o uso de componentes diversos exige alterações no código que, além do trabalho de repetição e recompilação por parte do programador, podem introduzir erros, até mesmo em programas simples de análise elastostática com um número reduzido de componentes. Tais desvantagens podem ser percebidas de forma mais clara em programas de tipos mais complexos de análise como, por exemplo, análise dinâmica, elastoplástica, de fraturas, ou combinações dessas, entre outras. Essas análises envolvem um número maior de componentes e um conjunto de relacionamentos que definem trechos do fluxo capazes de serem executados sequencialmente ou em paralelo em sistemas heterogêneos.

O desenvolvimento de aplicações com execução em paralelo torna-se mais complexa, pois a programação representa os códigos de forma linear, não sendo capaz de representar o paralelismo de forma intuitiva. A utilização de sistemas de gerenciamento de workflows tem se mostrado uma alternativa para a solução do problema, sendo empregada no desenvolvimento de experimentos científicos em diversas áreas como, por exemplo, bioinformática, meteorologia e engenharias. Esses sistemas representam o desenvolvimento de experimentos através de workflows que, no contexto apresentado, são representações gráficas de processos compostos por um conjunto de componentes relacionáveis.

Os sistemas de gerenciamento de workflows científicos disponíveis atualmente têm por objetivo automatizar experimentos científicos que possam ser executados para manipular grandes volumes de dados de maneira automatizada. Essa característica dificulta a utilização desses sistemas para o desenvolvimento de componentes baseados em GPU, pois os recursos de memória para GPU são limitados se compararmos com o volume de dados a serem manipulados nesse tipo de experimento. A utilização de GPUs para o desenvolvimento de aplicações baseadas no MEF é desejável, pois é possível desenvolver componentes para execução específica em GPU, aproveitando sua capacidade de processamento.

Outra característica procurada nos sistemas disponíveis é a facilidade de alteração do workflow através da substituição de componentes, permitindo que o usuário possa comparar diferentes instâncias do mesmo processo. A substituição de um componente A por um componente B é realizada nos sistemas pesquisados através da exclusão do primeiro componente, desfazendo todas as associações de A com outros componentes. Em seguida o componente B é posicionado no lugar do componente original, sendo necessário refazer todos os relacionamentos presentes anteriormente. Esse procedimento pode demandar muito tempo se considerarmos workflows contendo um grande número de componentes relacionando-se com várias instâncias do componente A, além de induzir o usuário à introdução de erros.

Desenvolvemos neste trabalho um sistema de gerenciamento de workflows que permite a criação e edição de workflows através da utilização de uma abordagem distinta do conceito de workflows paramétricos. Diferentemente dos sistemas disponíveis, os workflows paramétricos deste projeto são fluxos de execução definidos a partir da utilização de argumentos de tipo. Cada argumento representa um molde de atividade, que pode ser instanciado por um tipo de atividade compatível para a criação de workflows executáveis. Essa abordagem oferece maior facilidade na edição de workflows, pois a estrutura do fluxo de execução pode ser definida em função dos argumentos de tipo, permanecendo inalterada enquanto o usuário substitui os componentes através de diferentes instanciações do mesmo workflow. Ao contrário de outros sistemas, que realizam a substituição de componentes através da quebra e reconstrução de conexões, o sistema proposto substitui uma única conexão entre o argumento de tipo do workflow e o tipo de atividade a ser utilizada, permitindo que o usuário compare diferentes componentes e escolha os mais adequados.

O sistema proposto apresenta um conjunto de componentes básicos que permitem ao usuário o desenvolvimento de workflows e, a partir da biblioteca disponível, estender a API através da criação de novas classes de componentes ou através da definição de workflows paramétricos, que podem ser utilizados no desenvolvimento de aplicações maiores. A API desenvolvida dispõe ainda de componentes desenvolvidos para execução em GPU, facilitando a especificação e execução de aplicações baseadas no MEF.

1.2 Objetivos e Contribuições

O objetivo geral deste projeto é o desenvolvimento de um sistema de workflows paramétricos para especificação e execução de aplicações baseadas no MEF em ambientes paralelos heterogêneos. O sistema foi codificado em C++ e é constituído de três componentes principais:

C1 Uma interface gráfica através da qual o usuário pode interativamente criar, modificar, armazenar e executar workflows.

C2 Um motor que atua como uma máquina virtual paralela responsável pela execução de workflows.

C3 Uma biblioteca de atividades primitivas que representam os principais blocos básicos de construção de um programa, tais como sentenças de seleção, repetição, desvio e expressões.

Os objetivos específicos do projeto são:

O1 Implementar todos os componentes do sistema proposto.

O2 Criar uma biblioteca de atividades correspondentes aos componentes C++ da API do MEF e, com isto, workflows para análise elastostática de sólidos em CPU e/ou GPU.

As contribuições pretendidas são:

- A partir do conceito de workflow paramétrico, oferecer ao usuário novos recursos que auxiliam o processo de desenvolvimento e reuso de workflows. Este conceito, somado às informações que serão geradas pelo motor durante a execução, constituirá uma ferramenta capaz de avaliar a eficiência de diferentes instâncias de workflows e assim comparar o desempenho dos componentes utilizados.
- Proporcionar maior facilidade na especificação de soluções em paralelo para elementos finitos em ambientes heterogêneos. Espera-se que, através da extensão da biblioteca de atividades primitivas, o usuário poderá desenvolver atividades específicas e utilizar a interface para construir de forma mais intuitiva processos que utilizem tais atividades.

1.3 Organização do Texto

O restante do texto é organizado em cinco capítulos conforme descrito a seguir.

No Capítulo 2 apresentamos os conceitos básicos para a caracterização de workflows e sua utilidade no desenvolvimento de pesquisas. Apresentamos ainda as principais características presentes nos sistemas de desenvolvimento de workflows científicos e comparamos o sistema proposto às principais ferramentas disponíveis atualmente para a comunidade científica.

O Capítulo 3 descreve os conceitos envolvidos na análise de sólidos elásticos e o processo de solução do problema através da utilização do método dos elementos finitos. Descrevemos também a metodologia utilizada para tratar o problema, apresentando as diferenças entre as soluções em CPU e GPU.

No Capítulo 4 apresentamos a visão geral do sistema, descrevemos de forma detalhada o modelo proposto para a representação de workflows paramétricos. Em seguida apresentamos o conjunto de classes utilizadas para a representação do modelo proposto. Apresentamos também a interface do sistema desenvolvido para a representação de workflows e descrevemos a utilização das funcionalidades disponíveis.

O Capítulo 5 apresenta exemplos de workflows desenvolvidos através do sistema proposto. Os workflows desenvolvidos definem a estrutura de execução de uma aplicação do MEF composta por atividades presentes na API do sistema proposto. Os exemplos demonstram como é possível instanciar a aplicação para que seja executável em CPU ou GPU.

O Capítulo 6 apresenta as considerações e conclusões obtidas com o trabalho desenvolvido e algumas possibilidades de trabalhos futuros.

CAPÍTULO 2

Workflows Científicos

2.1 Introdução

O desenvolvimento da tecnologia computacional viabilizou a realização de experimentos científicos, onde o gerenciamento de dados e o relacionamento entre os componentes eram muito difíceis [15]. Os experimentos eram realizados através da utilização de diferentes aplicações, cujo monitoramento e comunicação dependiam da intervenção humana. A automação do experimento passou a ser realizada através da utilização de scripts [8, 30], onde um roteiro de execução coordena a inicialização das aplicações e auxilia a comunicação entre as aplicações a partir de leituras e escritas em arquivos.

A automação dos experimentos tornou possível obter os resultados das pesquisas em menos tempo, pois era possível desenvolver aplicações específicas para a realização de cada etapa do processo e a execução em paralelo de etapas independentes proporcionou melhor aproveitamento dos recursos computacionais disponíveis [28]. Entretanto, o aumento da complexidade dos processos e a execução em paralelo necessitava de uma nova abordagem para a definição dos fluxos de trabalho. Notou-se que a representação dos fluxos de execução dos experimentos assemelhava-se à representações de processos de negócio [20, 42] e pesquisas científicas passaram a adotar abordagem semelhante, representando graficamente os fluxos de trabalho e facilitando o gerenciamento de experimentos com grande complexidade [10] ou volume de dados [4, 18]. Desde então muitas discussões foram feitas a respeito da definição e importância de workflows científicos [20, 30].

Um workflow é a representação de um processo, que pode ser estruturado computacionalmente como um grafo direcionado acíclico [16], cujos vértices representam atividades a serem executadas e as arestas representam relações de dependência entre as atividades. Cada atividade representa um conjunto de uma ou mais instruções utilizadas para a manipulação de dados, resultando na delimitação de uma etapa do processo. O relacionamento entre duas atividades A e B pode ser definido através de fluxos de controle, onde a execução de B depende da conclusão da execução de A, ou fluxos de dados, onde A precisa transferir os dados resultantes de sua execução para B, permitindo que B utilize esses dados em sua execução [19, 40]. As atividades podem ser classificadas como produtoras, quando fornecem dados manipulados em sua execução, consumidoras, quando necessitam receber dados a serem manipulados, ou de transição, quando ocupam etapas intermediárias do processo e transformam os dados de entrada em dados de saída.

O desenvolvimento de workflows apresenta conjuntos de padrões divididos entre gerenciamento de fluxo, dados, recursos e tratamentos de exceção [25]. Padrões para gerenciamento de

controle [39] definem modelos para representação de dependência ou prioridade de execução entre as atividades como, por exemplo, o paralelismo, o sincronismo e a execução sequencial. Padrões para gerenciamento de dados [38] definem modelos de representação e acesso a dados e formas de relacionamento entre atividades produtoras e/ou consumidoras de dados, permitindo o tráfego das informações no decorrer do processo. Padrões para o gerenciamento de recursos [36] definem formas de representar e utilizar os recursos computacionais em workflows. Padrões de tratamento de exceção [37] definem formas de identificar exceções na execução de workflows e realizar seu tratamento. Baseamo-nos nos padrões de fluxo de controle e de dados para o desenvolvimento de workflows híbridos, permitindo que o tráfego de dados seja conduzido a partir de decisões tomadas em função dos resultados de cada etapa do processo.

Um workflow científico é a representação de processos de pesquisa através de workflows [21], cujo objetivo é gerenciar computacionalmente experimentos envolvendo processamento de grandes volumes de dados. Workflows científicos podem ser empregados em diversos campos de ciências e engenharia, tais como bioinformática [24], meteorologia [41], entre outros. A utilização de workflows permite o monitoramento da origem e trajetória das informações até a obtenção dos resultados desejados, desenvolvimento de pesquisas em conjunto com outras equipes e otimização dos recursos disponíveis.

Neste capítulo apresentamos um resumo dos principais sistemas de workflows científicos relacionados ao proposto neste trabalho. Na Seção 2.3, apresentamos as principais características presentes nos sistemas de gerenciamento de workflows científicos. Na Seção 2.4, descrevemos características distintas das principais ferramentas de gerenciamento de workflows.

2.2 Aspectos Gerais

A difusão dos workflows científicos resultou na criação de aplicações voltadas para o desenvolvimento e gerenciamento de workflows. As aplicações mais completas como, por exemplo, Ptolemy [17], Kepler [13, 29], Pegasus [26], Vistrails [33] e Taverna [31] apresentam a característica de desenvolvimento de workflows baseados em aplicação [27]. Nessa abordagem de desenvolvimento, os workflows são desenvolvidos a partir de componentes, que representam aplicações a serem utilizadas na transformação dos dados de entrada em um determinado resultado.

Os sistemas de workflows (SWFs) são ambientes de desenvolvimento de workflows baseados nos modelos de fluxo de dados, onde componentes independentes consomem e/ou produzem dados e se relacionam, definindo um fluxo e uma relação de dependência de dados.

De acordo com [20], os sistemas de workflows científicos tem por objetivo representar graficamente processos de transformações de dados cujos resultados podem ou não ser representados através de imagens. Esses sistemas são desenvolvidos a partir de um modelo composto por três partes: funcional, visualização e objetos.

A parte funcional é a representação da estrutura do workflow utilizado para transformar dados. Essa estrutura é representada por um grafo orientado. Cada nó do workflow representa uma etapa de transformação responsável por receber determinados tipos de dados, realizar operações sobre eles e retornar um resultado. Cada aresta do workflow é uma seta que representa a ligação entre dois nós e a direção em que os dados serão transferidos.

A visualização é responsável por transformar os dados resultantes do processamento de um workflow em elementos gráficos primitivos. Esses elementos envolvem as estruturas de dados utilizadas para representar computacionalmente os itens gráficos e as diferentes formas de represen-

tação gráfica que facilitem o entendimento da informação. Esse modelo é composto por elementos capazes de transformar dados em gráficos, tabelas, diagramas, arquivos de saída de diferentes formatos, imagens bidimensionais ou tridimensionais, entre outros.

Composição de Objetos

Os objetos são responsáveis por descrever os elementos que compõem o desenvolvimento de workflows. Os objetos podem ser desenvolvidos sob duas abordagens diferentes:

- Combinação entre o armazenamento de dados e o processamento;
- Divisão entre objetos de armazenamento de dados e objetos de processamento.

A primeira abordagem define componentes que contém estruturas de armazenamento de dados e métodos de processamento que agem sobre os dados armazenados. Essa abordagem permite que os processos tenham total acesso aos dados, porém dificulta do entendimento por parte do usuário, pois as etapas do processo podem ser encaradas como objetos. Outra desvantagem é que para cada estrutura de dados que precise ser manipulada deve haver um componente capaz de armazená-la, resultando em replicações do mesmo método.

A segunda abordagem define objetos responsáveis pela representação de dados e objetos responsáveis por representar etapas do processo. Essa abordagem define uma representação mais intuitiva para os usuários, pois deixa clara a diferença entre as etapas do processo e os dados que são manipulados, além de permitir uma fácil extensão dos componentes.

Os objetos de dados são responsáveis pela representação, armazenamento e manipulação direta dos dados, definindo quais operações podem ser aplicadas e como outros objetos podem obter os dados armazenados dentro de si.

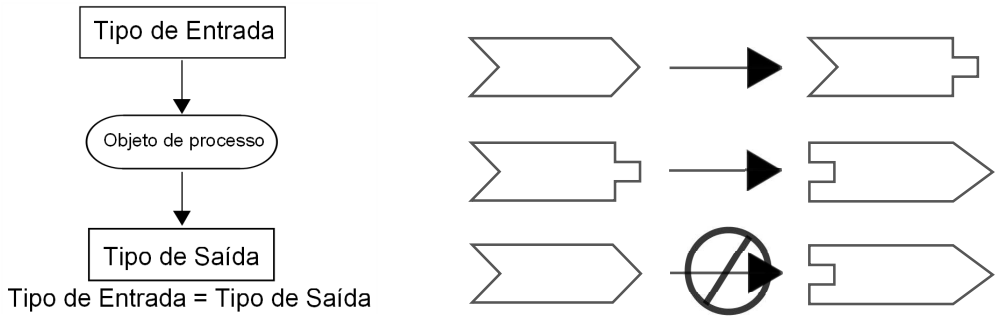
Os objetos de processo são responsáveis por definir um conjunto de operações a serem realizadas em objetos de dados recebidos como entrada, transformando-os em dados a serem disponibilizados como saída. Esses objetos são ligados uns aos outros de acordo com o tipo de informação que necessitam para serem executados e pelo tipo de informação que produzem e são capazes de encaminhar para outros objetos. Os objetos de dados são classificados em três categorias:

- Produtores: objetos capazes de enviar objetos de dados a partir de seu processamento próprio;
- Consumidores: objetos que recebem objetos de dados e os transformam em dados de saída do processo, podendo ou não apresentá-los de forma visual;
- Filtros: objetos que recebem objetos de dados e os manipula para disponibilizar um novo resultado.

Conectividade entre Objetos

Os objetos de processo são conectados para permitir o tráfego de objetos de dados. Essas conexões podem ser realizadas baseadas em um único tipo ou baseadas em múltiplos tipos. Os modelos de tipo único permitem a conexão entre quaisquer componentes de processo, pois os dados produzidos e consumidos são representados sem tipo ou por um único tipo, conforme representado pela Figura 2.1(a). Essa abordagem limita o desenvolvimento do workflow, pois não permite que objetos de dados tenham métodos próprios de manipulação das informações. As conexões de múltiplos

tipo são aquelas em que há uma hierarquia entre as classes de tipos de dados e uma entrada aceita qualquer objeto de dados cuja classe seja igual ou derivada do tipo do destino, conforme representado pela Figura 2.1(b). Essa abordagem permite maior flexibilidade, pois o mesmo objeto de processo pode manipular vários tipos de dados compatíveis, porém a criação indiscriminada de tipos de objetos de dados diferentes pode dificultar a compreensão do sistema.



(a) Conexão de tipo único. (b) Conexão de tipo múltiplo.

Figura 2.1: Compatibilidade de tipos para a realização de conexões.

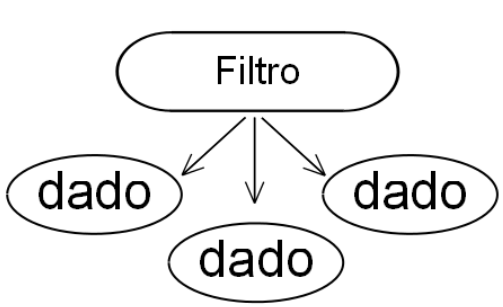
As conexões entre objetos de processo também são determinadas de acordo com a capacidade de recebimento de dados das entradas e o método de proliferação dos dados produzidos e disponibilizados nas saídas. As vias de entrada presentes em objetos consumidores ou filtros podem admitir apenas uma ou várias entradas, operando sequencialmente sobre elas. As vias de saída presentes em objetos produtores ou filtros são capazes de prover um resultado para uma ou mais vias de entrada de objetos de processo dependentes. O resultado pode ser um conjunto contendo um ou mais objetos de dados, que podem ser disponibilizados de forma repetida ou compartilhada.

O resultado é disponível de forma repetida quando cada via de entrada que depende desse resultado o recebe em uma cópia distinta, conforme representado pela Figura 2.2(a). Essa abordagem permite maior independência entre os objetos de processo, pois cada cópia pode ser manipulada livremente.

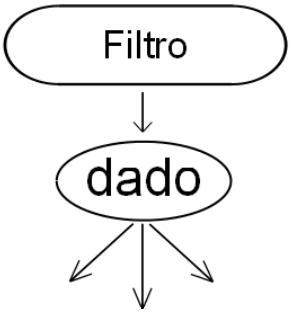
O resultado é disponibilizado de forma compartilhada quando cada via de entrada dependente do resultado recebe apenas uma referência para o objeto de dados produzido, conforme representado pela Figura 2.2(b). Essa abordagem permite menor consumo de memória e atuação de diferentes objetos de processo sobre um mesmo objeto de dados, porém resulta em condição de concorrência de acesso aos dados para que a consistência seja mantida.

Método de Execução

A execução de workflows tem por objetivo verificar a corretude de sua estrutura de execução, avaliar seu desempenho e obter um resultado baseado no processamento dos dados de entrada. A execução de workflows contendo grande número de componentes e relacionamentos pode consumir grande quantidade de recursos computacionais, principalmente quando o objetivo é o processamento de grandes volumes de dados. Para contornar esse problema, os sistemas de gerenciamento de workflows científicos utilizam políticas de execução de workflows baseadas em demanda ou evento. A política baseada em demanda executa apenas os objetos de processo cuja saída seja necessária para a execução de outros objetos e apenas as porções do workflow que influenciem no resultado final. Essa política possui a vantagem de não executar componentes desnecessários,



(a) Envio de resultado em objetos distintos.



(b) Envio de resultado por objetos compartilhados.

Figura 2.2: Abordagens para envio de resultados.

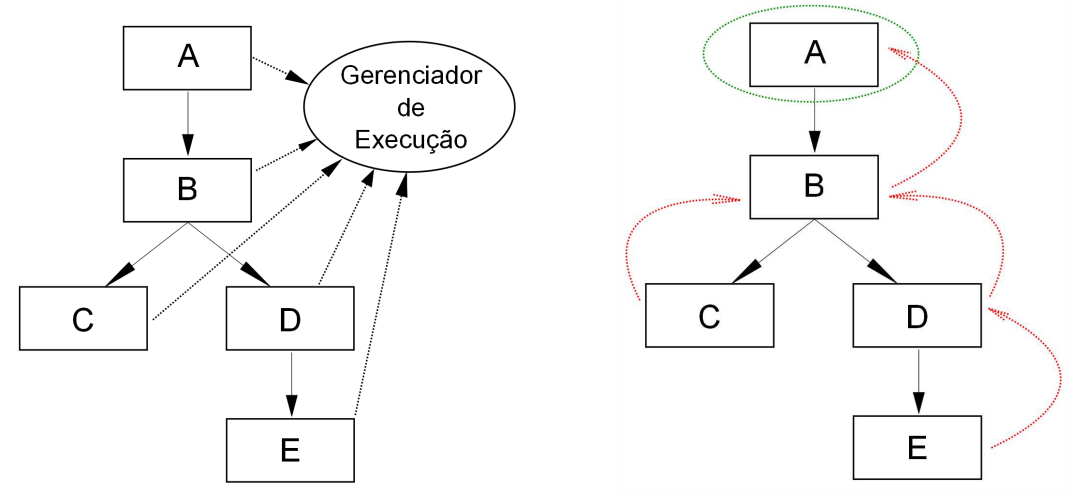
minimizando a utilização dos recursos. A política baseada em eventos executa o workflow a cada modificação de sua estrutura ou dos dados de entrada, apresentando como vantagem o fato de manter os objetos de dados de saída sempre atualizados

A execução de workflows deve ser gerenciada para garantir que os relacionamentos entre componentes sejam respeitados e o resultado seja produzido corretamente. Os objetos de processo são executados quando não possuem dependências com os objetos a ela relacionados e essa análise pode ser feita de maneira explícita ou implícita.

A análise explícita prevê a verificação constante das condições de dependência entre os componentes do workflow a partir de um gerenciador externo à estrutura de execução, conforme demonstrado pela Figura 2.3(a). Essa abordagem possui a vantagem de permitir maior controle sobre a sincronia entre os elementos e permite que subgrafos sejam definidos para que sejam distribuídos entre os recursos computacionais disponíveis. A desvantagem é a necessidade de os objetos de processo se reportarem ao gerenciador de execução para que o fluxo seja controlado.

A análise implícita é realizada pelos próprios objetos de processo pertencentes ao workflow. A saída do workflow solicita os objetos de dados para os componentes responsáveis por enviar seus resultados, conforme demonstrado pela Figura 2.3(b). Cada componente verifica se é possível enviar a informação e, caso não seja, notifica os componentes dos quais depende para que seus resultados sejam enviados. O processo se repete até que os componentes produtores sejam notificados. Os produtores passam a ser executados e seus resultados enviados aos componentes filtros que solicitaram os dados. O caminho inverso se repete até que o resultado do workflow seja conhecido. Apesar de facilmente implementada, essa opção dificulta a distribuição dos componentes em um ambiente distribuído, constituído de diferentes recursos computacionais conectados através de uma rede.

O sistema proposto, conforme apresentaremos no Capítulo 4, apresenta um modelo de representação no qual descrevemos objetos de processo como *atividades* [11], que se conectam por suas vias de acesso chamadas *portas*. As portas são ligadas através de canais para transmitir objetos de dados no decorrer da execução do workflow. As portas possuem a capacidade de armazenar um conjunto de objetos de dado, que podem ser acessados pelo fluxo de execução da atividade de acordo com a necessidade do usuário. As portas possuem um tipo de dados associado, que representa o tipo de dados base com a qual é compatível, permitindo que qualquer objeto de dados com tipo derivado seja atribuído a elas. As portas de saída disponibilizam seus dados de forma compartilhada, permitindo melhor aproveitamento de memória através do controle de referências, porém necessita realizar maior controle sobre a concorrência ao acesso dos dados. A execução



(a) Análise explícita de execução. (b) Análise implícita de execução.

Figura 2.3: Abordagens para análise de execução de workflows.

dos workflows é realizada através de análise explícita, onde gerenciadores de execução de blocos chamados *motores de execução* são responsáveis por coordenar a execução de atividades contidas no bloco, sendo o workflow um bloco global. Essa abordagem permite maior controle sobre a execução do workflow e, por distribuir o gerenciamento entre vários motores, permite gerenciar a execução de workflows em ambiente distribuído com maior facilidade.

2.3 Caracterização de Sistemas de Workflow

Para auxiliar o usuário no desenvolvimento dos workflows, é desejável que os SWFs - Sistemas de Gerenciamento de Workflows apresentem as seguintes características [30]:

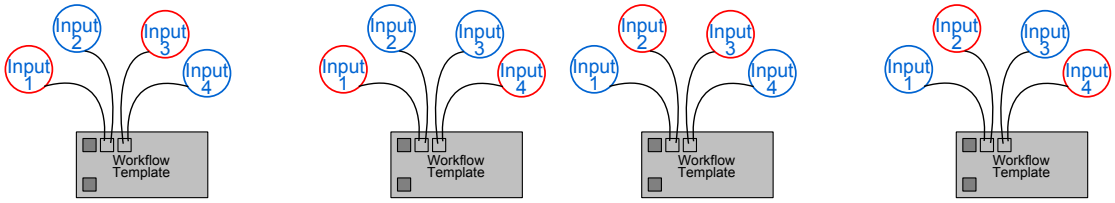
- Sejam capazes de validar a construção e o tráfego de dados definido;
- Apresentem os componentes e workflows de forma clara, permitindo que o usuário possa compreender o processo sem a necessidade de executar o workflow;
- Permitam que o usuário obtenha os dados produzidos para verificar a corretude do processo;
- Forneçam um conjunto de componentes básicos e a possibilidade de desenvolvimento de novos componentes;
- Permitam o reuso de workflows desenvolvidos como componentes na construção de novos workflows;
- Permitam a modelagem de dados de acordo com a necessidade do usuário.

A validação dos workflows é feita a partir da utilização de linguagens de representação de workflows próprias, que impedem, por exemplo, a criação ciclos nos fluxos de dados ou o tráfego de dados entre componentes incompatíveis entre si. Essas linguagens, portanto, definem as regras a serem seguidas para a boa formação de um workflow.

O desenvolvimento de workflows é realizado a partir da utilização de bibliotecas básicas de componentes, que determinam estruturas e comportamentos essenciais para qualquer componente e possibilitam ao usuário a extensão da biblioteca de acordo com o experimento a ser realizado. As principais características determinadas pelas bibliotecas são a presença de vias de entrada e saída de dados, formas de relacionamento dos componentes e métodos abstratos para determinar, por exemplo, o procedimento de manipulação dos dados que trafegam em cada etapa do fluxo a ser definido.

Os SWFs citados permitem a execução de workflows em sistemas distribuídos, com o objetivo de permitir a integração entre equipes geograficamente distantes, possibilitar a utilização de diferentes bases de dados e compartilhar conhecimentos e recursos de centros de processamento de dados. Para garantir a utilização das diferentes bases de dados, as bibliotecas padrão também disponibilizam componentes para leitura e escrita para tipos básicos de dados e sua conversão em metadados que possam ser manipulados corretamente pelos componentes. Essa funcionalidade permite que recursos variados sejam aproveitados, desde computadores pessoais a clusters e supercomputadores.

Outra característica importante é a parametrização de workflows, onde elementos utilizados como molde podem ser instanciados a fim de definir diferentes características do comportamento do componente. Projetos desenvolvidos nos SWFs citados, utilizam a parametrização de workflows em duas abordagens distintas. A primeira abordagem envolve a instanciação de dados de entrada, onde os componentes produtores podem apresentar número variável de dados de entrada, sem necessariamente conhecer as fontes de dados. Esse tipo de abordagem realiza um produto cartesiano entre os dados de entrada, combinando-os para gerar instâncias executáveis com saídas distintas para cada combinação, conforme representado pela Figura 2.4.



(a) Execução com inputs 1 e 3. (b) Execução com inputs 1 e 4. (c) Execução com inputs 2 e 3. (d) Execução com inputs 2 e 4.

Figura 2.4: Produto cartesiano entre entradas de dados.

A segunda abordagem é a utilização de uma família ou conjunto de componentes semelhantes, sendo usados como alternativas de instanciação de um componente molde. Essa abordagem se faz necessária, por exemplo, em componentes de leitura de dados, permitindo que parâmetros de dados em diferentes formatos sejam aceitos pelo produtor. A manipulação de grandes volumes de dados, provenientes de bases remotas heterogêneas, exige que os workflows desenvolvidos sejam capazes de ler diferentes formatações dos dados de entrada. Utilizando a extensão da biblioteca disponível é possível desenvolver componentes de leitura para padronização dos dados de entrada. Esses componentes, de acordo com a formatação dos dados obtidos, transformam os dados em metadados padronizados que possam ser reconhecidos pelos demais componentes do workflow [21, 26, 16]. Durante a criação do workflow, o usuário insere ao molde todos os componentes da família e, em tempo de execução, o componente apropriado é escolhido de acordo com os tipos de dados de entrada disponíveis.

A transferência de dados entre os componentes é feita através de leituras e escritas em arquivos, pois o volume de dados produzidos dificulta sua transferência direta entre os componentes. Ao concluir sua execução, o componente escreve seus dados em arquivos para que os componentes consumidores possam acessá-los. Os sistemas apresentados utilizam as extensões de arquivos como tipos de dados a serem aceitos por cada componente, permitindo apenas o relacionamento entre componentes capazes de manipular as mesmas extensões de arquivos. Além de possibilitar a comunicação entre os componentes, a persistência dos dados em arquivos possibilita ao pesquisador o monitoramento do processo de obtenção dos resultados.

2.4 Sistemas de Workflows Científicos

Apresentaremos os principais sistemas de gerenciamento de workflows científicos, descrevendo as funcionalidades que os distingue das demais aplicações.

2.4.1 Ptolemy

O projeto Ptolemy [17, 7], Figura 2.5, foi desenvolvido em 1991 com o propósito de unir características dos sistemas Blosim e Gabriel, destinados à simulação de processamento de sinais e prototipação de ambientes para processamento de sinais, respectivamente. Seu propósito inicial foi facilitar a concepção de sistemas heterogêneos de tempo real, que são compostos pela união de subsistemas desenvolvidos para tratar um problema específico [1]. O projeto apresenta uma série de componentes, chamados atores, disponíveis em uma biblioteca desenvolvida em Java e permite a representação de workflows sob diferentes modelagens como, por exemplo, fluxos de dados, visualização 3D, modelos de tempo contínuo, modelos baseados em eventos, entre outros. Os atores são conectados a partir de suas portas e a execução do processo definido é realizada a partir de motores de execução, chamados diretores. Cada diretor é responsável por controlar um modelo específico e podem ser compostos hierarquicamente para permitir sua integração. Esse sistema serve de base para a criação de sistemas especializados como, por exemplo, o sistema Kepler.

2.4.2 Pegasus

O projeto Pegasus [23, 29, 16, 14], Figura 2.6, foi desenvolvido em 2002 com o propósito de desenvolver e gerenciar a execução de workflows em grids, analisando os recursos disponíveis e utilizando aplicações e serviços como componentes para a manipulação de dados. O sistema permite a utilização de vários ambientes de execução diferentes, variando desde simples computadores a clusters e sistemas distribuídos. O sistema é capaz de mapear os fluxos de trabalho e identificar os recursos e as fontes de dados disponíveis. Através do mapeamento, o sistema é capaz de gerenciar os recursos e reorganizar o fluxo do processo para garantir execuções mais eficientes.

Para auxiliar o desenvolvimento de workflows com níveis de abstração maiores, o usuário dispõe do projeto Wings [1, 22], criado em 2006 com o objetivo de desenvolver *workflows moldes*, onde a instanciação é realizada através da escolha de aplicações e fontes de dados a serem utilizadas, enquanto o Pegasus fica responsável pela distribuição e controle da execução dos componentes de acordo com os recursos computacionais disponíveis.

[1] Sistemas embarcados são exemplos desse tipo de composição, onde hardwares com características diferentes são unidos para compor a solução de um problema maior.

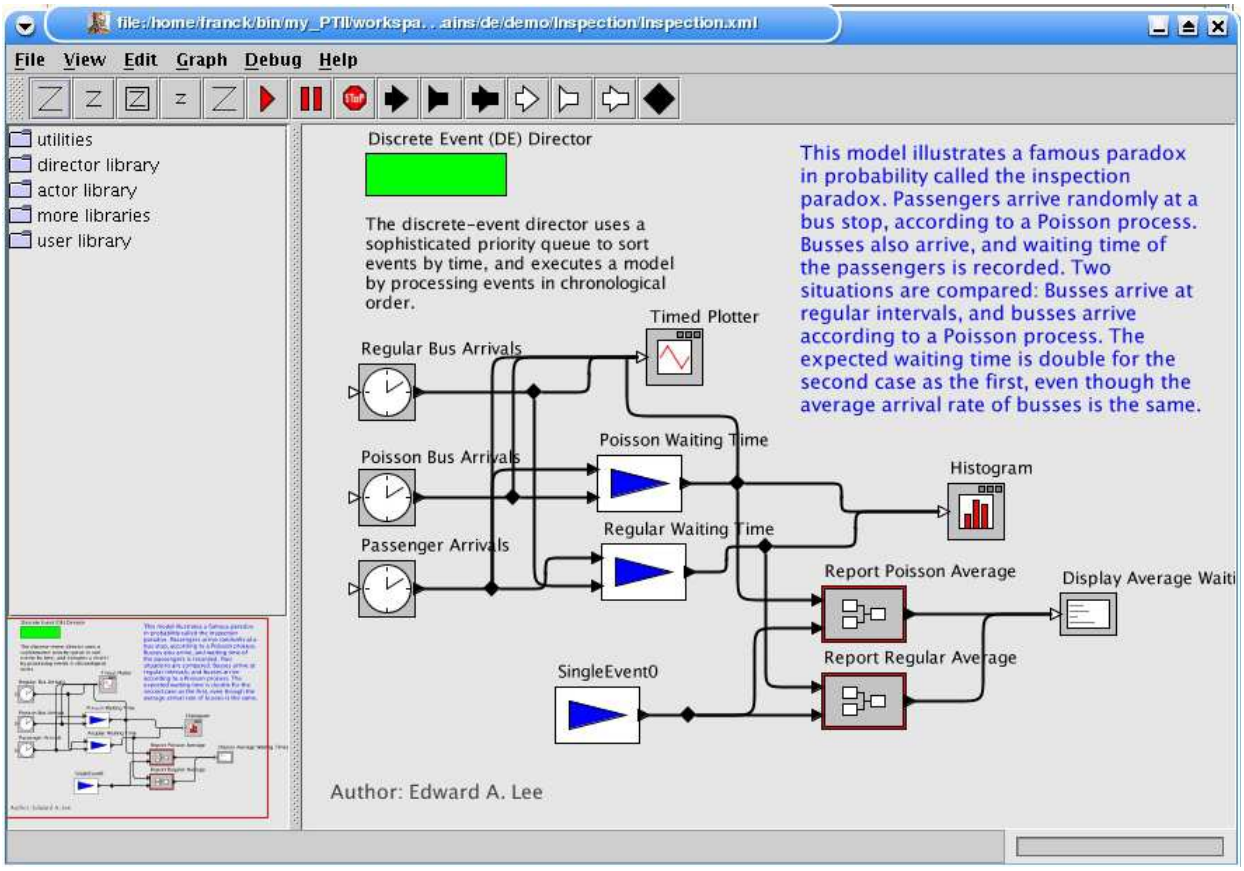


Figura 2.5: Interface do sistema Ptolemy (ptolemy.eecs.berkeley.edu/ptolemyII).

Outra característica importante é a política de tratamento de exceções adotada pelo sistema. Ao detectar uma falha de execução, o sistema procura isolar a tarefa onde a falha ocorreu e tenta repetir sua execução, utilizando o mesmo conjunto de dados. Caso a falha persista, um novo conjunto de dados é escolhido e uma nova tentativa de execução é realizada. O sucesso dessa nova tentativa indica, por exemplo, que o motivo da falha pode ser um conjunto de dados de entrada irregular. A ocorrência de nova falha pode indicar, por exemplo, que não havia memória disponível no recurso utilizado, sendo tratado pelo sistema através da liberação de memória para novas tentativas de execução. Ao persistir o problema, o sistema finalmente interrompe as tentativas e alerta o usuário sobre o problema no componente.

2.4.3 Kepler

O projeto Kepler [29], Figura 2.7, foi criado em 2004 a partir do sistema Ptolemy e é aplicado ao desenvolvimento de workflows com fluxo de dados. O sistema se destaca pela capacidade de gerenciar a criação e execução de workflows em sistemas distribuídos heterogêneos, onde diferentes equipes participando de uma mesma pesquisa podem contribuir com o desenvolvimento de componentes. Essa característica permite também a utilização de diferentes bases de dados, que podem estar distantes geograficamente e que utilizadas em conjunto fornecem maior cobertura de informações. O desenvolvimento de experimentos entre diferentes equipes facilita a padronização de informações presentes nas diferentes bases de dados, pois cada equipe participante pode gerar componentes próprios para a conversão dos dados em metadados utilizáveis no processo.

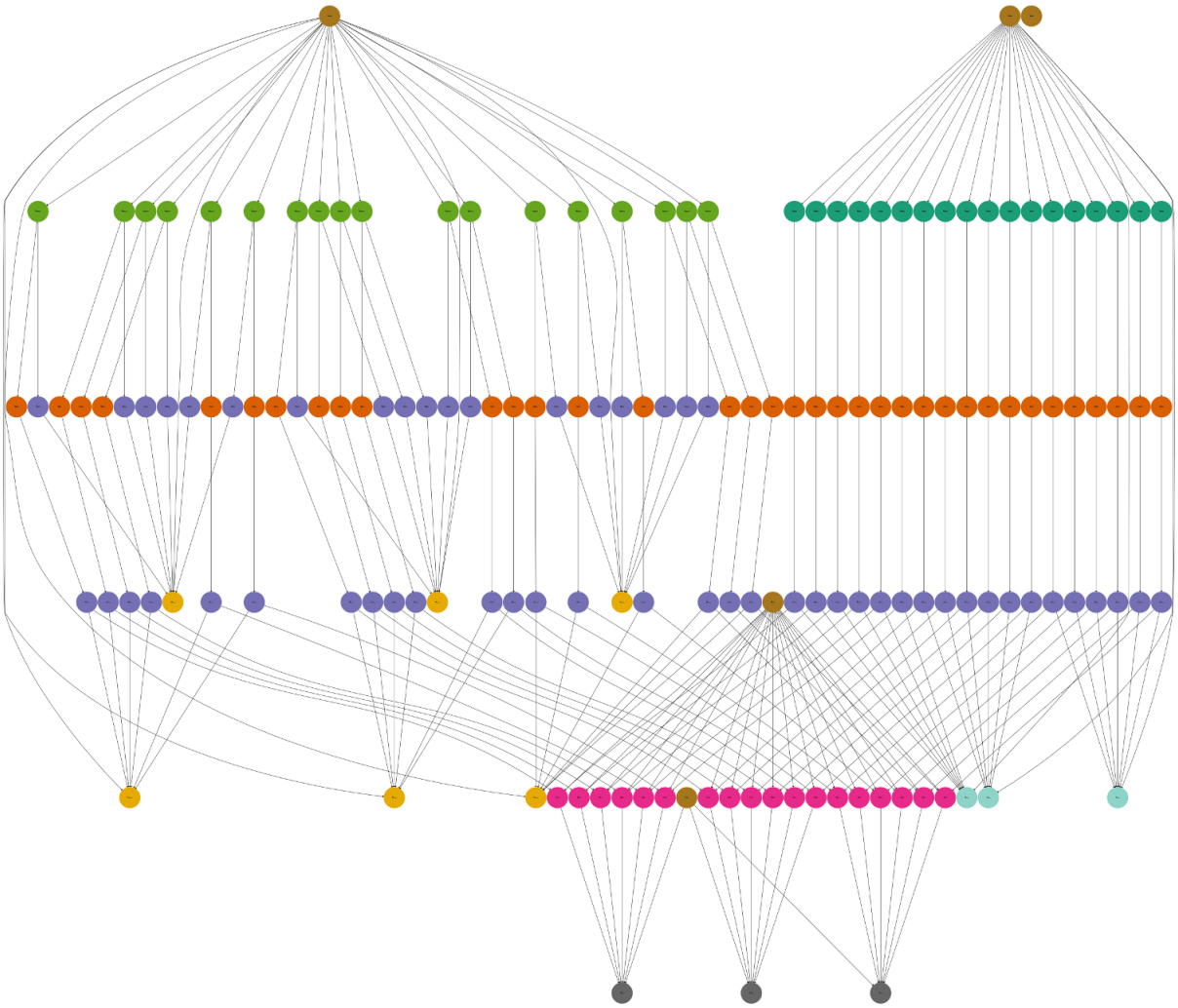


Figura 2.6: Workflow desenvolvido no sistema Pegasus (pegasus.isi.edu/applications).

O sistema também permite a transformação de dados entre diferentes linguagens como, por exemplo, XSLT, XQuery, Perl, entre outras, a fim de integrar diferentes serviços web. Por ser um dos primeiros gerenciadores de workflow disponíveis, esse sistema apresenta uma biblioteca contendo mais de 350 componentes disponíveis, que podem ser personalizados para o desenvolvimento de novos workflows. Os componentes disponíveis oferecem leitores e escritores de dados para diferentes formatos de arquivos como, por exemplo, planilhas de Excel e XML e um sistema de persistência de workflows próprio que permite o compartilhamento das pesquisas para diferentes grupos de estudo.

2.4.4 Sistemas de Workflows para Visualização

VisTrails

O projeto VisTrails [33, 9, 8, 3] foi desenvolvido em 2005 com o objetivo de gerenciar workflows de fluxo de dados destinados à visualização de resultados em diferentes métodos. Este projeto é uma ferramenta open-source utilizada para monitorar o processo de desenvolvimento de pesquisas,

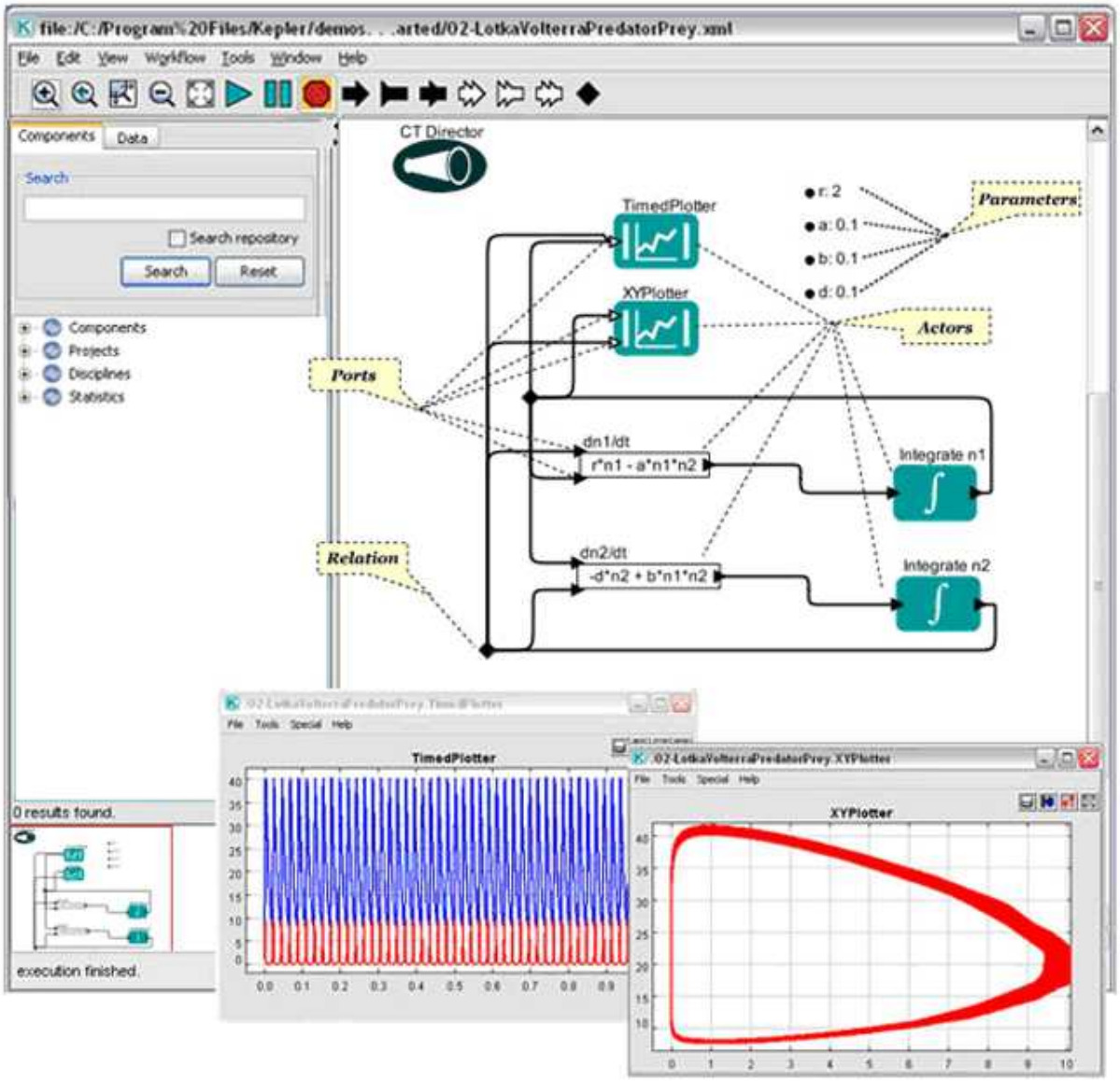


Figura 2.7: Interface do sistema Kepler (kepler-project.org/users/sample-workflows).

onde a evolução do workflow e a metodologia de exploração de dados são registradas. O sistema acompanha o desenvolvimento do workflow gerando arquivos em xml ou bancos de dados relacionais para representar cada alteração. Os arquivos são estruturados em uma árvore, onde cada nó representa um estágio de desenvolvimento do workflow e cada aresta representa uma ligação entre diferentes etapas de desenvolvimento. A partir da árvore de versões, é possível explorar as diferentes linhas de pesquisa, escolhendo um nó como ponto de partida para novas explorações de dados. A cada linha de pesquisa criada, uma nova subárvore é definida a partir do ponto de partida escolhido.

Outra característica importante desse sistema é o monitoramento das manipulações de dados envolvidas na execução de workflows. Cada resultado obtido pela execução de diferentes versões do workflow pode ser comparada em tabelas e é possível acompanhar a transformação dos dados até a obtenção dos resultados desejados. A visualização dos resultados pode ser feita sob diversas formas de representação e permite a integração com bibliotecas externas.

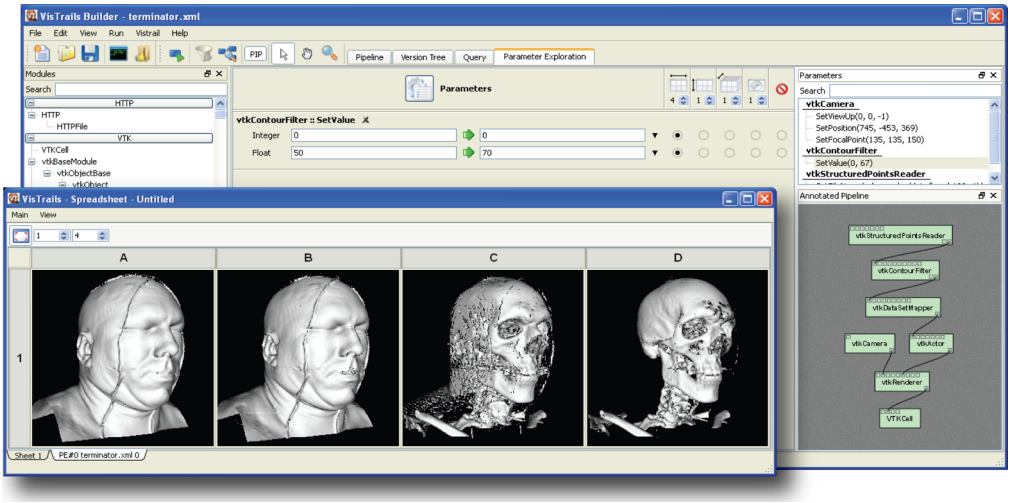


Figura 2.8: Interface do sistema Vistrails (www.vistrails.org/index.php/Documentation).

O projeto VTK (Visualization Toolkit) [5, 2] é uma vasta API de componentes voltados para o processamento e visualização de imagens. Esta API é muito utilizada pelos projetos VisTrails e VTK Designer na construção de workflows de diversas áreas como, por exemplo, análise de imagens médicas e geometria computacional, porém não se limita à exposição dos resultados a partir de imagens, podendo dar suporte à visualização dos resultados por gráficos, tabelas, árvores, etc. Utilizando essa API, o sistema proporciona um histórico detalhado do processo de exploração de dados para a comparação de resultados.

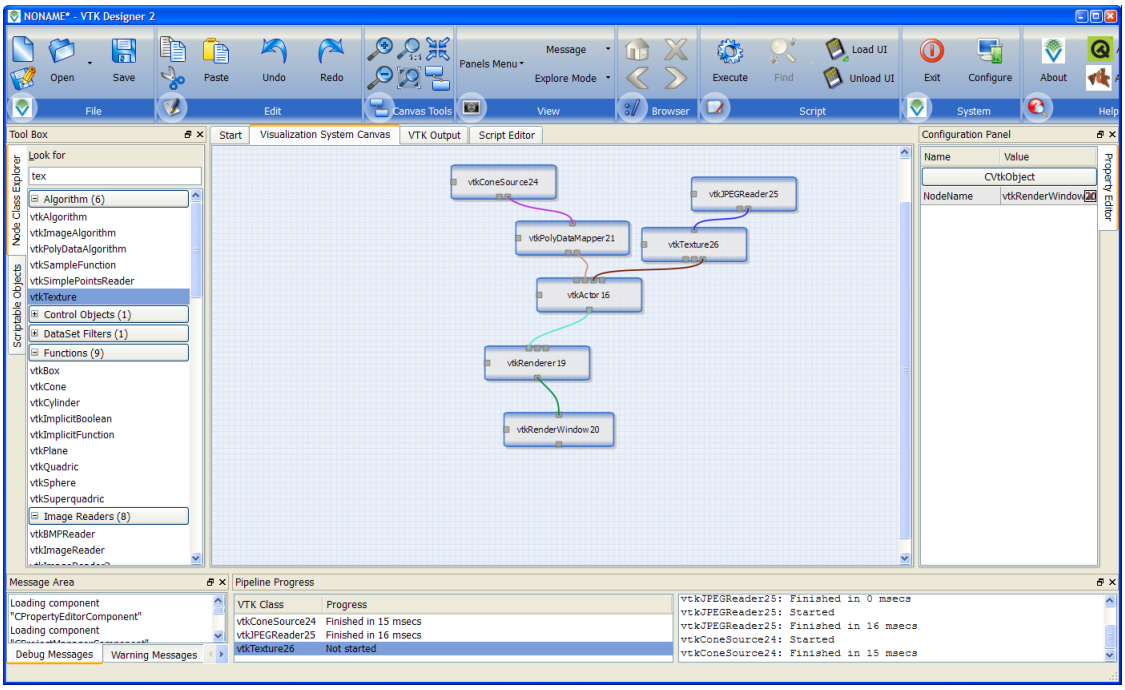


Figura 2.9: Interface do VTK Designer (www.thelins.se/johan/2007/10/vtk-designer.html).

2.5 Comentários Finais

Desenvolvemos este projeto visando contemplar as características apresentadas nesse capítulo, porém acrescentando algumas abordagens distintas. Ao contrário dos sistemas apresentados, o sistema proposto baseia-se em um modelo híbrido entre fluxos de dados e fluxos de controle, permitindo que o tráfego dos dados entre as atividades seja estabelecido de acordo com condições de controle a serem satisfeitas no decorrer do processo.

A contribuição deste trabalho é propor uma ferramenta simples, que forneça uma maneira intuitiva de desenvolvimento de aplicações para computação gráfica e possibilite a utilização dos recursos de GPU para a execução dos workflows. Outra aplicação para este projeto é a prática de ensino em disciplinas de programação para sistemas paralelos heterogêneos, onde o estudante pode aprender como utilizar estruturas de controle típicas de linguagens de programação, visualizar graficamente o desenvolvimento de aplicações contendo paralelismo e distinguir problemas onde é possível aproveitar os recursos de GPU para agilizar a resolução de seus problemas.

CAPÍTULO 3

Análise de Sólidos Elásticos pelo Método dos Elementos Finitos

3.1 Introdução

Neste capítulo apresentamos um resumo dos fundamentos e do pipeline do método dos elementos finitos (MEF) para análise estática de sólidos elásticos. Um corpo deformável é (perfeitamente) *elástico* se, partindo de uma forma inicial submetida à ação de um conjunto externo de forças, volve a essa forma inicial quando cessadas as forças causadoras da deformação. O MEF pode ser aplicado a vários outros e muito mais complexos problemas em ciências e engenharia, mas, visto que o projeto não tem como único objetivo a simulação de corpos deformáveis em geral, mas, principalmente, a construção de workflows para simulação de corpos deformáveis em ambientes paralelos heterogêneos formados por CPUs de vários núcleos e uma ou mais unidades de processamento gráfico (GPUs), restringimos a mecânica ao caso mais simples de corpos elásticos sob ação de forças estáticas. (Uma força é aplicada estaticamente quando o tempo de aplicação é suficientemente longo, podendo-se negligenciar impactos e outros efeitos dinâmicos decorrentes da aplicação.) Além disso, consideram-se apenas materiais que obedecem à Lei de Hooke, a qual estabelece uma relação linear entre as forças aplicadas e a deformação.

Na Seção 3.2 apresentamos as equações governantes da elasticidade linear e os passos de solução sequencial via MEF em CPU. Maiores detalhes podem ser encontrados em [35]. O objetivo aqui é apenas introduzir os termos necessários à compreensão das funcionalidades dos componentes disponíveis na API de elementos finitos desenvolvida em [34] (uma extensão daquela desenvolvida em [12]) e usados nos workflows do Capítulo 5. Na Seção 3.3 discutimos a implementação do pipeline em unidades de processamento gráfico (GPUs) com arquitetura CUDA (*compute unified device architecture*). Na Seção 3.4 descrevemos a análise com decomposição de domínio.

3.2 Análise Elastostática Sequencial em CPU

Seja um sólido definido por um domínio Ω e um contorno Γ, no qual são aplicados forças de volume e de superfície. Pode-se expressar o equilíbrio estático em cada ponto do sólido, em termos de tensões e forças aplicadas ao volume, em notação inicial, através da equação diferencial:

$$\sigma_{ji,j} + b_i = 0, \tag{3.1}$$

onde σ_{ji} representa o *tensor de tensões* e b_i representa a *força de volume* por unidade de volume. Na equação acima, $\sigma_{ji,j}$ denota, para $i = 1, 2, 3$, a soma das derivadas parciais

$$\frac{\partial \sigma_{ji}}{\partial x_j} = \frac{\partial \sigma_{1i}}{\partial x_1} + \frac{\partial \sigma_{2i}}{\partial x_2} + \frac{\partial \sigma_{3i}}{\partial x_3},$$

onde $x_1 \equiv x$, $x_2 \equiv y$ e $x_3 \equiv z$. Assim, a Equação (3.1) representa o sistema:

$$\begin{aligned}
\frac{\partial \sigma_{xx}}{\partial x} + \frac{\partial \sigma_{yx}}{\partial y} + \frac{\partial \sigma_{zx}}{\partial z} + b_x &= 0, \\
\frac{\partial \sigma_{xy}}{\partial x} + \frac{\partial \sigma_{yy}}{\partial y} + \frac{\partial \sigma_{zy}}{\partial z} + b_y &= 0, \\
\frac{\partial \sigma_{xz}}{\partial x} + \frac{\partial \sigma_{yz}}{\partial y} + \frac{\partial \sigma_{zz}}{\partial z} + b_z &= 0.
\end{aligned}$$

Se o sólido apresentar pequenos deslocamentos e deformações, pode-se escrever

$$\epsilon_{ij} = \frac{1}{2}(u_{i,j} + u_{j,i}), \tag{3.2}$$

onde ϵ_{ij} representa o *tensor de* (pequenas) *deformações* e u_i é o campo de *deslocamentos*. Se for assumido que o material do sólido é homogêneo e perfeitamente elástico, sua *relação constitutiva* é expressa pela Lei de Hooke

$$\sigma_{ij} = C_{ijlm}\epsilon_{lm}. \tag{3.3}$$

Para materiais isotrópicos, o *tensor de módulos elásticos* C_{ijlm} é dado por

$$C_{ijlm} = \lambda \delta_{ij}\delta_{lm} + \mu(\delta_{il}\delta_{jm} + \delta_{im}\delta_{jl}), \tag{3.4}$$

onde λ e μ são as *constantes de Lame* e δ_{ij} é o delta de Kronecker. Substituindo a Equação (3.4) na Equação (3.3), a versão isotrópica da Lei de Hooke fica

$$\sigma_{ij} = \lambda \epsilon_{kk}\delta_{ij} + 2\mu\epsilon_{ij}. \tag{3.5}$$

Pode-se relacionar as constantes de Lame ao *módulo de elasticidade transversal* E, *coeficiente de Poisson* ν e *módulo de elasticidade transversal* G na forma

$$G = \mu = \frac{E}{2(1+\nu)} \qquad \text{e} \qquad \lambda = \frac{\nu\, E}{(1+\nu)(1-\nu)}. \tag{3.6}$$

Considerando a Equação (3.6) e substituindo a Equação (3.2) na Equação (3.5) e a equação resultante na Equação (3.1), obtém-se

$$G\, u_{i,jj} + \frac{G}{1-2\nu}\, u_{j,ij} + b_i = 0. \tag{3.7}$$

A Equação (3.7) é conhecida como *equação de Navier-Cauchy* da elasticidade. Tal equação expressa o equilíbrio estático de um sólido em função do campo de deslocamentos u_i e da força de volume b_i aplicada ao sólido.

A unicidade da solução da Equação (3.7) deve satisfazer duas espécies de *condições de contorno*: deslocamentos prescritos (condições de contorno essenciais)

$$u_i = \overline{u}_i \qquad \text{em } \Gamma_1 \tag{3.8}$$

e forças de superfície prescritas (condições de contorno naturais)

$$p_i = \sigma_{ji} n_j = \overline{p}_i \qquad \text{em } \Gamma_2, \tag{3.9}$$

onde $\Gamma = \Gamma_1 + \Gamma_2$ é a superfície do contorno do sólido e n_j é a normal de Γ_2.

A Equação (3.1), derivada da teoria da mecânica do contínuo, admitidas as hipóteses simplificadoras listadas anteriormente, juntamente com as condições de contorno dadas pela Equação (3.8) e pela Equação (3.9), caracteriza o modelo matemático de um sólido elástico.

Uma solução (aproximada, para o caso geral de geometria e condições de contorno) do modelo matemático pode ser obtida através de métodos numéricos tais como o MEF. Este se baseia na subdivisão de um domínio Ω em uma malha definida por um conjunto de células, os elementos finitos, conectadas através de nós. Um elemento finito é uma sub-região do domínio Ω, definido pela sequência de nós sobre os quais o mesmo incide. A essa sequência dá-se o nome de lista de incidência do elemento.

A formulação do MEF — derivada de princípios variacionais ou, mais genericamente, do método dos resíduos ponderados [35] — reduz o problema contínuo à solução do seguinte sistema de equações lineares:

$$\mathbf{KU} = \mathbf{F}, \tag{3.10}$$

onde **K** é a *matriz de rigidez global* do modelo, **F** é o *vetor de carregamentos nodais equivalentes* e **U** é o vetor de deslocamentos nodais de todos os nós do elemento, os quais constituem as incógnitas do problema. (Uma vez obtido o vetor **U**, deslocamentos em pontos quaisquer do domínio podem ser interpolados em um elemento finito a partir de suas *funções de forma* e dos deslocamentos nodais.)

A matriz de rigidez **K** e o vetor **F** são computados a partir da contribuição individual de cada elemento finito. Assim, com a subdivisão do domínio do sólido em NE elementos finitos, tem-se

$$\sum_{e=1}^{\mathrm{NE}}{}^{*} \mathbf{K}^{(e)}\mathbf{U} = \sum_{e=1}^{\mathrm{NE}}{}^{*} \mathbf{f}^{(e)} + \sum_{p=1}^{\mathrm{NN}}{}^{*} \mathbf{f}^{(p)}, \tag{3.11}$$

onde $\mathbf{K}^{(e)}$ é a matriz de rigidez local do elemento e, $\mathbf{f}^{(e)}$ corresponde às forças de volume aplicadas em cada elemento, $\mathbf{f}^{(p)}$ às forças externas aplicadas diretamente aos nós e NN o número de nós da malha. (O símbolo $\sum^{*}$ usado na equação acima deve ser entendido não somente no sentido usual de somatório, mas também como um operador de "colocação", isto é, que soma cada elemento da matriz ou vetor local na linha e coluna correspondentes na matriz ou vetor global.)

Como exemplo de elemento finito 3D, considere o tetraedro linear (quatro faces triangulares e quatro nós). A matriz de rigidez local do tetraedro tem 3×4 elementos (3 *graus de liberdade* — deslocamentos nas direções x, y e z — em cada um dos 4 nós) e é definida pela expressão:

$$\mathbf{K}^{(e)} = \frac{1}{6V}\, \mathbf{B}^T\, \mathbf{E}\, \mathbf{B}, \tag{3.12}$$

onde V é o volume do tetraedro, **E** é a matriz de elasticidade e **B** é uma matriz definida a seguir. O volume do tetraedro pode ser encontrado através da equação:

$$V = \frac{1}{6} \det \begin{bmatrix} 1 & 1 & 1 & 1 \\ x_1 & x_2 & x_3 & x_4 \\ y_1 & y_2 & y_3 & y_4 \\ z_1 & z_2 & z_3 & z_4 \end{bmatrix}, \tag{3.13}$$

onde $(x_i,\ y_i,\ z_i)$ são as coordenadas da posição do nó $1 \leq i \leq 4$ no tetraedro tridimensional. A matriz de elasticidade é dada por:

$$\mathbf{E} = \begin{bmatrix} \xi_1 & \xi_2 & \xi_2 & 0 & 0 & 0 \\ \xi_2 & \xi_1 & \xi_2 & 0 & 0 & 0 \\ \xi_2 & \xi_2 & \xi_1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \xi_3 & 0 & 0 \\ 0 & 0 & 0 & 0 & \xi_3 & 0 \\ 0 & 0 & 0 & 0 & 0 & \xi_3 \end{bmatrix}, \tag{3.14}$$

onde os valores de ξ_1, ξ_2 e ξ_3 são definidos em termos das propriedades do material pelas seguintes equações:

$$\xi_1 = \frac{E}{(1+\nu)(1-2\nu)}(1-\nu), \quad \xi_2 = \frac{E}{(1+\nu)(1-2\nu)}\nu, \quad \xi_3 = \frac{E}{(1+\nu)(1-2\nu)}(\frac{1}{2}-\nu), \tag{3.15}$$

sendo E é o módulo de elasticidade e ν é o coeficiente de Poisson do material elástico que constitui o tetraedro. Tomando-se

$$\begin{aligned} a_1 &= y_2 z_{43} - y_3 z_{42} + y_4 z_{32}, \\ a_2 &= -y_1 z_{43} + y_3 z_{41} - y_4 z_{31}, \\ a_3 &= y_1 z_{42} - y_2 z_{41} + y_4 z_{21}, \\ a_4 &= -y_1 z_{32} + y_2 z_{31} - y_3 z_{21}, \\ b_1 &= -x_2 z_{43} + x_3 z_{42} - x_4 z_{32}, \\ b_2 &= x_1 z_{43} - x_3 z_{41} + x_4 z_{31}, \\ b_3 &= -x_1 z_{42} + x_2 z_{41} - x_4 z_{21}, \\ b_4 &= x_1 z_{32} - x_2 z_{31} + x_3 z_{21}, \\ c_1 &= x_2 y_{43} - x_3 y_{42} + x_4 y_{32}, \\ c_2 &= -x_1 y_{43} + x_3 y_{41} - x_4 y_{31}, \\ c_3 &= x_1 y_{42} - x_2 y_{41} + x_4 y_{21}, \\ c_4 &= -x_1 y_{32} + x_2 y_{31} - x_3 y_{21}, \end{aligned} \tag{3.16}$$

onde $x_{ij} = x_i - x_j$, $y_{ij} = y_i - y_j$ e $z_{ij} = z_i - z_j$, a matriz **B** é definida como:

$$\mathbf{B} = \frac{1}{6V} \begin{bmatrix} a_1 & a_2 & a_3 & a_4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & b1 & b2 & b3 & b4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & c1 & c2 & c3 & c4 \\ b1 & b2 & b3 & b4 & a1 & a2 & a3 & a4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & c1 & c2 & c3 & c4 & b1 & b2 & b3 & b4 \\ c1 & c2 & c3 & c4 & 0 & 0 & 0 & 0 & a1 & a2 & a3 & a4 \end{bmatrix}. \tag{3.17}$$

A matriz de rigidez global na (3.10) é simétrica, esparsa e tem característica de banda (isto é, os elementos não nulos residem em uma faixa em torno da diagonal principal). A fim de se tirar proveito da simetria e esparsidade, é usual em aplicações do MEF a utilização de esquemas de representação de matrizes esparsas que armazenam em memória de computador apenas seus elementos não nulos. Por exemplo, pode-se armazenar **K** em um arranjo retangular $n \times s$, onde n é o número de incógnitas do sistema e s é a largura da semi-banda (que inclui a diagonal principal), uma vez que todos os elementos fora da banda são nulos. Em problemas tridimensionais, cada nó da malha pode se deslocar nas direções x, y e z, ou seja, tem 3 graus de liberdade (ou

DOFs). Assim, o número de incógnitas do sistema é $3 \times$ NN. A largura da (semi-)banda, por sua vez, depende da numeração dos graus de liberdade dos nós que incidem em cada elemento finito. Por isso, neste esquema, utilizam-se métodos de renumeração dos graus de liberdade nodais com o propósito de diminuir a largura da banda e, consequentemente, o espaço necessário ao armazenamento dos elementos não nulos da matriz de rigidez. Podem-se empregar métodos distintos de armazenamento da matriz de rigidez e de renumeração dos graus de liberdade, sendo essa última tarefa responsabilidade de um componente chamado *numerador de DOFs*.

A matriz de rigidez global é também singular e, portanto, não admite inversa. A fim de tornar o sistema (3.10) determinado, é necessário um número suficiente de deslocamentos prescritos, isto é, condições de contorno essenciais, sendo o caso mais comum a versão homogênea da Equação (3.8), ou seja, deslocamentos nodais prescritos iguais a zero. A imposição de tais condições de contorno no domínio discretizado pode ser implementada pela remoção, na matriz de rigidez global, das linhas e colunas correspondentes aos graus de liberdade nulificados. Igualmente, são removidas as linhas correspondentes dos vetores de deslocamentos e de esforços nodais equivalentes. (Há maneiras distintas de tratar das condições de contorno, sendo essa tarefa de responsabilidade de um componente chamado *manipulador de restrições*.) Feito isso, a solução do sistema linear reduzido determina os graus de liberdade incógnitos. Novamente, há vários métodos para resolução de sistemas lineares, por exemplo, o método direto de Gauss ou o iterativo gradientes conjugados.

Em síntese, o esquema computacional do MEF, implementado em CPU, é composto pelos seguintes passos:

1. **Discretização do domínio**. O domínio Ω é subdividido em NE elementos finitos sobre os quais é aproximado o campo de deslocamentos com a utilização de funções de interpolação e de parâmetros nodais. Essa etapa é chamada pré-processamento.

2. **Computação das contribuições dos elementos**. Para cada elemento finito e, determina-se a matriz de rigidez $\mathbf{K}^{(e)}$ e os carregamentos nodais $\mathbf{f}^{(e)}$.

3. **Montagem do sistema linear**. Montam-se o lado esquerdo e o lado direito do sistema, com base nas contribuições de todos os NE elementos finitos do e também dos carregamentos aplicados diretamente nos NN nós do modelo.

4. **Introdução das condições de contorno**. A matriz $\mathbf{K}$, originalmente singular, é transformada em uma matriz regular, considerando-se um número apropriado de condições de contorno essenciais, as quais definem a vinculação do sólido.

5. **Solução do sistema linear**. Com a solução do sistema (na implementação, pelo método dos gradientes conjugados, dado que a matriz de rigidez é simétrica positiva definida) determinam-se os deslocamentos incógnitos nos pontos e direções onde esses valores não são prescritos.

3.3 Análise Elastostática em GPU

As GPUs consideradas neste trabalho são de arquitetura CUDA. No modelo de programação paralela de CUDA, uma aplicação consiste de um programa sequencial que executa no *host* (a CPU, para simplificar) e que pode invocar programas paralelos, os *kernels*, que executam em um dispositivo paralelo (a GPU). Um *kernel* é um processo SPMD (*single program multiple data*) que é executado em várias threads em paralelo. Cada thread executa o mesmo programa sequencial escalar.

O programador organiza as threads de um *kernel* em uma grade de blocos de threads. As threads de um determinado bloco podem cooperar entre si através do uso de barreiras de sincronização e de uma memória compartilhada privada do bloco (de acesso até 600 vezes mais rápido que a memória global). A criação, escalonamento e gerenciamento de threads é feita totalmente em hardware.

Uma GPU CUDA é composta de um arranjo de multiprocessadores (MPs) equipados com 8 processadores escalares cada. Em um MP podem residir até 1024 threads ativas. GPUs tais como a Tesla C1060 contém 30 multiprocessadores, em um total de até 30K threads ativas. A fim de gerenciar este número eficientemente, a GPU emprega uma arquitetura SIMT (*single instruction multiple thread*) na qual as threads de um mesmo bloco são executadas em grupos de 32 chamado *warp*.

Todas as threads de um *warp* são sincronizadas pelo hardware para executar uma única instrução ao mesmo tempo. As threads de um *warp* podem cada qual seguir seu próprio caminho de execução, sendo a possível divergência dai resultante gerenciada automaticamente pelo hardware. Contudo, é mais eficiente que as threads de um *warp* sigam todas o mesmo caminho de execução, o que implica que não se deve ter threads ociosas (devido a algum desvio condicional, por exemplo) e outras não.

As threads de um *warp* também podem, cada qual, usar endereços arbitrários para acessar memória global da GPU. A leitura ou escrita a posições aleatórias da memória global (por threads de um mesmo *warp*) resulta em divergência de memória e requer que a GPU realize uma transação de memória para cada thread. Por outro lado, se as posições sendo acessadas são suficientemente próximas, as transações de memória por thread podem ser *coalescidas*, resultando um uma maior eficiência de acesso. A memória global é conceitualmente organizada em uma sequência de segmentos de 128 bytes. Requisições a dados na memória global são atendidas para 16 threads (meio *warp*) de cada vez. O número de transações de memória realizadas para um meio *warp* é igual ao número de segmentos distintos acessados por suas threads. Portanto, se todas as threads de um meio *warp* acessam posições de um mesmo segmento, somente uma transação totalmente coalescida é realizada; por outro lado, se cada thread acessa um segmento distinto, 16 transações de memória são realizadas sequencialmente.

A execução de um programa em GPU envolve pelo menos três passos básicos. Primeiro, deve-se alocar memória global da GPU para armazenamento de dados de entrada e saída. Usualmente, os dados de entrada gerados em CPU e transferidos da memória da CPU para a memória alocada da GPU. O segundo passo consiste na computação paralela desses dados em GPU, o que é feito pelo lançamento pela CPU de um ou mais kernels. Por último, ocorre a transferência dos dados de saída resultantes do processamento de volta para a memória da CPU. As operações de transferência de dados entre CPU e GPU têm grande latência e acabam por muitas vezes se tornando o gargalo do programa. Além disso, a memória da GPU é, quase sempre, menor que a memória da CPU, o que incapacita a computação em GPU para grande volume de dados. Nestes casos, é preciso considerar abordagens baseadas em divisão de dados e reagrupação.

O esquema computacional do MEF para análise elastostática em GPU adiciona dois passos ao esquema em CPU: (1) após a discretização do domínio (passo 1), os dados são transferidos para memória global alocada em GPU; e (2) após a solução do sistema linear (passo 5), o vetor solução é transferido de volta para a memória da CPU. A montagem e solução do sistema linear são executadas inteiramente em GPU. O passo de montagem é modificado como explicado a seguir.

Montagem do Sistema Linear em GPU

Como visto anteriormente, a matriz de rigidez global **K** e o vetor **F** são calculados a partir das contribuições de cada elemento finito. Na análise em GPU, a abordagem mais direta para montagem do sistema linear em paralelo seria um kernel que aplica uma thread para cada elemento finito do domínio. Cada thread seria responsável por computar a matriz de rigidez e o vetor de esforços nodais de um elemento e adicioná-los aos lados esquerdo e direito do sistema linear, respectivamente. Como este processamento se daria em paralelo, haveria concorrência de escrita, isto é, uma condição de corrida, quando duas ou mais threads responsáveis por elementos finitos que incidem em um mesmo vértice v fossem adicionar as contribuições aos elementos de **K** e **F** correspondentes aos graus de liberdade do nó compartilhado v. Em GPU, não há como sincronizar tal operação de escrita: uma e somente uma thread efetuará a escrita, gerando uma inconsistência no sistema linear.

Por causa da condição de corrida decorrente da abordagem descrita acima, foram adotadas estratégias alternativas para montagem da matriz de rigidez em GPU, baseadas na construção a partir das contribuições nodais e também em um esquema de coloração de grafos para a malha. Neste, os elementos finitos são rotulados com uma cor tal que elementos da mesma cor não compartilham nenhum vértice e, portanto, nenhum grau de liberdade, conforme ilustrado para o caso 2D na Figura 3.1. Após a coloração, são geradas tantas partições quanto o número de cores distintas dos elementos da malha. A montagem em paralelo pode ser agora executada com segurança, partição por partição, uma vez que os elementos de uma partição não compartilham vértices e, portanto, não geram condições de corrida a **K** e **F**.

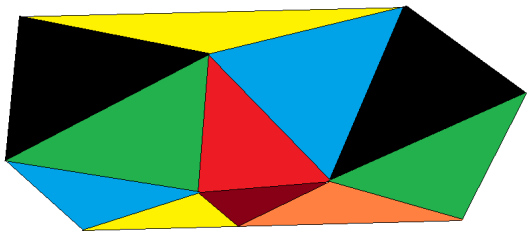


Figura 3.1: Uma malha triangular 2D com elementos coloridos.

3.4 Análise por Decomposição de Domínio

Para análise de domínios muitos elementos e/ou nós em CPUs e GPUs, emprega-se um método de decomposição de domínio conhecido como *subestruturação*. A subestruturação ou particionamento estrutural consiste na divisão de um domínio em um número de *subdomínios*, ou subestruturas, cujas fronteiras podem ser especificadas arbitrariamente. Por conveniência, geralmente o particionamento estrutural é feito de acordo com o particionamento físico. Se cada subestrutura possui propriedades próprias de rigidez, cada uma delas pode ser tratada como um elemento estrutural complexo. Segue que os métodos de análise podem também ser aplicados à estrutura particionada. Encontrados os deslocamentos ou forças nas fronteiras das subestruturas, cada uma pode ser analisada separadamente, sob a ação de deslocamentos ou forças conhecidas em suas fronteiras.

No MEF, cada subestrutura é analisada separadamente, assumindo que suas fronteiras comuns com subestruturas adjacentes estão completamente fixas. Após isso, tais fronteiras são "relaxadas" simultaneamente e os deslocamentos reais das mesmas são determinados a partir das equações de equilíbrio de forças nas junções de fronteira. A solução dos deslocamentos das fronteiras envolve um número de incógnitas reduzido quando comparado à solução da estrutura não particionada. Cada estrutura pode ser analisada separadamente sob carga e deslocamentos de fronteira conhecidos, o que pode ser feito facilmente, uma vez que as matrizes envolvidas são de tamanho relativamente pequeno.

Conforme visto na Seção 3.2 o equilíbrio estático de um sólido é representado, no MEF, pela equação

$$\mathbf{K}\,\mathbf{U} = \mathbf{F}. \tag{3.10-repetida}$$

Com a divisão do sólido em subestruturas são introduzidas fronteiras interiores no domínio a ser analisado. A matriz coluna dos deslocamentos comuns a mais de uma subestrutura será denotada por $\mathbf{U}_b$ e a matriz de deslocamentos dos pontos interiores denotada por $\mathbf{U}_i$. Representando as matrizes de forças externas de fronteira e interiores por, respectivamente, $\mathbf{F}_b$ e $\mathbf{F}_i$, pode-se reescrever a Equação (3.10) como:

$$\begin{bmatrix} \mathbf{K}_{bb} & \mathbf{K}_{bi} \\ \mathbf{K}_{ib} & \mathbf{K}_{ii} \end{bmatrix} \begin{bmatrix} \mathbf{U}_b \\ \mathbf{U}_i \end{bmatrix} = \begin{bmatrix} \mathbf{F}_b \\ \mathbf{F}_i \end{bmatrix}. \tag{3.18}$$

Assume-se, agora, que os deslocamentos totais da estrutura podem ser calculados pela sobreposição de duas matrizes tais que

$$\mathbf{U} = \mathbf{U}^{(\alpha)} + \mathbf{U}^{(\beta)}, \tag{3.19}$$

onde $\mathbf{U}^{(\alpha)}$ representa a matriz dos deslocamentos devidos a $\mathbf{F}_i$ com $\mathbf{U}_b = 0$, e $\mathbf{U}^{(\beta)}$ representa as correções necessárias aos deslocamentos $\mathbf{U}^{(\alpha)}$ para permitir os deslocamentos de fronteira $\mathbf{U}^{(\beta)}$ com $\mathbf{F}_i = 0$. Dessa forma, a Equação (3.19) pode ser reescrita como:

$$\mathbf{U} = \begin{bmatrix} \mathbf{U}_b \\ \mathbf{U}_i \end{bmatrix} = \begin{bmatrix} \mathbf{U}_b^{(\alpha)} \\ \mathbf{U}_i^{(\alpha)} \end{bmatrix} + \begin{bmatrix} \mathbf{U}_b^{(\beta)} \\ \mathbf{U}_i^{(\beta)} \end{bmatrix}. \tag{3.20}$$

Na Equação (3.20) o último termo representa as correções devidas ao relaxamento das fronteiras, no qual, por definição:

$$\mathbf{U}_b^{(\alpha)} = 0. \tag{3.21}$$

De maneira similar, as forças externas representadas em $\mathbf{F}$ podem ser separadas em

$$\mathbf{F} = \mathbf{F}^{(\alpha)} + \mathbf{F}^{(\beta)}, \tag{3.22}$$

ou ainda,

$$\mathbf{F} = \begin{bmatrix} \mathbf{F}_b \\ \mathbf{F}_i \end{bmatrix} = \begin{bmatrix} \mathbf{F}_b^{(\alpha)} \\ \mathbf{F}_i^{(\alpha)} \end{bmatrix} + \begin{bmatrix} \mathbf{F}_b^{(\beta)} \\ \mathbf{F}_i^{(\beta)} \end{bmatrix}, \tag{3.23}$$

onde, por definição, tem-se que

$$\mathbf{F}_i^{(\alpha)} = \mathbf{F}_i, \tag{3.24}$$

$$\mathbf{F}_i^{(\beta)} = 0. \tag{3.25}$$

Quando as fronteiras da subestrutura estão fixas, pode-se mostrar, utilizando a Equação (3.18), que

$$\mathbf{U}_i^{(\alpha)} = \mathbf{K}_{ii}^{-1}\mathbf{F}_i, \tag{3.26}$$

$$\mathbf{F}_b^{(\alpha)} = \mathbf{K}_{bi}\mathbf{K}_{ii}^{-1}\mathbf{F}_i = \mathbf{R}_b. \tag{3.27}$$

Na Equação (3.27) $\mathbf{F}_b^{(\alpha)}$ representa as reações necessárias para garantir que $\mathbf{U}_b$ seja igual a $\mathbf{0}$, sob a ação das forças internas $\mathbf{F}_i$. Com o relaxamento das fronteiras, os deslocamentos $\mathbf{U}^{(\beta)}$ podem ser determinados pela mesma Equação (3.18) de tal forma que

$$\mathbf{U}_i^{(\beta)} = -\mathbf{K}_{ii}^{-1}\mathbf{K}_{ib}\mathbf{U}_b^{(\beta)}, \tag{3.28}$$

$$\mathbf{U}_b^{(\beta)} = [\mathbf{K}_{bb} - \mathbf{K}_{bi}\mathbf{K}_{ii}^{-1}\mathbf{K}_{ib}]^{-1}\mathbf{F}_b^{(\beta)}, \tag{3.29}$$

onde

$$\mathbf{K}_b = \mathbf{K}_{bb} - \mathbf{K}_{bi}\mathbf{K}_{ii}^{-1}\mathbf{K}_{ib} \tag{3.30}$$

representa a matriz de rigidez das fronteiras. Com o uso da Equação (3.23) e da Equação (3.27), pode-se determinar

$$\mathbf{F}_b^{(\beta)} = \mathbf{F}_b - \mathbf{F}_b^{(\alpha)} = \mathbf{F}_b - \mathbf{K}_{bi}\mathbf{K}_{ii}^{-1}\mathbf{F}_i = \mathbf{S}_b. \tag{3.31}$$

Com os deslocamentos de fronteira iguais a zero, tem-se as subestruturas isoladas umas das outras. Dessa maneira, a aplicação de uma força interna causa deslocamentos em uma única subestrutura. Os deslocamentos internos com as fronteiras fixas podem ser calculados separadamente para cada subestrutura, utilizando a Equação (3.26). Os deslocamentos de fronteira $\mathbf{U}_b^{(\beta)}$ são calculados com a Equação (3.29), que envolve a inversão de $\mathbf{K}_b$, a qual possui ordem muito menor do que a matriz de rigidez completa $\mathbf{K}$.

Os passos da análise com subestruturação são:

1. **Discretização do domínio e carregamento das condições de contorno**: etapa de pré-processamento na qual a malha de elementos é obtida e as condições de contorno (restrições e carregamentos são aplicadas sobre o domínio.
2. **Particionamento do domínio em n subdomínios**: a malha é particionada em n partes com a utilização de um algoritmo para particionamento de grafos; o domínio agora pode ser visto como sendo composto por n "super-elementos", correspondentes aos subdomínios.
3. **Computação das contribuições dos elementos**: para que o sistema global possa ser montado, cada subdomínio deve formar sua matriz de rigidez e seu vetor de esforços, independentemente da contribuições dos subdomínios vizinhos.
4. **Montagem do sistema global equivalente às interfaces**: de posse das contribuições de cada subdomínio, o sistema global pode ser montado.
5. **Solução do sistema global**: o sistema global é resolvido.
6. **Atualização dos resultados nos subdomínios**: os subdomínios atualizam suas soluções internas com os resultados recebidos das interfaces.

Os componentes envolvidos na análise por decomposição de domínio serão introduzidos nos exemplos do Capítulo 5.

3.5 Comentários Finais

Neste capítulo apresentamos um resumo dos passos de análise estática de sólidos elásticos pelo método dos elementos finitos, tal como implementado na API desenvolvida em [34]. A vários

componentes dessa API foram associados atividades a partir das quais construímos workflows paramétricos para análise via MEF em CPU e GPU, com ou sem particionamento de domínio, conforme exemplificado no Capítulo 5.

CAPÍTULO 4

Descrição do Sistema

4.1 Introdução

Neste capítulo descrevemos o sistema de workflows desenvolvido neste trabalho. Na Seção 4.2 introduzimos os conceitos envolvidos no modelo de representação de workflows proposto. Na Seção 4.3 descrevemos as classes de ojetos que implementam o modelo, detalhando suas estruturas e as relações entre os componentes. Na Seção 4.4 apresentamos a biblioteca básica desenvolvida e exemplificamos a criação de classes para a representação de novos tipos de dados e atividades para extensão da API. Na Seção 4.5 abordamos a interface proposta para a utilização do modelo no desenvolvimento de workflows paramétricos e suas principais funções, e também o processo de desenvolvimento e execução de um workflow através da utilização da interface.

4.2 Modelo de Workflows Paramétricos

No contexto deste trabalho, um workflow representa um processo computacional composto por conjuntos de dois elementos básicos chamados *atividades* e *canais*. As atividades são *componentes conectáveis* que representam conjuntos de instruções executáveis. As atividades são utilizadas para manipular dados e são graficamente representadas por retângulos cinza claro. Conforme apresentado na Figura 4.1, cada atividade é composta pelos seguintes elementos:

- Um rótulo que descreve o tipo da atividade, situado no centro do retângulo;
- Uma porta de entrada e uma porta de saída de sinais de controle, representadas por quadrados cinza escuro localizados nos cantos superior e inferior esquerdos, respectivamente.

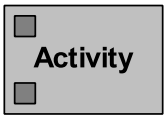


Figura 4.1: Representação de atividade básica.

As portas de sinais de controle são *conectores* que permitem a comunicação com outras atividades para criar um fluxo de controle. O fluxo de controle representa a ordem de precedência

de execução entre as atividades, sendo definido pela ligação entre a porta de saída de da atividade de maior precedência à porta de entrada da atividade de menor precedência. As *conexões* entre as portas são feitas por *canais*, responsáveis pela transferência de sinais entre as atividades de origem e destino e representados por segmentos curvilíneos. A Figura 4.2 demonstra a definição de um fluxo de controle, onde a atividade do tipo `Activity A` é conectada à atividade do tipo `Activity B`, indicando que somente após concluir sua execução a atividade de destino poderá ser executada.

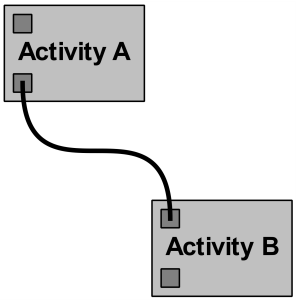


Figura 4.2: Fluxo de controle entre duas atividades.

As atividades podem ser classificadas entre produtoras, consumidoras e filtros, de acordo com a forma de obtenção de dados e disposição dos resultados gerados por sua execução. As atividades são chamadas de produtoras quando sua execução transforma dados originados de uma fonte externa ao workflow em objetos de dados. Uma fonte de dados externa pode ser, por exemplo, um arquivo ou uma entrada de dados realizada pelo usuário. As atividades produtoras possuem uma ou mais portas de saída de dados, utilizadas para transferir os objetos de dados resultantes de sua execução. Cada porta de saída de dados possui um tipo de dado, que corresponde ao tipo de objeto de dados que a atividade produz. As portas de saída de dados são representadas por quadrados cinza claro localizados à direita da porta de saída de sinais de controle, conforme demonstrado pela atividade `MatrixReader` representada na Figura 4.3, que lê o arquivo indicado pela porta de entrada `path` e produz um objeto de dados do tipo `Matrix`.

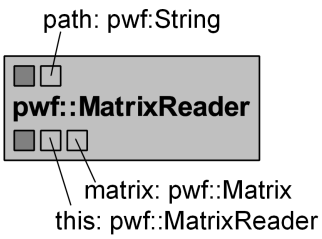


Figura 4.3: Atividade `MatrixReader`.

As atividades são chamadas de consumidoras quando sua execução transforma dados originados de uma outra atividade em uma saída externa ao workflow como, por exemplo, um arquivo de saída, uma imagem, um gráfico entre outros. As atividades consumidoras possuem uma ou mais portas de entrada de dados, utilizadas para receber os objetos de dados necessários para sua execução. Cada porta de entrada de dados possui um tipo de dado, que corresponde ao tipo de objeto de dados que a atividade é capaz de manipular. As portas de entrada de dados são representadas por quadrados cinza claro localizados à direita da porta de entrada de sinais de controle, conforme demonstrado pela atividade `MatrixWriter` representada na Figura 4.4, que converte

um objeto de dados do tipo `Matrix` em um arquivo de saída cujo nome foi definido pela porta `path`.

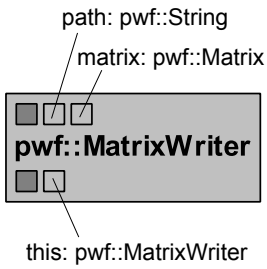


Figura 4.4: Atividade `MatrixWriter`.

As atividades filtro são aquelas que recebem dados de outras atividades e transformam-os em sua execução, resultando em novos dados que serão disponibilizados para outras atividades. A Figura 4.5 apresenta a atividade filtro `MulMatrix`, que recebe em suas portas de entrada de dados dois objetos do tipo `Matrix` para realizar a multiplicação entre matrizes, resultando em um novo objeto do tipo `Matrix` retornado em sua porta de saída de dados.

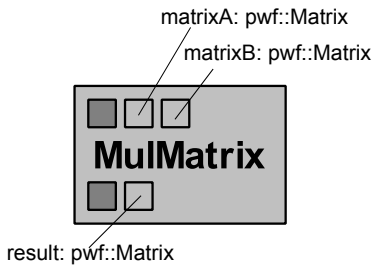
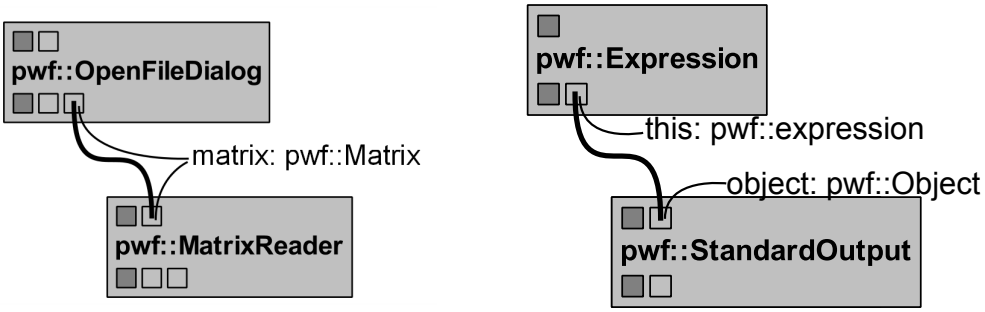


Figura 4.5: Atividade `MulMatrix`.

As atividades que manipulam dados podem se relacionar através da criação de canais entre portas cujos tipos de dados sejam compatíveis. Um tipo de dados é compatível a outro quando ambos são iguais, como ocorre entre as atividades da Figura 4.6(a), ou quando o tipo de dados de origem é um subtipo do tipo de dados do destino, conforme demonstrado pela Figura 4.6(b).

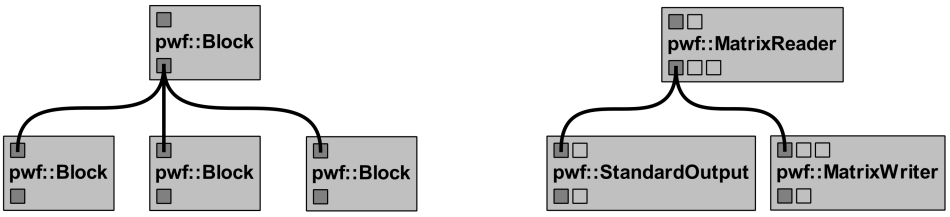


(a) Conexão de portas de mesmo tipo

(b) Conexão de portas com tipos distintos

Figura 4.6: Conexões entre portas de tipos de dados compatíveis.

As portas de saída de dados ou sinais de controle podem servir de origem a vários canais conectados a portas distintas, servindo como um ponto de paralelismo do processo. Em um fluxo de sinais de controle, o paralelismo indica que após a execução da atividade de origem, todas as atividades de destino deixam de depender da origem e podem ser executadas em paralelo. Em um fluxo de dados, o paralelismo indica que o resultado produzido pela atividade de origem poderá ser utilizado por todas as atividades de destino e, após a transferência do resultado, as atividades podem ser executadas paralelamente. As figuras 4.7(a) e 4.7(b) representam, respectivamente, o paralelismo realizado em um fluxo de controle e um fluxo de dados.



(a) Fluxo de controle (b) Fluxo de dados

Figura 4.7: Paralelismo entre atividades.

De maneira semelhante, as portas de entrada de dados ou sinais de controle podem servir de destino a vários canais originados de portas distintas. Em um fluxo de sinais de controle, a criação de canais destinados à mesma porta indica que a atividade de destino precisa aguardar a execução de todas as origens para ser executada, servindo como um ponto de sincronismo do fluxo. A Figura 4.8 demonstra que a atividade `Block`, localizada na parte inferior da imagem, é um ponto de sincronismo entre as atividades `Block` superiores.

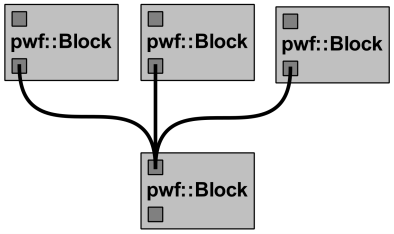
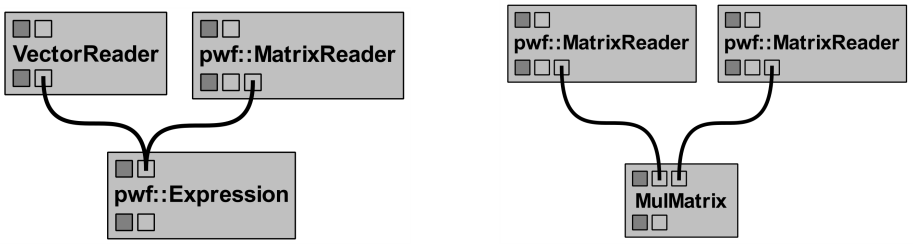


Figura 4.8: Sincronismo em fluxos de controle.

Em um fluxo de dados, a criação de canais destinados à mesma porta indica que a atividade de destino precisa receber dados de todas as atividades de origem para que possa ser executada, conforme representado pela Figura 4.9(a). Outra forma de sincronismo é representado em um fluxo de dados por um conjunto de canais originados de atividades distintas que se conectam em diferentes portas de uma atividade de destino. A Figura 4.9(b) demonstra que a atividade `MulMatrix` é um ponto de sincronismo entre as duas atividades `MatrixReader`.

As atividades que compõem um workflow podem ser definidas estática ou dinamicamente. As atividades estáticas são criadas através da definição de novos tipos de atividades, conforme apresentaremos na Seção 4.4. As atividades dinâmicas são definidas através do preenchimento de uma atividade vazia chamada *bloco*. Os blocos são atividades responsáveis por abrigar e gerenciar o relacionamento entre componentes, criando fluxos de execução próprios. Cada bloco possui os seguintes componentes básicos:



(a) Mesma porta de destino (b) Várias portas de destino

Figura 4.9: Sincronismos em fluxos de dados.

- Um conjunto de portas de entrada de dados, responsáveis por receber os dados a serem processados internamente;
- Um conjunto de portas de saída de dados, responsáveis por transmitir os resultados produzidos;
- Um conjunto de proxies de portas de entrada;
- Um conjunto de proxies de portas de saída;
- Um conjunto de atividades internas;
- Um conjunto de canais.

O *proxy de porta* é um componente conectável que representa uma porta do bloco em seu ambiente interno. Um proxy de porta de entrada é representado por um retângulo verde contendo um rótulo que o identifica e uma porta de saída utilizada para transferir o dado de entrada para o fluxo de execução do bloco. Um proxy de porta de saída é representado por um retângulo vermelho contendo um rótulo que o identifica e uma porta de entrada utilizada para repassar os resultados produzidos à porta de saída representada pelo componente. A Figura 4.10 representa os proxies de porta `Input` e `Output` presentes em um bloco.

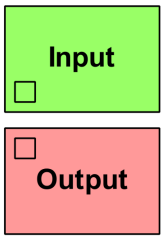


Figura 4.10: Proxies de porta de um bloco.

Um workflow é um bloco global e, por não ser utilizado como atividade em um workflow maior, não admite a declaração de portas de entrada ou saída. A criação de workflows reutilizáveis é feita através da definição de *atividades paramétricas*. Essas atividades possuem os seguintes componentes:

- Um tipo que define o tipo de atividade que ela representa;

- Um bloco utilizado para definir o fluxo de execução da atividade;
- Um conjunto de argumentos de tipos de atividades.

Os *argumentos de tipos de atividades* são conectores cuja função é permitir a criação de moldes de atividades baseados em um tipo base, que são utilizados para definir um fluxo de execução genérico e transformar a atividade em um *template*. Cada argumento de tipo possui um tipo base de atividade e um rótulo que o identifica. A Figura 4.11 representa uma atividade paramétrica `StaticAnalysis`, que possui um argumento de tipo `D` do tipo `DofNumberer`, representado por um quadrado vermelho localizado próximo ao canto superior direito da atividade.

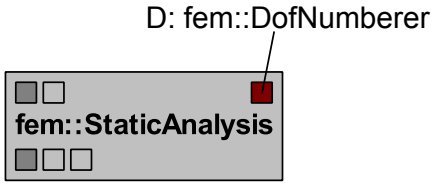


Figura 4.11: Atividade paramétrica `StaticAnalysis`.

Os moldes de atividade são rotulados com o nome do argumento de tipo correspondente e definem um molde de fluxo de execução. O molde só pode ser executado após a *instanciação* dos argumentos de tipos por tipos de atividades compatíveis. Os tipos de atividades são representados por componentes conectáveis que descrevem as instâncias de atividades de seu tipo, apresentando seu tipo base, suas portas de entrada e saída e seus próprios argumentos de tipo. Além das características do tipo, os tipos de atividades possuem uma *porta de tipo* utilizada para conectá-los ao argumento de tipo que deverá ser instanciado. Um tipo de atividade é graficamente representado por um retângulo azul de bordas tracejadas contendo uma porta verde localizada próximo ao canto inferior direito, conforme representado pela Figura 4.12.

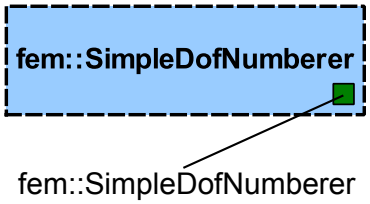


Figura 4.12: Tipo de atividade `SimpleDofNumberer`.

O tipo de atividade `SimpleDofNumberer`, representado pela Figura 4.12, pode ser acoplado a um argumento de tipo para instanciar uma atividade paramétrica através de um *acoplamento de tipo*. Um acoplamento de tipo é representado por um segmento curvilíneo tracejado. A Figura 4.13 apresenta a instanciação da atividade paramétrica `StaticAnalysis` pelo tipo de atividade `SimpleDofNumberer` através de seu acoplamento.

A cada argumento de tipo definido em uma atividade paramétrica, um *proxy de argumento de tipo* é inserido ao bloco do template. O proxy de argumento é um componente conectável que representa o tipo de atividade aceito por seu argumento de tipo correspondente. O proxy de argumento possui uma porta de tipo utilizada para instanciar atividades paramétricas pertencentes ao

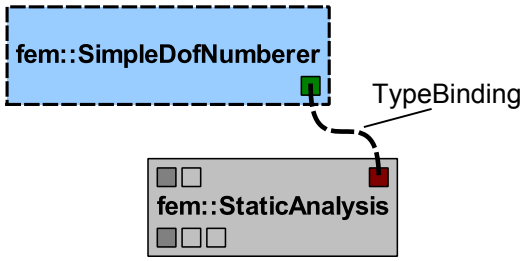


Figura 4.13: Instanciação de atividade paramétrica.

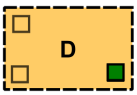


Figura 4.14: Proxy de argumento de tipo.

fluxo de execução da atividade paramétrica e é representado como um tipo de atividade alaranjado, conforme demonstrado pela Figura 4.14.

A utilização de workflows paramétricos permite que o usuário possa criar diferentes instâncias de um mesmo workflow alterando apenas os tipos de atividades acoplados aos argumentos de tipos. A partir dos componentes apresentados é possível representar workflows paramétricos utilizando fluxos de dados e sinais de controle, permitindo que o fluxo de execução seja conduzido em função das transformações de dados realizadas em cada atividade.

4.3 Estrutura de Classes do Modelo

Todo componente do modelo de workflows proposto é um objeto cuja classe deriva da classe abstrata `Component`, que por sua vez deriva da classe `Object`. A classe `Object` representa um objeto genérico contendo um contador de referências. A classe `Component` representa um componente genérico pertencente ao modelo proposto. Cada componente possui um nome e um conjunto de flags utilizadas para indicar diferentes status como, por exemplo, se o componente está ou não habilitado. Os componentes do modelo são divididos entre as categorias conectores, conexões e componentes conectáveis, representados pelas classes `Connector`, `Connection` e `ConnectableComponent` respectivamente.

Um conector representa um ponto de conexão que pode ser associado a outro para definir uma ligação entre dois componentes conectáveis. Os tipos específicos de conectores são representados pelas classes `Port`, `PortType` e `TypeParameter`. Uma conexão representa a ligação entre dois conectores. Os tipos específicos de conexões são representadas pelas classes `Channel` e `TypeBinding`. Os componentes conectáveis são elementos que possuem um ou mais conjuntos de conectores, podendo comunicar-se com outros componentes através de conexões. Os componentes conectáveis específicos são representados pelas classes `Activity`, `PortProxy` e `ActivityType`.

4.3.1 Conectores

Classe `Port`

As portas representam provedores ou receptores de informações de atividades e proxys de porta. As portas possuem os seguintes atributos:

- Um conjunto de flags;
- Um argumento de tipo;
- Um ponteiro para um objeto de dados.

O conjunto de flags indica se a porta é utilizada para entrada ou saída de informações e, caso seja uma porta de entrada, se o recebimento de informações é opcional. O argumento de tipo é uma caracterização do tipo de dados aceito pela porta, representado pelo ponteiro de objeto de dados. As portas cujo argumento de tipo aceito é `Boolean` são consideradas portas de sinais de controle. As portas de dados aceitam quaisquer outros tipos de dados.

Uma porta de saída pode ser conectada a uma porta de entrada através de um canal, definindo um tráfego de informações entre dois componentes conectáveis. Os objetos de dados das portas de saída representam dados produzidos pela execução de suas respectivas atividades. As portas de entrada possuem uma lista objetos de dados, onde cada nó recebe um objeto provindo de um canal. As portas de entrada cujo tipo de dados é um tipo básico e cujo recebimento de dados é opcional podem ter um valor padrão definido pelo usuário. Esse valor é analisado e convertido em um objeto de dados de acordo com o argumento de tipo definido para a porta.

Classe `PortType`

As portas de tipo são conectores que permitem a ligação de um tipo de atividade a um argumento de tipo da classe `TypeArgument`. Assim como as portas da classe `Port`, as portas de tipo possuem um argumento de tipo que corresponde ao tipo de atividade de seu componente conectável. A porta de tipo é conectada a um argumento de tipo através de um acoplamento de tipo da classe `TypeBinding`, instanciando o argumento da atividade paramétrica.

Classe `TypeArgument`

Um argumento de tipo representa um tipo base de atividade aceito por uma atividade paramétrica. Cada argumento de tipo possui um tipo de atividade associado e permite que tipos de atividades sejam utilizados na criação de um molde de fluxo de execução. Cada argumento de tipo precisa ser instanciado por um tipo de atividade para transformar a atividade paramétrica em uma atividade executável.

4.3.2 Conexões

As conexões são ligações entre dois conectores de tipos compatíveis. As conexões são realizadas através de reflexão computacional simples, onde cada conector possui uma definição de tipo de dados aceitável. Dois tipos são compatíveis quando o tipo pertencente à origem faz parte da hierarquia de classes do tipo pertencente ao destino.

Classe `Channel`

Os canais são conexões realizadas entre duas portas de componentes distintos com o objetivo de estabelecer o tráfego de dados ou sinais de controle. As portas possuem uma referência para

uma porta de saída, utilizada como origem da informação a ser trafegada, e uma referência para uma porta de entrada, utilizada como destino da informação.

Classe `TypeBinding`

Os acoplamentos de tipo são conexões responsáveis pela instanciação de uma atividade paramétrica. Os acoplamentos de tipo possuem uma referência a um argumento de tipo, pertencente a uma atividade paramétrica, e uma referência a uma porta de tipo, pertencente a um tipo de atividade que instanciará o template.

4.3.3 Componentes Conectáveis

Classe `Activity`

A classe abstrata `Activity` representa uma atividade genérica. Cada atividade é composta pelos seguintes elementos:

- Uma meta atividade;
- Uma porta de entrada e uma porta de saída de sinais de controle;
- Um conjunto de portas de entrada de dados;
- Um conjunto de portas de saída de dados.

A meta atividade é uma descrição do conjunto de características que definem o tipo da atividade como, por exemplo, os tipos e quantidades de parâmetros que definem as portas de entrada e saida da atividade. As portas de entrada e saída de sinais de controle são utilizadas para definir fluxos de controle entre atividades. Da mesma forma, os conjuntos de portas de entrada e saída de dados são utilizados para definir o fluxo de dados entre as atividades.

A classe `Activity` possui um método virtual puro chamado `run()`, utilizado para acessar os dados de entrada, transformá-los através da execução de um conjunto de instruções e adicionar os dados resultantes nas portas de saída.

Classe `PortProxy`

A classe `PortProxy` representa um repetidor de dados de uma porta externa a uma porta interna à atividade, permitindo o tráfego de dados entre o ambiente interno e externo. Os proxys de portas de entrada permitem que dados do meio externo sejam enviados a atividades internas através de canais. De maneira semelhante, os proxys de portas de saída permitem que dados produzidos por atividades internas sejam enviados para o meio externo. Os objetos dessa classe são importantes no desenvolvimento de atividades paramétricas, pois são equivalentes à utilização dos métodos `getData()` e `setData()` utilizados no método `run()` para acessar os dados das portas.

Classe `ActivityType`

A classe `ActivityType` representa um tipo de atividade utilizado na instanciação de um argumento de tipo de atividade. Cada `ActivityType` possui um objeto da classe `PortType` utilizado para conectá-lo ao argumento de tipo de uma atividade paramétrica.

Classe `TypeParameter`

Os argumentos de tipos são atividades molde utilizadas no desenvolvimento de fluxos de execução em atividades paramétricas. Cada objeto da classe `TypeParameter` representa uma

referência a um argumento de tipo da atividade paramétrica. Ao executar um `TypeParameter`, o argumento referenciado é consultado para que a execução de seu `ActivityType` seja invocada.

Classe `ActivityTemplate`

As atividades paramétricas são atividades cujo fluxo de execução pode ser definido através da utilização de argumentos de tipos. As atividades paramétricas possuem um conjunto de argumentos de tipo, que caracterizam os argumentos de tipos a serem utilizados. Essa atividade permite que o usuário defina um fluxo de execução genérico utilizando os argumentos de tipos e os instancie com diferentes tipos de atividade a cada execução, eliminando a necessidade de substituir componentes em todo o workflow.

A Figura 4.15 representam o diagrama de classes completo dos componentes que definem o modelo proposto. Além das classes discutidas nesta seção, a figura mostra também algumas das classes de atividades básicas, descritas a seguir.

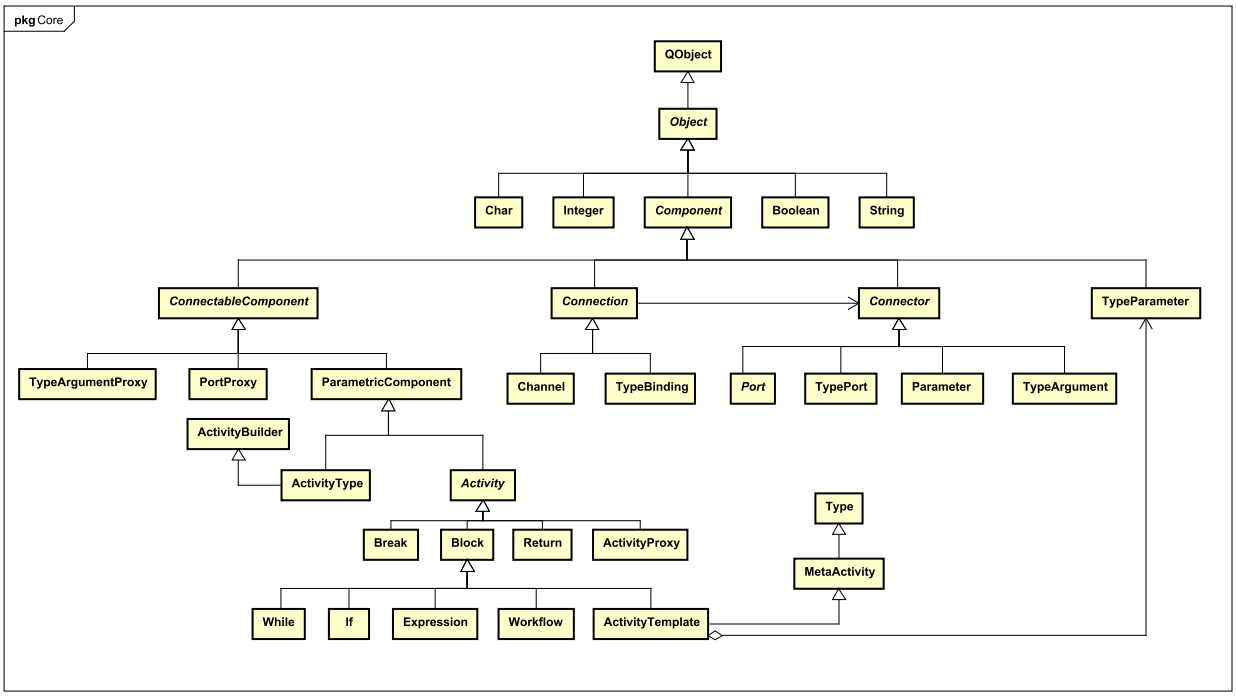


Figura 4.15: Diagrama de classes do modelo proposto.

4.4 Atividades Primitivas

Desenvolvemos neste projeto um conjunto de atividades primitivas utilizado para a definição de fluxos de controle e atividades paramétricas. As atividades desenvolvidas são categorizadas inicialmente nos seguintes grupos:

- Basic Math: conjunto de atividades destinadas à representação de expressões matemáticas;
- Basic Cuda: atividades para configuração de estruturas de dados destinadas à execução de workflows em GPUs;

- FEM: atividades para o desenvolvimento de aplicações baseadas no MEF, contendo atividades passíveis de execução em CPU e GPU.

Apresentamos ainda um conjunto de atividades básicas destinadas à representação das principais estruturas presentes em linguagens de programação, possibilitando a criação de fluxos de controle. Essas atividades possuem seus respectivos métodos `run()` implementados de maneiras particulares e serão apresentadas a seguir.

4.4.1 Atividades Básicas

Este conjunto de atividades provê as estruturas básicas presentes nas principais linguagens de programação.

Classe `Expression`

A classe `Expression` representa um bloco especial que encapsula uma atividade da classe `Code`, que define um conjunto de sentenças a serem executadas em função de objetos de dados disponíveis em suas portas de entrada. Uma expressão possui os seguintes elementos:

- Um conjunto de portas de entrada de dados a ser definida pelo usuário;
- Uma atividade `Code` para descrição do conjunto de sentenças;
- Uma porta de saída de dados `result`.

As portas de entrada são criadas pelo usuário para permitir que objetos de dados de diferentes tipos sejam utilizados na execução da atividade `Code`. Para definir as sentenças o usuário precisa descrevê-las em C++ e ter conhecimento prévio dos métodos utilizados para obter os dados das portas de entrada, inclusão dos resultados na porta de saída e métodos pertencentes às classes de tipos de dados para que possa manipulá-los. Após da definição da atividade `Code`, o usuário pode definir portas de saída de dados para disponibilizar os resultados produzidos. O corpo do método `run()` é definido a partir da compilação das sentenças utilizando o compilador nativo da linguagem C++, resultando em uma DLL que será invocada pelo método.

Classe `Block`

Os blocos são atividades que definem o escopo responsável por gerenciar o relacionamento entre componentes. Conforme representado pelo diagrama da Figura 4.16, cada bloco é composto pelos seguintes elementos:

- Um conjunto de portas de entrada, utilizadas para receberem os dados a serem manipulados;
- Um conjunto de portas de saída, utilizadas para disponibilizar o resultado da execução do bloco;
- Um conjunto de `PortProxy`, utilizados para transmitir dados de entrada para o fluxo interno e enviar os resultados para as portas de saída;
- Um conjunto de `Activity`, que definem o fluxo de execução interno do bloco;
- Um conjunto de `Channel`, que conectam as atividades internas;

- Um conjunto de `ActivityType`, utilizados para instanciar os argumentos de tipos de atividades paramétricas;
- Um conjunto de `TypeBinding`, que conectam cada `ActivityType` a seu respectivo argumento de tipo.

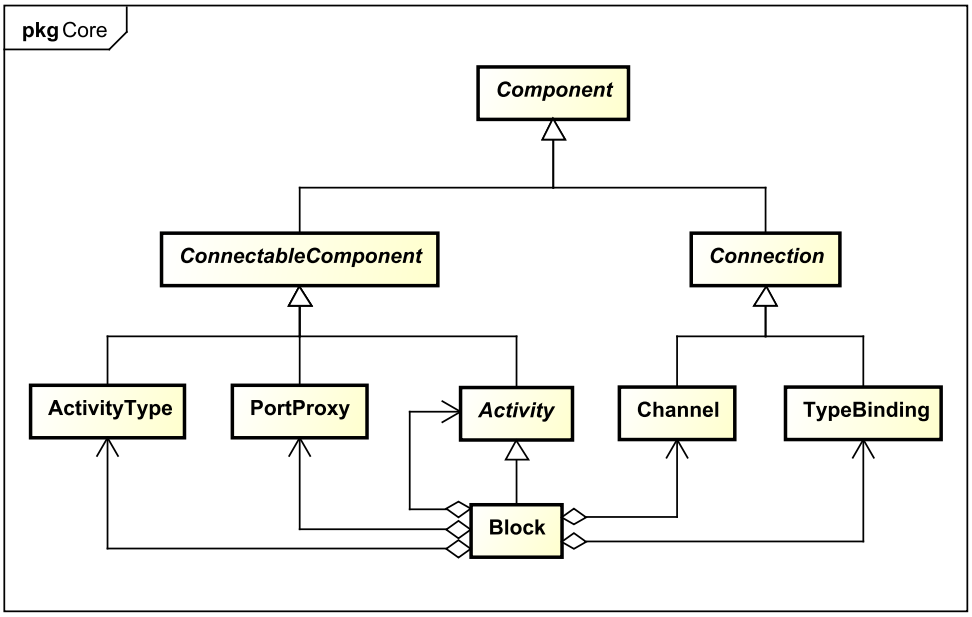


Figura 4.16: Composição de um bloco.

O bloco é executado representando o workflow como um grafo orientado acíclico cujos nós são representados pelas atividades e as arestas são representadas pelos canais. As atividades que não servem de origem para os canais são executadas, transmitindo seus resultados para os respectivos nós de destino. Os nós executados são removidos juntamente com as arestas conectadas. O processo se repete até que o grafo fique vazio. O processo de execução de um bloco é descrito detalhadamente na Seção 4.5.

Classe `If`

A classe `If` é uma atividade que representa a estrutura condicional *se*. A atividade `If` é um bloco não editável que possui os seguintes elementos não deletáveis:

- Uma atividade do tipo `Expression`, chamada `condition`;
- Duas atividades do tipo `Block`, chamadas `trueBlock` e `elseBlock`.

A execução dessa atividade é realizada através da avaliação da expressão `condition`. Se o resultado da expressão for diferente de zero ou nulo, a condição é considerada verdadeira e o bloco `trueBlock` é executado, caso contrário o bloco `elseBlock` é executado. A classe `If` permite que o usuário defina portas de entrada e saída de dados, tornando possível o tráfego de dados através da conexão entre suas atividades internas e as instâncias da classe `ProxyPort` utilizadas para representar as portas internamente. A atividade `If` permite ainda a conexão da `condition` a uma porta de dados do `trueBlock`, proporcionando a utilização do valor da expressão para novas comparações.

Classe `While`

A classe `While` é uma atividade que representa a estrutura de repetição *enquanto*. A atividade `While` é um bloco não editável contendo os seguintes componentes não deletáveis:

- Uma atividade do tipo `Expression`, chamada `condition`;
- Uma tividade do tipo `Block`, chamada `statement`.

O método `run()` dessa classe executa a expressão `condition` e, caso o resultado da expressão seja diferente de zero ou nulo, o bloco `statement` é executado. Esse processo se repete enquanto o resultado da expressão for verdadeiro. Assim como a classe `If`, a classe `While` permite a criação de portas de entrada e saída de dados, permitindo a comunicação do bloco `statement` com o meio externo, além de permitir a transmissão o resultado da expressão à sentença.

Classe `For`

A classe `For` representa a estrutura de repetição *para*. A atividade é um bloco não editável contendo as seguintes atividades:

- Uma atividade do tipo `Interval`, chamada `interval`;
- Uma atividade do tipo `Block`, chamada `statement`.

O método `run()` se inicia pela execução da atividade `interval`, resultando em um conjunto de valores. Em seguida, o bloco `statement` é executado uma vez para cada valor contido na lista. Se necessário, o usuário pode definir uma porta de entrada de dados com o mesmo tipo definido para os elementos do intervalo e, para cada iteração, passar o elemento correspondente da lista como dado de entrada para `statement`.

Classe `OpenFileDialog`

Atividade responsável por permitir que o usuário escolha um arquivo a ser utilizado por outras atividades através de uma caixa de diálogo. Sua execução é definida pela abertura de uma caixa de diálogo para a escolha do arquivo e, após escolhido, a atividade retorna um objeto de dados do tipo `String` contendo o caminho absoluto do arquivo.

Classe `MatrixReader`

Atividade utilizada para realizar a leitura de uma matriz a partir de um arquivo. A atividade contem uma porta de entrada do tipo `String`, responsável por receber o caminho referente ao arquivo. Seu método `run()` realiza a leitura do arquivo cujo nome foi passado como entrada e tenta converter seu conteúdo a um objeto do tipo `Matrix`. Ao final da execução a matriz produzida é atribuída à porta de saída `matrix`.

Classe `MatrixWriter`

Atividade utilizada para realizar a escrita de uma matriz em arquivo. A atividade contem uma porta de entrada do tipo `String`, responsável por receber o caminho referente ao arquivo e uma porta de entrada do tipo `Matrix`, responsável por receber a matriz a ser escrita. Seu método `run()` realiza a escrita de todos os elementos da matriz no arquivo definido pelo caminho.

Classe `StandardOutput`

A atividade `StandardOutput` representa um escritor de objetos na saída padrão do sistema. Sua execução invoca o método virtual puro `toString()` do objeto definido como entrada da atividade e apresenta a escrita no terminal do sistema.

4.4.2 Estendendo a API

A partir do conjunto de classes disponível é possível criar novos tipos de dados e atividades a serem utilizadas no desenvolvimento de workflows. Cada informação que trafega entre as atividades é representada por um objeto de dados. Toda classe de objeto de dados deriva direta ou indiretamente de `Object`. A criação de um novo tipo de objeto de dados exige que o usuário utilize o macro `PWF_TYPE`, que define e registra um novo tipo de dados em um mapa de tipos disponível no sistema. Como exemplo, seja a classe `Domain`, a qual representa um domínio a ser analisado pelo MEF (a classe é descrita no Capítulo 5). A fim de tornar `Domain` uma classe de objeto de dados do sistema, escrevemos em um arquivo de cabeçalhos:

```
class Domain: public Object
{
  PWF_TYPE(Domain, Object)
  ...
};
```

Na trecho de código acima, derivamos `Domain` da classe de `Object` e utilizamos `PWF_TYPE` para declarar todos os membros necessários ao registro do novo tipo de dados, bem como os métodos de instanciação de objetos da classe. Tais membros implementam um esquema simples de reflexão computacional, mas que é suficiente para as operações envolvendo tipos realizadas pelo sistema, tais como a possibilidade de declarar portas de entrada e saida do tipo `Domain` ou de qualquer tipo derivado direta ou indiretamente de `Domain`. Além disso, a qualquer porta de entrada do tipo `Object` pode ser conectada um porta de saída do tipo `Domain`, pois a última é derivada da primeira. Nesse esquema de reflexão, um tipo é representado pela classe `Type`, cujas principais funcionalidades são verificar se um tipo herda de outro e criar instâncias do tipo representado.

A criação de novas atividades ocorre de maneira semelhante. As classes de atividades definem um novo tipo de atividade e devem derivar direta ou indiretamente da classe `Activity`. O macro `PWF_ACTIVITY` é usado para declaração dos membros necessários para o registro do novo tipo no mapa de tipos do sistema e para instanciação de objetos da classe. Além disso, os tipos de atividades possuem metadados que caracterizam as portas de entrada e saída e os argumentos de tipo. O macro `BEGIN_ACTIVITY_TYPE` tem por objetivo definir a descrição do tipo de atividade declarado, podendo incluir portas de entrada, portas de saída e argumentos de tipo através da utilização dos macros `INPUT_PARAMETER`, `OUTPUT_PARAMETER` e `TYPE_PARAMETER`, respectivamente, os quais definem o nome e o tipo aceito pelo parâmetro criado. Como exemplo, seja a classe `StaticAnalysis`, que representa um analisador estático do MEF (a classe é descrita no Capítulo 5). A fim de torná-la uma classe de objetos que são atividades do sistema, escrevemos em um arquivo de cabeçalhos:

```
class StaticAnalysis: public Activity
{
  PWF_ACTIVITY(StaticAnalysis, Activity)
  ...

private:
  RunCode run();
  ...
};
```

Com o macro `PWF_ACTIVITY`, o tipo de atividade `StaticAnalysis` é inserido no mapa de tipos do sistema como um subtipo da atividade `Activity`. O método privado `run()` deve ser sobreescrito por toda classe de atividade concreta e é chamado pelo motor de execução, conforme descrito

na Seção 4.5.3. Para declarar as portas de entrada e saída e os argumentos de tipo do novo tipo de atividade, escrevemos um arquivo de código fonte:

```
#include "StaticAnalysis.h"

BEGIN_ACTIVITY_TYPE(StaticAnalysis)
  INPUT_PARAMETER("domain", Domain)
  TYPE_PARAMETER("D",DofNumberer)
  OUTPUT_PARAMETER("this", DofNumberer)
  OUTPUT_PARAMETER("domain", Domain)
END_ACTIVITY_TYPE
...
```

O macro `BEGIN_ACTIVITY_TYPE` inicia a descrição do tipo de atividade `StaticAnalysis`. O macro `INPUT_PARAMETER` define uma porta entrada de dados cujo nome é `domain` e o tipo de dado aceito é um objeto da classe `Domain`. A seguir, o macro `TYPE_PARAMETER` define um argumento de tipo chamado `D`, que aceita atividades do tipo `DofNumberer` para sua instanciação. Em seguida, o macro `OUTPUT_PARAMETER` é utilizado para definir duas portas de saída de dados cujos nomes são `this` e `domain` e cujos tipos de dados de retorno são `DofNumberer` e `Domain` respectivamente. A declaração da atividade `StaticAnalysis` resulta em uma atividade paramétrica, representada pela Figura 4.17.

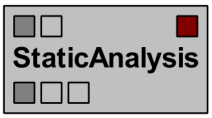


Figura 4.17: Representação da atividade `StaticAnalysis`.

4.5 Interface Gráfica

A interface desenvolvida neste projeto possui um conjunto de classes responsáveis pela representação gráfica do modelo proposto na Seção 4.2. Conforme apresentado na Figura 4.18, esta interface é composta por:

- Um menu principal, localizado na região superior, contendo as principais funcionalidades do sistema;
- Um `WorkflowBrowser`, localizado na região central esquerda;
- Uma `ActivityBox`, localizada na região central direita;
- Uma área de trabalho, localizada na região central;
- Um `OutputView`, localizado na região inferior.

O menu principal agrupa funcionalidades para criar, salvar e abrir workflows, criar, salvar e abrir atividades paramétricas, executar e monitorar workflows. As funcionalidades serão descritas detalhadamente no decorrer desta seção.

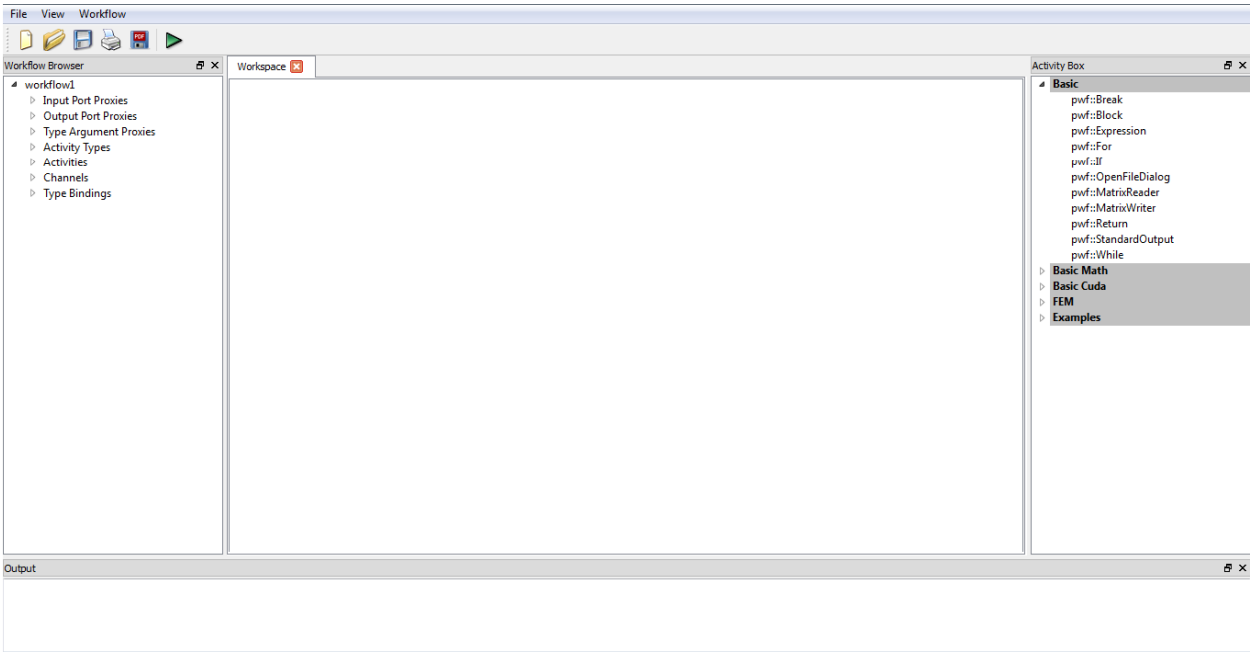


Figura 4.18: Interface Gráfica.

O `WorkflowBrowser` é uma área de texto responsável por exibir a estrutura xml correspondente ao workflow em desenvolvimento, que é atualizada a cada modificação realizada. Ao salvar o workflow, um arquivo contendo a estrutura xml é criado, possibilitando que o usuário utilize-o posteriormente.

A `ActivityBox` é uma estrutura de árvore utilizada para listar as atividades disponíveis no sistema. A árvore é composta pelo conjunto de atividades primitivas apresentadas na Seção 4.4, um conjunto de atividades utilizadas para representar o MEF e um conjunto de atividades paramétricas desenvolvidas pelo usuário.

A área de trabalho é a região principal, utilizada para criar abas que representam as áreas de desenvolvimento do workflow principal e das atividades paramétricas. Aba principal, denominada *Workspace*, é utilizada para a representação do workflow principal e é a única aba que pode ser executada pelo sistema. As abas secundárias são criadas à direita da aba principal e representam atividades cujo fluxo de execução interno pode ser desenvolvido pelo usuário como, por exemplo, as atividades das classes `Block` e `ActivityTemplate`.

Os componentes de desenvolvimento de workflows representados na interface são representados por um conjunto de classes de representação gráfica, onde cada componente apresentado na Seção 4.3 possui uma classe de representação gráfica correspondente. Essa distinção é feita para que o modelo proposto não dependa de uma representação gráfica para ser utilizado, permitindo que o usuário estenda o conjunto de classes do modelo sem a necessidade de definir ou utilizar elementos da interface.

A classe `Item`, correspondente da classe `Component`, é uma classe abstrata utilizada como base para a representação de itens gráficos. A classe `Item` possui retângulo que representa a região da tela onde o objeto será desenhado e dois objetos da classe `Style`, que representam as características gráficas a serem adotadas quando o objeto está em seu estado normal e selecionado. De maneira semelhante, as representações gráficas são divididas entre as categorias de itens conectáveis, conectores e conexões, conforme representado pela Figura 4.19.

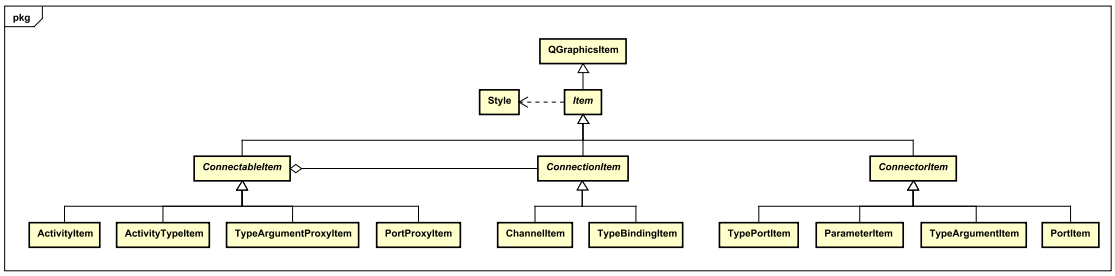
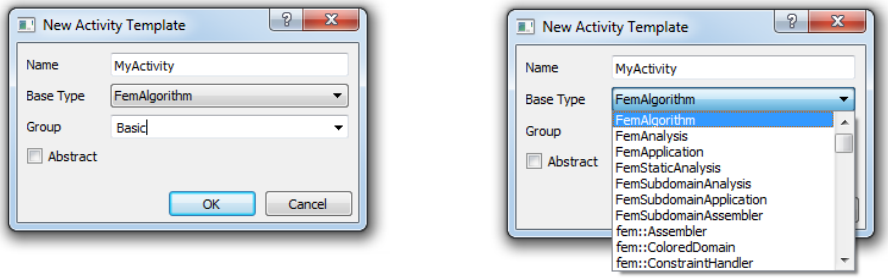


Figura 4.19: Diagrama de classes dos itens gráficos.

4.5.1 Desenvolvimento de Atividades Paramétricas

A interface permite a criação de atividades paramétricas através de `New-Activity Template`, localizado no menu `File`. Ao selecionar essa opção, uma caixa de diálogo se abre para que o usuário defina o nome e o tipo base da nova atividade, escolhendo-o de acordo com os tipos de atividades disponíveis, conforme demonstrado na Figura 4.5.1. Essa definição permite que o usuário defina uma atividade que poderá ser utilizada para instanciar outras atividades paramétricas cujos argumentos aceitem seu tipo base.



(a) Caixa de diálogo (b) Definição do tipo base

Figura 4.20: Criação de atividade paramétrica.

Em seguida o sistema cria uma nova aba destinada à caracterização de uma nova atividade paramétrica, onde o usuário pode pressionar o cursor com o botão direito do mouse para definir portas de entrada ou saída e argumentos de tipo, conforme apresentado na Figura 4.21. A nova atividade também é incluída na subárvore do `ActivityBox` correspondente ao grupo escolhido, tornando-se permanente após a persistência da atividade. A inclusão da atividade no `ActivityBox` permite que o usuário a utilize no desenvolvimento de novos workflows e atividades, possibilitando inclusive a utilização de instâncias em sua própria definição, criando atividades paramétricas recursivas.

Ao selecionar a opção de inclusão de portas, uma janela é aberta para que o usuário defina o nome, a função e o tipo de dados aceito pela nova porta de acordo com os tipos de dados disponíveis no sistema. Após a configuração da porta, um `PortProxy` é inserido à área de desenvolvimento para permitir que o usuário trafegue informações com o meio externo, conforme ilustrado na Figura 4.22.

Ao selecionar a opção de inclusão de argumento de tipo, uma janela é aberta para que o usuário defina o nome e o tipo de atividade que será aceita pelo novo argumento. Definidas as informações,

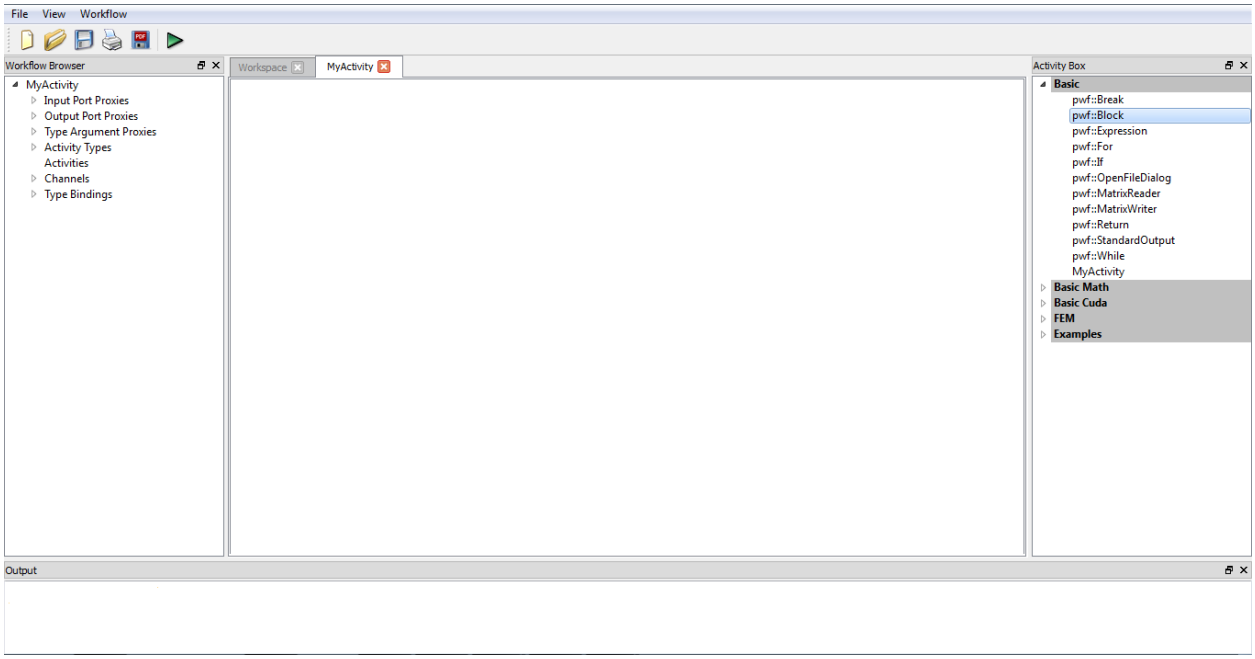


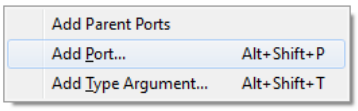
Figura 4.21: Caracterização de atividades paramétricas.

um `TypeArgumentProxy` é adicionado à atividade paramétrica, permitindo que o usuário instancie atividades paramétricas internas com o mesmo tipo de atividade aceito pelo argumento. Ao clicar com o botão direito do mouse sobre um `TypeArgumentProxy`, o usuário pode criar atividades da classe `ActivityProxy`, que representam moldes que referenciam o argumento para serem instanciadas posteriormente. A Figura 4.23 representa a criação de um argumento de tipo `D` utilizado para instanciar a atividade paramétrica `StaticAnalysis` e um proxy de atividade do tipo `D` que referencia o argumento.

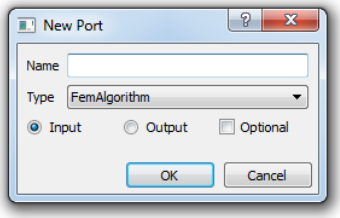
A inclusão de atividades no fluxo de execução é realizada de acordo com as atividades disponíveis no `ActivityBox`. Para incluir uma nova atividade, o usuário pressiona o botão esquerdo do mouse com o cursor posicionado sobre o nome da atividade a ser criada, arrastando-o até o local desejado. Ao soltar o botão do mouse, o sistema cria uma nova instância da atividade escolhida e adiciona a representação xml correspondente à estrutura representada pelo `WorkflowBrowser`. A Figura 4.24 representa a inclusão de uma instância de atividade do tipo `Block` no fluxo de execução da atividade paramétrica do tipo `MyActivity`.

Um bloco é uma atividade cujo fluxo de execução é definido em uma aba auxiliar através do relacionamento entre componentes, criada à direita da última aba disponível através de um clique duplo sobre a instância. Assim como as abas de desenvolvimento de atividades paramétricas, as abas de desenvolvimento de blocos não são executáveis e permitem a definição de parâmetros de entrada e saída.

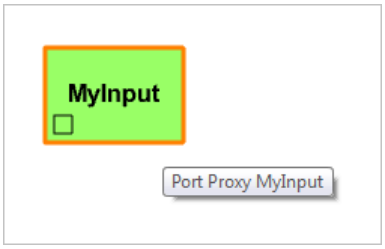
Após a inclusão de atividades em uma aba é possível definir o fluxo de execução através da criação de canais. A criação de um canal é feita pelo usuário pressionando o botão esquerdo do mouse com o cursor posicionado sobre a porta de saída da atividade de origem, arrastando-o e posicionando-o sobre a porta de entrada da atividade de destino. Após a liberação do cursor, o sistema verifica se o tipo de dado produzido pela porta de origem é compatível pelo tipo de dado aceito pelo destino e, caso sejam compatíveis, um canal é estabelecido, conforme representado pela Figura 4.25.



(a) Menu de adição de portas

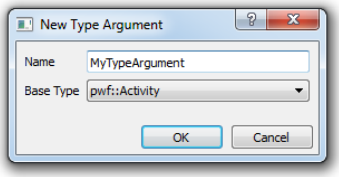


(b) Caixa de diálogo

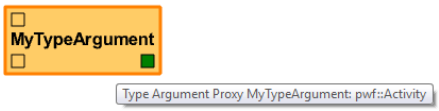


(c) Proxy de porta criado

Figura 4.22: Criação de portas.



(a) Caixa de diálogo



(b) Proxy de argumento de tipo criado

Figura 4.23: Criação de argumentos de tipos.

A inclusão de atividades paramétricas exige a instanciação de seus respectivos argumentos de tipo para que seja possível executar o workflow. Para instanciá-las, o usuário precisa criar representações de tipos de atividades e conectá-las aos argumentos de tipo através da criação de conexões da classe `TypeBinding`. A criação de objetos da classe `ActivityType` é feita pressionando-se o botão direito do mouse com o cursor posicionado sobre um dos tipos de atividades presentes no `ActivityBox`, arrastando-o e soltando-o sobre o local desejado. Após a liberação do cursor, o sistema cria uma nova instância do tipo de atividade escolhido. Após a criação de um `ActivityType`, o usuário pressiona o botão esquerdo sobre o `TypePort` do `ActivityType` e o arrasta até o argumento de tipo que se deseja instanciar. Após a liberação do cursor, o sistema verifica se o tipo representado pelo `ActivityType` é compatível ao tipo de atividade aceito pelo argumento de tipo e, caso sejam compatíveis, um acoplamento de tipo da classe `TypeBinding`, conforme representado pela Figura 4.26.

Ao concluir o desenvolvimento da atividade paramétrica, o usuário pode salvá-la e utilizá-la no desenvolvimento de novas atividades e workflows. As atividades paramétricas são salvas em um diretório do sistema chamado `templates` para que o componente possa ser disponibilizado na árvore de atividades e possa ser requisitado durante a execução de workflows que a utilizem.

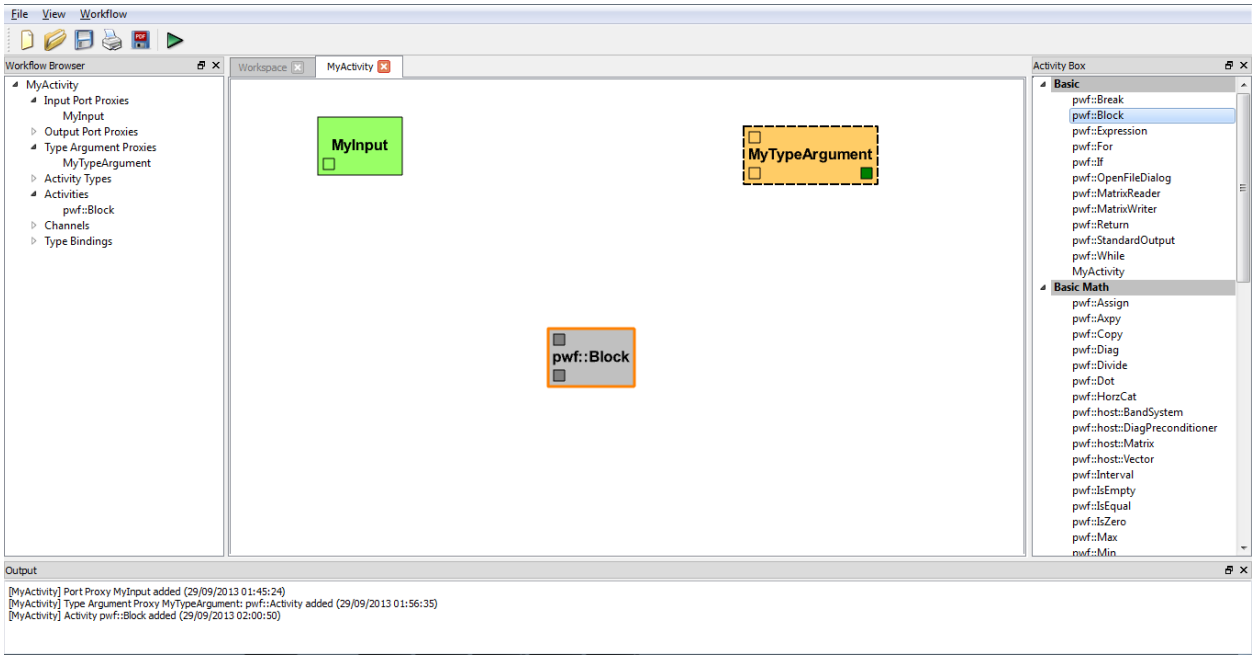
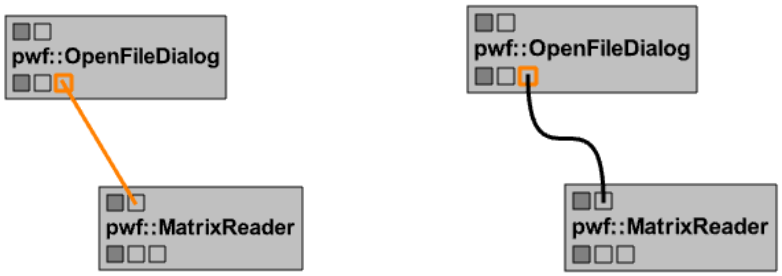


Figura 4.24: Criação da atividade `Block`.



(a) Seleção das portas (b) Canal criado

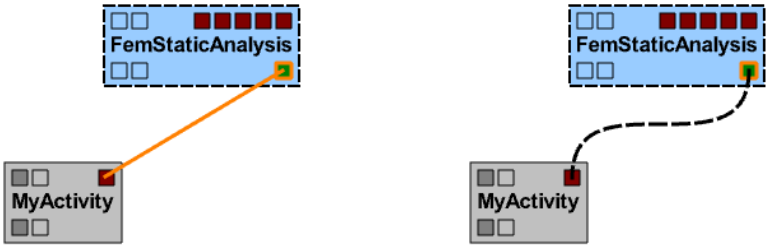
Figura 4.25: Criação de fluxo de execução através de canais.

4.5.2 Desenvolvimento de Workflows

Conforme apresentado na Seção 4.5, o desenvolvimento de workflows é realizado em abas de desenvolvimento posicionadas na área de trabalho, utilizando como base as atividades disponíveis no `ActivityBox`. O desenvolvimento pode ser descrito através dos seguintes procedimentos:

- Criação de atividades paramétricas;
- Inserção de instâncias de atividades ao workflow;
- Conexão de atividades através da ligação entre portas;
- Instanciação dos argumentos de tipos das atividades paramétricas.

Após a criação das atividades necessárias, conforme descrito neste capítulo, o usuário desenvolve o workflow da mesma forma como se desenvolve um bloco, porém seu desenvolvimento



(a) Seleção das portas (b) Acoplamento de tipo criado

Figura 4.26: Instanciação de atividade paramétrica.

resulta em um arquivo xml que pode ser executado pelo sistema utilizando o motor de execução descrito na Seção 4.5.3.

4.5.3 Execução de Workflows

O processo de execução de workflows é acionado a partir do submenu `Execute`. Este processo utiliza como base o workflow presente na aba principal e converte sua representação gráfica em representação executável.

Cada bloco, assim como o workflow, possui um motor de execução da classe `Engine`, que é responsável por executar o fluxo de execução contido no bloco. Um motor possui um *pool de threads* responsável por administrar a execução das atividades, assim como uma lista a ser preenchida com atividades passíveis de execução, conforme representado pela Figura 4.27.

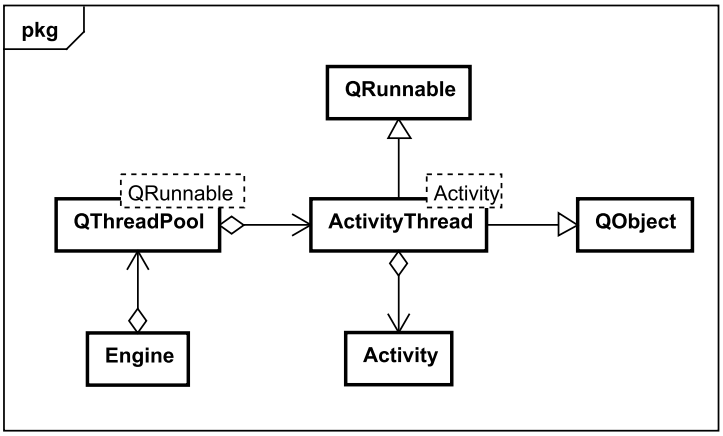
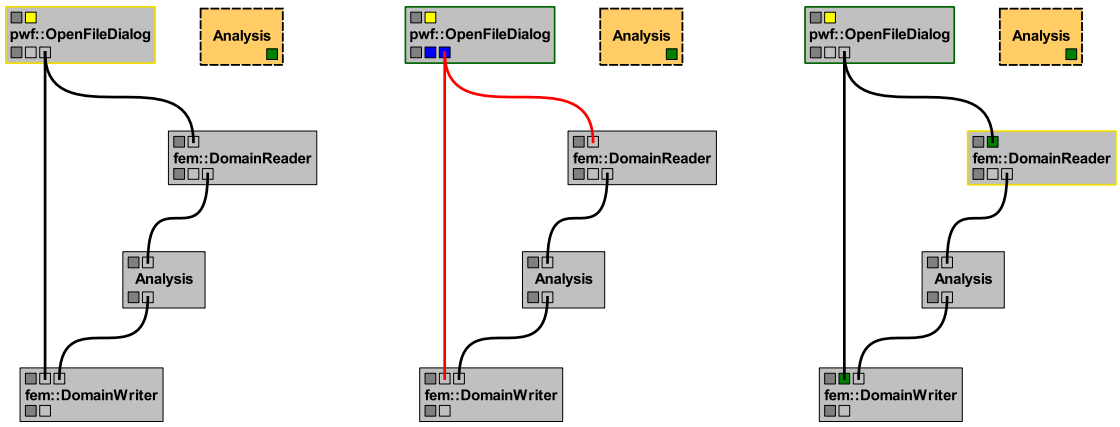


Figura 4.27: Diagrama de classes do motor de execução.

A execução do workflow se inicia através da transferência de seu fluxo de execução para o motor. O motor verifica inicialmente quais atividades não possuem dependência as insere em uma fila de atividades passíveis de execução RA. Em seguida, o motor inicia a criação de threads em seu pool de threads PT e transfere uma atividade para cada thread enquanto houver recurso disponível. Uma atividade é passível de execução se satisfizer as seguintes condições:

- Suas portas de entrada não possuem dependência de dados ou sinais de controle ou;
- Sua porta entrada de sinais de controle possuem um ou mais canais conectados e todos os sinais de controle foram recebidos, indicando que todas as atividades antecessoras concluíram suas respectivas execuções, e;
- Suas portas de entrada de dados receberam informações de outras atividades.

Ao ser alocada em uma thread, a atividade muda seu status de execução para a condição de iniciada, registrando o início de sua execução no log e alterando a cor de sua representação gráfica para amarelo. Em seguida a atividade é executada de acordo com a definição de seu método `run()`. Ao final da execução, a thread registra a conclusão da execução da atividade no log, muda o status de execução para a condição sucesso ou falha e altera a cor de sua representação gráfica, tornando-se verde em caso de sucesso ou vermelha em caso de falha. Ao final da execução, a atividade atribui o sinal de controle apropriado à porta de saída de sinais de controle e, em caso de sucesso, transfere os resultados produzidos às portas de saída de dados. A Figura 4.28 representa as três primeiras etapas de execução do algoritmo do MEF.



(a) Abertura do arquivo (b) Transferência dos dados (c) Leitura do domínio

Figura 4.28: Execução do MEF.

Após a conclusão do método `run()`, a atividade acorda o motor de seu bloco comunicando-o a conclusão de sua execução. O motor faz com que os canais ligados às portas de saída da atividade transfiram os dados produzidos às atividades de destino. O motor retira do grafo de execução a atividade e os canais já utilizados, procurando por novas atividades passíveis de execução para serem inseridas na RA. O ciclo se repete até que o grafo de execução e o pool de threads do motor fiquem vazios.

4.6 Comentários Finais

O modelo proposto, conforme demonstrado neste capítulo, possibilita o desenvolvimento de sistemas através de uma representação gráfica de uma linguagem de programação, capaz de criar, relacionar e reutilizar componentes baseados na biblioteca disponível. A partir da extensão do modelo foi possível criar componentes voltados para o desenvolvimento de aplicações baseadas no FEM.

O conjunto de atividades primitivas podem ser estendidas para a criação de componentes próprios. Os componentes existentes representam as estruturas básicas utilizadas em programação como, por exemplo, estruturas condicionais, estruturas de repetição, blocos e sentenças. Esses componentes tornam possível a criação de workflows com fluxos de controle. Através da extensão da biblioteca básica, é possível criar novos tipos de dados e atividades, capazes de instanciar atividades paramétricas e realizar trocas de diferentes tipos dados entre si. Dessa forma, é possível definir fluxos híbridos entre fluxos de dados e fluxos de controle, permitindo que o tráfego de dados seja definido em função de condições específicas decorrentes do processo de execução. Através dessa abordagem é possível desenvolver programas e gerenciar sua execução de forma mais fácil e intuitiva.

Alguns projetos foram desenvolvidos para mesclar fluxos de dados e de controle a partir das ferramentas apresentadas [32, 6]. Esses projetos tem por objetivo produzir dados a partir de estruturas condicionais e estruturas de repetição, onde o processamento interno dos componentes produz diferentes tipos de dados de acordo com as condições presentes nas estruturas condicionais ou manipula e produz diferentes conjuntos de dados a cada iteração das estruturas de repetição. Essa abordagem auxilia, por exemplo, a repetição automática do mesmo experimento para várias entradas de dados, porém o comportamento dessas estruturas de controle não representam o comportamento de estruturas de linguagens de programação.

O conceito de workflow paramétrico utilizado neste projeto apresenta uma abordagem distinta, onde os workflows apresentam argumentos de tipos capazes de criar referências cujos tipos de atividade aceitos podem ser definidos pelo usuário. As atividades de referência, por não apresentarem um método de execução próprio, impedem que o workflow seja executado sem sua instanciação prévia. Após salvar o workflow paramétrico, é possível utilizá-lo como atividade em outros workflows, permitindo que sua instanciação e execução sejam realizadas. A abordagem apresentada permite que o usuário controle as instanciações a serem realizadas, possibilitando a comparação entre diferentes conjuntos de instanciações e a escolha da melhor combinação entre os componentes disponíveis, resultando assim em uma aplicação mais eficiente.

O tráfego de dados também é restrito a atividades cujas portas apresentem tipos de dados compatíveis, porém o tráfego de dados é realizado de maneira direta, sem a necessidade de sucessivas leituras e escritas em arquivos. Os tipos de dados utilizados são definidos a partir de metadados e permitem que o usuário modele tipos de dados mais complexos a serem trafegados entre as atividades.

CAPÍTULO 5

Exemplos

5.1 Introdução

Neste capítulo mostramos exemplos de algoritmos definidos a partir do sistema de workflows paramétricos proposto. Na Seção 5.2 apresentamos o desenvolvimento de um solucionador de sistemas lineares pelo método dos gradientes conjugados usando um workflow paramétrico. Apresentamos também a instanciação do workflow para execução em CPU e GPU. A Seção 5.3 demonstra o desenvolvimento de uma aplicação de análise estática via MEF utilizando como solucionador de sistemas lineares a atividade `CGSolver`. A Seção 5.4 apresenta uma solução alternativa utilizando um analisador por decomposição de domínio, cuja execução é realizada através de GPUs. Ao final do capítulo apresentamos nossas considerações diante dos exemplos abordados.

5.2 Gradientes Conjugados

Tomemos, como exemplo inicial, o desenvolvimento de um workflow paramétrico correspondente ao algoritmo de resolução de sistemas lineares pelo método dos gradientes conjugados. Uma implementação em C++ do algoritmo é listada a seguir. A função `CG_Solver` recebe como argumentos de entrada a matriz `A`, o vetor de termos independentes `b` e o pré-condicionador `M` a ser usado no método. O vetor solução `x` é o argumento de saída. A tolerância `tol` e o número máximo de iterações `maxIter` são argumentos de entrada e saída. A função funciona para quaisquer matrizes, vetores e pré-condicionador dos tipos `Matrix`, `Vector` e `Preconditioner`, respectivamente, que satisfizerem as operações codificadas no corpo de `CG_Solver`. O exemplo a seguir é mais acadêmico, embora ilustrativo das capacidades do sistema de workflows paramétricos proposto. A implementação direta em C++ é, obviamente, mais eficiente e faz parte da API de elementos finitos desenvolvida em [34], contando com uma atividade que faz uso direto de tal implementação. Não obstante, o exemplo pode ser útil até mesmo para o ensino em disciplinas de cálculo numérico, por exemplo.

O algoritmo pode ser representado por um workflow paramétrico contendo os seguintes componentes, conforme ilustrado na Figura 5.1:

- Quatro portas de entrada de dados correspondentes aos argumentos de entrada `A`, `b`, `tol` e `maxIter`;

```
01  template <class Matrix, class Vector,
02            class Preconditioner, typename real>
03  bool
04  CG_Solver(const Matrix& A,
05     Vector& x,
06     const Vector& b,
07     const Preconditioner& M,
08     int& maxIter,
09     real& tol)
10 {
11     int n = b.getSize();
12
13     x = Vector::Zero(n);
14
15     real res = 1, rho_1;
16     Vector p(n), z(n), q(n), r(n);
17
18     r.copy(b);
19     for (int i = 1; i <= maxIter; i++)
20     {
21         M.solve(z, r);
22         real rho = r.dot(z);
23
24         if (i > 1)
25             z.add(rho / rho_1, p);
26         p.copy(z);
27         A.mul(q, p);
28
29         real alpha = rho / p.dot(q);
30
31         x.add(+alpha, p);
32         r.add(-alpha, q);
33         if ((res = z.norm() / x.norm()) <= tol)
34         {
35             maxIter = i;
36             tol = res;
37             return true;
38         }
39         rho_1 = rho;
40     }
41     tol = res;
42     return false;
43 }
```

Algoritmo 1: CG_Solver.

- Três portas de saída de dados correspondentes aos argumentos de saída `x`, `tol` e `maxIter`;
- Três argumentos de tipo correspondentes aos parâmetros de tipos `Matrix`, `Vector` e `Preconditioner`.

As portas de entrada de dados `A`, `b`, `tol` e `maxIter` recebem, respectivamente, uma matriz, um vetor, a tolerância e o número máximo de iterações a serem executados para de resolução do sistema. As portas de saída de dados `x`, `tol` e `maxIter` produzem, respectivamente, o vetor solução do sistema linear, a tolerância atingida e a quantidade de iterações efetivamente feitas para a solução do sistema.

O argumento de tipo `M` representa uma *meta-matriz* genérica, isto é, uma atividade cuja execução cria uma instância de um objeto da classe `Matrix` ou dela derivada direta ou indiretamente (em outras palavras, uma atividade do tipo meta-`X` é um construtor de objetos do tipo `X`). O argumento de tipo `M` tem como tipo base a atividade abstrata `MetaMatrix`, que possui como portas de entrada as dimensões da matriz a ser criada ou então uma matriz já existente. Nesse caso, a matriz a ser criada será uma cópia rasa ou profunda da matriz a ser copiada, dependendo se a primeira reside no mesmo dispositivo (CPU ou GPU) da segunda. Neste exemplo, a (meta-)atividade `M`, instância do argumento de tipo `M`, é utilizada para copiar a matriz que chega na porta `A` para a CPU, caso ela esteja na GPU e o algoritmo deva executar em CPU, ou então para a GPU, caso ela esteja em CPU e o algoritmo deva executar em GPU. De maneira semelhante, o argumento de tipo `V` é um meta-vetor (de tipo base `MetaVector` e sua instanciação é utilizada para criar uma cópia rasa ou profunda do vetor de entrada que chega pela porta `b`. O argumento de tipo `P` representa um meta-pré-condicionador (tipo base `MetaPreconditioner`), responsável por criar o pré-condicionador a ser usado na resolução do sistema linear.

O algoritmo inicia pela declaração do vetor zero `x` e dos vetores `p`, `q` e `r`, todos contendo o mesmo tamanho do vetor `b` passado como parâmetro de entrada (linhas 11 a 16). Conforme representado pela Figura 5.1, iniciamos o fluxo de execução a partir da transferência simultânea dos dados de entrada `tol` e `maxIter` para a atividade `For 1..maxIter` e as entradas `A` e `b` às instâncias dos parâmetros de tipos de atividades `M A` e `V b`. Enquanto as atividades `M A` e `V b` são executadas, os valores escalares `res` e `rho_1` são produzidos pelas atividades `Scalar res` e `Scalar rho_1` e transferidos a atividade `For 1..maxIter`.

Após a produção de `A` e `b` formatados por `M A` e `V b`, o workflow se ramifica para que, paralelamente, seja criado o vetor `r` a partir da cópia de `b` pela atividade `Copy r`, a instanciação de um pré-condicionador através da atividade `P` e a identificação do tamanho de `b` através da atividade `Size n`. Em seguida, `n` é transferido para as atividades `V x` e `V p`, produzindo os vetores `v` e `p`. Todos os vetores produzidos são enviados para `For 1..maxIter`, permitindo que seu conteúdo seja executado.

A atividade `For 1..maxIter` representa as linhas 19 a 40 e, de acordo com a Figura 5.2 é composta por uma atividade `statement`, que consome todas as entradas disponíveis e, ao final de sua sequência de execuções, retornar o sistema linear resolvido juntamente com a quantidade de iterações necessárias e o valor de tolerância alcançado.

Ao abrirmos a atividade `statement`, podemos observar um grande conjunto de relacionamentos entre atividades. Conforme pode ser observado pela Figura 5.3, o algoritmo apresenta grande dependência de dados em cada linha. Em alguns casos, é necessário que o usuário crie relações de dependência de controle para garantir a sincronia da execução e os dados compartilhados por dois ramos distintos do fluxo como, por exemplo, a dependência de `Assign rho_1 = rho` (linha 39) com a atividade `if current > 1` (linha 25), pois sem o relacionamento de controle a atribuição

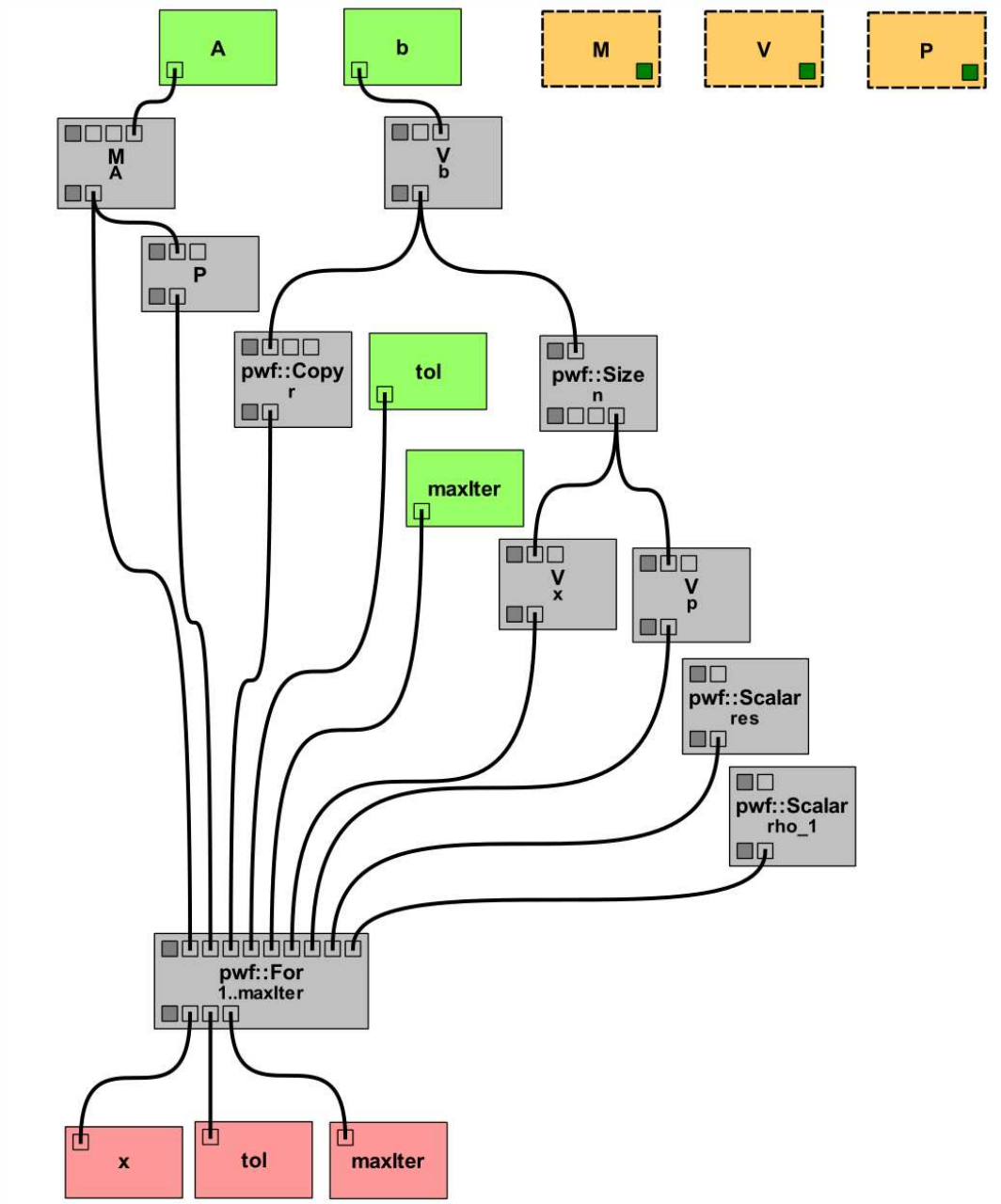


Figura 5.1: Workflow `CGSolver`.

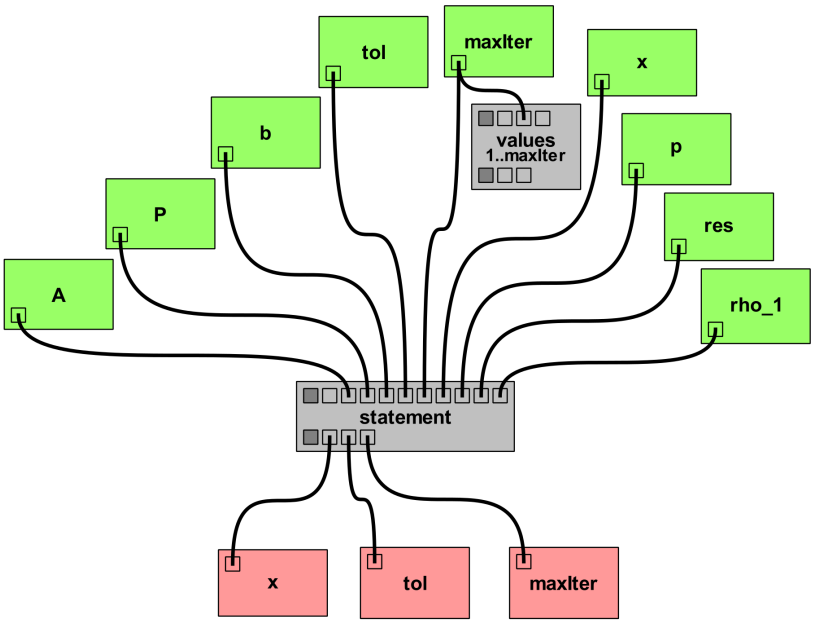


Figura 5.2: Atividade `For 1..maxIter`.

poderia ser feita antes que o if pudesse ser avaliado e, caso sua condição fosse verdadeira, o resultado de seu bloco seria afetado.

Considerando o tempo de execução de cada atividade constante e a transferência de dados ou sinais de controle entre as atividades como sendo instantânea ou com tempo insignificante, podemos definir a ordem de execução das atividades da seguinte forma:

1 Transferências de dados de entrada `b`, `P`, `p`, `rho_1`, `current`, `A`, `x`, `tol`, `res` e `maxIter` para seus respectivos destinos;

2 `Expression x = P.solve(b)` (linha 21);

3 `Norm norm(z)` (parte da condição do if da linha 33) e `Dot rho = dot(z,b)` (linha 22);

4 `if current > 1` (linhas 24 e 25);

5 `Copy copy(z)` (linha 26) e `Assign rho_1 = rho` (linha 39);

6 `Assign p = copy(z)` (linha 26);

7 `Dot dot(p,q)` (linha 27);

8 `Divide alpha = rho / dot(p,q)` (linha 29);

9 `Minus -alpha` e `Axpy axpy(x,alpha,p);`

10 `Norm norm(x)` (parte da condição do if da linha 33);

11 `Divide norm(z) / norm(x)` (parte da condição do if da linha 33);

12 `if res <= tol` (linhas 33 a 38).

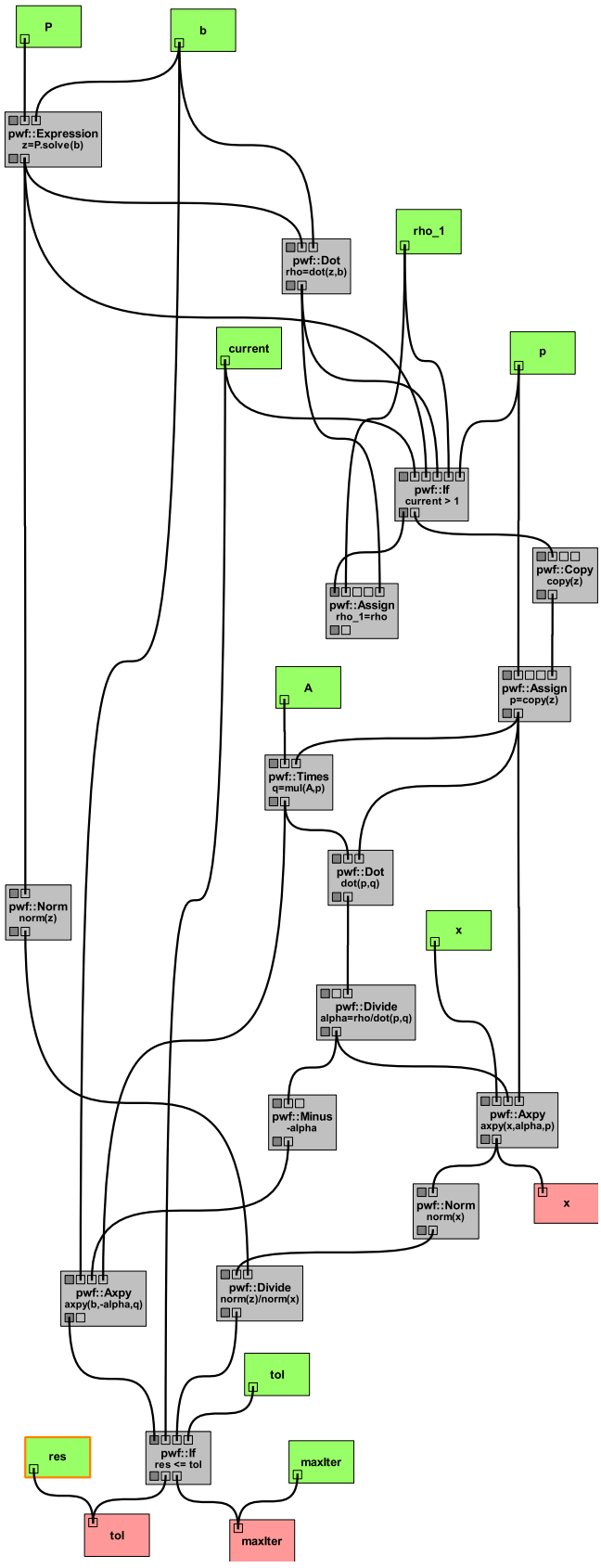


Figura 5.3: Bloco da atividade `statement`.

A atividade `CGSolver` utilizada a seguir foi desenvolvida em C++, apresentando como argumento de tipo apenas o `Preconditioner`, que pode ser desenvolvida para que seja executada em CPU ou GPU. A Figura 5.4 representa um workflow desenvolvido para realizar a leitura de duas matrizes, resolver o sistema linear e persistir seu resultado em um arquivo de saída. Esse exemplo realiza a instanciação da atividade `CGSolver` através da atividade `DiagPreconditioner`.

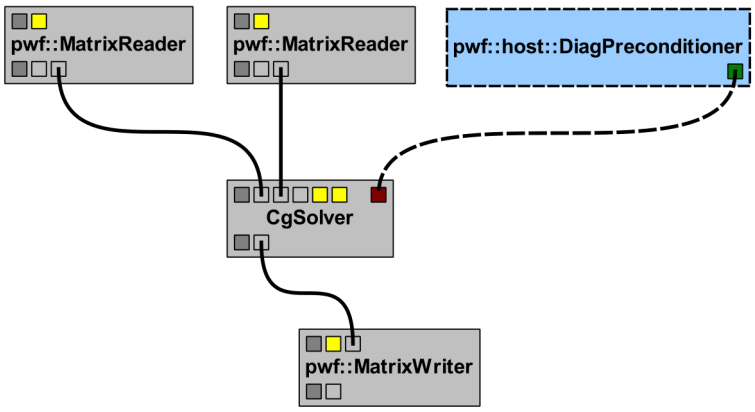


Figura 5.4: Instanciação de `CGSolver` para CPU.

O mesmo workflow pode ser instanciado por um `Preconditioner` destinado à execução em GPU. Podemos observar pela Figura 5.5 que a atividade `CGSolver` é instanciada apenas pela troca do tipo de atividade executável em CPU. Outra diferença é utilização das atividades `Vector` e `Matrix`, responsáveis, respectivamente, por formatar a representação de vetores e matrizes para permitir a manipulação dos dados em GPU. Uma alternativa para a utilização das atividades `Vector` e `Matrix` seria a extensão da atividade `MatrixReader` para que a leitura do arquivo de entrada produza o objeto de dados próprio para a manipulação em GPU.

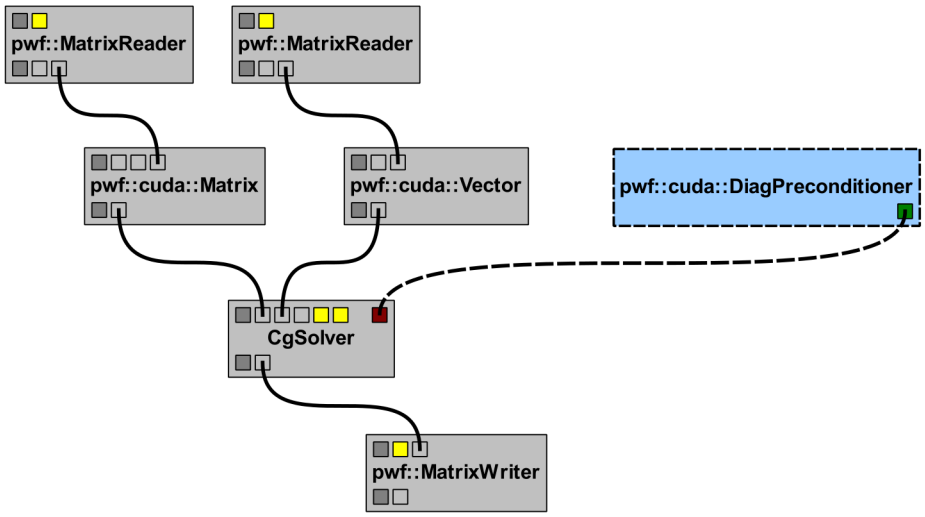


Figura 5.5: Instanciação de `CGSolver` para GPU.

5.3 Análise Estática via MEF

A aplicação do MEF foi desenvolvida através de um workflow paramétrico composto pelos seguintes componentes, Figura 5.6:

- `OpenFileDialog`: atividade utilizada para abrir um diálogo onde o usuário pode selecionar o arquivo de entrada contendo o domínio a ser analisado;
- `DomainReader`: leitor de domínio responsável por ler o arquivo escolhido e fornecer o `Domain` ao analisador;
- `Analysis`: atividade responsável pela análise do domínio lido;
- `DomainWriter`: escritor de domínio responsável por devolver ao arquivo inicial o domínio resultante da análise.

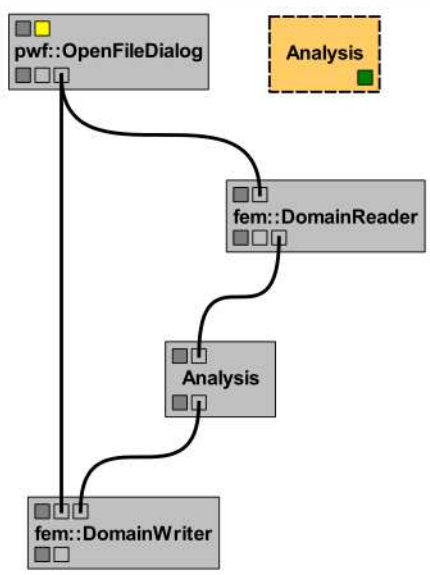


Figura 5.6: Atividade `FEMMain`.

Conforme representado pela Figura 5.6, o componente `Analysis` é uma instância do argumento de tipo `Analysis`, de tipo base `FemAnalysis`, e que deverá ser fornecido à atividade `FemMain`. Um componente do tipo `FemAnalysis` é caracterizado por elementos dos seguintes tipos:

- `Domain`: contêiner de elementos finitos, nós e condições de contorno (dadas por casos de carregamento e restrições de DOFs) que representa o domínio a ser analisado. Além desses dados, um objeto do tipo `Domain` armazena, para cada nó da malha de elementos finitos, um vetor contendo os índices dos elementos que incidem no nó. Esse vetor de incidências é usado, por exemplo, no esquema de montagem por nós da matriz de rigidiz em GPU, conforme explicado no Capítulo 3;
- `ConstraintHandler`: atividade responsável por manipular as condições de contorno impostas ao `Domain`;

- `DOFNumberer`: atividade responsável por mapear os DOFs do domínio em números de equações do sistema linear (com o objetivo de minimizar a largura de banda da matriz de rigidez ou, mais genericamente, minimizar o consumo de memória para armazenamento dos elementos não-nulos da matriz de rigidez);

- `FemAlgorithm`: atividade de gerenciamento dos passos executados no processo de análise.

A partir do tipo de atividade `Analysis` é possível desenvolver analisadores estáticos, dinâmicos ou baseados em decomposição de domínio para realizar a instanciação do workflow. Para cada tipo de analisador é possível utilizar diferentes componentes que, juntos, definirão o fluxo de execução do analisador. Mais especificamente, a atividade da Figura 5.7 representa um analisador estático do tipo `FemStaticAnalysis` cujo fluxo de execução é definido em função dos seguintes parâmetros de tipo a serem acoplados à atividade:

- `C`, de tipo base `ConstraintHandler`: gerenciador de restrições de DOFs usado na análise;

- `D`, de tipo base `DofNumberer`: numerador de DOFs do domínio;

- `LS`, de tipo base `MetaLinearSolver`: construtor do sistema linear (objeto de uma classe derivada da classe abstrata `LinearSystem`) a ser usado na análise;

- `Algorithm`, de tipo base `FemAlgorithm`: algoritmo a ser usado na análise;

- `Assembler`, de tipo base `MetaAssembler`: construtor do montador do sistema linear a ser usado na análise. Um montador é um componente da API de elementos finitos responsável por adicionar à matriz de rigidez e ao vetor de esforços nodais equivalentes as contribuições de todos os elementos e nós de um domínio, Equação (3.11).

A criação da atividade `FemStaticAnalysis` permite que o usuário configure cada parâmetro com diferentes métodos sem que a estrutura de execução seja alterada. Os parâmetros `C` e `D` podem ser instanciados por atividades dos tipos `PlainConstraintHandler` e `SimpleDofNumberer`, respectivamente, presentes na biblioteca do sistema. A atividade `PlainConstraintHandler` é responsável por rotular com um número negativo todos os DOFs prescritos, indicando que as linhas e colunas da matriz de rigides global (e também os elementos do vetor de esforços nodais equivalentes) correspondentes serão eliminados do sistema linear. A atividade `SimpleDofNumberer` é responsável por numerar os DOFs do domínio (rotulados como não negativos) com o objetivo de reduzir a largura de banda da matriz de rigidez global. O método consiste em numerar os DOFs incógnitos de cada elemento na sequência, começando de um elemento qualquer e prosseguindo nos elementos vizinhos daqueles com os DOFs já numerados. O usuário também pode criar novas atividades cujos tipos sejam compatíveis com os parâmetros de `FemStaticAnalysis` e então instanciá-los. Os passos da atividade são:

1. Execução da atividade `C` (instância do argumento de tipo `C`), responsável pela manipulação das condições de contorno aplicadas ao domínio de entrada;

2. Execução da atividade `D` (instância do argumento de tipo `D`), responsável pela numeração dos DOFs incógnitos;

3. Execução de uma expressão que envia uma mensagem ao sistema linear `LS` (instância do argumento de tipo `LS`) para que este determine as dimensões de sua matriz e vetores em função do número de DOFs incógnitos do domínio;

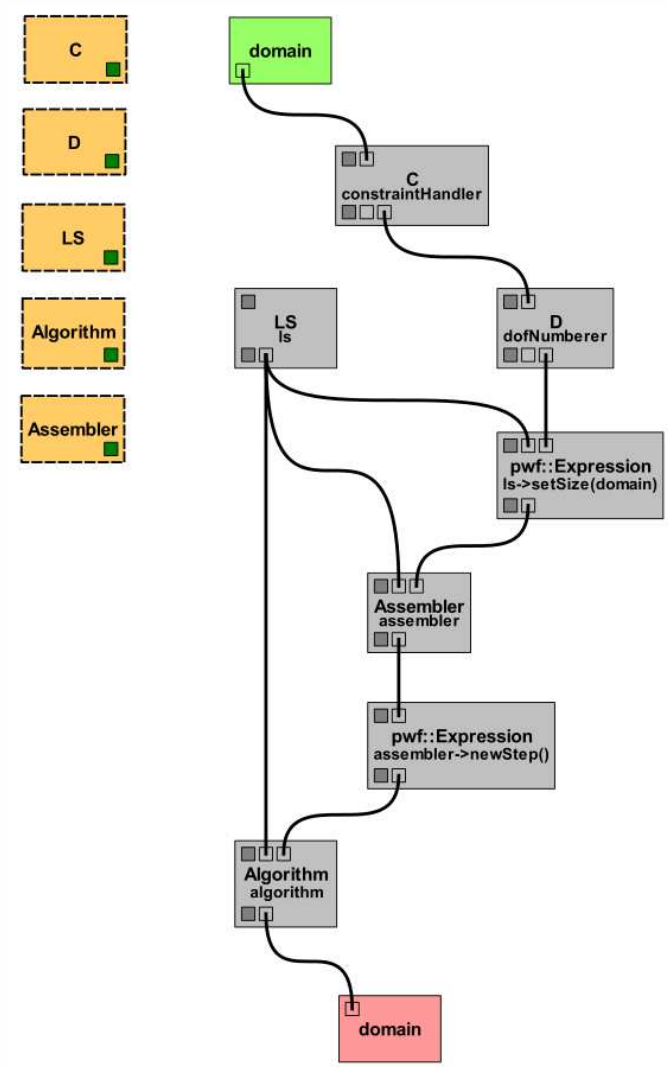


Figura 5.7: Atividade do tipo `FemStaticAnalysis`.

4. Execução da atividade `Assembler` (instância do argumento de tipo `Assembler`, meta-montador responsável pela criação do montador de sistema linear a ser usado na análise;

5. Execução de uma expressão que envia uma mensagem ao montador para que este inicie um novo passo de análise (zerar a matriz e os vetores do sistema a ser montado, entre outras tarefas); e

6. Execução do algoritmo de análise (instância do argumento de tipo `Algorithm`), descrito a seguir.

A atividade `Algorithm` orquestra a montagem e resolução do sistemas de equações. A Figura 5.8 representa a atividade `FemLinearAlgorithm`, responsável pela invocação de métodos de um algoritmo de análise via MEF baseado em equilíbrio estático [35]. A atividade tem como portas de entrada `assembler` e `ls`, respectivamente o montador e o sistema linear da análise, e como porta de saída `domain`, o domínio analisado. Além disso, a atividade conta com o argumento de tipo `S`, um meta-solucionador de sistemas lineares genérico, usado para criar o componente que irá determinar a solução do sistema linear `ls`.

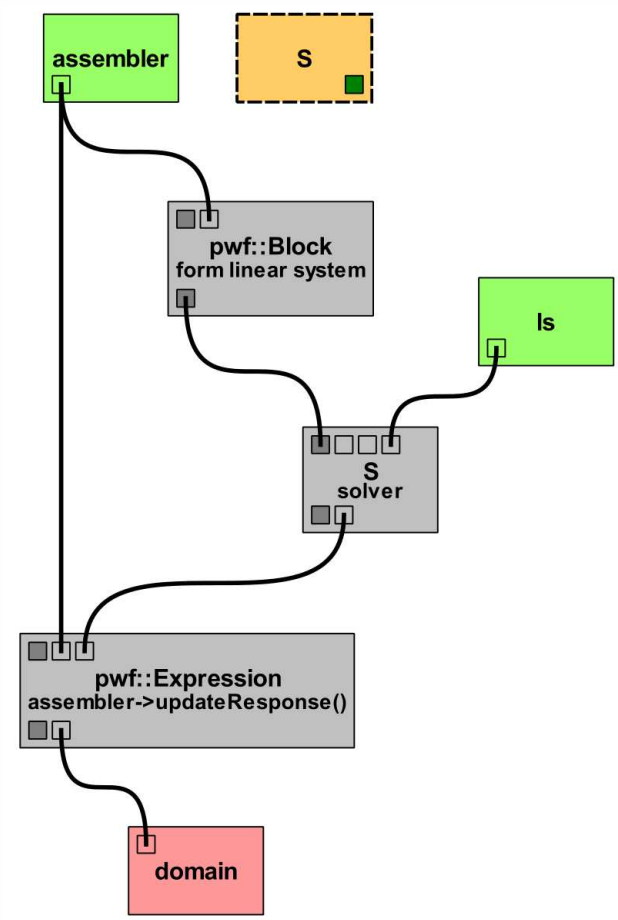


Figura 5.8: Atividade `FemLinearAlgorithm`.

Os passos executados pelo algoritmo linear são:

1. Execução de um bloco no qual são enviadas mensagens ao montador para que este adicione a contribuição de cada elemento finito e nó do domínio ao lado esquerdo e direito do sistema linear, Figura 5.9;

2. Execução da atividade `S` (instância do argumento de tipo `S`), responsável pela solução do sistema linear;

3. Execução de uma expressão que envia uma mensagem ao montador `assembler` para que este atualize o domínio com os dados oriundos do vetor solução do sistema linear `ls`.

A partir dos componentes desenvolvidos podemos instanciar o algoritmo do MEF. No workflow da Figura 5.10, a atividade `FemApplication` é instanciada usando-se os seguintes tipos de atividade: manipulador de restrições do tipo `PlainConstraintHandler` e numerador de DOFs do tipo `SimpleDofNumberer` (veja [12] para detalhes); construtor de sistema linear do tipo `BandSystem`, o qual cria um sistema linear cuja matriz, em banda, é armazenada em um arranjo retangular na memória da CPU; um algoritmo do tipo `FemLinearAlogorithm`, descrito acima; e um montador do tipo `StaticAssembler`, o qual monta o sistema linear em CPU usando o esquema tradicional de elementos finitos, isto é, percorrendo sequencialmente os elementos finitos e adicionando a matriz de rigidez local e o vetor de esforços nodais equivalentes

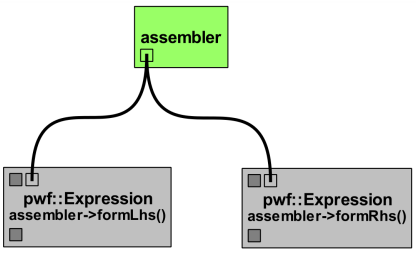


Figura 5.9: Bloco de formação do sistema na atividade `FemLinearAlgorithm`.

de cada elemento a **K** e **F**, respectivamente. Podemos observar, na Figura 5.10, que a atividade `FemLinearAlgorithm` também é paramétrica, podendo ser instanciada, como feito neste exemplo, pela atividade `CGSolver`. Finalmente, a atividade `CGSolver`, quando instanciada, usará um pré-condicionador diagonal em CPU, um objeto a ser criado por um meta-pré-condicionador da classe `DiagPreconditioner`.

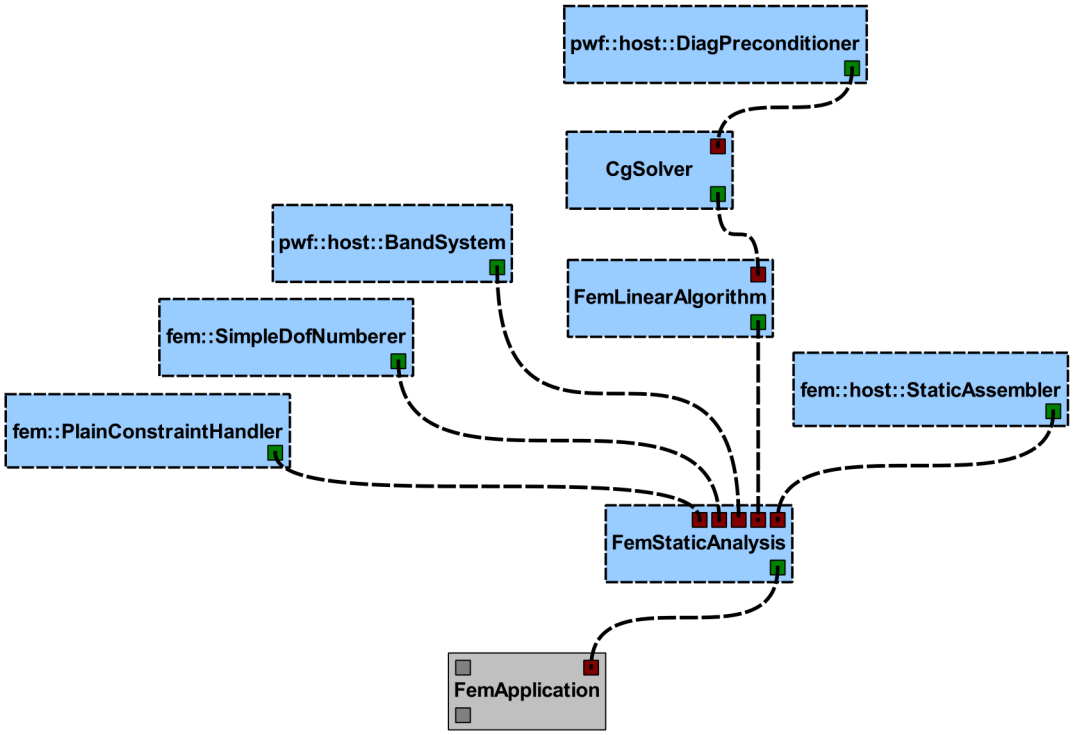


Figura 5.10: `FEMMain` para CPU.

As atividades `BandSystem`, `DiagPreconditioner` e `StaticAssembler` foram desenvolvidas para que o workflow seja executado em CPU. O mesmo workflow pode ser redefinido para que sua execução seja executada em GPU, alterando-se apenas a instanciação das atividades `BandSystem` e `DiagPreconditioner` por suas respectivas representações para CUDA e a substituição de `StaticAssembler` pelo tipo `ColorStaticAssembler`. Este é um meta-montador que cria um montador de sistema linear armazenado em GPU usando o esquema de cores mencionado no Capítulo 3. Tal montador usa um algoritmo de coloração de elementos finitos de um domínio que é implementado em métodos da classe `DomainColor`, conforme mostrado na Figura 5.11.

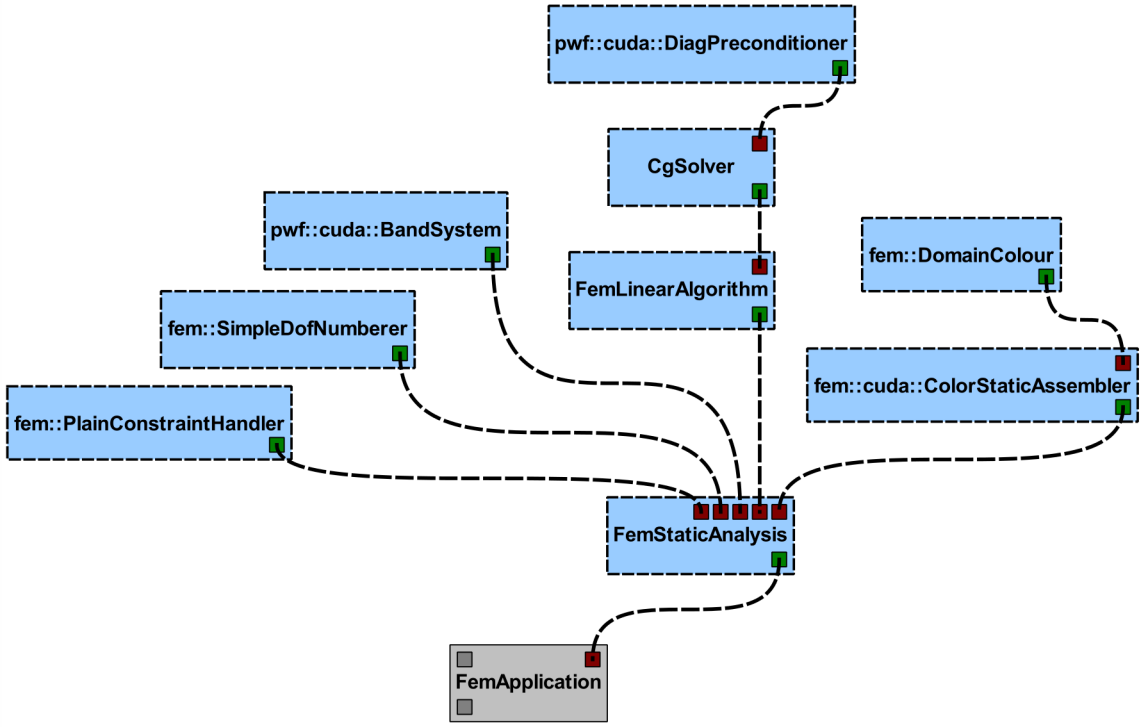


Figura 5.11: `FEMMain` para GPU.

5.4 Análise por Decomposição de Domínio

Uma aplicação de análise via MEF com decomposição de domínio pode ser gerada pelo workflow paramétrico `FemSubdomainApplication`, mostrado na Figura 5.12.

Tal como a aplicação de análise estática descrita anteriormente, a aplicação baseada em decomposição de domínio usa atividades dos tipos `DomainReader` e `DomainWriter` para ler e, finda a análise, salvar o domínio analisado em arquivo, respectivamente. Antes de executar a análise numérica, contudo, a aplicação efetua um particionamento do domínio lido. Esta tarefa é realizada com a execução da atividade `P`, instância do argumento de tipo `P`, cujo tipo base é `DomainPartitioner`. Um particionador de domínio é um componente da API que toma como entrada um objeto da classe `Domain` e gera como saída um objeto da classe `PartitionedDomain`, derivada de `Domain`. Um domínio particionado é um domínio cujos elementos finitos são subdomínios, isto é, objetos da classe `Subdomain`, derivada de `Domain` e também de `Element`, e cujos nós são os nós externos de todos os subdomínios, conforme definido no Capítulo 3 (veja [12] para detalhes). Gerado o domínio particionado, o workflow executa `FemStaticAnalysis`, já vista, para efetuar normalmente a análise estática, a qual toma como entrada o domínio gerado pelo particionador.

Os argumentos de tipo `C` (de tipo base `ConstraintHandler`), `D` (de tipo base `DOFNumberer`), `LS` (de tipo base `MetaLinearSystem`) e `Algorithm` (de tipo base `FemAlgorithm`) são os mesmos requeridos para a análise estática e, de fato, acoplados à atividade `FemStaticAnalysis`, como observado na Figura 5.12. A diferença é que o montador acoplado à análise estática de uma aplicação com decomposição de domínio é um objeto do tipo `FemSubdomainAssemler`, responsável por:

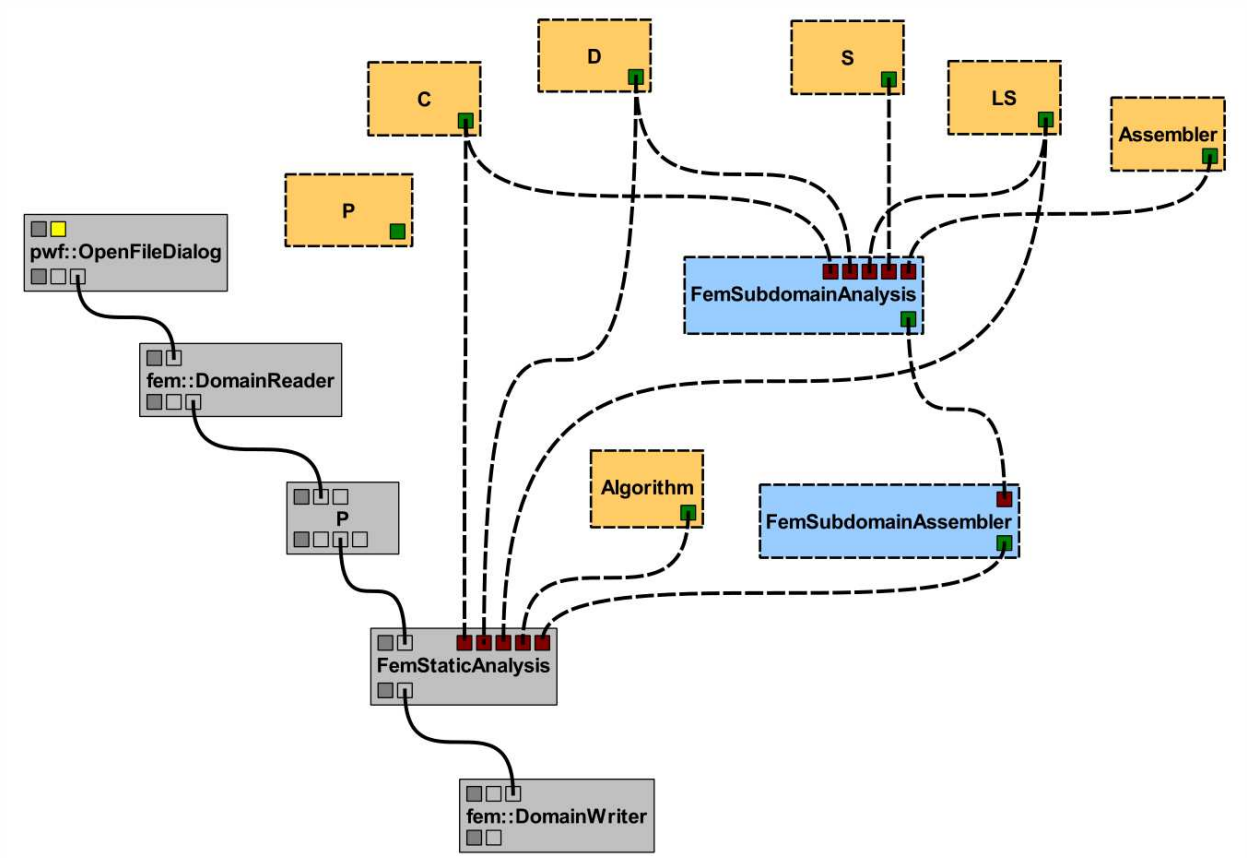


Figura 5.12: Aplicação do MEF para subdomínios.

1. Determinar e adicionar a contribuição de cada subdomínio ao lado esquerdo e direito do sistema global. Tal determinação envolve a análise estática do subdomínio (considerado para isso como domínio ao invés de elemento) seguida da condensão estática para formação da matriz $\mathbf{K}_b$ e vetor $\mathbf{F}_b^{(\beta)}$, dados pelas Equações (3.30) e (3.31), respectivamente. Esta análise é efetuada por uma atividade paramétrica do tipo `FemSubdomainAnalysis`, mostrada no workflow da Figura 5.13 e descrita a seguir.

2. Atualizar os resultados nos subdomínios logo após a solução do sistema linear global, conforme descrito no Capítulo 3.

A atividade paramétrica `FemSubdomainAnalysis` é responsável pela análise estática de um subdomínio, com a diferença que, ao invés de resolver o sistema de equações, efetua a condensação estática do mesmo a fim de computar a matriz de rigidez local e o vetor de esforços equivalentes do subdomínio. Os argumento de tipos são os mesmos da análise estática, sendo a diferença `S`, de tipo base `SubdomainDomainSolver`. Este é um tipo de solucionador de sistema linear com a capacidade de efetuar a condensação estática do sistema. Feita essa distinção, o workflow da Figura 5.13 não é complicado de seguir. O bloco responsável pelas mensagens que computam as contribuições do subdomínio para o sistema global é mostrado na Figura 5.14.

O workflow da Figura 5.15 ilustra uma aplicação de análise com decomposição de domínio em GPU, usando um particionador baseado no Metis [12], um montador em GPU com esquema nodal (`NodalStaticAssembler`) e um solucionador de subdomínio cujo sistema linear tem a matriz armazenada em banda (`BandSubdomainSolver`).

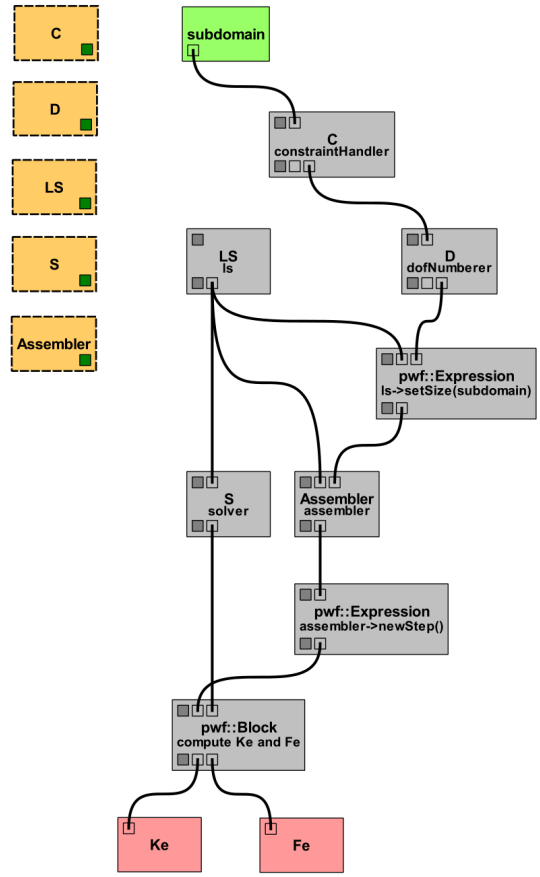


Figura 5.13: Atividade `FemSubdomainAnalysis`.

5.5 Comentários Finais

A utilização do modelo apresentado proporciona grande liberdade no desenvolvimento de workflows paramétricos, pois o usuário pode definir diferentes métodos de manipulação e representações de tipos de dados, além de permitir que novas atividades sejam desenvolvidas tanto através da criação de workflows baseados em componentes existentes quanto através da relação de herança de novas classes com as atividades presentes no modelo.

Os exemplos presentes neste capítulo mostram que a abordagem de workflows paramétricos desenvolvida neste projeto permite maior flexibilidade através da substituição de tipos de atividades. Suponhamos que o exemplo apresentado seja desenvolvido em uma das aplicações apresentadas no Capítulo 2 e seja necessário comparar o desempenho de diferentes tipos de `Assembler` mantendo a estrutura original do workflow. O usuário deverá desenvolver cada componente a ser comparado e substituí-lo no analisador representado pela Figura 5.7.

A substituição do componente é realizada retirando-se o `Assembler` presente e incluindo a nova atividade. A remoção do `Assembler` original resulta na perda de suas conexões com as outras atividades, sendo necessário recriá-las para o novo `Assembler`. Em workflows maiores, a quantidade de instâncias das atividade que podem ser substituídas pode resultar em dezenas de conexões refeitas, exigindo tempo e conhecimento prévio do fluxo para minimizar a possibilidade de inserções de erros. A utilização do modelo apresentado exige que apenas uma conexão seja recriada, preservando o fluxo de execução independente dos componentes que o compõe.

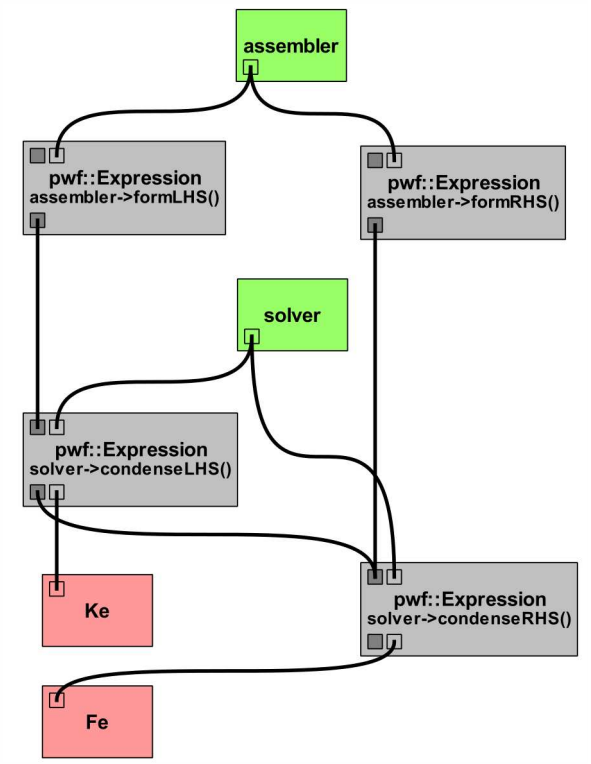


Figura 5.14: Bloco de `FemSubdomainAnalysis` que computa as contribuições do subdomínio.

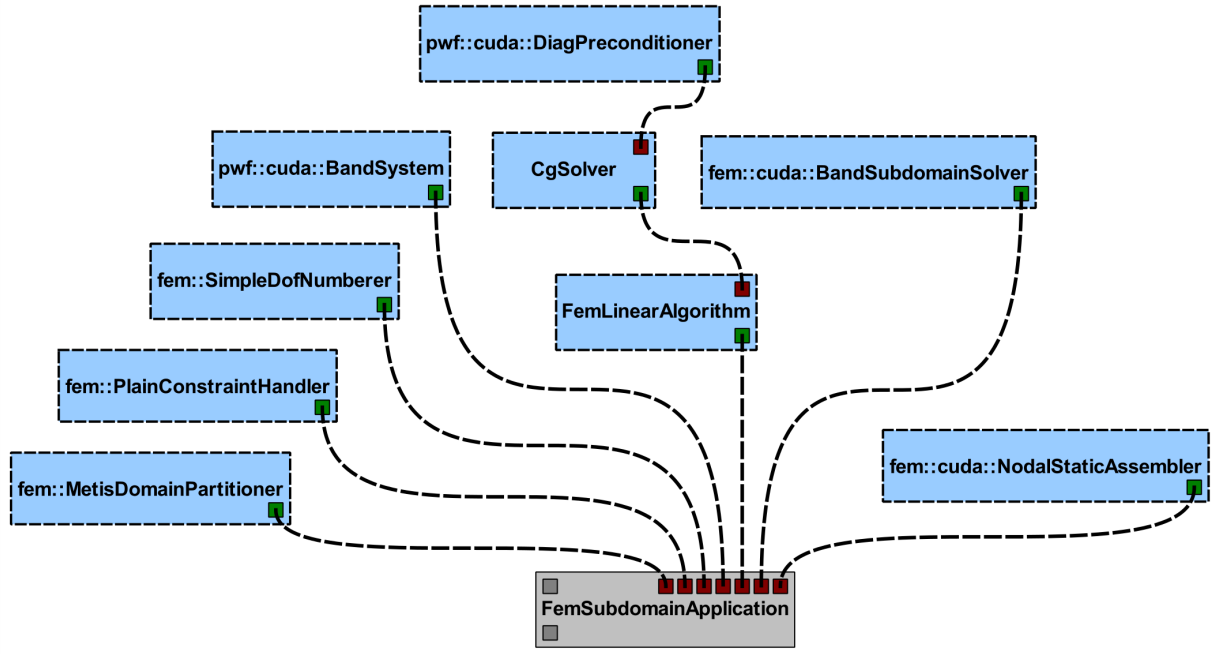


Figura 5.15: Análise com decomposição de domínio em GPU.

CAPÍTULO 6

Conclusão

6.1 Discussão dos Resultados Obtidos

O objetivo geral deste trabalho foi o desenvolvimento de um sistema de workflows paramétricos para especificação e execução de aplicações baseadas no método dos elementos finitos (MEF) em ambientes paralelos heterogêneos.

Apresentamos no Capítulo 2 os conceitos básicos que definem workflows científicos e descrevemos as principais características presentes nas ferramentas de gerenciamento de workflows. Através dessa apresentação, justificamos o desenvolvimento deste trabalho, pois apesar de haver ferramentas robustas para a representação de workflows científicos, não foram encontrados projetos que permitissem o desenvolvimento de workflows baseados em componentes capazes de aproveitar os recursos de GPU.

Outro fator motivacional foi a proposição de uma nova abordagem para o conceito de workflows paramétricos, que nos permitiu desenvolver workflows cuja substituição de componentes não necessita do rompimento e reconstrução de ligações entre diversos componentes. Em nossa abordagem um workflow é definido a partir da utilização de parâmetros de tipos de atividades, que permitem a construção de fluxos de execução genéricos e podem ser instanciados através do acoplamento de tipos. Cada tipo de atividade acoplado é utilizado para substituir as atividades genéricas por instâncias do tipo especificado, resultando em um workflow executável. Nossa abordagem foi incluída ao modelo proposto, apresentado no Capítulo 4, permitindo a representação dos principais conceitos presentes em linguagens de programação. Esse modelo serviu de base para o desenvolvimento do sistema apresentado no Capítulo 4, permitindo representar o desenvolvimento de aplicações baseadas no MEF a partir de uma interface gráfica e proporcionando ao usuário maior facilidade na comparação entre diferentes componentes disponíveis.

Os objetivos específicos foram:

- *Implementar todos os componentes do sistema proposto.* Os componentes desenvolvidos foram baseados em um modelo de representação de workflows e apresentados no Capítulo 4. Desenvolvemos representações gráficas dos componentes propostos e as disponibilizamos em uma interface gráfica, apresentada na Seção 4.5, que permite ao usuário a criação, edição e execução de workflows, possibilitando seu reuso como atividades no desenvolvimento de novos workflows. A partir dos componentes apresentados, desenvolvemos um conjunto de atividades primitivas que representam as principais estruturas e conceitos presentes em linguagens de programação. Desenvolvemos também um conjunto de atividades responsáveis

pela representação de estruturas e processos utilizados em pipelines de análises baseadas no MEF, resultando em uma API capaz de representar fluxos híbridos com fluxos de controle e dados.

A partir dos componentes descritos, possibilitamos a representação de uma abordagem distinta ao conceito de workflow paramétrico, que possibilita a definição de um processo que pode ser facilmente instanciado com diferentes componentes. Com a comparação entre a execução das diferentes instanciações, é possível verificar qual a melhor combinação de componentes e observar quais aplicações são favorecidas pela utilização dos recursos de GPU.

Desenvolvemos também um motor de execução de workflows, inserido à biblioteca e apresentado em detalhes na Seção 4.5. O modelo de execução apresentado utiliza um motor de execução para gerenciar cada bloco presente no processo. O gerenciamento descentralizado da execução de componentes permite que cada bloco tenha sua execução alocada de acordo com os recursos computacionais disponíveis, permitindo, por exemplo, a utilização de clusters. A execução de workflows gera um log de execução que permite a comparação entre diferentes componentes utilizados para a mesma finalidade.

- *Criar uma biblioteca de atividades correspondentes aos componentes da API do MEF e, com isto, workflows para análise elastostática de sólidos em CPU e/ou GPU.* Os conceitos apresentados no Capítulo 3 e representados pela API desenvolvida em [34] foram representados como atividades a partir da extensão de nosso modelo, conforme apresentado na Seção 4.4. As atividades criadas passaram a compor o acervo de atividades disponíveis na interface desenvolvida, permitindo que workflows fossem desenvolvidos conforme apresentado no Capítulo 5.

Apesar de consistente, o modelo apresenta limitações com relação à gestão de dados. Atualmente as atividades possuem acesso apenas aos dados recebidos em suas portas, não sendo possível, por exemplo, compartilhar dados entre todos os componentes de um mesmo escopo sem a criação de canais. A transferência de dados entre escopos aninhados só é permitida mediante a replicação de portas para os blocos internos. Essa limitação torna o procedimento de transferência de dados moroso em algoritmos maiores e mais complexos. Uma solução para essa limitação poderá ser desenvolvida através da adequação do modelo para permitir a declaração de variáveis de escopo, onde cada bloco teria seu escopo próprio e poderia acessar variáveis acessíveis ao escopo de seu bloco pai. A declaração de variáveis teria por objetivo o compartilhamento de objetos de dados considerados públicos, enquanto os dados transferidos entre canais seriam privados.

A transferência de forma compartilhada dos dados produzidos, apesar de garantir maior aproveitamento da memória e permitir que várias atividades atuem sobre partes de um mesmo objeto, obriga que o usuário realize o controle de concorrência entre as atividades para garantir a consistência dos dados. Uma solução que deverá ser estudada é a criação de mecanismos de gerenciamento de acesso aos atributos de objetos de dados. Cada objeto poderia, por exemplo, gerenciar bloqueios de acesso com permissão de consulta e escrita a seus dados, garantindo a consistência em condições de concorrência entre as atividades.

O sistema proposto, conforme apresentado no Capítulo 5, permite o desenvolvimento de aplicações graficamente e facilita a comparação entre diferentes componentes através da definição de uma abordagem distinta para o conceito de workflows paramétricos. Diante do exposto, acreditamos que os objetivos deste trabalho foram alcançados.

6.2 Trabalhos Futuros

O trabalho desenvolvido apresentou um pequeno conjunto de funcionalidades destinadas ao desenvolvimento de workflows paramétricos, que poderá ser aprimorado através da inclusão de novas funcionalidades e união de novos conceitos e padrões de desenvolvimento de workflows. Destacamos algumas características e funcionalidades que poderão ser adicionadas ou aprimoradas neste projeto:

- A interface poderá ser melhorada a partir da criação de novos eventos, responsáveis pelo tratamento de comandos de teclado para selecionar, copiar e colar um conjunto de atividades, permitir a exportação dos logs de execução em arquivos de saída e visualização gráfica dos dados produzidos pela execução de workflows.

- Novos padrões de desenvolvimento de workflows poderiam ser adicionados como, por exemplo, o tratamentos de exceções e tolerância a falhas, permitindo que o usuário identifique com maior facilidade os erros presentes no workflow em execução. A execução de workflows também poderia ser alterada, permitindo que o processo fosse pausado e que os dados produzidos até o momento fossem visualizados.

- Estudos podem ser realizados para proporcionar o gerenciamento de recursos disponíveis através do sistema, permitindo que o usuário administre recursos em diferentes ambientes computacionais, desde o gerenciamento de núcleos de CPU e GPU de um único computador até a utilização de clusters ou computadores conectados em rede.

- A biblioteca básica de componentes pode ser aprimorada para possuir estruturas mais complexas como, por exemplo, um componente capaz de criar instâncias de uma mesma atividade dinamicamente baseado na quantidade de recursos computacionais disponíveis. É possível criar classes responsáveis pela definição de estruturas mais complexas de dados, proporcionando maior flexibilidade e eficiência na manipulação e tráfego de informações entre as atividades.

- O modelo proposto pode ser estendido para permitir a representação de classes, permitindo que um conjunto de métodos sejam representados através de workflows e agrupados juntamente com a declaração de atributos. Essa extensão pode ser desenvolvida a partir da criação de uma classe `TypeClass`, que permitirá representar atributos e métodos com os diferentes tipos de visibilidade e deverá ser capaz de validar a existência e a permissão de acesso a cada elemento. O modelo também deverá ser capaz de gerenciar seus componentes de modo a representar fielmente as características que definem o paradigma orientado a objetos.

- As diferentes abordagens utilizadas para a definição de workflows paramétricos poderiam ser agrupadas no mesmo sistema. A mesclagem entre as abordagens permitiria, por exemplo, a criação de templates de workflows cujos parâmetros podem ser instanciados por vários tipos de atividade. A execução desse tipo de workflow criaria um conjunto de instâncias através do produto cartesiano entre os tipos de atividades utilizados para instanciar parâmetros distintos, sendo executadas sucessivamente. Ao final da execução das instâncias, o sistema poderia comparar o desempenho dos diferentes tipos de atividades utilizados para instanciar o mesmo parâmetro, retornando a melhor combinação de componentes disponível.

- Estudar a possibilidade de utilizar o sistema proposto como método auxiliar de apresentação de disciplinar de programação, desenvolvimento de sistemas paralelos, distribuídos ou apli-

cações baseadas em métodos numéricos, pois a representação gráfica de componentes pode auxiliar o entendimento dos conceitos e estruturas envolvidos.

Referências

[1] Wings project. www.wings-workflows.org/about, 2012. Acessado em 10/08/2013.

[2] AHRENS, J., LAW, C., SCHROEDER, W., MARTIN, K., INC, K., AND PAPKA, M. A parallel approach for efficiently visualizing extremely large, time-varying datasets. Tech. rep., 2000.

[3] BAVOIL, L., CALLAHAN, S. P., SCHEIDEGGER, C. E., VO, H. T., CROSSNO, P., SILVA, C. T., AND FREIRE, J. Vistrails: Enabling interactive multiple-view visualizations. In *IEEE Visualization* (2005), IEEE Computer Society, p. 18.

[4] BONIFATI, A., CASATI, F., DAYAL, U., AND SHAN, M.-C. Warehousing workflow data: Challenges and opportunities. In *Procs. of VLDB'01* (2001), pp. 649–652.

[5] BOSTIC, K., BROWN, T., CHANSLER, R., BRYANT, R., BRYANT, R., CANINO-KOENING, R., CESARINI, F., AND ALLMAN, E. The architecture of open source applications - vtk. www.aosabook.org/en/vtk.html, 2011. Acessado em 10/08/2013.

[6] BOWERS, S., LUDASCHER, B., NGU, A. H. H., AND CRITCHLOW, T. Enabling scientific workflow reuse through structured composition of dataflow and control-flow. In *Proceedings of the 22nd International Conference on Data Engineering Workshops* (Washington, DC, USA, 2006), ICDEW '06, IEEE Computer Society, pp. 70–.

[7] BUCK, J., HA, S., LEE, E. A., AND MESSERSCHMITT, D. G. Ptolemy: A mixed-paradigm simulation/prototyping platform in c++. In *In Proceedings of the C++ At Work Conference* (1991).

[8] CALLAHAN, S. P., FREIRE, J., SANTOS, E., SCHEIDEGGER, C. E., SILVA, C. T., AND VO, H. T. Vistrails: visualization meets data management. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 2006), SIGMOD '06, ACM, pp. 745–747.

[9] CHANSLER, R., BRYANT, R., BRYANT, R., CANINO-KOENING, R., CESARINI, F., ALLMAN, E., BOSTIC, K., AND BROWN, T. The architecture of open source applications - vistrails. www.aosabook.org/en/vistrails.html, 2011. Acessado em 10/08/2013.

[10] CHASE, J., SCHUCHARDT, K., CHIN, G., NULL JR., DAILY, J., AND SCHEIBE, T. Iterative workflows for numerical simulations in subsurface sciences. *Services, IEEE Congress on 0* (2008), 461–464.

[11] COALITION, W. M. Wfmc-tc-1011 ver 3 terminology and glossary english. www.wfmc.org/Download-document/WFMC-TC-1011-Ver-3-Terminology-and-Glossary-English.html, 1999. Acessado em 10/08/2013.

[12] DANTAS, B. Um framework para análise sequencial e em paralelo de sólidos elásticos pelo método dos elementos finitos. Master's thesis, Universidade Federal de Mato Grosso do Sul, 2006.

[13] DAVIS, U., BARBARA, U. S., AND DIEGO, U. S. The kepler project. kepler-project.org, 2012. Acessado em 10/08/2013.

[14] DEELMAN, E., BLYTHE, J., GIL, Y., KESSELMAN, C., MEHTA, G., VAHI, K., BLACKBURN, K., LAZZARINI, A., ARBREE, A., CAVANAUGH, R., AND KORANDA, S. Mapping abstract complex workflows onto grid environments. *J. Grid Comput. 1*, 1 (2003), 25–39.

[15] DEELMAN, E., GANNON, D., SHIELDS, M., AND TAYLOR, I. Workflows and e-science: An overview of workflow system features and capabilities. *Future Gener. Comput. Syst. 25*, 5 (May 2009), 528–540.

[16] DEELMAN, E., SINGH, G., SU, M.-H., BLYTHE, J., GIL, Y., AND KESSELMAN, C. Pegasus : A framework for mapping complex scientific workflows onto distributed systems. *Sci. Program. 13*, 3 (2005), 219–237.

[17] EKER, J., JANNECK, J., LEE, E. A., LIU, J., LIU, X., LUDVIG, J., SACHS, S., XIONG, Y., AND NEUENDORFFER, S. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE 91*, 1 (2003), 127–144.

[18] EL-GAYYAR, M., LENG, Y., SHUMILOV, S., AND CREMERS, A. New execution paradigm for data-intensive scientific workflows. *Services, IEEE Congress on 0* (2009), 334–339.

[19] ELMROTH, E., HERNÁNDEZ, F., AND TORDSSON, J. Three fundamental dimensions of scientific workflow interoperability: Model of computation, language, and execution environment. *Future Generation Computer Systems 26*, 2 (2010), 245 – 256.

[20] GIL, Y., DEELMAN, E., ELLISMAN, M., FAHRINGER, T., FOX, G., GANNON, D., GOBLE, C., LIVNY, M., MOREAU, L., AND MYERS, J. Examining the Challenges of Scientific Workflows. *IEEE COMPUTER VOL 40*, 12 (2007), 24–32.

[21] GIL, Y., GROTH, P., RATNAKAR, V., AND FRITZ, C. Expressive reusable workflow templates. In *Proceedings of the 2009 Fifth IEEE International Conference on e-Science* (Washington, DC, USA, 2009), E-SCIENCE '09, IEEE Computer Society, pp. 344–351.

[22] GIL, Y., KIM, J., RATNAKAR, V., AND DEELMAN, E. Wings for pegasus: A semantic approach to creating very large scientific workflows. In *Proceedings of the OWLED*06 Workshop on OWL: Experiences and Directions, Athens, Georgia, USA, November 10-11, 2006* (2006).

[23] INSTITUTE, I. S. Pegasus 4.0 user guide. pegasus.isi.edu/wms/docs/4.0/pegasus-user-guide.pdf, 2012. Acessado em 10/08/2013.

[24] JAMIL, H., AND EL-HAJJ-DIAB, B. Bioflow: A web-based declarative workflow language for life sciences. *Services, IEEE Congress on 0* (2008), 453–460.

[25] JUVE, G., CHERVENAK, A. L., DEELMAN, E., BHARATHI, S., MEHTA, G., AND VAHI, K. Characterizing and profiling scientific workflows. *Future Generation Comp. Syst. 29*, 3 (2013), 682–692.

[26] KIM, J., DEELMAN, E., GIL, Y., MEHTA, G., AND RATNAKAR, V. Provenance trails in the wings/pegasus system. *Concurrency - Practice and Experience 20*, 5 (2008), 587–597.

[27] KUMAR, V. S., KURC, T., RATNAKAR, V., KIM, J., MEHTA, G., VAHI, K., NELSON, Y. L., SADAYAPPAN, P., DEELMAN, E., GIL, Y., HALL, M., AND SALTZ, J. Parameterized specification, configuration and execution of data-intensive scientific workflows. *Cluster Computing 13*, 3 (2010), 315–333.

[28] LATHERS, A., SU, M.-H., KULUNGOWSKI, A., LIN, A. W., MEHTA, G., PELTIER, S. T., DEELMAN, E., AND ELLISMAN, M. H. Enabling parallel scientific applications with workflow tools. In *Challenges of Large Applications in Distributed Environments* (2006).

[29] MANDAL, N., DEELMAN, E., MEHTA, G., SU, M.-H., AND VAHI, K. Integrating existing scientific workflow systems: the Kepler/Pegasus example. In *Proceedings of the 2nd workshop on Workflows in support of large-scale science* (New York, NY, USA, 2007), WORKS '07, ACM, pp. 21–28.

[30] MCPHILLIPS, T., BOWERS, S., ZINN, D., AND LUDÄSCHER, B. Scientific workflow design for mere mortals. *Future Generation Computer Systems 25*, 5 (2008), 541–551.

[31] MISSIER, P., SOILAND-REYES, S., OWEN, S., TAN, W., NENADIC, A., DUNLOP, I., WILLIAMS, A., OINN, T., AND GOBLE, C. Taverna, reloaded. In *Scientific and Statistical Database Management*, M. Gertz and B. Ludäscher, Eds., vol. 6187 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2010, pp. 471–481.

[32] NGU, A., BOWERS, S., HAASCH, N., MCPHILLIPS, T., AND CRITCHLOW, T. Flexible scientific workflow modeling using frames, templates, and dynamic embedding. In *Scientific and Statistical Database Management*, B. Ludäscher and N. Mamoulis, Eds., vol. 5069 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2008, pp. 566–572.

[33] OF UTAH, U. Vistrails 2.0 user guide. www.vistrails.org/usersguide/ v2.0/html/VisTrails.pdf, 2012. Acessado em 10/08/2013.

[34] PAGLIOSA, L. C. Simulação de corpos deformáveis em sistemas paralelos heterogêneos. Tech. rep., Universidade Federal de Mato Grosso do Sul, 2012.

[35] PAGLIOSA, P. *Um sistema de modelagem estrutural orientado a objetos*. PhD thesis, Universidade de Sâo Paulo, 1998.

[36] RUSSELL, N., EDMOND, D., TER HOFSTEDE, A., AND VAN DER AALST., W. Workflow resource patterns. www.workflowpatterns.com/documentation/ documents/Resource%20Patterns%20BETA%20TR.pdf, 2004. Acessado em 10/08/2013.

[37] RUSSELL, N., AND HOFSTEDE, A. H. M. T. Exception handling patterns in process-aware information systems. www.workflowpatterns.com/documentation/documents/BPM-06-04.pdf, 2006. Acessado em 10/08/2013.

[38] RUSSELL, N., TER HOFSTEDE, A., EDMOND, D., AND VAN DER AALST., W. Workflow data patterns. www.workflowpatterns.com/documentation/ documents/data_patterns%20BETA%20TR.pdf, 2004. Acessado em 10/08/2013.

[39] RUSSELL, N., TER HOFSTEDE, A., VAN DER AALST, W., AND MULYAR., N. Workflow control-flow patterns: A revised view. www.workflowpatterns.com/documentation/documents/BPM-06-22.pdf, 2006. Acessado em 10/08/2013.

[40] SHIELDS, M. Control- versus data-driven workflows. In *Workflows for e-Science*, I. Taylor, E. Deelman, D. Gannon, and M. Shields, Eds. Springer London, 2007, pp. 167–173.

[41] WANG, J., CRAWL, D., AND ALTINTAS, I. Kepler + hadoop: a general architecture facilitating data-intensive applications in scientific workflow systems. In *Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science* (New York, NY, USA, 2009), WORKS '09, ACM, pp. 12:1–12:8.

[42] YILDIZ, U., GUABTNI, A., AND NGU, A. H. H. Towards scientific workflow patterns. In *Proceedings of the 4th Workshop on Workflows in Support of LargeScale Science* (New York, NY, USA, 2009), WORKS '09, ACM, pp. 13:1—-13:10.