Universidade Federal do Ceará
Departamento de Computação
Mestrado e Doutorado em Ciências da Computação

Um Arcabouço para a Construção de Aplicações Baseadas em Componentes sobre uma Plataforma de Nuvem Computacional para Serviços de Computação de Alto Desempenho

Jefferson de Carvalho Silva

Fortaleza – Ceará
Fevereiro 2016

Universidade Federal do Ceará
Departamento de Computação
Mestrado e Doutorado em Ciências da Computação

Um Arcabouço para a Construção de Aplicações Baseadas em Componentes sobre uma Plataforma de Nuvem Computacional para Serviços de Computação de Alto Desempenho

Autor
Jefferson de Carvalho Silva

Orientador
Prof. Dr. Francisco Heron de Carvalho Junior

Tese apresentada à Coordenação do Programa de Mestrado e Doutorado em Ciências da Computação da Universidade Federal do Ceará como parte dos requisitos para obtenção do grau de ***Doutor em Ciência da Computação****.*

Fortaleza – Ceará
Fevereiro 2016

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca de Ciências e Tecnologia

S578a Silva, Jefferson de Carvalho.
Um arcabouço para a construção de aplicações baseadas em componentes sobre uma plataforma de nuvem computacional para serviços de computação de alto desempenho / Jefferson de Carvalho Silva. – 2016.
187 f. : il.; color.

Tese (doutorado) – Universidade Federal do Ceará, Centro de Ciências, Departamento de Computação, Programa de Pós-Graduação em Ciência da Computação, Fortaleza, 2016.
Área de Concentração: Ciência da Computação.
Orientação: Prof. Dr. Francisco Heron de Carvalho Junior.

1. Fluxo de trabalho. 2. Framework (Arquivo de computador). 3. Computação de alto desempenho. I. Título.

CDD 005

JEFFERSON DE CARVALHO SILVA

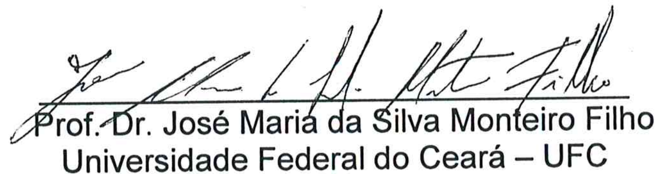
Um Arcabouço para a Construção de Aplicações Baseadas em Componentes sobre uma Plataforma de Nuvem Computacional para Serviços de Computação de Alto Desempenho.

Tese submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação, da Universidade Federal do Ceará, como requisito para a obtenção do grau de Doutor em Ciência da Computação.

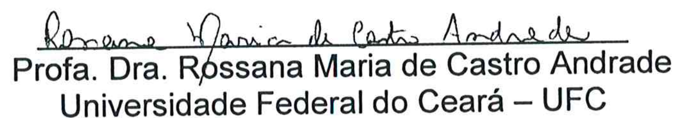
Aprovada em 29 de fevereiro de 2016.

BANCA EXAMINADORA

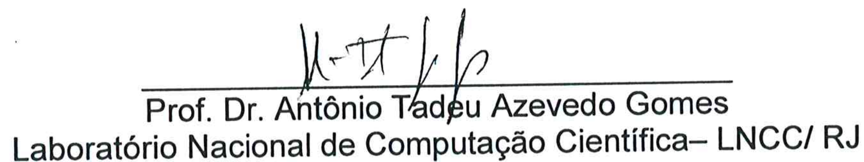
Prof. Dr. Francisco Heron de Carvalho Junior
Presidente
Universidade Federal do Ceará – UFC



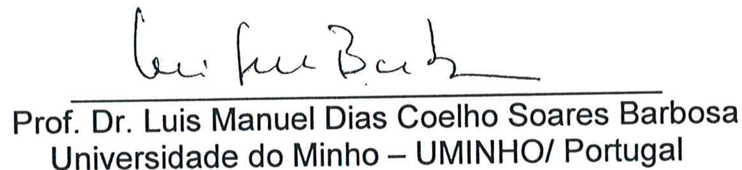
Prof. Dr. José Maria da Silva Monteiro Filho
Universidade Federal do Ceará – UFC



Profa. Dra. Rossana Maria de Castro Andrade
Universidade Federal do Ceará – UFC



Prof. Dr. Antônio Tadeu Azevedo Gomes
Laboratório Nacional de Computação Científica– LNCC/ RJ



Prof. Dr. Luis Manuel Dias Coelho Soares Barbosa
Universidade do Minho – UMINHO/ Portugal

Resumo

Desenvolver aplicações de Computação de Alto Desempenho (CAD), que acessem os recursos computacionais disponíveis de forma otimizada e em um nível maior de abstração, é um desafio para cientistas de diversos domínios. Esta Tese apresenta a proposta de uma nuvem de componentes chamada HPC Shelf, pano de fundo onde as aplicações CAD executam, e o arcabouço SAFe, *Front-End* para criação de aplicações na HPC Shelf e contribuição principal do autor.

O SAFe toma como base o projeto de sistemas gerenciadores de *workflows* científicos (SGWC), permitindo a implementação de soluções computacionais baseadas em componentes para resolver os problemas especificados por meio de uma interface de nível de abstração mais alto. Para isso, foi desenvolvido o SAFeSWL, uma linguagem de descrição arquitetural e orquestração de *worflows* científicos.

Comparado com outros SGWC, além de livrar usuários finais de preocupações em relação à construção de soluções computacionais paralelas e eficientes a partir dos componentes oferecidos pela nuvem, o SAFe faz uso de um sistema de contratos contextuais integrado a um sistema de descoberta (resolução) dinâmica de componentes. A linguagem SAFeSWL permite o controle explícito das etapas do ciclo de vida de um componente em execução (resolução, implantação, instanciação e execução), através de operadores embutidos, a fim de otimizar o uso dos recursos da nuvem e minimizar os custos de sua utilização.

Montage e Map/Reduce constituem os estudos de caso aplicados para demonstração e avaliação das propriedades originais do SAFe e do SAFeSWL na construção de aplicações de CAD.

Abstract

Developing High Performance Computing applications (HPC), which optimally access the available computing resources in a higher level of abstraction, is a challenge for many scientists. To address this problem, we present a proposal of a component computing cloud called HPC Shelf, where HPC applications perform and SAFe framework, a front-end aimed to create applications in HPC Shelf and the author's main contribution.

SAFe is based on Scientific Workflows Management Systems (SWMS) projects and it allows the specification of computational solutions formed by components to solve problems specified by the expert user through a high level interface. For that purpose, it implements SAFeSWL, an architectural and orchestration description language for describing scientific worflows.

Compared with other SWMS alternatives, besides rid expert users from concerns about the construction of parallel and efficient computational solutions from the components offered by the cloud, SAFe integrates itself to a system of contextual contracts which is aligned to a system of dynamic discovery (resolution) of components. In addition, SAFeSWL allows explicit control of life cycle stages (resolution, deployment, instantiation and execution) of components through embedded operators, aimed at optimizing the use of cloud resources and minimize the overall execution cost of computational solutions (workflows).

Montage and Map/Reduce are the case studies that have been applied for demonstration, evaluation and validation of the particular features of SAFe in building HPC applications aimed to the HPC Shelf platform.

Dedicatória

À minha mãe, Jacqueline Macêdo de Carvalho Silva
e ao meu pai, Francisco Jales da Silva.

Agradecimentos

Em primeiro lugar gostaria de agradecer ao meu orientador, Francisco Heron de Carvalho Junior, por me guiar na área acadêmica desde o mestrado, por sua dedicação na orientação desta Tese e por ser um exemplo de professor, pesquisador e pessoa para mim.

Agradeço também a banca examinadora por aceitar o convite desta defesa e por todas as valiosas contribuições feitas durante o doutorado.

Aos amigos do grupo de pesquisa ParGO/CAD, pelas discussões e apoio que certamente foram imprescindíveis para a realização deste trabalho.

Ao apoio e amizade dos colegas e profissionais do Campus da Universidade Federal do Ceará em Quixadá e do programa de Mestrado e Doutorado em Ciência da Computação - MDCC.

Por último, agradeço em especial os meus pais Jacqueline Macêdo e Francisco Jales, meus irmãos Thomas e Luciana e a minha esposa Rainara, pela paciência que tanto precisei durante incontáveis momentos que tive que me ausentar e por sempre expressarem palavras de incentivo e motivação para mim.

Sumário

Lista de Figuras **ix**

1 Introdução **1**
1.1 Computação de Alto Desempenho Baseada em Componentes (CBHPC) 3
1.2 Computação de Alto Desempenho em Nuvens 7
1.3 HPC Shelf, Uma Nuvem de Componentes para Aplicações de Computação de Alto Desempenho 10
1.3.1 *Workflows* Científicos: Base para as Aplicações na HPC Shelf . 12
1.4 Contribuição à HPC Shelf: *Shelf Application Framework* 13
1.5 Revisitando os Objetivos da Proposta de Tese 14
1.6 Contribuições e Resultados desta Tese 15
1.7 Estrutura do Documento 18

2 A Nuvem HPC Shelf **20**
2.1 O Modelo de Componentes Hash 22
2.2 Espécies de Componentes da HPC Shelf 26
2.2.1 Portas de Componentes da HPC Shelf 27
2.2.2 Fontes de Dados 29
2.3 Atores Envolvidos na HPC Shelf 30
2.3.1 Usuários Especialistas 30
2.3.2 Provedor de Aplicações 31
2.3.3 Desenvolvedor de Componentes 31
2.3.4 Mantenedor de Plataformas 32
2.4 Relacionamento entre Atores 33
2.5 Visão Arquitetural da HPC Shelf 33
2.5.1 O *Front-End* 33
2.5.2 O *Core* 35
2.5.3 O *Back-End* 35
2.5.4 Visão Arquitetural 36
2.6 Contratos Contextuais 38
2.7 Considerações Finais 40

3 SAFe: O *Framework* de Aplicações da HPC Shelf **41**
3.1 Características e Requisitos do SAFe 42
3.2 Sistemas Computacionais SAFe 45
3.3 A Arquitetura da Implementação do SAFe 46
3.3.1 O Pacote `safe-framework` 47
3.3.2 O Pacote `safe-language` 55
3.4 A Linguagem SAFeSWL 57
3.4.1 O Subconjunto Arquitetural 58
3.4.2 O Processamento de uma Especificação Arquitetural 59
3.4.3 Contratos Contextuais 61
3.4.4 O Subconjunto de Orquestração 61
3.4.5 O Processamento de um Código de Orquestração de SAFeSWL 63
3.5 Ciclo de Vida de uma Aplicação 65
3.5.1 Fase de Desenvolvimento 65
3.5.2 Fase de Execução 67
3.6 Um Protótipo de Aplicação 68
3.7 Considerações Finais 84

4 Estudos de Caso **86**
4.1 Estudo de Caso 1: Montage 86
4.1.1 Visão Geral dos Componentes do Montage 87
4.1.2 *Workflows* do Montage 89
4.1.3 Implementação da Aplicação SAFe do Montage 89
4.1.4 O *Workflow* M101 91
4.1.5 Conexão do M101 com o Componente *Workflow* 99
4.1.6 Criação do M101 na Aplicação MoEx 100
4.1.7 Execução do M101 na Aplicação MoEx 103
4.1.8 O *Workflow* Plêiades 105
4.1.9 Criação dos *Workflows* de Projeção do Plêiades na Aplicação MoEx 107
4.1.10 Execução do Plêiades na Aplicação MoEx 109
4.1.11 Sobrecarga do SAFe sobre o M101 e Plêiades 110
4.2 Estudo de Caso 2: Processamento Map/Reduce 111
4.2.1 Componentes HPC Shelf para o Map/Reduce 113
4.2.2 Arquitetura de um *Workflow* de Processamento Map/Reduce: Contador de Palavras 117
4.2.3 A Interface da Aplicação Map/Reduce 123
4.2.4 A Execução da Aplicação Map/Reduce 128
4.3 Considerações Finais 129

5 Trabalhos Relacionados **132**
5.1 ASKALON 132
5.2 BPEL Sedna 135
5.3 Kepler 137
5.4 OSC 138
5.5 Pegasus 140

5.6 Taverna . . . 142
5.7 Triana . . . 144
5.8 Considerações Finais . . . 145

6 Conclusões e Perspectivas de Trabalhos Futuros 150
6.1 Trabalhos Futuros . . . 153
6.2 Considerações Finais . . . 156

Referências Bibliográficas 164

Apêndice A Linguagem Arquitetural 165

Apêndice B Linguagem de Orquestração 172

Lista de Figuras

1.1 Visão geral da relação entre o SAFe e os demais elementos arquiteturais da HPC Shelf. 18

2.1 Composição por sobreposição (perspectiva hierárquica). 23
2.2 Composição por sobreposição (perspectiva UML). 24
2.3 Visão em alto nível dos módulos da HPC Shelf, com destaque em pontilhado da contribuição desta Tese. 36

3.1 Arquitetura de um sistema computacional para o Map/Reduce no SAFe. 46
3.2 Visão Arquitetural do SAFe. 47
3.3 Componentes CCA. 48
3.4 Relacionamento entre as Classes do Pacote `safe-framework`. 50
3.5 A criação de sessões por parte do Framework de uma Aplicação. Uma mesma aplicação pode fazer parte de várias sessões, cada uma com um único Workflow a qual orquestra um conjunto de componentes. . 54
3.6 Relacionamento, em alto nível, entre classes do pacote de linguagem. 56
3.7 A Linguagem SAFeSWL e seus subconjuntos. 57
3.8 Visão abstrata da linguagem arquitetural do SAFeSWL. 58
3.9 Classes da linguagem arquitetural, contidas no pacote `safe-language`. 60
3.10 Sintaxe abstrata da linguagem de orquestração do SAFeSWL 62
3.11 Classes da linguagem execução, contidas no pacote `safe-language`. 64
3.12 Diagrama da fase de execução da aplicação. 69
3.13 Componentes Cliente e Servidor 70
3.14 Diagrama do Fluxo de Orquestração da Aplicação de Exemplo 74
3.15 Componentes Cliente, Fila, Servidor, Aplicação e *Workflow*. 75
3.16 Faceta Servidora de um *Binding*, implementado como um Serviço *Web*. 76
3.17 Faceta Cliente de um *Binding*, Implementado como um Serviço *Web*. 76
3.18 Componente *Proxy* para o Componente Cliente 81
3.19 Trecho de Código do Componente Aplicação SAFe 82
3.20 Interface Simples, para a Aplicação de Exemplo 84
3.21 Exemplo de Saída 84

4.1 Relacionamento entre o componente Montage encapsulado a aplicação do lado do especialista. 88
4.2 Exemplo de um workflow em Montage, mostrando a paralelização de seus componentes. 88
4.3 Interface gráfica para geração de workflows Montage, MoEx 91
4.4 Visão simples do fluxo do workflow M101, com a imagem resultante da galáxia ao centro. 93
4.5 Visão geral do workflow M101 sem as portas de tarefas. 94
4.6 O componente workflow conectado às portas de tarefas dos outros componentes do Montage. 99
4.7 Processo de criação do workflow M101 através da aplicação MoEx. . . 100
4.8 Componentes repositórios e suas portas provedoras. 106
4.9 Os três fluxos para a execução do *workflow* Plêiades. No final as imagens em cada faixa de cor são combinadas pelo componente `mJpeg`. 106
4.10 Visão geral do Plêiades sem as portas de tarefas. 107
4.11 Componente *workflow* e suas conexões com os componentes do Pleiades em uma *workflow* de projeção. 108
4.12 Código *shell script* para o M101 e Plêiades. 110
4.13 Exemplo clássico de contagem de palavras do Map/Reduce. 112
4.14 Arquitetura Map/Reduce com as portas de tarefas. 118
4.15 Interface para criação de workflows Map/Reduce. 124

A.1 *Tipos disponíveis para a montagem de uma arquitetura em XML.* . . 165
A.2 *A estrutura de uma tag* `<architecture>` 165
A.3 *A estrutura de uma tag* `<body>` 166

B.1 *Os tipos disponíveis de workflow executável.* 172
B.2 *A estrutura da elemento* `XML_SAFe_Prim_Oper` (elemento principal), implementado de acordo com a gramática abstrata. 173

Capítulo 1

Introdução

Na última década, a computação continuou a testemunhar grandes avanços na disponibilização de recursos de *hardware* voltados a aplicações com requisitos severos de capacidade de processamento, especialmente alavancados com o surgimento de processadores com múltiplos núcleos de processamento (*multi-core*) e aceleradores computacionais munidos de alguns milhares de núcleos de processamento (*many-core*). Somado a isso, tem-se observado o avanço da capacidade de integrar tais recursos em sistemas de computação de escala cada vez maior, tais como servidores equipados com vários processadores de múltiplos núcleos e aceleradores integrados, os quais podem ser, com relativa facilidade e baixo custo, integrados em plataformas de computação paralela denominadas *clusters*. Por sua vez, *clusters* podem ser integrados entre si através de tecnologias consagradas pela efervescente pesquisa em grades computacionais empreendida nos anos 2000 e, mais recentemente, oferecidas sob a perspectiva de serviços de infraestrutura sob demanda através de nuvens computacionais.

Os avanços do *hardware* permitiram o rápido surgimento de plataformas de computação paralela com desempenho na ordem dos Petaflops ainda antes de 2010, segundo o *benchmarking* utilizado no ranque Top 500[1], mantido por instituições acadêmicas e indústria. Atualmente, os limites inferior e superior desse ranque são de 206 Teraflops e 35 Petaflops, respectivamente. O desafio atual é a construção, antes de 2020, de plataformas de *Exascale Computing*, cujo desempenho deve atingir a ordem dos Exaflops. Porém, através da integração de recursos em grades e nuvens, tal escala de desempenho já é possível em computação de larga escala, integrando recursos de diversos centros através de serviços de computação de alto desempenho.

[1] http://www.top500.org

Entretanto, explorar o desempenho de plataformas de computação paralela de larga escala de caráter heterogêneo (*multi-core*, *many-core* e apresentando hierarquia profunda de memória e paralelismo), distribuídas e integradas sobre a infraestrutura da *internet*, continua a ser um desafio para cientistas e engenheiros de *software*, apesar dos esforços empreendidos ao longo das décadas com linguagens e vários tipos de ferramentas de suporte ao desenvolvimento de programas paralelos, incluindo *frameworks* de programação e ambientes de solução de problemas de alto nível.

Aplicações de Computação de Alto Desempenho (CAD) caracterizam-se pelo alto volume de operações que precisam realizar, possivelmente sobre um grande volume de dados. Para que a resolução de problemas ocorra em tempo viável, essas aplicações devem utilizar algoritmos com técnicas de paralelismo adequadas para cada tipo de problema, de acordo com a arquitetura da plataforma de computação paralela onde devem executar. Tais técnicas influenciam no tempo gasto para o desenvolvimento da solução e muitas vezes não são de conhecimento dos especialistas, interessados unicamente nos resultados produzidos pelas soluções, os quais estão mais preocupados com a lógica do negócio do que com questões de programação e configuração de plataformas de execução.

Nesse contexto, torna-se clara a necessidade de abstrações para o desenvolvimento de *software* que agilizem o tempo de desenvolvimento da *lógica de negócio* de aplicações com requisitos de CAD que sejam capazes de explorar o desempenho potencial de plataformas de computação de alto desempenho de larga escala. A forma como o código é paralelizado e executado torna-se o menos importante do ponto de vista do usuário da aplicação. No entanto, conciliar **modularidade**, **eficiência**, **abstração**, **interoperabilidade** e **generalidade** em computação paralela é reconhecido como um grande desafio, descrito na literatura [Steen 2006, Carvalho Junior et al. 2007, Bernholdt, Nieplocha e Sadayappan 2004, Post e Votta 2005], apesar dos esforços empreendidos e resultados alcançados nas últimas décadas pelos pesquisadores da academia e indústria.

Aplicações de CAD, muitas vezes, situam-se em ambientes extremamente heterogêneos, com uma grande variedade de linguagens, arquiteturas e sistemas operacionais. O desenvolvimento de ferramentas que auxiliem a construção de aplicações paralelas, aumentando a produtividade e diminuindo o tempo gasto com a implementação do código paralelo, faz-se estritamente necessário.

A programação orientada a componentes, por exemplo, é uma das técnicas disponíveis na engenharia de *software* a qual facilita a criação de sistemas complexos,

a partir de módulos independentes. O trabalho de pesquisa cujos resultados são apresentados nesta Tese situa-se dentro do contexto de uma tentativa de convergência de tecnologias de componentes de software e nuvens computacionais a fim de atender as necessidades de aplicações com requisitos de CAD, voltadas à execução de processamento em plataformas de computação paralela de larga escala, implementado através de uma plataforma proposta, chamada HPC Shelf.

A HPC Shelf, a qual será descrita no Capítulo 2 desta Tese, constitui contribuição de um esforço conjunto entre membros do grupo de pesquisa em Computação de Alto Desempenho ligado ao programa de Mestrado e Doutorado em Ciência da Computação (MDCC) da Universidade Federal do Ceará (UFC), dentre alunos de mestrado e doutorado, incluindo o autor desta Tese. Em particular, esta Tese propõe um arcabouço para o desenvolvimento de aplicações que façam uso dos recursos disponibilizados pela HPC Shelf, denominado SAFe (*Shelf Application Framework*), cujas contribuições são delineadas ao longo deste documento.

As seções que se seguem apresentam uma melhor contextualização das áreas de desenvolvimento baseado em componentes aplicados à computação de alto desempenho e nuvens computacionais, bem como descreve melhor as motivações deste trabalho, seus objetivos, principais contribuições da Tese e estrutura deste documento que a apresenta.

1.1 Computação de Alto Desempenho Baseada em Componentes (CBHPC)

Componentes são módulos de computação independentes, com interface bem definida, interoperáveis e auto-executáveis [Wang 2005, Szyperski 2000]. Inicialmente, foram usados na área comercial devido a vantagens tais como: reuso de código; facilidade de composição, gerando novos componentes a partir de outros; interfaces de serviços, o que aumenta a coesão dos módulos da aplicação e diminui o acoplamento entre eles e, finalmente, o baixo custo de desenvolvimento e evolução do *software* (adição de novas funcionalidades).

Nos ramos das ciências computacionais e na computação científica, alguns modelos, explicados adiante, foram propostos para a implementação de componentes que satisfaçam as necessidades das aplicações de CAD. Ao longo dos anos 2000, esse interesse levou a caracterização de um ramo de pesquisa em CAD que ficou conhecido como Computação de Alto Desempenho Baseado em Componentes (CBHPC[2]) A

[2] *Component-Based High Performance Computing*

Nome	Modelo	Referência e/ou URL
CCA	CCAffeine	[Allan et al. 2002]
	XCAT	[Krishnan e Gannon 2004], http://www.extreme.indiana.edu/xcat
	DCA	[Bertrand e Bramley 2004]
	SciRun2	[Zhang et al. 2004]
	SciJump	[Parker et al. 2010]
	MOCCA	[Malawski, Kurzyniec e Sunderam 2005]
Fractal	Julia	http://fractal.ow2.org/julia
	Cecilia	http://fractal.ow2.org/cecilia-site/current
	Think	http://think.ow2.org
	AOKell	http://fractal.ow2.org/aokell/
GCM	ProActive	[Amedro et al. 2010], http://proactive.activeeon.com
Hash	HPE	http://hash-programming-environment.googlecode.com

Tabela 1.1: Plataforma de Componentes para Computação de Alto Desempenho

Tabela 1.1 apresenta um resumo de implementações de sistemas CBHPC baseados nos principais modelos de componentes destinados aos requisitos de aplicações CAD.

O CCA (*Common Componente Architecture*) foi desenvolvido por um grupo de pesquisadores envolvidos em um projeto da iniciativa SciDAC (*Scientific Discovery Through Advanced Computing*)[3] do Departamento de Energia (DoE) do governo dos EUA. O modelo CCA [Armstrong et al. 1999, Armstrong et al. 2006], inspirado no CORBA, é considerado o padrão de maior impacto. Componentes CCA comunicam-se por meio de portas provedoras e usuárias, as quais são ligadas de forma direta quando os componentes residem no mesmo espaço de endereçamento a fim de reduzir o tempo de comunicação (troca de dados) entre os mesmos.

A plataforma de componentes (*framework*) de referência do modelo CCA chama-se CCAffeine [Allan et al. 2002], a qual usa a ferramenta Babel/SIDL para conexão entre as portas de componentes [Epperly et al. 2012] e introduz a idéia de regimento de componentes (*cohort*) para suporte ao paralelismo de memória compartilhada. Entretanto, outros *frameworks* CCA foram implementados ou derivados a partir de *frameworks* computacionais existentes [Krishnan e Gannon 2004, Zhang et al. 2004, Bertrand e Bramley 2004, Malawski, Kurzyniec e Sunderam 2005], explorando diferentes requisitos, tais como paralelismo, distribuição, reconfiguração dinâmica, etc. Destacam-se também as iniciativas dos integrantes do Fórum CCA em estudar formas de exploração do paralelismo em modelos de

[3]http://www.scidac.gov/

componentes [MCMD-WG Wiki at CCA Forum,Forum 2009].

Por sua vez, o modelo de componentes Fractal [Baude, Caromel e Morel 2003], desenvolvido no contexto do consórcio CoreGrid, financiando pela União Européia (UE), implementa abstrações para lidar com a separação entre interesses funcionais e não-funcionais em componentes. Para isso, os componentes possuem uma *membrana* e um *conteúdo*, sendo a membrana a parte do componente responsável pelo tratamento de interesses não-funcionais, através de *controladores*. Os componentes em Fractal também podem ser aninhados, ou seja, formados por vários componentes que podem ser compartilhados com outros, o que motiva a sua denominação.

Fractal é especificado como um conjunto de interfaces padrões, disponíveis em Java, C e OMG IDL [Blair, Coupaye e Stefani 2009]. Implementações devem aderir a níveis de conformidade, identificados por números naturais. No nível 0, componentes Fractal são simples objetos. Em cada um dos níveis subsequentes (0.1, 1.0, 2.1, 3.1, 3.2 e 3.3), é requisitada a implementação de um conjunto de interfaces e/ou o suporte a um novo conjunto de *controladores*. A Tabela 1.1 apresenta as principais plataformas de componentes que implementam o modelo Fractal.

O modelo GCM (*Grid Component Model*) [Baude 2012], é um modelo de componentes projetado como uma extensão do modelo Fractal. Ele tem como objetivo auxiliar no desenvolvimento de aplicações de larga escala voltadas a infraestruturas de grades computacionais, que são caracterizadas pela sua heterogeneidade e uso compartilhado de recursos. GCM também é um modelo de componentes hierárquicos, com suporte à comunicação através de portas de comunicação coletivas. Tal dinamicidade torna possível a implementação de todos os tipos de padrões de interação coletiva, derivados do uso de componentes compostos. O modelo GCM foi implementado como uma extensão da plataforma ProActive, a fim de atender, nessa plataforma, requisitos de integração com tecnologias de grades computacionais.

O modelo Hash [Carvalho Junior e Lins 2005] foi desenvolvido para lidar com o aspecto de paralelismo em modelos de componentes. Para isso, um componente é composto de unidades, distribuídas sobre os nós de processamento de um computador paralelo de memória distribuída. Componentes do modelo Hash podem ser combinados em componentes maiores e aplicações através de uma forma de composição hierárquica denominada *composição por sobreposição* [Carvalho Junior e Lins 2008]. Além disso, componentes podem ser distinguidos em *espécies de componentes*, com diferentes modelos de implantação e comunicação em uma mesma

plataforma de componentes.

A implementação de referência do modelo Hash é o HPE (*Hash Programming Environment*) [Carvalho Junior e Rezende 2013], o qual também foi refatorado posteriormente para compatibilidade com o CCA [Carvalho Junior e Corrêa 2010] e ao mesmo tempo propôr uma noção geral de componente CCA paralelo. O objetivo do HPE é prover um ambiente que facilite a implementação de aplicações paralelas para executarem sobre *clusters*, construídas pela composição hierárquica de um conjunto de componentes. Para o especialista de um domínio, o qual não tem muito conhecimento (ou nenhum) em desenvolvimento de programas paralelos, o trabalho resume-se a conectar os componentes, repassando os dados de entrada e escolhendo a plataforma de execução.

Componentes também podem ser implementados como *serviços remotos*, disponibilizando suas interfaces para outros componentes a fim de formar aplicações complexas. Em um ambiente distribuído, essas interfaces devem trocar dados obedecendo a um certo protocolo de comunicação. Um exemplo é uso de tecnologias como Serviços *Web*, onde um componente pode ser criado em qualquer linguagem convencional e disponibilizado na *internet* (ou em uma rede interna). O cliente que "consome" os serviços desse componente necessita saber apenas sua localização e descrição (geralmente em uma linguagem genérica como XML), de suas operações públicas. Tais conceitos combinam facilmente com o ambiente distribuído da *internet*, onde a transparência de acesso é explorada. Nesse cenário, componentes intercambiáveis e a rede mundial de computadores, aliados a definição de protocolos de comunicação robustos, deu origem a "computação de serviços" (*Services Computing*)[4]. Fortemente baseado em SOA (*Services Oriented Architecture*), seu objetivo é disponibilizar serviços de tecnologia para executar tarefas da forma mais eficiente possível.

A computação de serviços ganhou ainda mais força na última década, com o barateamento do *hardware*, aumento da velocidade da *internet* e a melhoria das técnicas de Engenharia de *Software*, com o surgimento de bibliotecas e linguagens de programação naturalmente implementadas para desenvolvimento *web*. A indústria percebeu então que disponibilizar recursos de *hardware*, sistemas operacionais e plataformas de programação como um serviço semelhante ao oferecimento de energia elétrica tornaria-se uma tendência. Daí nasceram soluções de *software* com formas transparentes de acesso a esses recursos. A ideia é fazer com que o usuário pague

[4]http://tab.computer.org/tcsc/

apenas por aquilo que usar (mais uma vez, da mesma forma que os serviços de abastecimento de água e energia). Isso motivou o conceito de Computação em Nuvens (*Cloud Computing*), onde os serviços, sejam eles de *hardware* e *software*, são vistos como uma "nuvem" de funcionalidades, acessada transparentemente pelo cliente. Nuvens podem disponibilizar componentes como serviços de acesso remoto, inclusive serviços implementados por componentes de alto desempenho. A seguir, veremos como a tecnologia de componentes de alto desempenho apresentada pode ser usada no desenvolvimento de aplicações que executem em nuvens computacionais.

1.2 Computação de Alto Desempenho em Nuvens

Nuvens computacionais surgiram como uma solução de baixo custo, confiável e com a finalidade de executar os mais diversos tipos de aplicações, desde as comerciais até científicas. As nuvens trabalham com o conceito de *utility computing* o qual é oferecido um serviço e o cliente paga pelo tempo de processamento e uso de recursos.

O NIST (*National Institute of Standards and Technology*)[5] é um instituto do departamento de comércio do governo dos EUA o qual trabalha em sociedade com a indústria a fim de desenvolver tecnologia aplicada, métricas e padrões. Segundo o NIST, nuvens computacionais constituem um modelo de computação capaz de propiciar o acesso ubíquo, conveniente e sob demanda a um conjunto de recursos computacionais (processamento, armazenamento, aplicações e serviços, por exemplo), através de um rede dedicada. Os recursos podem ser liberados ou alocados com relativa facilidade, através de um provedor de serviços. O documento de padronização do NIST [Mell e Grance 2011] ainda discrimina três modelos de serviços oferecidos pelas nuvens computacionais. A seguir, um breve resumo desses modelos e suas principais funcionalidades:

- **SaaS - *Software as a Service***: nesse modelo, o cliente tem acesso a uma gama de aplicações que são oferecidas através de diversos tipos de interface, podendo ser acessadas por dispositivos móveis ou navegadores *web*. As aplicações executam sobre uma infraestrutura de nuvem transparente ao usuário e tem como missão estarem sempre "disponíveis".

- **PaaS - *Platform as a Service***: nesse modelo, o cliente tem a possibilidade de criar ou adquirir aplicações e implantá-las na nuvem computacional. Através do uso de linguagens de programação adequadas, bibliotecas e serviços,

[5]http://www.nist.gov/

o próprio consumidor projeta aplicações e ainda pode monitorar o processo de implantação e configuração.

- **IaaS - *Infrastructure as a Service***: nesse modelo, é oferecido ao cliente uma gama de recursos computacionais básicos como rede de alta velocidade, armazenamento, memória e processamento (gráfico inclusive). O objetivo do consumidor é o de implantar sobre a infraestrutura "alugada" *softwares* elementares, como sistemas operacionais ou aplicações criadas por ele mesmo. Todo o serviço de manutenção do *hardware* é unicamente de responsabilidade do provedor.

No ambiente comercial, podemos citar diversos exemplos de nuvens computacionais bem sucedidas, as quais cobram taxas de acordo com o serviço oferecido; A *Amazon Elastic Compute Cloud*, ou simplesmente *Amazon EC2*[6], é um Serviço *Web* que fornece funcionalidades de computação redimensionável (elasticidade) em um ambiente de nuvem, através de uma interface simples. Por sua vez, a *Windows Azure*[7] é uma plataforma de nuvem aberta e flexível que possibilita a rápida construção, implantação e gerenciamento de aplicações sobre uma rede de *datacenters Microsoft*. Ela é aberta no sentido de permitir qualquer linguagem, estrutura ou ferramenta para a criação de aplicativos; *Google App Engine*[8] é uma plataforma de serviço de nuvens para o desenvolvimento e hospedagem de aplicações *web* em *datacenters* gerenciados pelo Google.

Estão disponíveis também ferramentas de código aberto para implementação de nuvens. Dentre elas podemos citar o *OpenNebula*[9] e *OpenStack*[10]. O *OpenNebula* é um *toolkit* de código fonte livre e aberto para gerenciar *datacenters* heterogêneos e orquestra tecnologias de armazenamento, segurança, virtualização, monitoramento e rede para implantar serviços em infraestruturas distribuídas. Ele ainda oferece uma interface gráfica simples e fácil de configurar máquinas virtuais. O *OpenStack* é um esforço conjunto de desenvolvedores e especialistas em nuvens computacionais com o intuito de produzir uma plataforma aberta de nuvem privada e pública. O ambiente consiste de uma série de projetos relacionados entre si, os quais fornecem componentes adequados para implementação de uma solução de infraestrutura

[6]http://aws.amazon.com/
[7]http://www.windowsazure.com/
[8]https://developers.google.com/appengine/
[9]http://opennebula.org/
[10]http://www.openstack.org/

baseada em nuvem computacional. Sua arquitetura modular é constituída de sete componentes distintos, responsáveis em gerenciar elasticidade, armazenamento, rede, virtualização, interface gráfica, segurança e imagens.

De acordo com [Khalidi 2011], uma plataforma de nuvem que vise melhorar o tempo de desenvolvimento, aumentando a produtividade, deve prover um rico conjunto de serviços, os quais aceleram o trabalho do programador, não tomando tempo com detalhes de configuração e operações de baixo nível. O elemento essencial é a automatização de boa porte do processo de desenvolvimento.

Com o sucesso das nuvens comerciais, a tecnologia de nuvens computacionais rapidamente despertou o interesse de usuários, pesquisadores e provedores de serviços de computação de alto desempenho, sobretudo aqueles outrora interessados e envolvidos em projetos na área de grades computacionais. As iniciativas de pesquisa em nuvens de computação de alto desempenho variam, quanto ao seu modelo de serviço, desde o modelo SaaS, com o oferecimento de aplicações com interfaces de alto nível via navegadores *web* para execução de soluções computacionalmente intensivas de problemas em domínios específicos das ciências e engenharias (e.g. [Vecchiola, Pandey e Buyya 2009, Niehorster et al. 2010, Church et al. 2012]), e IaaS, através de provedores de infraestruturas de computação de alto desempenho, ou seja, conjuntos de plataformas de computação paralela, públicos ou privados, que vislumbraram o oferecimento de infraestrutura para execução direta de programas paralelos e/ou computações de outras nuvens SaaS, sendo de especial interesse de cientistas computacionais e engenheiros, tanto da academia quanto da indústria (e.g. [Rehr et al. 2010, Zaspel e Griebel 2011, Montero, Vozmediano e Llorente 2011]). No meio termo, há ainda iniciativas PaaS, onde podemos destacar o Aneka, solução comercial com origem acadêmica mantida pela empresa Manjrasoft, baseada na plataforma .NET [Sukumar, Vecchiola e Buyya 2010].

Além das iniciativas de pesquisa, observou-se nos últimos anos a proliferação de provedores comerciais e públicos de serviços de CAD em nuvens, atendendo a usuários tanto com interesses acadêmicos quanto industriais. Um conjunto representativo de provedores é apresentado na Tabela 1.2, a qual aponta ainda os modelos de nuvens adotados por cada um.

Nome	url	SaaS	IaaS	PaaS
Nimbix	http://www.nimbix.net/cloud-supercomputing	•	•	
CST[11]	http://www.cst.com	•		
Bull Extreme Factory	http://www.extremefactory.com	•		•
SGI Cyclone	http://www.sgi.com/products/hpc_cloud/cyclone	•	•	
Amazon EC2	http://aws.amazon.com/pt/ec2		•	
Sabalcore	http://www.sabalcore.com		•	
Peguin Computing	http://www.penguincomputing.com	•	•	
Softlayer	http://www.softlayer.com		•	
Peer1 Hosting	http://www.peer1.com/cloud-hosting/hpc-cloud		•	
R-HPC	http://www.r-hpc.com		•	
Univa	http://www.univa.com			•
Gompute	http://www.gompute.com		•	
Aneka	http://www.manjrasoft.com/products.html			•

Tabela 1.2: Alguns Provedores de Computação de Alto Desempenho em Nuvens

1.3 HPC Shelf, Uma Nuvem de Componentes para Aplicações de Computação de Alto Desempenho

Vistas de maneira isolada, nuvens computacionais e plataformas de componentes constituem tecnologias promissoras para alavancar a engenharia de *software* voltada a aplicações de CAD.

Enquanto nuvens permitem o acesso transparente dessas aplicações sobre plataformas de computação paralela de alta tecnologia, as quais, como argumentado anteriormente, apresentam um crescente grau de complexidade arquitetural e de heterogeneidade, componentes oferecem o grau de modularidade e abstração necessário para lidar com a construção de soluções em larga escala, propiciando a integração e reuso de contribuições entre cientistas computacionais e engenheiros, possivelmente de forma inter e multidisciplinar.

De forma integrada, nuvens e componentes permitem a usuários e provedores de aplicações abstrair-se dos interesses de paralelização na construção de soluções de *software* para aplicações com requisitos de CAD. Tais aplicações, como já discutido, exigem um alto grau de especialização para explorar o desempenho de plataformas de computação paralela de alta tecnologia, porém ainda de tal forma que os componentes que as constituem sejam construídos de maneira a explorar as peculiaridades arquiteturais de cada uma dessas plataformas, buscando um melhor aproveitamento dos seus recursos de processamento, bem como minimizando os custos energético e de acesso.

Componentes, por serem módulos independentes, com interfaces bem definidas, podem ser facilmente inseridos em ambientes distribuídos. Desenvolvedores, então,

criam bibliotecas de componentes específicas para determinados domínios e as aplicações apenas necessitam ter conhecimento de suas interfaces. Essas interfaces podem ser implementadas como um conjunto de serviços os quais servem às mais diversas finalidades. Ambientes de nuvens computacionais tornam transparente o acesso a ambientes distribuídos através de seus serviços. Essa facilidade de acesso tornou-se extremamente popular no meio industrial e, com o tempo, ganhou a atenção da comunidade científica.

A motivação por trás do projeto da plataforma de aplicações CAD denominada HPC Shelf é fazer uso das abstrações do modelo Hash, principalmente do seu sistema de tipos e natureza paralela dos componentes que propõe, para representar recursos que compõem sistemas de computação paralela e oferecê-los como serviços de CAD por meio da tecnologia de nuvens computacionais. Essa tecnologia agrega funcionalidades relativas às plataformas de execução, as quais, uma vez alinhadas com a tecnologia de componentes Hash implementada pelo HPE, permitirá a construção, implantação e execução de aplicações que apresentam requisitos de CAD. Sendo assim, a nuvem HPC Shelf pode ser considerada uma nuvem PaaS, especializada na construção, implantação e execução de aplicações de CAD, as quais podem ser vistas em conjunto como uma formação de nuvens do tipo SaaS[12].

Vista de uma outra perspectiva, a HPC Shelf também pode ser enxergada como uma camada de abstração para um conjunto de infraestruturas de computação paralela distribuídas, vistas como um plataforma de computação única de alcance global sobre as quais são executadas aplicações que demandam computação paralela de larga escala para atender seus requisitos de computação de alto desempenho.

Os atores em torno da plataforma HPC Shelf são divididos em quatro categorias: *especialistas*, interessados em fazer uso das aplicações (usuário final); *provedores de aplicações*, interessados no desenvolvimento e implantação das aplicações para um determinado domínio de problemas; *desenvolvedores de componentes*, interessados em desenvolver componentes que serão usados para construção das aplicações; e *mantenedores de plataformas*, interessados em fornecer infraestruturas de computação paralela para usufruto das aplicações, para as quais os componentes são otimizados para execução.

A coreografia desses atores é realizada em torno de três elementos arquiteturais:

[12] O termo *shelf* vem do inglês "prateleira", remetendo à ideia de prateleira a partir dos quais componentes são escolhidos para construção de aplicações. *Shelf* também designa uma formação de nuvens típica de tempestades, que se movem rapidamente, associada a frentes de rajada. Essa conjunção de analogias (prateleira, nuvens e velocidade) nos levou à denominação HPC Shelf.

o *Front-End*, o *Core* e o *Back-End*. O *Front-End* é representado pelo conjunto de aplicações implementadas sobre a nuvem, incluindo as aplicações usadas por provedores de aplicações, desenvolvedores de componentes e mantenedores de plataformas para usufruir dos serviços que os interessam na nuvem. Por sua vez, o *Core* representa a biblioteca de componentes, de diferentes espécies, inclusive representando as plataformas de computação paralela oferecidas por mantenedores de plataformas. O *Core* implementa um sistema de descoberta e resolução dinâmica de componentes baseada na noção de *contratos contextuais*, chamado Alite. Finalmente, o *Back-End* representa o conjunto de infraestruturas de computação paralela da HPC Shelf, sobre as quais *plataformas virtuais* de computação paralela podem ser instanciadas (componentes da espécie *plataforma*).

Detalhes sobre a plataforma HPC Shelf serão apresentados no Capítulo 2.

1.3.1 *Workflows* Científicos: Base para as Aplicações na HPC Shelf

No que diz respeito às aplicações sobre nuvens computacionais, em computação científica, podemos nos deparar com cenários onde os componentes que formam a aplicação podem necessitar de muito tempo para executar, tanto devido ao tempo que leva a execução de cada etapa de computação em si, bem como a troca de dados entre componentes dependentes entre si, quanto devido ao tempo de espera em filas de submissão de plataformas de computação paralela. Para aplicações CAD de longa duração, uma plataforma deve suportar um modelo automático de submissão e controle da execução de etapas de computação e trocas de dados de uma etapa para outra, além de oferecer meios para a descrição arquitetural de soluções computacionais baseadas em componentes e a especificação do fluxo de orquestração dos componentes que formam a arquitetura da solução. A reconfiguração dinâmica, tanto na arquitetura quanto no fluxo de orquestração, também é um requisito importante, permitindo a adaptação a cenários imprevistos de execução que eventualmente surgem ao longo do tempo de execução.

Para atender a esses requisitos, a tecnologia de *workflows* permite criar um fluxo de execução de ações computacionais oferecidas por componentes, interconectados entre si, com o fim de resolver um problema. *Workflows* compreendem linguagens extensivamente usadas no meio comercial e que, nos últimos anos, testemunharam significante avanço na comunidade científica. Apesar de usados de forma natural pela indústria do *software* comercial, os *workflows* sofreram uma série de adaptações para se adequarem ao ambiente das aplicações científicas.

No âmbito comercial, um *workflow* define uma sequência de tarefas necessárias para gerenciar um negócio ou um processo de engenharia. Essa sequência de tarefas pode ser programada em um *template* e instanciada para problemas específicos. O avanço das técnicas de engenharia de *software* aliada a melhoria das tecnologias de comunicação, sobretudo a *internet*, fez com que simples *workflows* baseados em *scripts* e recursos locais evoluíssem para o desenvolvimento de tarefas baseadas em componentes complexos disponibilizados como Serviços *Web* e serviços de grades computacionais, por exemplo. *Workflows* passaram então a conectar componentes espalhados em uma rede, formando uma grande aplicação.

No âmbito científico, *workflows* podem ser usados para orquestrar aplicações de alto desempenho, de forma relativamente fácil e rápida. Sua adoção promove a programação com reuso e para reuso, onde *workflows* podem ser compostos por componentes CAD e reusados por outras aplicações, alterando-se apenas seus parâmetros de entrada.

Esta Tese propõe um *framework* de *workflows* científicos que atenda aos requisitos das aplicações sobre a plataforma HPC Shelf, denominada SAFe (*Shelf Application Framework*). A abordagem baseada em componentes paralelos torna esta solução mais natural e simples, pois permite lidar com unidades de computação padronizadas e intercambiáveis. *Workflows* científicos constituem parte essencial de uma aplicação, representando soluções computacionais para problemas descritos pelo especialista, as quais orquestram um conjunto de componentes a fim de resolvê-los.

1.4 Contribuição à HPC Shelf: *Shelf Application Framework*

O SAFe é originalmente proposto neste trabalho como o *framework* usado por *provedores* para o desenvolvimento de aplicações da HPC Shelf. Através de uma aplicação, o *especialista* é capaz de utilizar uma interface de alto nível de abstração para descrever problemas dentro de um certo domínio de interesse, cujas soluções computacionais são intensivas em computação, justificando a sua execução em uma plataforma de computação paralela de larga escala.

Soluções computacionais para problemas descritos pelo especialista podem ser construídas, usando SAFe, pela composição de componentes disponíveis no catálogo do *Core* da HPC Shelf. Tais soluções constituem sistemas computacionais baseados em componentes paralelos, descritos através de uma linguagem de descrição de *workflows* científicos denominada SAFeSWL (*SAFe Scientific Workflow Language*).

O SAFe compreende uma API (*Application Program Interface*) de classes Java

que permitem a criação de aplicações baseadas em componentes paralelos que executam em plataformas de computação paralela remotas, comunicando-se entre si e com a própria aplicação, interativamente, durante o seu ciclo de vida. A linguagem SAFeSWL encontra-se implementada em XML, com uma especificação contida em uma gramática XSD.

Uma importante peculiaridade de SAFeSWL é sua divisão em dois subconjuntos ortogonais, usados para descrever, em arquivos distintos, a arquitetura do sistema computacional, especificando quais os componentes que serão instanciados e como serão ligados entre si através de portas de ambiente e de tarefas, bem como o fluxo de orquestração desses componentes. Na execução do *workflow*, componentes podem ser explicitamente resolvidos (descoberta do componente a partir de um contrato contextual), implantados em uma plataforma virtual e executados, através de operadores de orquestração da própria linguagem.

Detalhes sobre o SAFe serão apresentados no Capítulo 3. Por sua vez, estudos de caso demonstrando a sua utilização prática e validando seus principais aspectos (Montage e Map/Reduce) são apresentados no Capítulo 4.

1.5 Revisitando os Objetivos da Proposta de Tese

A seguir, são reapresentados os objetivos, gerais e específicos, anunciados no documento de Proposta de Tese aprovado que deu origem a este trabalho, com a finalidade de avaliar o cumprimento desses objetivos.

Objetivo Geral Projetar, implementar e validar o *framework* de aplicações da HPC Shelf, denominado SAFe, cujo objetivo é oferecer uma base para construção e provimento de aplicações através da abstração de nuvens computacionais que oferecem serviços de Computação de Alto Desempenho para usuários especialistas, viabilizando os conceitos e abstrações inovadores que circundam sua proposta.

Objetivos Específicos Os objetivos específicos da Tese proposta são:

i. Apresentar uma visão geral e atualizada do estado-da-arte da tecnologia de *workflows* científicos aplicados a computação de alto desempenho em nuvens computacionais, identificando as principais limitações das abordagens atuais;

ii. Propor e implementar uma linguagem de alto nível para a especificação de *workflows* científicos para orquestração de larga escala de computações paralelas na HPC Shelf, baseada em componentes paralelos do modelo Hash;

iii. Propor e implementar um conjunto de abstrações e artefatos para derivação de aplicações que provêem serviços de computação de alto desempenho na HPC Shelf, sob a perspectiva de nuvens computacionais do tipo SaaS (*Software-as-a-Service*).

Os objetivos, geral e específicos, foram cumpridos. O SAFe foi desenvolvido segundo as especificações anunciadas (Capítulo 3), incluindo a linguagem de descrição arquitetural e de orquestração SAFeSWL, também prevista em um dos objetivos específicos, bem como comparado com outras tecnologias de gerenciamento de *workflows* científicos com o objetivo de enfatizar sua originalidade e contribuições relevantes (Capítulo 5). Sua validação foi realizada através de dois estudos de caso (Montage e Map/Reduce), descritos no Capítulo 4. Ambos os estudos de caso, especialmente o Montage, possuem implementações em vários outros sistemas gerenciadores de *workflows* científicos, oferecendo um referencial de comparação.

1.6 Contribuições e Resultados desta Tese

Esta Tese de Doutorado é a primeira a apresentar a arquitetura geral da plataforma HPC Shelf. Pode-se afirmar que essa plataforma é, de forma parcial, uma das contribuições relevantes desta Tese, tendo em vista a participação ativa de seu autor nesse projeto. O SAFe constitui a sua contribuição principal, sendo desenvolvido sob duas premissas:

i. servir aos requisitos da HPC Shelf com relação ao mecanismo oferecido a provedores de aplicações para construi-las;

ii. propor inovações dentro do contexto de sistemas gerenciadores de *workflows* científicos.

Com relação ao primeiro ponto, em si próprio constitui uma contribuição inovadora, tendo em vista o caráter inovador da plataforma HPC Shelf, o que exigiu a pesquisa que levaria às decisões de projeto por trás do SAFe, consolidadas nesta Tese, com especial destaque ao seu alinhamento com tecnologias de gerenciadores de *workflows* científicos. Isso exigiu o estudo do estado-da-arte de tecnologias de *frameworks* para o desenvolvimento de aplicações de computação de alto desempenho, além de *workflows*.

- Dificuldade em adaptar linguagens existentes, como o BPEL por exemplo, às necessidades da HPC Shelf;

- Facilidade em manipular XML através de classes Java e padrões de *software*, como o *Visitor* usado nessa Tese;

- Flexibilidade na criação de novos comandos a partir dos requisitos da HPC Shelf, como a porta de tarefas de ciclo de vida, por exemplo.

Com relação ao segundo ponto, são listados abaixo as principais peculiaridades e inovações do SAFe em relação aos outros gerenciadores de *workflows* científicos, cuja amostra significativa é apresentada no Capítulo 5:

- separação entre especificação e implementação por meio de *contratos contextuais*[13] integrados com um sistema de descoberta (resolução) de implementações de componentes com base em propriedades a respeito do contexto de execução do componente, incluindo requisitos (funcionais e não-funcionais) impostos pela aplicação e características da plataforma de computação paralela sobre a qual vai executar;

- nível mais elevado de abstração para os usuários finais, chamados de *especialistas*, promovendo transparência em relação aos recursos (componentes).

- controle do ciclo de vida do componente (resolução, implantação, instalação e execução) de forma explícita através de operadores embutidos na própria linguagem de descrição dos *workflows* (SAFeSWL), o qual identificamos como uma característica importante para otimização do uso dos recursos na nuvem HPC Shelf e suportada por alguns gerenciadores de *workflows* existentes, embora de maneira implícita (instanciação incremental);

Uma discussão mais comparativa dessas características do SAFe com outros sistemas gerenciadores de *workflows* científicos é apresentada no Capítulo 5.

Finalmente, com relação ao desafio mais geral, discutido anteriormente, da oferta de artefatos de desenvolvimento para aplicações de CAD em conciliar modularidade, eficiência, poder de abstração, interoperabilidade e generalidade, o SAFe, em conjunto com a HPC Shelf, ataca esses pontos da seguinte forma:

[13] Este documento não apresenta detalhes maiores sobre o sistema de *contratos contextuais*, chamado Alite, uma vez que se trata de contribuição de uma outra Tese de Doutorado em andamento, também relacionada a contribuições sobre o projeto HPC Shelf.

- **modularidade**: a implementação de aplicações baseadas em componentes estimula a programação de módulos de computação dinamicamente intercambiáveis e que podem se comunicar através de suas portas específicas, ou seja, promovendo a programação com reuso e para reuso.

- **eficiência**: a programação de componentes paralelos altamente especializados para um determinado tipo de plataforma, incluindo aquelas de arquitetura heterogênea, por meio da abstração de contratos contextuais, permite a otimização do desempenho global da aplicação, sendo pouco significativo, para aplicações típicas, o peso das interconexões nos *workflows* SAFeSWL.

- **abstração**: o encapsulamento da lógica de computação paralela dentro de componentes compatíveis com o modelo Hash e exposição de seus serviços através de portas, inspiradas no padrão CCA, permite ao provedor de aplicações montar *workflows* em alto nível, sem necessidade de lidar com aspectos de computação paralela e otimização destas para a arquitetura da plataforma de execução alvo, preocupações atribuídas aos desenvolvedores de componentes. Analogamente, especialistas não precisam preocupar-se com a estruturação de soluções computacionais das aplicações que fazem uso, sendo essa uma preocupação do provedor de aplicações.

- **interoperabilidade**: o uso de Serviços *Web* para comunicação entre seus elementos arquiteturais permite que virtualmente qualquer infraestrutura de computação paralela distribuída possa ser incorporada à plataforma HPC Shelf, bastando que exista um *Back-End* capaz de instanciar plataformas virtuais sobre essa infraestrutura e que perfis de plataformas (contratos contextuais de componentes da espécie *plataforma*) sejam disponibilizados pelos seus mantenedores. Além disso, componentes podem virtualmente estar implementados sobre qualquer linguagem de programação, desde que suportado pelo *Back-End* da plataforma, muito embora esteja restrito, em sua implementação atual, aos componentes da plataforma Mono[14].

- **generalidade**: a plataforma HPC Shelf serve-se a propósitos gerais, dentro do contexto de aplicações com requisitos de CAD, uma vez que não há restrições para os domínios científicos e arquiteturas de plataformas de computação

[14] http://www.mono-project.com

paralela que os seus componentes podem servir, tampouco para os padrões de computação paralela suportados por esses componentes.

1.7 Estrutura do Documento

Este documento está organizado em 6 capítulos, incluindo este capítulo introdutório (**Capítulo 1**) e o de conclusão (**Capítulo 6**). Os demais são:

Capítulo 2 Descreve a plataforma HPC Shelf, pano de fundo para as contribuições desta Tese de Doutorado;

Capítulo 3 Descreve o *framework* de aplicações SAFe, a principal contribuição desta Tese de Doutorado;

Capítulo 4 Apresenta dois estudos de caso (Montage e Map/Reduce) que demonstram a utilização do SAFe para construção de aplicações sobre a plataforma HPC Shelf, com vistas à sua validação prática;

Capítulo 5 Discute as alternativas dentre sistemas gerenciadores de *workflows* científicos, explorando suas características e comparando-os com o SAFe, apresentado nos capítulos anteriores.

No **Capítulo 6**, além das conclusões do estudo, também é incluída uma discussão sobre limitações dos resultados alcançados por esse trabalho, bem como sugestões de trabalhos futuros relacionados ao SAFe.

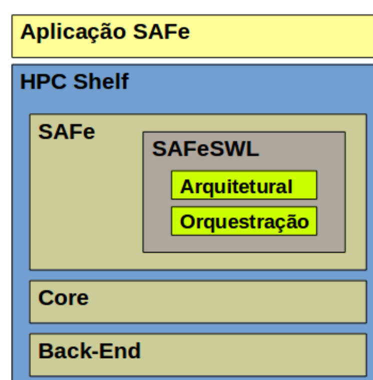


Figura 1.1: Visão geral da relação entre o SAFe e os demais elementos arquiteturais da HPC Shelf.

A Figura 1.1 apresenta uma visão geral da posição do SAFe e, internamente, o SAFeSWL frente à HPC Shelf. A **Aplicação SAFe** é desenvolvida pelo *provedor*

de aplicações e disponibilizada para o usuário *especialista* que, teoricamente, não tem conhecimento da arquitetura abaixo. o SAFe é responsável pela implementação das aplicações que serão usadas pelo especialista, comunicando-se, via Serviços *Web*, com os outros elementos arquiteturais da HPC Shelf, como o *Core* e o *Back-End*.

Capítulo 2

A Nuvem HPC Shelf

A HPC Shelf é uma proposta de nuvem computacional que oferece um conjunto de serviços voltados a composição e execução de aplicações com requisitos de computação de alto desempenho. Essas aplicações auxiliam a solução de problemas computacionalmente intensivos de interesse de usuários especialistas. A HPC Shelf é produto de projetos de pesquisa desenvolvidos pelo grupo de Computação de Alto Desempenho do Mestrado e Doutorado em Ciência da Computação (MDCC) da Universidade Federal do Ceará e constitui pano de fundo para as contribuições apresentadas por esta Tese de Doutorado. Os principais elementos arquiteturais da HPC Shelf são o *Front-End*, o *Core* e o *Back-End*. Esta Tese de Doutorado tem como contribuição principal um *framework* para composição e execução de aplicações baseadas nas orquestração de componentes paralelos por meio de *workflows* científicos, chamado SAFe, o qual será explicado em mais detalhes no próximo capítulo. O SAFe é a base para o *Front-End* da HPC Shelf. Este capítulo tem por objetivo apresentar uma visão geral a HPC Shelf.

A HPC Shelf tem como principal característica a orientação a componentes paralelos do modelo Hash [Carvalho Junior e Lins 2008], cuja composição resulta em *sistemas de computação paralela* que implementam soluções computacionalmente intensivas para resolver problemas de interesse de seus usuários finais. Esses sistemas, em sua especificação, levam em consideração as *plataformas* de computação paralela e os *componentes* de *software* que deverão executar sobre elas. Em computação de alto desempenho, essa visão integrada do *hardware* (plataforma) e do *software* (algoritmos) objetiva explorar ao máximo o potencial de desempenho das plataformas, pois muitos algoritmos e suas implementações fazem suposições sobre as arquiteturas de execução para serem eficientes. Nesse sentido, os componentes

que formam as aplicações HPC Shelf são ditos *componentes de sistema* o qual é uma noção mais geral do que o termo usual *componente de software*.

HPC Shelf adota um sistema de *contratos contextuais* (ver Seção 2.6) os quais, além de separar as implementações dos componentes de suas respectivas interfaces, também gerenciam a escolha das implementações mais eficientes de um determinado componente de acordo com o seu *contexto de implantação* [1]. Tal contexto representa os requisitos que a aplicação faz sobre o componente de *software* e as características da plataforma a qual esse mesmo componente irá executar.

O objetivo geral da nuvem HPC Shelf é disponibilizar um ambiente para construção e execução de aplicações que ofereçam interfaces amigáveis nas quais os usuários especialistas podem resolver problemas de seu interesse. Para isso, toda a descrição de problemas deve ser feita através de abstrações do domínio da aplicação, que escondem os detalhes da implementação paralela da solução, bem como qual plataforma de computação paralela a aplicação irá utilizar em sua execução.

Sendo assim, usuários especialistas podem se beneficiar do poder computacional de plataformas de computação paralela, de forma transparente, sem lidar com questões técnicas relativas ao desenvolvimento do *software* e arquiteturas de computadores paralelos.

Os objetivos específicos, descritos abaixo, delineiam os principais pontos que deverão estar presentes na implementação da HPC Shelf.

- Oferecer uma plataforma para integração de usuários especialistas de domínio, interessados em soluções de problemas computacionalmente intensivos através de aplicações disponibilizadas pela plataforma e na interação com outros especialistas;

- Fornecer um arcabouço para a montagem de aplicações a partir de componentes que satisfazem o modelo Hash e representam tanto os elementos de *hardware* quanto os elementos de *software* envolvidos, sob uma perspectiva de *sistemas de computação paralela* [Grama et al. 2003];

- Possibilitar que aplicações possam acompanhar o estado da execução de computações (de longa duração) na solução de problemas de interesse de seus

[1] Na prática, umas série de valorações inseridas no contrato contextual serve como entrada para um algoritmo de resolução o qual irá escolher a implementação adequada de um determinado componente. A implementação escolhida é, teoricamente, otimizada para executar em uma determinada plataforma escolhida pelo cliente da aplicação.

usuários, bem como a visualização adequada das suas saídas e armazenamento de dados resultantes para posterior uso em outras computações e visualização;

- Garantir que aplicações escolham, em tempo de execução, a implementação mais eficiente de um componente de software de acordo com a arquitetura da plataforma de computação paralela onde será instanciado para executar, a fim de melhor explorar os seus recursos, bem como de acordo com os requisitos a ele impostos pela aplicação, como, por exemplo, as características dinâmicas da estrutura de dados que será processada pelo componente;

- Permitir a execução de código-fonte legado em plataformas modernas de computação paralela, utilizando dos serviços oferecidos pela nuvem computacional. Fazendo uso da linguagem Mono[2], por exemplo, é possível criar bibliotecas para execução de código legado nas linguagens mais conhecidas. A ideia é encapsular o código legado em componentes, fornecendo interfaces de acesso ao mesmo;

- Fornecer a opção de escolha da plataforma de computação paralela onde serão executadas computações na solução de problemas descritos pela aplicação, de forma a minimizar os custos associados ao uso dos recursos, tais como o custo financeiro e energético. Uma das vantagens da tecnologia de nuvens computacionais é a de usar máquinas virtuais para instanciar plataformas de computação paralela, caso não existam fisicamente;

- Suportar uma linguagem para descrição de *workflows* que representam orquestrações de componentes na solução de problemas de interesses de uma aplicação. De fato, problemas declarativamente descritos pelos usuários de uma aplicação, através de abstrações de mais alto nível e próximas ao seus domínio de conhecimento, devem ser mapeadas para essa linguagem, que pode ser vista como a linguagem de programação da HPC Shelf, vista como uma plataforma de computação de larga escala.

2.1 O Modelo de Componentes Hash

O modelo de componentes Hash [Carvalho Junior et al. 2013] tem como objetivo apresentar uma noção geral de componentes paralelos os quais possam ser usados na programação baseada em componentes para plataformas de computação de alto

[2]http://www.mono-project.com/

desempenho. O modelo introduz um conceito geral de componentes naturalmente paralelos, os chamados *componentes-#*, que são constituídos por um conjunto de *unidades* que representam processos executando em nós distintos de processamento de uma plataforma de computação paralela com memória distribuída.

Um componente-# pode ser formado por outros componentes-#, chamados de *componentes aninhados*. Isso é possível através de uma abordagem de composição hierárquica chamada de *composição por sobreposição* [Carvalho Junior e Corrêa 2010]. Para tanto, uma *função de sobreposição* mapeia as unidades dos componentes aninhados às unidades do componente-# mais externo. Essa operação define dependências do componente hospedeiro em relação aos seus componentes aninhados. Cada mapeamento da função de sobreposição define uma *fatia* da unidade alvo.

Uma *configuração* especifica unidades, componentes internos, e a função de sobreposição de um componente-#, usando para isso uma linguagem de descrição arquitetural, ou ADL (*Architecture Description Language*).

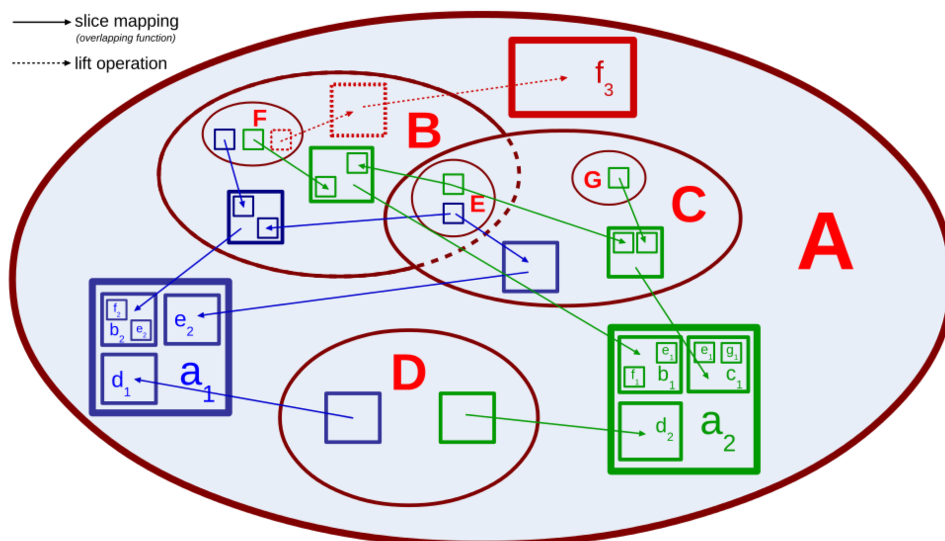


Figura 2.1: Composição por sobreposição (perspectiva hierárquica).

A Figura 2.1 demonstra a composição por sobreposição de um componente hipotético chamado **A**, com unidades a_1, a_2 e f_3. O componente **A** foi construído através da composição por sobreposição dos componentes **B, C, D, E, F** e **G**, os quais podem ser chamados de componentes aninhados de **A**. A função de sobreposição mapeia as unidades d_1, e_2 e b_2, respectivamente de **D**, **C** e **B** para a unidade a_1 de **A**. Sendo assim, pode-se afirmar que d_1, e_2 e b_2 são fatias da unidade a_1 do componente **A**. Por sua vez, as fatias da unidade a_2 são b_1, c_1 e d_2,

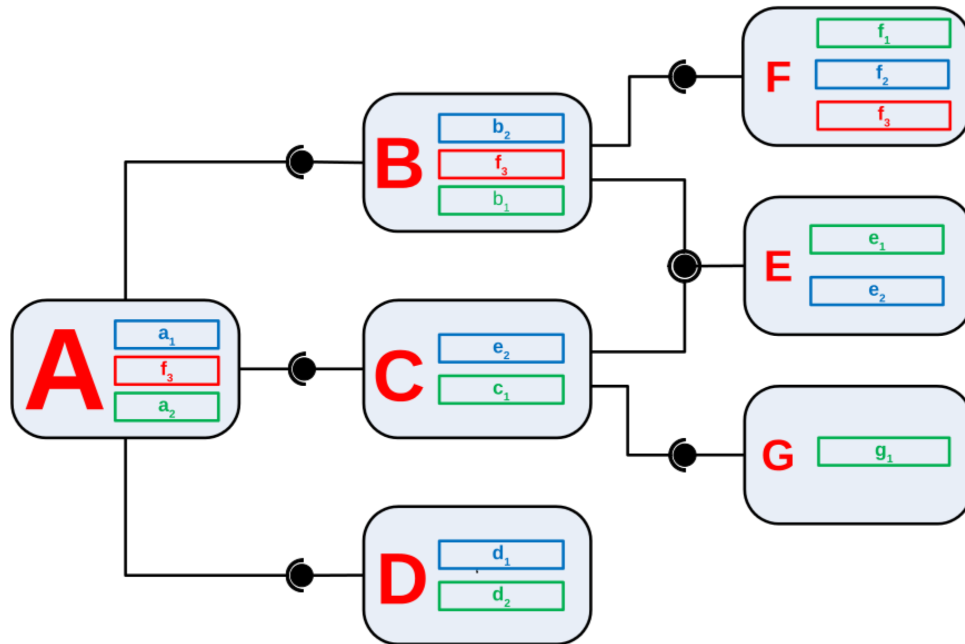


Figura 2.2: Composição por sobreposição (perspectiva UML).

fazendo uso do mesmo raciocínio. É possível notar também que o componente **A** tem uma outra unidade chamada f_3, a qual vem diretamente de seu componente aninhado **F**. Essa configuração é chamada de *lifting*. O *lifting* define um unidade de um componente aninhado que também é unidade de seu componente externo. Por exemplo, e_2 é uma unidade tanto de **E** (diretamente) e **C** (transitividade, por *lifting*). Note também que o componente **E** é compartilhado entre os componentes **B** e **C**. Por exemplo, **E** pode representar uma instância de uma estrutura de dados compartilhada entre os componente **B** e **C**. A figura ilustra que a sobreposição por composição pode ser aplicada recursivamente aos componentes internos de **A**.

A Figura 2.2 mostra uma visão alternativa, inspirada em UML (*Unified Modeling Language*), da composição por sobreposição mostrada no diagrama da Figura 2.1, o qual foca nas dependências entre os componentes, provendo intuição de como o paralelismo é encapsulado dentro de componentes através da abstração de unidades.

Além do suporte a unidades distribuídas e composição por sobreposição, uma plataforma de componentes compatível com o modelo Hash também deve suportar um conjunto finito de espécies de componentes. Espécies de componentes agrupam componentes com interesses comuns, específicos para a plataforma alvo. É possível, por exemplo, usá-las para a criação de bibliotecas de conectores reusáveis especializados em implementar a interoperabilidade entre componentes em uma

plataforma de computação de alto desempenho (*cluster*, nuvem computacional ou grade computacional). Espécies de componentes podem também ser usadas para lidar com requisitos não funcionais.

A implementação de referência do modelo Hash se chama HPE (*Hash Programming Environment*) [Carvalho Junior e Rezende 2013]. O HPE é uma plataforma de componentes voltada à construção, implantação e execução de programas paralelos destinados a plataformas de *cluster computing*, a partir da composição de componentes paralelos que satisfazem o modelo Hash, além de também compatíveis com o padrão CCA (*Commom Componenet Architecture*). O HPE é então um tido como um *framework* CCA com expressividade completa para descrever padrões de paralelismo distribuído [Carvalho Junior e Corrêa 2010].

O HPE suporta oito espécies de componentes:

i. *plataformas*, representando as plataformas de computação paralela onde as unidades são os nós de processamento;

ii. *computações*, representando as implementações do algoritmos paralelos, onde as unidades são processos paralelos;

iii. *estrutura de dados*, representando entrada, saída e dados intermediários os quais são processados pelas computações, onde as unidades representam partições dos dados distribuídos pelos nós de processamento;

iv. *sincronizadores*, representando padrões de comunicação e sincronização entre processos paralelos;

v. *topologias*, representando a organização topológica de processos paralelos em uma computação paralela;

vi. *ambientes*, representando meios de comunicação e sincronização entre processos paralelos;

vii. *qualificadores*, representando características e propriedades que influenciam a forma como componentes, de qualquer espécie, são implementados;

viii. *aplicações*, representando a composição de mais alto nível de componentes os quais são orquestrados para resolver um problema de um dado domínio, provendo uma interface amigável.

A plataforma HPE é formada por três elementos arquiteturais independentes principais:

- O *Front-End* é a interface através da qual desenvolvedores podem controlar o ciclo de vida de componentes, incluindo as etapas de criação e montagem (uso na composição de outros componentes), bem como a implantação, instanciação e execução em plataformas de computação paralela (*cluster*). Atualmente, encontra-se implementada como um *plug-in* para o ambiente de desenvolvimento Eclipse, com um ambiente visual para composição de componentes por sobreposição.
- O *Core* representa um repositório de componentes, onde componentes são instalados e oferecidos aos usuários através do *Front-End*.
- O *Back-End* gerencia a plataforma de computação paralela, ou seja, o *cluster*, no qual implementações de componentes, buscados no *Core*, são implantados e instanciados para execução de aplicações.

2.2 Espécies de Componentes da HPC Shelf

As espécies de componentes que podem ser utilizados para construção de sistemas computacionais sobre a plataforma HPC Shelf são:

- ***Plataformas***: representam plataformas de computação paralela de memória distribuída, também chamadas de plataformas virtuais, formando a infraestrutura da HPC Shelf.
- ***Computações*** representam algoritmos paralelos otimizados para certas características arquiteturais de uma classe de plataformas virtuais. Em um sistema de computação paralela, um componente de computação está sempre associado a um componente plataforma, o qual representa a plataforma virtual onde é implantado e instanciado para execução.
- ***Conectores***: desempenham, entre componentes de computação, o papel de "palco" para sua coreografia (comunicação e sincronização) ou de "maestro" para sua orquestração, podendo ainda implementar "*hubs*" de acessos a múltiplas fontes de dados entre esses componentes de computação. Um componente conector é composto por um conjunto de facetas, cada uma das quais associada a um componente (computação ou fonte de dados) ao qual o

conector está associado, de modo que cada faceta é implantada e instanciada sobre a plataforma virtual associada a esse componente. As facetas de um conector podem comunicar-se através de um canal de comunicação encapsulado no conector, para troca de mensagens, possivelmente sobre a *internet*.

- ***Fontes de dados*** representam interfaces de acesso a repositórios de grandes massas de dados de interesse de sistemas de computação, melhor explicadas adiante (Seção 2.2.2).

Componentes da HPC Shelf expõem portas para comunicação com outros componentes. As portas são conectadas através de ligações (*bindings*) as quais podem ser vistas como tipos primitivos de componentes conectores. As portas podem ser de dois tipos: *portas de ambiente* e *portas de tarefas*.

Uma porta de ambiente pode ser uma porta do tipo *provedora* ou *usuária*. Uma porta provedora disponibiliza serviços de um componente para serem consumidos via porta usuária de um outro componente. A comunicação entre as duas portas é intermediada por meio de um *binding* compatível, que faz uso das capacidades da arquitetura ao qual os componentes em questão estão instalados. Portas de ambiente de componentes da plataforma HPC Shelf seguem o padrão de projeto análogo ao utilizado para ligação de componentes no modelo de componentes CCA (*Common Component Architecture*) [Armstrong et al. 2006].

No caso das portas de tarefas, não existe a noção de usuário e provedor. Portas de tarefas de componentes diferentes são compatíveis quando exportam o mesmo conjunto de *ações*. Ligações entre um conjunto de duas ou mais portas de tarefas compatíveis fazem com que os componentes que as possuem se tornem parceiros, os quais reagem à ativação síncrona de suas ações, ou seja, a ativação de uma ação em uma porta de tarefas, por parte de um componente parceiro que a possui, permanece bloqueada até que todos os componentes parceiros dessa porta ativem a mesma ação.

Uma aplicação da HPC Shelf orquestra a execução do conjunto de componentes de um sistema de computação paralela conectando-se às suas portas de tarefas, atuando, na prática, como um conector de orquestração das ações expostas por esses componentes. Os detalhes de como isso é realizado são apresentados nas próximas seções e, com mais profundidade, no Capítulo 3.

2.2.1 Portas de Componentes da HPC Shelf

Componentes conectores e de computação da HPC Shelf obrigatoriamente devem possuir pelo menos uma porta de tarefa, que exporte ações computacionais que

devem ser ativadas segundo um certo protocolo e representam seus interesses funcionais. Por outro lado, fontes de dados e plataformas não possuem portas de tarefa. Qualquer componente pode possuir uma ou mais portas de ambiente. Além disso, componentes da HPC Shelf possuem portas de dois tipos padrões, de *alocação* e de *ciclo de vida*, de acordo com as suas espécies, explicadas a seguir.

Portas de Alocação

Componentes de computação, conectores e plataformas possuem portas de ambiente to tipo Allocation. Para computações, a porta é única e no papel de usuária, enquanto, para plataformas, também é única porém no papel de provedora. Por sua vez, conectores possuem uma porta de alocação usuária associada a cada faceta. A porta de alocação estabelece em que instância de plataforma um componente de computação ou faceta de conector deve ser implantada e instanciada. Além disso, em tempo de execução, essa porta oferece serviços de plataforma, que permitem ao componente trocar informações com a plataforma virtual sobre a qual está executando. Em um sistema de computação bem formado para a HPC Shelf, a propriedade de comutatividade do grafo de ligação das portas exige que facetas de conectores devem conectar suas portas, de ambiente e de tarefas, somente com componentes de computação que estejam associados, através de suas portas de alocação, às mesmas plataformas virtuais onde encontra-se alocadas essas facetas.

Portas de Ciclo de Vida

Componentes de computação, conectores e plataformas possuem em comum uma porta de tarefas do tipo LifeCycle, com as seguintes ações: *resolve*, *deploy*, *instantiate* e *release*. Essas portas estão conectadas à portas de tarefas do componente *workflow*, um componente conector de natureza especial que deve existir em qualquer sistema computacional preparado para execução na HPC Shelf (melhor explicado a partir da Seção 3.2). A partir do início da execução, supõe-se que os componentes de computação, conectores e plataformas que formam o sistema ativam as ações dessa porta na seguinte ordem: *resolve*, *deploy* e *instantiate*. O componente *workflow* deve portanto sincronizar ativando as ações correspondentes na sua porta correspondente, na mesma ordem. O efeito de completar essas ações é explicado a seguir:

resolve executa o procedimento de resolução de componentes aplicado a um contrato, dito *contrato contextual*, que descreve as propriedades do componente requisitado pelo sistema, resultando na escolha de uma implementação de componente propriamente dita;

deploy para computações e conectores, completa-se com os componentes estando preparados para que instâncias desses componentes sejam lançadas nas plataformas virtuais as quais estão associadas, enquanto para plataformas, prepara a infraestrutura de computação paralela para instanciação de uma plataforma virtual (conjunto de máquinas virtuais que representarão os nós de uma plataforma de computação paralela distribuída);

instantiate para computações e conectores, completa-se com os componentes estando preparados para receber e fazer requisições através de suas demais portas de tarefas e de ambiente, iniciando-se o seu laço principal de execução (função *main*), enquanto, para componentes plataforma, completa-se com a plataforma virtual estando preparada para receber a instanciação de componentes a ela associados através de portas de alocação;

A ação release é ativada automaticamente quando um componente finaliza o seu laço principal de execução. Caso o componente *workflow* também a ative, o componente é liberado da memória, no caso de computação ou faceta de conector, não podendo mais haver acesso às suas portas. No caso de componentes plataformas, as máquinas virtuais são liberadas.

2.2.2 Fontes de Dados

Ao contrário de componentes das demais espécies, fontes de dados são componentes previamente instalados e disponíveis para aplicações da HPC Shelf, ou mesmo de outras plataformas de computação baseadas em *workflows* científicos, tais como as plataformas apresentadas no Capítulo 5. Por esse motivo, não possuem portas de ciclo de vida. De maneira mais geral, possuem apenas portas de ambiente no papel de provedoras, que oferecem uma interface para acesso aos dados de interesse do sistema de computação.

Fontes de dados podem ser ligadas, por meio de suas portas de ambiente provedoras, tanto a componentes de computação quanto a componentes conectores. No primeiro caso, a ligação entre as portas é feita de maneira indireta, onde a computação e a fonte de dados residem em plataformas de computação distintas. No último caso, a conexão entre as portas é direta, sendo uma das facetas do conector, especialmente preparada para isso, instanciada na mesma plataforma onde encontram-se os dados. Dessa forma, no caso do sistema computação necessitar acessar apenas uma síntese de uma grande massa de dados que pode ser obtida

por computações relativamente pouco custosas sobre esses dados, o custo da comunicação é evitado. Essa técnica, de mover as computações em direção aos dados, é bastante comum em sistema de computação científica de larga escala.

2.3 Atores Envolvidos na HPC Shelf

A HPC Shelf prevê a sua utilização por quatro tipos de atores (*stakeholders*), que diferem em seus papéis e obrigações. O foco da HPC Shelf são os *usuários especialistas*, ou seja, são eles os usuários finais da nuvem, interessados em soluções de problemas através das aplicações. Sendo assim, os atores envolvidos são:

- Usuário Especialista;
- Provedor de Aplicações;
- Desenvolvedor de Componentes;
- Mantenedor de Plataformas.

2.3.1 Usuários Especialistas

O especialista de domínio é o usuário final da nuvem HPC Shelf. Tem o interesse em executar aplicações para resolver problemas inseridos no seu domínio de conhecimento. Para isso, não necessita ser um especialista em ciências da computação ou dominar soluções computacionais para problemas em sua área de conhecimento. Preocupa-se apenas com a expressão do problema que deseja resolver usando abstrações de alto nível. Portanto, suas qualidades compreendem:

- São especialistas no domínio da aplicação (físico, químico, economista, engenheiro, biólogo, etc.), sendo capazes de expressar problemas de seu interesse dentro desse domínio;
- Não precisam entender detalhes sobre os componentes da nuvem usados pela aplicação, ou mesmo serem cientes da sua arquitetura baseada em componentes, necessitando de uma interface transparente para descrição dos problemas que desejam resolver por intermédio das aplicações;
- Não precisam conhecer a infraestrutura de computação paralela usada para a execução das soluções computacionais construídas pela aplicação.

O especialista nada conheço sobre a API do SAFe, tampouco sobre sua linguagem SAFeSWL. Este usuário preocupa-se apenas em manipular a aplicação desenvolvido pela usuário provedor de aplicações.

2.3.2 Provedor de Aplicações

O provedor de aplicações tem o perfil de engenheiro ou arquiteto de *software*, interessado na montagem das aplicações para os usuários especialistas a partir de componentes disponibilizados na nuvem. Devido a natureza das aplicações de CAD, o provedor de aplicações deve possuir conhecimentos em técnicas de ciências computacionais. Ele não é, portanto, conhecedor profundo de técnicas de computação paralela, ou apenas as compreende em alto nível, sem se ater a detalhes arquiteturais e que irá configurar uma aplicação formada por outros componentes pré-instalados. Podemos resumir suas atividades em:

- Construir uma interface de alto nível para especificação de problemas por parte dos especialistas, bem como para acompanhamento, possivelmente interativo, da execução de soluções computacionais e apresentação de seus resultados;
- Compor soluções computacionais para problemas que devem ser resolvidos pela aplicação a partir dos componentes existentes na nuvem, incluindo a especificação de como devem ser orquestrados;
- Conhecer as características arquiteturais de plataformas de computação paralela que são importantes para atender aos requisitos, de desempenho e de custo, impostos pelos usuários especialistas.

O provedor de aplicações tem conhecimento profundo da API do SAFe e sua linguagem SAFeSWL para poder criar aplicações de alto nível para o especialista.

2.3.3 Desenvolvedor de Componentes

O desenvolvedor de componentes deve ter conhecimento em programação paralela e arquiteturas de plataformas de computação paralela, com o propósito de criar componentes, definindo suas regras de composição e interesses funcionais a serem atendidos. Um componente deve ser implantado e armazenado em um repositório de componentes, bem como publicado em um catálogo, de modo que possa ser usado por outros desenvolvedores de componentes e provedores de aplicações. Resumidamente, suas características e responsabilidades são:

- Formação em ciência da computação (essencial) e/ou ciência computacional (desejável);

- Implementação de componente primitivos, usando linguagens e bibliotecas próprias para explorar ao máximo os recursos de plataformas de computação paralela;
- Uso de técnicas de otimização de código para explorar características arquiteturais de plataformas de computação paralela;
- Conhecimento em computação paralela, incluindo algoritmos, técnicas de programação e arquitetura de plataformas de computação paralela;
- Habilidade para construção de componentes por composição de outros existentes (programação em larga escala);
- Conhecimento do processo de criação de componentes paralelos usando o modelo Hash.

O desenvolvedor de componentes não tem conhecimento do *SAFe*. Ele apenas implementa componentes de acordo com o padrão de portas de ambiente e tarefas, as quais serão conectadas pela linguagem SAFeSWL.

2.3.4 Mantenedor de Plataformas

O mantenedor de plataforma será responsável em implantar, configurar, disponibilizar e monitorar o bom funcionamento de plataformas de computação paralela. Na HPC Shelf, essas plataformas são representadas pelos componentes da espécie *plataforma*. Esse tipo de componente encapsula a informação necessária para instanciar plataformas virtuais, as quais representam computadores paralelos instanciados na nuvem sobre a infraestrutura do mantenedor. Ele também é responsável em instalar as bibliotecas necessárias para a execução dos problemas sobre essas plataformas. Suas principais funções são:

- Garantir a segurança das aplicações que executam sobre a sua infraestrutura, a partir das políticas de segurança gerais da HPC Shelf;
- Instalar bibliotecas e pacotes de interesse das aplicações habilitadas para executar sobre as plataformas virtuais que mantém;
- Implantar, configurar e monitorar as plataformas virtuais instanciadas na nuvem pelas aplicações em execução.

O mantenedor de plataformas restringe-se a apenas gerenciar as plataformas virtuais da HPC Shelf, não se preocupando com o SAFe nem as aplicações construídos sobre seu arcabouço.

2.4 Relacionamento entre Atores

De uma forma sucinta, o relacionamento entre os atores apresentados nas subseções anteriores pode ser resumido da seguinte forma. O provedor de aplicações é responsável em criar as aplicações para o usuário especialista. Por sua vez, o mantenedor de plataformas é responsável em prover componentes que representam plataformas virtuais de computação paralela para o provedor de aplicações. Além disso, o mantenedor também disponibiliza, para o conhecimento dos desenvolvedores de componentes, contratos, ditos *contratos contextuais*, que representam as características das plataformas que é capaz de instanciar sobre a infraestrutura computacional que mantém. Finalmente, o desenvolvedor de componentes implementa componentes (de *software*) para uso do provedor de aplicações, otimizados de acordo com plataformas de computação paralela alvo cujos contratos contextuais são conhecidos.

É importante reforçar que o ator final da HPC Shelf é o usuário especialista. Os outros três atores servem como meios para que o especialista atinja o seu objetivo principal, ou seja, executar e monitorar soluções de problemas por meio de aplicações da HPC Shelf.

2.5 Visão Arquitetural da HPC Shelf

A arquitetura da HPC Shelf é constituída por três tipos de elementos: *Front-End*, *Back-End* e *Core*.

2.5.1 O *Front-End*

O *Front-End* representa interfaces a partir das quais os atores acessam os serviços da nuvem. Para o usuário especialista de domínio, tratam-se das *aplicações*. Para o provedor de aplicações, trata-se do SAFe (*Shelf Application Framework*), o *framework* para construção de aplicações da HPC Shelf. Para o desenvolvedor de componentes, trata-se de uma extensão ao ambiente de composição de componentes do HPE (*Hash Programming Environment*), mencionado na Seção 2.1. Esse mesmo ambiente tem sido adaptado para servir aos mantenedores de plataformas.

O SAFe tem como objetivo fornecer o suporte para a construção de interfaces amigáveis para descrição de problemas dentro de um domínio específico, através

de abstrações adequadas para usuários especialistas nesse domínio, bem como o mapeamento desses problemas em soluções computacionais especificadas por meio de *workflows* de orquestração de componentes publicados na HPC Shelf. Tais interfaces e mecanismos de mapeamento de problemas através delas descritos para soluções computacionais, na forma de *workflows* científicos baseados em componentes, definem as aplicações da HPC Shelf.

Fazendo um paralelo com o padrão de projeto Modelo/Visão/Controlador (MVC[3]) [Schmidt et al. 1996], podemos afirmar que a *aplicação* a ser executada é a *visão* do especialista, o qual trabalha com abstrações de alto nível. O papel de controlador é exercido pelo SAFe, através do qual *workflows* de componentes são definidos pelo provedor de aplicações a partir da descrição, por parte dos especialistas, dos problemas que devem ser resolvidos e cujos resultados devem ser apresentados ao especialista. Finalmente, o *modelo* é a infraestrutura de componentes da HPC Shelf, a qual irá executar o *workflow*, após a instanciação dos componentes a serem orquestrados, tanto os de *software*, representando implementações de algoritmos e estruturas de dados, quanto os de *hardware*, representando plataformas de computação paralela.

Por ser a principal contribuição que esta Tese apresenta dentro do contexto da plataforma HPC Shelf, os detalhes sobre o projeto e a implementação do SAFe serão apresentados separadamente no Capítulo 3. Por sua vez, as principais contribuições do SAFe em relação ao estado-da-arte de plataformas de construção e execução de *workflows* científicos baseados em componentes são discutidas no Capítulo 5.

O *Front-End* do desenvolvedor de componentes oferece a ele um conjunto de interfaces que possibilita a construção de componentes de *software* que serão disponibilizados na HPC Shelf. Os componentes devem ser programados com linguagens e bibliotecas comumente usadas na programação de aplicações paralelas. Dessa forma, esses componentes podem ser otimizados segundo as características arquiteturais de uma classe de plataformas de computação paralela. Atualmente, uma extensão do ambiente visual de composição de componentes do HPE é usado como *Front-End* do desenvolvedor.

O *Front-End* do mantenedor oferece a ele um conjunto de interfaces que possibilitam o gerenciamento das plataformas virtuais sobre as quais os componentes desenvolvidos irão ser implantados, instanciados e executados. Essas plataformas virtuais serão instanciadas sobre as infraestruturas reais de computação paralela.

[3] Do inglês, *Model-View-Controller*.

É importante ressaltar que as plataformas virtuais também serão vistas pela HPC Shelf sob a abstração de componentes, os quais irão encapsular todas as informações necessárias para a sua alocação.

Sob a perspectiva dos usuários especialistas de domínio, a HPC Shelf é vista como uma formação de nuvens SaaS, onde as aplicações constituem o serviço. Por sua vez, sob a perspectiva do provedor de aplicação e do desenvolvedor de componentes, a HPC Shelf é uma nuvem PaaS, cujo serviço permite a construção e implantação da infraestrutura de *software* das aplicações. Finalmente, sob a perspectiva do mantenedor, a HPC Shelf pode ser vista como uma nuvem IaaS, oferecendo serviços de infraestrutura de computação paralela para as aplicações.

2.5.2 O *Core*

O *Core* tem como função gerenciar a biblioteca de componentes da HPC Shelf, bem como implementar o sistema de contratos contextuais chamado *Alite*[4]. Para cada componente requisitado pelo *workflow*, o *Core* retorna uma lista de pares que incluem uma implementação do componente de *software* (computação ou faceta de conector) e o componente da espécie *plataforma* que representa uma plataforma de computação paralela capaz de executá-lo. É responsabilidade da aplicação, com possível intervenção do usuário, escolher a melhor alternativa dentre as sugeridas pelo *Core*, de modo a melhor atender ao cronograma de execução determinado pelo *workflow*, bem como a restrições orçamentárias do usuário especialista, antes de executá-lo (restrições de qualidade). O *Core* então se comunica com o *Back-End* a fim de instanciar o componente plataforma e o componente de *software*, esse último dentro da plataforma virtual resultante da instanciação do primeiro.

2.5.3 O *Back-End*

O *Back-End* gerencia as plataformas de computação paralela de um mesmo mantenedor, controlando a instanciação de plataformas virtuais em seu domínio. Ele informa ao *Core* quais são os perfis de plataformas disponíveis, representados como componentes da espécie *plataforma*. O contrato contextual desses componentes inclui toda a informação necessária para a instanciação de plataformas virtuais com as características determinadas pelo contrato.

[4] O nome Alite vem do tipo mais comum de cimento, cuja formulação é descrita por C_3S, que, coincidentemente, é o acrônimo inicialmente adotado para *Component Contextual Contract System*. Entretanto, cimento remete à ideia de união entre duas superfícies, portanto adequado para o conceito de sistema de contratos contextuais como cola entre a aplicação e os componentes.

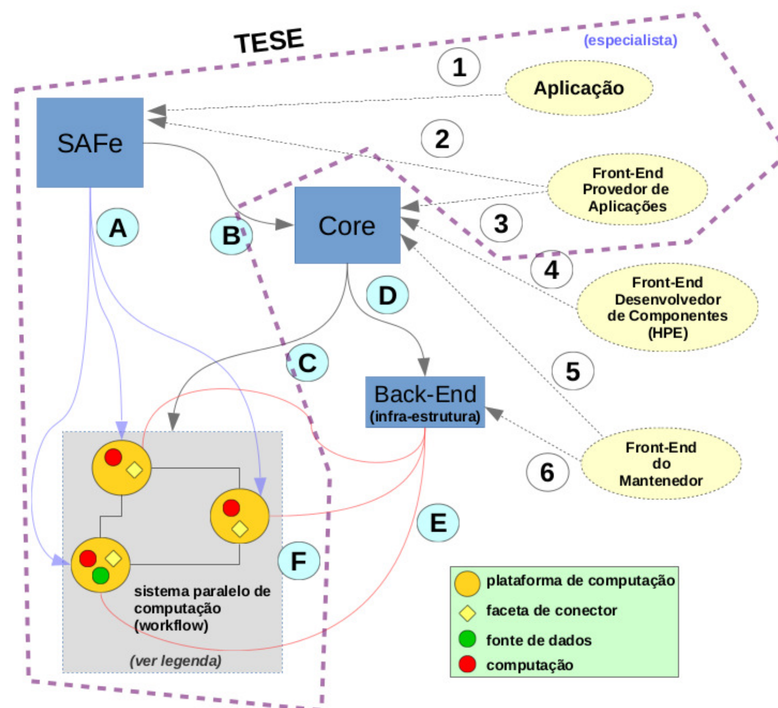


Figura 2.3: Visão em alto nível dos módulos da HPC Shelf, com destaque em pontilhado da contribuição desta Tese.

2.5.4 Visão Arquitetural

As instâncias dos elementos arquiteturais da HPC Shelf se relacionam através de *Serviços Web*, oferecendo serviços de interesse mútuo. A Figura 2.3 ilustra os três elementos arquiteturais principais da HPC Shelf (SAFe, *Core* e *Back-End*), além dos serviços entre eles. Nos parágrafos a seguir, uma descrição mais detalhada da figura.

Em **A**, o SAFe se comunica diretamente com os componentes instanciados nas plataformas de computação, através das portas de tarefa (funcional) e de ambiente (não funcional), implementadas como *Serviços Web*. As portas são o único meio de comunicação entre o SAFe e os componentes instanciados. O SAFe é de uso do especialista indiretamente, através das aplicações, e do provedor de aplicações, o qual faz uso de sua API para construir aplicações usufruídas pelos usuários especialistas.

Em **B**, o SAFe envia comandos ao *Core* (graças a uma interface de comunicação entre ambos) para que o mesmo resolva componentes (decida a melhor implementação, por meio da resolução de contratos contextuais), implante os componentes em uma determinada plataforma computacional e os instancie para que fiquem prontos para a orquestração pelo *workflow* gerido pelo SAFe. Componentes instanciados no *Back-End* disponibilizam suas portas de ambiente e de tarefa

conectadas à aplicação através de *Serviços Web*.

Em **C**, o *Core* envia comandos para implantação e instanciação de componentes nas plataformas virtuais instanciadas pelo *Back-End*. Nessa fase, ao receber o comando, é de responsabilidade do *Back-End* instalar o componente, bem como suas dependências, e instanciá-lo. No caso da HPC Shelf, instanciar um componente significa tornar suas portas disponíveis para invocação.

Em **D**, o *Core* envia comandos para o *Back-End* para que o mesmo implante as plataformas virtuais a serem usadas na implantação dos componentes do *workflow* no passo **C**. As plataformas virtuais são instanciadas de acordo com as necessidades da aplicação.

Em **E**, o *Back-End* cria as plataformas virtuais e disponibiliza esses mesmos recursos para os componentes escolhidos pelo *Core*. As plataformas virtuais são na verdade perfis de computação instanciados de acordo com as necessidades de QoS da aplicação, detalhadas no contrato contextual.

Em **F**, temos o sistema paralelo de computação formado por diversas plataformas virtuais, cada qual representando uma instância de um perfil de plataforma (contrato contextual de componente da espécie plataforma) adequado para executar os componentes a ela alocados da forma mais otimizada possível. Além disso, as plataformas virtuais comunicam-se entre si através de conectores. Deve-se lembrar que conectores constituem a espécie de componentes dedicada a servir de meio para coordenação entre diversos componentes de outras espécies. Para que isso seja possível, possui facetas distribuídas entre diversas plataformas virtuais.

No que diz respeito ao conjunto de *Front-Ends* envolvidos, além de cada ator para cada *Front-End*, a Figura 2.3 apresenta vários relacionamentos, destacados pelos números. A seguir, detalhamos cada um.

O relacionamento **1** é entre a aplicação e o SAFe. A aplicação submete problemas ao SAFe e recebe resultados através de portas de ambiente dos componentes dos *workflows* que representam as soluções computacionais para esses problemas. Na verdade, a aplicação é construída sobre o SAFe, usando a sua API.

O relacionamento **2** é entre o SAFe e o *Front-End* do provedor. Esse último cria aplicações compatíveis com o SAFe. Ainda no mesmo *Front-End*, o relacionamento **3** é a comunicação com o *Core*. Nessa comunicação, o *Front-End* lê o catálogo de componentes disponíveis para construir a aplicação.

O relacionamento **4** é entre o *Front-End* do desenvolvedor e o *Core*. Atualmente, o *Front-End* usado é o HPE. A partir dele, são construídos os componentes que

irão executar nas plataformas paralelas. Os componentes desenvolvidos são então registrados no catálogo do *Core* para posterior uso pelo provedor de aplicações.

O relacionamento **5** diz respeito ao *Front-End* do mantenedor e o *Core*. Nesse caso, também tem como objetivo registrar componentes mas apenas aqueles da espécie plataforma. Ainda no mesmo *Front-End*, o relacionamento **6** diz respeito ao *Back-End*. Nesse relacionamento, as plataformas instanciadas no *Back-End* são gerenciadas pelo mantenedor de plataformas.

2.6 Contratos Contextuais

Nas discussões anteriores mencionou-se, como uma das tarefas delegadas ao *Core*, a implementação do sistema de contratos contextuais chamado Alite, responsável pela escolha das implementações dos componentes requisitados pelos *workflows* submetidos pelas aplicações. No entanto, não foi descrita a forma pelo qual os componentes requisitados são especificados pela linguagem de *workflow*, ou seja, a abstração de *contratos contextuais*, usados para essa finalidade.

A escolha da implementação de um componente para ser instanciado na execução de um *workflow* é feita através da análise das características da arquitetura da plataforma de computação paralela alvo e dos requisitos da aplicação com relação ao componente. Por exemplo, a plataforma de computação paralela alvo poderia ser equipada em suas unidades de processamento com aceleradores gráfico do tipo GPU, de um certo fabricante e modelo específico. Então, a prioridade de escolha seria por componentes que implementem algoritmos capazes de explorar a presença desse acelerador gráfico, usufruindo da sua capacidade de processamento.

Além da presença de aceleradores gráficos, outros exemplos de restrições ou requisitos arquiteturais que também podem influenciar a escolha de componentes que implementam algoritmos computacionais são: quantidade de memória disponível, podendo exigir componentes que demandem por menos memória para armazenar estruturas de dados em processamento, ou então que privilegiem o tempo de computação ao invés da economia do uso da memória; bibliotecas pré-instaladas, como bibliotecas de suporte a passagem de mensagens ou bibliotecas de subrotinas de computação científica pré-paralelizadas que podem auxiliar a realização de computações; a topologia da interconexão de rede que conecta as unidades de processamento, de modo a explorar melhor padrões de comunicação de algoritmos paralelos; os parâmetros de desempenho dessa interconexão, de modo a escolher algoritmos que não a sobrecarreguem; e a quantidade de unidades de processamento,

já que diferentes algoritmos possuem diferentes características de escalabilidade, de modo que o número de unidades de processamento disponíveis pode afetar o melhor algoritmo a ser utilizado para resolver um certo problema.

Por outro lado, exemplos de requisitos da aplicação que podem afetar a escolha da implementação de um componente podem ser: o tamanho do problema a ser resolvido, também devido às diferentes características de escalabilidade dos programas paralelos; a definição do tipo de estrutura de dados que será utilizada, de forma que sejam escolhidos componentes capazes de processar as propriedades dessa estrutura de dados; propriedades das estruturas de dados a serem processadas, como, por exemplo, no caso de matrizes, as quais podem ser densas ou esparsas e, neste último caso, obedecendo a padrões específicos em relação a localização dos elementos não-nulos (diagonal, tridiagonal, bloco-tridiagonal, pentadiagonal, etc); e a escolha de um certo algoritmo específico, por imposição da aplicação.

Contratos contextuais na nuvem HPC Shelf descrevem os requisitos como os descritos acima para escolha de componentes, de uma variedade de espécies, que os que atendam. Uma vez que plataformas de computação paralela também são representados por meio da abstração de componentes, contratos contextuais também são usados para caracterizar requisitos de plataformas que devem ser escolhidas para realizar computações de *workflows*.

O Alite é uma extensão do sistema de tipos HTS (*Hash Type System*) usado pelo HPE [Carvalho Junior et al. 2013]. No HTS, um contrato contextual é definido como um *componente abstrato* aplicado a um conjunto de *argumentos de contexto* que satisfazem os parâmetros de contexto especificados pelo componente abstrato, formando assim um *tipo de instanciação* que descreve as suposições de implementação de um componente implementado para satisfazer o interesse representado pelo componente abstrato.

A principal extensão sobre o HTS para a definição do sistema de contratos contextuais da HPC Shelf diz respeito a possibilidade de definir parâmetros de contexto com valorações numéricas, a fim de representar propriedades de contexto que assumem valores nesses domínios. No HTS, o domínio de parâmetros de contexto encontra-se restrito a um conjunto finito de tipos de instanciação, ordenados segundo a relação de subtipos.

Em um primeiro caso de uso para a extensão proposta ao HTS, um componente abstrato que representa um conjunto de componentes que implementam algoritmos paralelos para resolver um certo problema matemático pode incluir um parâmetro

de contexto que representa o número de unidades de processamento engajados na execução, de modo que podem coexistir componentes implementando algoritmos diferentes de acordo com a quantidade de unidades de processamento empregadas na computação paralela. Em um outro caso de uso vislumbrado, um componente abstrato combina parâmetros de contexto representando quantidade de memória necessária ao componente e a carga de trabalho a ser executada, a qual pode ser representado por um valor numérico, a fim de restringir sua escolha a plataformas capazes de suportar em sua memória as estruturas de dados criadas pelo algoritmo. A combinação desses dois parâmetros de contexto, por exemplo, podem ser utilizados para distinguir componentes conforme as características de escalabilidade dos algoritmos e programas paralelos que implementam [Grama et al. 2003].

É importante ressaltar que, por ser o Alite uma contribuição de uma outra Tese de Doutorado associado ao projeto HPC Shelf, ainda em conclusão, seus detalhes são propositalmente omitidos deste documento.

2.7 Considerações Finais

Esta Tese se preocupa com o módulo *Front-End* (mais especificamente o usuário especialista e provedor de aplicações), implementando o SAFe e sua linguagem de *workflow* SAFeSWL.

O SAFe é inspirado em um modelo de componentes conhecido como CCA. Os componentes acessados pelo *workflow*, também são baseados no CCA, implementados sobre a plataforma HPE. Esses componentes exportam portas no formato de *Serviços Web* as quais são acessadas de forma coordenada pelo SAFe, orquestrando assim a aplicação cujo alvo é o usuário especialista.

Em conjunto com os outros módulos da HPC Shelf, o *framework* proposto nesta Tese propicia um ambiente robusto para construção aplicações padronizadas as quais acessam componentes de alto desempenho. Além de se basear no modelo CCA, o SAFe apresentar um linguagem de construção de *workflows* caracterizada por dois subconjuntos: arquitetural e orquestração.

Capítulo 3

SAFe: O *Framework* de Aplicações da HPC Shelf

O SAFe (***S****helf* ***A****pplication* ***F****ramework*) é o *framework* para construção de aplicações da HPC Shelf, e constitui a principal contribuição desta Tese de Doutorado. Foi implementado em Java, como uma coleção de classes independentes a partir das quais uma aplicação pode ser derivada. A aplicação é construída fazendo uso de abstrações de alto nível que são mapeadas em sistemas computacionais constituídos pela composição de componentes paralelos disponibilizados pela plataforma. Esses componentes são orquestrados através de um *workflow* descrito por meio de uma linguagem própria, através da qual é possível expressar a arquitetura de componentes de um sistema computacional e um fluxo de execução para as ações oferecidas pelos componentes de computação e conectores que dele fazem parte. Essa linguagem é chamada de SAFeSWL, um acrônimo para SAFe *Scientific Workflow Language*.

O SAFe não impõe ao provedor de aplicações a forma como uma aplicação deve ser apresentada ao especialista. Ela pode ser uma aplicação *web*, uma solução *desktop*, um *plug-in* para o Eclipse, um aplicativo Android/iOS, ou até mesmo um programa acessado via linha de comando.

Este capítulo irá apresentar detalhes da arquitetura, projeto de classes e ciclo de vida de uma aplicação projetada no SAFe. Para fins didáticos, iremos demonstrar um exemplo simples de aplicação compatível com o SAFe, mas que é completa o suficiente para exercitar as suas características e funcionalidades mais importantes. Outros exemplos mais completos e realísticos serão apresentados no Capítulo 4.

3.1 Características e Requisitos do SAFe

O SAFe tem por objetivo prover uma interface para que a aplicação possa acessar os recursos da HPC Shelf, ou seja, a criação, instanciação e execução de *workflows* científicos baseados em componentes paralelos, gerados para a solução computacional de problemas especificados pelo especialista através da interface disponibilizada pela aplicação. Desse modo, para o especialista, o usuário final da HPC Shelf, basta apenas que a aplicação apresente uma interface de alto nível voltada a resolver problemas no seu domínio de interesse.

Para atingir esse objetivo, SAFe implementa:

- um conjunto de classes a partir das quais aplicações capazes de usar os recursos da HPC Shelf podem ser derivadas;
- uma linguagem de especificação de *workflows*, chamada de SAFeSWL, baseada em XML e com uma gramática modular, que divide os subconjuntos arquitetural e de orquestração da linguagem;
- uma interface para o desenvolvimento de aplicações destinada aos provedores de aplicações, sendo sua responsabilidade o mapeamento da especificação de problemas descritos pelo especialista por meio da interface da aplicação para soluções computacionais (*workflows*) especificadas na linguagem SAFeSWL;
- mecanismos para monitoramento e intervenção na execução de soluções computacionais, através de meios que permitam a interação entre a aplicação e várias soluções computacionais em execução (e.g. configuração/reconfiguração de parâmetros e coleta de informações em tempo real);
- meios para controle explícito das fases do ciclo de vida dos componentes de uma solução computacional (resolução, implantação, instanciação e liberação), tendo em vista os custos potenciais de tempo associados a cada uma dessas fases, bem como os custos financeiros potenciais associados ao uso dos componentes, uma vez instanciados, em aplicações de longa duração na HPC Shelf, de modo que componentes possam ser resolvidos, implantados, instanciados e liberados em fases distintas e à medida que sejam necessários (sob demanda), paralelamente à execução de computações em progresso.

Para o provedor construir aplicações, é necessário conhecer apenas a API do SAFe e a linguagem SAFeSWL. Preconiza-se ao provedor que o SAFe deve ser transparente

em relação à aplicação final oferecida para o usuário especialista, de modo que não se preocupe, na descrição de problemas, com detalhes técnicos da HPC Shelf que sejam alheios a descrição do problema ou mesmo à solução computacional que será construída pela aplicação. Uma vez declarado um problema e submetido à aplicação, cabe ao especialista apenas acompanhar a execução, possivelmente interagindo com o *workflow*, dependendo dos requisitos da aplicação.

O SAFe deve se comunicar com o *Core* via Serviços *Web*, garantindo transparência de localização, a fim de tornar *workflows* gerados por uma aplicação prontos para execução, estado no qual estão acessíveis os serviços das portas de ambiente e de tarefa de componentes do *workflow* que estejam conectadas à aplicação. Deve-se recordar que o *Core* é responsável pelo controle de todo o ciclo de vida de um componente da HPC Shelf.

Como foi discutido no capítulo anterior, o SAFe faz uso dos serviços do *Core* em dois momentos do ciclo de vida de uma aplicação:

i. durante a criação da aplicação;

ii. na execução das soluções computacionais geradas pela aplicação.

Durante a criação da aplicação, o provedor de aplicações fará uso de seus conhecimentos em computação para estender as classes do SAFe com o intuito de gerar um interface amigável para o usuário especialista. A aplicação, uma vez finalizada, é capaz de gerar *workflows* em SAFeSWL para descrever soluções computacionais com o intuito de resolver os problemas descritos pelo especialista através da interface de alto nível da aplicação. Para isso, o provedor de aplicações irá acessar os componentes abstratos publicados no catálogo mantido pelo *Core*, a fim de gerar os *contratos contextuais* que guiarão a escolha de implementações de componentes adequadas para os requisitos da aplicação e as características arquiteturais das plataformas de computação paralela que serão engajadas para a execução das soluções computacionais.

Por sua vez, durante a execução da aplicação, o usuário especialista interage com os sistemas computacionais (*workflows*) criados pela aplicação em SAFeSWL a partir dos problemas declarados pelo especialista. Para essa interação, a aplicação deve fornecer uma interface adequada, de alto nível de abstração, que deve se comunicar com os componentes dos *workflows* em execução por meio de suas portas de ambiente. A interpretação desses *workflows* consiste na execução do código de

orquestração descrito no código SAFeSWL. Nesse intervalo, entre o início e o término de cada *workflow*, o SAFe acessa os serviços do *Core*, de forma transparente para a aplicação, com o objetivo de realizar as seguintes operações:

- A ***resolução*** aplica-se sobre os contratos contextuais que representam os componentes da arquitetura do *workflow*, os quais são submetidos pelo sistema de execução do *workflow* ao *Core*. Então, o *Core* aplica o algoritmo de resolução implementado pelo sistema de contratos contextuais (Alite). Como resultado, é retornada uma lista de possíveis candidatos que satisfazem às restrições impostas pelo contrato, ordenada segundo critérios de qualidade propostos pelo *Core*. A aplicação deve então selecionar o candidato adequado dentre os sugeridos pelo *Core*, incluindo outros critérios de qualidade próprios que reordenem a lista, ou aceitar a sugestão padrão do *Core*.

- A ***implantação*** aplica-se sobre uma referência para um componente previamente escolhido como resultado da *resolução* de um contrato contextual, o qual é submetido ao *Core*. Então, o *Core* instala o componente na plataforma virtual onde será posteriormente instanciado, resolvendo todas as suas dependências (componentes e bibliotecas).

- A ***instanciação*** aplica-se sobre uma referência a um componente previamente implantado em uma plataforma virtual, como resultado de uma operação de *implantação*, o qual é submetido ao *Core*. Então, o *Core* comunica-se com a plataforma virtual para criar uma instância do componente, deixando-o pronto para ser orquestrado pela aplicação através de suas portas de tarefa. Além disso, a partir daí, a aplicação tem acesso às suas portas de ambiente, para interação em tempo de execução. Note que a plataforma virtual deve ser instanciada, a partir de um componente plataforma, antes da implantação de qualquer componente que dela depende.

- A ***liberação*** aplica-se a uma referência de um componente previamente instanciado em uma plataforma virtual, o qual é submetido ao *Core* para liberação dos recursos do componente após não ser mais necessário na execução do *workflow*. No caso de instâncias de componentes plataforma, a plataforma virtual pode ser mantida pelo *Back-End*, ao invés de liberada, possivelmente para atender alguma outra aplicação em um momento futuro, ou mesmo a

própria aplicação, que não precisará instanciar uma nova plataforma caso já existir uma outra de mesmo perfil previamente em execução no *Back-End*.

3.2 Sistemas Computacionais SAFe

Um sistema computacional em SAFe representa todos os componentes responsáveis pela solução de um problema descrito pela linguagem SAFeSWL, bem como os seus relacionamentos através de *bindings* de ambiente e de tarefas. Em um sistema computacional, além dos componentes responsáveis pela solução propriamente dita do problema, doravante chamados de *componentes de solução*, os quais podem ser das espécies *computação*, *conector*, *fontes de dados* e *plataforma*, temos o componente *aplicação*, o qual interage com as portas de ambiente dos outros componentes para fins de monitoração e troca de dados de entrada, saída e intermediários, e o componente *workflow*, o qual controla o ciclo de vida dos demais componentes de solução propriamente ditos.

Como exemplo ilustrativo, a Figura 3.1 apresenta um sistema computacional para a aplicação Map/Reduce, que será apresentada como estudo de caso para validação do SAFe no Capítulo 4. O componente aplicação expõe portas de ambiente do tipo usuárias, as quais se conectam às portas provedoras de outros componentes, e também expõe portas de ambiente do tipo provedoras, as quais fornecem serviços aos outros componentes do sistema computacional. O componente *workflow* tem como propósito mais proeminente conectar-se às portas de tarefas dos componentes de computação e conectores, as quais publicam ações que são ativadas de acordo com um fluxo de orquestração especificado por meio da linguagem SAFeSWL.

Além dos componentes aplicação e *workflow*, podemos notar, na Figura 3.1, dois *conectores*, representados por instâncias dos componentes SHUFFLER e SPLITTER; três computações, representados pelos componentes MAPPER e REDUCER (o qual é usado em dois momentos distintos); e duas fontes de dados, representadas pelos componentes DATASINK e DATASOURCE. Além disso, há duas plataformas (PA e PB), sobre as quais encontram-se alocados os componentes MAPPER e REDUCER, bem como, implicitamente, as facetas dos conectores SHUFFLER e SPLITTER voltadas para esses componentes, respectivamente, por meio de ligações de ambiente.

Enfim, o sistema computacional executará sobre um conjunto de plataformas virtuais (remotamente ao SAFe). Plataformas virtuais, instanciadas a partir de componentes plataforma, representam plataformas de computação paralela criadas no *Back-End* da HPC Shelf, de acordo com as necessidades do usuário especialista

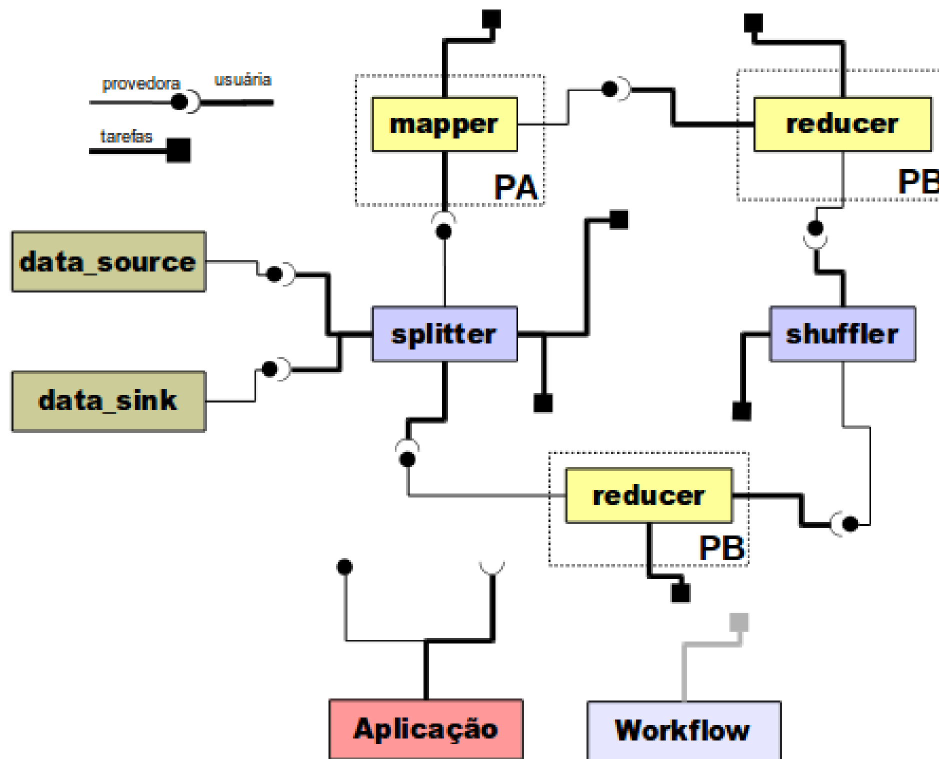


Figura 3.1: Arquitetura de um sistema computacional para o Map/Reduce no SAFe.

expressas em seus contratos contextuais (perfis de plataforma).

3.3 A Arquitetura da Implementação do SAFe

A Figura 3.2 apresenta a visão geral do *framework* SAFe e seus dois principais pacotes de classes: o `safe-framework`, para construção de aplicações, e o `safe-language`, referente à implementação da linguagem SAFeSWL. Além disso, a figura também ilustra a participação de dois importantes atores do ciclo de vida de uma aplicação: o especialista e o provedor de aplicações.

O provedor de aplicações trabalha diretamente com o pacote `safe-framework`, através do qual ele deriva uma aplicação compatível com o grau de abstração necessária para uso do especialista. A aplicação criada ficará em outro pacote. No exemplo ilustrado, chama-se `concrete-application-project`. O usuário especialista, através da interface implementada pelo provedor de aplicações, acessa esse pacote para especificação de problemas e acompanhamento da execução das soluções computacionais implementadas pelos *workflows* gerados pela aplicação. Teoricamente, o usuário especialista não tem ideia da arquitetura de execução que funciona sob sua aplicação.

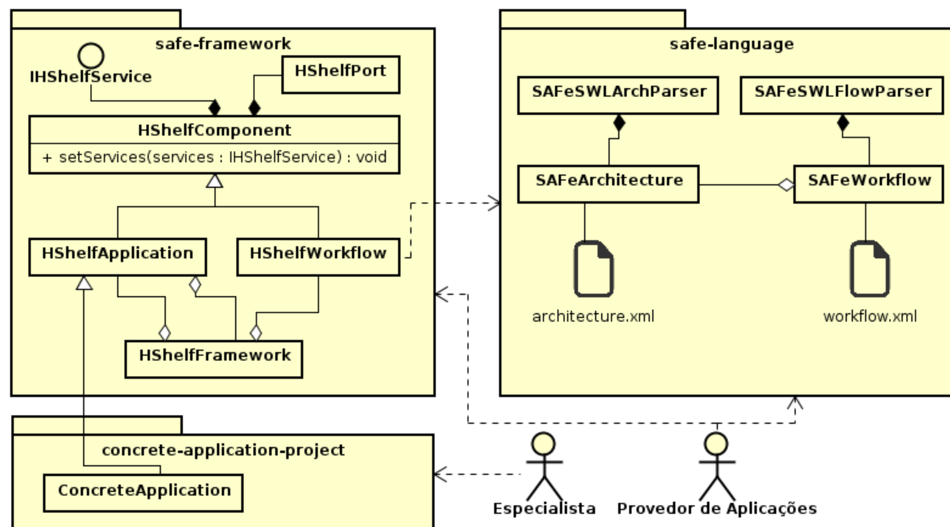


Figura 3.2: Visão Arquitetural do SAFe.

Nas seções a seguir, são apresentados detalhes sobre a implementação dos pacotes `safe-framework` e `safe-language`.

3.3.1 O Pacote `safe-framework`

O pacote `safe-framework` agrupa as principais classes e interfaces para a construção de aplicações baseadas em componentes e suas portas. A forma como componentes trocam informações entre si foi inspirada no modelo CCA [Armstrong et al. 2006]. O modelo CCA foi introduzido rapidamente no Capítulo 1. Com o intuito de facilitar o entendimento do SAFe, faz-se necessária uma explicação mais detalhada.

O Modelo CCA

No modelo CCA, um componente implementa uma interface `Component`, a qual exige a implementação de um método chamado `setServices`, através do qual o *framework* passa para o componente um objeto do tipo `Services`, doravante denominado *services*. O objeto *services* serve para a comunicação do componente com o *framework* CCA. Através dele, o componente informa suas portas provedoras (*provides ports*), as quais expõem os serviços que oferecem para os outros componentes, e suas portas usuárias (*uses ports*), as quais determinam suas dependências por serviços oferecidos por outros componentes.

Na Figura 3.3, há dois componentes compatíveis com o modelo CCA, cada qual possuindo um objeto `services`. O objetivo é demonstrar o compartilhamento do serviço de um objeto de um subtipo da interface `Port` (`MyPortType`), doravante

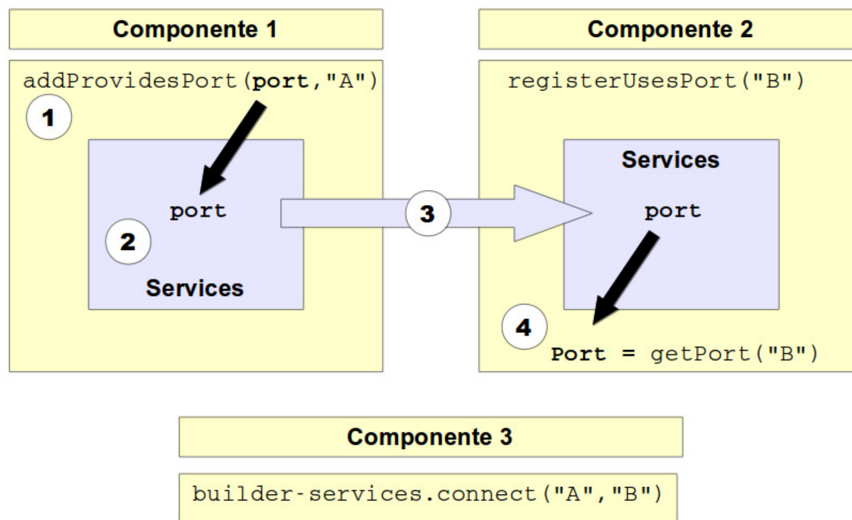


Figura 3.3: Componentes CCA.

chamado de *port*, do *primeiro componente* (**Componente 1**), através da porta provedora referenciada por "A", com o *segundo componente* (**Componente 2**), por meio de uma porta usuária de nome "B".

O primeiro componente registra a disponibilidade do objeto *port*, através do método `addProvidesPort`, pertencente ao objeto `Services` (passo 1). Fazendo isso, a porta é registrada no catálogo de portas do *framework* sob o nome escolhido no momento da chamada do método. No caso, o identificador escolhido para a porta é "A" (passo 2). O segundo componente, por sua vez, registra seu interesse por um objeto do tipo `MyPortType`, ou seja, do mesmo tipo do objeto *port* publicado pelo primeiro componente, através do método `registerUsesPort`, passando o tipo da porta usuária requisitada (`MyPortType`) e um nome para essa porta (passo 3). O nome escolhido é "B". Esse método também faz parte da interface do objeto *services* do segundo componente.

Note que o segundo componente ainda não recebeu nenhuma referência ao objeto *port* do primeiro componente. De fato, ainda não foi feita qualquer relação entre a porta usuária do segundo componente com a porta provedora do primeiro. A conexão entre as duas portas é feita por um terceiro componente (**Componente 3**), através da sua conexão a uma porta de serviço disponibilizada pelo *framework*, do tipo `BuilderServices`, doravante chamada *builder services*. Esse terceiro componente deve conectar-se a porta *builder services* através de uma chamada a `registerUsesPort`, para registar a porta usuária, seguida de uma chamada a `getPort`, para pegar a referência ao objeto *builder services*. Por ser uma porta de

serviço do *framework*, a conexão à porta *builder services* é feita automaticamente. De posse do objeto *builder services*, o terceiro componente invoca o seu método `connect` passando como argumentos as referências às portas "A" e "B" previamente registradas pelo primeiro e segundo componentes, respectivamente, cujo efeito é registrar uma ligação entre as duas portas.

A próxima etapa (passo 4) para o segundo componente é fazer uso da porta no primeiro componente, associada pelo terceiro componente. Para isso, o segundo componente invoca o método `getPort`, passando o nome da sua porta usuária ("B"). Uma vez que já tenha sido ligada a uma porta provedora do mesmo tipo, no caso a porta "A", a referência ao objeto do primeiro componente é retornada. Caso contrário, o método `getPort` bloqueia até que a conexão seja realizada por um componente que faça o papel do terceiro componente do exemplo ilustrativo.

CCA é um modelo arquitetural. Um *framework* de componentes compatível com o CCA é responsável por implementar, da forma apropriada para o ambiente de execução, as interfaces que tornam possível a conexão entre os componentes mutuamente interessados nas suas portas. Além disso, nada impede que dois componentes sejam implementados em linguagens totalmente diferentes entre si ou que estejam localizados em domínios de rede distintos (componentes distribuídos), sendo o *framework* responsável pela intermediação das trocas de dados entre os componentes através de suas portas. É também responsabilidade de cada *framework* definir como lidar com aspectos de paralelismo, muito embora o Fórum CCA já tenha proposto uma interface de suporte ao paralelismo que deveria ser suportada por *frameworks* CCA, chamada MPIServices [Forum 2009].

Implementações importantes do modelo CCA são: CCAffeine [Allan et al. 2002], XCAT2 [Govindaraju et al. 2002], SciRun2 [Zhang et al. 2004], MOCCA [Malawski, Kurzyniec e Sunderam 2005], DCA [Bertrand e Bramley 2004], dentro outros, incluindo o próprio HPE, precursor da plataforma HPC Shelf, mencionado no Capítulo 2, o qual propõe uma noção de componente paralelo baseada no modelo Hash e compatível com o modelo CCA [Carvalho Junior e Rezende 2013].

O SAFe e o CCA

O SAFe pode ser visto como um *framework* de componentes. Porém, não tem a intenção de ser um *framework* compatível com o modelo CCA, mas busca utilizar alguns de seus padrões de projeto para comunicação entre componentes, adaptando-os conforme as suas peculiaridades.

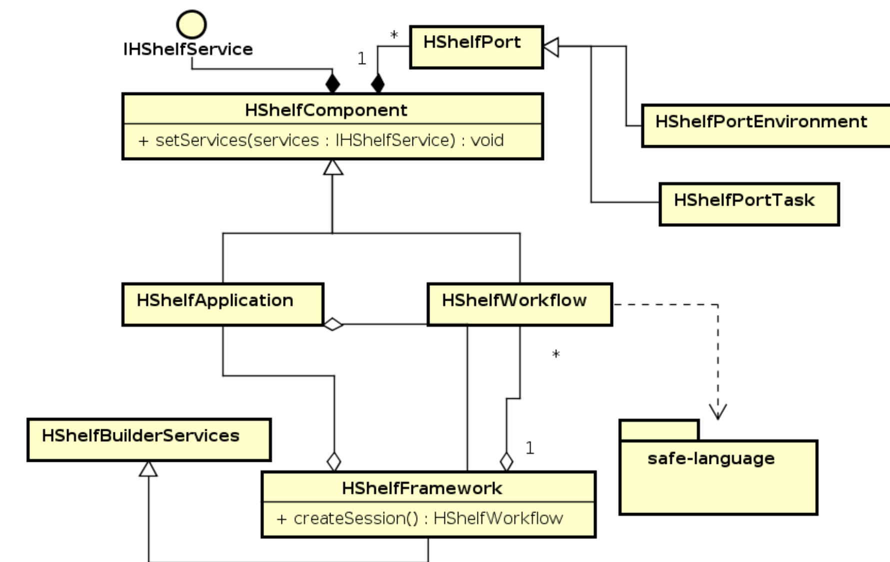


Figura 3.4: Relacionamento entre as Classes do Pacote `safe-framework`.

Assim como o CCA, os componentes que constituem os *workflows* executados pelo SAFe também recebem do *framework* um objeto *services*, responsável pela sua comunicação com a aplicação e, consequentemente, com o próprio SAFe. Através desse objeto, portas usuárias e provedoras podem ser registradas. Para isso, um *workflow* SAFeSWL possui dois componentes especiais, o ***componente aplicação*** e o ***componente workflow***, os quais separam entre si os interesses do SAFe. O primeiro representa a própria aplicação, interessada em interagir com os componentes de solução, enquanto o último representa o motor de execução do *workflow*, que controla o ciclo de vida dos componentes. A comunicação dos componentes aplicação e *workflow* com os componentes de solução, instanciados em plataformas virtuais em domínios remotos, é feita por intermédio dos componentes *binding*, de modo que o provedor de aplicação não precisa necessariamente preocupar-se como ela é realizada. Atualmente, os componentes *binding* que ligam a aplicação aos componentes de solução, remotamente distribuídos, usufruem da tecnologia de Serviços *Web* em sua implementação.

Sendo assim, a especificação do CCA serviu como um guia para melhores práticas de implementação de interfaces e padrões de comunicação entre componentes distribuídos implementados em tecnologias diferentes. Ao longo do resto deste capítulo, será melhor detalhada a forma como esses componentes são implementados e disponibilizados, bem como as particularidades do SAFe que motivam adaptações ao modelo CCA.

As Classes e Interfaces do Pacote `safe-framework`

Os principais membros do pacote `safe-framework` são:

- **Classe `HShelfFramework`**: representa o objeto *framework*, no sentido empregado pelo CCA, ou seja, o ambiente que controla os componentes em execução e suas ligações através das portas. Cada sistema computacional criado por uma aplicação derivada a partir do SAFe cria um objeto do tipo *framework* o qual centraliza a informação sobre todos os componentes e suas respectivas portas. Esse objeto é o responsável pelo compartilhamento de portas registradas por cada componente através de seu objeto *services*;

- **Interface `IHShelfService`**: é o tipo dos objetos *services*, analogamente ao CCA, ou seja, aqueles responsáveis pela comunicação dos componentes com o *framework*, com o propósito de registrar portas provedoras e usuárias.

- **Classe abstrata `HShelfComponent`**: representa a implementação de componentes da descrição arquitetural de *workflows* do SAFe. Possui um único método a ser sobrescrito, denominado `setServices`, através do qual o *framework* fornece o objeto *services*.

- **Classe abstrata `HShelfApplication`**: representa a base para derivações de aplicações da HPC Shelf. Ela deve ser implementada, através de herança, pelo provedor de aplicações. Tendo em vista que `HShelfApplication` é uma subclasse de `HShelfComponent`, um objeto instanciado a partir de `HShelfApplication` representa o *componente aplicação*, referenciado na descrição arquitetural de um sistema computacional descrito com SAFeSWL. Através do objeto *services*, recebido do *framework* por meio do método sobrescrito `setServices`, a aplicação pode conectar-se às portas de ambiente de outros componentes da arquitetura de um ou mais *workflows* em execução.

- **Classe `HShelfWorkflow`**: é o tipo dos objetos que implementam os interesses relativos à interpretação do código SAFeSWL dos *workflows*. Correspondem à parte do SAFe que não precisa ser estendida para derivação da aplicação. Assim como `HShelfApplication`, também é subclasse de `HShelfComponent`, de modo que seus objetos representam os componentes *workflow* anteriormente mencionados. O componente *workflow* comunica-se com o componente aplicação através de portas de ambiente padrões. Por

IHShelfService	
setProvidesPort (HShelfPort port)	Registra um objeto que representa uma porta de ambiente através da qual o componente expõe serviços para outros componentes.
registerUsesPort (String name, String tipo)	Registra o interesse pela conexão a uma porta de ambiente de um certo tipo informado provida por algum outro componente.
registerTaskPort (String name, HShelfPort port)	Registra uma porta de tarefas para sincronização de ações com um outro componente.
waitPort(String name)	Aguarda até que seja realizada uma conexão à uma porta usuária ou porta de tarefas previamente registrada com registerUsesPort ou registerTaskPort, respectivamente, a uma outra porta de mesma natureza e tipo.
testPort(String name)	Retorna o valor booleano "verdade" até que seja realizada uma conexão à uma porta usuária ou porta de tarefas previamente registrada com registerUsesPort ou registerTaskPort, respectivamente, a uma outra porta de mesma natureza e tipo, ou "falso", caso contrário.
getPort(String name)	Caso já exista uma conexão à uma porta usuária ou porta de tarefas previamente registrada com registerUsesPort ou registerTaskPort, respectivamente, retorna uma referência para a porta parceira, ou lança uma exceção, caso contrário.

Tabela 3.1: Operações da Interface IHShelfService

exemplo, com a finalidade de receber os códigos arquitetural e de orquestração em SAFeSWL. A parte arquitetural oferece a informação a partir do qual o componente *workflow* pode comandar a resolução, a implantação, a instanciação e a ligação entre componentes, à medida que interpreta a parte de orquestração. A orquestração das ações expostas pelas portas de tarefa dos componentes é possível através das portas de tarefa que registra à medida que os componentes de computação e conectores são instanciados.

- **Classe HShelfPort**: deve ser estendida na derivação de uma aplicação para representar cada tipo de porta que a interessa, seja de ambiente (subclasse HShelfPortEnvironment), usuária ou provedora, ou de tarefas (subclasse HShelfPortTask).

As tabelas 3.1, 3.2 e 3.3 resumem as principais operações das classes do pacote safe-framework. O provedor de aplicações deve ter um conhecimento da API do SAFe. Com o objetivo de derivar uma aplicação, a partir do SAFe, deve seguir

HShelfComponent	
setServices (IHShelfService services)	Esse método deve ser sobrescrito pelo componente. Através do objeto services, passado como parâmetro, o componente pode registrar portas provedoras, portas de tarefas e também notificar/registrar que necessita usar uma outra porta.

Tabela 3.2: Operações da Classe HShelfComponent

HShelfFramework via HShelfBuilderServices	
createComponent (String name, String className)	Cria um componente por reflexão. Método usado pelo *workflow* para instanciação automática de componentes.
List getProvidesPortList()	Lista todas as portas provedoras disponíveis no *framework*. Portas que foram registrados por todos os componentes que fazem parte da aplicação.
List getTaskPortList()	Lista todas as portas de tarefas disponíveis no *framework*. Portas que foram registrados por todos os componentes que fazem parte da aplicação.
addComponent (HShelfComponent component)	Adiciona um componente ao *framework*. No momento que um componente é adicionado ao *framework*, um objeto *services* é criado e repassado a ele.

Tabela 3.3: Operações da Classe HShelfBuilderServices

os seguintes passos:

i. Derivar a classe que representa a aplicação por herança a partir da classe HShelfApplication. Ao herdar dessa classe, o provedor está criando um componente compatível com o SAFe, pois HShelfApplication é subtipo de HShelfComponent. Sendo subtipo de HShelfComponent, uma aplicação também deve obrigatoriamente sobrescrever o método setServices, onde serão expostas as suas portas provedoras e usuárias.

ii. Implementar as classes que implementam as interfaces correspondentes às portas de ambiente provedoras do componente aplicação.

iii. Sobrescrever o método setServices da classe derivada a partir de HShelfApplication no primeiro passo para receber o objeto *services*, que será usado para registro de suas portas provedoras e usuárias para o *framework*, as quais serão posteriormente ligadas, por *bindings* de ambiente, pelo componente *workflow*.

iv. Implementar uma interface para a sua aplicação de acordo com as necessidades do especialista, a qual pode ser gráfica ou textual (linguagem de propósito

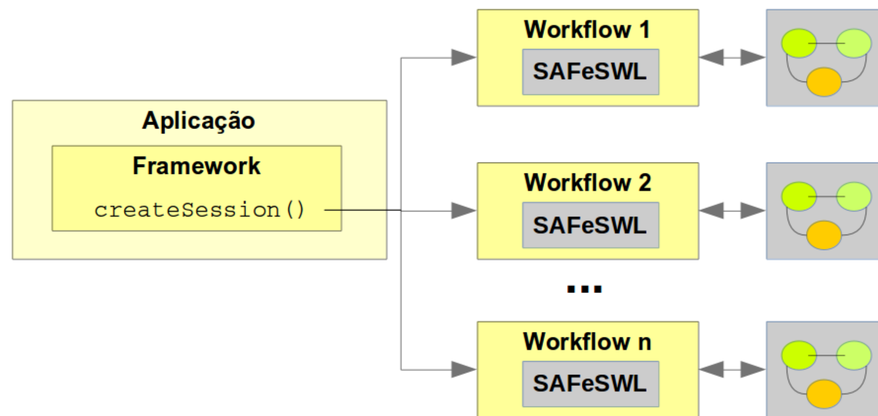


Figura 3.5: A criação de sessões por parte do Framework de uma Aplicação. Uma mesma aplicação pode fazer parte de várias sessões, cada uma com um único Workflow a qual orquestra um conjunto de componentes.

especial), visando uma aplicação *stand-alone* para computadores *desktop* ou uma aplicação *web*, ou até mesmo uma interface baseada em linha de comando. Em qualquer situação, a interface serve tanto para a declaração de problemas quanto para o acompanhamento e interação com a execução dos *workflows* que representam as soluções computacionais para esses problemas.

De acordo com a Figura 3.4, um objeto do tipo `HShelfApplication` instancia apenas um único objeto do tipo `HShelfFramework`, ou seja, o *framework* da aplicação criada pelo provedor de aplicações. A classe `HShelfFramework`, por sua vez, oferece o método `createSession` através de sua porta *builder services*, cuja invocação cria uma nova *sessão*, criando um objeto do tipo `HShelfWorkflow`. Cada objeto *workflow* corresponde a uma solução computacional distinta, para um problema distinto ou até mesmo para um mesmo problema, de acordo com o interesse da aplicação. Portanto, cada sessão é formada inicialmente pelo componente aplicação e por um novo componente *workflow*. O componente aplicação pode então conectar-se às portas do componente *workflow* através das quais pode submeter o código SAFeSWL e comandar o início da execução do *workflow*. Os componentes descritos na descrição arquitetural do *workflow* são instanciados e conectados, por *bindings* de ambiente, à medida que as operações de resolução, implantação e instanciação são executadas na orquestração do *workflow*, através da porta *builder services* do *framework*. O mesmo componente aplicação pode fazer parte de várias sessões simultâneas, cada uma com um único componente *workflow* distinto.

A Figura 3.5 ilustra o conceito da criação de sessões, onde cada sessão pertence a uma aplicação. Cada sessão é formada pelo componente aplicação e seu respectivo componente *workflow*. Cada componente *workflow* possui a sua própria descrição de um sistema computacional na linguagem SAFeSWL, a qual cuida da orquestração de um conjunto de componentes.

Ainda na Figura 3.4, o *workflow* comunica-se diretamente com o pacote responsável em manipular a linguagem SAFeSWL (pacote `safe-language`). Esse pacote, também implementado em Java, será explicado com mais detalhes adiante.

3.3.2 O Pacote `safe-language`

O pacote `safe-language` reúne um conjunto de classes responsáveis por interpretar código escrito na linguagem SAFeSWL, como detalhado na Seção 3.4. A linguagem SAFeSWL é representada por duas gramáticas distintas: uma que representa o seu subconjunto arquitetural e outra que representa o seu subconjunto de orquestração. Na implementação atual do SAFe, ambas são representadas por gramáticas XSD (*XML Schema Definition*), usadas para validação dos códigos XML, arquitetural e de orquestração, respectivamente, submetidos ao componente *workflow* pelo componente aplicação.

O subconjunto arquitetural define quais são os componentes que fazem parte do *workflow*, suas portas de ambiente (provedores e usuárias) e de tarefas e como essas portas estão relacionadas entre si através de *bindings*, notadamente de ambiente e de tarefas. O subconjunto de classes associado ao subconjunto arquitetural de SAFeSWL tem a responsabilidade de ler e interpretar o arquivo XML onde se encontra a especificação arquitetural do *workflow*.

O subconjunto de orquestração define o fluxo de controle segundo o qual as ações das portas de tarefas dos componentes especificados por meio do subconjunto arquitetural serão orquestradas. As classes associadas ao subconjunto de orquestração de SAFeSWL tem a responsabilidade de ler e interpretar o arquivo XML onde se encontra a especificação do código de orquestração de um *workflow*.

Sendo assim, o pacote `safe-language` subdivide-se em dois pacotes de classes: `grammar.arch` e `grammar.flow`. O primeiro trata do subconjunto arquitetural, enquanto o último trata do subconjunto de orquestração.

Classes do Pacote `grammar.arch`

As principais classes desse pacote são:

- **Classe `SAFeSWLArchParser`**: recebe em seu construtor o caminho para o

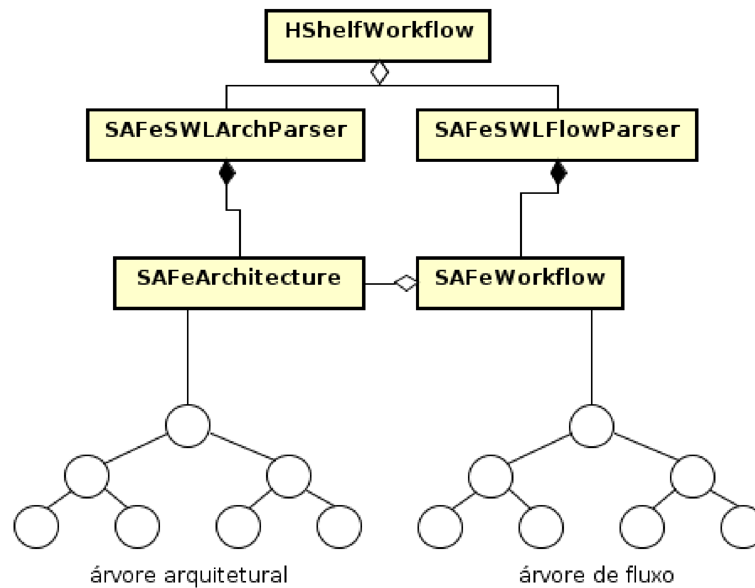


Figura 3.6: Relacionamento, em alto nível, entre classes do pacote de linguagem.

arquivo XML relativo a descrição arquitetural de um sistema computacional descrito por um *workflow* gerado pela aplicação. Sua função é ler o arquivo XML e gerar diversos objetos Java que representam cada uma das *tags*. Serão gerados, por exemplo, objetos que representam componentes, portas e seus relacionamentos. Dessa forma, a manipulação da informação arquitetural será mais facilmente realizada na execução da orquestração.

- **Classe `SAFeArchitecture`**: representa o objeto raiz da descrição arquitetural. É formado por diversos outros objetos que também representam objetos filhos da arquitetura: o componente aplicação, o componente *workflow* e cada um dos componentes de solução, bem como os *bindings* que definem os relacionamentos entre as suas portas de ambiente e de tarefas.

Classes do Pacote `grammar.flow`

As principais classes desse pacote são:

- **`SAFeSWLFlowParser`**: recebe em seu construtor o caminho para o arquivo XML relativo a lógica de orquestração do *workflow* gerado pela aplicação. Assim como o seu análogo ao arquivo arquitetural, essa classe também lê o arquivo XML de orquestração e o interpreta. Cada *tag* é transformada em uma classe separada, com o seu significado próprio.

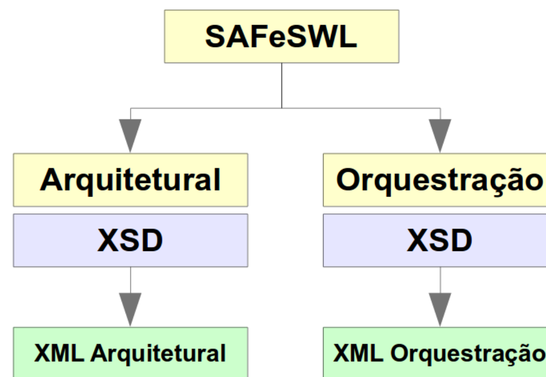


Figura 3.7: A Linguagem SAFeSWL e seus subconjuntos.

- **`SAFeWorkflow`**: representa um nó da árvore sintática concreta do código de orquestração gerada pela classe anterior. Usando o padrão *Visitor* [Schmidt et al. 1996], essa classe é executada a partir do nó raiz, percorrendo-se seus nós recursivamente e executando a sua lógica. Por exemplo, caso o nó represente um sequenciamento de ações (*tag* `sequence`), todas as ações representadas nas suas sub-árvores serão executadas em sequência, uma após a outra.

A Figura 3.6 mostra o relacionamento entre as principais classes do pacote `safe-language`. O objeto que inicia o processamento dos arquivos XML referentes aos códigos arquitetural e de orquestração de SAFeSWL é da classe `HShelfWorkflow`, representando o componente *workflow*. Ele irá receber, do componente aplicação, os arquivos em XML e os repassará para objetos das classes de análise arquitetural e de orquestração, `SAFeSWLArchParser` e `SAFeSWLFlowParser` respectivamente. Após validar os códigos através das gramáticas XSD apropriadas, geram as estruturas de dados que serão acessadas durante a execução do *workflow*. A primeira estrutura gerada é uma tabela de símbolos que contém as informações arquiteturais. A segunda estrutura gerada é uma árvore de representação intermediária que será interpretada usando o padrão *Visitor*, executando a lógica do fluxo de controle do *workflow* e consultando a tabela de símbolos arquitetural sempre que necessário. A estrutura lembra a de um interpretador convencional de uma linguagem de programação.

3.4 A Linguagem SAFeSWL

Como dito anteriormente, a linguagem SAFeSWL é formada por dois subconjuntos que definem as descrições arquiteturais e de orquestração de *workflows* da HPC Shelf, como ilustrado na Figura 3.7. Para os propósitos do SAFe, a sintaxe

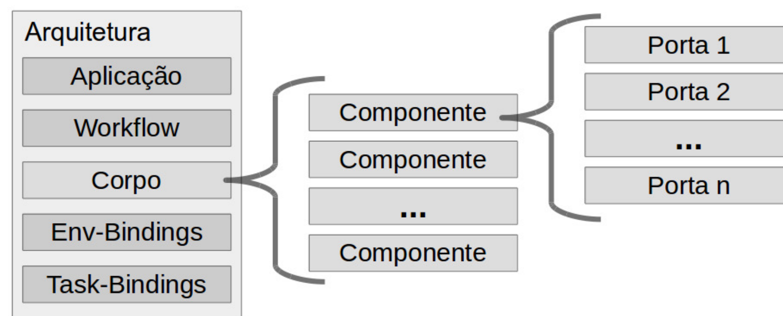


Figura 3.8: Visão abstrata da linguagem arquitetural do SAFeSWL.

de ambos encontra-se especificada em gramáticas XSD distintas de forma a serem representadas através do formato de intercâmbio de dados XML.

O objetivo do subconjunto arquitetural é apresentar quais são os componentes da solução computacional descrita por meio do *workflow*, bem como a forma como os mesmos estão relacionados através de *bindings* que conectam suas portas de ambiente e de tarefas. Por sua vez, o subconjunto de orquestração da linguagem define construtores que especificam o fluxo de execução das ações exportadas por portas de tarefas dos componentes de computação e conectores. As seções seguintes apresentam, com mais detalhes, as características de cada subconjunto da linguagem bem como os propósitos de seus elementos. Detalhes técnicos adicionais serão encontrados nos apêndices A (subconjunto arquitetural) e B (subconjunto de orquestração).

3.4.1 O Subconjunto Arquitetural

O subconjunto arquitetural da linguagem SAFeSWL tem como objetivo a especificação do conjunto de componentes e ligações entre eles (*bindings*) que serão orquestrados na execução de uma solução computacional gerada por uma aplicação. Essas informações são essenciais para o subconjunto de orquestração da linguagem que, ao executar, precisará consultar as informações arquiteturais do *workflow*.

A Figura 3.8 ilustra o corpo de uma descrição arquitetural. Essa representação abstrata é implementada na forma de uma gramática no formato XSD, a ser apresentada posteriormente. Os principais elementos, presentes na figura, são:

- **Arquitetura**: é o elemento principal, que engloba todos os outros componentes. Não tem valor semântico para a linguagem de orquestração, sendo exigido apenas para representar a *tag* raiz do arquivo XML.

- **Aplicação**: é o elemento que representa o *componente aplicação*. Por ser um

componente, engloba elementos que representam portas de ambiente.

- **Workflow**: é o elemento que representa o *componente workflow*. Por ser um conector especial, que define a orquestração geral dos componentes de solução do *workflow*, engloba elementos que representam tanto portas de ambiente quanto de tarefas.
- **Corpo**: é o elemento que engloba o conjunto de componentes de solução, os quais estarão executando remotamente, em plataformas virtuais oferecidas pelos mantenedores. As portas podem ser de ambiente ou de tarefas. No último caso, somente para componentes de computação e conectores.
- **Env-Bindings**: é o elemento que indica uma ligação entre duas portas de ambiente, uma provedora e outra usuária, de tipos compatíveis.
- **Task-Bindings**: é o elemento que indica uma ligação entre duas portas de tarefa parceiras compatíveis (mesmo conjunto de ações).

3.4.2 O Processamento de uma Especificação Arquitetural

A gramática XSD valida o arquivo XML que representa a descrição arquitetural. Como já dito, o pacote `safe-language` é responsável pela leitura do arquivo fonte do XML. Internamente, cada elemento é convertido em um objeto Java. Esses objetos irão armazenar as informações do XML e serão usados pela linguagem de orquestração quando o *workflow* for executado.

A Figura 3.9 apresenta o diagrama de classes responsáveis pela leitura do arquivo arquitetural. A classe que inicia a leitura chama-se `SAFeSWLArchParser`. O seu método `readXML` recebe como entrada o caminho do arquivo arquitetural a ser lido. A partir daí, ele gera um objeto central principal chamado de `SAFeArchitecture`. A classe `SAFeArchitecture` reúne os principais objetos pertencentes a uma arquitetura. De acordo com o diagrama de classes, são eles:

- **`ArchApplication`**: denota o objeto que representa o componente aplicação, armazenando informações sobre as suas portas de ambiente.
- **`ArchWorkflow`**: denota o objeto que representa o componente *workflow*, armazenando informações sobre as suas portas de ambiente e de tarefas;
- **`ArchBody`**: denota um objeto que representa o conjunto de objetos da classe `ArchComponent`, explicado a seguir;

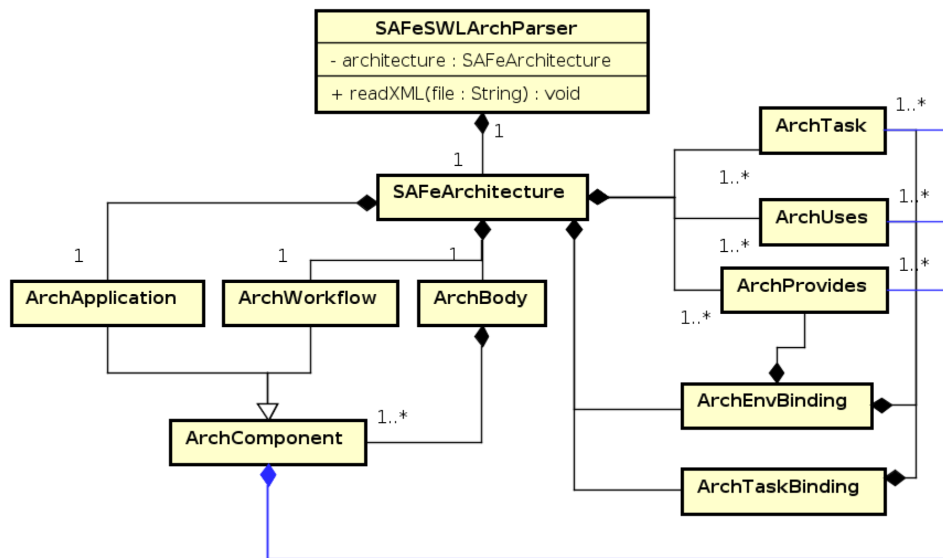


Figura 3.9: Classes da linguagem arquitetural, contidas no pacote `safe-language`.

- **`ArchComponent`**: denota os objetos que representam os componentes propriamente ditos da solução computacional construída pela aplicação, armazenando informações sobre sua espécie, as suas portas (de ambiente e de tarefas) e referências para os seus contratos contextuais (Seção 3.4.3), essenciais para a sua resolução (escolha de implementação conforme o contexto) durante a execução do *workflow*;

- **`ArchTask`**: denota os objetos que representam portas de tarefa, armazenando informações sobre as ações que definem o seu tipo;

- **`ArchUses`**: denota os objetos que representam portas de ambiente usuárias;

- **`ArchProvides`**: denota os objetos que representam portas de ambiente provedoras;

- **`ArchTaskBinding`**: denota os objetos que representam ligações entre duas portas de tarefas interessadas em sincronizar a ativação de ações;

- **`ArchEnvBinding`**: denota os objetos que representam ligações entre duas portas de ambiente, uma no papel de usuária e outra no papel de provedora.

As informações armazenadas nessas classes ficarão em uma estrutura dados hierárquica, a qual poderá ser acessada pelo componente *workflow* durante sua execução para angariar informações arquiteturais. Por exemplo, o componente

workflow precisará em um determinado momento acionar uma ação de uma porta de tarefa. As informações sobre essa porta poderão ser acessadas via estrutura de dados em memória, através de métodos apropriados para isso, pela linguagem de orquestração.

3.4.3 Contratos Contextuais

Contratos contextuais dos componentes do *workflow* são armazenados separadamente ao código de descrição arquitetural, em arquivos de extensão *cc*. A localização do arquivo, bem como o seu nome, é indicado através de uma *tag* `contract` no código arquitetural, conforme será exemplificado no estudo de caso referente à aplicação Map/Reduce (Seção 4.2). Isso garante independência em relação ao formato do contrato contextual esperado pelo *Core* no mecanismo de resolução de componentes. Atualmente, usa-se o mesmo formato XML usado em *tipos de instanciação* do HPE. Em particular, o que motivou essa decisão foi a necessidade de utilizar o *Core* do HPE, o qual implementa uma versão mais simples do sistema de contratos contextuais da HPC Shelf, o HTS [Carvalho Junior et al. 2013], nos primeiros protótipos da HPC Shelf, ainda não dispondo de uma implementação de Alite que pudesse ser utilizada até o fechamento desta Tese de Doutorado.

3.4.4 O Subconjunto de Orquestração

O papel do componente *workflow* é atuar como um *conector* especial, o qual dirige a execução da solução computacional através da orquestração das ações de portas de tarefas de componentes de computação e conectores. Para isso, ele é conectado às portas de tarefas dos componentes especificados na descrição arquitetural. Cada porta de tarefa exporta um conjunto de *ações*, as quais definem as operações suportadas por um componente remoto. A linguagem de orquestração, executada pelo *workflow*, especifica um fluxo de ativação para essas ações através de *combinadores* capazes de especificar ativações sequenciais, concorrentes, alternativas, iterativas e assíncronas das ações.

A Figura 3.10 apresenta uma versão abstrata da gramática do subconjunto de orquestração da linguagem SAFeSWL. A gramática concreta, especificada no formato XSD, é mais complexa e pode ser vista em mais detalhes no Apêndice B.

Uma *ação*, representada na gramática pela variável ACTION, pode ser:

- uma ação de *ativar* uma ação declarada em uma porta de tarefa do *workflow* (compute *action_id*);

ACTION ::= **resolve** *component_id* | **deploy** *component_id* | **instantiate** *component_id* |
release *component_id* | **compute** *acion_id*
TASK ::= SKIP_TASK | BREAK_TASK | CONTINUE_TASK |
START_TASK | WAIT_TASK | CANCEL_TASK | ACTIVATE_TASK |
SEQUENCE_TASK | PARALLEL_TASK | CHOICE_TASK | ITERATE_TASK
SKIP_TASK ::= **skip**
BREAK_TASK ::= **break**
CONTINUE_TASK ::= **continue**
START_TASK ::= **start** *handle_id*? ACTION
WAIT_TASK ::= **wait** *handle_id* ACTION
CANCEL_TASK ::= **cancel** *handle_id* ACTION
ACTIVATE_TASK ::= **invoke** ACTION
SEQUENCE_TASK ::= **sequence** TASK+
PARALLEL_TASK ::= **parallel** TASK+
CHOICE_TASK ::= **select** {*action_id*: TASK}+
ITERATE_TASK ::= **iterate** TASK

Figura 3.10: Sintaxe abstrata da linguagem de orquestração do SAFeSWL

- uma ação de *resolver* o contrato contextual de um componente declarado na descrição arquitetural (*resolve component_id*);
- uma ação de *implantar* um componente em uma plataforma virtual hospedeira (*deploy component_id*), sob o pressuposto de que seu contrato contextual tenha sido resolvido anteriormente; ou
- uma ação de *instanciar* um componente (*instantiate component_id*), sob o pressuposto de que tenha sido anteriormente implantado.
- uma ação de *liberar* um componente (*release component_id*), sob o pressuposto de que tenha sido anteriormente instanciado, devolvendo à infraestrutura recursos alocados pelo componente.

As ações *resolve*, *deploy* e *instantiate* são “açúcares sintáticos” de SAFeSWL para diferenciar as ações da porta de tarefas LifeCycle, de caráter não funcional, das ações funcionais das portas de tarefa declaradas por cada componente de computação ou conector.

Uma *tarefa* é definida como uma orquestração de várias *ações*. O *workflow* é definido como uma tarefa de mais alto nível, orquestrando ações de tarefas distintas. Sendo assim, um *workflow* é representado pelo não-terminal TASK. Existem cinco tipos de tarefas *primitivas* e quatro *combinadores* de tarefas.

Os tipos primitivos de tarefas são:

- SKIP_TASK denota uma ação vazia;
- BREAK_TASK termina a iteração onde encontra-se diretamente aninhada;
- CONTINUE_TASK volta ao início da iteração onde encontra-se diretamente aninhada;
- START_TASK denota a ativação assíncrona de uma ação, a qual pode ser manipulada por um identificador (*handle_id*);
- WAIT_TASK denota a espera do término de uma ação assíncrona previamente ativada, possivelmente bloqueando a *thread* do *workflow* onde foi executada;
- CANCEL_TASK denota o cancelamento da ativação de uma ação previamente ativada de forma assíncrona;
- ACTIVATE_TASK denota a ativação síncrona de uma ação, equivalente a uma invocação assíncrona (**start**) diretamente seguida por uma espera (**wait**).

Os combinadores de tarefas são:

- SEQUENCE_TASK denota a execução sequencial de uma lista de ações, na ordem em que são declaradas;
- PARALLEL_TASK denota a execução concorrente de um conjunto de tarefas, onde a tarefa combinadora só termina quando todas suas tarefas internas terminam (modelo *fork-join*);
- CHOICE_TASK denota a execução de uma das tarefas dentro de um conjunto de tarefas que estejam associadas a uma ação de uma porta de tarefa que esteja preparada para ser completada;
- ITERATE_TASK denota a execução iterativa de uma tarefa a qual termina quando uma tarefa break é alcançada.

3.4.5 O Processamento de um Código de Orquestração de SAFeSWL

Durante a execução do *workflow*, o arquivo de orquestração será carregado e transformado em um conjunto de objetos os quais serão organizados em uma estrutura de dados em árvore. Essa estrutura será lida recursivamente e, ao chegar em um nó que expresse uma ação da linguagem, efetuará sua lógica. Por exemplo,

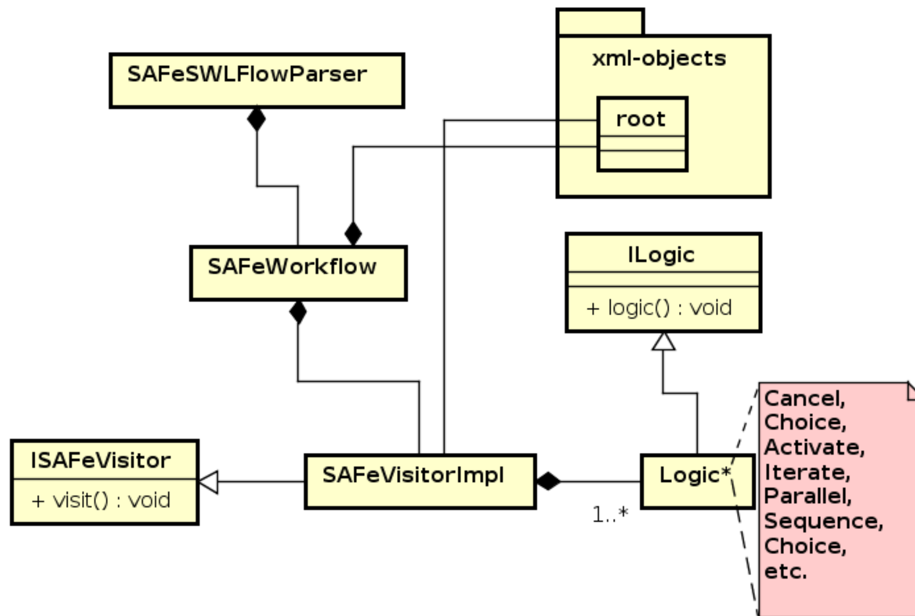


Figura 3.11: Classes da linguagem execução, contidas no pacote `safe-language`.

quando é lido o nó do tipo ACTIVATE_TASK, será chamada uma classe específica que implementa a lógica de ativação de uma tarefa.

De acordo com a Figura 3.11, a classe que centraliza a leitura do arquivo XML é `SAFeSWLFlowParser`. Ela irá gerar o objeto `SAFeWorkflow` que contém todos os elementos da descrição XML da orquestração, também no formato de objetos Java. Para cada elemento da orquestração, contido em `SAFeWorkflow` dentro do pacote `xml-objects` onde o objeto `root` representa a raiz da árvore, foi implementado o padrão *Visitor*. Nesse padrão, uma interface `ISAFeVisitor` obriga que suas implementações sobrescrevam o método `visit()`. Cada elemento do pacote `xml-objects` está relacionado com um *visitor*. Durante a leitura da árvore em tempo de execução do *workflow*, cada *visitor* é executado chamando-se o seu método `visit()`. Quando o método `visit()` é chamado, internamente, via reflexão, é disparado uma chamada para um objeto `Logic*` o qual representa a lógica de um elemento contido na gramática XSD da linguagem de orquestração.

Quando, por exemplo, chega-se ao nó referente ao elemento SEQUENCE_TASK, o padrão *visitor* dispara a criação de um objeto do tipo `LogicSequence`, o qual internamente irá executar todas as suas tarefas filhas em sequencia. A mesma abordagem é adotada para todos os outros comandos da linguagem.

3.5 Ciclo de Vida de uma Aplicação

O ciclo de vida de uma aplicação compatível com o SAFe possui duas fases:

- a *fase de desenvolvimento*, onde o provedor cria uma aplicação, constituída de uma interface de alto nível voltada ao especialista, tanto para descrição de problemas quanto para acompanhamento da execução de soluções computacionais propostas pela aplicação, bem como de um mecanismo de geração dessas soluções computacionais, na forma de *workflows* em SAFeSWL;
- a *fase de execução*, onde soluções computacionais são sintetizadas e executadas pela aplicação.

3.5.1 Fase de Desenvolvimento

Para a fase de desenvolvimento, o provedor de aplicações deve ter conhecimento da API do SAFe, manipulação de arquivos XML, para compreensão e interpretação de arquivos de código SAFeSWL, e programação Java.

A primeira obrigação do provedor de aplicações, em uma visão *top-down*, é a concepção, especificação e implementação da interface de alto nível da aplicação. Para isso, o provedor de aplicação deverá usar seus conhecimentos sobre o domínio da aplicação para discernir que tipos de problemas podem ser resolvidos através dela. Assim, deverá elaborar uma interface, possivelmente na forma de uma ou mais linguagens de propósito especial, através da qual especialistas possam especificar problemas de seu interesse, dentro do domínio da aplicação.

A segunda obrigação do provedor de aplicações é especificar a arquitetura das soluções computacionais para resolver cada um dos problemas que podem ser especificados através da interface de alto nível da aplicação, na forma de *workflows* em SAFeSWL. Uma aplicação pode definir diferentes estratégias, desde a sintetização dinâmica dos arquivos de código SAFeSWL a partir da especificação de um problema específico por parte do especialista, até a organização de uma biblioteca de *workflows*, ou *templates* de *workflows*, escritos em arquivos SAFeSWL pré-definidos para cada problema possível. Independente da forma como esses códigos SAFeSWL são obtidos, a seguinte sequência de tarefas é necessária para especificação de cada solução computacional distinta:

i. **Escolha dos componentes:** através dos serviços do *Core*, o provedor de aplicações tem acesso aos componentes abstratos disponíveis no catálogo da

HPC Shelf para a composição de uma solução computacional que resolva o problema em questão. Vale lembrar que os componentes são implementados pelo desenvolvedor de componentes e os mesmos devem disponibilizar portas de ambiente, tanto para comunicação com a aplicação quanto com outros componentes.

ii. Escolha dos *bindings*: os *bindings* são componentes especializados na comunicação entre as portas. Por exemplo, uma porta de ambiente usuária necessita de um *binding* para poder se comunicar com uma porta de ambiente provedora de mesmo tipo. O mesmo raciocínio pode ser usado para portas de tarefas. Os *bindings* são importados diretamente do *Core*, como componentes distribuídos. Uma parte do *binding* fica de posse da aplicação e uma outra parte fica de posse do componente em execução na plataforma virtual remota. Podemos ter, por exemplo, um *binding* constituído de duas partes, um lado servidor implementado como um Serviço *Web* e um lado cliente sendo implementado como um conjunto de classes *stubs* para conexão com o Serviço *Web*. É interessante notar que a implementação de um *binding* independe totalmente do SAFe. O desenvolvedor de componentes, além de criar os componentes de computação e conectores, disponibiliza os componentes *bindings*, que podem ser implementados de diferentes maneiras, sendo os Serviços *Web* uma das alternativas.

iii. Criação do código arquitetural: uma vez decidido quais componentes irão fazer parte de uma solução computacional, o provedor de aplicações irá montar o código SAFeSWL contendo a descrição arquitetural e de orquestração da solução, ligando as portas dos componentes entre si, incluindo as portas de ambiente do componente aplicação e as portas de ambiente e de tarefas do componente *workflow*. Também são especificados os contratos contextuais de cada componente, usados pelo *Core* no processo de resolução. A comunicação dos *bindings* entre componentes de computação e conectores da solução computacional é realizada sem interferência da aplicação.

iv. Criação dos componentes *Proxies*: os componentes *proxies* são classes do tipo `HShelfComponent` que representam versões locais dos componentes reais localizados remotamente em uma plataforma virtual. Os *proxies* acessam o *Core* para efetuar as operações de *resolução*, *implantação*, *instanciação* e *liberação*, mencionadas anteriormente. No início da aplicação, após a leitura do

arquivo SAFeSWL, todos os componentes *proxies* são criados. Entretanto, suas contrapartes remotas, os componentes reais, são instanciados sob demanda, à medida que o arquivo de orquestração é lido e as ações da porta de tarefas LifeCycle são ativadas.

v. Interface do *workflow* com a aplicação: nesse ponto, o provedor de aplicações sabe quais são as portas de ambiente disponíveis para comunicação entre a aplicação e os componentes que formam a solução computacional (via descrição arquitetural que ele mesmo criou). O provedor de aplicações deve então escolher aquelas portas que interessam à aplicação, para que dados sejam enviados aos componentes remotos e resultados sejam recebidos pela aplicação. Em aplicações mais sofisticadas, as portas de ambiente podem permitir a interação entre a aplicação e os componentes do *workflow* durante a execução. É possível ainda a aplicação instanciar vários *workflows* simultaneamente, intermediando comunicação entre eles. É responsabilidade do provedor de aplicações escrever o código para registro e uso das portas de ambiente da aplicação, sendo responsabilidade do componente *workflow* sua ligação adequada aos componentes da arquitetura do *workflow*.

3.5.2 Fase de Execução

A fase de execução é efetuada pelo especialista. Através da interface da aplicação (uma interface gráfica, por exemplo) criada pelo provedor de aplicações, o especialista especifica um problema que deseja que seja resolvido e a aplicação gera um *workflow* que representa uma solução computacional para esse problema, de forma transparente. Feito isso, o especialista pode dar início a execução da aplicação, possivelmente interagindo com o *workflow* dinamicamente e recebendo ao final o resultado do processamento. Tanto a forma de interação quanto a forma como os resultados são apresentados são responsabilidades da aplicação. A ordem das tarefas para tal processo é explicitada a seguir:

i. Disparo do *workflow*: a execução do *workflow* é disparada pelo especialista quando deseja executar a solução computacional de um problema, cujos parâmetros e demais configurações de entradas são lidos da descrição fornecida através da interface de alto nível da aplicação. Para ativar o *workflow*, o componente aplicação acessa a porta do tipo `GoPort` oferecida pelo componente *workflow* e executa o seu método `go`.

ii. **Interação com o *workflow*:** durante a execução, a aplicação interage com o *workflow* através dos *bindings* de ambiente entre suas portas de ambiente e as portas a ela ligadas de outros componentes do *workflow*, tanto no papel de usuária quanto no papel de provedora. As portas foram registradas previamente através do objeto *services* do componente aplicação. No caso onde a aplicação pretende acessar uma porta usuária, invocando o método `getPort` do objeto *services*, o SAFe já está de posse dos *proxies* para acessá-las e permitir a interação com o mesmo. Porém, a invocação de `getPort` só se completa no código da aplicação após as operações de resolução, implantação e instanciação terem sido executadas sobre o componente dono da porta, quando os componentes *binding* encontram-se instanciados e prontos para executar operações de comunicação.

iii. **Finalização do *workflow*:** ao receber a notificação de que o fluxo de controle do *workflow* chegou ao final, a aplicação, através do método `releasePort` de seu objeto *services*, deve liberar o uso das portas que registrou como usuárias. O mesmo é feito por todos os componentes do *workflow*. Quando todas as portas encontram-se liberadas por seus respectivos componentes, o componente *workflow*, através de sua porta *builder services*, executa as operações de desconexão das ligações entre os componentes (método `disconnect`), seguidas da operação de liberação dos componentes instanciados (método `releaseComponent`), incluindo plataformas virtuais.

A Figura 3.12 apresenta o diagrama do processo de execução. Por questões de simplificação, o diagrama resolve apenas um único componente, mas o mesmo processo se repete quantas vezes o *workflow* desejar.

3.6 Um Protótipo de Aplicação

Esta seção apresenta um protótipo simples de aplicação construída com o SAFe, composta de:

- um par de componentes de computação, um dos quais denominado *Cliente*, doravante chamado *componente cliente*, e outro denominado *Servidor*, doravante chamado *componente servidor*;
- um componente conector, denominado *Fila*, doravante chamado *componente fila*, através do qual os outros componentes de computação realizam uma coreografia simples e bastante conhecida, a interação produtor-consumidor.

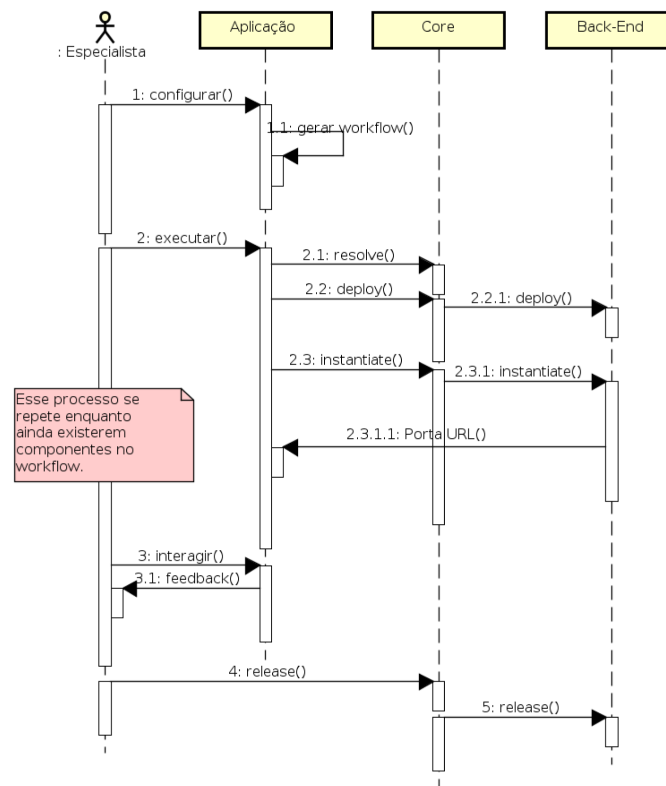


Figura 3.12: Diagrama da fase de execução da aplicação.

Cada um dos componentes possui portas (usuárias, provedoras e de tarefas) para comunicação entre si e com a aplicação. O objetivo da aplicação é ilustrar a comunicação entre esses componentes e entre eles e a aplicação, bem como a sua orquestração.

A Figura 3.13 apresenta os componentes cliente, servidor e fila com suas respectivas portas de ambiente (provedoras e usuárias) e de tarefas. A seguir, são explicados cada um dos componentes e suas respectivas portas.

O componente cliente requisita uma mensagem (no formato *string*) da aplicação para fins de repassá-la ao componente servidor, usando o componente fila como intermediário, capaz de armazenar em uma fila as mensagens enviadas pelo componente cliente enquanto o componente servidor ainda não recebeu a mensagem. Para isso, implementa as portas de ambiente e tarefas apresentadas na Tabela 3.4.

O fluxo de execução, por parte do componente cliente, se dá da seguinte forma. O componente *workflow* ativa a ação `post`, na porta de tarefa do componente cliente, a fim de requisitar que esse componente solicite uma mensagem para o componente aplicação. A solicitação é realizada pela invocação, por parte do componente cliente, da operação `requestMessage`, na sua porta usuária que, no exemplo em questão,

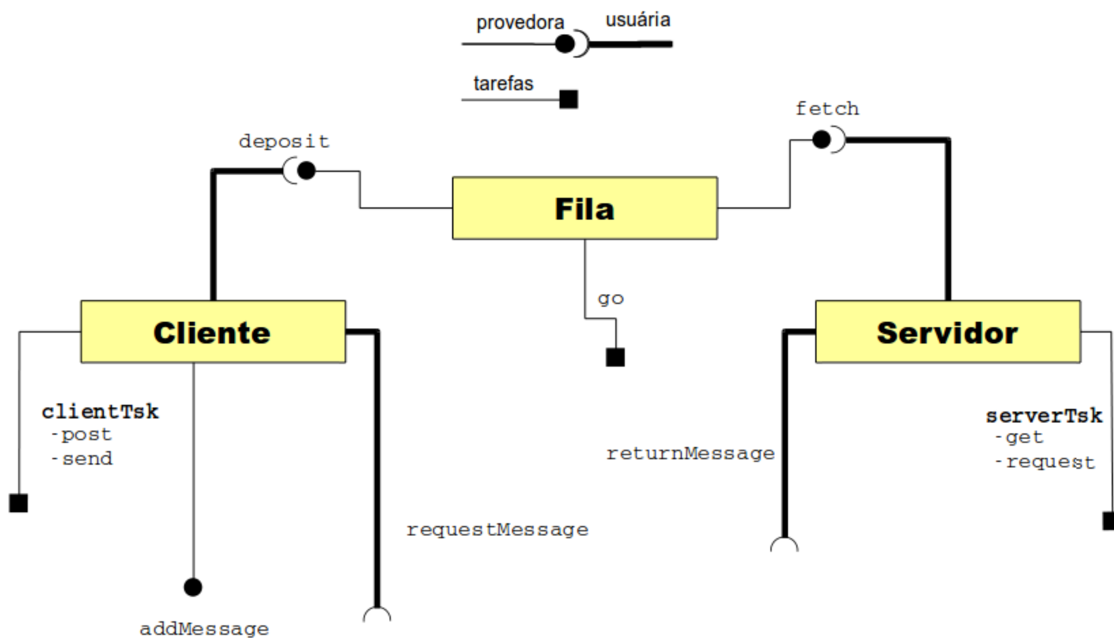


Figura 3.13: Componentes Cliente e Servidor

Porta de Ambiente Provedora (operações)	
addMessage	Através da qual um componente usuário envia uma mensagem para o *Client* (no exemplo, o componente usuário é o componente aplicação).
Porta de Ambiente Usuária (operações)	
requestMessage	Através da qual o *Client* requisita uma mensagem a outro componente (no exemplo, ao componente aplicação).
Porta de Ambiente Usuária (operações)	
deposit	Através da qual o *Client* deposita uma mensagem em uma fila gerenciada por outro componente (no exemplo, o componente fila).
Porta de Tarefas	
post	Ativada para que o componente cliente "acorde" e requisite uma mensagem a um outro componente via requestMessage.
send	Ativada para que o componente cliente deposite uma mensagem recebida via addMessage através da operação deposit.

Tabela 3.4: Operações das Portas do Componente Cliente

Porta de Ambiente Provedora (operações)	
`deposit`	Através da qual um componente produtor (no exemplo, o componente cliente) deposita uma mensagem na fila.
Porta de Ambiente Provedora (operações)	
`fetch`	Através da qual um componente consumidor (no exemplo, o componente servidor) pega uma mensagem da fila.
Porta de Tarefas (ações)	
`go`	Inicia o laço de execução do componente, no qual aguarda requisições dos componentes produtor e consumidor.

Tabela 3.5: Operações das Portas do Componente Fila

está conectada a uma porta provedora do componente aplicação. Como resposta, o componente aplicação invoca a operação `addMessage` em uma porta usuária que supõe-se, no exemplo em questão, estar conectada a uma porta provedora do mesmo componente cliente, passando uma *string* que representa a mensagem como argumento. Finalmente, o componente *workflow* ativa a ação `send` da porta de tarefas do componente cliente para que a mensagem enviada pelo componente aplicação seja "depositada" no componente fila, invocando-se a operação `deposit` em sua porta usuária ligada à porta provedora correspondente do componente fila.

Observe que o fluxo acima, para esse exemplo simples em particular, poderia ser bastante simplificado. Porém, seu objetivo é demonstrar uma conversação relativamente complexa entre o componente cliente e os componentes aplicação e fila. A mesma observação vale para as descrições a seguir.

O componente (conector) fila tem como função atuar como um *buffer* de mensagens entre os componentes cliente e servidor, para isso implementando as portas de ambiente e de tarefas apresentadas na Tabela 3.5. Em seu laço de execução principal, pega mensagens depositadas na fila da faceta do lado produtor, pelo componente cliente, através da operação `deposit` da sua porta de ambiente provedora da mesma faceta, e as envia para uma fila na faceta do lado consumidor, onde espera-se que sejam lidas pelo componente servidor como efeito da invocação da operação `fetch`, através de uma outra porta de ambiente provedora.

Portas de Tarefas	
`get`	Ativada para que o componente pegue uma nova mensagem de uma fila gerenciada por um outro componente (no exemplo, o componente fila).
`request`	Ativada para que envie para um outro componente (no exemplo, o componente aplicação) a mensagem mais recentemente pegada da fila como efeito da ativação da ação `get`.
Porta de Ambiente Usuária (operações)	
`fetch`	Através da qual pega uma mensagem da fila gerenciada pelo outro componente.
Porta de Ambiente Usuária (operações)	
`returnMessage`	Através da qual envia para um outro componente (no exemplo, o componente aplicação) uma mensagem requisitada através da ação `request`.

Tabela 3.6: Operações das Portas do Componente Servidor

O componente servidor tem como função receber mensagens oriundas do componente cliente, usando o componente fila como *buffer* intermediário, e enviá-las de volta para o componente aplicação, uma vez que essas mensagens foram inicialmente geradas pela própria aplicação, que as enviou para o componente cliente. Para isso, expõe as portas de ambiente e de tarefas apresentadas na Tabela 3.6.

Quanto ao fluxo de execução, o componente *workflow* ativa a ação `get` de sua porta de tarefas, a fim de fazer com que retire uma mensagem da fila do componente fila, através da operação `fetch` de sua porta usuária conectada a porta provedora de mesmo tipo do componente fila. Então, o componente *workflow* ativa a ação `request` para que envie a mensagem para o componente aplicação através da operação `returnMessage` de sua porta usuária conectada a uma porta provedora do componente aplicação.

Toda a lógica de comunicação remota entre os componentes cliente e servidor, os quais não residem na mesma plataforma virtual, encontra-se encapsulada no componente conector fila. Por sua vez, toda a lógica de sincronização e comunicação entre as portas de ambiente que ligam os componentes cliente, servidor e fila ao componente aplicação encontram-se encapsuladas nos componentes *binding*, os quais fazem isso através de Serviços *Web*. Por outro lado, os *bindings* entre os componentes clientes, servidor e fila são diretos.

A descrição acima mostra que a aplicação executa uma lógica *ping-pong*, onde uma mensagem fornecida pelo usuário especialista por meio da interface de alto nível da aplicação circula entre os componentes aplicação, cliente, fila e servidor,

voltando novamente para o componente aplicação, respectivamente nessa ordem. O fluxo de envio e recebimento da mensagem repete-se quantas vezes forem requisitadas pelo especialista, executada pelo seguinte protocolo de orquestração sobre as ações das portas de tarefas, executada pelo componente *workflow*:

```
repeat seq { post; send; get; request}
```

O protocolo acima, explicado a seguir, resume o fluxo de execução durante o ciclo de vida da aplicação de exemplo:

1) É iniciada uma repetição das ações encapsuladas pelo comando **`repeat`**, ou seja, várias mensagens serão repassadas;

2) O componente *workflow* ativa a ação **`post`** para solicitar ao componente cliente que invoque a operação `requestMessage` do componente aplicação, a qual responde com a invocação da operação `addMessage` do componente cliente a fim de entregá-lo uma mensagem [1];

3) O componente *workflow* ativa a ação **`send`** para solicitar ao componente cliente que invoque a operação `deposit` do componente fila, depositando assim a mensagem recebida pela aplicação em seu *buffer*;

4) O componente *workflow* ativa a ação **`get`** para solicitar que o componente servidor invoque a operação `fetch` do componente fila a fim de recuperar uma mensagem pendente lá armazenada;

5) O componente *workflow* ativa a ação **`request`** a fim de solicitar que componente servidor invoque a operação `returnMessage` do componente aplicação, a fim de que receba de volta a mensagem que foi originalmente enviada por ele;

6) Os passos **2** a **5** são repetidos de acordo com as necessidades da aplicação.

[1] Uma alternativa de projeto seria retornar a mensagem diretamente pela operação `requestMessage`, ao invés de através de uma chamada a uma operação do componente cliente.

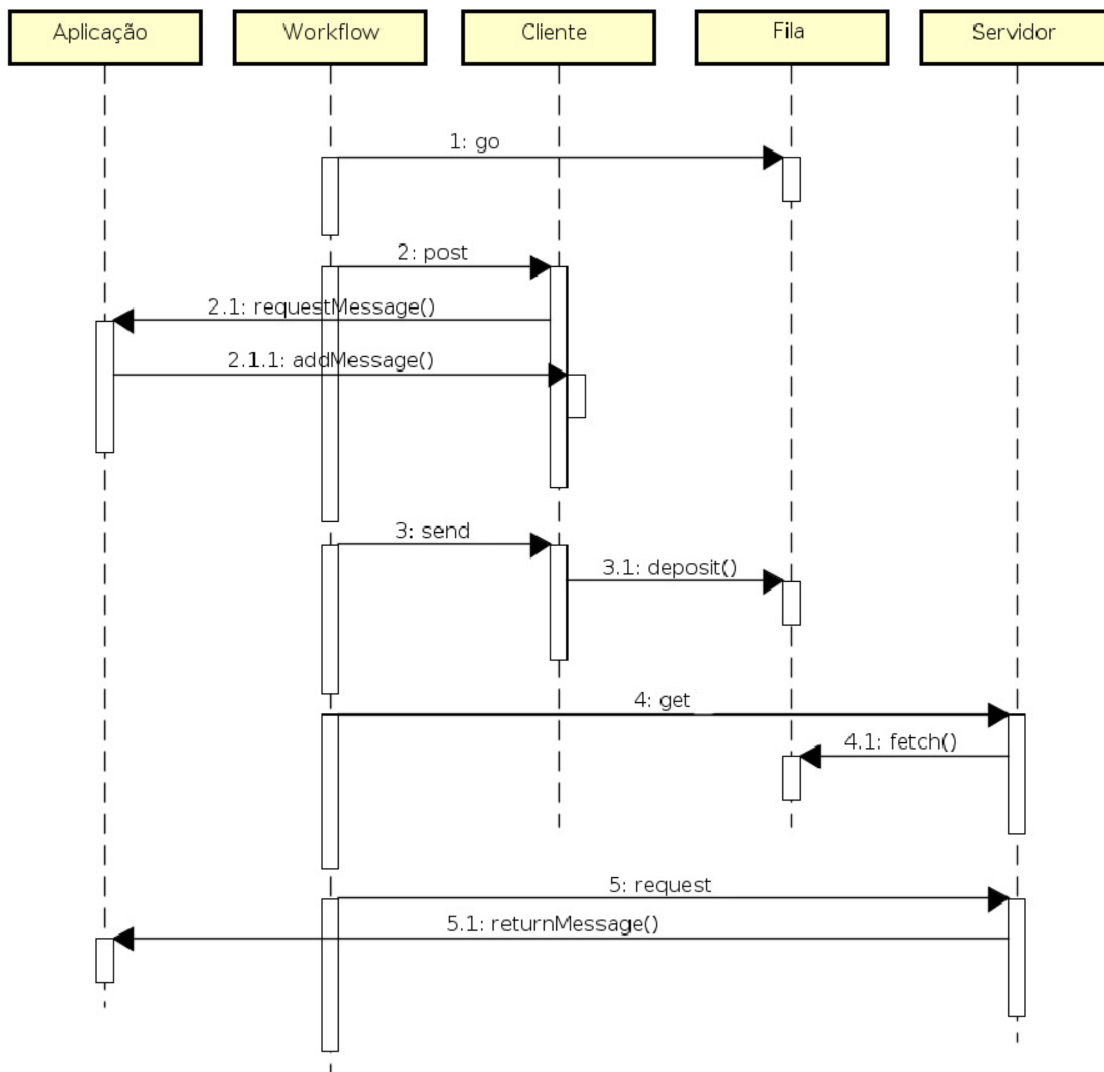


Figura 3.14: Diagrama do Fluxo de Orquestração da Aplicação de Exemplo

A Figura 3.14 apresenta um diagrama de sequência da execução do protocolo de orquestração descrito anteriormente. Os passos 2 a 5 são repetidos em um laço. A seguir, explicaremos as etapas envolvendo a criação de código no SAFe para o protótipo exemplo totalmente funcional.

a) Criação dos *Bindings*

Como explicado anteriormente, a comunicação dos componentes aplicação e *workflow* para com os componentes de solução é realizada por intermédio dos componentes *binding*, para portas de ambiente e de tarefa. Uma vez que os componentes de solução encontram-se remotamente distribuídos, as facetas cliente e servidora desses componentes, quando ligam a aplicação e os componentes de

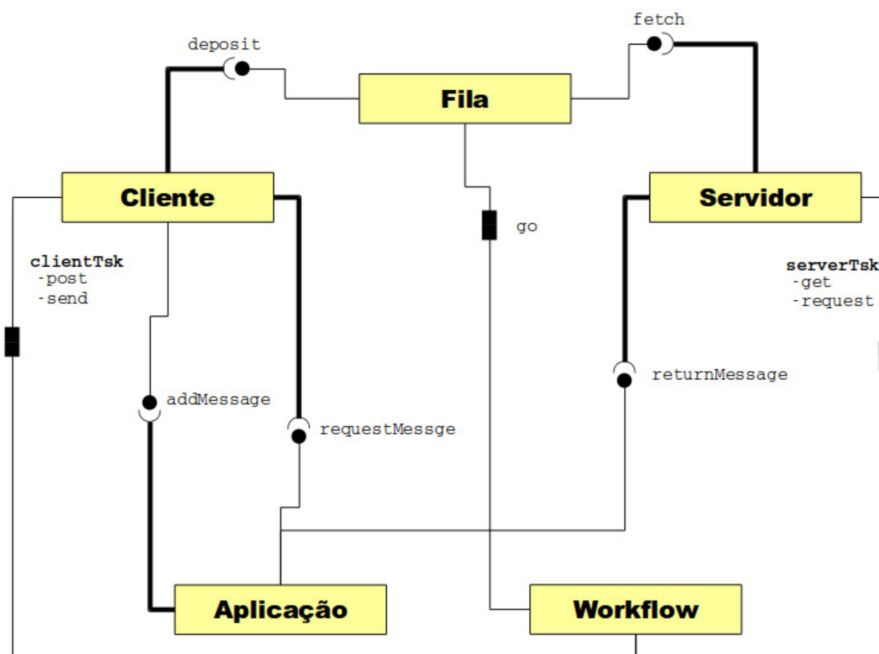


Figura 3.15: Componentes Cliente, Fila, Servidor, Aplicação e *Workflow*.

solução, devem ser implementados por meio de algum mecanismo de comunicação distribuído. Entretanto, o uso desse mecanismo é transparente para o provedor de aplicações, uma vez que encontra-se encapsulado na implementação dos componentes *binding*, cujas implementações são resolvidas em tempo de execução em função de seus contratos contextuais. O desenvolvedor de componentes é o responsável por implementar componentes *binding* que possam ser usados pela aplicação, bem como publicá-los no catálogo de componentes mantido pelo *Core*, sendo responsabilidade do provedor apenas reusá-los a partir do catálogo. Por outro lado, nada impede que o provedor de aplicação também atue no papel de desenvolvedor de componentes quando lhe convier, notadamente a fim de atender as necessidades particulares de uma aplicação. Levando em consideração esse papel dual do provedor de aplicações, e também considerando a importância de explicar o mecanismo de comunicação entre a aplicação e os componentes de solução, é explicado a seguir como são implementados os componentes *binding* da aplicação de exemplo em questão.

Na aplicação de exemplo, as portas de ambiente e tarefas dos três componentes de solução que são de interesse dos componentes *workflow* e de aplicação (cliente, servidor e fila) são expostas na forma de Serviços *Web*. Nesse caso, a faceta servidora do componente *binding* oferece um Serviço *Web*, ou seja, sua interface em WSDL, enquanto a faceta cliente representa os *stubs*, ou seja, as classes responsáveis em

estabelecer conexão e comunicar-se com o Serviço *Web*.

A Figura 3.15 completa a Figura 3.13 apresentando os componentes que executam ao lado do SAFe: o componente aplicação, o qual se conecta com as portas de ambiente provedora do componente cliente (`addMessage`), e o componente *workflow*, o qual se conecta com as portas de tarefas dos componentes cliente (`clientTsk`), servidor (`serverTsk`) e fila (`queueTsk`). Note também que a aplicação possui duas portas de ambiente provedoras (`requestMessage` e `returnMessage`), as quais serão conectadas às portas usuárias do componente cliente e do componente servidor

```
import javax.jws.WebMethod;

@WebService
public interface IClientServiceEnv {

    @WebMethod
    public void addMessage(String message);
}
```

Figura 3.16: Faceta Servidora de um *Binding*, implementado como um Serviço *Web*.

O *binding* que conecta a porta usuária `addMessage` da aplicação com a porta provedora do componente cliente, por exemplo, é implementado de acordo como mostrado na Figura 3.16. Fazendo uso do pacote `javax.jws.WebMethod` é possível criar um serviço *web* de forma simples. Para esse *binding*, existe apenas um único método que é o `addMessage(String message)` o qual será exposto como um serviço *web* a ser usado pela aplicação. Na Figura 3.16, é apresentada apenas a interface do serviço. Sua implementação irá se comunicar com a porta provedora do componente cliente para repassar a mensagem em *string* gerada pela aplicação.

```
public class ClientAddMessageEnvPort extends HShelfProvidesPort{

    public void addMessage(String message){
        ClientServiceEnvService service = new ClientServiceEnvService();
        IClientServiceEnv port = service.getClientServiceEnvPort();
        port.addMessage(message);
    }
}
```

Figura 3.17: Faceta Cliente de um *Binding*, Implementado como um Serviço *Web*.

Já pelo lado usuário do mesmo *binding* (do lado da aplicação), temos o método de acesso ao serviço *web* remoto representado pela Figura 3.17 (`addMessage(String`

message)). Esse método, na verdade, acessa uma série de classes criadas automaticamente pela interpretação do arquivo WSDL do servidor do Serviço *Web*.

Para as outras portas de ambiente, inclusive as de tarefas, um código análogo também foi criado para os *bindings* que as conectam. No caso das portas providas pela aplicação, o raciocínio também é o mesmo. A diferença é que, dessa vez, o servidor do serviço *web* ficará do lado da aplicação, expondo o serviço para os componentes cliente e servidor.

b) Desenvolvendo o Arquivo com a Descrição Arquitetural

Nesse passo, o provedor de aplicações deve especificar o arquivo de descrição arquitetural em SAFeSWL, o qual define quais componentes fazem parte do *workflow*, suas portas e como estão relacionados. Abaixo, explicaremos o arquivo em parte para melhor entendimento. Excluiremos da explicação partes do arquivo que podem ser entendidas por simples analogia.

```
<architecture>

 <application id="0" name="app-sample-example">
     <uses_port id="14" name="c-add-msg-uses" id_component="0"/>
     <uses_port id="15" name="port_SAFeSWL" id_component="0"/>
     <uses_port id="16" name="port_Go" id_component="0"/>
     <provides_port id="17" name="a-req-msg-prov" id_component="0"/>
     <provides_port id="18" name="a-ret-msg-prov" id_component="0"/>
 </application>
     ...
```

Listagem 3.1: Componente aplicação.

O código XML mostrado na Listagem 3.1 apresenta o componente aplicação através da *tag* <application> (a raiz de um arquivo arquitetural é a *tag* <architecture>). É importante ressaltar que, na representação XML, alguns nomes não condizem exatamente com os nomes das classes e métodos mostrados anteriormente. No entanto, podemos facilmente mapeá-los. As *tags* <uses_port> e <provides_port> definem portas de ambiente usuárias e provedoras. A porta de nome c-add-msg-uses é a porta usuária que se conecta com a porta provedora do componente cliente. As duas portas seguintes (port_SAFeSWL e port_Go) são portas padrão de interesse do SAFe, através das quais comunicam-se os componentes aplicação e *workflow*. Através da primeira, o componente aplicação envia os arquivos SAFeSWL para o componente *workflow*. Por sua vez, através da segunda, do tipo GoPort, o componente aplicação dispara a execução do *workflow*

invocando o seu método go, semelhante ao CCA. As duas últimas portas são do tipo <provides_port>, de nomes a-req-msg-prov e a-ret-msg-prov, as quais fornecem serviços para os componentes remotos, ou seja, nesse caso, a aplicação faz o papel de servidor dos componentes do *workflow*.

```
...
 <workflow id="1" name="workflow-tutorial">
  <provides_port id="21" name="port_SAFeSWL" id_component="1"/>
  <provides_port id="22" name="port_Go" id_component="1"/>
  <task_port id="23" name="wf-task-client" id_component="1"/>
     <task_port id="24" name="wf-task-server" id_component="1"/>
     <task_port id="25" name="wf-task-queue" id_component="1"/>
 </workflow>
 ...
```

Listagem 3.2: Componente *workflow*.

A Listagem 3.2 representa o componente *workflow* e suas portas. A mesma ideia da listagem anterior é usada aqui. Note que agora o componente *workflow* provê duas portas que serão conectadas às portas usuárias da aplicação (port_SAFeSWL e port_go). Além disso, o componente *workflow* também tem três portas de tarefas, denominadas wf-task-client, wf-task-server e wf-task-queue, que serão conectadas às portas de tarefa do componente cliente, servidor e fila, respectivamente, para ativação de suas ações durante a orquestração.

```
...
  <body>
   <computation name="client" id="2">
     <uses_port id="30" name="client-req-msg-uses" id_component="2"/>
     <provides_port id="31" name="c-add-msg-prov" id_component="2"/>
     <task_port name="client-tsk" id="32" id_component="2">
             <action id="321" name="post"/>
             <action id="322" name="send"/>
     </task_port>
   </computation>

   <computation name="server" id="3">
      <uses_port id="40" name="server-ret-msg-uses" id_component="3"/>
      <task_port name="server-tsk" id="41" id_component="3">
             <action id="411" name="get"/>
             <action id="412" name="request"/>
     </task_port>
   </computation>
```

```
  <computation name="queue" id="4">
    <task_port name="queue-tsk" id="51" id_component="4">
           <action id="511" name="go"/>
   </task_port>
  </computation>
</body>
...
```

Listagem 3.3: Componente cliente, servidor e fila.

A Listagem 3.3 mostra os componentes cliente (`client`), servidor (`server`) e fila (`queue`). Tomemos como exemplo o componente cliente, já que os outros podem ser entendidos de forma análoga. Note que, primeiramente, é definida sua porta usuária (`client-req-msg-uses`), as quais farão conexão com a aplicação, como explicado anteriormente (aplicação como servidora). Logo após, sua porta provedora `c-add-msg-prov`, que também será conectada à aplicação no papel de provedora para esta última. Já a *tag* `<task_port>` demonstra as portas de tarefas do componente cliente, as quais serão conectadas ao componente *workflow*. Note também as ações `post` e `send`.

```
...

<env_binding>
     <uses_port id="14" name="c-add-msg-uses" id_component="0"/>
     <provides_port id="31" name="c-add-msg-prov" id_component="2"/>
</env_binding>

<env_binding>
     <uses_port id="15" name="port_SAFeSWL" id_component="0"/>
     <provides_port id="21" name="port_SAFeSWL" id_component="1"/>
</env_binding>

<env_binding>
     <uses_port id="16" name="port_Go" id_component="0"/>
     <provides_port id="22" name="port_Go" id_component="1"/>
</env_binding>
....
```

Listagem 3.4: Conexões de ambiente.

A Listagem 3.4 apresenta as informações de conexões entre as portas de ambiente, ou seja, qual porta usuária está conectada com qual porta provedora. Isso é definido

por várias *tags* <env_binding> que, internamente, casa uma *tag* <uses_port> com uma outra *tag* <provides_port>.

```
...
 <task_binding>
     <left_peer id="23" name="wf-task-client" id_component="1"/>
     <right_peer name="client-tsk" id="32" id_component="2"/>
 </task_binding>

 <task_binding>
     <left_peer id="24" name="wf-task-server" id_component="1"/>
     <right_peer name="server-tsk" id="41" id_component="3"/>
 </task_binding>

 <task_binding>
     <left_peer id="25" name="wf-task-queue" id_component="1"/>
     <right_peer name="queue-tsk" id="51" id_component="4"/>
 </task_binding>

</architecture>
```

Listagem 3.5: Conexões de tarefas.

A mesma ideia da listagem anterior é usada na Listagem 3.5, onde agora são ligadas portas de tarefas com o componente *workflow*. A informação obtida por essas ligações é usada em tempo de execução, quando há a necessidade de comunicação entre componentes.

c) Criando Componentes *Proxy*

Componentes *proxy* são representações locais, em relação ao SAFe, dos componentes remotos que fazem parte do *workflow*. Herdam da classe `HShelfComponent` e registram o conjunto de portas de ambiente e de tarefas do componente real, que são disponibilizadas através do objeto *services*. Além disso, se comunicam com o *Core* para efetuarem as operações de *resolução*, *implantação*, *instanciação* e *liberação*. Para isso, oferecem uma porta de tarefas ao componente *workflow*, responsável pelo controle do seu ciclo de vida, com as ações **resolve**, **deploy**, **instantiate** e **release**. Essas portas e suas ligações são transparentes em relação à arquitetura do *workflow*, não sendo referenciadas no código SAFeSWL.

```
public class ClientComponentProxie extends HShelfComponent{

    @Override
    public void setServices(IHShelfService services) {
        this.services = services;

        ClientAddMessageEnvPort clientAddMessageEnvPort =
                new ClientAddMessageEnvPort();
        clientAddMessageEnvPort.setName("client-add-msg-provides");

        ClientTskPort tsk = new ClientTskPort();
        tsk.setName("client-tsk");

        try {
            this.services.setProvidesPort(clientAddMessageEnvPort);
            this.services.registerTaskPort(tsk);
        } catch (HShelfException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

    }

}
```

Figura 3.18: Componente *Proxy* para o Componente Cliente

O código da Figura 3.18 apresenta o componente *proxy* relativo ao componente cliente. Note que ele herda de `HShelfComponent` e portanto deve reimplementar o método `setServices`, o qual ele expõe para o *framework* suas portas de tarefas e portas de ambiente provedoras e usuárias. Para o componente servidor e fila, também temos o mesmo raciocínio.

d) Montando a Aplicação

Como anteriormente dito, a aplicação é implementada por uma classe derivada de `HShelfApplication`, a qual sobrescreve um método `setServices` por ser uma subclasse de `HShelfComponent`, de modo que a aplicação é vista pelo SAFe como o componente aplicação. Em seu método `setServices`, a aplicação poderá acessar as portas padrões que o componente *workflow* de cada sessão iniciada provê para o componente aplicação.

Uma das portas provedoras do componente *workflow* é destinada a receber do componente aplicação os arquivos de descrição arquitetural e de orquestração de um *workflow* escrito em SAFeSWL, a fim de descrever a solução computacional para o problema que se deseja resolver, previamente declarado pelo especialista. Uma outra porta provedora permite iniciar a execução do código de orquestração do *workflow*.

```
@Override
public void setServices(IHShelfService services) {

    this.services = services;
    AppRequestMessageEnvPort appRequestMessageEnvPort = this.createAppRequestMessageEnvPort();
    appRequestMessageEnvPort.setName("app-req-msg-provides");
    AppReturnMessageEnvPort appReturnMessageEnvPort = this.createAppReturnMessageEnvPort();
    appReturnMessageEnvPort.setName("app-ret-msg-provides");



    try {
        this.services.setProvidesPort(appRequestMessageEnvPort);
        this.services.setProvidesPort(appReturnMessageEnvPort);
        this.services.registerUsesPort("client-add-msg-uses", IHShelfPortTypes.NO_TYPE);
        this.services.registerUsesPort("port_SAFeSWL",IHShelfPortTypes.DEFAULT);
        this.services.registerUsesPort("port_Go",IHShelfPortTypes.DEFAULT);

    } catch (HShelfException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}
```

Figura 3.19: Trecho de Código do Componente Aplicação SAFe

Na Figura 3.19 temos um trecho de código relativo a aplicação exemplo. Nesse trecho, destacamos o método `setServices` onde a aplicação tem acesso às portas do *workflow* descritas no arquivo arquitetural. São elas a porta `portSWL_WF` (responsável pelos arquivos SAFeSWL) e a `portGo_WF` (responsável em disparar o *workflow*, começando assim a computação).

Somente após o início da execução, o componente aplicação pode conectar suas portas usuárias às portas provedoras dos componentes do *workflow*. De posse dessas portas, a aplicação poderá enviar mensagens ao componente cliente e aguardar as mesmas chegarem do lado do componente servidor. Além disso, a aplicação também expõe suas portas provedoras, a serem usadas pelos componentes de solução.

e) Geração do Código de Orquestração

Para esta aplicação simples, a descrição arquitetural é fixa, variando apenas o código de orquestração de acordo com a entrada fornecida pelo usuário da aplicação, via protocolo de orquestração explicado anteriormente. Isso demonstra uma das conveniências de separar-se os códigos arquitetural e de orquestração de SAFeSWL. O código de orquestração será lido de forma dinâmica, criando os componentes *proxies* por reflexão e estabelecendo conexões com os objetos remotos. Feita a conexão, os dados são repassados pelas portas de ambiente. Um exemplo de código de orquestração gerado pode ser descrito logo abaixo.

```
...
 <!-- go -->
 <operation>
      <invoke_oper action="compute" id="511" />
 </operation>

 <operation>
      <iterate_oper max="4">
            <operation>
                  <sequence_oper>

                        <!-- post -->
                        <operation>
                              <invoke_oper action="compute" id="321" />
                        </operation>

                        <!-- send -->
                        <operation>
                              <invoke_oper action="compute" id="322" />
                        </operation>

                        <!-- get -->
                        <operation>
                              <invoke_oper action="compute" id="411" />
                        </operation>

                        <!-- request -->
                        <operation>
                              <invoke_oper action="compute" id="412" />
                        </operation>

                  </sequence_oper>
            </operation>
      </iterate_oper>
 </operation>
</workflow>
```

Listagem 3.6: Invocando portas.

Na Listagem 3.6, a linguagem de orquestração executa a lógica principal da aplicação anteriormente explicada: o *workflow* ativa a ação go e repete as ações post, send, get e request dentro de uma *tag* <iterate_oper>.

f) Executando a Aplicação

A execução da aplicação depende da forma como a mesma foi implementada pelo provedor de aplicações. Pode ser apresentada ao especialista com uma solução *desktop*, uma aplicação *web*, uma interface de linha de comando, um aplicativo Android ou iOS, dentre outras alternativas. O importante é que seja implementada de forma que melhor atenda às necessidades do especialista.

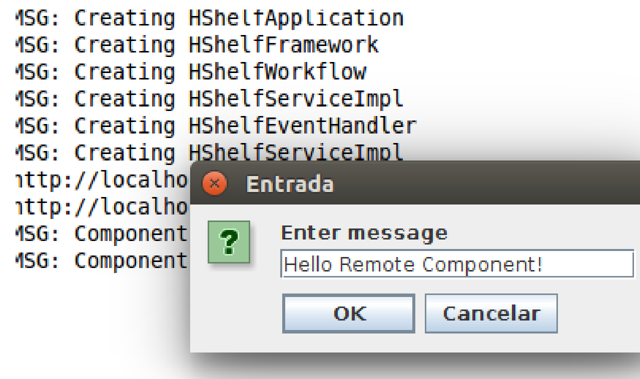


Figura 3.20: Interface Simples, para a Aplicação de Exemplo

No exemplo apresentado, a interface é um programa simples que recebe do especialista, como argumento, a mensagem a ser transmitida para o componente cliente. A Figura 3.20 apresenta uma interface simples, porém funcional, para a aplicação exemplo aqui explicada. A partir do momento que o especialista clica em "OK", a lógica do *workflow* é executada.

```
NAME:client
NAME:server
MSG: invoking action: get; from port: server-tsk
NAME:client
NAME:server
MSG: invoking action: request; from port: server-tsk
RECEIVED MESSAGE FROM SERVER: ALTERED : Hello Remote Component!
MSG: ENDED SEQUENCE TASKS
MSG: **WORKFLOW READING ENDED**
```

Figura 3.21: Exemplo de Saída

A Figura 3.21 mostra a saída da aplicação exemplo, via console. O componente servidor envia a mensagem original de volta ao componente aplicação.

3.7 Considerações Finais

Este capítulo introduziu os principais conceitos do SAFe, o *framework* para construção de aplicações da HPC Shelf proposto por esta Tese de Doutorado.

Viu-se que o SAFe é formado por dois pacotes principais: o `safe-framework` e o `safe-language`. O primeiro trata das classes que formam a API do SAFe, possibilitando a criação de aplicações que ofereçam interfaces de alto nível para descrição de problemas e acompanhamento da execução de soluções por parte do especialista. O segundo trata da implementação da linguagem SAFeSWL, de descrição arquitetural e orquestração de soluções computacionais baseada em componentes, a qual constitui uma das principais contribuições do nosso trabalho.

O código SAFeSWL é dividido em duas partes, submetidos separadamente para o componente *workflow* pelo componente aplicação, contendo, respectivamente, a descrição arquitetural da solução computacional (componentes, portas e ligações) e o fluxo de orquestração de suas ações. Vale ressaltar que o SAFe baseia-se, na medida do possível, no modelo CCA, o que garante maior respaldo da comunidade científica.

O exemplo apresentado neste capítulo serve apenas como propósito de teste de conceito. É um exemplo simples, formado por três componentes. O objetivo é demonstrar que a aplicação implementada no SAFe é totalmente independente da tecnologia usada na construção dos componentes. Basta apenas que eles exponham suas portas como serviços e definam quais são as portas de ambiente e quais são as portas de tarefas. Tal abordagem separa interesse funcionais de interesses não-funcionais da aplicação.

No próximo capítulo, serão apresentados dois exemplos mais completos de aplicações construídas sobre o SAFe, com o propósito de demonstrar mais aspectos relativos às inovações propostas nesta Tese de Doutorado. Assim, o leitor estará melhor preparado, no Capítulo 5, para uma discussão mais detalhada sobre as contribuições do SAFe em relação a outros *frameworks* para construção de *workflows* científicos e sua execução sobre plataformas de Computação de Alto Desempenho.

Capítulo 4

Estudos de Caso

Este capítulo tem como objetivo avaliar o *framework* SAFe quanto a sua utilização em aplicações reais. A primeira aplicação diz respeito a dois *workflows* distintos construídos através de componentes da ferramenta Montage. A segunda aplicação diz respeito a um *workflow* clássico do Map/Reduce. As seções seguintes irão detalhar cada umas das aplicações bem como suas implementações no SAFe.

4.1 Estudo de Caso 1: Montage

Montage[1] é um conjunto de ferramentas de astrofotografia[2] com o intuito de montar imagens de natureza astronômica em um mosaico (conjunto de imagens organizadas de uma determinada área). Um mosaico preserva a calibração e a posição fiel das imagens de entrada originais além de representar regiões do céu que são grandes demais para serem reproduzidas por câmeras astronômicas.

A ferramenta consiste em vários componentes autoexecutáveis e independentes os quais trabalham com arquivos de entrada e alguns parâmetros, gerando arquivos de saída para outros componentes até chegar ao resultado final (mosaico). Para os problemas apresentados nesta Tese, a entrada consiste em imagens em estado natural (já disponibilizadas pelo site do projeto) que serão tratadas por diversos componentes agrupados em um *workflow*, tendo como saída imagens finais de um objeto astronômico.

A importância do Montage para o nosso estudo de caso é resultante das seguintes características que apresenta:

- Componentes independentes, com entradas e saídas bem definidas, os quais

[1] *Montage Astronomical Image Mosaic Engine* - http://montage.ipac.caltech.edu

[2] Astrofotografia é um tipo específico de fotografia de imagens de corpos celestiais.

podem ser facilmente encapsulados em componentes SAFe, disponibilizando assim suas funcionalidades por meio de portas de ambiente e portas de tarefas;

- *Workflows* pré-definidos com saídas previamente testadas, os quais podem ser transcritos para SAFeSWL através de uma aplicação HPC Shelf;
- Portabilidade entre praticamente quaisquer plataformas compatíveis com Linux/Unix;
- Reconhecimento pela comunidade científica, evidenciado pela sua utilização como estudo de caso em vários trabalhos de pesquisa e desenvolvimento relacionados a *workflows* científicos, incluindo a maioria dos sistemas gerenciadores de *workflows* apresentados no Capítulo 5 [3].

4.1.1 Visão Geral dos Componentes do Montage

O Montage, em sua versão 4.0, oferece diversos componentes implementados como bibliotecas de subrotinas e programas executáveis compatíveis com os sistemas operacionais mais populares. Como entrada, cada componente recebe parâmetros de configuração (opcionais) e arquivos, que podem representar informações sobre imagens ou até mesmo imagens em formatos específicos. Como saída, cada componente pode gerar um único arquivo ou uma série de arquivos de imagens, o quais poderão ser usados como entrada para outros componentes.

Para fins deste estudo de caso, cada componente Montage que participará dos *workflows* escolhidos como casos de teste será encapsulado em um componente compatível com a plataforma HPC Shelf. Além disso, suas entradas e saídas serão adaptadas para portas de ambiente provedoras e usuárias, respectivamente. Para implementar sua lógica de orquestração, cada componente disporá de portas de tarefas, com uma única ação a ser ativada, denominada `go`.

A Figura 4.1 apresenta o relacionamento entre um exemplo genérico de componente do Montage e os componentes aplicação e *workflow*. Nessa abordagem, através das portas de ambiente, é possível passar e receber parâmetros e, através das portas de tarefas, é possível que o componente *workflow* orquestre os componentes de solução, disparando suas ações de acordo com o código de orquestração.

[3] Mais informações sobre o uso do Montage são acessíveis através do sítio eletrônico no endereço http://montage.ipac.caltech.edu/news.html

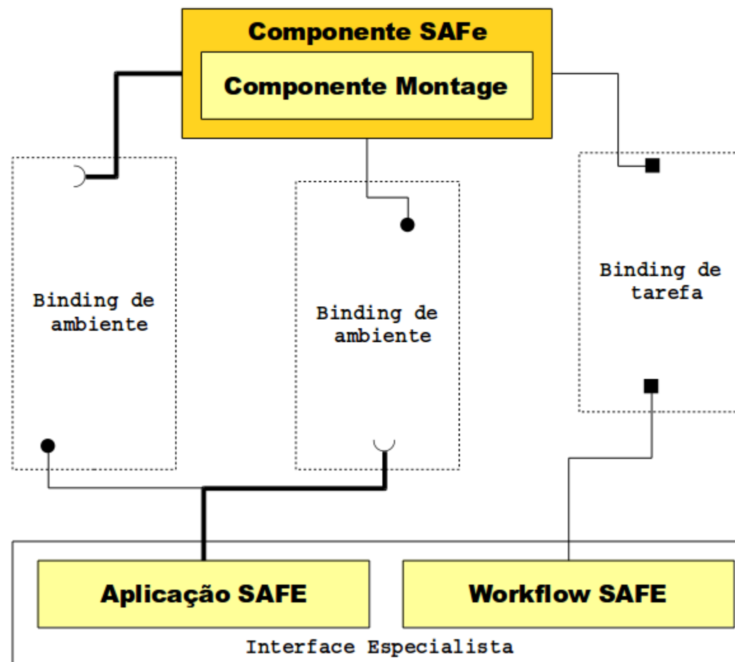


Figura 4.1: Relacionamento entre o componente Montage encapsulado a aplicação do lado do especialista.

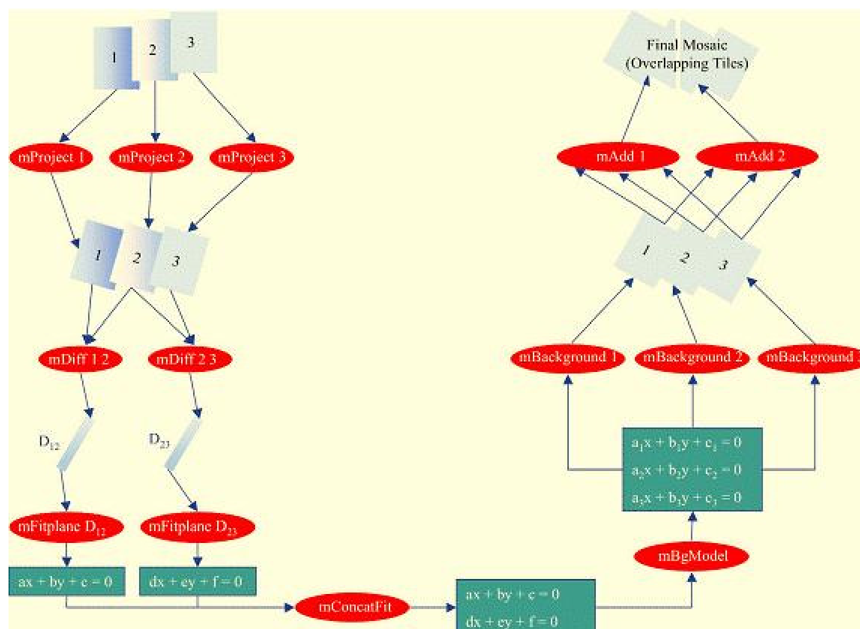


Figura 4.2: Exemplo de um workflow em Montage, mostrando a paralelização de seus componentes.

(Fonte: `http://montage.ipac.caltech.edu/docs/grid.html`)

4.1.2 *Workflows* do Montage

Workflows do Montage são formados pela composição dos componentes disponibilizados pela ferramenta a fim de atender um certo efeito pretendido. Esses componentes, como explicado anteriormente, trabalham com arquivos e parâmetros de entrada no formato *string* (no caso de arquivos, a *string* é a localização do arquivo no sistema de arquivos). Ao compor um *workflow*, o desenvolvedor deve especificar o fluxo de execução dos componentes (sequencial, paralelo, ramificado, repetido, etc), sincronizando os arquivos de entrada e saída de cada tarefa. Componentes do Montage irão ler os arquivos necessários para a sua execução. A não presença de um determinado arquivo resultará em um erro ou até mesmo na produção de uma saída inconsistente.

A Figura 4.2 demonstra um *workflow* o qual faz uso dos componentes `mProject`, `mDiff`, `mFitplane`, `mConcatFit`, `mBgModel`, `mBackground` e `mAdd` (elipses). Nesse exemplo, o *workflow* tem como entrada as imagens denotadas por `1`, `2` e `3` e gera como saída um mosaico final. O único passo onde os componentes não podem executar em paralelo é na execução do componente `mBgModel`, o qual implementa a computação do modelo de *background*. O `mProject`, por exemplo, é executado em três instâncias próprias, onde cada instância trabalha com uma imagem diferente, logo no início do *workflow*. Os retângulos da figura denotam os cálculos que são feitos internamente nos componentes.

4.1.3 Implementação da Aplicação SAFe do Montage

Para validar o SAFe, escolhemos dois *workflows* pré-existentes, disponibilizados no próprio site do projeto Montage. O primeiro é o *workflow* que gera o mosaico da galáxia **M101**[4]. O segundo *workflow* refere-se ao aglomerado de estrelas **Plêiades**[5].

Para gerar ambos os *workflows*, faz-se necessária uma aplicação compatível com o SAFe que deve ser suficientemente expressiva para atender as necessidades do especialista. No caso, o especialista irá trabalhar no nível de abstração dos componentes disponíveis pelo Montage, conectando suas portas, inserindo entradas, acompanhando o processo de execução e esperando a saída (mosaico).

Uma abordagem alternativa seria o especialista trabalhar em um nível de

[4] A galáxia M101, também conhecida como galáxia de Catavento, é uma galáxia em espiral situada a 21 milhões de anos-luz da Terra, descoberta em 1781. Seu diâmetro é de cerca de 170.000 anos-luz, um pouco maior que a Via Láctea, nossa galáxia.

[5] A Plêiades consiste em um grupo de estrelas situadas na constelação de Touro, a cerca de 400 anos-luz da Terra. Foi descoberto durante a antiguidade.

abstração ainda maior, onde apenas forneceria a localização de um repositório de imagens, algumas informações de configuração e então a aplicação faria a composição automática de componentes do Montage para produzir um mosaico com as imagens. Nesse caso, o especialista não precisaria nem mesmo conhecer a semântica dos componentes do Montage, que tem correspondência um-para-um com os componentes oferecidos na HPC Shelf, tampouco preocupar-se em como compô-los para gerar o mosaico pretendido. Essa segunda alternativa seria a mais adequada para o que se propõe ser um especialista da HPC Shelf, uma das características que distinguem o SAFe de outros sistemas de gerenciamento de *workflows*. Porém, usa-se a primeira alternativa neste estudo de caso, tendo em vista a sua maior simplicidade e possibilidade de exercitar algumas características particulares do SAFe.

Para este estudo de caso, foi desenvolvida a aplicação *Montage Expert GUI* (doravante chamada MoEx) a qual encontra-se implementada em Java *Swing* de forma a ser compatível com o SAFe (ou seja, derivação a partir da classe `HShelfApplication`), onde o especialista pode montar fluxos de execução baseados nos componentes do Montage. Cada componente foi imaginado de acordo com o padrão de componentes do SAFe, onde oferece portas de ambiente e de tarefas. O especialista organiza visualmente os componentes que deseja e a aplicação gera *workflows* de forma automática, em SAFeSWL. Uma vez gerado o *workflow*, o próximo passo é sua execução e acompanhamento através da interface gráfica do MoEx. Note que o uso do SAFeSWL é transparente para o especialista.

A Figura 4.3 apresenta a interface gráfica desenvolvida para o especialista executar *workflows* compatíveis com os componentes do Montage, fazendo uso das abstrações do SAFe e da nuvem HPC Shelf. No lado esquerdo da aplicação, o especialista escolhe quais são os componentes que farão parte do *workflow*. É possível ainda ver quais são as portas disponíveis para cada componente. No lado direito, os componentes escolhidos são organizados em *atos*, os quais definem o fluxo de controle de execução (componentes em um mesmo ato são executados em paralelo). Uma vez escolhidos os componentes, o especialista deve ligar suas portas de ambiente usuárias e provedores entre si, além de ligar suas portas de tarefas com as portas correspondentes do componente *workflow*. O especialista também tem a opção de reusar *workflows* previamente construídos.

Apesar de termos escolhido implementar dois exemplos de *workflows* (M101 e Plêiades), é importante ressaltar que a partir da aplicação MoEx é possível criar outros *workflows*, desde que seus componentes já tenha sido compatibilizados com o

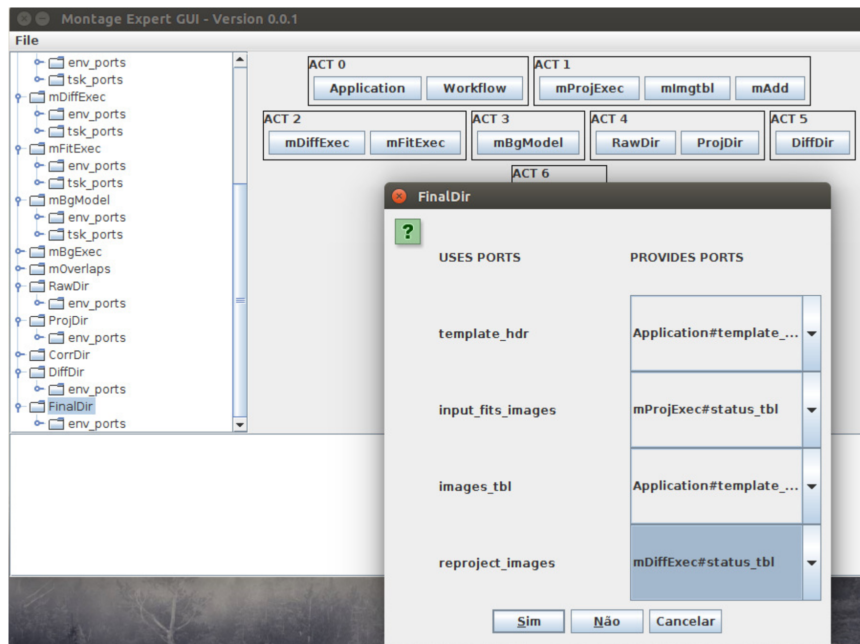


Figura 4.3: Interface gráfica para geração de workflows Montage, MoEx

SAFe. Nas próximas seções, são apresentados detalhes sobre o fluxo de orquestração e dos componentes SAFe envolvidos nos *workflows* M101 e Plêiades.

Uma outra observação importante é que, nesta implementação, a qual usa os componentes Montage diretamente encapsulados em componentes HPC Shelf, não há propriamente componentes abstratos e, por consequência, contratos contextuais, uma vez que os componentes abstratos não possuem parâmetros de contexto. Dessa forma, cada componente abstrato possui uma única implementação, uma vez que não há parâmetros de contexto para variação de implementações (derivação de contratos contextuais). Sendo assim, a apresentação que se segue abstrai-se dos contratos contextuais. Por outro lado, o próximo estudo de caso (Map/Reduce) apresentará parâmetros de contexto e, portanto, componentes abstratos e contratos contextuais.

4.1.4 O *Workflow* M101

O *workflow* M101 é formado pela composição dos componentes do Montage apresentado na Tabela 4.1. O produto final é a imagem da galáxia em espiral M101, em formato JPEG.

Como mencionado anteriormente, cada um dos componentes necessita de arquivos de entradas ou diretórios de entrada e geram arquivos de saída ou escrevem vários arquivos de saída em um único diretório. No projeto para os componentes da HPC Shelf, diretórios de arquivos são implementados como componentes da espécie

mImgtbl	gera metadados em uma tabela a partir de imagens de um diretório.
mProjExec	reprojeta as imagens de um diretório, usando tabela de metadados.
mOverlaps	sobrepõe imagens de cenário para um efeito mais nítido.
mDiffExec	cria imagens de diferença entre as sobreposições.
mFitExec	calcula coeficientes para cada imagem de diferença.
mBgModel	cria uma tabela de correções através as imagens de cenário.
mBgExec	aplica o cenário recalculado sobre as imagens projetadas.
mAdd	agrupa todas as correções anteriores para a imagem final.
mJpeg	gera um arquivo JPEG da imagem final.

Tabela 4.1: Descrição dos componentes Montage usados no M101.

fontes de dados. Sendo assim, além das instâncias dos componentes de computação citados, existem, no *workflow* M101, instâncias de componentes fontes de dados para representarem repositórios de imagens denominadas `rawDir`, `projDir`, `diffDir`, `corrDir` e `finalDir`.

Além dos componentes, é necessário definir os tipos de portas de ambiente necessários para comunicação entre os componentes dos *workflows* Montage. Para *workflows* do Montage, incluindo o M101, são usados três tipos de portas:

- *HdrPort*: serve para comunicação de parâmetros de configuração do mosaico da aplicação para os componentes de solução que calcularão os mosaicos;
- *DirPort*: serve para acesso aos repositórios de imagens;
- *TblPort*: serve para oferecer sínteses de informações relevantes sobre as imagens localizadas por um repositório recentemente processado por um componente.

Nas descrições a seguir, a nomenclatura das portas de ambiente seguirá a convenção definida por `*-hdr-port`, `*-dir-port` e `*-tbl-port`, notadamente para portas dos tipos *HdrPort*, *DirPort* e *TblPort*, respectivamente, onde a máscara $\star$ é atribuída de acordo com o significado da porta.

Quanto às portas de tarefa, há somente um tipo, chamado *GoPort*, com uma única ação, chamada `go`. Para portas desse tipo, usaremos a convenção de nomenclatura `*-go-port`, onde a máscara $\star$ serve, quando necessário, para distinguir múltiplas portas de tarefas desse tipo que serão conectadas a diferentes

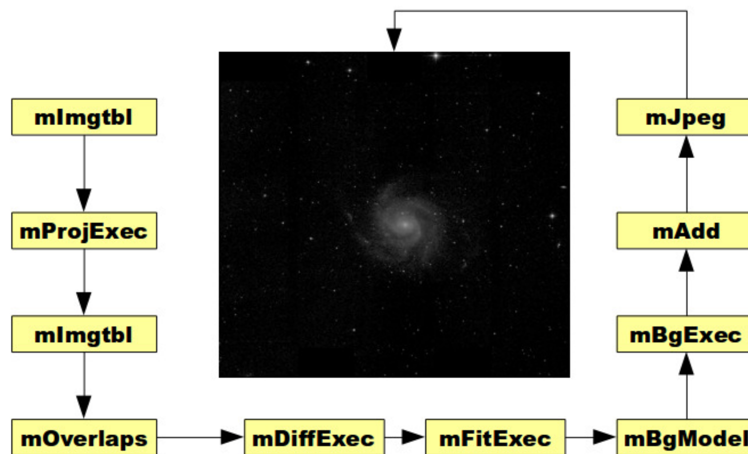


Figura 4.4: Visão simples do fluxo do workflow M101, com a imagem resultante da galáxia ao centro.

Componente `mImgtbl`	
Portas de Ambiente Provedoras	
`tbl-port`	prover descrição (síntese) das imagens do repositório de entrada
Portas de Ambiente Usuárias	
`dir-port`	ler imagens de entrada de um repositório
Portas de Tarefas	
`go-port`	inicia a computação através da ação `go`

Tabela 4.2: Portas genéricas para o componente `mImgtbl`

portas de tarefas de outros componentes. Uma vez que os componentes de solução do Montage possuem apenas uma única porta de tarefa, esta será denominada `go-port`. De fato, apenas o componente *workflow* possuirá múltiplas portas do tipo *GoPort*, a fim de conectar-se com os componentes de solução.

A Figura 4.4 demonstra o fluxo de orquestração dos componentes do *workflow* M101. Nota-se que se trata de um exemplo de *workflow* sequencial, onde cada componente espera o término do componente anterior para realizar a sua execução. A execução se dá através da ativação da ação `go` da porta `go-port`, presente em cada um dos componentes. Por sua vez, a Figura 4.5 demonstra a arquitetura do *workflow*, porém com ênfase apenas nas portas de ambiente dos componentes que o compõem, as quais são descritas a seguir.

A Tabela 4.2 apresenta as portas do componente `mImgtbl`. Ao ter a sua ação go ativada, esse componente processa as imagens de um repositório, através da porta `dir-port`. Ao final, são geradas informações sobre esses arquivos, que serão lidas por uma instância de um outro componente através da porta provedora `tbl-port`.

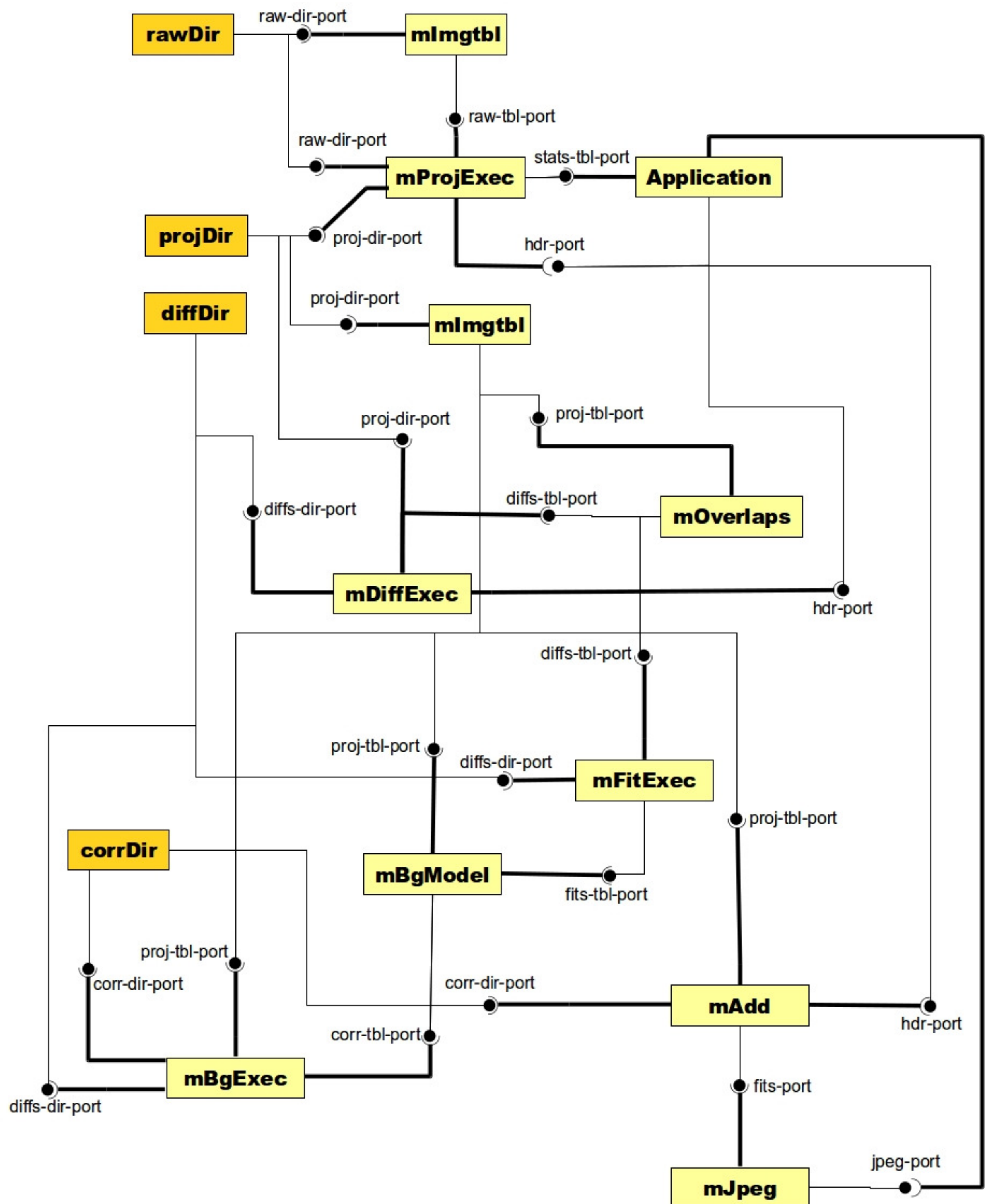


Figura 4.5: Visão geral do workflow M101 sem as portas de tarefas.

O *workflow* M101 inclui duas instâncias de mImgtbl, denominadas mImgtbl_raw e mImgtbl_proj. A primeira é executada sobre o repositório rawDir, que é o repositório de entrada do *workflow* M101, produzindo informações sobre esse repositório que serão oferecidas através da porta tbl-port para o componente mProjExec. Após a execução do mProjExec, a instância mImgtbl_proj é executada sobre o repositório projDir, gerando informações que serão utilizadas pelo componente mOverlaps.

Componente mProjExec	
Portas de Ambiente Provedoras	
stats-tbl-port	prover informações estatísticas sobre a reprojeção de cada imagem
Portas de Ambiente Usuárias	
hdr-port	ler configuração do mosaico, fornecida pela aplicação
raw-dir-port	ler de um repositório de imagens a serem projetadas
raw-tbl-port	ler descrições dos arquivos de imagens a serem projetadas
proj-dir-port	escrever em um repositório de imagens projetadas
Portas de Tarefas	
go-port	inicia a computação através da ação go

Tabela 4.3: Portas genéricas para o componente mProjExec

A Tabela 4.3 apresenta as portas do componente mProjExec. Esse componente lê o repositório rawDir, de imagens a serem reprojetadas, a partir da porta usuária raw-dir-port, usando as descrições dos arquivos fornecidas por mImgtbl_raw através da porta usuária raw-tbl-port. Como resultado, escreve imagens reprojetadas em um outro repositório, chamado projDir, conectado através da porta usuária proj-dir-port. Através da porta hdr-port, o componente mProjExec obtém parâmetros da aplicação que configuram o mosaico a ser gerado, os quais são necessários para sua computação.

Além de gerar imagens no repositório projDir, o componente mProjExec também gera dados estatísticos e de tempo de reprojeção de cada imagem através da porta stats-tbl-port. Essa porta está conectada a uma porta usuária da aplicação, a qual pode apresentar as informações estatísticas ao especialista.

Componente **mOverlaps**	
Portas de Ambiente Provedoras	
diffs-tbl-port	prover descrições das imagens que se sobrepõem
Portas de Ambiente Usuárias	
proj-tbl-port	ler descrições das imagens de entrada
Portas de Tarefas	
go-port	iniciar a computação através da ação go

Tabela 4.4: Portas genéricas para o componente mOverlaps

A Tabela 4.4 apresenta as portas do componente mOverlaps, que gera a descrição das imagens que se sobrepõem (porta provedora diffs-tbl-port) a partir da descrição de um conjunto de imagens reprojetadas geradas por mImgtbl_proj (porta usuária proj-tbl-port) calculadas anteriormente pelo componente mProjExec e acessíveis pela porta usuária proj-dir-port.

Componente **mDiffExec**	
Portas de Ambiente Provedoras	
-	
Portas de Ambiente Usuárias	
hdr-port	ler configuração do mosaico, fornecida pela aplicação
proj-dir-port	ler imagens projetadas de um repositório
diffs-tbl-port	ler descrições das imagens diferença de entrada
diffs-dir-port	escrever imagens diferença (saída) em um repositório
Portas de Tarefas	
go-port	iniciar a computação através da ação go

Tabela 4.5: Portas genéricas para o componente mDiffExec

A Tabela 4.5 apresenta as portas do componente mDiffExec. Esse componente realiza o cálculo de imagens de diferença a partir do repositório de imagens projDir, lido através da porta usuária proj-dir-port, com uso das descrições anteriormente geradas por mOverlaps, lidas através da porta usuária diffs-tbl-port. As imagens resultantes são armazenadas no repositório diffDir, de imagens diferença, acessado através da porta usuária diffs-dir-port.

Componente **mFitExec**	
Portas de Ambiente Provedoras	
`fits-tbl-port`	prover coeficientes de preenchimento de plano (*plane-fitting*)
Portas de Ambiente Usuárias	
`diffs-dir-port`	ler repositório de imagens diferença
`diffs-tbl-port`	ler descrições das imagens diferença do repositório de entrada
Portas de Tarefas	
`go-port`	iniciar a computação através da ação `go`

Tabela 4.6: Portas genéricas para o componente `mFitExec`

A Tabela 4.6 apresenta as portas do componente `mFitExec`. Esse componente calcula coeficientes de preenchimento de planos e os fornece através de sua porta provedora `fits-tbl-port` para o componente `mBgModel`. O cálculo é realizado a partir do repositório `diffDir`, anteriormente gerado por `mDiffExec`, acessado através da porta usuária `diffs-dir-port`. Usa ainda informações sobre as imagens diferença geradas por `mOverlaps`, através da porta `diffs-tbl-port`.

Componente **mBgModel**	
Portas de Ambiente Provedoras	
`corr-tbl-port`	prover tabela de correções globais
Portas de Ambiente Usuárias	
`proj-tbl-port`	ler descrição de imagens de entrada
`fits-tbl-port`	ler coeficientes de preenchimento de plano
Portas de Tarefas	
`go-port`	iniciar a computação através da ação `go`

Tabela 4.7: Portas genéricas para o componente `mBgModel`

A Tabela 4.7 apresenta as portas do componente `mBgModel`, o qual lê informações sobre imagens reprojetadas geradas por `mImgtbl_proj` (porta usuária `proj-tbl-port`), bem como os coeficientes de preenchimento de plano gerados por `mFitExec` (porta usuária `fits-tbl-port`), e calcula uma tabela de correções globais, oferecida através da porta `corr-tbl-port` para o componente `mBgExec`, da próxima etapa.

Componente **mBgExec**	
Portas de Ambiente Provedoras	
-	
Portas de Ambiente Usuárias	
proj-dir-port	ler repositório de imagens projetadas
proj-tbl-port	ler descrições de imagens projetadas
corr-dir-port	escrever imagens corrigidas no repositório
corr-tbl-port	ler tabela de correções globais a serem aplicadas
Portas de Tarefas	
go-port	inicia a computação através da ação go

Tabela 4.8: Portas genéricas para o componente mBgExec

A Tabela 4.8 apresenta as portas do componente mBgExec, o qual executa a correção das imagens do repositório projDir, acessado através da porta usuária proj-dir-port, usando a tabela de correções globais calculada por mBgModel, que é acessada por meio da porta usuária corr-tbl-port. As imagens corrigidas são depositadas no repositório corrDir, através da porta corr-dir-port, que será acessado pelo componente mAdd na próxima etapa do *workflow*.

Componente **mAdd**	
Portas de Ambiente Provedoras	
fits-port	prover a imagem no formato FITS
Portas de Ambiente Usuárias	
corr-dir-port	ler imagens do repositório de imagens projetadas e corrigidas
proj-tbl-port	ler informações sobre as imagens no repositório de entrada
hdr-port	ler configuração do mosaico, fornecida pela aplicação
Portas de Tarefas	
go-port	inicia a computação através da ação go

Tabela 4.9: Portas genéricas para o componente mAdd

A Tabela 4.9 apresenta as portas do componente mAdd. A partir das imagens reprojetadas e com pano de fundo corrigido geradas por mBgExec no repositório corrdir, acessado através da porta usuária corr-dir-port, e usando as descrições das imagens obtidas por acesso à porta usuária proj-tbl-port, calculadas por mImgtbl_proj. Esse componente calcula o mosaico final, fornecido através da porta provedora fits-port, do tipo *FitsPort*. Esse mosaico é gerado no formato FITS (*Flexible Image Transport System*)[6]. Usa também, para o cálculo

[6]Formato padrão em astronomia, patrocinado pela NASA (*National Aeronautics and Space*

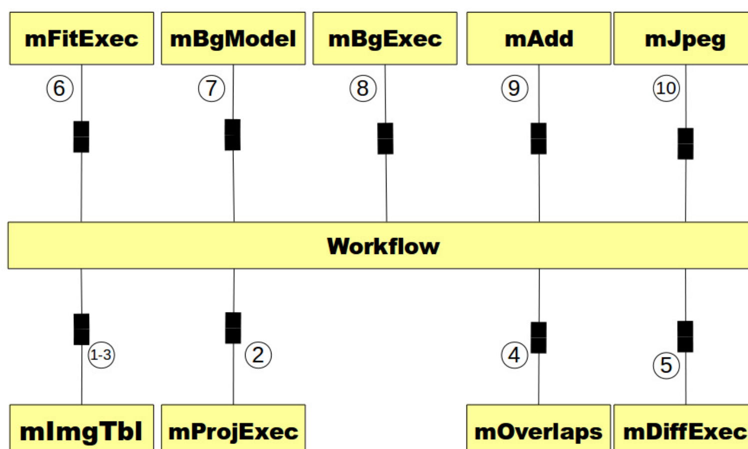


Figura 4.6: O componente workflow conectado às portas de tarefas dos outros componentes do Montage.

do mosaico final, as informações fornecidas pela aplicação sobre o mosaico a ser gerado, obtidas através da porta `hdr-port`.

Componente `mJpeg`	
Portas de Ambiente Provedoras	
`jpeg-port`	prover imagem no formato JPEG (binário)
Portas de Ambiente Usuárias	
`fits-port`	ler imagem no formato FITS
Portas de Tarefas	
`go-port`	inicia a computação através da ação `go`

Tabela 4.10: Portas genéricas para o componente `mJpeg`

Finalmente, a Tabela 4.10 apresenta as portas do componente `mJpeg`, o qual tem o propósito de transformar o mosaico no formato FITS (porta usuária `fits-port`) em uma imagem no formato JPEG (porta provedora `jpeg-port`), que possa ser visualizado pelo especialista através da aplicação.

4.1.5 Conexão do M101 com o Componente *Workflow*

Ao componente *workflow* de M101, interessam apenas as portas de tarefas disponibilizadas pelos componentes de solução do M101 (aquelas denominadas `go-port`). As portas de tarefas de cada componente serão ativadas durante a execução do fluxo de orquestração do *workflow* descrito em SAFeSWL.

As portas de tarefas são do tipo *GoPort*, com uma única ação denominada *go*. Ao serem ativadas pelo componente *workflow*, as ações *go* iniciam a

Administration). Mais informações em: http://fits.gsfc.nasa.gov/.

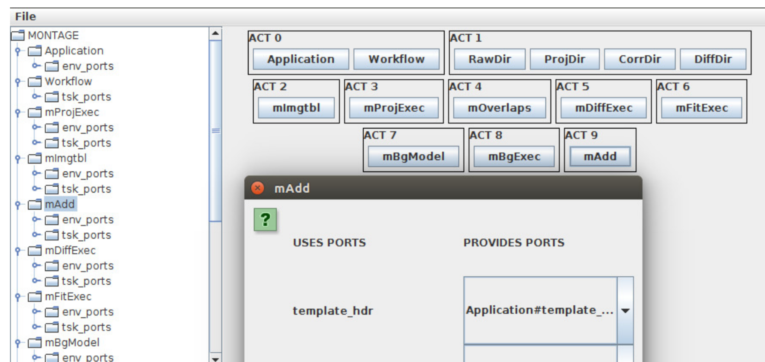


Figura 4.7: Processo de criação do workflow M101 através da aplicação MoEx.

computação associada ao componente Montage em questão. Os dados de entrada e configurações necessários à execução do componente podem ser obtidos das portas de ambiente usuárias apropriadas tão logo estejam disponíveis, sem necessidade de ativação prévia da ação *go* por parte do componente *workflow*. Dados de saída a serem fornecidos para outros componentes por meio de portas de ambiente (usuárias ou provedoras) podem ser produzidos incrementalmente, à medida que o componente executa, de modo que também podem ser lidos incrementalmente pelos componentes que dependerão desses dados para sua execução. Usando esses artifícios, busca-se a sobreposição entre comunicação e computação na execução do *workflow*, potencialmente melhorando o seu desempenho.

A Figura 4.6 demonstra, de forma bem simples, a conexão entre as portas de tarefas dos componente *workflow* e as portas de tarefas dos componentes do M101. Os números na ilustração ressaltam a ordem de ativação das ações *go* de cada componente até a geração da imagem final.

4.1.6 Criação do M101 na Aplicação MoEx

O *workflow* M101 é apenas um exemplo que pode ser construído através da interface MoEx. O especialista é livre para configurar qualquer arranjo de componentes disponíveis no catálogo de componentes Montage. No entanto, para fins deste estudo de caso, o M101 é um caso que demonstra um *workflow* sequencial, já testado por outros *frameworks* e com uma saída bem definida.

A Figura 4.7 apresenta o processo parcial da criação do *workflow* M101 na aplicação MoEx. O especialista deve escolher dentre os componentes do catálogo localizado no lado esquerdo e decidir o momento no qual serão executados através da associação com valores inteiros que representam *atos*, de modo que componentes no mesmo ato devem executar paralelamente. Em uma abordagem em mais alto nível,

o especialista também pode carregar um conjunto de *workflows* já definidos pelo desenvolvedor de aplicações. Ao especialista caberia apenas fornecer os parâmetros iniciais que, no caso desta aplicação, seriam o endereço do repositório de imagens fonte (rawDir) e os dados requeridos pelas portas hdr-port.

Após montado o *workflow* na interface gráfica do MoEx, o especialista pode gerar o arquivo SAFeSWL arquitetural (informações sobre os componentes e portas) e de orquestração (informações sobre a lógica de execução), de forma transparente, apenas clicando na barra de menu superior. Feito isso, o *workflow* está pronto para execução. O especialista não tem acesso ao código SAFeSWL gerado.

```
<architecture>
 <application id="0" name="app-sample-example">
     <uses_port id="14" name="status-port" id_component="0"/>
     <uses_port id="15" name="jpeg-port" id_component="0"/>
     <provides_port id="16" name="hdr-port" id_component="0"/>
 </application>
     ...
```

Listagem 4.1: Componente aplicação.

A Listagem 4.1 apresenta o trecho da linguagem arquitetural para o *workflow* M101 relativo às portas do componente aplicação o qual oferece dados de configuração do mosaico a ser gerado via porta provedora hdr-port, a qual, como visto anteriormente, será conecta a portas usuárias de mesmo nome de alguns dos componentes de solução do M101. O componente aplicação possui ainda duas portas usuárias: status-port, que será conectada à porta provedora stats-tbl-port do componente mProjExec, e jpeg-port, que deve receber a imagem final gerada pelo *workflow*. A porta jpeg-port tem como parceira a porta provedora de mesmo nome oferecida ao ambiente pelo componente mJpeg do *workflow* M101.

```
...
 <workflow id="1" name="m101-driver">
   <provides_port id="21" name="port_SAFeSWL" id_component="1"/>
   <provides_port id="22" name="port_Go" id_component="1"/>
   <task_port id="23" name="mimgtbl_raw-go-port" id_component="1"/>
   <task_port id="24" name="mprojexec-go-port" id_component="1"/>
   <task_port id="23" name="mimgtbl_proj-go-port" id_component="1"/>
   <task_port id="26" name="madd-go-port" id_component="1"/>
   <task_port id="27" name="mbgmodel-go-port" id_component="1"/>
   <task_port id="28" name="mbgexec-go-port" id_component="1"/>
   <task_port id="29" name="mdiffexec-go-port" id_component="1"/>
```

```
  <task_port id="30" name="mfitexec-go-port" id_component="1"/>
  <task_port id="31" name="moverlaps-go-port" id_component="1"/>
  <task_port id="32" name="mjpeg-go-port" id_component="1"/>
 </workflow>
     ...
```

Listagem 4.2: Componente *workflow*.

A Listagem 4.2 apresenta um trecho da linguagem arquitetural relativo às portas do componente *workflow*. Podemos notar que esse componente apenas se conecta com as portas de tarefas dos outros componentes do M101, como explicado anteriormente.

```
...
<computation name="mimgtbl_raw" id="2">
     <uses_port id="33" name="dir-port" id_component="2"/>
     <provides_port id="34" name="tbl-port" id_component="2"/>
     <task_port name="go-port" id="32" id_component="2">
           <action id="321" name="go"/>
     </task_port>
</computation>
     ...
```

Listagem 4.3: Componente workflow.

A Listagem 4.3 refere-se ao código arquitetural do componente `mImgtbl_raw`, explicado anteriormente. Podemos notar a definição de sua porta usuária `dir-port`, a qual deverá ser conectada ao componente `rawDir` com fim de ler as imagens fontes e produzir os dados requeridos pela porta provedora `tbl-port`. Sua única porta de tarefas, `go-port`, será conectada a uma porta de tarefas do *workflow*. Analogamente, o mesmo processo para geração do XML ocorre com os demais componentes.

```
...
<env_binding>
     <uses_port id="15" name="jpeg-port" id_component="0"/>
     <provides_port id="103" name="jpeg-port" id_component="10"/>
</env_binding>
     ...
```

Listagem 4.4: *Binding* de ambiente entre componente aplicação e `mJpeg`.

A Listagem 4.4 mostra a declaração do *binding* entre a porta usuária `jpeg-port` da aplicação e a porta provedora homônima do componente `mJpeg`. Declarações análogas declaram outros *bindings* de ambiente de M101.

```
...
 <task_binding>
      <left_peer id="23" name="mimgtbl_raw-go-port" id_component="1"/>
      <right_peer id="33" name="go-port" id_component="2"/>
  </task_binding>
      ...
```

Listagem 4.5: *Binding* de tarefa entre componente *workflow* e mImgtbl.

A Listagem 4.5 apresenta a declaração do *binding* de tarefas entre a porta mimgtbl_raw-go-port do componente *workflow* e a porta de tarefas go-port, do componente mImgtbl_raw. Os demais *bindings* de tarefa entre o componente *workflow* e os demais componentes de solução de M101 são analogamente definidos.

4.1.7 Execução do M101 na Aplicação MoEx

Para executar o *workflow*, é necessário que o arquivo de orquestração seja gerado pela aplicação. Fica a cargo do provedor de aplicações a forma como sintetizar o arquivo de orquestração. No caso do MoEx, a interface organiza os componentes em *atos*, onde um ou mais componentes podem ser colocados em um mesmo *ato*. Caso o componente seja da espécie computação, o especialista irá escolher, naquele *ato*, qual ação será executada. De posse dessas informações a interface gera então o arquivo SAFeSWL necessários a execução do *workflow*.

De forma abstrata, a de orquestração segue o seguinte fluxo:

```
seq { mimgtbl-raw-go-port;
      mprojexec-go-port;
      mimgtbl-proj-go-port;
      moverlaps-go-port;
      mdiffexec-go-port;
      mfitexec-go-port;
      mbgmodel-go-port;
      madd-go-port;
      mjpeg-go-port }
```

O fluxo acima apresenta de forma abstrata a lógica do arquivo de orquestração. Podemos notar a execução sequencial (seq) de várias ativações de ações de portas de tarefas dos componentes do *workflow* M101.

```
...
      <sequence_oper>
```

```
<!-- mImgtbl-go -->
<operation>
      <invoke_oper action="compute" id="321" />
</operation>

<!-- mProjExec-go -->
<operation>
      <invoke_oper action="compute" id="1121" />
</operation>

<!-- mImgtbl-go -->
<operation>
      <invoke_oper action="compute" id="321" />
</operation>

<!-- mOverlaps -->
<operation>
      <invoke_oper action="compute" id="921" />
</operation>

<!-- mDiffExec-go -->
<operation>
      <invoke_oper action="compute" id="721" />
</operation>

<!-- mFitExec-go -->
<operation>
      <invoke_oper action="compute" id="821" />
</operation>

<!-- mBgModel-go -->
<operation>
      <invoke_oper action="compute" id="521" />
</operation>

<!-- mBgExec-go -->
<operation>
      <invoke_oper action="compute" id="621" />
</operation>

<!-- mAdd-go -->
<operation>
```

```
                <invoke_oper action="compute" id="421" />
            </operation>

            <!-- mJpeg-go -->
            <operation>
                <invoke_oper action="compute" id="1021" />
            </operation>

        </sequence_oper>
        ...
```

Listagem 4.6: Trecho do arquivo de orquestração.

A Listagem 4.6 apresenta o trecho principal do código de orquestração gerado pelo MoEx para o *workflow* M101. Nele podemos notar as chamadas aos códigos das ações `go` de cada componente, na ordem em que serão executados.

4.1.8 O *Workflow* Plêiades

O segundo *workflow* que usaremos para demonstrar o MoEx chama-se Plêiades, o qual gera um mosaico para a constelação da *Plêiades*. Esse *workflow* é formado pela composição dos componentes `mImgtbl`, `mProjExec`, `mAdd` e `mJpeg`, já explicados para o *workflow* M101. Como particularidade, esse *workflow* exercita a execução de vários *workflows* distintos em uma mesma aplicação para produzir o resultado.

Além dos componentes de computação, são necessários três componentes para representar repositórios para as imagens temporárias, notadamente da espécie *fonte de dados*, denominados `dss2rDir`, `dss2irDir` e `dss2bDir`. Cada repositório serve para armazenar as imagens processadas referentes a uma faixa de cor, respectivamente vermelho, infravermelho e azul. Cada faixa de cor será processada por um *workflow* distinto, doravante chamado *workflow* de projeção, variando apenas o repositório. Ao final, as imagens são sobrepostas, por meio de um quarto *workflow*, doravante chamado *workflow* de sobreposição, gerando a imagem final. Como ilustrado na Figura 4.8, cada repositório tem duas portas de ambientes provedoras: `proj-dir-port`, diretório interno que fornece imagens projetadas e `raw-dir-port`, diretório interno o qual trabalha com as imagens fontes.

Como já mencionado, uma característica deste estudo de caso é o uso de mais de um *workflow* para produzir o resultado. A Figura 4.9 apresenta a lógica de orquestração combinada dos componentes responsáveis pela computação dos *workflows* de Plêiades. Inicialmente, os três *workflows* de projeção (vermelho, infravermelho e azul) serão instanciados pela aplicação para executar em paralelo,

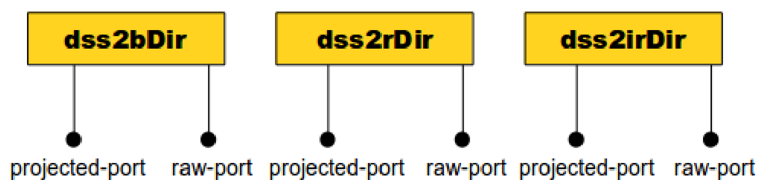


Figura 4.8: Componentes repositórios e suas portas provedoras.

possuindo a mesma arquitetura, variando apenas o repositório (`dss2bDir`, `dss2rDir` ou `dss2irDir`). Dentro de cada *workflow*, são acessadas as portas `proj-dir-port` e `raw-dir-port` de cada repositório, para ler as imagens fonte originais (*raw*) e escrever as imagens projetadas. Ao final, teremos três conjuntos de imagens, armazenadas em cada repositório. Então, usando o componente `mJpeg`, o *workflow* de projeção sobrepõe as imagens dos três repositórios.

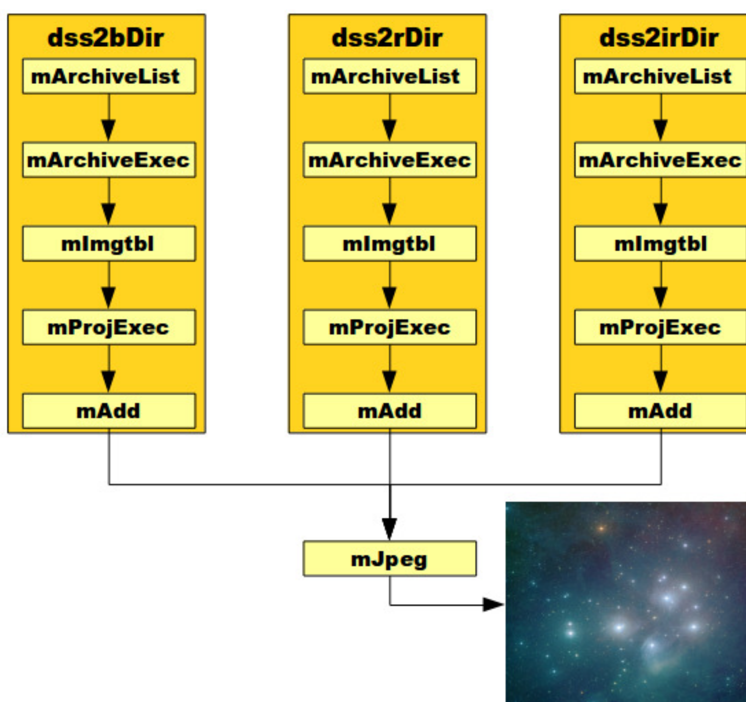


Figura 4.9: Os três fluxos para a execução do *workflow* Plêiades. No final as imagens em cada faixa de cor são combinadas pelo componente `mJpeg`.

Os componentes repositórios `dss2bDir`, `dss2rDir` e `dss2irDir` são responsáveis por realizar o *download*, de forma transparente, das imagens fontes do repositório on-line DSS2 (*Digitized Sky Survey*[7]). No Montage em si, isso é realizado através dos comandos `mArchiveList` e `mArchiveGet`.

[7] `http://archive.eso.org/dss/dss`

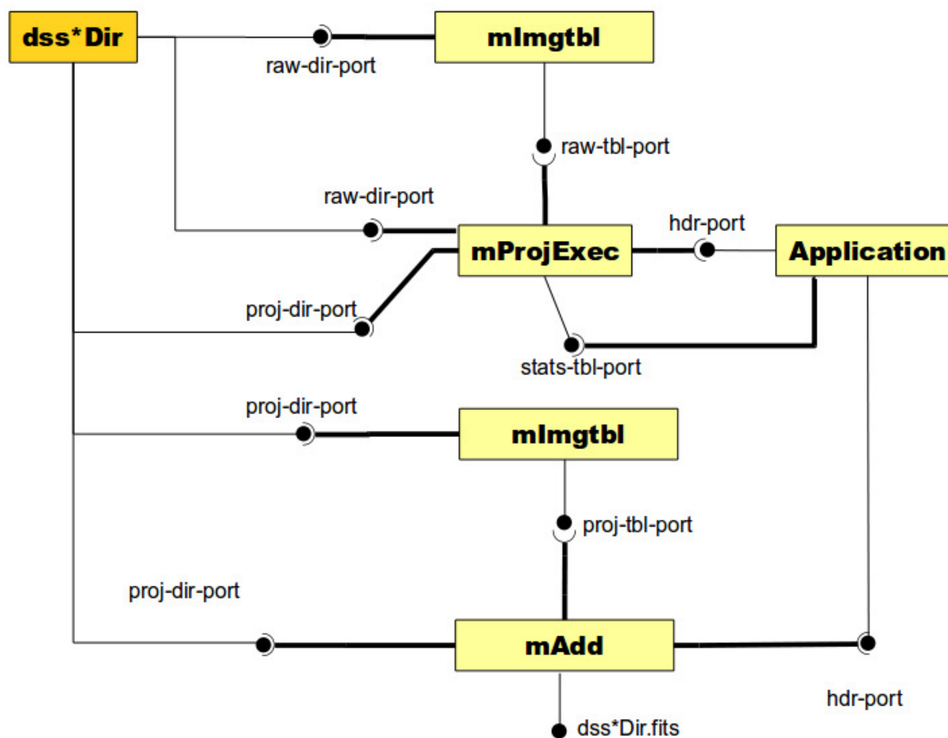


Figura 4.10: Visão geral do Plêiades sem as portas de tarefas.

A Figura 4.10 ilustra a arquitetura de um *workflow* de projeção do Plêiades. O repositório (`dss2bDir`, `dss2rDir` ou `dss2irDir`) em questão é representado por `dss*Dir`. A ilustração mostra apenas as portas de ambiente dos componentes de solução. Por sua vez, a Figura 4.11 apresenta a conexão das portas de tarefa dos componentes às portas de tarefa do componente *workflow*. Nota-se que, assim como em M101, o componente *workflow* interessa-se apenas pelas portas de tarefas dos componentes de solução, denominadas `go-port`.

4.1.9 Criação dos *Workflows* de Projeção do Plêiades na Aplicação MoEx

Assim como explicado no caso do *workflow* M101, para a composição de cada *workflow* do Plêiades, o especialista deverá escolher os componentes adequadas via interface gráfica do MoEx para geração do arquivo arquitetura e arquivo de orquestração de cada *workflow*. Nesta seção, será descrito apenas o código arquitetural em SAFeSWL referente aos *workflows* de projeção.

```
<architecture>
...
<application id="0" name="app-101">
  <uses_port id="13" name="port_SAFeSWL" id_component="0"/>
  <uses_port id="14" name="port_Go" id_component="0"/>
```

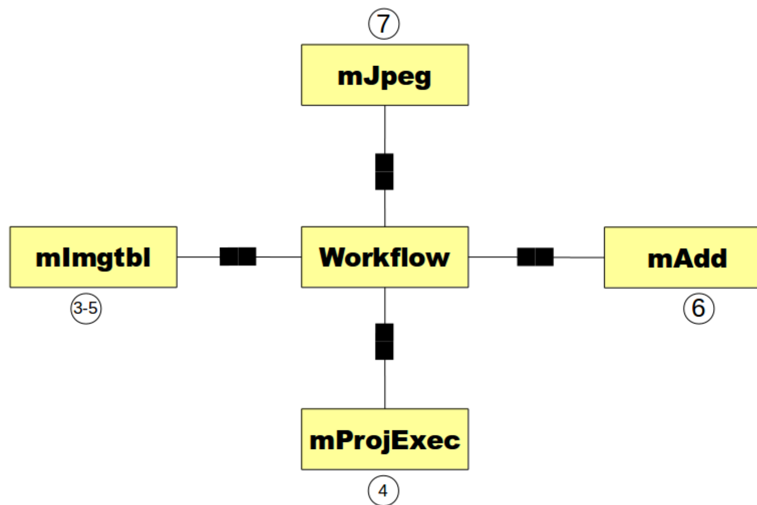


Figura 4.11: Componente *workflow* e suas conexões com os componentes do Pleiades em uma *workflow* de projeção.

```
  <provides_port id="12" name="hdr-port" id_component="0"/>
</application>
      ...
```

Listagem 4.7: Descrição arquitetural do componente aplicação no Plêiades.

A Listagem 4.7 apresenta a representação arquitetural do componente aplicação dentro de um *workflow* de projeção. Podemos notar a existência das portas usuárias padrões de acesso ao componente *workflow* e uma porta provedora, chamada `hdr-port`, para os parâmetros de configuração do mosaico, fornecido pela aplicação.

```
...
<workflow id="1" name="workflow-101">
  <provides_port id="21" name="port_SAFeSWL" id_component="1"/>
  <provides_port id="22" name="port_Go" id_component="1"/>
  <task_port id="23" name="mimgtbl1-go-port" id_component="1"/>
  <task_port id="24" name="mprojexec-go-port" id_component="1"/>
  <task_port id="23" name="mimgtbl2-go-port" id_component="1"/>
  <task_port id="25" name="madd-go-port" id_component="1"/>
</workflow>
      ...
```

Listagem 4.8: Descrição arquitetural do componente *workflow* no Plêiades.

A Listagem 4.8 apresenta o componente *workflow* e todas as suas portas de tarefas que serão conectadas às portas de tarefas dos componentes de solução. Além das portas de tarefas, o componente *workflow* também provê as portas de ambiente

para a aplicação, port_SAFeSWL e port_Go.

4.1.10 Execução do Plêiades na Aplicação MoEx

A execução do Plêiades no MoEx se assemelha a execução do M101 explicada anteriormente: o especialista escolhe os componentes que farão parte do *workflow* e define, para cada *ato* a interface gráfica, qual ação será executada. Para esta seção, explicaremos o arquivo de orquestração gerado pelo MoEx para um *workflow* de projeção, enfatizando as diferenças entre o *workflows* Plêiades e o M101.

seq { **mimgtbl1-go-port;**
mprojexec-go-port;
mimgtbl2-go-port;
madd-go-port }

A representação abstrata acima resume a lógica de execução de um dos *workflows* de projeção do Plêiades. Essa mesma lógica é executada três vezes, em uma mesma aplicação para três *workflows* distintos, uma para cada espectro de cor, gerando três imagens fits diferentes. A Listagem 4.9 apresenta o trecho principal desse código.

```
...
<operation>
      <sequence_oper>
            <!-- ACTIONS -->
            <!-- mImgtbl1-go -->
            <operation>
                  <invoke_oper action="compute" id="341" />
            </operation>

            <!-- mProjExec-go -->
            <operation>
                  <invoke_oper action="compute" id="1151" />
            </operation>

            <!-- mImgtbl2-go -->
            <operation>
                  <invoke_oper action="compute" id="346" />
            </operation>

            <!-- mAdd-go -->
            <operation>
                  <invoke_oper action="compute" id="451" />
```

```
            </operation>
      </sequence_oper>
</operation>
      ...
```

Listagem 4.9: Trecho do código de orquestração de um dos *workflows* do Plêiades.

4.1.11 Sobrecarga do SAFe sobre o M101 e Plêiades

Nesta seção é apresentada a comparação de tempos entre a execução dos *workflows* M101 e Plêiades fazendo uso apenas da linguagem *shell script* do Linux e a execução dos mesmos *workflows* M101 e Plêiades fazendo uso do SAFe. A Figura 4.12 mostra os códigos *shell script* dos dois *workflows* de teste (Plêiades à esquerda e M101 à direita). Note que o código do Plêiades executa em laço de três iterações. Sendo assim, implementamos o Plêiades no SAFe de duas formas: uma sequencial, assim como implementado pelo *shell script* e outra paralela, como explicado anteriormente.

```
#!/bin/bash
# Pleiades Image creation BASH script.
# Inseok Song, 2007
ts=$(date +%s%N)
  for bands in DSS2B DSS2R DSS2IR; do echo Processing ${bands};

  cd $bands;


  mImgtbl raw rimages.tbl ;
  mProjExec -p raw rimages.tbl ../pleiades.hdr projected stats.tbl ;
  mImgtbl projected pimages.tbl ;
  mAdd -p projected pimages.tbl ../pleiades.hdr ${bands}.fits ;
  cd .. ;
  done

  mJPEG -blue DSS2B/DSS2B.fits -1s 99.999% gaussian-log \
        -green DSS2R/DSS2R.fits -1s 99.999% gaussian-log \
        -red DSS2IR/DSS2IR.fits -1s 99.999% gaussian-log \
        -out DSS2_BRIR.jpg
```

```
#!/bin/bash
ts=$(date +%s%N)
mImgtbl rawdir images-rawdir.tbl
mProjExec -p rawdir images-rawdir.tbl
template.hdr projdir stats.tbl
mImgtbl projdir images.tbl
mOverlaps images.tbl diffs.tbl
mDiffExec -p projdir diffs.tbl template.hdr
diffdir
mFitExec diffs.tbl fits.tbl diffdir
mBgModel images.tbl fits.tbl corrections.tbl
mBgExec -p projdir images.tbl corrections.tbl
corrdir
mAdd -p corrdir images.tbl template.hdr final/
m101_mosaic.fits
mJPEG -gray final/m101_mosaic.fits 0s max
gaussian-log -out final/m101_mosaic.jpg
tt=$((($(date +%s%N) - $ts)/1000000)) ; echo
"Time taken: $tt"
```

Figura 4.12: Código *shell script* para o M101 e Plêiades.

(Adaptado de Inseok Song, 2007)

A Tabela 4.11 apresenta os tempos relativos as execuções no *shell script* e no SAFe (o tempo foi calculado como a média de três execuções em cada *workflow*). O interessante de usar *shell script* é que ele é a forma mais "pura" de executar os componentes do Montage, dando uma boa ideia da sobrecarga de qualquer *framework* que venha criar camadas sobre a ferramenta.

	M101	Plêiades Sequencial	Plêiades Paralelo
SAFe	23040ms	2117859ms	757061ms
Shell Script	18469ms	2089836ms	-
Diferença	4571ms (25%)	28033ms (1%)	-

Tabela 4.11: Tempos de execução do M101 e Plêiades.

A execução do M101 é a mais direta. A diferença entre a implementação *shell script* e SAFe foi de 25% (em torno de 4,5 segundos). Isso se deve por causa da criação de vários objetos Java, além da comunicação remota com os Serviços *Web* adequados, conexões com portas, leitura do arquivo arquitetural e de orquestração e a execução do arquivo de orquestração, o qual aciona as ações das portas de tarefas.

A execução do Plêiades sequencial frente ao *shell script* teve um desempenho menor em segundos brutos (em torno de 28 segundos), mas aceitável em percentual (cerca de 1% mais lento). A criação de quatro *workflows* gera o mesmo processo explicado para o M101 só que aproximadamente com quatro vezes mais de carga. Comparando com a execução do M101 (4,5 segundos de diferença), faz sentido que o Plêiades tenha alcançado a marca de 28 segundos (cerca de 6 vezes mais lento que o M101).

A execução do Plêiades paralelo foi bem eficiente, calculando os três níveis de cor ao mesmo tempo para gerar a imagem final com um ganho de aproximadamente $2,8$ vezes mais sobre sua mesma versão sequencial e um pouco menos sobre sua versão sequencial em *shell script*. Não implementamos uma versão em *shell script* paralela visto que o objetivo era apenas demonstrar a sobrecarga do SAFe. Uma versão paralela em *shel script* seria obviamente mais rápida que a mesma versão em SAFe.

É importante lembrar que otimização do SAFe não é ainda objetivo deste trabalho. A versão desta Tese ainda será melhorada com o passar do tempo visto que diversas aplicações ainda serão implementadas. Vislumbram-se melhorias na estrutura de dados que armazena informações sobre a linguagem SAFeSWL e na comunicação com componentes remotos, não fazendo uso apenas de Serviços *Web*, mas outros protocolos de comunicação com menor sobrecarga.

4.2 Estudo de Caso 2: Processamento Map/Reduce

O Map/Reduce é um modelo para *frameworks* de processamento paralelo voltados para o processamento em larga escala de grandes volumes de dados, o qual utiliza um padrão de programação paralela bastante simples que tem suas origens na programação funcional. Foi originalmente proposto e implementado pela *Google Inc.* [Dean e Ghemawat 2008]. Seu sucesso na aplicação a um grande número de problemas levou ao surgimento de várias outras implementações, muitas das quais introduzindo extensões ao modelo original, tais como MR-MPI [Plimpton e Devine 2011] e Hadoop [Apache Hadoop Project 2013]. O Hadoop, projeto da *Apache*

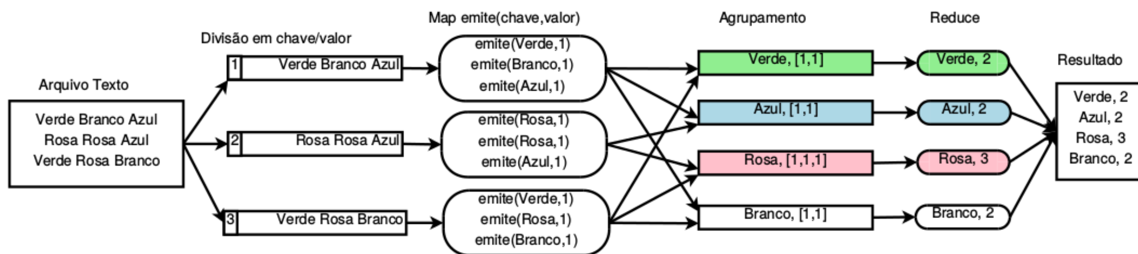


Figura 4.13: Exemplo clássico de contagem de palavras do Map/Reduce.

Foundation, é atualmente reconhecido como a implementação mais disseminada, com uma extensa base de usuários e utilização em vários projetos de pesquisa.

Um típico programa Map/Reduce é constituído de uma função de *mapeamento*, a qual implementa uma algoritmo de filtragem, transformação e ordenação sobre um conjunto de dados representado por um conjunto de pares chave/valor, e uma função de *redução*, a qual implementa uma operação de sumarização dos dados produzidos na etapa de mapeamento. As tarefas relativas a paralelização, tolerância a falhas, segurança, disponibilidade, concorrência e escalonamento são todas de responsabilidade do *framework* Map/Reduce, restando ao desenvolvedor, em sua forma mais básica, apenas a definição das funções de mapeamento e redução, bem como o ajuste de parâmetros que regulam a alocação de recursos computacionais e como as funções de mapeamento e redução devem ser mapeados a esses recursos.

A forma mais intuitiva de explicar o processamento Map/Reduce é através de um exemplo simples e clássico para essa finalidade, o contador de ocorrências de palavras em um texto. A Figura 4.13 apresenta uma ilustração de uma instância simples desse tipo de processamento. Seja um arquivo texto dado como entrada formado por um conjunto de palavras determinado: *verde*, *branco*, *azul* e *rosa*. Esse texto encontra-se agrupado em "frases", representadas por linhas do arquivo texto. O objetivo é contar quantas palavras existem para cada cor. O passo inicial é agrupar as frases em tuplas de *chave-valor* onde a chave é um inteiro (número da linha) e o valor é a frase correspondente. De acordo com as chaves, essas frases são distribuídas entre um conjunto de agentes de mapeamento paralelos, os quais aplicam a função de mapeamento a cada par, um a um. Para cada frase, a função de mapeamento lê sequencialmente cada palavra e emite, para essa palavra, um par $\langle cor, 1\rangle$, onde *cor* pode ser *verde*, *branco*, *azul* ou *rosa*, ou seja, a função de mapeamento gera um conjunto de pares, ditos intermediários, para cada par de entrada.

Ao final da etapa de mapeamento, cada agente de mapeamento terá gerado um

conjunto de pares intermediários. Para a fase de redução, os pares referentes à mesma chave (cor) devem ser agrupados, formando pares $\langle cor, [1, 1, \ldots, 1] \rangle$. Isso é realizado por uma fase intermediária, e computacionalmente custosa, chamada de *embaralhamento* (*shuffle*). Os pares agrupados são então distribuídos, de acordo com as suas chaves (função de particionamento) entre um conjunto de agentes de redução paralelos, responsáveis por aplicar a função de redução a cada par que receba. A função de redução recebe cada um dos pares agrupados e soma os 1's correspondentes à chave, gerando um par $\langle cor, n \rangle$, onde n é o número de ocorrências da cor *cor* no texto. Ao final, cada agente de redução terá contado o número de ocorrências de uma das cores no texto (resultado final).

Opcionalmente, uma etapa de *combinação* pode ser introduzida. Cada agente de combinação estaria associado a um agente de mapeamento e aplicaria a mesma função de redução aplicada pelos agentes de redução sobre os dados produzidos pelo agente de mapeamento a ele associado. Dessa forma, caso um mesmo agente de mapeamento emita um total de k pares $\langle c, 1 \rangle$, para uma determinada cor c, o agente de mapeamento emitirá o valor $\langle c, k \rangle$. A vantagem de incluir-se uma etapa de combinação é reduzir a quantidade de comunicação na etapa de embaralhamento, uma vez que ao invés de enviar k pares $\langle c, 1 \rangle$, idênticos, para o agente de redução associado à cor c, envia-se um único par $\langle c, k \rangle$, que resume a informação.

Para cada problema de processamento em particular, é preciso definir o mapeamento da estrutura de dados original a ser processada (no exemplo, o texto) para o conjunto de pares chaves/valor de entrada, bem como definir as funções de mapeamento e redução que gerarão os pares intermediários e finais. Além disso, o desenvolvedor pode definir as funções que distribuem os pares entre os agentes de mapeamento e redução, de forma a balancear a carga de computação atribuída aos processos distribuídos. O desempenho do *framework* é fortemente influenciado pela implementação da operação de embaralhamento.

4.2.1 Componentes HPC Shelf para o Map/Reduce

Para este estudo de caso, foi implementada uma aplicação para submissão de trabalhos iterativos de Map/Reduce sobre um conjunto de plataformas de computação paralela. Para isso, são propostos contratos contextuais para os seguintes componentes abstratos, os quais serão compostos, através do SAFe, para formar sistemas computacionais:

- DataSource, da espécie *fonte de dados*, representa o repositório da estrutura

de dados de entrada do processamento;

- DataSink, da espécie *fonte de dados*, representa o repositório onde será armazenada a estrutura de dados resultante do processamento;
- Mapper, da espécie *computação*, o qual implementa um agente de mapeamento;
- Reducer, da espécie *computação*, o qual implementa um agente de redução;
- Splitter, da espécie *conector*, responsável por distribuir as chaves de entrada entre os agentes de mapeamento, bem como receber pares produzidos pelos agentes de redução e redistribuí-los entre os agentes de mapeamento para iniciar uma nova iteração;
- Shuffler, da espécie *conector*, responsável por agrupar as chaves intermediárias produzidas pelos agentes de mapeamento e redistribui-las entre os agentes de redução.

Vale ressaltar que os componentes de computação e facetas de conectores associados a eles através da conectividade entre portas de ambientes, são paralelos, possivelmente executando sobre plataformas virtuais distintas.

O mapeamento da estrutura de dados lida do repositório DataSource para os pares de entrada esperada pelo componente Splitter é realizado pelo *binding* que liga esses dois componentes, bem como o mapeamento dos pares de saída para a estrutura de dados que será armazenado no repositório DataSink.

Para que possamos usar esses componentes em uma aplicação implementada pelo SAFe, devemos definir suas portas de ambiente e de tarefas, as quais serão de utilidade para a aplicação e para o *workflow*, respectivamente. A aplicação fará uso das portas de ambiente para enviar parâmetros de configuração e acompanhar o progresso da execução. Já o *workflow* necessita das portas de tarefa para orquestrar os componentes de solução.

Contratos Contextuais

Os contratos contextuais utilizados para escolha de implementações apropriadas dos componentes Map/Reduce, por parte do mecanismo de resolução implementado pelo sistema Alite, são determinados pelos parâmetros de contextos associados a cada um dos componentes. Para este estudo de caso, nos restringimos a parâmetros de

contexto referentes a propriedades da aplicação em si, não tratando sobre parâmetros relacionados às características das plataformas, bem como parâmetros de contexto relacionados a qualidade e custo.

input_key_type	IMK	DATA	tipo da chave de entrada
input_key_value	IMV	DATA	tipo do valor de entrada
map_function	Mf	MAPFUNCTION	tipo da função de mapeamento
intermediary_key_type	OMK	DATA	tipo da chave intermediária
intermediary_key_value	OMV	DATA	tipo do valor intermediário

Tabela 4.12: Parâmetros de Contexto do Contrato de MAPPER

Os parâmetros de contexto do componente abstrato MAPPER encontram-se apresentados na Tabela 4.12. Através deles, a aplicação pode configurar o tipo dos pares chave/valor de entrada e intermediários, bem como o tipo da função de mapeamento, determinado por um contrato contextual para o componente abstrato MAPFUNCTION, com a seguinte valoração para os seus parâmetros de contexto determinada no contrato de MAPPER:

input_key_type	IMK
input_key_value	IMV
intermediary_key_type	OMK
intermediary_key_value	OMV

intermediary_key_type	OMK	DATA	tipo da chave intermediária
intermediary_key_value	OMV	DATA	tipo do valor intermediário
reduce_function	Rf	REDUCEFUNCTION	tipo da função de redução/combinação
output_key_type	ORK	DATA	tipo da chave de saída
output_key_value	ORV	DATA	tipo do valor de saída

Tabela 4.13: Parâmetros de Contexto do Contrato de REDUCER

Os parâmetros de contexto do componente abstrato REDUCER encontram-se apresentados na Tabela 4.13. Através deles, a aplicação pode configurar o tipo dos pares chave/valor de intermediários de saída, bem como o tipo da função de redução, determinado por um contrato contextual para o componente abstrato REDUCEFUNCTION, com a seguinte valoração para os seus parâmetros de contexto determinada no contrato de REDUCER:

intermediary_key_type	OMK
intermediary_key_value	OMV
output_key_type	ORK
output_key_value	OMV

input_key_type	IMK	DATA	Tipo da chave de entrada
input_key_value	IMV	DATA	Tipo do valor de entrada
bin_function	Bf	PARTITIONFUNCTION	Tipo da função de distribuição das chaves de entrada entre os agentes de mapeamento
output_key_type	ORK	DATA	Tipo da chave de saída
output_key_value	ORV	DATA	Tipo do valor de saída

Tabela 4.14: Parâmetros de Contexto do Contrato de SPLITTER

Os parâmetros de contexto do componente abstrato SPLITTER encontram-se apresentados na Tabela 4.14. Através deles, a aplicação pode configurar o tipo dos pares chave/valor de entrada e de saída, bem como o tipo da função de distribuição das chaves de entrada entre os agentes de mapeamento, determinado por um contrato contextual para o componente abstrato PARTITIONFUNCTION, o qual atribui o argumento IMK (tipo da chave dos pares de entrada) ao seu único parâmetro de contexto, de nome ***input_key***.

intermediary_key_type	OMK	DATA	Tipo da chave de entrada
intermediary_key_value	OMV	DATA	Tipo do valor de entrada
partition_function	Pf	PARTITIONFUNCTION	Tipo da função de distribuição das chaves de entrada entre os agentes de redução

Tabela 4.15: Parâmetros de Contexto do Contrato de SHUFFLER

Os parâmetros de contexto do componente abstrato SHUFFLER encontram-se apresentados na Tabela 4.15. Através deles, a aplicação pode configurar o tipo dos pares chave/valor intermediários, bem como o tipo da função de distribuição das chaves intermediárias entre os agentes de redução, determinado por um contrato contextual para o componente abstrato PARTITIONFUNCTION, o qual atribui o argumento OMK (tipo da chave dos pares intermediários) a ***input_key***.

Finalmente, os componentes fontes de dados DataSource e DataSink não possuem parâmetros de contexto de aplicação.

O sistema de contrato contextuais é capaz de escolher implementações de cada um dos componentes da aplicação Map/Reduce particulares para determinadas escolhas dos tipos dos pares de entrada, intermediários e de saída, bem como das funções de mapeamento e redução/combinação. Entretanto, para o estudo de caso em questão, são utilizados componentes genéricos, que abstraem-se dos tipos dos pares chave/valor e funções de mapeamento e redução.

4.2.2 Arquitetura de um *Workflow* de Processamento Map/Reduce: Contador de Palavras

A Figura 4.14 apresenta um arranjo particular entre os componentes Map/Reduce que representa uma arquitetura básica de sistema computacional para uma computação Map/Reduce, cujo intuito é ilustrar o papel de cada componente participante, bem como as portas de ambiente entre eles para comunicação de dados (pares chave/valor). Esse arranjo pode ser particularizado para executar um processametno de contagem de ocorrências de palavras em arquivos de texto armazenados em um repositório, apenas apontando para os contratos contextuais apropriados dos componentes, preservando o mesmo código SAFeSWL de descrição arquitetural e de orquestração. Entretanto, outros arranjos de processamento Map/Reduce são possíveis, alguns dos quais não são tipicamente suportados por *frameworks* Map/Reduce, tais como:

- uso da etapa de *combinação*, com a inserção de um agente de redução (`combiner`) associado a cada agente de mapeamento, dentro da mesma plataforma virtual, a fim de evitar a comunicação desnecessária de chaves na etapa de embaralhamento;

- *computações map-reduce iterativas*, com a retroalimentação de chaves de saída, produzidas pelos agentes de redução, como chaves de entrada, para o início de uma nova etapa de mapeamento;

- não utilização da etapa de mapeamento, onde a função de mapeamento é tratada como uma função identidade;

- múltiplas etapas de mapeamento em sequência, com diferentes funções de mapeamento;

- múltiplos agentes de mapeamento e/ou de redução, cada um executando em plataformas virtuais distintas.

Esses arranjos alternativos, dentre outros que podem surgir a partir de requisitos de processamentos em particular, podem ser construídos pela configuração apropriada de tipos de chaves (de entrada, intermediários e de saída), bem como a ligação entre as portas dos componentes, dependendo das necessidades da computação pretendida. A descrição que se segue sobre essa arquitetura ainda é geral, válida para qualquer processamento Map/Reduce que se enquadre nessa arquitetura e fluxo de orquestração. Ao final desta seção, será mostrado como pode ser particularizada para a contagem de palavras em um repositório de arquivos texto.

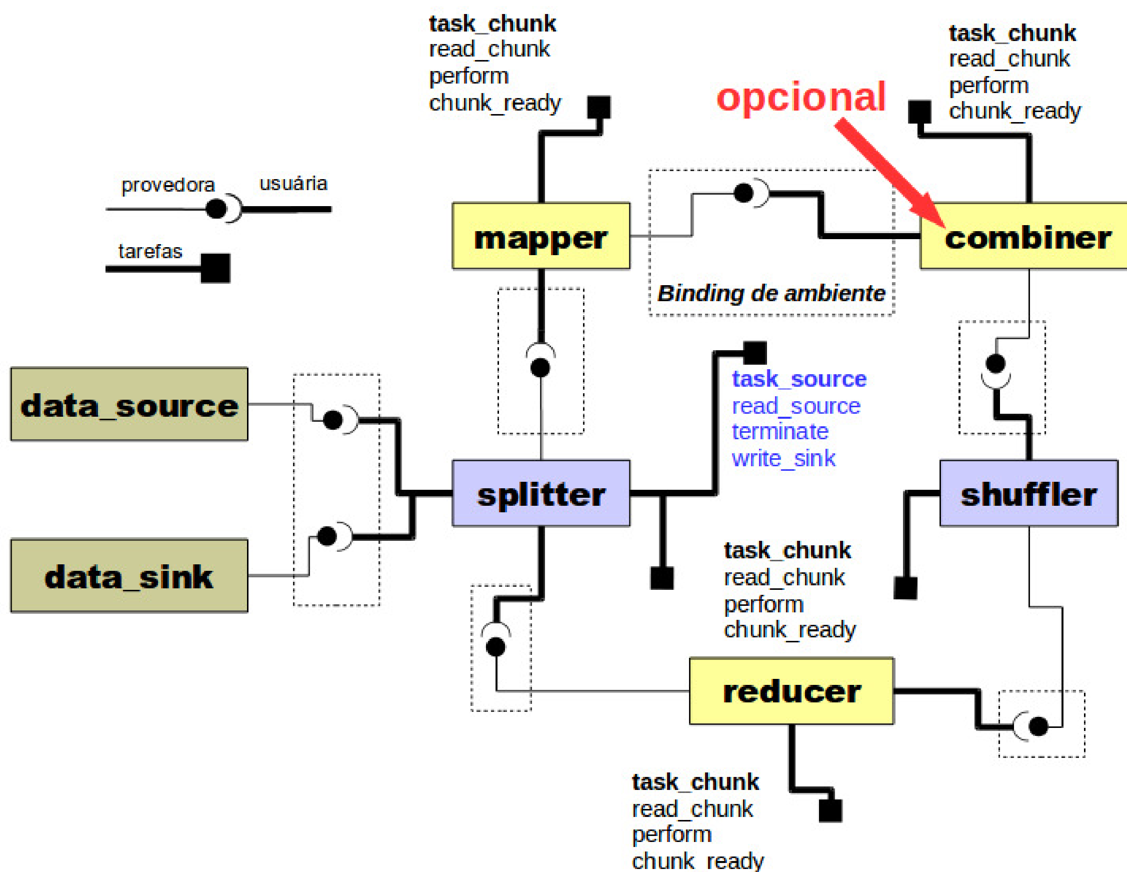


Figura 4.14: Arquitetura Map/Reduce com as portas de tarefas.

A comunicação entre os componentes em um *workflow* de processamento Map/Reduce é realizada por meio de *bindings* de ambiente do tipo *IteratorPort* (retângulos pontilhados), através dos quais um componente provê para um outro uma *stream* de agrupamentos de pares (chave/valor), doravante denominados

chunks. Devido ao uso de componentes conectores (`shuffler` e `splitter`), cujas facetas residem na mesma plataforma virtual onde residem seus componentes parceiros via portas de ambientes, esses *bindings* do tipo *IteratorPort* são diretos. Além disso, sua implementação é de responsabilidade do desenvolvedor de componentes, sendo escolhidos em tempo de execução de acordo com o mecanismo de resolução de contratos contextuais.

Como podemos notar ainda na Figura 4.14, há dois tipos de portas de tarefas: a porta `task_chunk`, exposta pelos componentes `splitter`, `reducer`, `mapper` e `shuffler`; e a porta de tarefas `task_source_sink`, exposta exclusivamente pelo componente `splitter`. A porta `task_chunk` possui as seguintes ações:

- **`read_chunk`**: ler um *chunk* (conjunto de pares), de tamanho determinado pela granularidade configurada pela aplicação, através da porta usuária apropriada, o qual será colocado em uma fila de entrada;
- **`perform`**: efetuar computação sobre o próximo *chunk* da fila de entrada, possivelmente gerando um conjunto de pares que serão também agrupados em *chunks*, colocados em uma fila de saída e providos por meio de uma porta provedora apropriada;
- **`chunk_ready`**: sinalizar que há *chunks* disponíveis na fila de saída, os quais podem ser lidos por meio da porta provedora apropriada.

Por sua vez, a porta de tarefas `task_source_sink`, que implementa ações para acessar os repositórios `data_source` e `data_sink`, possui as seguintes ações:

- **`read_source`**: ler pares de entrada do repositório `data_source`, através da porta usuária apropriada;
- **`terminate`**: indicar que a iteração atual é a iteração final do processamento (condição de terminação do algoritmo);
- **`write_sink`**: escrever os pares de saída, após o término do processamento, no repositório `data_sink`, através da porta usuária apropriada.

Na computação Map/Reduce básica inicialmente apresentada, a ativação das ações das portas de tarefas e invocações dos serviços das portas de ambiente é realizada conforme o seguinte protocolo:

- O componente `splitter` inicia a computação com a ativação da ação `read_source`, que faz com que, em sua faceta que executa no mesmo espaço de endereçamento do componente `data_source`, pares chave/valor de entrada sejam lidos através do *binding* de ambiente entre esses dois componentes, onde `splitter` é usuário, e agrupados em *chunks* associados a cada agente de mapeamento (`mapper`). Ao ter a ação `perform` ativada, o componente `splitter` deve então distribuir os *chunks* entre suas facetas associadas aos agentes de mapeamento e provê-las para eles através de uma porta provedora. A ação `chunk_ready` associada a uma das facetas é ativada quando um *chunk* já está pronto para ser lido por um agente de mapeamento;

- O componente `mapper` (agente de mapeamento) lê um *chunk* de `splitter` quando sua ação `read_chunk` é ativada. Ao ativar a ação `perform`, `mapper` irá executar a função de mapeamento sobre cada par de entrada do *chunk* lido. Os pares intermediários calculados vão sendo acumulados em *chunks* à medida que vão sendo gerados. A ação `chunk_ready` é ativada para sinalizar quando um novo *chunk* de saída for gerado. Nesse caso, um *chunk* já pode ser lido pelo próximo componente, que pode ser tanto o componente `combiner`, do tipo Reducer, implementando a etapa opcional de combinação, ou o `shuffler`;

- O componente `combiner` (agente de combinação) lê um *chunk* de `mapper` quando sua ação `read_chunk` é ativada. A função de redução (combinação) sobre os pares chave/valor lidos é efetuada devido à ativação da ação `perform`. A ação `chunk_ready` sinaliza que um *chunk* calculado já pode ser lido pelo próximo componente (`shuffler`);

- O conector `shuffler` lê um *chunk* provido pelo componente `combiner` (ou `mapper`, no caso de não haver etapa de combinação) ao ter a ação `read_chunk` ativada. Os dados recebidos são pares intermediários. Ao ativar a ação `perform`, o componente `shuffler` irá combinar os pares que tem a mesma chave, produzindo novos *chunks* que serão redistribuídos entre as facetas associadas a cada agente de redução. A ação `chunk_ready` sinaliza que um determinado *chunk* está pronto para ser consumido pelo próximo componente, que no caso é o `reducer`.

- O componente `reducer` (agente de redução) lê um *chunk* de `shuffler` quando sua ação `read_chunk` é ativada. A função de redução sobre os pares

chave/valor lidos é efetuada devido à ativação da ação `perform`. A ativação da ação `chunk_ready` sinaliza que um *chunk* calculado já pode ser lido pelo próximo componente (`splitter`), o qual decidirá se o processamento Map/Reduce deve terminar, ativando a ação `terminate`, ou iniciar uma nova iteração, redistribuindo entre os agentes de mapeamento os pares recebidos do `reducer`, provenientes da iteração anterior;

- Após a ativação da ação `terminate`, sinalizando que o `splitter` determinou a terminação do processamento Map/Reduce, a ação `write_sink` é ativada, fazendo com que os pares de saída, provenientes dos agentes de redução na última iteração, sejam submetidos à porta usuária ligada ao repositório `data_sink`.

Embora a arquitetura básica sugira a existência de um único agente de mapeamento e um único agente de redução, os componentes `splitter` e `shuffler` são preparados para estar conectados a múltiplos agentes, tanto de mapeamento quanto de redução. Isso é possível graças ao conceito de facetas múltiplas, as quais podem ser replicadas e conectadas, através de suas portas de ambiente e de tarefa, a diferentes componentes de computação.

Contagem de Palavras em um Repositório de Arquivos de Texto

Usando a arquitetura e fluxo de orquestração acima descrito, o processamento Map/Reduce pode ser particularizado para realizar um processamento em particular apontando-se para os arquivos de contratos contextuais (`*.cc`) apropriados, os quais associam os argumentos de contexto exigidos para os parâmetros de contexto de cada componente da arquitetura. Neste estudo de caso, será utilizado como demonstração o clássico exemplo da contagem de ocorrência de palavras em arquivos de texto armazenados em um repositório. A Tabela 4.16 apresenta as valorações dos parâmetros de contexto dos componentes de computação e conectores da arquitetura, contida nos arquivos dos contratos contextuais. A tabela apresenta o nome do parâmetro de contexto, a variável a ela associada, quais os componentes que possuem esses parâmetros e qual o valor do seu argumento.

O componente fonte de dados que representa o repositório de entrada implementa o acesso a uma pasta de um sistema de arquivos, contendo arquivos de texto. Através do *binding* que o liga ao componente `splitter`, são fornecidas os pares contendo um valor inteiro e um conteúdo de texto, formado pela concatenação de

nome do parâmetro	variável	componentes	valor
input_key_type	IMK	**mapper** **splitter**	INTEGER
input_value_type	IMV	**mapper** **splitter**	STRING
map_function	Mf	**mapper**	WORDCOUNTER
combine_function	Cf	**combiner**	REDUCESUM
intermediary_key_type	OMK	**mapper** **splitter** **combiner** **shuffler** **reducer**	STRING
intermediary_value_type	OMV	**mapper** **splitter** **combiner** **shuffler** **reducer**	INTEGER
reduce_function	Rf	**combiner** **reducer**	REDUCESUM
output_key_type	ORK	**reducer** **splitter**	STRING
output_value_type	ORV	**reducer** **splitter**	INTEGER

Tabela 4.16: Argumentos de Contexto para os Componentes do Processamento Map/Reduce de Contagem de Palavras.

uma determinada quantidade de linhas de texto lidas dos arquivos que cabem em um *buffer* (a origem de cada linha é indiferente para esse processamento em particular).

Para a função de distribuição dos pares aos agentes de mapeamento (`mapper`), determinado pelo argumento associado ao parâmetro de contexto ***bin_function***, é usado um componente *default* que trabalha com chaves inteiras, o qual usa a função módulo para distribuir as chaves entre os agentes de mapeamento. Deve-se notar, na Tabela 4.16, que o parâmetro ***bin_function***, do componente `splitter`, não recebe um argumento, levando à escolha da implementação *default* de PartitionFunction para a associação do tipo de componente Integer, como argumento, ao parâmetro ***input_key***.

Para cada par de entrada, cada agente de mapeamento emite pares intermediários $\langle c, 1\rangle$, para cada ocorrência de uma palavra c encontrada no texto, usando a função de mapeamento determinada por um componente do tipo WordCounter. Então, o agente de combinação (`combiner`) associado usa a função de redução, um componente do tipo ReduceSum, para somar os valores associados a uma mesma chave, emitindo pares intermediários da forma $\langle c, k\rangle$, a partir de k pares intermediários $\langle c, 1\rangle$ emitidos pelo agente de mapeamento. São esses pares intermediários emitdos pelos agentes de combinação que são distribuídos entre os agentes de redução (`reducer`) pelo componente `shuffler` (etapa de embaralhamento), de modo que os valores associados à mesma chave são somados pela função de redução por um mesmo agente de redução. A distribuição das chaves intermediárias entre os agentes de redução também é realizada usando a função *default* de distribuição de chaves inteiras usada na distribuição das chaves de entrada entre os agentes de mapeamento, uma vez que também não é fornecido um argumento para o parâmetro ***partition_function***, do componente `shuffler`, na Tabela 4.16. Finalmente, os pares de saída $\langle c, n\rangle$, onde n é o número total de ocorrências da palavra c no repositório, são emitidos para o componente `splitter`, que encerra a computação ao final da primeira e única iteração e repassa dos dados ao repositório de saída.

4.2.3 A Interface da Aplicação Map/Reduce

Para executar o Map/Reduce, foi implementada uma interface simples onde o especialista define quais são os componentes que fazem parte da arquitetura e como os mesmos estão conectados através de suas portas de ambiente do tipo *IteratorPort*. A escolha dos componentes inclui a definição dos argumentos de contexto nos

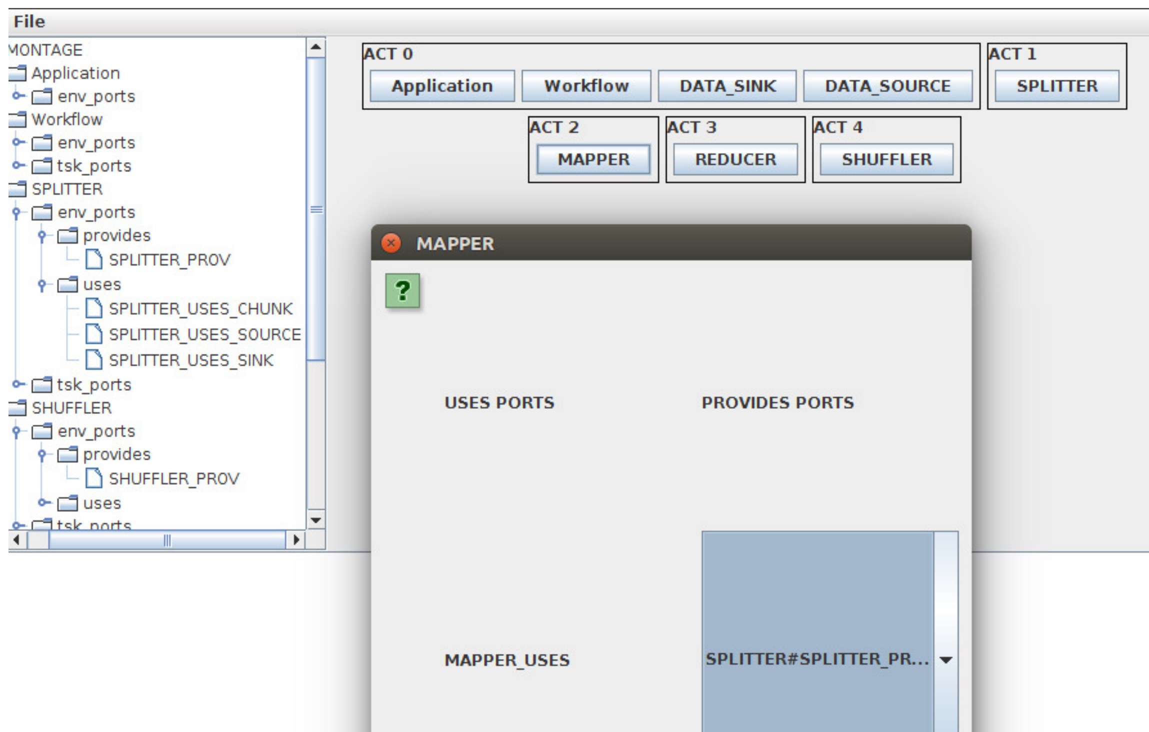


Figura 4.15: Interface para criação de workflows Map/Reduce.

contratos contextuais das instâncias de MAPPER, REDUCER, SPLITTER e SHUFFLER empregados nas diversas etapas do algoritmo, determinando, desse modo, os tipos dos pares chave/valor, bem como das funções de mapeamento, redução e distribuição de chaves entre agentes de mapeamento e de redução, respectivamente.

A Figura 4.15 apresenta a interface gráfica implementada usando *Java Swing* para a construção e execução de *workflows* usando os componentes apresentados do Map/Reduce. Do lado esquerdo, os componentes abstratos MAPPER, REDUCER, SPLITTER, SHUFFLER, DATASINK e DATASOURCE são mostrados, além de suas portas de ambiente e tarefas. Ao clicar em um componente, o especialista escolhe se deseja ter uma instância do mesmo no *workflow*. Em caso afirmativo, o componente reaparece no centro da interface como um botão que a ser clicado abre uma janela *pop-up* onde o especialista conecta as portas de ambiente (relacionando uma porta usuária com uma porta provedora). Para conectar as portas de tarefas, o especialista deve clicar no componente *workflow* onde a mesma janela *pop-up* irá aparecer.

Portanto, a partir da interface visual da aplicação, o especialista pode escolher quais componentes constituirão a arquitetura do *workflow* Map/Reduce e como estarão ligados por meio de suas portas de ambiente, levando a geração, por parte da aplicação, do código no subconjunto arquitetural da linguagem SAFeSWL. Abaixo,

iremos descrever trechos de um código arquitetural para a arquitetura básica de processamento Map/Reduce discutido anteriormente na Figura 4.14.

```
...
 <!-- workflow -->
 <workflow id="1" name="workflow-mr">
  <provides_port id="21" name="port_SAFeSWL" id_component="1"/>
  <provides_port id="22" name="port_Go" id_component="1"/>
  <task_port id="23" name="wf-splitter-ss" id_component="1"/>
  <task_port id="24" name="wf-splitter-ck" id_component="1"/>
  <task_port id="25" name="wf-mapper-ck" id_component="1"/>
  <task_port id="27" name="wf-reducer-ck" id_component="1"/>
  <task_port id="28" name="wf-shuffler-ck" id_component="1"/>
 </workflow>
 ...
```

Listagem 4.10: Componente workflow.

A Listagem 4.10 mostra o trecho do código arquitetural referente ao componente *workflow*, destacando a definição de suas portas de tarefas, as quais serão conectadas às demais portas de tarefas dos outros componentes. O componente *workflow* se conecta às duas portas de tarefas do componente `splitter`, do tipo SPLITTER, através das portas `wf-splitter-ss` (conecta-se a porta `task_source_sink`) e `wf-splitter-ck` (conecta-se a porta `task_chunk`). De forma análoga, o mesmo vale para os outros componentes com portas de tarefas do tipo `task_chunk`. Desta forma, é possível que o *workflow* orquestre suas ações.

```
...
 <repository name="data_sink" id="200">
  <provides_port id="2001" name="data_sk_prov" id_component="200"/>
 </repository>

 <repository name="data_source" id="201">
  <provides_port id="2010" name="data_src_prov" id_component="201"/>
 </repository>

 <computation name="splitter" id="2">
  <uses_port id="31" name="splitter_source_uses" id_component="2"/>
  <uses_port id="32" name="splitter_sink_uses" id_component="2"/>
  <uses_port id="36" name="splitter_chunk_uses" id_component="2"/>
  <provides_port id="33" name="splitter_chunk_prov" id_component="2"/>
  <contract>
     <tns:uri>/home/safe-user/contracts/splitter_contract.cc</tns:uri>
```

```
  </contract>
  <task_port name="splitter_ss" id="34" id_component="2">
    <action id="341" name="read_source"/>
    <action id="342" name="terminate"/>
    <action id="343" name="write_sink"/>
  </task_port>
  <task_port name="splitter_chunk" id="35" id_component="2">
    <action id="351" name="read_chunk"/>
    <action id="352" name="perform"/>
    <action id="353" name="chunk_ready"/>
  </task_port>
</computation>
...
```

Listagem 4.11: Repositórios (data_source e data_sink) e componente splitter

A Listagem 4.11 mostra o código gerado para os componentes repositórios data_sink e data_source, respectivamente dos tipos DataSink e DataSource. Esses componentes expõem apenas uma única porta provedora, as quais terão como usuário o componente splitter, detalhado logo em seguida. O componente splitter apresenta três portas de ambiente usuárias, duas para se conectar aos repositórios e uma para se conectar a algum componente do tipo computação para receber a lista de *chunks*. Além disso, trata-se do único componente que apresenta duas portas de tarefas de tipos diferentes. Pela Listagem 4.11, podemos notar as ações de cada uma dessas portas. A *tag* <contract> indica a localização do arquivo que representa o contrato contextual para o componente splitter. Para os outros componentes da arquitetura (reducer, mapper, combiner e shuffler), o código de descrição arquitetural é análogo.

Com relação aos contratos contextuais de cada um dos componentes (splitter, mapper, combiner, shuffler e reducer), descrevendo os seus respectivos tipos que guiarão o mecanismo de resolução de componentes de Alite, destaca-se que são armazenados em arquivos separados do código SAFeSWL (extensão ".cc"). É importante ressaltar que é através da especificação dos argumentos de contexto nesses contratos contextuais que o processamento Map/Reduce é particularizado para uma determinada tarefa (e.g. contagem de palavras), especificando-se, para as diversas etapas, possivelmente iterativas, do processamento, os tipos dos pares chave/valor, bem como os tipos das funções de mapeamento, redução e distribuição das chaves entre os agentes de mapeamento e redução.

```
...
```

```
<!-- splitter bindings -->
<env_binding>
    <uses_port id="31" name="splitter_source_uses" id_component="2"/>
    <provides_port id="2010" name="data_src_pv" id_component="201"/>
</env_binding>
<env_binding>
    <uses_port id="32" name="splitter_sink_uses" id_component="2"/>
    <provides_port id="2001" name="data_sk_prov" id_component="200"/>
</env_binding>
<env_binding>
    <uses_port id="36" name="splitter_chunk_uses" id_component="2"/>
    <provides_port id="41" name="red_02_prov_ck" id_component="4"/>
</env_binding>
...
```

Listagem 4.12: *Bindings* do Componente *splitter*

A Listagem 4.12 mostra o trecho do arquivo arquitetural relativo às conexões das portas de ambiente do componente `splitter`. Nesse caso, o componente `splitter` é usuário dos dois repositórios e também do componente `reducer`. A porta usuária `splitter_source_uses` conecta-se à porta provedora de `data_source`, para ler os dados fontes de entrada. Por sua vez, a porta `splitter_sink_uses` conecta-se à porta provedora do componente `data_sink`, para salvar os dados resultantes do processamento. Finalmente, a porta usuária `splitter_chunk_uses` é usada para leitura de uma lista de *chunks* providos por algum outro componente do *workflow*, notadamente o componente `reducer`, a fim de iniciar uma nova iteração do processamento Map/Reduce.

```
...
 <task_binding>
    <task_port id="23" name="wf-splitter-ss" id_component="1"/>
    <right_peer name="splitter_ss" id="34" id_component="3"/>
 </task_binding>
 <task_binding>
    <task_port id="24" name="wf-splitter-ck" id_component="1"/>
    <right_peer name="splitter_chunk" id="35" id_component="3"/>
 </task_binding>
 <task_binding>
    <task_port id="25" name="wf-mapper-ck" id_component="1"/>
    <right_peer name="wf_mapper_ck" id="45" id_component="4"/>
 </task_binding>
 <task_binding>
```

```
        <task_port id="27" name="wf-reducer-ck" id_component="1"/>
        <right_peer name="wf_reducer-ck" id="55" id_component="5"/>
  </task_binding>
  <task_binding>
        <task_port id="28" name="wf-shuffler-ck" id_component="1"/>
        <right_peer name="wf_shuffler_ck" id="6" id_component="6"/>
  </task_binding>
  ...
```

Listagem 4.13: *Binding* de tarefas.

A Listagem 4.13 apresenta o trecho da linguagem arquitetural que relaciona as portas de tarefa dos componentes de solução do *workflow* Map/Reduce de exemplo com o componente *workflow*. Esse relacionamento habilita que o componente *workflow* para orquestrar as ações das portas dos tipos `task_chunk` e `task_source_sink`.

4.2.4 A Execução da Aplicação Map/Reduce

Uma vez especificado o código arquitetural, a aplicação deve gerar o código de orquestração do processamento Map/Reduce configurado pelo especialista em mais alto nível. A Figura 4.14 apresenta um exemplo de código de orquestração referente ao processamento Map/Reduce básico descrito na Seção 4.2.2. Para facilitar o entendimento, os identificadores inteiros foram substituídos por nomes o que, obviamente, não reflete o código real (por exemplo, a invocação de `splitter.read_source` significa que está sendo ativada a ação `read_source` da porta `splitter_ss`, do componente `splitter`)

```
...
<invoke_oper action="compute" id="splitter.read_source" />
<invoke_oper action="compute" id="splitter.perform" />
<iterate_oper max="-1">
  <choice_oper>
    <select action_id="splitter.chunk_ready">
      <invoke_oper action="compute" id="mapper.read_chunk" />
      <invoke_oper action="compute" id="mapper.perform" />
    </select>
    <select action_id="mapper.chunk_ready">
      <invoke_oper action="compute" id="combiner.read_chunk" />
      <invoke_oper action="compute" id="combiner.perform" />
    </select>
    <select action_id="combiner.chunk_ready">
      <invoke_oper action="compute" id="shuffler.read_chunk" />
```

```
    <invoke_oper action="compute" id="shuffler.perform" />
   </select>
   <select action_id="shuffler.chunk_ready">
    <invoke_oper action="compute" id="reducer.read_chunk" />
    <invoke_oper action="compute" id="reducer.perform" />
   </select>
   <select action_id="reducer.chunk_ready">
    <invoke_oper action="compute" id="splitter.read_chunk" />
    <invoke_oper action="compute" id="splitter.perform" />
   </select>
   <select action_id="splitter.terminate">
    <invoke_oper action="compute" id="splitter.write_sink" />
    <break/>
   </select>
  </choice_oper>
</iterate>
 ...
```

Listagem 4.14: Arquivo de orquestração.

A orquestração inicia com a ativação da ação `read_source` do componente `splitter`, o qual lê os dados de entrada do componente repositório `data_source`. Após ler os dados, é ativada a sua ação `perform`, a qual gera os *chunks* e os distribui entre os agentes de mapeamento (no caso, existe um único agente de mapeamento, o componente `mapper`, bem como um único de redução).

Após essa preparação, é iniciado um laço (`iterate`), o qual irá escolher (`choice_oper`) dentre os componentes do Map/Reduce qual *chunk* está pronto, via chamada da ação `chunk_ready`. Por exemplo, caso `shuffler.chunk_ready` esteja ativado, o *workflow* irá ativar as ações `read_chunk` e `perform` do componente `reducer`. Essa lógica de orquestração se repete para todos os outros componentes até que a ação `splitter.terminate` encontre-se ativada.

Para terminar a iteração, a ação `splitter.terminate` é escolhida, por ter sido ativada. Caso verdadeiro, a iteração termina, com execução da ação `break`, e os dados são salvos no componente repositório `data_sink`, finalizando o processamento Map/Reduce em questão.

4.3 Considerações Finais

Esta seção tem como objetivo avaliar o SAFe e sua linguagem SAFeSWL no que diz respeito a construção de aplicações sobre a ferramenta Montage e Map/Reduce, no seguintes pontos:

i. **Reusabilidade**: o SAFe e sua linguagem SAFeSWL apresentam características reusáveis na composição de aplicações e *workflows*;

ii. **Modularidade**: aplicações em SAFe são modulares;

iii. **Abstração**: o nível de abstração ao implementar aplicações em SAFe é relativamente alto, deixando ao especialista apenas a resolução do problema;

iv. **Exequibilidade**: *workflows* em SAFeSWL são plenamente executáveis.

No que diz respeito a **reusabilidade**, os *workflows* para o Montage e Map/Reduce são constituídos de componentes que expõem portas de ambiente usuárias, portas de ambiente provedoras e portas de tarefas, uma abordagem inspirada no CCA. A forma como a aplicação se comunica com essas portas é via *bindings* de ambiente e de tarefas, respectivamente. Atualmente esses *bindings* são implementados como Serviços *Web*, logo, os mesmos componentes podem ser reusados em diversos *workflows* diferentes, como no M101 e no Plêiades, por exemplo (programação com reuso). Pelas portas de ambiente é possível enviar e receber parâmetros e pelas portas de tarefas disparar ações de computação, conectando-as com outros componentes do *workflow* (programação para reuso).

A **modularidade** dos *workflows* SAFe toma vantagem da programação orientada a componentes (com reuso e para reuso) aliada em sua inspiração com o modelo CCA. Os mesmos componentes podem ser combinados de diversas formas, conectados através da linguagem de descrição arquitetural e orquestrados pela linguagem de orquestração. Para adicionar ou eliminar componentes do *workflow*, basta modificar o arquivo SAFeSWL, o qual exige conhecimento de XML e das características do *framework* SAFe.

Para o provedor de aplicações, a **abstração** é no nível dos componentes. Não importa como os componentes do Montage ou Map/Reduce foram implementados e sim como eles estão disponibilizados para a construção de *workflows*. No caso das aplicações aqui mostradas, os componentes expõem portas que são conectadas e orquestradas pela linguagem SAFeSWL. Já para o especialista, a abstração depende da forma como o provedor de aplicações implementou a aplicação. A interface gráfica, permite que o especialista escolha os componentes do Montage ou Map/Reduce e os conecte via uma interface implementada em *Java Swing*. Poderíamos ter implementado algo em linha de comando, uma aplicação *web* ou até mesmo um *plug-in* para o Eclipse.

A **exequibilidade** do SAFe é comprovada pelas execução da aplicação MoEx sobre os *workflows* M101 e Plêiades, gerando dos resultados esperados (as imagens da galáxia e constelação, respectivamente), assim como se tivessem sido executados por *scripts* em *shell* (solução original, dada pelo próprio *site*). Da mesma forma, o *workflow* Map/Reduce.

Capítulo 5

Trabalhos Relacionados

Este capítulo tem por objetivo caracterizar as contribuições do SAFe frente a outras alternativas de soluções também dedicadas a oferecer a usuários finais, especialistas em geral em certo domínio do conhecimento, tais como cientistas e engenheiros, interfaces para construção de soluções computacionais para problemas de seu interesse. Após um amplo estudo, incluindo um mapeamento sistemático apresentado no Exame de Qualificação do autor desta Tese de Doutorado, essas alternativas foram identificadas dentro do contexto de sistemas gerenciadores de *workflows* científicos, dentre os quais foram escolhidos um conjunto representativo de sistemas dessa natureza. A seguir, cada uma das alternativas escolhidas é apresentada. Para cada uma delas, são discutidas as principais semelhanças e diferenças em relação a plataforma HPC Shelf e ao SAFe propriamente dito. Ao final, é apresentada uma síntese, que busca caracterizar as principais contribuições e proeminências do SAFe dentro do contexto das alternativas discutidas.

5.1 ASKALON

ASKALON [Qin e Fahringer 2012] oferece um interface gráfica, baseada em UML, para a construção de *workflows* de aplicações científicas que devem executar sobre infraestruturas de *grade computacional*. Sendo assim, o acesso a infraestrutura é feita de forma transparente pelo desenvolvedor de aplicações. De outra maneira, o usuário também pode especificar um *workflow* usando uma linguagem própria do ambiente, baseada em XML, conhecida como AGWL (*Abstract Grid Workflow Language*). O código AGWL é então submetido à *middleware* de serviços, interna ao sistema de execução, para então ser executado de forma confiável. Os principais componentes da arquitetura de ASKALON são:

- Um *Gerenciador de Recursos* fica responsável pela negociação, reserva, alocação de recursos e implantação automática dos serviços necessários à execução das aplicações desenvolvidas em ASKALON. Sendo assim, a infraestrutura de grade é acessada de forma transparente pela AGWL;

- O *Agendador* é um componente da arquitetura de ASKALON responsável em determinar de forma efetiva o mapeamento de um ou mais *workflows* nos recursos computacionais, fazendo uso de heurísticas otimizadas baseadas em grafos. Além disso, o Agendador gerencia de forma dinâmica a execução da aplicação para que a mesma se adapte à natureza mutável das grades computacionais;

- Um *Motor de Execução* tenta garantir uma execução confiável e tolerante a falhas, através de técnicas de *checkpoint*, migração de computações, reinicio de execução, re-tentativas e replicação de computações de dados;

- Um *Analisador de Performance* verifica automaticamente a execução do *workflow* em busca de problemas como gargalos;

- Um *Preditor de Performance* estima o tempo de execução das atividades de um *workflow* através de métodos estatísticos.

A linguagem AGWL permite a construção de *workflows* a partir de unidades atômicas chamadas *activity types*, compostos através de combinadores de fluxo de controle (`sequence`, `if`, `for`, `forEach`, `do-while`, `parallel`, etc.) ou de fluxo de dados (portas de dados). Cada *activity type* está associado a um ou mais *activity deployments*, os quais especificam como uma *activity type* encontra-se implementado e como implantada nos recursos de grades computacionais. O GLARE (*Grid-Level Activity Registration*) [Siddiqui et al. 2005] é o componente do ASKALON responsável pelo gerenciamento de *activities*, enquanto o GridARM [Siddiqui e Fahringer 2005] é responsável pelo gerenciamento dos recursos de grade. O usuário pode informar requisitos funcionais e não funcionais através de propriedades e restrições sobre as atividades e o fluxo de dados.

A plataforma HPC Shelf possui as seguintes analogias com ASKALON:

- Diferente de ASKALON, a HPC Shelf não depende de tecnologias de grades computacionais para usufruir de recursos de computação paralela oferecidos por provedores de infraestrutura. De fato, embora tratada como uma nuvem

computacional (de componentes), nem mesmo depende de tecnologias típicas de nuvens, tais como virtualização de *hardware*, a despeito do uso do termo *plataforma virtual* para referir-se a instâncias de componentes plataforma. Um mantenedor de um computador paralelo pode oferecer sua plataforma física, ou partições dela, diretamente como perfis de plataforma na nuvem HPC Shelf, sem usar virtualização, bastando para isso implementar os serviços que devem ser consumidos de um *Back-End* compatível com a HPC Shelf.

- a distinção entre *activity types* e *activity deployments* em ASKALON é análoga à separação entre contratos contextuais e componentes na HPC Shelf, respectivamente. Tanto em *activity types* quanto em *contratos contextuais*, é possível definir restrições sobre os recursos que serão alocados para implantação do *activity deployments* ou componente. Sendo assim, pode-se afirmar que o GLARE e o GridARM, em conjunto, cumprem, no ASKALON, um papel análogo ao *Core*, sendo que o GridARM cuida das plataformas disponíveis para as aplicações. Entretanto, o *Core* trata tanto plataformas de execução quanto os componentes de *software* sob uma abstração comum, que são os componentes Hash (de diferentes espécies), provendo um nível mais alto de abstração para o provedor de aplicações. Além disso, o sistema de contratos contextuais tem sido proposto como um mecanismo mais geral de alocação de recursos, incluindo um conjunto mais rico de restrições, não apenas em relação à plataforma de execução, mas também em relação à propriedades da aplicação no contexto da qual o componente deverá executar, as quais influenciam diretamente como deverão ser implementadas. Um importante aspecto do sistema de contratos contextuais é estar assentado sobre um sistema de tipos, garantido segurança nas ligações entre aplicações e componentes.

- HPC Shelf preconiza uma separação de interesses mais clara entre os usuários especialistas, interessados em usufruir das funcionalidades de aplicações (executar *workflows*), e os envolvidos no desenvolvimento de aplicações, incluindo provedores de aplicação, desenvolvedores de componentes e mantenedores de plataformas, como uma das premissas no projeto da plataforma. Em ASKALON, percebe-se uma distinção clara dos provedores dos recursos (da grade), os quais devem configurar seus ambientes em caso de interesse em ter sua infraestrutura acessível para as aplicações do ASKALON. Entretanto, ASKALON não promove uma distinção muito

nítida entre especialistas (usuários dos *workflows*), provedores de aplicações (desenvolvedores dos *workflows*) e desenvolvedores de componentes, não sendo essa distinção estimulada pela própria arquitetura da plataforma. Observa-se que o usuário especialista tende a ser aquele próprio que desenvolverá os *workflows* ASKALON para seu usufruto, acessando diretamente os componentes (*activities*) disponíveis na plataforma e possivelmente desenvolvendo-os e disponibilizando-os para a sua própria aplicação. Por sua vez, especialistas da HPC Shelf tem acesso à uma aplicação oferecida por uma outra parte, o provedor de aplicações, sem conhecimento a respeito da utilização da infraestrutura da HPC Shelf. Porém, é ainda possível oferecer a especialistas da HPC Shelf uma aplicação através da qual possam descrever soluções computacionais diretamente sobre os componentes do *Core*.

Em relação ao SAFe propriamente dito, ASKALON não provê um *framework* para o desenvolvimento de aplicações de alto nível. Como dito anteriormente, o usuário especialista deve fazer uso de uma interface gráfica para construir *workflows* diretamente sobre os componentes disponibilizados na plataforma ou, alternativamente, escrever código em AGWL. Em ambos os casos, as competências do especialista se confundem com algumas das competências do provedor de aplicações da HPC Shelf, em especial com o conhecimento sobre manipulação de arquivos XML, soluções computacionais dentro do seu domínio de interesse e grades computacionais. Comparativamente, a interface do ASKALON para o seu usuário final compara-se a uma aplicação específica na HPC Shelf que expusesse aos seus especialistas eventualmente interessados a possibilidade de escrever *workflows* diretamente em SAFeSWL, acessando os componentes do catálogo do *Core*. Por outro lado, um *framework* análogo ao SAFe poderia ser de grande valia para ASKALON, permitindo desenvolver aplicações de alto nível, independentes de plataforma, para usufruir de forma transparente dos recursos oferecidos.

5.2 BPEL Sedna

Sedna [Wassermann et al. 2007] é um ambiente de modelagem visual para uso do BPEL [Juric 2006] para aplicações científicas, abstraindo assim detalhes relativos a tecnologia de Serviços *Web* do seu usuário final, o desenvolvedor de aplicações. Assim como em ASKALON, o desenvolvedor de aplicações em Sedna acumula os papéis do especialista e do provedor de aplicações da HPC Shelf, ou seja, embora seja vislumbrado como usuário final um cientista especialista, meramente interessado em

resolver problemas em seu domínio de conhecimento, é necessário o conhecimento sobre soluções computacionais e composição de componentes cujos serviços estão expostos em Serviços *Web* para construção de *workflows*. A linguagem visual tem o objetivo de esconder nuances da *middleware* interna e da linguagem de orquestração, que é uma extensão de BPEL. Sendo assim, cientistas poderiam se beneficiar das vantagens dos *workflows* científicos sem se ater aos detalhes técnicos.

O trabalho em torno de Sedna tem suas motivações iniciais em estudar as limitações da linguagem BPEL, bastante disseminada em aplicações das áreas de negócios, para aplicações científicas. Para isso, várias extensões são propostas sob uma camada de abstração chamada de Scientific PEL, o qual introduz construtores de propósito geral para a implementação de aplicações científicas, como por exemplo, construtores para execução de atividades concorrentes. Além disso, sobre Scientific PEL foi implementada uma outra camada de abstração, chamada Domain PEL, a qual especifica o domínio de uma aplicação, permitindo o encapsulamento de atividades complexas requiridas em um certo nicho científico. As principais alterações são:

- *Indexadores de Fluxos*: abstrações sobre fluxos BPEL com o objetivo de trabalhar com um grande número de atividades de forma concorrente. Através dos indexadores de Sedna, o desenvolvedor especifica a quantidade de fluxos BPEL sobre uma mesma atividade que serão executados em paralelo.

- *Composição Hierárquica*: Sedna utiliza-se da natureza WSDL dos *workflows* BPEL para abstrair a construção de *workflows* maiores constituídos de *subworkflows*.

- *Plug-ins*: para evitar reusar um *workflow* inteiro desnecessariamente (composição hierárquica), Sedna oferece aos desenvolvedores a abstração de *plug-ins* os quais acessam, de forma comprimida, apenas subconjuntos de atividades de um *workflow* maior.

- *Macros*: também focado em reuso, permitindo que um conjunto de atividades cuja utilidade é frequente possa ser, de forma otimizada, transformada em uma macro para ser aproveitada em outros *workflows*, inclusive por terceiros.

Sedna não incorpora, ou está integrada, a mecanismos de abstração, catalogação e descoberta de componentes comparáveis com *activities* de ASKALON e os contratos contextuais da HPC Shelf. Seu projeto pode ser comparado especificamente ao

projeto de SAFeSWL, sendo restrito ao desenvolvimento de uma linguagem que estenda BPEL para composição de serviços de interesse de aplicações científicas, os quais fazem o papel dos componentes Hash orquestrados pelo SAFe. Porém, SAFe controla todo o ciclo de vida desses componentes, incluindo a implantação e instanciação em uma plataforma virtual de computação paralela, enquanto Sedna assume a existência dos serviços, já implantados. Finalmente, assim como ASKALON, o ambiente visual de Sedna para facilitar a construção de *workflows* pode ser visto como uma aplicação específica da HPC Shelf, que expõe os componentes diretamente para que os especialistas, também no papel de provedores, desenvolvam aplicações e as disponibilizem. De fato, um *framework* de aplicações, como o proposto pelo SAFe, poderia ser incorporado para proporcionar o desenvolvimento de aplicações de mais alto nível, voltados a especialistas que não possuam as competências exigidas para um provedor montar aplicações.

5.3 Kepler

Kepler [Altintas et al. 2004] tem como objetivo prover aos cientistas um sistema fácil de usar, porém robusto, para o projeto, execução, monitoramento e re-execução de *workflows* científicos. Kepler combina o projeto em alto nível de um *workflow* com execução e interação sob demanda, acesso a dados locais e remotos além da invocação de serviços distribuídos.

Processos independentes são chamados de atores[1] pelo sistema, e podem representar fontes de dados, transformadores de dados, agregadores de dados ou computações. Atores possuem um conjunto de portas de entrada e saída por onde o fluxo de dados é transportado. Além disso, atores tem parâmetros os quais podem definir um comportamento específico. A checagem de tipos é feita em tempo de projeto (estática) mas também em tempo de execução (dinâmica). Algumas das características do ambiente são:

- Interface gráfica com o usuário para projeto e execução de *workflows* científicos;
- Prototipagem de *workflows* antes da implementação do mesmo;
- Execução distribuída através de Serviços *Web* e serviços de grade (Kepler disponibiliza atores específicos para execução em grades computacionais e invocação de serviços escritos em WSDL);

[1] *actors*.

- Acesso a banco de dados através de atores especializados;
- Capaz de implementar *workflows* aninhados.

Kepler é comparável ao Sedna, com a diferença de que Sedna parte da motivação em adaptar BPEL, uma linguagem para expressar *workflows* de negócios, a fim de atender requisitos de *workflows* científicos, ou seja, visa a orquestração de serviços de computação científica, tanto serviços *web* quanto serviços de grade. Por sua vez, Kepler propõe uma linguagem de modelagem visual para especificação de *workflows*, herdada do sistema Ptolemy II. Essa linguagem permite que os atores, como são chamados os componentes do *workflow* descrevam não apenas serviços computacionais, como descrito acima.

Comparado a HPC Shelf, e mais especificamente ao SAFe, são válidos os mesmos comentários feitos em relação ao ambiente Sedna. Ou seja, os usuários finais de Kepler também precisam de competências tanto de especialistas quanto de provedores de aplicações do SAFe. Além disso, não há sistema de descoberta de implementações de componentes com base em especificações de contratos. O ambiente visual do Kepler, herdado do sistema Ptolemy II, pode ser visto como uma aplicação sobre o SAFe, acessando componentes que encapsulam atores de Kepler.

5.4 OSC

OSC [Medeiros e Gomes 2013] é uma linguagem de especificação de *worflows* científicos especificada como um estilo arquitetural em Acme [Garlan, Monroe e Wile 2010], uma meta-linguagem de descrição arquitetural. OSC emprega conectores como entidades de "primeira classe" além de regras definidas em Acme para governar as interações entre componentes de um sistema. Ao usar essa abordagem, OSC aumenta o seu potencial de reuso, composicionalidade e configuração de um *workflow* científico. A ideia é focar no tratamento dos atributos não funcionais de um sistema.

O modelo de *workflow* descrito em OSC define elementos como *tarefas*, *conectores*, *portas* e *papéis*. Portas de entrada e saída devem ser obrigatoriamente conectadas a papéis de origem e destino dos conectores. Essas ligações representam dependências de dados e controle do *workflow*. Sobre Acme, OSC define um conjunto de tipos de conectores, tarefas, papéis e portas.

Além dos tipos específicos, são também elaboradas um conjunto de regras que organizam a forma como os conectores podem ser ligados às tarefas. Dentre os tipos estendidos, destacam-se àqueles referentes ao tratamento de requisitos não funcionais, tais como tolerância a falhas e proveniência de dados.

Sendo assim, *workflows* em OSC são instâncias da extensão definida sobre Acme, usando tipos para o tratamento de paralelismo de tarefas (PThreads e MPI), paralelismo de dados (bifurcações e junções), tolerância a falhas (mascaramento, detecção e correção) e proveniência de dados (baseado no modelo *Open Provenance Model*). O *workflow* pode ser modelado graficamente usando um plug-in do Eclipse ou o próprio Acme Studio. Estruturas de controle são programadas através de conectores, garantindo assim a lógica de controle de fluxo da aplicação.

OSC distingue-se por seu mecanismo para tratamento de aspectos não funcionais, particularmente a proveniência de dados e tolerância a falhas, requisitos importantes em sistemas gerenciadores de *workflows* científicos. Tal mecanismo, baseado em um sistema de tipos associados a tarefa e conectores, é extensível, ou seja, outros aspectos não funcionais podem ser acrescentados. A hierarquia de tipos e subtipos em OSC permite definir o controle de fluxo de tarefas e, de uma maneira mais geral, o comportamento do *worflow*. Regras em Acme, expressas através de um estilo arquitetural, regem a forma como devem ser montados *workflows* válidos, por meio de um plug-in para a plataforma Eclipse para modelagem gráfica de *workflows*.

Assim como OSC, HPC Shelf também permite o tratamento de aspectos não funcionais de forma extensível, devido ao suporte ao modelo Hash de componentes e o sistema de contratos contextuais. Entretanto, o aspecto de paralelismo é inerente ao modelo de componentes. Outro aspecto em comum é o tratamento de conectores como entidades de primeira classe. Porém, HPC Shelf trata os conectores como componentes Hash de uma espécie particular, enquanto OSC distingue conectores de tarefas em abstrações distintas.

No que diz respeito a comparação com o SAFe, OSC assemelha-se ao Sedna e ao Kepler, também exigindo dos especialistas, chamados de *cientistas*, conhecimentos sobre soluções computacionais para os seus problemas de interesse e composição de componentes (tarefas e conectores) do modelo. No SAFe, tratam-se de conhecimentos exigidos para os provedores de aplicações. Porém, OSC considera, em seu projeto, usuários conhecidos como *projetistas*, responsáveis pela criação e implementação dos tipos concretos de tarefas e conectores (desenvolvimento para reuso), enquanto os usuários conhecidos como *cientistas* tratam da montagem de soluções. Projetistas possuem conhecimentos que, na HPC Shelf, são atribuídos a provedores de aplicações e desenvolvedores de componentes.

Finalmente, vale ressaltar, que OSC também não implementa mecanismos de descoberta de componentes com base em critérios não funcionais.

5.5 Pegasus

Pegasus [Deelman et al. 2015] é um sistema gerenciador de *workflows* científicos que busca o mapeamento eficiente de uma descrição abstrata de um *workflow* para uma infraestrutura distribuída de recursos computacionais, constituindo um *workflow* concreto. É usada há mais de dez anos pela comunidade científica, com diversas aplicações reais, e está em constante evolução. Algumas de suas características mais importantes são:

- Separação da descrição do *workflow* da descrição do ambiente de execução. Essa abordagem onde o *workflow* é independente da plataforma de execução auxilia na portabilidade e possibilita a inclusão de otimizações em tempo de compilação e execução. No entanto, a representação concreta do *workflow* pode ser bem diferente do original abstrato submetido pelo usuário;

- Assim como outras soluções, seu *workflow* é um *Grafo Acíclico Direcionado* (do inglês, DAG), onde os nós são as computações e as arestas o controle de fluxo e dados;

- Nós de um grafo podem representar outros subgrafos, permitindo assim composição hierárquica de *workflows*;

- Dados e computações podem estar distribuídos em um ambiente possivelmente heterogêneo;

- Cientistas podem interagir com o Pegasus via linha comando, interfaces e ferramentas de alto nível para composição de *workflows*;

- A descrição do grafo do *workflow* é feita em XML, sendo chamada de DAX (*Directed Acyclic graph* XML) a qual provê uma representação independente de recursos. Para gerar o DAX, Pegasus oferece um conjunto de APIs em Python, Java e Perl. Com o XML gerado, é necessário primeiro executar o *parser* para só depois mapear o *workflow*;

- Através de uma abordagem de *catálogos*, Pegasus realiza diversas consultas para achar as informações necessárias ao processo de transformação do *workflow* abstrato em um *worflow* executável (ou seja, escolha dos recursos computacionais).

- Informações sobre o *job* em execução são gerenciadas pelo *pegasus-kickstart*, o qual reúne importantes dados sobre proveniência e performance durante a execução.

- Pegasus gerencia falhas em múltiplos níveis da execução do *workflow*. Nós que falham são reexecutados, *workflows* que falham podem ser replanejados pelo usuário e dados e arquivos tem a garantia de serem transferidos com exatidão.

Pegasus não utiliza uma linguagem própria para descrição de orquestrações de *workflows*, tal como SAFeSWL. Para gerar o *workflow*, a aplicação é inicialmente escrita em Java, Perl ou Python para depois ser transcrita para XML. O XML resultante representa um grafo acíclico que será executado por uma motor de execução (*engine*) específica.

A idéia de separar a representação de *workflows* e suas versões mais abstratas e concreta, a fim diminuir os requisitos para que usuários finais construam soluções, constitui um ponto de convergência com as premissas de projeto do SAFe, o qual vai um pouco mais longe ao preconizar que os *workflows* sejam totalmente escondidos dos usuários final (especialistas) através da interface de alto nível de uma aplicação construída sobre o SAFe. Para realizar isso, o Pegasus aprofunda sobremaneira os mecanismos de análise e transformação de *workflows* com a finalidade de otimizar o uso dos recursos, tomando para si, de forma automatizada, muitas responsabilidades associadas, através do SAFe, aos provedores de aplicações. Mecanismos com propósitos comparáveis ainda são propostos como trabalhos futuros dentro do projeto HPC Shelf.

Por outro lado, Pegasus também não oferece meios comparáveis com contratos contextuais para especificar requisitos de *workflows* abstratos em relação aos recursos que serão alocados aos *workflows*, sendo resolvido por mecanismos internos. Essa decisão de projeto faz parte da premissa de tornar o *workflow* mais abstrato em nível do usuário final. Por outro lado, os catálogos de Pegasus comparam-se ao *Core*, com a finalidade da classificação e busca por recursos.

Assim como Pegasus, SAFe também oferece mecanismos para instanciação incremental de recursos, também com o propósito de otimizar a sua utilização. Porém, ao invés de fazer isso de forma automática, oferece meios que devem ser utilizados em nível de programação em SAFeSWL, através da porta de controle de ciclo de vida oferecida pelos componentes de solução. Portanto, cabe aos provedores de aplicação lidar com esse tipo de otimização, muito embora seja ainda possível

oferecer meios de automatizar essa tarefa, possivelmente, assim como Pegasus, definindo níveis de abstração de *workflow*, suportados por SAFeSWL.

5.6 Taverna

Taverna [Wolstencroft et al. 2013] foi originalmente projetado, dentro da iniciativa myGrid, como uma suíte de ferramentas para o desenvolvimento de *workflows* para aplicações (experimentos *in-silico*) na área de bioinformática, formados por serviços distribuídos ou locais, interligados através de *pipelines*. *Pipelines* interligam recursos locais ou distribuídos em infraestruturas de larga escala, tais como grades computacionais, nuvens computacionais, supercomputadores, *clusters*, etc. Uma vez construídos, os *workflows* podem ser reusados, como serviços disponíveis na *web*. Atualmente, Taverna é um projeto incubado dentro da iniciativa Apache[2], já bastante consolidado e com uma considerável base de usuários, assim como Pegasus, sendo um dos *workflows* científicos mais usados pela comunidade em diversos domínios do conhecimento, extrapolando a bioinformática.

Em suma, *workflows* em Taverna são uma mistura de Serviços *Web* distribuídos, *scripts* locais e outros tipos de serviços. Nada impede também que um *workflow* Taverna seja formado apenas de serviços locais, na tentativa de otimizar a comunicação ou por questões de segurança e privacidade.

As principais características do Taverna, segundo a sua última versão, são:

- Interação com novos tipos de serviços, além dos *web services* usuais;
- Introdução de um repositório para compartilhamento de *workflows*;
- Inclusão de serviço para descoberta e uso de *web services*;
- Servidor Taverna, o qual possibilita a execução de *workflows* de forma remota;
- O Serviço de Interação, o qual permite a manipulação de parâmetros do *workflow* por cientistas durante a sua execução;
- Uma suíte de proveniência, a qual registra o andamento do *workflow* em formato padronizado;
- Melhorias em *plug-ins*, facilitando a contribuição e extensão da plataforma por cientistas;

[2] *The Apache Software Foundation* (http://apache.org/).

- Componentes, com a finalidade de encapsular *workflows* em serviços acessíveis por outros *workflows* (uma forma de composição hierárquica visando o reuso).

Taverna tem como premissa de projeto trabalhar com grandes massas de dados heterogêneos, bem como a suposição sobre a natureza mutável dos serviços distribuídos. "*Ao suportar descoberta de serviço, projeto de workflow reuso e execução, o* Taverna *possibilita a utilização de dados de bioinformática distribuídos e métodos de análise*" [Wolstencroft et al. 2013].

A arquitetura de Taverna implementa uma abordagem em multicamadas, tentando assim abstrair detalhes de execução de código para os cientistas especialistas. A linguagem de *workflow* usada é o Scufl [Oinn et al. 2004], centrada em fluxo de dados, a qual define um grafo de interações entre diversos serviços. Possui um conjunto de *inputs, outputs*, *processadores* (serviço lógico, o qual recebe dados pelas portas de *input* e produz dados na portas de *output*), um conjunto de *data links* (conectando processadores) e *links* de coordenação (controle de fluxo). Durante a execução, informações sobre o *status* dos processos é enviado aos usuários os quais podem controlar seu andamento manualmente.

Assim como os sistemas gerenciadores tradicionais de *workflows*, Taverna visa a integração de serviços independentes de computação científica, embora muitos dos hoje existentes tenham sido desenvolvidos para serem usados em *workflows* baseados em Taverna. Por outro lado, HPC Shelf é orientada a componentes paralelos que oferecem serviços através de suas portas de ambiente e de tarefas. Ou seja, o suporte a serviços tais como os serviços suportados por Taverna depende de seu encapsulamento em componentes e plataformas virtuais em HPC Shelf ou, alternativamente, o suporte a novas espécies de componentes que ofereçam meios mais abstratos para que provedores de aplicações possam acessá-los.

Em sua arquitetura original, Taverna também não suporta um sistema de descoberta de serviços com base em contratos embutido em sua linguagem de descrição de *workflows* (Scufl), tal como o sistema de contratos contextuais. Esses serviços são referenciados diretamente, sendo oferecido ao especialista camadas de abstração para facilitar essa tarefa.

Notadamente, os usuários finais de Taverna são cientistas computacionais que dominam técnicas de experimentação *in-silico*. Possuem, portanto, competências tanto de especialistas quanto de provedores de aplicações de HPC Shelf. Porém, Taverna, pela sua própria natureza extensível, tem servido como base para a construção de aplicações de mais alto nível oferecidas como serviços para usuários

menos especializados em técnicas computacionais, seja pelo compartilhamento de *workflows* ou pela disponibilização de portais de solução de problemas, análogas a aplicações que seriam construídas sobre o arcabouço SAFe.

5.7 Triana

Triana [Harrison et al. 2008] consiste num ambiente gráfico para a construção de *workflows* baseados em serviços (chamados internamente de *tools*). Usuários da interface gráfica selecionam os serviços, tal qual um sistema de diretórios, e os arrastam para o ambiente de trabalho, conectando suas interfaces. Triana também suporta o agrupamento de serviços em componentes compostos, facilitando a visualização e incentivando o reuso.

Internamente, Triana trabalha com diferentes tipos de serviços e *middlewares*, interligando várias tecnologias (entre elas *web services*) o qual possibilita a criação de *workflows* formados por componentes de diversas naturezas. Sendo assim, fazendo uso de um *middleware* de grades, componentes de manipulação de arquivos são interligados com componentes de submissão de *jobs* na grade computacional e seus resultados são apresentados em componentes implementados em Java.

Componentes Triana aceitam dados de entrada, processam, e geram uma saída. Componentes podem ser implementados como uma chamada local a um método de um objeto Java ou uma interface interligada com uma gama de soluções para processos distribuídos, tais como RMI, Serviços *Web* ou *Grid Jobs*. Para tornar possível os *bindings* entre seus componentes, Triana faz uso da *Grid Application Prototype* (GAP) e da *Grid Application Toolkit* (GAT). O desenvolvedor também pode fazer uso das duas tecnologias simultaneamente para criação de *workflows* heterogêneos e dinamicamente mutáveis.

A linguagem usada para a construção de *workflows* Triana é baseada em XML, lembrando um pouco a definição WSDL. Componentes são representados por portas de entrada, de saída e voltadas ao provimento de parâmetros iniciais. Uma *tag* específica detalha as conexões entre as portas. Triana não oferece, assim como outras linguagens, suporte a construtores de controle. Laços e disparos de atividades em paralelo (*forks*) são implementados por componentes específicos. Após construído, o *workflow* passa por um processo de refinamento, o qual mapeia a representação serial em uma versão distribuída, descobrindo os recursos que serão utilizados em tempo de execução.

Assim como o SAFe, Triana faz uso de um formato XML para representar

os componentes de seu *workflow* e conexões entre eles. O fluxo de controle de orquestração não é feito diretamente na linguagem, por meio de combinadores como aqueles suportados por SAFeSWL, mas através de componentes específicos, tornando a linguagem um pouco mais abstrata que o SAFe. Entretanto, tal mecanismo é comparável ao uso de componentes conectores de orquestração na HPC Shelf, o qual encapsularia estruturas de controle, abstraindo seu uso no código SAFeSWL. É possível, portanto, enxergar o uso de conectores (de orquestração) como componentes em Triana, um aspecto que o distingue das demais linguagens de *workflows*, muito embora de uma forma menos expressiva daquela suportada pelos componentes da espécie conector propostos pela HPC Shelf.

Com relação ao nível de abstração oferecido ao usuário final, aquele interessado em construir *workflows*, Triana assemelha-se aos demais sistemas tradicionais de gerenciamento de *workflows*, buscando simplificar a tarefa de composição através do suporte de uma ferramenta de modelagem visual. Entretanto, isso não é suficiente para retirar desses usuários finais responsabilidades e requisitos de competências exigidos para provedores de aplicações na plataforma HPC Shelf, algo que o SAFe objetiva atingir, muito embora esse não seja um objetivo do Triana.

Assim com as demais alternativas, com exceção de Askalon, Triana também não suporta mecanismo de descoberta de componentes baseada em contratos. Os componentes do *workflow* são referenciados diretamente.

5.8 Considerações Finais

Taverna e o Pegasus são extensamente usadas pela comunidade, com uma grande e qualificada base de usuários, e apresentam resultados de anos e evolução sobre seu *software*. Por outro lado, Askalon mostra-se uma ferramenta bastante completa em termos de aspectos não funcionais tratados, incluindo mecanismos sofisticados de tolerância a falhas, suporte à proveniência de dados e separação de interface e implementação através de um sistema análogo a um sistema de contratos. Em geral, esses sistemas gerenciadores visam a integração de um conjunto de recursos, tanto de computação quanto de dados, pré-existentes, acessíveis por meio de diferentes tecnologias, tais como Serviços *Web*, serviços de grades computacionais, arquivos executáveis locais, programas paralelos MPI executando em um *cluster* remoto, etc. Os métodos de ligação desses serviços ao *workflow* variam de uma alternativa para outra, bem com o nível de abstração oferecido ao usuário final.

Por outro lado, SAFe visa atender a requisitos particulares de uma plataforma

	SAFe	Askalon	Sedna	Kepler	OSC	Pegasus	Taverna	Triana
A	X	X	X	X	X	X	X	X
B	X	X	-	-	X	X	X	X
C	X	X	X	-	-	X	X	X
D	X	-	X	X	X	X	X	X
E	X	-	-	-	X	-	-	-
F	X	X	X	X	X	X	X	X
G	X	-	-	-	X	-	-	-
H	X	-	-	-	X	-	-	-
I	X	-	-	-	-	-	-	-
J	X	X	-	-	-	-	-	-
L	X	-	-	-	-	X	-	-
H	X	-	-	-	-	-	-	-

Tabela 5.1: Tabela resumo das características dos sistemas gerenciadores de *workflows* apresentados neste capítulo em comparação com SAFe.

com características específicas, a HPC Shelf. Por esse motivo, SAFe deve ser comparado com suas alternativas em relação aos requisitos específicos que visa tratar, particularmente a separação de interesses entre os usuários finais de sistemas gerenciadores de *workflows* científicos, chamados de especialistas e provedores de aplicações, bem como a expressividade de sua linguagem de *workflows*.

A Tabela 5.1 compara algumas características consideradas mais relevantes no projeto do SAFe com relação ao seu suporte nas demais alternativas apresentadas neste capítulo. As características analisadas são descritas abaixo:

A: Interface gráfica (GUI[3]) para atender ao usuário especialista/montador de aplicações na construção de *workflows*;

B: Linguagem própria para descrição da orquestração de *workflows* científicos;

C: Linguagem para especificação de *workflows* baseia-se em XML;

D: Multi-arquitetural, quando suas aplicações são compatíveis com diferentes arquiteturas de computação de alto desempenho;

E: Descrição arquitetural na linguagem de descrição de *workflows*;

[3] *Graphical User Interface*

F: Suporte a componentes, ou seja, suas aplicações são baseadas na tecnologia de componentes intercambiáveis;

G: Requisitos funcionais e não funcionais, ou seja, de alguma forma o *framework* deve suportar esses dois tipos de requisitos para aplicação.

H: Distinção entre montadores de *workflows* e especialistas, usuários que não desejam lidar diretamente com a construção de soluções computacionais e programação diretamente de *workflows*;

I: Separação entre interface e implementação/implantação de componentes, associado a meios de descoberta de componentes com base em propriedades especificadas em contratos que expressam as suas interfaces;

J: Instanciação incremental dos recursos utilizados por *workflows* de longa duração;

L: Controle explícito do ciclo de vida dos componentes (recursos), permitindo a sua descoberta/resolução, implantação e instanciação explícitas por meio da linguagem de orquestração.

Praticamente todas as soluções oferecem interfaces gráficas (característica A). No caso do SAFe, a interface gráfica é a nível de especialista, criada pelo provedor de aplicações. Essa separação dos usuários montadores de *workflows* em especialistas e provedores de aplicações é uma característica peculiar de SAFe, relacionada à característica H, com o objetivo de que especialistas não preparados para desenvolver soluções computacionais para problemas de seus interesses utilizem uma interface de mais alto nível de abstração, que abstraia-se inclusive dos *workflows* e de seus componentes, centrando-se em abstrações típicas do seu domínio de conhecimento. Uma interface gráfica análoga àquelas oferecidas pelas alternativas ao SAFe poderiam ser implementadas como uma aplicação da HPC Shelf, voltada a especialistas interessados em acessar diretamente os componentes da HPC Shelf e construir *workflows* através de uma versão visual de SAFeSWL.

A característica B faz-se presente na maioria das soluções. Sedna não propõe uma nova linguagem de orquestração, própria, mas propõe extensões o BPEL que constituem, por si só, uma importante contribuição do projeto de pesquisa em torno de si. Kepler, por sua vez, restringe-se a uma linguagem visual de descrição de *workflows*, herdada de um sistema anteriormente proposto pelos seus autores,

o Ptolemy 2. SAFeSWL possui algumas características próprias, cuja avaliação constitui contribuição relevante:

- Relacionado à característica L, o controle do ciclo de vida dos componentes (resolução, implementação, instanciação, execução e liberação) dentro do fluxo de orquestração, permitindo que o provedor de aplicações controle melhor o uso dos recursos em computações de longa duração, típicas de *workflows* científicos, nas aplicações que desenvolve. Outras linguagens de *workflows* controlam apenas o fluxo orquestração relacionado a execução das computações dos componentes (e.g. ativação de ações nas portas de tarefas de componentes da HPC Shelf). Algumas alternativas, tais como Pegasus, oferecem a possibilidade da instanciação incremental de recursos (característica J), porém por meios automáticos, ao invés de inseridos explicitamente na linguagem de *workflows* para controle por parte do usuário final, algo que tornaria a linguagem muito complexa para usuários que não desejam fazer suposições sobre os recursos. SAFe torna isso possível ao fazer distinção entre especialistas e provedores de aplicações, sendo estes últimos supostamente competentes para desenvolvimento de aplicações que fazem uso eficiente dos recursos (componentes da HPC Shelf).

- A separação entre interface e implementação dos componentes do *workflow* (característica I) através de contratos ditos contextuais (sistema Alite), que constituem um mecanismo geral para descrever propriedades sobre o ambiente de execução do componente, incluindo requisitos da aplicação e características desejadas da plataforma de computação paralela onde deverá executar. Dentre as alternativas estudadas, apenas Askalon propõe algo semelhante, através da distinção de *atividades*, como são chamados os componentes de seus *workflows*, em *activity types* e *activity deployments*, porém usando um mecanismos menos expressivo para descrever propriedades para guiar o mapeamento entre ambos.

- Uma descrição arquitetural estática associada à descrição da orquestração, embora em subconjuntos independentes da linguagem, permitindo que a uma mesma descrição arquitetural possam estar associadas várias possibilidades de orquestrações. Conforme a característica E, além do SAFe, apenas OSC oferece o suporte a descrição arquitetural, tendo em vista sua base em ACME.

Quase todas as soluções de *frameworks* que apresentam a implementação de alguma linguagem usam o formato XML para descrever *workflows* (característica

C). XML é um padrão de fato para intercâmbio de arquivos é facilmente manipulado pela maioria das linguagens de programação de alto nível. No caso do SAFe, os componentes que formam seu *workflow* são executados remotamente, sendo suas portas expostas por meio de Serviços *Web*. Sendo assim, o XML é um formato natural para comunicação (interface WSDL).

A característica D diz se a solução pode ser usada em qualquer tipo de arquitetura de alto desempenho. Apenas Askalon restringe-se a grades computacionais. No entanto, isso não significa necessariamente uma desvantagem. Muitas vezes é uma escolha de projeto permitir que uma determinada ferramenta apenas trabalhe com um único tipo de arquitetura, restrição que deve ser considerada na construção de suas aplicações. Uma vez que o HPC Shelf encontra-se assentado sobre nuvens computacionais de infraestrutura (IaaS), oferecidas por mantenedores de plataformas, SAFe seria capaz de usar componentes em plataformas de computação de naturezas distintas, incluindo grades computacionais, clusters, MPP's (*Massive Parallel Processors*), ou mesmo execução local, desde que estejam disponíveis serviços de *Back-End* de infraestrutura, bem como componentes associados a perfis de plataformas oferecidos pelos mantenedores dessas infraestruturas, através do sistema de contratos contextuais. No entanto, essa possibilidade não é avaliada nesta Tese de Doutorado.

Quanto ao suporte a requisitos não funcionais, SAFe o faz em dois níveis, através do sistema de contratos contextuais, onde são expressas propriedades não funcionais que devem ser atendidas por implementações de componentes, e de portas de ambiente, presentes em componentes de todas as espécies, por meio das quais componentes podem ter seu comportamento configurado e trocar informações com a aplicação e demais componentes de solução.

Capítulo 6

Conclusões e Perspectivas de Trabalhos Futuros

A HPC Shelf, uma plataforma de aplicações que demandam pelo uso de computação paralela de larga-escala, é um projeto do grupo de pesquisa em Computação de Alto Desempenho do Programa de Pós-Graduação em Ciência da Computação da Universidade Federal do Cerá. Inserida nesse contexto, o autor desta Tese de Doutorado propôs, como sua principal contribuição, o SAFe, um *framework* para construção de soluções computacionais baseadas em *workflows* científicos construídos com componentes do modelo Hash.

O SAFe é um *framework* implementado sobre a plataforma Java, a fim de garantir a portabilidade sobre um conjunto abrangente de plataformas de computação. Sua implementação é dividida em dois pacotes: um pacote que manipula as classes responsáveis pela construção da aplicação (`safe-framework`) e outro responsável pela linguagem de orquestração de *workflows* chamada de SAFeSWL (`safe-language`). Herda alguns conceitos e padrões de projeto que o modelo CCA propôs, na última década, para *frameworks* computacionais a ele compatíveis.

SAFeSWL é uma linguagem de descrição de *workflows* científicos, implementada em XML, dividida em dois subconjuntos, destinados à descrição arquitetural dos componentes do *workflow* e à orquestração das ações que definem as características funcionais desses componentes, respectivamente. Embora não implemente todas as características mais sofisticadas de linguagens de *workflows* científicos existentes e bem difundidas, SAFeSWL apresenta algumas peculiaridades as quais são exercitadas através dos estudos de caso apresentados nesta tese, tais como o suporte ao sistema de contratos contextuais e o controle explícito das etapas do ciclo de

vida de componentes (descoberta/resolução, implantação, instanciação e execução) através de operadores embutidos na própria linguagem. O uso de XML na sua implementação é possível devido ao fato de que *workflows* (soluções computacionais) não são diretamente escritos através dela pelos usuários finais da HPC Shelf, ditos *especialistas*, mas gerados automaticamente pela aplicação desenvolvida sobre o SAFe pelos usuários chamados de *provedores de aplicações*.

Para demonstrar e validar as características do SAFe e sua implementação, foram selecionadas duas aplicações bem difundidas no meio científico:

- computação de mosaicos usando a ferramenta Montage, de astrofotografia;
- computação paralela de larga-escala através do modelo Map/Reduce.

A aplicação voltada a construir *workflows* do Montage, chamada MoEx, demonstra particularmente a compatibilização de um conjunto de componentes pré-existentes para execução na plataforma HPC Shelf. Uma vez que é usado como estudo de caso na apresentação de vários sistemas gerenciadores de *workflows* científicos, facilita a comparação da expressividade de SAFe com suas alternativas. Por outro lado, a aplicação baseada em Map/Reduce demonstra um projeto especialmente desenhado para a HPC Shelf, incluindo a especificação de contratos contextuais e um protocolo de orquestração mais complexo.

Sendo assim, a partir do que foi implementado e avaliado, podemos tomar as seguintes conclusões a respeito do SAFe:

- Quanto ao **reuso**, utiliza o modelo de programação orientado a componentes, inspirando-se em padrões de projeto e conceitos do modelo CCA adaptados para o modelo Hash de componentes. Componentes devem ser implementados de forma independente, expondo interfaces, tanto funcionais (*portas de tarefas*) quanto não-funcionais (*portas de ambiente*), que facilitam a programação *com* reuso e *para* reuso. Dessa forma, qualquer provedor de aplicações pode *reusar* um mesmo conjunto de componentes em diversas aplicações, gerando *workflows* distintos. Além disso, um componente *workflow* em SAFe também é um componente que pode ser implementado com um conjunto de portas de ambientes e tarefas e ser reusado por outro *workflow* como um componente interno (abordagem semelhante ao BPEL).
- Quanto à **complexidade** na criação de uma aplicação em SAFe, podemos dizer que há um nível de dificuldade semelhante ao criar uma aplicação em

frameworks compatíveis com o modelo CCA. O provedor de aplicações deve ter experiência em programação Java e manipulação de arquivos XML, duas tecnologias bastante disseminadas. Ele também deve conhecer a API do SAFe para, de alguma forma, programar aplicações de mais alto nível que usem essas interfaces para gerar arquivos SAFeSWL de forma transparente ao especialista. Podemos concluir que o SAFe apresenta uma modesta curva inicial de aprendizagem para o provedor de aplicações, que ao longo do tempo se estabiliza, uma vez que torna-se natural a criação de novas aplicações com o reaproveitamento do que já foi utilizado anteriormente.

- Quanto à **eficiência**, para o tipo de sistema computacional executado sobre a HPC Shelf, o peso das operações executadas pelo SAFe na execução de *workflows* de SAFeSWL, bem como das interfaces de serviços, implementados por componentes de ligação específicos (*bindings*) que o permite se comunicar com os componentes de solução, é pouco significativo em relação ao custo das operações de resolução, implantação, instanciação e execução dos componentes, os quais são inerentes à plataforma HPC Shelf. Sistemas computacionais da HPC Shelf pressupõem componentes de grossa granularidade, que exportam ações computacionais intensivas em computação a serem orquestradas por parte do *workflow*. Soma-se a isso o peso da implantação e instanciação de plataformas virtuais, para componentes da espécie plataforma, sempre que a infraestrutura do mantenedor fizer uso de tecnologias de virtualização para oferecer máquinas virtuais. A descoberta e resolução de componentes através de contratos contextuais, implementada pelo sistema Alite, também é uma operação custosa, embora possa ser bastante otimizada para o fim de uma implementação para uso em produção com o emprego de estruturas de dados adequadas e técnicas de memorização de computações previamente realizadas.

- Quanto à **interoperabilidade**, trata-se de um aspecto inerente ao projeto da HPC Shelf, que pode ser avaliado sob duas vertentes: seu modelo de componentes e seu modelo de interconexão (entre os componentes). Com relação aos componentes, virtualmente qualquer tipo de infraestrutura de computação paralela pode ser suportada na HPC Shelf, desde que seus mantenedores ofereçam seus serviços por meio da abstração de plataformas virtuais, para as quais devem ser definidos *perfis de plataformas* (denominação

atribuída aos contratos contextuais para componentes da espécie *plataforma*), cujas implementações representam procedimentos para instanciação de plataformas virtuais, com as características delineadas pelos seus respectivos contratos, sobre a infraestrutura. Entretanto, tais plataformas somente serão utilizadas caso existam implementações de componentes compatíveis para execução sobre ela, possivelmente explorando suas características mais peculiares (e.g. o uso de algum tipo de acelerador computacional específico). Portanto, é responsabilidade do sistema Alite, de contratos contextuais, a ligação entre os componentes e as infraestruturas de computação paralela, bem como, por consequência, das aplicações às plataformas que empregam para execução de soluções computacionais. Com relação à interconexão entre os componentes, *bindings*, de portas de tarefa e de ambiente, também são tratados componentes, sendo portanto também regidos pelo sistema de contratos contextuais. Dessa forma, implementações compatíveis de *bindings* devem existir para conexão entre portas de componentes que executam sobre infraestruturas de computação paralela distintas e de arquiteturas diferentes.

- Quanto à **extensibilidade**, SAFe pode ser estendida pela implementação de novas portas de ambiente, suportando novos serviços ou estendendo os serviços existentes, bem como novas espécies de portas, além de portas de ambiente e de tarefas, capazes de lidar com certos requisitos específicos de comunicação com os demais componentes que compõem um *workflow* (e.g. comunicação de *streams* de *bits* entre a aplicação e componentes de solução, para, por exemplo, comunicação de imagens de visualização de resultados intermediários de uma computação). De uma maneira mais geral, novas espécies de componentes podem ser incorporadas à HPC Shelf e suportadas pelo SAFe. As extensões também são possíveis sobre a linguagem SAFeSWL, as quais são facilitadas pelo uso do padrão *Visitor* na implementação dessa linguagem. Todo o código fonte do SAFe deve ser publicamente disponibilizado.

6.1 Trabalhos Futuros

Esta Tese de Doutorado é o ponto de partida para outros projetos de pesquisas que visam estender o SAFe com outras funcionalidades suportadas por outras alternativas de sistemas gerenciadores de *workflows* científicos, buscando atender às necessidades da plataforma HPC Shelf para uso em produção.

Os parágrafos que se seguem buscam delinear questões de pesquisa a serem tratadas em trabalhos futuros que estenderão os resultados desta Tese de Doutorado.

Segurança

A segurança no acesso aos recursos (componentes) da nuvem é um dos fatores essenciais para uso da plataforma HPC Shelf em produção. No protótipo de implementação atual, componentes da HPC Shelf são acessíveis pelo SAFe através de serviços implementados através de *bindings*. Portanto, a segurança no acesso aos componentes é uma tarefa de responsabilidade do desenvolvedor de componentes. A principal preocupação para implementação de requisitos de segurança diz respeito aos serviços do *Core* e de *Back-End*'s conectados ao *Core*, para os quais ainda não há proteção de acesso aos serviços, o que pode motivar um trabalho pesquisa para estudar requisitos e alternativas de técnicas para sua implementação.

Dentro do conjunto de alternativas de tecnologias que podem ser aplicadas inclui-se a autenticação digital de arquivos XML (*XML Digital Signature*) e sua encriptação (*XML Encryption*). Além disso, a WS-Security é um extensão do padrão SOAP que aplica rotinas de segurança sobre Serviços *Web*. Ela apresenta um protocolo que especifica como questões de integridade e confidencialidade podem ser implementadas nas mensagens trocadas. Diversos são os produtos que implementam essas funcionalidades, dentre os quais podemos citar: OpenSAML[1], Fusion *Middleware*[2], Apache CFX[3], JBoss SSO [4] dentre outras alternativas de código fechado. No entanto, para outras abordagens de comunicação remota, as quais não incluem Serviços *Web*, novas alternativas devem ser estudadas. Por exemplo, uma implementação em *sockets* poderia usar JSSE[5], a qual provê um conjunto de funcionalidades de encriptação e autenticação.

Proveniência de Recursos

Proveniência de recursos é um requisito considerado importante para sistemas gerenciadores de *workflows* científicos modernos. A proveniência de um recurso diz respeito à capacidade de registrar as entidades e os processos envolvidos na produção, transformação e distribuição desse recurso dentro do ambiente de computação. Por exemplo, contextualizando para aplicações de ciências computacionais, quando o

[1] https://wiki.shibboleth.net/confluence/display/OpenSAML/Home
[2] https://www.oracle.com/middleware/index.html
[3] https://cxf.apache.org/
[4] http://www.jboss.org/
[5] https://docs.oracle.com/javase/8/docs/technotes/guides/security/jsse/JSSERefGuide.html

recurso diz respeito a dados de experimentos científicos *in-silico* produzidos ao longo das etapas de um *workflow*, a proveniência (de dados) oferece a possibilidade de autenticação desses dados e reprodutibilidade do experimento, aumentando o seu grau de confiabilidade perante a comunidade científica.

Apesar de ser uma característica importante em *workflows*, o primeiro protótipo da HPC Shelf não apresenta a implementação de uma solução para proveniência de recursos, em especial quando esses recursos são dados. Um dos motivos para isso é que, além de se tratar de um requisito ortogonal aos demais, considerados essenciais, explorados no projeto inicial do SAFe, as próprias oportunidades de pesquisa relacionada a esse assunto são bastante ricas na plataforma HPC Shelf, podendo motivar projetos a serem executados com maior grau de rigor e oferecendo contribuições dentro do contexto geral do assunto.

Uma das ideias iniciais seria ampliar o escopo da espécie de componentes *fonte de dados* para lidar com esse requisito, acrescentando ainda um novo tipo de ator, o *mantenedor de dados*, sobre os quais seriam implementados os requisitos de proveniência de dados. Assim, *fontes de dados* teriam capacidades para monitoramento e catalogação de dados processados pelo *workflow*, através de suas portas de ambiente ou outras espécies de portas e *bindings* que se fizessem úteis.

Linguagem Intermediária

Atualmente, o código SAFeSWL de orquestração é interpretado diretamente, em tempo de execução. Uma outra abordagem seria compilá-lo para uma linguagem intermediária, como, por exemplo, BPEL, uma linguagem difundida para orquestração de *workflows* de aplicações nas áreas corporativas e de negócios, mas que tem sido adaptada e utilizada por sistemas gerenciadores de *workflows* voltados a aplicações científicas, como mostrado no Capítulo 5. Dessa forma, poderíamos criar *workflows* BPEL com as características do SAFe, sendo possível executá-los em quaisquer arquitetura com as bibliotecas compatíveis do BPEL ou até mesmo transformar o *workflow* inteiro em um serviço BPEL, visto como um componente pela HPC Shelf, o qual poderia ser disponibilizado pelo *Core* para outras aplicações. É um trabalho futuro que oferece contribuições para o SAFe e, potencialmente, para a linguagem BPEL, visto que é provável que seja necessário adaptá-la para os propósitos do SAFe.

Certificação Formal de Componentes

Em uma nuvem de componentes, como a plataforma HPC Shelf, oferecer mecanismos que atestem a confiabilidade da implementação de componentes é um requisito importante, por questões de segurança tanto no que diz respeito a comportamento malicioso que pode ser assumido por códigos desenvolvidos por terceiros (desenvolvedores de componentes) quanto no que diz respeito a comportamento anômalo ou errôneo motivado por erros de programação. Tais erros tornam-se potencialmente mais comuns no contexto das aplicações da HPC Shelf por se tratarem de componentes paralelos. Tornam-se ainda mais custosos, tendo em vista que podem causar a invalidação de estudos científicos, devido a produção de resultados errôneos ou cuja confiabilidade não pode ser atestada, além de desperdiçar tempo precioso, tendo em vista que tratam-se de aplicações de longa duração.

Para os propósitos da HPC Shelf, componentes certificadores tem sido propostos garantir que componentes de soluções, que integram *workflows*, sejam certificados, ou seja, que atendam a certas propriedades formais impostas pela aplicação em uma versão estendida de seus contratos contextuais. Essa propriedades visam permitir aumentar a confiabilidade de que o componente produz resultados corretos e em tempo razoável. A "corretude" deve garantir as aplicações cumpram corretamente as tarefas as quais ela se propõe a executar, enquanto a segurança deve impedir que a aplicação atinja estados indesejáveis, tais como, por exemplo, o acesso a posições de memória inexistentes, a existência de *deadlocks*, erros de sincronização em operações de comunicação, o uso de componentes e bibliotecas não instaladas, etc.

6.2 Considerações Finais

Apesar do SAFe ainda estar em estado inicial, ele é uma peça importante para o projeto HPC Shelf, visto que trata-se da interface principal para criação de novas aplicações da plataforma. Tanto a sua API quanto a própria linguagem ainda apresentam bastante espaço e flexibilidade para extensões as quais irão se adequar às necessidades da plataforma e, possivelmente, de outras plataformas alternativa. A linguagem deverá evoluir naturalmente, sendo otimizada e refatorada. Além disso, uma trabalho paralelo de criação de documentação e tutoriais deverá ser posto em prática, facilitando o uso por novos adeptos e disponibilização *on-line*.

A principal dificuldade encontrada para conclusão deste trabalho foi a de não haver ainda uma implementação finalizada, para a HPC Shelf, do sistema Alite, do *Core*, e do serviço de instanciação de plataformas virtuais do *Back-End*. Ambos

dependem da conclusão de trabalhos de pesquisa de Doutorado em andamento. Para contornar essa dificuldade, foram feitas adaptações no *Core* e *Back-End* da plataforma HPE, já em funcionamento, de modo que pudessem ser usados para implementação do SAFe. O HPE ficou então responsável pela resolução de contratos contextuais e também pela implantação, instanciação e execução de componentes concretos definidos pelo *workflow* SAFe. De fato, foi realizada uma extensão não trivial do *Back-End* do HPE, com a inclusão da espécie de componentes *conector* e da abstração de *facetas*, o que ofereceu a possibilidade de uma aplicação estar instanciada sobre múltiplas instâncias do *Back-End* executando em *clusters* distintos. Em cada partição da aplicação, devem estar executando o subconjunto dos componentes de computação e facetas de conectores do *workflow* que estejam mapeados ao *cluster*, representando uma plataforma virtual, onde a partição está instanciada. Através dos conectores, incluindo *bindings* indiretos, as partições de uma aplicação podem se comunicar. Essas lacunas na implementação da HPC Shelf devem ser resolvidas em breve.

Referências Bibliográficas

[Allan et al. 2002]ALLAN, B. A.; ARMSTRONG, R. C.; WOLFE, A. P.; RAY, J.; BERNHOLDT, D. E.; KOHL, J. A. The CCA Core Specification in a Distributed Memory SPMD Framework. *Concurrency and Computation: Practice and Experience*, Wiley, v. 14, n. 5, p. 323–345, 2002.

[Altintas et al. 2004]ALTINTAS, I.; BERKLEY, C.; JAEGER, E.; JONES, M.; LUDASCHER, B.; MOCK, S. Kepler: an Extensible System for Design and Execution of Scientific Workflows. In: IEEE. *Scientific and Statistical Database Management, 2004. Proceedings. 16th International Conference on.* [S.l.], 2004. p. 423–424.

[Amedro et al. 2010]AMEDRO, B.; BAUDE, F.; CAROMEL, D.; DELBÉ, C.; FILALI, I.; HUET, F.; MATHIAS, E.; SMIRNOV, O. An Efficient Framework for Running Applications on Clusters, Grids, and Clouds. In: *Cloud Computing.* [S.l.]: Springer, 2010. p. 163–178.

[Apache Hadoop Project 2013]APACHE Hadoop Project. [s.n.], 2013. Disponível em: <http://hadoop.apache.org/>. Acesso em: 1 mar. 2013.

[Armstrong et al. 1999]ARMSTRONG, R.; GANNON, D.; GEIST, A.; KEAHEY, K.; KOHN, S.; MCINNES, L.; PARKER, S.; SMOLINSKI, B. Toward a Common Component Architecture for High-Performance Scientific Computing. In: IEEE. *High Performance Distributed Computing, 1999. Proceedings. The Eighth International Symposium on.* [S.l.], 1999. p. 115–124.

[Armstrong et al. 2006]ARMSTRONG, R.; KUMFERT, G.; MCINNES, L. C.; PARKER, S.; ALLAN, B.; SOTTILE, M.; EPPERLY, T.; TAMARA, D. The CCA

Component Model For High-Performance Scientific Computing. *Concurrency and Computation: Practice and Experience*, Wiley, v. 18, n. 2, p. 215–229, 2006.

[Baude 2012]BAUDE, F. A Perspective on the CoreGRID Grid Component Model. In: SPRINGER. *Euro-Par 2011: Parallel Processing Workshops*. [S.l.], 2012. p. 115–116.

[Baude, Caromel e Morel 2003]BAUDE, F.; CAROMEL, D.; MOREL, M. From Distributed Objects to Hierarchical Grid Components. *Lecture Notes in Computer Science*, Springer-Verlag, Germany, v. 2888, p. 1226–1242, 2003. ISSN 0302-9743.

[Bernholdt, Nieplocha e Sadayappan 2004]BERNHOLDT, D. E.; NIEPLOCHA, J.; SADAYAPPAN, P. Raising Level of Programming Abstraction in Scalable Programming Models. In: CITESEER. *IEEE International Conference on High Performance Computer Architecture (HPCA), Workshop on Productivity and Performance in High-End Computing (P-PHEC)*. [S.l.], 2004. p. 76–84.

[Bertrand e Bramley 2004]BERTRAND, F.; BRAMLEY, R. DCA: A Distributed CCA Framework Based on MPI. In: *Proceedings of the 9th International Workshop on Highlevel Parallel Programming Models and Supportive Environments (HIPS'2004)*. [S.l.]: IEEE Computer Society, 2004. ISBN 0-7695-2151-7.

[Blair, Coupaye e Stefani 2009]BLAIR, G.; COUPAYE, T.; STEFANI, J.-B. Component-Based Architecture: The Fractal Initiative. *Annals of Telecommunications*, Springer Paris, v. 64, p. 1–4, 2009. ISSN 0003-4347. Disponível em: <http://dx.doi.org/10.1007/s12243-009-0086-1>.

[Carvalho Junior et al. 2007]Carvalho Junior; LINS, R. D.; CORRÊA, R. C.; ARAÚJO, G. A. Towards an Architecture for Component-Oriented Parallel Programming: Research Articles. *Concurr. Comput. : Pract. Exper.*, John Wiley and Sons Ltd., Chichester, UK, UK, v. 19, n. 5, p. 697–719, 2007. ISSN 1532-0626.

[Carvalho Junior e Corrêa 2010]Carvalho Junior, F. H. de; CORRÊA, R. C. The Design of a CCA Framework with Distribution, Parallelism, and Recursive Composition. In: *Workshop on Component-Based High Performance Computing (CBHPC'2010)*. [S.l.]: IEEE, 2010. p. 339–348. ISBN 9781424493470.

[Carvalho Junior e Lins 2005]Carvalho Junior, F. H. de; LINS, R. D. Separation of Concerns for Improving Practice of Parallel Programming. *INFORMATION, An*

International Journal, International Information Institute, v. 8, n. 5, p. 621–638, set. 2005. ISSN 1343-4500.

[Carvalho Junior e Lins 2008]Carvalho Junior, F. H. de; LINS, R. D. An Institutional Theory for #-Components. *Electronic Notes in Theoretical Computer Science*, Springer Verlag, v. 195, p. 113–132, jan. 2008.

[Carvalho Junior e Rezende 2013]Carvalho Junior, F. H. de; REZENDE, C. A. A Case Study on Expressiveness and Performance of Component-Oriented Parallel Programming. *Journal of Parallel and Distributed Computing*, v. 73, n. 5, p. 557–569, 2013. ISSN 0743-7315. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0743731512002882>.

[Carvalho Junior et al. 2013]Carvalho Junior, F. H. de; REZENDE, C. A.; SILVA, J. C.; ALAM, W. G. Al. Contextual Abstraction in a Type System for Component-Based High Performance Computing Platforms. Springer, v. 8129, p. 90–104, set. 2013.

[Church et al. 2012]CHURCH, P.; WONG, A.; BROCK, M.; GOSCINSKI, A. Toward Exposing and Accessing HPC Applications in a SaaS Cloud. In: *Proceedings of the 2012 IEEE 19th International Conference on Web Services*. IEEE, 2012. p. 692–699. ISBN 978-1-4673-2131-0. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6257957>.

[Dean e Ghemawat 2008]DEAN, J.; GHEMAWAT, S. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, ACM, v. 51, n. 1, p. 107–113, 2008. ISSN 0001-0782.

[Deelman et al. 2015]DEELMAN, E.; VAHI, K.; JUVE, G.; RYNGE, M.; CALLAGHAN, S.; MAECHLING, P. J.; MAYANI, R.; CHEN, W.; SILVA, R. F. da; LIVNY, M. et al. Pegasus, a Workflow Management System for Science Automation. *Future Generation Computer Systems*, Elsevier, v. 46, p. 17–35, 2015.

[Epperly et al. 2012]EPPERLY, T. G. H.; KUMFERT, G.; DAHLGREN, T.; EBNER, D.; LEEK, J.; PRANTL, A.; KOHN, S. High-Performance Language Interoperability for Scientific Computing Through Babel. *International Journal of High Performance Computing Applications*, SAGE Journals, v. 26, n. 3, p. 260–274, 2012.

[Forum 2009]FORUM, C. *Proposal: MPI for CCA Components*. [S.l.], jul. 2009.

[Garlan, Monroe e Wile 2010]GARLAN, D.; MONROE, R.; WILE, D. Acme: an Architecture Description Interchange Language. In: IBM CORP. *CASCON First Decade High Impact Papers*. [S.l.], 2010. p. 159–173.

[Govindaraju et al. 2002]GOVINDARAJU, M.; KRISHNAN, S.; CHIU, K.; SLOMINSKI, A.; GANNON, D.; BRAMLEY, R. Xcat 2.0: A Component-Based Programming Model for Grid Web Services. In: *Submitted to Grid 2002, 3rd International Workshop on Grid Computing*. [S.l.: s.n.], 2002.

[Grama et al. 2003]GRAMA, A.; GUPTA, A.; KARYPIS, J.; KUMAR, V. *Introduction to Parallel Computing*. [S.l.]: Addison-Wesley, 2003. 256 p. ISBN 0-201-64865-2.

[Harrison et al. 2008]HARRISON, A.; TAYLOR, I.; WANG, I.; SHIELDS, M. WS-RF Workflow in Triana. *International Journal of High Performance Computing Applications*, SAGE Publications, v. 22, n. 3, p. 268–283, 2008.

[Juric 2006]JURIC, M. B. *Business Process Execution Language for Web Services BPEL and BPEL4WS 2Nd Edition*. [S.l.]: Packt Publishing, 2006. ISBN 1904811817.

[Khalidi 2011]KHALIDI, Y. Building a Cloud Computing Platform for New Possibilities. *Computer*, IEEE Computer Society, Los Alamitos, CA, USA, v. 44, n. 3, p. 29–34, 2011. ISSN 0018-9162.

[Krishnan e Gannon 2004]KRISHNAN, S.; GANNON, D. XCAT3: A Framework for CCA Components as OGSA Services. In: *Proceedings of the HIPS2004 - 9th International Workshop on Highlevel Parallel Programming Models and Supportive Environments*. [S.l.: s.n.], 2004.

[Malawski, Kurzyniec e Sunderam 2005]MALAWSKI, M.; KURZYNIEC, D.; SUNDERAM, V. MOCCA - Towards a Distributed CCA Framework for Metacomputing. In: *Proceedings of the Joint Workshop on High-Performance Grid Computing and High-Level Parallel Programming Models (HIPS-HPGC)*. [S.l.]: IEEE Computer Society, 2005. ISBN 0-7695-2312-9.

[Malawski, Kurzyniec e Sunderam 2005]MALAWSKI, M.; KURZYNIEC, D.; SUNDERAM, V. MOCCA-Towards a Distributed CCA Framework for

Metacomputing. In: IEEE. *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*. [S.l.], 2005. p. 8–pp.

[MCMD-WG Wiki at CCA Forum]MCMD-WG Wiki at CCA Forum. https://www.cca-forum.org/wiki/tiki-index.php?page=MCMD-WG.

[Medeiros e Gomes 2013]MEDEIROS, V.; GOMES, A. T. A. Expressando Atributos Não-Funcionais em Workflows Científicos. *CoRR*, abs/1304.5099, 2013. Disponível em: <http://arxiv.org/abs/1304.5099>.

[Mell e Grance 2011]MELL, P.; GRANCE, T. The NIST Definition of Cloud Computing. *NIST special publication*, v. 800, n. 145, p. 7, 2011.

[Montero, Vozmediano e Llorente 2011]MONTERO, R. S.; VOZMEDIANO, R. Moreno; LLORENTE, I. M. An Elasticity Model for High Throughput Computing Clusters. *Journal of Parallel and Distributed Computing*, v. 71, n. 6, p. 750–757, 2011. ISSN 0743-7315. Special Issue on Cloud Computing. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0743731510000985>.

[Niehorster et al. 2010]NIEHORSTER, O.; BRINKMANN, A.; FELS, G.; KRÜGER, J.; SIMON, J. Enforcing SLAs in Scientific Clouds. In: *Proceedings of the 2010 IEEE International Conference on Cluster Computing (CLUSTER'2010)*. [S.l.: s.n.], 2010. p. 178–187.

[Oinn et al. 2004]OINN, T.; ADDIS, M.; FERRIS, J.; MARVIN, D.; SENGER, M.; GREENWOOD, M.; CARVER, T.; GLOVER, K.; POCOCK, M. R.; WIPAT, A. et al. Taverna: a Tool for the Composition and Enactment of Bioinformatics Workflows. *Bioinformatics*, Oxford Univ Press, v. 20, n. 17, p. 3045–3054, 2004.

[Parker et al. 2010]PARKER, S. G.; DAMEVSKI, K.; KHAN, A.; SWAMINATHAN, A.; JOHNSON, C. R. The SCIJump Framework for Parallel and Distributed Scientific Computing. *Advanced Computational Infrastructures for Parallel and Distributed Adaptive Applications*, p. 151, 2010.

[Plimpton e Devine 2011]PLIMPTON, S. J.; DEVINE, K. D. MapReduce in MPI for Large-scale graph algorithms. *Parallel Computing*, v. 37, n. 9, p. 610–632, 2011. ISSN 0167-8191. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0167819111000172>.

[Post e Votta 2005]POST, D. E.; VOTTA, L. G. Computational Science Demands a New Paradigm. *Physics Today*, MIT Press, v. 58, n. 1, p. 35–41, 2005.

[Qin e Fahringer 2012]QIN, J.; FAHRINGER, T. *Scientific Workflows: Programming, Optimization, and Synthesis with ASKALON and AWDL*. [S.l.]: Springer Science & Business Media, 2012.

[Rehr et al. 2010]REHR, J. J.; VILA, F. D.; GARDNER, J. P.; SVEC, L.; PRANGE, M. Scientific Computing in the Cloud. *Computing in Science Engineering*, v. 12, n. 3, p. 34–43, 2010. ISSN 1521-9615.

[Schmidt et al. 1996]SCHMIDT, D.; STAL, M.; ROHNERT, H.; BUSCHMANN, F. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. [S.l.]: John Wiley & Sons, 1996.

[Siddiqui e Fahringer 2005]SIDDIQUI, M.; FAHRINGER, T. GridARM: Askalon's Grid Resource Management System. In: SLOOT, P. M. A.; HOEKSTRA, A. G.; PRIOL, T.; REINEFELD, A.; BUBAK, M. (Ed.). *Advances in Grid Computing - EGC 2005*. [S.l.]: Springer Berlin Heidelberg, 2005, (Lecture Notes in Computer Science, v. 3470). p. 122–131. ISBN 978-3-540-26918-2.

[Siddiqui et al. 2005]SIDDIQUI, M.; VILLAZON, A.; HOFER, J.; FAHRINGER, T. GLARE: A Grid Activity Registration, Deployment and Provisioning Framework. In: *Proceedings of the ACM/IEEE Conference on Supercomputing (SC'2005)*. [S.l.: s.n.], 2005. p. 52–52.

[Steen 2006]STEEN, A. J. van der. Issues In Computational Frameworks. *Concurr. Comput. : Pract. Exper.*, John Wiley and Sons Ltd., Chichester, UK, v. 18, n. 2, p. 141–150, 2006. ISSN 1532-0626.

[Sukumar, Vecchiola e Buyya 2010]SUKUMAR, K.; VECCHIOLA, C.; BUYYA, R. The Structure of the New IT Frontier: Aneka Platform for Elastic Cloud Computing Applications. *Strategic Facilities Magazine*, Pacific & Strategic Holdings Pte Ltd, 2010.

[Szyperski 2000]SZYPERSKI, C. Component Software and the Way Ahead. Cambridge University Press, New York, NY, USA, p. 1–20, 2000.

[Vecchiola, Pandey e Buyya 2009]VECCHIOLA, C.; PANDEY, S.; BUYYA, R. High-Performance Cloud Computing: A View of Scientific Applications. In: IEEE.

Pervasive Systems, Algorithms, and Networks (ISPAN), 2009, Proceedings of the 10th International Symposium on. [S.l.], 2009. p. 4–16.

[Wang 2005]WANG, K. Q. A. J. A. *Component-Oriented Programming.* [S.l.]: Wiley Inter-Science, 2005. (LNCS).

[Wassermann et al. 2007]WASSERMANN, B.; EMMERICH, W.; BUTCHART, B.; CAMERON, N.; CHEN, L.; PATEL, J. Workflows for E-Science: Scientific Workflows for Grids. In: TAYLOR, I. J.; DEELMAN, E.; GANNON, D. B.; SHIELDS, M. (Ed.). London: Springer London, 2007. cap. Sedna: A BPEL-Based Environment for Visual Scientific Workflow Modeling, p. 428–449. ISBN 978-1-84628-757-2. Disponível em: <http://dx.doi.org/10.1007/978-1-84628-757-2_26>.

[Wolstencroft et al. 2013]WOLSTENCROFT, K.; HAINES, R.; FELLOWS, D.; WILLIAMS, A.; WITHERS, D.; OWEN, S.; SOILAND-REYES, S.; DUNLOP, I.; NENADIC, A.; FISHER, P. et al. The Taverna Workflow Suite: Designing and Executing Workflows of Web Services on the Desktop, Web or in the Cloud. *Nucleic Acids Research*, Oxford Univ Press, p. gkt328, 2013.

[Zaspel e Griebel 2011]ZASPEL, P.; GRIEBEL, M. Massively Parallel Fluid Simulations on Amazon's HPC Cloud. In: *Proceedings of the First International Symposium on Network Cloud Computing and Applications (NCCA'2011).* [S.l.: s.n.], 2011. p. 73–78.

[Zhang et al. 2004]ZHANG, K.; DAMEVSKI, K.; VENKATACHALAPATHY, V.; PARKER, S. SCIRun2: A CCA Framework for High Performance Computing. In: *Proceedings of the 9th International Workshop on Highlevel Parallel Programming Models and Supportive Environments (HIPS'2004).* [S.l.]: IEEE Computer Society, 2004. ISBN 0-7695-2151-7.

[Zhang et al. 2004]ZHANG, K.; DAMEVSKI, K.; VENKATACHALAPATHY, V.; PARKER, S. G. SCIRun2: A CCA Framework for High Performance Computing. In: IEEE. *High-Level Parallel Programming Models and Supportive Environments, 2004. Proceedings. Ninth International Workshop on.* [S.l.], 2004. p. 72–79.

Apêndice A

Linguagem Arquitetural

Neste apêndice, apresentamos a implementação do subconjunto arquitetural da linguagem SAFeSWL.

A Gramática Arquitetural em XSD

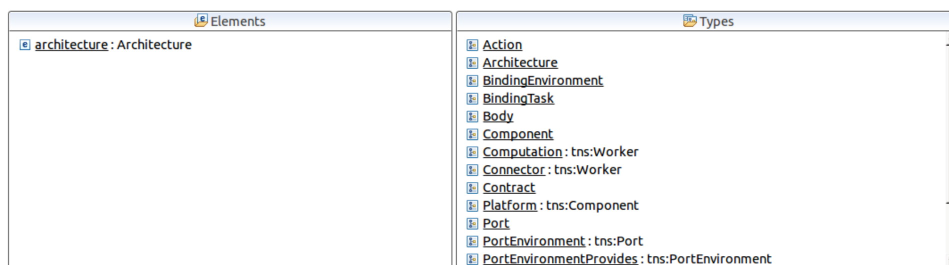


Figura A.1: *Tipos disponíveis para a montagem de uma arquitetura em XML.*

A gramática do subconjunto arquitetural da linguagem SAFeSWL foi implementada dentro de um arquivo de extensão `.xsd`. Ela serve apenas para validação de arquivos XML que representam arquiteturas descritas em SAFeSWL, de modo que possam ser repassadas entre a aplicação e os módulos do SAFe. A Figura A.1 apresenta os principais tipos da linguagem.

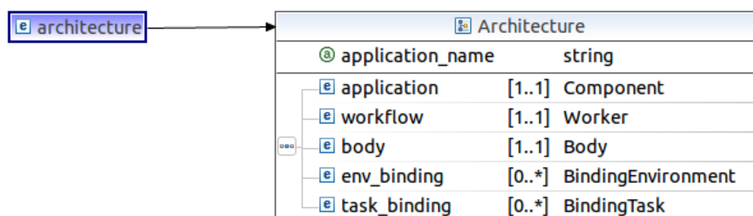


Figura A.2: *A estrutura de uma tag* `<architecture>`

A Figura A.2 apresenta o “corpo” principal de uma arquitetura, iniciada por um *Workflow* (*tag* <architecture>). Note também que são definidas as multiplicidades de cada *tag*. Só podemos ter uma única <application> e um único <workflow>. A <body> é formado por diversos componentes, como veremos a seguir. Já <env_binding> e <task_binding> pode aparecer múltiplas vezes pois posso ter várias conexões entre portas.

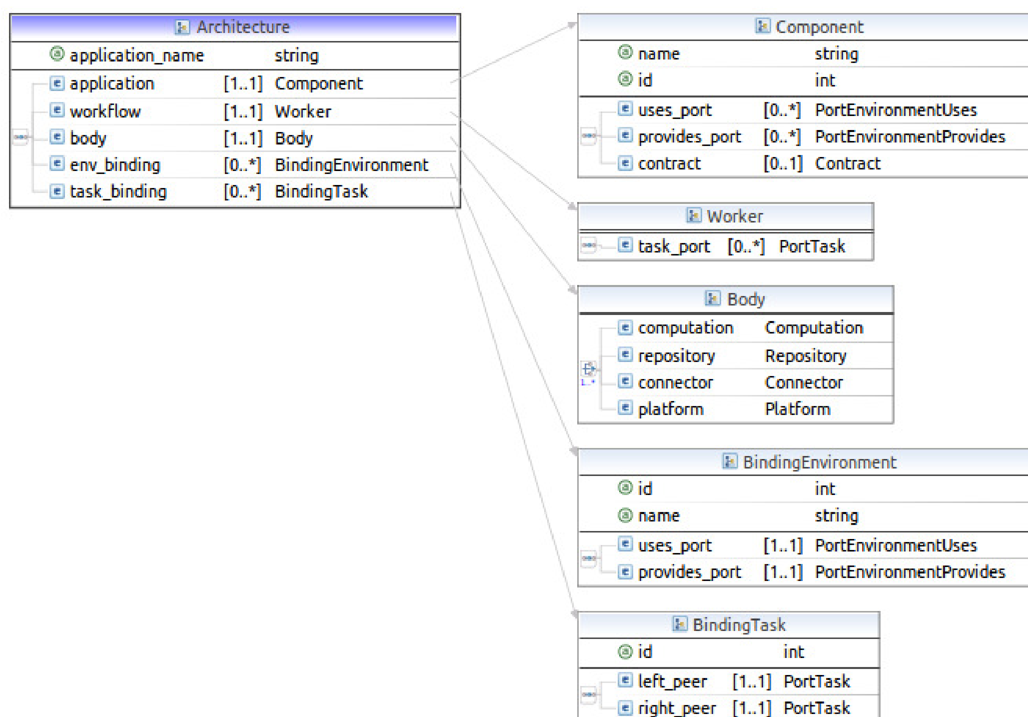


Figura A.3: *A estrutura de uma tag* <body>

A Figura A.3 mostra que um corpo (<body>) é formado por diversas *tags*, as quais devem ser obrigatoriamente escritas em uma sequencia. As *tags* são componentes que fazem parte do *workflow* e podem ser de diversas espécies. A *tag* <env_binding> é formada pela dupla <uses_port> e <provides_port> que formam uma conexão entre duas portas de ambientes de componentes distintos. Já a *tag* <task_binding> também é formada por uma dupla que representa duas portas “parceiras” de tarefas, ou seja, a conexão entre dois componentes via porta de tarefas.

Código XSD da Gramática Arquitetural

Nesta seção apresentamos o código em XML para o subconjunto arquitetural. O código completo pode ser baixado em

https://github.com/UFC-MDCC-HPC/HPC-Storm-SAFe.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.example.org/SAFe_architecture_V4"
  xmlns:tns="http://www.example.org/SAFe_architecture_V4"
  elementFormDefault="qualified"
  xmlns:Q1="http://www.example.org/instantiator">

  <import schemaLocation="instantiator.xsd"
    namespace="http://www.example.org/instantiator">
  </import>

  <element name="architecture" type="tns:Architecture"></element>

  <complexType name="Architecture">
    <sequence>
      <element name="application" type="tns:Component"
        maxOccurs="1" minOccurs="1">
      </element>
      <element name="workflow" type="tns:Worker" maxOccurs="1"
        minOccurs="1">
      </element>
      <element name="body" type="tns:Body" maxOccurs="1"
        minOccurs="1">
      </element>
      <element name="env_binding" type="tns:BindingEnvironment"
        maxOccurs="unbounded" minOccurs="0">
      </element>
      <element name="task_binding" type="tns:BindingTask"
        maxOccurs="unbounded" minOccurs="0">
      </element>
    </sequence>
    <attribute name="application_name" type="string"></attribute>
  </complexType>

  <complexType name="Component">
    <sequence>
      <element name="uses_port" type="tns:PortEnvironmentUses"
        maxOccurs="unbounded" minOccurs="0">
      </element>
      <element name="provides_port"
        type="tns:PortEnvironmentProvides" maxOccurs="unbounded"
```

```
        minOccurs="0">
      </element>
      <element name="contract" type="tns:Contract"
      maxOccurs="1" minOccurs="0"></element>
    </sequence>
    <attribute name="name" type="string"></attribute>
    <attribute name="id" type="int"></attribute>
  </complexType>

  <complexType name="Body">
    <choice maxOccurs="unbounded" minOccurs="1">
      <element name="computation" type="tns:Computation"></element>
      <element name="repository" type="tns:Repository"></element>
      <element name="connector" type="tns:Connector"></element>
      <element name="platform" type="tns:Platform"></element>
    </choice>
  </complexType>


  <complexType name="BindingEnvironment">
    <sequence>
      <element name="uses_port" type="tns:PortEnvironmentUses"
        maxOccurs="1" minOccurs="1">
      </element>
      <element name="provides_port"
        type="tns:PortEnvironmentProvides"
        maxOccurs="1" minOccurs="1">
      </element>
    </sequence>
    <attribute name="id" type="int"></attribute>
    <attribute name="name" type="string"></attribute>
  </complexType>

  <complexType name="BindingTask">
    <sequence>
      <element name="left_peer" type="tns:PortTask"
      maxOccurs="1"
        minOccurs="1">
      </element>
      <element name="right_peer" type="tns:PortTask"
      maxOccurs="1"
        minOccurs="1">
```

```
      </element>
    </sequence>
    <attribute name="id" type="int"></attribute>
  </complexType>

  <complexType name="Computation">
    <complexContent>
      <extension base="tns:Worker"></extension>
    </complexContent>
  </complexType>

  <complexType name="Repository">
    <complexContent>
      <extension base="tns:Component"></extension>
    </complexContent>
  </complexType>



  <complexType name="Connector">
    <complexContent>
      <extension base="tns:Worker"></extension>
    </complexContent>
  </complexType>

  <complexType name="Platform">
    <complexContent>
      <extension base="tns:Component"></extension>
    </complexContent>
  </complexType>

  <complexType name="Worker">
    <complexContent>
      <extension base="tns:Component">
        <sequence>
          <element name="task_port" type="tns:PortTask"
            maxOccurs="unbounded" minOccurs="0">
          </element>
        </sequence>
      </extension>
    </complexContent>
  </complexType>
```

```
<complexType name="PortEnvironment">
  <complexContent>
    <extension base="tns:Port"></extension>
  </complexContent>
</complexType>

<complexType name="PortTask">
  <complexContent>
    <extension base="tns:Port">
      <sequence>
        <element name="action" type="tns:Action"
          maxOccurs="unbounded" minOccurs="0">
        </element>
      </sequence>
    </extension>
  </complexContent>
</complexType>

<complexType name="Port">
  <attribute name="id" type="int"></attribute>
  <attribute name="id_component" type="int"></attribute>
  <attribute name="name" type="string"></attribute>
  <attribute name="wsdl_path" type="string"></attribute>
</complexType>

<complexType name="PortEnvironmentUses">
  <complexContent>
    <extension base="tns:PortEnvironment"></extension>
  </complexContent>
</complexType>

<complexType name="PortEnvironmentProvides">
  <complexContent>
    <extension base="tns:PortEnvironment"></extension>
  </complexContent>
</complexType>

<complexType name="Contract">
  <choice>
    <element name="uri" type="anyURI"></element>
    <element name="instantiation_type"
```

```
      type="Q1:InstanceType"></element>
    </choice>
    <attribute name="id" type="int" use="required"></attribute>
    <attribute name="name" type="string"></attribute>
  </complexType>

  <complexType name="Action">
    <attribute name="id" type="int"></attribute>
    <attribute name="name" type="string"></attribute>
  </complexType>
</schema>
```

Listagem A.1: *Gramática arquitetural em XSD.*

Apêndice B

Linguagem de Orquestração

Neste apêndice, apresentamos a implementação do subconjunto de orquestração da linguagem SAFeSWL.

A Gramática da Linguagem de Orquestração

A implementação da linguagem de fluxo de execução compreende em uma gramática escrita em XSD a qual explicita um conjunto de *tags* responsáveis pela construção da lógica do *workflow*. Como dito, XSD permite a construção de elementos complexos, formados por outros elementos, de forma recursiva.

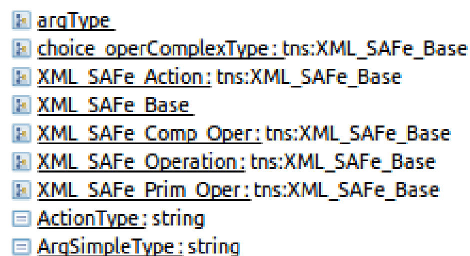


Figura B.1: *Os tipos disponíveis de workflow executável.*

A Figura B.1 mostra uma representação dos tipos disponíveis da gramática em XSD de *workflows*, visualizada com a ferramenta Eclipse. Nela, um programa escrito com a linguagem de fluxo de execução inicia com a *tag* `<workflow>`, a qual é um elemento complexo do tipo `XML_SAFe_Prim_Oper`.

O elemento `XML_SAFe_Prim_Oper`, na Figura B.2 é formado pela escolha de um dos vários elementos que forma um *workflow* em SAFe.

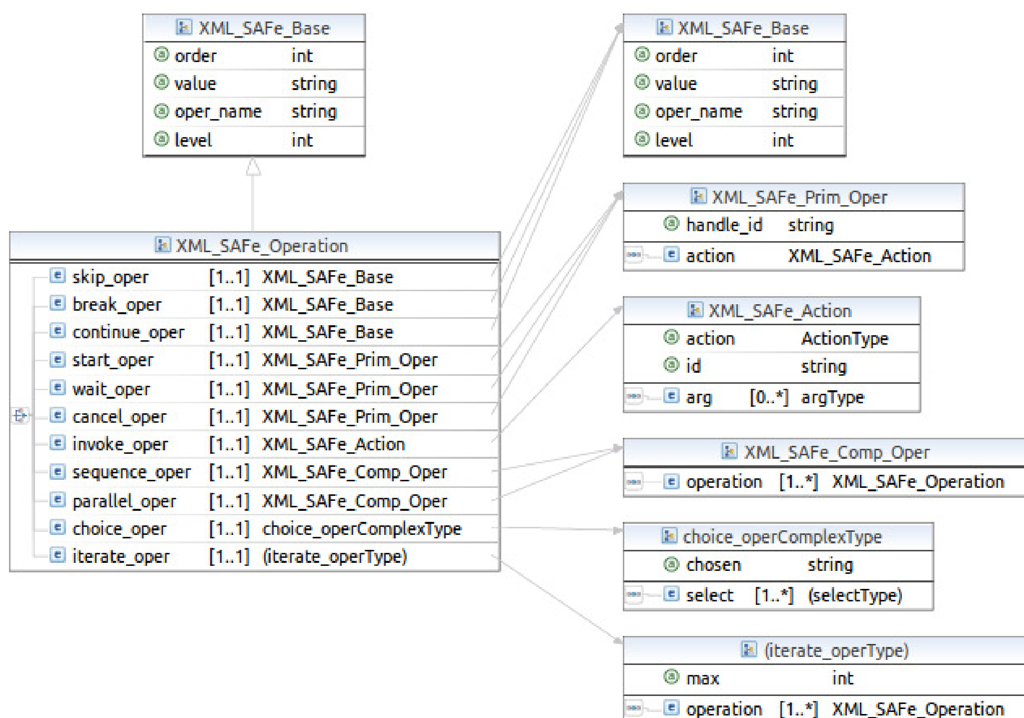


Figura B.2: *A estrutura da elemento* `XML_SAFe_Prim_Oper` (elemento principal), implementado de acordo com a gramática abstrata.

Código XSD da Linguagem de Orquestração

Nesta seção apresentamos o código em XML para o subconjunto de orquestração. O código completo pode ser baixado em https://github.com/UFC-MDCC-HPC/HPC-Storm-SAFe.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.example.org/SAFe_workflow_V4"
xmlns:tns="http://www.example.org/SAFe_workflow_V4"
elementFormDefault="qualified">

  <element name="workflow" type="tns:XML_SAFe_Operation"></element>

  <complexType name="XML_SAFe_Operation">
    <complexContent>
      <extension base="tns:XML_SAFe_Base">
        <choice>
          <element name="skip_oper" type="tns:XML_SAFe_Base"
            maxOccurs="1" minOccurs="1">
          </element>
```

```
            <element name="break_oper" type="tns:XML_SAFe_Base"
              maxOccurs="1" minOccurs="1">
            </element>
            <element name="continue_oper" type="tns:XML_SAFe_Base"
              maxOccurs="1" minOccurs="1">
            </element>
            <element name="start_oper" type="tns:XML_SAFe_Prim_Oper"
              maxOccurs="1" minOccurs="1">
            </element>
            <element name="wait_oper" type="tns:XML_SAFe_Prim_Oper"
              maxOccurs="1" minOccurs="1">
            </element>
            <element name="cancel_oper" type="tns:XML_SAFe_Prim_Oper"
              maxOccurs="1" minOccurs="1">
            </element>
            <element name="invoke_oper" maxOccurs="1" minOccurs="1"
              type="tns:XML_SAFe_Action">
            </element>
            <element name="sequence_oper" type="tns:XML_SAFe_Comp_Oper"
              maxOccurs="1" minOccurs="1">
            </element>
            <element name="parallel_oper" type="tns:XML_SAFe_Comp_Oper"
              maxOccurs="1" minOccurs="1">
            </element>
            <element name="choice_oper" maxOccurs="1" minOccurs="1"
            type="tns:choice_operComplexType">
            </element>
            <element name="iterate_oper" maxOccurs="1" minOccurs="1">
              <complexType>
                <complexContent>
                  <extension base="tns:XML_SAFe_Base">
                    <sequence>
                      <element name="operation"
                        type="tns:XML_SAFe_Operation"
                        maxOccurs="unbounded" minOccurs="1">
                      </element>
                    </sequence>
                    <attribute name="max" type="int">
                    </attribute>
                  </extension>
                </complexContent>
              </complexType>
```

```
        </element>
      </choice>
    </extension>
  </complexContent>
</complexType>



<complexType name="XML_SAFe_Base">
  <attribute name="order" type="int"></attribute>
  <attribute name="value" type="string"></attribute>
  <attribute name="oper_name" type="string"></attribute>
  <attribute name="level" type="int"></attribute>
</complexType>

<complexType name="XML_SAFe_Action">
  <complexContent>
    <extension base="tns:XML_SAFe_Base">
      <sequence>
        <element name="arg" type="tns:argType"
        maxOccurs="unbounded" minOccurs="0"></element>
      </sequence>
      <attribute name="action" type="tns:ActionType"
        use="required">
      </attribute>
      <attribute name="id" type="string" use="required"></attribute>
    </extension>
  </complexContent>
</complexType>

<complexType name="XML_SAFe_Prim_Oper">
  <complexContent>
    <extension base="tns:XML_SAFe_Base">
      <sequence>

        <element name="action" type="tns:XML_SAFe_Action"
        nillable="true"></element>

      </sequence>
      <attribute name="handle_id" type="string"
      use="optional"></attribute>
    </extension>
```

```
    </complexContent>
  </complexType>

  <complexType name="XML_SAFe_Comp_Oper">
    <complexContent>
      <extension base="tns:XML_SAFe_Base">
        <sequence>
          <element name="operation" type="tns:XML_SAFe_Operation"
            maxOccurs="unbounded" minOccurs="1"></element>
        </sequence>
      </extension>
    </complexContent>
  </complexType>

  <simpleType name="ActionType">
    <restriction base="string">
      <enumeration value="resolve" />
      <enumeration value="instantiate" />
      <enumeration value="compute" />
      <enumeration value="deploy" />
    </restriction>
  </simpleType>
  <complexType name="choice_operComplexType">
    <complexContent>
      <extension base="tns:XML_SAFe_Base">
        <sequence>
          <element minOccurs="1" maxOccurs="unbounded"
            name="select">
            <complexType>
              <complexContent>
                <extension base="tns:XML_SAFe_Base">
                  <sequence>
                    <element minOccurs="1"
                      maxOccurs="1" name="operation"
                      type="tns:XML_SAFe_Operation">
                    </element>
                  </sequence>
                  <attribute name="action_id"
                    type="string">
                  </attribute>
                </extension>
              </complexContent>
```

```
            </complexType>
          </element>
        </sequence>
        <attribute name="chosen" type="string"></attribute>
      </extension>
    </complexContent>
  </complexType>


  <complexType name="argType">
    <attribute name="value" type="string"></attribute>
    <attribute name="type" type="tns:ArgSimpleType"></attribute>
  </complexType>

  <simpleType name="ArgSimpleType">
    <restriction base="string">
      <enumeration value="INTEGER"/>
      <enumeration value="DOUBLE"></enumeration>
      <enumeration value="STRING"></enumeration>
    </restriction>
  </simpleType>
</schema>
```

Listagem B.1: *Gramática de Orquestração em XSD.*