Universidade Federal do Rio Grande do Norte
Centro de Ciências Exatas e da Terra
Departmento de Informática e Matemática Aplicada
Programa de Pós-Graduação em Sistemas e Computação
Mestrado Acadêmico em Sistemas e Computação

Uma Máquina de Redução de Grafos Extensível para a Implementação de Fluxos de Trabalho

Márcio Alves de Macêdo

Natal-RN

Fevereiro de 2015

Márcio Alves de Macêdo

Uma Máquina de Redução de Grafos Extensível para a Implementação de Fluxos de Trabalho

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Sistemas e Computação do Departamento de Informática e Matemática Aplicada da Universidade Federal do Rio Grande do Norte como requisito parcial para a obtenção do grau de Mestre em Sistemas e Computação.

Linha de pesquisa:
Linguagens de Programação e Métodos Formais

Orientador

Martin Alejandro Musicante

Co-Orientador

Alberto Pardo

PPgSC – Programa de Pós-Graduação em Sistemas e Computação
DIMAp – Departamento de Informática e Matemática Aplicada
CCET – Centro de Ciências Exatas e da Terra
UFRN – Universidade Federal do Rio Grande do Norte

Natal-RN

Fevereiro de 2015

Catalogação da Publicação na Fonte. UFRN / SISBI / Biblioteca Setorial
Centro de Ciências Exatas e da Terra – CCET.

Macêdo, Márcio Alves de.
Uma máquina de redução de grafos extensível para a implementação de fluxos de trabalho / Márcio Alves de Macêdo. - Natal, 2015.
121 f.: il.

Orientador: Prof. Dr. Martin Alejandro Musicante.
Coorientador: Prof. Dr. Alberto Pardo.

Dissertação (Mestrado) – Universidade Federal do Rio Grande do Norte. Centro de Ciências Exatas e da Terra. Programa de Pós-Graduação em Sistemas e Computação.

1. Linguagens de máquina – Dissertação. 2. Serviços web – Dissertação. 3. Composição de serviços – Dissertação. 4. Máquina de redução de grafos – Dissertação. 5. Big data – Dissertação. 6. Linguagem BPEL – Dissertação. I. Musicante, Martin Alejandro. II. Pardo, Alberto. III. Título.

RN/UF/BSE-CCET CDU: 004.431.2

MÁRCIO ALVES DE MACÊDO

Uma máquina de redução de grafos extensível para a implementação de fluxos de trabalho

Esta Dissertação foi julgada adequada para a obtenção do título de mestre em Sistemas e Computação e aprovado em sua forma final pelo Programa de Pós-Graduação em Sistemas e Computação do Departamento de Informática e Matemática Aplicada da Universidade Federal do Rio Grande do Norte.

Martin Alejandro Musicante – UFRN
(Presidente)

Uirá Kulesza – UFRN
(Coordenador do Programa)

Banca Examinadora

Umberto Souza da Costa – UFRN
(Examinador)

Alberto Pardo – URep
(Examinador)

Genoveva Vargas Solar – CNRS
(Examinador)

Agradecimentos

Agradeço inicialmente a Deus por me dar, além dos obstáculos, forças para enfrentá-los.

Agradeço à minha mãe Maria da Assunção, meu porto seguro, pelo incentivo constante, que me ajudou a nunca desistir dos desafios.

Obrigado a Martin e Alberto, pela orientação nesses dois anos de trabalho.

Obrigado aos meus amigos, pelo apoio nas horas difíceis. Em especial, agradeço aos meus amigos Ítalo e Simone, pelo suporte e ajuda nos momentos de dificuldade.

Obrigado a minha amiga e companheira, Heloína, pelo apoio, ajuda e compreensão em todos os momentos.

A todos que contribuíram de alguma forma para essa realização, o meu muito obrigado.

Uma Máquina de Redução de Grafos Extensível para a Implementação de Fluxos de Trabalho

Autor: Márcio Alves de Macêdo
Orientador(a): Martin Alejandro Musicante
Co-Orientador (a): Alberto Pardo

Resumo

Máquinas de redução de grafos, são tradicionalmente utilizadas na implementação de linguagens de programação. Elas permitem executar programas (representados como grafos), através da aplicação sucessiva de regras de redução. A composição de serviços web permite a criação de novos serviços web a partir de serviços web já existentes. BPEL é a linguagem padrão para criar composições de serviços web como fluxos de trabalho. No entanto, o uso de BPEL para definir composições que usem outras tecnologias, além dos serviços web não é imediato. Na maioria dos casos, quando operações que não fazem parte do domínio dos serviços web precisam ser executadas nas regras de negócio de uma empresa, parte do trabalho é realizado de forma ad-hoc. Permitir que operações oriundas de diferentes tecnologias possam fazer parte de um mesmo fluxo de trabalho auxilia a criação de fluxos de trabalho mais adequados às necessidades das organizações. Esta dissertação define uma variante da linguagem BPEL para a criação de composições com operações de serviços web, tarefas de big data ou operações definidas pelo usuário. O suporte a esta linguagem é dado mediante a definição de uma máquina de redução de grafos extensível, a qual permite a execução de programas definidos na linguagem proposta. Esta máquina é implementada como prova de conceito. A proposta deste trabalho é avaliada mediante a apresentação de resultados experimentais.

Palavras-chave: Serviços web, Composição de serviços, Máquina de redução de grafos, Big data, BPEL.

An Extensible Graph Reduction Machine for Workflow Implementation

Author: Márcio Alves de Macêdo
Supervisor: Martin Alejandro Musicante
Co-Supervisor (a): Alberto Pardo

Abstract

Graph Reduction Machines, are a traditional technique for implementing functional programming languages. They allow to run programs by transforming graphs by the successive application of reduction rules. Web service composition enables the creation of new web services from existing ones. BPEL is a workflow-based language for creating web service compositions. It is also the industrial and academic standard for this kind of languages. As it is designed to compose web services, the use of BPEL in a scenario where multiple technologies need to be used is problematic: when operations other than web services need to be performed to implement the business logic of a company, part of the work is done on an ad hoc basis. To allow heterogeneous operations to be part of the same workflow, may help to improve the implementation of business processes in a principled way. This work uses a simple variation of the BPEL language for creating compositions containing not only web service operations but also big data tasks or user-defined operations. We define an extensible graph reduction machine that allows the evaluation of BPEL programs and implement this machine as proof of concept. We present some experimental results.

Keywords: Web Services, Service Composition, Graph Reduction Machines, Big data, BPEL.

Lista de abreviaturas e siglas

UDDI (*Universal Description, Discovery, and Integration*)

W3C (*Word Wide Web Consortium*)

WSDL (*Web Services Description Language*)

SOAP (*Simple Object Access Protocol*)

HTML (*Hypertext Markup Language*)

XML (*Extensible Markup Language*)

BPEL (*Business Process Execution Language*)

PEWS (*Path Expressions for Web Services*)

BPML (*Business Process Modeling Language*)

WSCI (*Web Service Choreography Interface*)

WSFL (*Web Services Flow Language*)

PRAM (*Parallel Random Access Machine*)

BSP (*Bulk Synchronous Parallel*)

HDFS (*Hadoop Distributed Filesystem*)

YARN (*Yet Another Resource Negotiator*)

STG (*Spineless Tagless G-machine*)

GHC (*Haskell Glasgow Compiler*)

FIFO (*First In First Out*)

PLY (*Python Lex-Yacc*)

Yacc (*Yet Another Compiler - Compiler*)

SQL (*Structured Query Language*)

Sumário

1 Introdução

Desenvolvimento baseado em componentes (caixas pretas) que se comunicam entre si é hoje uma realidade [1]. Serviços web [2] são uma implementação deste tipo de componentes.

A composição de serviços web [3] responde a muitas das exigências das novas formas de desenvolvimento. Principalmente, no que toca à descrição dos sistemas (ou, como era chamado na década de 80 e 90, *Programming in the large*).

No entanto, as linguagens de composição de serviços tem sido desenvolvidas para dar suporte direto apenas a serviços web [4]. O uso de componentes heterogêneos (para a formulação de composições híbridas) requer conhecimentos que extrapolam aqueles do programador usual.

Outras propostas (como MapReduce [5], BSP [6], Pig Latin [7], RPC/RMI [8]) oferecem primitivas que são complementares àquelas dos serviços web e que podem ser combinadas/usadas no desenvolvimento de aplicações de composição.

Isto motiva este trabalho: a proposta de uma ferramenta que integre, da forma mais transparente possível, as vantagens de se construir composições de módulos que não sejam apenas serviços web, mas oriundos de outras tecnologias.

Existem diversas ferramentas para facilitar a análise de grandes volumes de dados. Destas ferramentas, Hadoop [9] é a que atrai mais atenção por ser utilizada por grandes empresas, porque permite que os dados sejam processados por diversos modelos de programação e pode funcionar em *clusters* compostos de máquinas não tão potentes.

O Hadoop permite a execução de diversos modelos de programação para processamento dos dados, possuindo modelos que realizam processamento em lotes (MapReduce), processamento de grafos (Giraph [10]), consultas relacionais no estilo SQL (Hive [11]), dentre outros.

Assim como na composição de serviços, existem ferramentas que permitem a criação

de composições de tarefas de *big data* [12, 13]. No entanto, não existem ferramentas que permitam realizar composições mistas com operações de diferentes domínios, tais como serviços web, tarefas do Hadoop ou operações definidas pelo usuário.

1.1 Motivação

Este trabalho tem como motivação propor uma linguagem que permita a criação de fluxos de trabalho mistos, isto é, composições que possuam operações de chamadas de serviços web, tarefas de *big data*, funções definidas pelo usuário ou qualquer outro tipo de operação que o usuário queira utilizar. A linguagem proposta será uma abstração que representará linguagens baseadas em fluxos de trabalho, como BPEL [14] e PEWS [15]. O propósito principal é expandir linguagens de composição baseadas em fluxos de trabalho para que elas possam realizar composições com operações de diferentes domínios.

Acreditamos que, graças à ferramenta proposta nesta dissertação, pessoas que já tem conhecimento prévio de ferramentas de composição de serviços web, tais como BPEL e PEWS enfrentariam menos problemas para criar composições mistas, podendo combinar numa mesma composição tarefas de *big data*, serviços web, operações definidas pelo usuário ou operações de qualquer outro domínio.

1.2 Objetivos

O objetivo geral deste trabalho é propor uma máquina de redução de grafos extensível, de forma a permitir a criação de fluxos de trabalho envolvendo serviços web, tarefas de *big data*, funções definidas pelo usuário, bem como qualquer outro tipo de operação que o usuário queira utilizar.

Este trabalho possui os seguintes objetivos específicos:

- Propor uma linguagem que permita a criação de fluxos de trabalho com operações de diferentes domínios;
- Propor as regras de tradução e redução para compor uma máquina de redução de grafos extensível. As regras de tradução irão transformar programas escritos na linguagem proposta em grafos. As regras de redução transformarão estes grafos em novos grafos e a extensibilidade permitirá que novas operações possam ser utilizadas pela máquina utilizando pouco esforço;

- Implementar a máquina de redução de grafos proposta;
- Propor estudos de caso para demonstrar o funcionamento da máquina de redução implementada.

1.3 Organização do trabalho

Este trabalho está organizado em capítulos, divididos da seguinte forma: no capítulo 2 são apresentados os principais conceitos relacionados a serviços web, composição de serviços, *big data* e máquinas de redução. O capítulo 3 apresenta dois trabalhos relacionados: PEWS-AM e Pony. O capítulo 4 apresenta a máquina proposta MiniBPEL-AM. A implementação da máquina MiniBPEL-AM é mostrada no capítulo 5. Estudos de caso e análises são apresentados no capítulo 6, demonstrando o funcionamento da máquina para diferentes domínios. Por fim no capítulo 7 são apresentadas as conclusões do trabalho, principais contribuições, limitações e trabalhos a serem desenvolvidos.

2 Fundamentação

Este capítulo descreve os conceitos relacionados a serviços web, composição de serviços e linguagens de composição de serviços web. Serão apresentados conceitos de *big data*, a ferramenta Hadoop, composição de tarefas do Hadoop e será feita uma listagem das ferramentas que podem ser usadas para construir composições de tarefas do Hadoop, mostrando as vantagens e desvantagens de cada ferramenta. Por fim será mostrado o conceito de máquinas de redução de grafos, seu surgimento e onde são utilizadas atualmente.

2.1 Serviços Web

Serviços web são aplicações que podem ser executadas remotamente (via Internet) e possuem uma interface bem definida que descreve *como* as operações do serviço web devem ser acessadas, bem como os parâmetros de entrada e saída que cada operação utiliza.

Com o crescimento da Internet uma onda de inovação mudou o modo como os negócios são conduzidos nas organizações. Para acompanhar este crescimento as empresas tiveram de se adaptar. Visando uma melhor automação, processos de negócio mais eficientes e visibilidade global, as organizações e governos procuravam por uma solução de negócios mais robusta e integrada, que pudesse ser utilizada em suas aplicações já existentes, adaptar-se rapidamente às suas necessidades e evoluir junto com os os requisitos do modelo de negócios da organização [16].

Assim os serviços web começaram a se popularizar como solução para empresas que desejavam implementar modelos de negócio mais dinâmicos. Os serviços web promoveram a implementação de aplicações fracamente acopladas em detrimento das antigas aplicações fortemente acopladas, que eram utilizadas pelas empresas nos modelos de negócio anteriores.

Serviços web representam um meio de implementar componentes para aplicações com

acesso à Internet. São componentes de *software* modulares, auto contidos e auto descritos. Estes componentes são disponibilizados na Internet podendo ser acessados diretamente ou com o auxílio de diretórios de serviços web. A principal tecnologia utilizada para descrever estes diretórios é a UDDI (*Universal Description, Discovery, and Integration*) [17].

Com o auxílio de diretórios UDDI, serviços web podem ser facilmente publicados e buscados. Desta forma uma organização pode publicar um dos seus serviços na Internet, fazendo uso de diretórios UDDI, enquanto um cliente pode realizar uma busca no mesmo diretório UDDI, encontrar o serviço publicado pela empresa e fazer uso do serviço sem a necessidade de um contato direto com a empresa que publicou o serviço.

De acordo com o W3C (*Word Wide Web Consortium*) [18] os serviços web são sistemas de *software* desenhados para dar suporte a interações entre computadores utilizando uma rede. Serviços web possuem interfaces que são descritas por um documento em um formato que possa ser lido pelas máquinas, chamado WSDL (*Web Services Description Language*) [19]. Eles Interagem com outros serviços através de trocas de mensagens SOAP (*Simple Object Access Protocol*) [20], que normalmente são transportadas utilizando HTML (*Hypertext Markup Language*) [21] e XML (*Extensible Markup Language*) [22].

2.1.1 Composição de Serviços Web

A composição de serviços web cria um novo serviço a partir da combinação de outros serviços pré existentes, respeitando algumas regras e restrições definidas pela linguagem de composição utilizada [23].

Existem dois paradigmas de composição de serviços web: orquestração e coreografia [3]. Na orquestração existe a figura do serviço orquestrador, que fica responsável por coordenar os demais serviços da composição. Já a coreografia define uma forma não centralizada de colaboração entre os serviços que participam da composição.

Este trabalho terá foco na composição de serviços web por orquestração, desta forma, de agora em diante quando o termo composição de serviços web for utilizado, ficará implícito que se trata de uma composição de serviços web por orquestração.

Serviços web são descritos usando WSDL. Esta descrição declara apenas o nome e os tipos das operações do serviço web, definindo somente a especificação da interface do serviço. Ela não descreve o comportamento observável do serviço web. Além disto, WSDL não possui meios de descrever composições de serviços web. A necessidade por interfaces

de comportamento melhor definidas, motivou a definição de novas linguagens e técnicas de descrição para estas interfaces [24].

2.1.2 Linguagens de Composição

A literatura dispõe de uma série de linguagens para a especificação de composições, cada uma com suas regras, restrições e modos de montar a composição.

A linguagem BPEL (*Business Process Execution Language*) [14] é o padrão usado hoje na maioria das aplicações de serviços web compostos. Outras linguagens incluem PEWS (*Path Expressions for Web Services*) [15] que utiliza *predicate path expressions* para criar composições; BPML (*Business Process Modeling Language*) [25] que define um modelo abstrato e uma gramática para expressar processos executaveis, não limitado apenas a serviços web; WSCI (*Web Service Choreography Interface*) [26] é uma extenção do WSDL que define como os serviços web pertencentes a uma coreografia devem se comunicar [3]; WSFL (*Web Services Flow Language*) [27] é uma linguagem de composição de serviços web que define como os serviços devem trocar mensagens e como o fluxo de execução deve ser executado [3]; e XLANG [28] que tem foco na criação de processos de negócio e no comportamento da troca de mensagens entre serviços participantes de uma composição [3].

Neste trabalho será utilizada uma notação de composição de serviços web definida de forma abstrata, a qual contém os principais construtores presentes em linguagens de orquestração como BPEL e PEWS. Escolheu-se BPEL por ser a linguagem para criação de composições mais utilizada atualmente e PEWS por ter sido criada no grupo de pesquisa ao qual este trabalho está inserido.

Com o surgimento de novas tecnologias, se faz necessário utilizar novas operações, tais como operações de *big data*, em um fluxo de trabalho para realização de uma tarefa. A próxima sub-seção detalhará o que é *big data* e irá mostrar ferramentas e operações que podem ser utilizadas em um fluxo de trabalho em conjunto com serviços web.

2.2 Big Data: Processamento de Grandes Volumes de Dados

Big data refere-se ao conjunto de dados que é tão grande que os sistemas de banco de dados tradicionais falham em suas tarefas de captura, armazenamento, gerenciamento e

análise destes dados [13].

Esta definição não impõe o tamanho que *big data* pode ter. Com o avanço da tecnologia o tamanho do conjunto de dados que define *big data* também cresce. O tamanho de *big data* também varia dentro de diferentes setores da indústria, dependendo das ferramentas disponíveis para o tratamento tradicional de dados e do tamanho dos dados usados normalmente.

Considerando estes fatos, *big data* pode ter diferentes tamanhos dentro de diferentes setores da indústria, variando de alguns terabytes[1] a milhares de petabytes[2].

Para realizar o processamento de conjuntos de dados tão grandes e complexos novas tecnologias e ferramentas foram propostas. Dentre elas Hadoop é a que vem atraindo mais atenção, por ser de código livre e conseguir realizar o processamento de dados, reduzindo problemas com escalabilidade, falhas e tempo de processamento.

2.2.1 Modelos Paralelos

O tradicional modelo de Von Neumann serviu de base para a implementação e compreensão de algoritmos seqüenciais por muito tempo. Servindo até mesmo como modelo de referência para a criação de *hardware*. No entanto este modelo não apresenta bons resultados para algoritmos paralelos, de forma que novos modelos de máquinas foram propostos, tais como os modelos PRAM (*Parallel Random Access Machine*) [29] e BSP (*Bulk Synchronous Parallel*) [6].

Um dos primeiros modelos que permitia a implementação de algoritmos para programação paralela foi o modelo PRAM. Este modelo permitia um melhor raciocínio teórico sobre a implementação de algoritmos paralelos, porém possuía uma série de premissas que não podiam ser cumpridas pelo *hardware*, principalmente porque o custo de comunicação é maior que o custo computacional e o número de processadores é limitado [30].

Para refletir melhor as características de *hardware*, Valiant propôs o modelo BSP. Um computador BSP pode ser definido como um conjunto de p processadores, cada um com memória local e conectados por algum meio de ligação ponto a ponto. Algoritmos BSP realizam ações em *supersteps*, onde cada processador recebe uma entrada de dados no início do *superstep*, realiza processamento de forma assíncrona, e comunica qualquer dado de saída ao final do *superstep*. Uma barreira de sincronização é utilizada ao final de

[1] 10^{12} bytes ou 1000 gigabytes
[2] 10^{15} bytes ou 1000 terabytes

casa *superstep* com o propósito de sincronizar cada um dos p processadores do sistema. Cada processador pode se comunicar diretamente com os demais processadores provendo controle total de como os dados são distribuídos em cada *superstep*.

Mesmo bem especificados, nenhum destes modelos atraiu tanta atenção quanto o modelo MapReduce [5], por permitir que algoritmos paralelos possam ser facilmente implementados e executados. Este modelo vem atraindo cada vez mais adeptos e sendo aplicado por grandes empresas, processando uma quantidade de dados nunca antes vista.

2.2.2 MapReduce

MapReduce é um modelo de programação com uma implementação associada utilizado para processar e gerar grandes conjuntos de dados. Este modelo disponibiliza uma interface que facilita a implementação por parte do programador de algoritmos que processam eficientemente uma grande quantidade de dados.

Desenvolvido e largamente utilizado pelo Google [5], MapReduce também faz parte do projeto de *open source* Apache Hadoop, sendo utilizado por grandes empresas como Yahoo!, IBM, The New York Times, Facebook, Twitter e um grande número de universidades [31]. Empresas como Amazon e Microsoft permitem até que seus usuários executem programas MapReduce em suas nuvens [32, 33]. MapReduce vem sendo utilizado para resolver problemas complexos de diversas áreas, incluindo processamento de dados, mineração de dados e análise de grafos [30].

No *framework*[3] MapReduce, uma computação é definida como a seqüência de passos *map*, *shuffle* e *reduce* que operam sobre o conjunto de valores $X = \{x_1, x_2, \ldots, x_n\}$:

- No passo *map* uma função μ é aplicada a cada valor x_i para produzir um conjunto de chaves/valor (k, v). Para permitir o paralelismo o processamento da função $\mu(x_i)$ deve depender apenas de x_i.
- O passo *shuffle* combina todos os conjuntos chave/valor gerados no passo *map* que possuem uma mesma chave, produzindo um conjunto chave/valor intermediário $P_k = (k, L_k)$, onde $L_k = \{v_1, v_2, \ldots\}$ é uma lista com os valores dos pares que possuem a mesma chave k.

[3]Um *framework* é uma arquitetura codificada para resolver problemas de um domínio e que pode ser utilizado para resolver problemas específicos [34]. Trazendo como principais benefícios modularidade, reusabilidade, extensibilidade e inversão de controle [35].

- No passo *reduce* uma função ρ é aplicada a cada uma dos P_k pares de chave/valor intermediários criados no passo *shuffle*, para produzir um conjunto de valores, $y_1, y_2, \ldots$. A função ρ pode ser definida seqüencialmente em P_k, porém seu processamento deve ser independente de outras listas $P_{k'}$ com $k \neq k'$.

O paralelismo do MapReduce vem do fato de cada função μ e ρ pode ser executada em um processador independentemente dos demais. O usuário deve definir apenas as funções μ e ρ e o *framework* se encarrega de executar os passos *map-shuffle-reduce* e enviar dados para os processadores disponíveis [31].

2.2.3 Hadoop

O conjunto de bibliotecas Hadoop[4] [37] é um *framework* gratuito para programação, que dá suporte ao processamento de grandes conjuntos de dados em um ambiente de computação distribuída, utilizando modelos de programação simples. Atualmente ele faz parte do projeto Apache e é financiado pela fundação Apache [38].

O *framework* é adequado ao processamento em *clusters* de milhares de nós e analisa milhares de terabytes de dados, permitindo que desenvolvedores possam criar aplicações simples sem se preocuparem com os problemas comuns de aplicações distribuídas, tais como: concorrência, controle de falhas, gerenciamento de recursos, dentre outros.

O Hadoop tenta preservar a localidade dos dados enviando as aplicações para próximo dos dados, diminuindo assim o tráfego de dados pela rede, uma vez que as aplicações são bem menores que os conjuntos de dados em que se trabalha.

O HDFS (*Hadoop Distributed Filesystem*) [39] é um sistema de arquivos distribuído que permite o acesso em larga escala dos dados armazenados. Possui uma interface limitada que permite o gerenciamento de arquivos e possibilita a escalabilidade do sistema. O HDFS utiliza blocos de tamanho definido para armazenamento de dados. Estes blocos são replicados pelo *cluster* para permitir acesso rápido e confiável.

O Hadoop foi inspirado no *framework* MapReduce[5] do Google [5]. Por conta disto, as versões iniciais do Hadoop (Figura 1, lado esquerdo) possuíam forte ligação com o *framework* MapReduce. Este modelo antigo trazia problemas à manipulação de arquivos,

[4]O criador do Hadoop, Doug Cutting, usou o nome do elefante de pelúcia de seu filho para dar nome ao *framework* [36]

[5]Este *framework* quebra aplicações em pedaços menores que podem ser executados em diferentes nós de um *cluster* de forma concorrente.

pois o gerenciamento de recursos (HDFS) estava diretamente ligado ao modelo de programação MapReduce. Esta ligação dificultava a realização de certos tipos de operações com os dados armazenados, pois forçava o uso do MapReduce para que houvesse acesso aos dados. [40].

O *framework* YARN (*Yet Another Resource Negotiator*) [40] foi criado para diminuir o acoplamento entre o gerenciamento de recursos e o *framework* de programação. Neste contexto, MapReduce é apenas uma das aplicações que será executada utilizando o YARN como interface, como mostram as Figuras 1 e 2. Outros *frameworks* como Tez [41], Spark [42], Storm [43], Giraph [10], e outros, foram adaptados e passaram a ser executados pelo YARN. Desta forma, existe uma maior flexibilidade na escolha dos *frameworks* que acessarão os dados armazenados no sistema de arquivos distribuído.

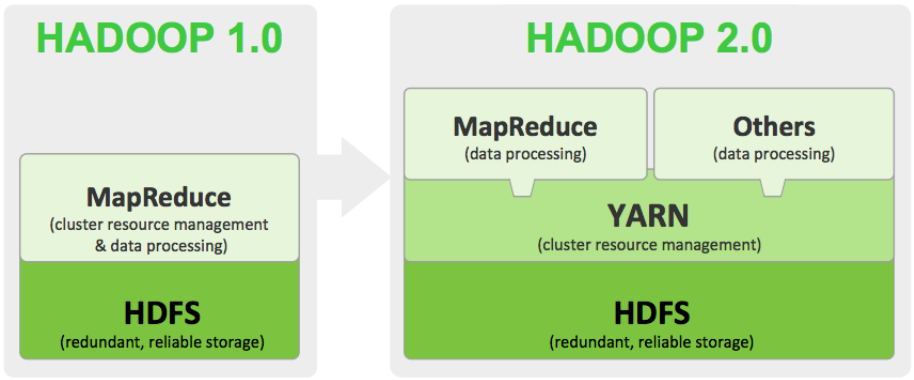


Figura 1: Desacoplamento do gerenciamento de recursos [44].

Como mostrado na Figura 1, MapReduce passa a realizar apenas o processamento de dados, deixando o gerenciamento de recursos a cargo do YARN.

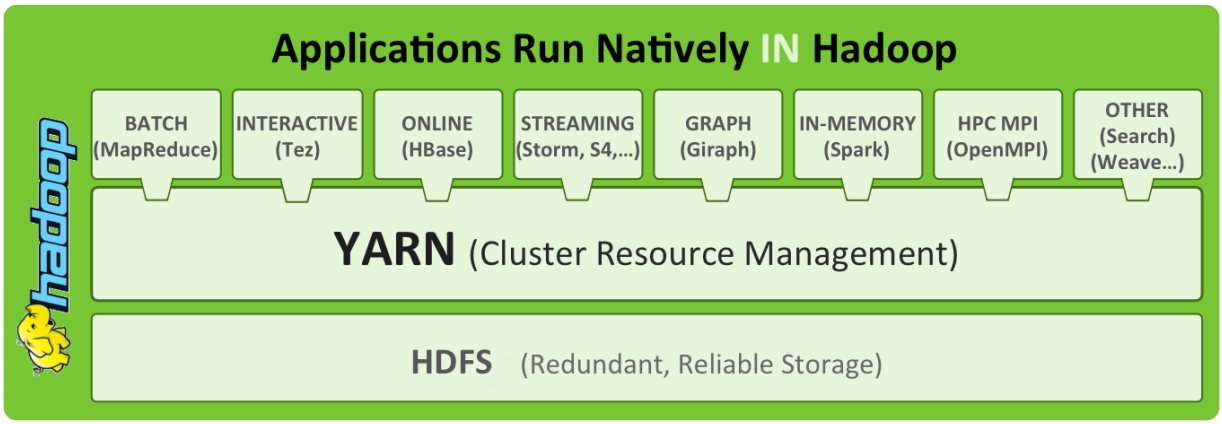


Figura 2: YARN gerencia recursos para múltiplos modelos de programação [44].

O Hadoop, em sua versão 2.2.0 (Figura 2), é formado pelo sistema de arquivos distribuídos do Hadoop (HDFS), *framework* YARN, *framework* MapReduce baseado no YARN

e um conjunto de utilitários comuns que dão suporte aos outros módulos do Hadoop.

Segue a lista de alguns dos projetos que podem ser utilizados em um *cluster* Hadoop:

- MapReduce permite o processamento em paralelo de grandes conjuntos de dados. Quebra o conjunto de dados em dados menores que são processados por códigos fornecidos pelos usuários dentro das funções *Map* e *Reduce*;
- Hive facilita a utilização do Hadoop por profissionais que estão mais familiarizados com SQL, permitindo-lhes consultar dados armazenados no Hadoop como nos bancos de dados relacionais tradicionais. Pesquisas podem ser feitas nestes dados utilizando um dialeto SQL chamado HiveQL, que por sua vez são traduzidas para tarefas MapReduce [12].
- Pig assim como Hive possibilita a criação de *scripts* que permitem que usuários possam trabalhar com dados armazenados no Hadoop sem a necessidade de implementar tarefas MapReduce diretamente. Pig utiliza uma linguagem de *script* de alto nível chamada Pig Latin. Primitivas desta linguagem são traduzidas para tarefas MapReduce. Caso a linguagem não possua alguma função, o usuário pode estender a linguagem criando suas próprias funções [45].

O interesse pela utilização do Hadoop vem crescendo tanto no meio acadêmico quanto no meio empresarial. Dentre as empresas que utilizam o Hadoop estão: Google, Yahoo, IBM, Amazon, Facebook, dentre muitas outras [9].

2.2.4 Composição de Tarefas do Hadoop

Assim como nos serviços web, existem ferramentas para realização de composições de tarefas para o Hadoop. As seguintes ferramentas são algumas das opções que se pode utilizar para criar composições:

- Cascading [46] é uma camada de abstração para o MapReduce. Permite a criação e execução de composições, escondendo a complexidade de tarefas MapReduce [36];
- Hamake [47] foi desenvolvido para resolver problemas de processamento incremental de grandes conjuntos de dados. Permite a formulação de soluções em forma de fluxos de dados, ao invés de composição de tarefas;

- Azkaban [48] é um *framework* de execução de propósito geral que permite a execução de tarefas de vários tipos, como tarefas MapReduce, Pig, Hive, shell *scripts*, dentre outras [49];
- CloudWF [50] um sistema de composição escalável executado pelo Hadoop. Suas composições são formadas por tarefas MapReduce ou programas legados (aplicações que não dependem da API do MapReduce);
- Apache Oozie [51] é um sistema de composição de tarefas que busca implementar escalabilidade, concorrência, segurança e operabilidade [52];

Estas ferramentas de composição possuem um conjunto limitado de primitivas para o processamento da composição. Em geral existem apenas os construtores: *fork*, *join* e *decision*, os quais fornecem, respectivamente, a capacidade de iniciar uma nova linha (*thread*) de processamento, juntar *threads* e fazer escolhas entre várias alternativas [53].

Cada uma destas abordagens adota um conjunto de práticas e métodos para realização de composições, o que dificulta a integração de tais ferramentas com o processo de negócio das empresas. Há um período de aprendizado, treinamento e adaptação para utilização destas ferramentas em conjunto com as ferramentas que gerenciam os processos de negócios das empresas. Empresas que já utilizam BPEL acabam tendo que gastar mais recursos para que seus sistemas possam trabalhar em conjunto com o Hadoop.

O objetivo deste trabalho é propor uma ferramenta que facilite a integração de tecnologias de composição. Permitindo que empresas que já utilizam alguma linguagem de composição de serviços web, como BPEL, PEWS, WSFL, dentre outras, possam utilizar o conhecimento adquirido nestas linguagens para criação de composições de serviços web em conjunto com outros tipos de operações, como por exemplo, tarefas do Hadoop, funções definidas pelo usuário ou outras.

A máquina proposta trará os benefícios de linguagens como BPEL para o âmbito do *big data*. Um conjunto maior de diretivas lógicas podem ser utilizadas, facilitando a adaptação de empresas que já usam linguagens de composição de serviços para resolução de problemas mais complexos. Possibilita ainda que tarefas do Hadoop possam trocar informações com operações de outros domínios dentro de uma mesma composição.

2.2.5 Apache Oozie

Oozie é um sistema de fluxos de trabalho desenhado especialmente para trabalhar com Hadoop [45]. Este sistema foi desenhado como uma aplicação web feita em Java [52].

Os fluxos de trabalho do Oozie são representados como sequencias de ações escritas em um arquivo xml, com cada ação apontando para a próxima ação a ser executada. Uma ação inicial e final devem ser informadas para que o Oozie possa saber onde iniciar e finalizar a execução do fluxo de trabalho. As ações *fork* e *join* do Oozie são utilizadas com o mesmo propósito da primitiva || (paralelo) da máquina MiniBPEL-AM, ou seja, permitir a execução de ações em paralelo.

Coordinators são utilizados para disparar fluxos de trabalho quando determinados eventos ocorrem, como a chegada de um arquivo em um diretório ou a verificação de uma determinada data. Oozie permite a execução de diversos tipos de tarefas Hadoop, tais como: mapreduce, pig, hdfs, hive e scoop. Porem o sub-conjunto de operações hdfs que pode ser executado é reduzido, não permitindo por exemplo que copia de arquivos possam ser realizadas.

2.3 Máquinas de Redução de Grafos

Uma máquina de redução de grafos representa um programa na forma de grafo e aplica sucessivas regras de redução até que o grafo seja reduzido a um grafo irredutível ou um valor. Este processo de redução representa a execução do programa.

O termo redução de grafos foi descrito pela primeira vez por Wadsworth [54] como parte de um processo que descreve uma solução para avaliação de expressões. Este processo permitia o compartilhamento de expressões avaliadas, desta forma uma expressão era avaliada zero ou no máximo uma vez. O processo ficou conhecido posteriormente como avaliação preguiçosa ou *lazy evaluation* [55, 56].

Em uma redução de grafos, programas e suas expressões são representados como grafos direcionados. Um conjunto de regras é definido para realizar transformações nestes grafos. A redução de grafos permite que uma expressão representada como um grafo possa ser reduzida, através de uma transformação, para um outro grafo que representa a avaliação da expressão. As regras de transformação são aplicadas sucessivamente até que não seja mais possível aplicar regras.

Uma das vantagens de se utilizar redução de grafos é a possibilidade de se compartilhar

o resultado de expressões já avaliadas, isto é, caso uma expressão E_1 tenha sido avaliada e uma outra expressão E_2 necessite do resultado de E_1, E_1 não será avaliada novamente para que E_2 possa usar seu resultado.

O Exemplo 1 a seguir descreve o processo de avaliação de um pequeno programa, mostrando como a redução de grafos pode ajudar na avaliação de expressões. Porém antes se faz necessário revisar alguns conceitos relacionados a linguagens de programação usados neste exemplo.

No contexto de linguagens de programação a estratégia de avaliação dos argumentos de uma função é um importante aspecto que deve ser observado. A estratégia de avaliação determina onde e quando a chamada das funções e a avaliação dos argumentos devem ocorrer. A avaliação pode ser *estrita* ou *não estrita*. Na avaliação *estrita* os parâmetros de uma função são sempre avaliados antes da chamada da função. Enquanto que na avaliação *não estrita* os argumentos de uma função só são avaliados caso estes sejam utilizados no corpo da função.

A ordem normal de avaliação na avaliação *não estrita* é uma estratégia de avaliação onde o *redex*[6] mais a esquerda é sempre reduzido, aplicando as funções antes de avaliar seus argumentos. No Exemplo 1 será feita a comparação entre as avaliações *chamada por nome* (*call-by-name*) e *chamada por necessidade* (*call-by-need*) da avaliação *não estrita*. Na estratégia de avaliação *chamada por nome* as expressões só são avaliadas quando chamadas e são avaliadas todas as vezes que elas forem utilizadas [57]. Já na estratégia de avaliação *chamada por necessidade* as expressões são avaliadas quando são chamadas e as expressões são avaliadas apenas uma vez, guardando o resultado da avaliação de uma expressão para uso posterior [58].

Exemplo 1:

O código mostrado na Figura 3 é o de um programa que calcula o quadrado do quadrado de sete. São definidas as funções: `sqr` que calcula o quadrado de um número e retorna o resultado através da aplicação da função `multi`; a função `main` realiza o cálculo do quadrado do quadrado de sete; e a função `multi` que realiza a multiplicação algébrica de dois números.

Realizando a execução deste programa pela ordem normal de avaliação e utilizando a estratégia de avaliação *chamada por nome*, tem-se a ordem de avaliação apresentada pela Figura 4.

[6] A palavra *redex* é uma abreviação para *reducible expression* e representa uma expressão que pode ser reduzida.

```
sqr x = multi x x
main = sqr ( sqr 7 )
```

Figura 3: Programa que calcula o quadrado do quadrado de sete.

```
1 main  -> sqr ( sqr 7 )
2       -> multi ( sqr 7 ) ( sqr 7 )
3       -> multi ( multi 7 7 ) ( sqr 7 )
4       -> multi ( 49 ) ( sqr 7 )
5       -> multi ( 49 ) ( multi 7 7 )
6       -> multi ( 49 ) ( 49 )
7       -> 2401
```

Figura 4: Avaliação *chamada por nome*.

As funções sublinhadas indicam quando cada uma das expressões foi avaliada. Para o código acima, a primeira função a ser avaliada, na linha 1, foi a função `main` resultando na expressão `sqr ( sqr 7 )`. Neste ponto a primeira função `sqr` é avaliada. Substituindo a ocorrência da primeira função `sqr` pela função `multi` tem-se como resultado a expressão da linha 2. Segue-se deste modo avaliando todas as funções quando necessário até obter como o resultado final da computação o valor 2401.

Neste exemplo pode-se notar que a avaliação *chamada por nome* realiza computação em expressões que já haviam sido calculadas antes. Como pode ser notado pela avaliação de `sqr 7` nas linhas 2 e 4, e de `multi 7 7` nas linhas 3 e 5. Este comportamento faz com que a avaliação de *chamada por nome*, não tenha um bom desempenho quando expressões aparecem mais de uma vez no corpo de uma função.

Utilizando a estratégia de avaliação *chamada por nome*, não há a possibilidade de se compartilhar o resultado de avaliações já computadas. Porém quando a expressão é representada em forma de grafo, há a possibilidade de se compartilhar o resultado das avaliações através da utilização de ponteiros. Esta estratégia foi proposta inicialmente por Wadsworth [54] e serviu de inspiração para a implementação da avaliação *chamada por necessidade*, ou *lazy evaluation* [55, 56].

A representação de uma expressão em forma de grafo é descrita na Figura 5, onde as aplicações de funções(@) são nós binários. Para a redução, procura-se o *redex* mais exterior e mais a esquerda(ordem normal). E ao reduzir um sub-termo atualiza-se o grafo com o resultado.

Os passos para redução de uma expressão são apresentado pelo Algoritmo 1. O algoritmo entra em um *loop*, se houver algum *redex* que não tenha sido avaliado ainda, o

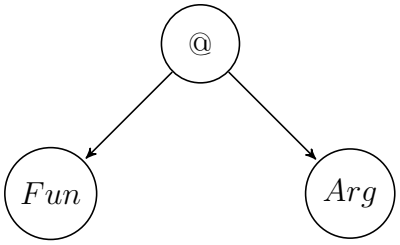


Figura 5: Grafo de uma chamada de método.

algoritmo seleciona o *redex* mais externo ao grafo, realiza sua redução, substitui o *redex* pelo resultado da redução e verifica novamente se existe mais algum *redex* que não foi avaliado. Caso exista *redex* que ainda não tenha sido avaliado o processo se repete, caso contrário o algoritmo é encerrado.

Algoritmo 1 Redução de grafos.

enquanto existe *redex* que não foi avaliado **faça**
  selecione o *redex* mais externo;
  reduza-o;
  substitua o *redex* pelo resultado de sua redução;
fim enquanto

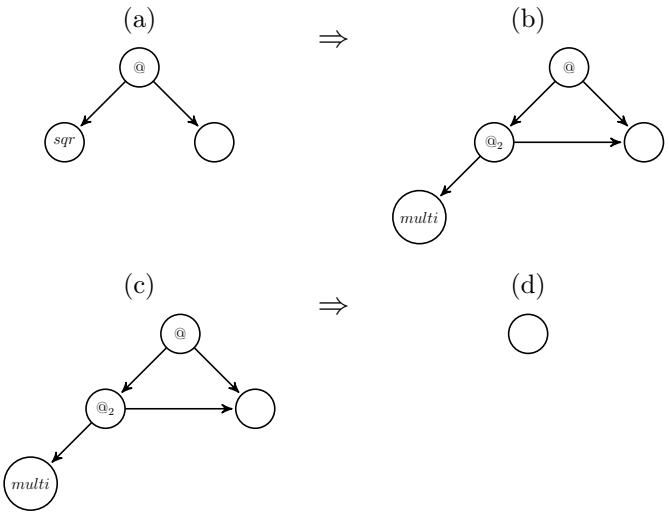


Figura 6: Regras utilizadas na redução de grafos.

Na Figura 6 são ilustradas as regras utilizadas para a redução do grafo da Figura 7. Neste exemplo só há duas regras: (i) a regra que transforma uma chamada da função `sqr` (6a) em uma chamada da função `multi` (6b); e (ii) a regra que transforma a chamada da função `multi` (6c) em um valor (6d) através da multiplicação dos valores de @ e $@_2$.

A Figura 7 mostra o processo de avaliação *chamada por necessidade* do programa mostrado na Figura 3. As expressões sublinhadas nos grafos são as expressões que serão

avaliadas (reescritas) utilizando as regras mostradas na Figura 6 e irão gerar o próximo grafo.

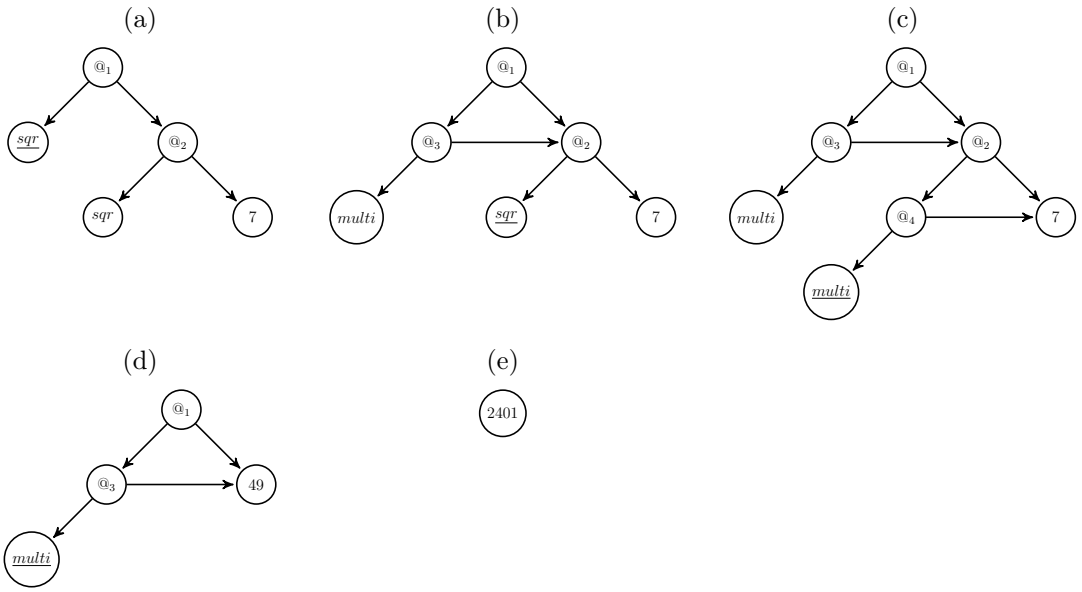


Figura 7: Avaliação chamada por necessidade(Redução de Grafos).

Na Figura 7a é visualizado o grafo que representa a chamada da função `main`. A raiz do grafo, $@_1$ representa a aplicação da função `main`, que possui arestas apontando para seus dois nós filhos, a chamada da função `sqr` a esquerda e a direita do argumento desta função, que é representado pelo subgrafo $@_2$. Por sua vez o subgrafo $@_2$ possui dois nós filhos, com arestas apontando para a chamada da função `sqr` a sua esquerda e a sua direita um nó com o valor 7.

A próxima expressão a ser avaliada neste grafo da Figura 7a é a expressão `sqr` a esquerda de $@_1$, que após sua reescrita, utilizando a regra 6a, será substituída no grafo original(Figura 7b) por um nó $@_3$ representando a aplicação da função `sqr`. São adicionados um nó `multi` representando a chamada da função `multi`, uma aresta de $@_3$ para `multi` e uma aresta de $@_3$ para o resultado da aplicação da expressão `sqr 7`($@_2$). Nesta última aresta adicionada ao grafo pode-se perceber que o resultado da avaliação de $@_2$ será utilizado por $@_1$ e $@_3$, ou seja, $@_2$ será avaliado apenas uma vez e o resultado de sua avaliação será compartilhado com os nós $@_1$ e $@_3$ evitando reavaliações desnecessárias.

No grafo da Figura 7b procura-se o próximo *redex* que pode ser avaliado, no caso a chamada da função `sqr`. Como foi feito no grafo anterior, substitui-se a ocorrência de `sqr` pela aplicação da função `sqr` (Regra 6a) com uma aresta apontando para um novo nó com a chamada da função `multi` e outra aresta apontando para o nó de valor 7. Percebe-se aqui novamente o compartilhamento do valor 7 pelos nós $@_2$ e $@_4$.

O próximo *redex* a ser avaliado no grafo da Figura 7c é a chamada da função `multi`, que após ser avaliado utilizando a regra 6c, retorna o valor 49. O nó $@_2$ é então substituído por um nó de valor 49, como pode ser visto na Figura 7d.

Finalmente, a função `multi` no grafo da Figura 7d é avaliada (Regra 6c) obtendo como resultado o valor 2401. O nó $@_1$ é substituído pelo resultado da avaliação, ficando apenas com um nó de valor 2401. Como não há mais nenhum *redex* que pode ser reduzido o algoritmo é encerrado.

Neste exemplo nota-se que a avaliação *chamada por necessidade* realiza menos avaliações que a estratégia *chamada por nome*, quando uma expressão ocorre mais de uma vez no corpo de uma função. Porém é necessário um trabalho extra para representar cada expressão na forma de grafos, realizar o gerenciamento correto das reduções e se tomar um cuidado especial para a coleta de lixo.

Normalmente linguagens imperativas adotam a avaliação *chamada por valor*, que realiza apenas uma avaliação de todas as expressões existentes no programa. Evitando assim o problema de duplicação de computações. O problema desta avaliação é que mesmo que uma expressão não seja utilizada no programa ela ainda será avaliada.

O processo redução de grafos se tornou uma ferramenta importante, utilizado pelas linguagens funcionais para implementação do lambda cálculo [59]. Permitindo a implementação de máquinas de redução tanto seqüenciais quanto paralelas. Várias máquinas abstratas foram propostas, trazendo melhorias a cada implementação.

Turner foi o primeiro a implementar uma máquina de redução de grafos, utilizando combinadores SKI [60]. Sua máquina executava com maior eficiência programas que exploravam funções de alta ordem[7], quando comparada com interpretadores convencionais da época [61].

Com o tempo, várias melhorias foram propostas para implementações de máquinas mais eficientes. Surgiram implementações com base no lambda cálculo, redução de combinadores ou redução de strings. Estas máquinas ficaram conhecidas como puras por selecionarem o próximo passo de redução dinamicamente [62].

Como alternativa às máquinas de redução puras, surgiram as máquinas de redução programada. Onde a escolha do próximo passo da redução é definido através da análise estática (compilação) da expressão original [63]. Por exemplo, em uma expressão condici-

[7]Funções de alta ordem são funções que utilizam outras funções como argumento de entrada, ou retorna uma função como saída.

onal, a condição é testada primeiro e depois um ou outro ramo do grafo é escolhido para ser avaliado, ao invés de se construir o grafo da expressão condicional e avaliar todo o grafo e depois se escolher o próximo ramo.

Este processo de redução programada trouxe novas possibilidades na busca de melhor eficiência para a implementação de máquinas de redução de grafos. Implementada utilizando redução programada, a máquina G criada por Johnsson e Augustsson [64, 65] tinha como objetivo a melhoria de performance das avaliações de aplicação de expressões. Posteriormente foi melhorada por Peyton Jones, em sua máquina STG (*Spineless Tagless G-machine*) [66]. Sendo que esta última é utilizada no GHC (*Haskell Glasgow Compiler*) que provavelmente é o compilador Haskell mais utilizado atualmente [67].

Outras máquinas de redução que podem ser citadas: GRIP [68], ALICE [69], máquina (v, G) [70], NORMA [71], dentre muitas outras [60, 63, 72]

2.4 Conclusões do Capítulo

Neste capítulo foi feita uma revisão da literatura e foram apresentados os principais conceitos relacionados a serviços web, tais como, composições de serviços, linguagens de composição e a definição da linguagem MiniBPEL proposta.

Um resumo sobre *big data* e suas principais ferramentas foi apresentado, demonstrando conceitos como o surgimento dos modelos paralelos, MapReduce, Hadoop e as ferramentas de composição existentes.

Por fim, foram expostos os conceitos por traz das máquinas de redução de grafos, detalhando seu funcionamento e mostrando onde são utilizadas hoje em dia. O próximo capítulo mostrará alguns trabalhos relacionados à proposta.

3 Trabalhos Relacionados

Neste capítulo serão apresentados alguns trabalhos relacionados. Na seção 3.1 será apresentada a máquina PEWS-AM que permite a criação de fluxos de trabalho para serviços web utilizando a linguagem PEWS. O sistema para criação de fluxos de trabalho para o Hadoop chamado Pony será apresentado na seção 3.2.

3.1 Máquina de Redução de Grafos PEWS-AM

Na tese de mestrado de Aguiar [73] foi proposta uma variante de PEWS, uma linguagem semelhante a BPEL, que permite a criação de fluxos de trabalho para composição de serviços web utilizando grafos. Uma máquina de redução de grafos foi proposta e implementada como prova de conceito. A esta máquina foi dado o nome de PEWS-AM.

A máquina implementada por Aguiar foi utilizada como modelo base para a criação da máquina MiniBPEL-AM proposta neste trabalho. Esta seção tratará de explicar o funcionamento da máquina PEWS-AM e no próximo capítulo (4) a máquina MiniBPEL-AM será detalhada.

A máquina PEWS-AM transforma uma composição escrita na linguagem PEWS em um grafo e permite a interpretação direta deste grafo, utilizando um conjunto de regras de redução propostas no trabalho.

A máquina PEWS-AM utiliza dois conjuntos de regras:

- regras de tradução $\mathcal{T}$, que traduzem programas escritos em PEWS para a sua representação na forma de grafos;
- regras de redução, que realizam transformações/reduções nos grafos.

A execução de um programa PEWS corresponde a aplicações sucessivas de regras de redução, a partir do grafo gerado pela função $\mathcal{T}$, até que o grafo não possa ser mais reduzido por nenhuma das regras de redução.

As regras de tradução e redução apresentadas nesta seção são baseadas no *draft* [74], onde as regras propostas foram revisadas e passaram por um refinamento em comparação com o trabalho original de Aguiar [73].

3.1.1 Construtores da Linguagem

O conjunto de regras a seguir define PEWS, uma linguagem que contém operações mais comuns para tratar composições de serviços web. A linguagem é definida como:

$$P ::= S(\overline{E}, \overline{X}) \mid P_1 || P_2 \mid P_1; P_2 \mid [E_1]P_1 + [E_2]P_2 \mid \text{if } [E_1]\ P_1 \mid unfolding\ P_1 \mid unfold$$

$$E ::= \text{id} \mid \text{n} \mid \text{t} \mid E_1 + E_2 \mid E_1 - E_2 \mid E_1 * E_2 \mid E_1/E_2 \mid -E_1 \mid E_1 \text{ and } E_2 \mid E_1 \text{ or } E_2$$
$$\mid \quad \text{not } E_1 \mid E_1 == E_2 \mid E_1 < E_2 \mid E_1 \leq E_2 \mid E_1 > E_2 \mid E_1 \geq E_2 \mid (\ E_1)$$

As regras de E descrevem a sintaxe de expressões aritméticas e booleanas. Permitindo o uso de identificadores, literais, operadores lógicos, aritméticos e relacionais mais comuns. Enquanto as regras de P definem a chamada de serviços web e regras de composição utilizadas pela linguagem, como:

- $S(\overline{E}, \overline{X})$: corresponde a chamada de um serviço web, onde S é o identificador do serviço, $\overline{E}$ é uma lista de expressões E que servirão como parâmetros de entrada para o serviço e $\overline{X}$ é uma lista de identificadores id que guardarão valores resultantes da execução do serviço;

- $P_1||P_2$: é uma composição em paralelo, onde os dois programas são executados concorrentemente e a composição só se encerra quando os dois programas terminarem suas execuções;

- $P_1; P_2$: define uma composição seqüencial, onde um programa P_2 só pode começar a executar depois do término de P_1;

- $[E_1]P_1 + [E_2]P_2$: define escolhas não determinísticas, onde E_1 e E_2 são expressões booleanas (guardas) que devem ser satisfeitas para a execução dos programas P_1 e P_2 respectivamente.

- if $[E_1]$ P_1: é um condicional, onde a guarda E_1 deve ser satisfeita para a execução do programa P_1

- $unfolding\ P_1$ e $unfold$: regras para repetição. Quando $unfolding\ P_1$ é avaliado, uma cópia P_1' é criada a partir de P_1 onde todas as ocorrências de $unfold$ dentro de P_1 são substituídas por $unfolding\ P_1$.

Para a implementação destas regras dois tipos de restrições são impostas: restrições de fluxo de controle e restrições de fluxo de dados.

As restrições de fluxo de controle são impostas diretamente pelos construtores da linguagem. Por exemplo, em comandos seqüenciais, há uma restrição de fluxo de controle entre um programa P_1 e P_2 indicando que o programa P_1 deve ser executado e concluído antes do início da execução de P_2.

Restrições de fluxo de dados garantem que cada variável vai estar associada a um valor antes de ser utilizada por alguma regra da linguagem, como uma expressão ou uma chamada de um serviço web. Por exemplo, dada a seguinte composição:

$$S1(X,Y)||S2(X,Z)||S3(Y,Z,W)$$

onde X é a entrada dos serviços web $S1$ e $S2$; Y e Z são saídas de $S1$ e $S2$ (respectivamente) e são entradas de $S3$, o qual produz W. Restrições de fluxo de dados entre os serviços $S1$ e $S3$ e entre os serviços $S2$ e $S3$, indicariam que $S3$ só pode ser executado uma vez que, os serviços $S1$ e $S2$ já tenham sido executados, e suas variáveis de saída Y e Z foram guardadas em memória para uso do serviço $S3$. Isto é, mesmo com os três serviços em paralelo, o serviço $S3$ só será executado depois da conclusão dos serviços $S1$ e $S2$ graças as restrições de fluxo de dados.

3.1.2 Máquina de Redução de Grafos PEWS-AM

A máquina permite a transformação de programas PEWS em grafos e a redução destes grafos através da aplicação de sucessivas regras de redução. A aplicação de regras de redução corresponde à avaliação do fluxo de trabalho de serviços web.

O processo de execução da máquina consiste de duas etapas: (i) a *transformação* de um programa PEWS em um grafo G utilizando regras de tradução $\mathcal{T}$; e (ii) *avaliação*, onde o grafo G é reduzido através da aplicação de sucessivas regras de redução.

No processo de tradução, os grafos gerados são representados por triplas $\langle V, A_c, A_d \rangle$, onde V representa o conjunto de vértices; A_c são arestas de controle, representando as

restrições de fluxo de controle e A_d são arestas de dependência de dados que representam as restrições de fluxo de dados.

Durante o processo de avaliação a máquina abstrata utiliza uma 4-tupla $\langle G, \rho, I, O \rangle$, onde G é o grafo; ρ é o ambiente de variáveis, que possui como função mapear nomes de variáveis para valores durante a execução de uma composição de serviços; I e O são *buffers* de entrada e saída. Cada *buffer* é uma lista de triplas (o, v, l) onde o é o nome de uma operação, v é o nome de um vértice e l é uma lista de valores. No *buffer* de saída O, l é uma lista de valores que devem ser utilizados como parâmetros para a chamada da operação o, e a chamada da operação deve retornar seu valor para vértice de retorno de operação v. Já No *buffer* de entrada I, l é uma lista de valores retornados pela execução de uma operação o, que deve ser repassada para o vértice v do grafo.

Os *buffers* de entrada e saída são manipulados por um componente externo à máquina de redução, que será chamado de gerenciador de serviços. Este componente tem como responsabilidade realizar as chamadas das operações por meio da utilização de informações do *buffer* de saída O da máquina de redução e preencher o *buffer* de entrada I da máquina com o resultado da execução das operações, como ilustrado na Figura 8.

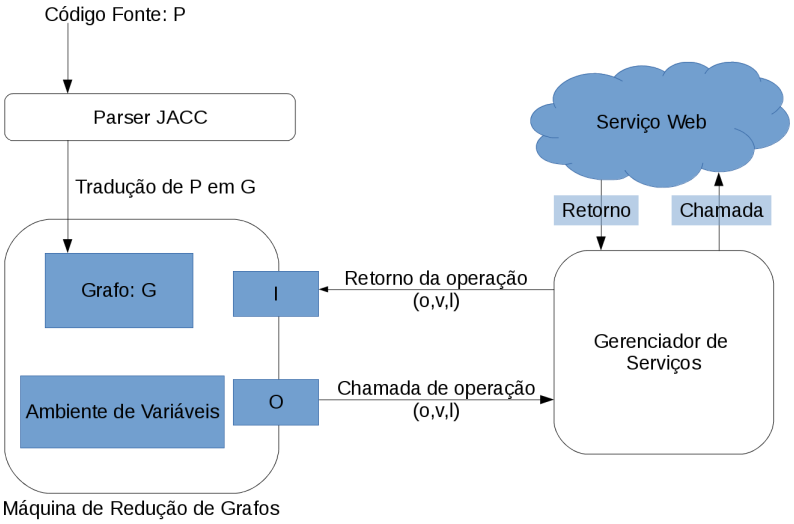


Figura 8: Comunicação da máquina de redução com o gerenciador de serviços.

No processo de avaliação a máquina de redução utiliza como configuração inicial uma 4-tupla $\langle \mathcal{T}[\![P]\!], \emptyset, \epsilon, \epsilon \rangle$, onde $\mathcal{T}[\![P]\!]$ é o grafo gerado a partir da transformação do programa P, utilizando as regras apresentadas na seção 3.1.3; $\emptyset$ indicando que o ambiente de variáveis começa sem variáveis; e com os *buffers* de entrada e saída vazios (ϵ).

As regras de tradução serão descritas na seção 3.1.3 e as regras de redução serão descritas na seção 3.1.4.

3.1.3 Regras de Tradução

As regras de tradução utilizadas pela máquina PEWS-AM serão definidas nesta subseção. A função $\mathcal{T}$ é definida para transformar programas P em grafos.

Como foi dito no final da seção 3.1.2 um grafo G é representado como uma tripla $\langle V, A_c, A_d \rangle$. A função $\mathcal{T}$ atribui um nome único a cada vértice do conjunto V. O conjunto de arestas de controle A_c representa as restrições de fluxo de controle impostas pelos construtores da linguagem. Arestas de dependência A_d representam dependências de dados, onde um nó do grafo precisa utilizar o valor atribuído a uma variável por outro nó do grafo.

Informações como o número de arestas que chegam a um nó do grafo são utilizadas pelas regras de tradução. Estas informações são disponibilizadas pelas 3 funções definidas a seguir:

- $indegree_c(v)$ que informa o número de arestas de controle que chegam ao nó v;
- $indegree_d(v)$ que informa o número de arestas de dependência de dados que chegam ao nó v;
- $indegree(v)$ que retorna o número total de arestas que chega ao nó v, isto é, $indegree(v) = indegree_c(v) + indegree_d(v)$.

3.1.3.1 Chamada de Operação

Dada uma chamada de operação $S(\overline{E}, \overline{X})$, na qual S representa o nome da operação de um serviço web, $\overline{E}$ representa uma lista de expressões de entrada e $\overline{X}$ uma lista de identificadores de saída, a regra de tradução $\mathcal{T}[\![S(\overline{E}, \overline{X}]\!])$ é definida a seguir:

$$\mathcal{T}[\![S(\overline{E}, \overline{X})]\!] = \langle \{w : S!\overline{E}, w' : S?\overline{X}\},\ \{w \mapsto w'\},\ \emptyset \rangle$$

A aplicação da regra da chamada de operação cria um grafo com 2 nodos w, w' e uma aresta de controle ligando os nodos. O nó w representa a chamada do serviço S utilizando a lista $\overline{E}$ de expressões como parâmetros de entrada. Enquanto o nó w' representa o retorno da operação. Quando a operação S terminar sua execução os valores de retorno serão associados à lista de variáveis de $\overline{X}$ no nó w'. A aresta de controle ligando os nós w e w' define que a chamada do serviço (w) deve ocorrer antes do retorno (w'), como ilustrado na Figura 9.

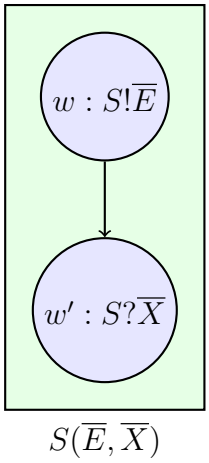


Figura 9: Grafo da chamada de operação.

3.1.3.2 Composição em Paralelo

Dados dois programas P_1 e P_2, em que cada um destes programas representa um programa PEWS válido, a regra de tradução $\mathcal{T}[\![P_1||P_2]\!]$ é definida por:

$$\begin{aligned}
&\mathcal{T}[\![P_1 \| P_2]\!] = \langle V' \cup V'',\ A'_c \cup A''_c,\ A'_d \cup A''_d \cup \Delta'_d \cup \Delta''_d \rangle \\
&\quad \textbf{where} \\
&\qquad \langle V', A'_c, A'_d \rangle = \mathcal{T}[\![P_1]\!], \quad \langle V'', A''_c, A''_d \rangle = \mathcal{T}[\![P_2]\!], \\
&\qquad \Delta'_d = crossDD(V', V''), \quad \Delta''_d = crossDD(V'', V')
\end{aligned}$$

onde a tradução de uma composição em paralelo é definida como a união dos vértices, arestas de controle e dependência de dados dos grafos que representam os programas P_1 e P_2.

O conjunto de arestas de dependência de dados é formado pela união das arestas de dependência de dados dos dois programas junto com os conjuntos Δ'_d e Δ''_d provenientes da função $crossDD$. Esta função retorna arestas de dependência de dados entre dois conjuntos de vértices, quando um nó de um conjunto de vértices precisar utilizar algum valor que vai ser resolvido por outro nó que ainda precisa ser avaliado. A definição desta função é dada a seguir:

$$\begin{aligned}
crossDD(V_1, V_2) = \ \{\ v_1 \mapsto v_2 \ \mid\ & v_1 : S_1?\overline{X} \in V_1 \wedge \\
& (v_2 : S_2!\overline{E} \in V_2 \wedge\ \exists x \in Vars(\overline{E}).\ x \in \overline{X} \\
& \vee\ v_2 : E \in V_2 \wedge \exists x \in Vars(E).\ x \in \overline{X})\ \ \}
\end{aligned}$$

Na Figura 10 é mostrado um exemplo do grafo resultante da tradução de dois programas P_1 e P_2. As arestas de dependência de dados são representadas como setas pontilhadas.

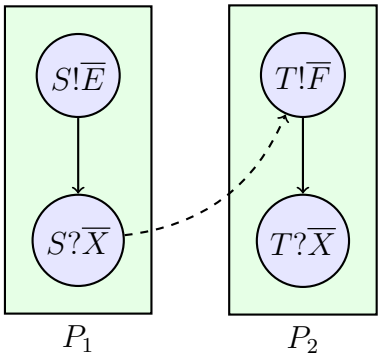


Figura 10: Grafo de uma composição em paralelo.

3.1.3.3 Composição Seqüencial

Dados dois programas PEWS válidos P_1 e P_2. A aplicação da regra de tradução $\mathcal{T}[\![P_1; P_2]\!]$ é definida por:

$$\mathcal{T}[\![P_1; P_2]\!] = \langle V' \cup V'',\ A'_c \cup A''_c \cup \Delta_c,\ A'_d \cup A''_d \cup \Delta_d \rangle$$
$$\textbf{where}$$
$$\langle V', A'_c, A'_d \rangle = \mathcal{T}[\![P_1]\!], \qquad \langle V'', A''_c, A''_d \rangle = \mathcal{T}[\![P_2]\!],$$
$$\Delta_c = \{ v_1 \mapsto v_2 \mid v_1 \in V',\ v_2 \in V'', \mathit{indegree}_c(v_2) = 0 \},$$
$$\Delta_d = \mathit{crossDD}(V', V'')$$

onde a tradução de uma composição sequencial é representada pela união dos vértices dos grafos dos dois programas. O conjunto de arestas de controle resultantes (A_c) é formado pela união das arestas de controle A'_c e A''_c mais Δ_c, onde Δ_c é o conjunto de arestas que partem de todos os vértices de P_1 com destino a todos os vértices de P_2 que possuem $\mathit{indegree}_c = 0$.

O conjunto de arestas de dependência de dados A_d é definido pela união das arestas A'_d, A''_d e Δ_d, onde Δ_d representa o conjunto de arestas de dependência das variáveis que foi resolvido no programa P_1 e é utilizado em P_2. Um exemplo disto é dado na Figura 11.

3.1.3.4 Escolha Não Determinística

O trecho de código PEWS "$[E_1]P_1{+}[E_2]P_2$", no qual P_1 e P_2 são programas PEWS e as expressões E_1 e E_2 são guardas dos respectivos programas, representa uma escolha

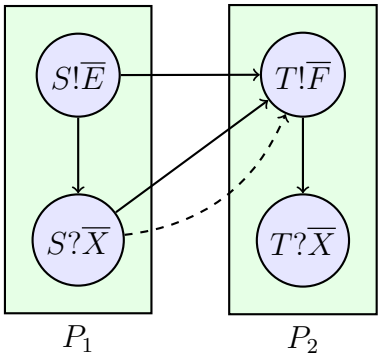


Figura 11: Grafo de uma composição seqüencial.

não determinística.

O grafo resultante da aplicação da regra de tradução de uma escolha não determinística possui um vértice adicional + que contém arestas de controle para as expressões a serem avaliadas. Cada expressão é contida em um nodo que possui arestas de controle apontando para seus respectivos programas.

A regra para tradução da escolha não determinística $\mathcal{T}[\![[E_1]P_1{+}[E_2]P_2]\!]$ define que o conjunto de vértices do grafo gerado é composto pela união dos vértices V', V'' e $\mathcal{V}$, onde $\mathcal{V}$ representa os vértices + e um vértice para cada guarda.

$$\mathcal{T}[\![[E_1]P_1{+}[E_2]P_2]\!] = \langle V' \cup V'' \cup \mathcal{V},\ A'_c \cup A''_c\ \cup \Delta_c, A'_d \cup A''_d \rangle$$

where

$$\langle V',\ A'_c,\ A'_d \rangle = \mathcal{T}[\![P_1]\!], \quad \langle V'',\ A''_c,\ A''_d \rangle = \mathcal{T}[\![P_2]\!],$$
$$\mathcal{V} = \{v : +, v_1 : E_1, v_2 : E_2\},$$
$$\Delta_c = \{v \mapsto v_1, v \mapsto v_2\} \cup \Delta'_c \cup \Delta''_c,$$
$$\Delta'_c = \{v_1 \mapsto w \mid w \in V', indegree_c(w) = 0\},$$
$$\Delta''_c = \{v_2 \mapsto w \mid w \in V'', indegree_c(w) = 0\}$$

O conjunto de arestas de controle é formado pelas arestas de controle de A'_c, A''_c e Δ_c, onde Δ_c é o conjunto de arestas que partem do vértice + a cada uma das expressões em união com o conjunto de arestas que partem de cada uma das expressões para os vértices com $indegree_c = 0$ de seus respectivos programas.

Como esta regra não cria arestas de dependência de dados, fora as criadas por P_1 e P_2, o conjunto A_d fica definido como a união das arestas de dependência de dados A'_d e A''_d.

A Figura 12 mostra o grafo resultante da execução da regra de tradução para escolha

não determinística $\mathcal{T}[\![[E_1]((A;B)||(C;D))+[E_2]((E;F)||(G;H))]\!]$.

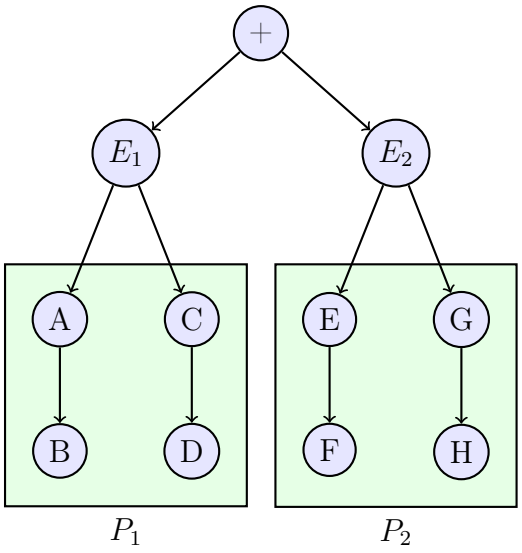


Figura 12: Grafo de uma escolha não determinística.

3.1.3.5 Condicional

A regra de tradução de expressão condicional é definida de forma similar à escolha não determinística, com a diferença de que na regra de tradução condicional há apenas uma expressão a ser avaliada. A segunda expressão, por não existir é definida como um nodo **false** sem nodos filhos.

Esta regra define que o conjunto de vértices do grafo gerado é formado pelos vértices V do programa P em união com um vértice $+$, um vértice representando a expressão e um vértice representando o valor **false**.

$$
\begin{aligned}
&\mathcal{T}[\![\textbf{if } E \textbf{ then } P]\!] = \langle V \cup \mathcal{V},\ A_c \cup \Delta_c,\ A_d \rangle \\
&\quad \textbf{where} \\
&\qquad \langle V,\ A_c,\ A_d \rangle = \mathcal{T}[\![P]\!], \\
&\qquad \mathcal{V} = \{v : +, v_1 : E, v_2 : \textbf{false}\}, \\
&\qquad \Delta_c = \{v \mapsto v_1,\ v \mapsto v_2\} \cup \Delta'_c, \\
&\qquad \Delta'_c = \{v_1 \mapsto w \mid w \in V, indegree_c(w) = 0\}
\end{aligned}
$$

As arestas de controle são formadas pelas arestas A_c do programa P em união com Δ_c, onde Δ_c é o conjunto com as arestas de controle que ligam o vértice $+$ aos vértices que representam a expressão a ser avaliada e o valor **false** em união com as arestas que partem da expressão com destino a todos os nós com $indegree_c = 0$ do programa P.

Assim como a regra para escolha não determinística, esta regra não cria arestas de dependência de dados, portanto estas arestas são representadas apenas pelas arestas A_d provenientes de P.

O grafo da Figura 13 é o resultado da aplicação da regra de tradução para o condicional $\mathcal{T}[\![\textbf{if } E \textbf{ then } ((A;B)||(C;D))]\!]$.

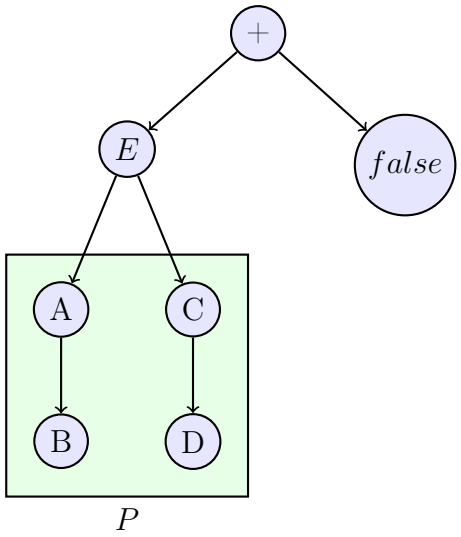


Figura 13: Grafo de um condicional.

3.1.3.6 Unfolding

O trecho de código "$unfolding\ P$", no qual P é um programa PEWS, representa uma estrutura de repetição *Unfolding*.

Um nodo μ é criado e um conjunto de arestas ligando este nodo aos vértices raizes do programa P.

O conjunto de vértices do grafo gerado pela aplicação da regra de tradução $\mathcal{T}[\![unfolding\ P]\!]$ é formado pelos vértices V vindos da tradução de P em união com o vértice μ. Uma cópia do programa P é anotada no vértice μ para ser utilizada posteriormente no processo de redução para criar novas iterações. Para indicar esta anotação o vértice μ será escrito como $\mu.P$.

$$\begin{aligned}
&\mathcal{T}[\![\textbf{unfolding } P]\!] = \langle V \cup \{v : \mu.P\},\ A_c \cup \Delta_c,\ A_d \rangle \\
&\quad \textbf{where} \\
&\qquad \langle V,\ A_c,\ A_d \rangle = \mathcal{T}[\![P]\!], \\
&\qquad \Delta_c = \{v \mapsto w \mid w \in V, indegree_c(w) = 0\}
\end{aligned}$$

As arestas de controle são formadas pela união das arestas A_c e as arestas que ligam o nó μ aos nós que possuem $indegree_c = 0$ em P. As arestas de dependência de dados são formadas apenas pelas arestas A_d vindas de P.

A regra de tradução $\mathcal{T}[\![\textbf{unfolding}\ ((A;B)||(C;D))]\!]$ cria o grafo apresentado na Figura 14.

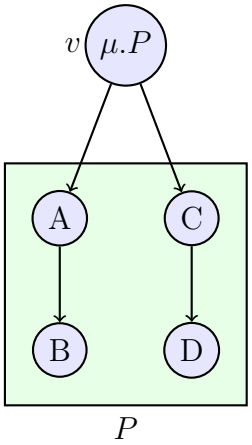


Figura 14: Grafo do unfolding.

3.1.3.7 Unfold

A regra de tradução $\mathcal{T}[\![unfold]\!]$ é a mais simples, criando um grafo com apenas um nodo *unfold*. Este tipo de nodo faz parte de estruturas de repetição *unfolding*. Assumindo que para cada vértice *unfolding* sempre existirá pelo menos um vértice *unfold* associado.

$$\mathcal{T}[\![\textbf{unfold}]\!] = \langle \{v : unfold\},\ \emptyset,\ \emptyset \rangle$$

O grafo gerado pela aplicação da regra *unfold* possui apenas um vértice *unfold*. Arestas de controle ou de dependência de dados não são geradas, deixando os respectivos conjuntos com valor vazio($\emptyset$). A Figura 15 representa o grafo gerado pela aplicação da regra de tradução $\mathcal{T}[\![\textbf{unfold}]\!]$.

Figura 15: Grafo do unfold.

3.1.4 Regras de Redução

Nesta subseção as regras de redução utilizadas pela máquina de redução são descritas. O processo de avaliação ocorre sempre que algum determinado vértice se torna candidato a avaliação. Para que um vértice se torne candidato a avaliação ele precisa atender as restrições de cada regra e ser um dos seguintes vértices: $S!\overline{E}$ (chamada de operações), $S?\overline{X}$ (retorno de operações), $+$ (usado em escolhas não determinísticas e em condicionais) e μ (usado pelo *unfolding*).

O processo de execução da máquina PEWS é definido pela relação de avaliação:

$$\langle G, \rho, I, O\rangle \rightarrow \langle G', \rho', I', O'\rangle$$

onde a partir de um estado inicial $\langle G, \rho, I, O\rangle$, após um passo na avaliação, utilizando as regras desta seção chega-se ao estado $\langle G', \rho', I', O'\rangle$. Para um conjunto H, escreve-se $H[x]$ com o significado $H \cup \{x\}$.

As regras de redução são apresentadas a seguir:

3.1.4.1 Chamada de Operação

$$\frac{\begin{array}{c} \mathit{indegree}(w) = 0, \qquad \overline{a} = \mathit{Eval}_\rho(\overline{E}), \\ A'_c = A_c - \{w \mapsto v \mid v \in V\}, \\ G = \langle V[w : S!\overline{E},\ w' : S?\overline{X}], A_c[w \mapsto w'], A_d\rangle \end{array}}{\langle G, \rho, I, O\rangle \ \rightarrow\ \langle\langle V[w' : S?\overline{X}], A'_c, A_d\rangle, \rho, I, O[(S, w', \overline{a})]\rangle}$$

A regra de redução para a chamada de operações pode ser executada quando existe no grafo um vértice w do tipo $S!\overline{E}$ com $\mathit{indegree} = 0$, um vértice w' do tipo $S?\overline{X}$ e uma aresta de controle partindo de w com destino a w'. Isto é, existe um vértice w pronto para ser avaliado que representa uma chamada de operação, como ilustrado no lado esquerdo da Figura 16.

O processo de avaliação consiste nos seguintes passos: *a*) escrever no *buffer* de saída O uma tripla $(S, w', \overline{a})$ com o nome do serviço S a ser chamado, o vértice w' ao qual a máquina deve retornar o resultado e uma lista $\overline{a}$ de valores avaliados a partir da lista de expressões $\overline{E}$ que serão utilizados como parâmetros de entrada do serviço S; *b*) remover todas as arestas de controle que partem de w e *c*) remover o vértice w do grafo.

O resultado da avaliação desta regra é mostrado na Figura 16, ficando o vértice w' e uma entrada no *buffer* de saída.

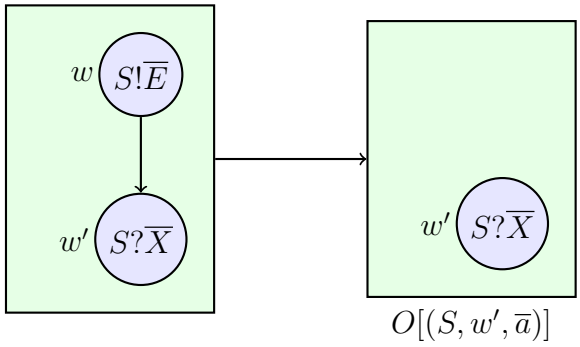


Figura 16: Grafo da chamada de uma operação.

Com a entrada no *buffer* de saída O um componente externo irá fazer uma chamada ao serviço S usando a lista $\overline{a}$ como parâmetro. Quando o serviço chamado retornar seu resultado, o componente externo irá apagar a tripla do *buffer* de saída O e escrever uma tripla no *buffer* de entrada I da máquina.

Uma tripla $(S, w', \overline{r})$ escrita no *buffer* de entrada é formada utilizando o nome do serviço S que foi chamado, o vértice w' ao qual o resultado deve ser retornado e uma lista $\overline{r}$ com os valores retornados pela chamada do serviço.

3.1.4.2 Retorno de Operação

$$\frac{\begin{array}{c} \mathit{indegree}(w) = 0, \qquad \overline{r} = (v_1, \ldots, v_k), \qquad \overline{X} = (x_1, \ldots, x_k), \\ \rho' = \rho \cup \{(x_i, v_i) \mid 1 \leq i \leq k\}, \\ A'_c = A_c - \{w \mapsto v \mid v \in V\}, \qquad A'_d = A_d - \{w \mapsto v \mid w' \in V\} \end{array}}{\langle\langle V[w : S?\overline{X}], A_c, A_d\rangle, \rho, I[(S, w, \overline{r})], O\rangle \;\rightarrow\; \langle\langle V, A'_c, A'_d\rangle, \rho', I, O\rangle}$$

A regra para retorno de operação espera que o grafo possua um vértice w do tipo $S?\overline{X}$, que possua $\mathit{indegree} = 0$ e uma entrada no *buffer* de entrada $I[(S, w, \overline{r})]$ que indique que existe um resultado de chamada de operação para o vértice w. Ou seja, existe um vértice de retorno de operação pronto para ser avaliado.

A avaliação associa cada um dos valores retornados $\overline{r}$ com cada uma das variáveis $\overline{X}$ no ambiente ρ. A tripla $(S, w, \overline{r})$ é removida do *buffer* de entrada I. O vértice w é removido do grafo, bem como todas as arestas de controle e de dependência de dados que partem de w. O resultado da avaliação desta regra pode ser visto na Figura 17.

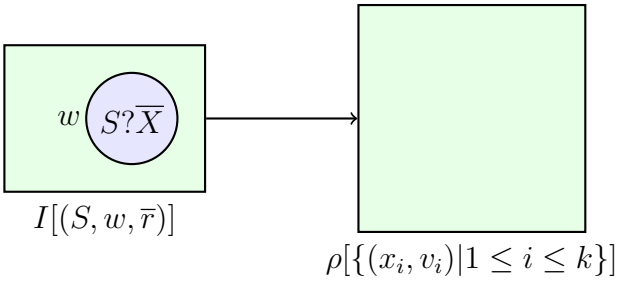


Figura 17: Grafo do retorno de uma operação.

3.1.4.3 Escolha Não Determinística

Quando o próximo vértice a ser avaliado é um vértice +, sabe-se que uma escolha não determinística será tratada.

Durante a fase de aplicação das regras de tradução as escolhas não determinísticas e os condicionais criam grafos semelhantes. Escolhas não determinísticas utilizam duas expressões guardas (Figura 18 esquerda), enquanto que condicionais utilizam apenas uma expressão guarda (Figura 18 direita), deixando a segunda expressão com valor **false** e sem vértices filhos.

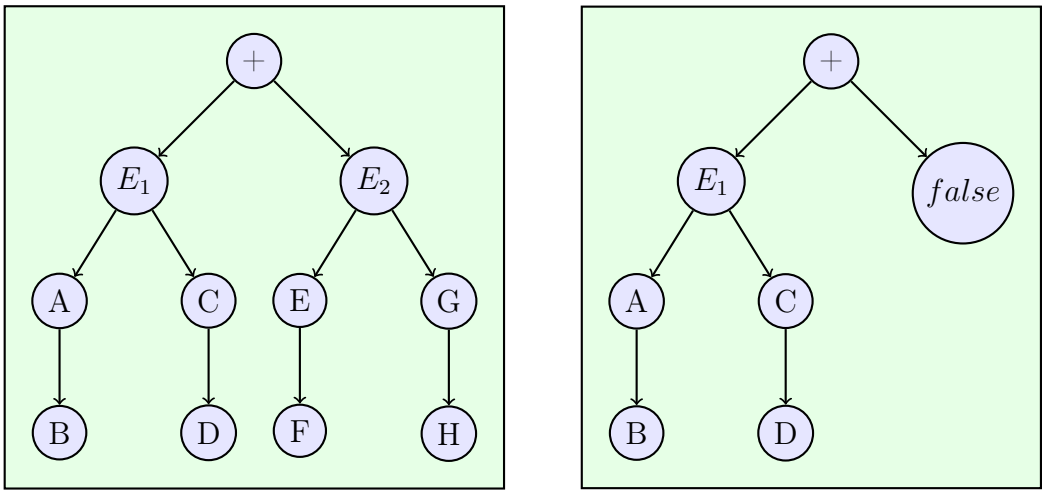


Figura 18: Grafo de uma escolha não determinística (esquerda) e de um condicional (direita).

Para o caso de escolha não determinística são utilizadas duas regras de avaliação: (i) uma para quando uma das expressões avaliadas possui valor **true**; e (ii) as duas expressões avaliadas possuem valor **false**. Quando existem duas expressões com valor verdadeiro a primeira regra é utilizada e seleciona-se uma das expressões. Sendo que esta escolha é não determinística, isto é, a especificação não define qual das opções será adotada. A escolha fica, então, a cargo da implementação.

A regra a seguir é usada quando uma das expressões da escolha não determinística avaliada possui valor **true**. Neste caso não é necessário verificar as outras expressões da escolha não determinística, removendo assim: o subgrafo filho da outra expressão; o vértice +; e os vértices das expressões, como ilustrado na Figura 19.

$$\frac{\begin{array}{c} indegree_c(v) = 0, \qquad indegree_d(w) = 0, \qquad Eval_\rho(E) = \textbf{true}, \\ V' = \{u \in V \mid w' \mapsto_c^* u \wedge w \not\mapsto_c^* u\}, \qquad G = \langle V, A_c, A_d\rangle \setminus (V' \cup \{v, w\}) \end{array}}{\langle\langle V[v : +, w : E, w' : E'], A_c[v \mapsto w, v \mapsto w'], A_d\rangle, \rho, I, O\rangle \;\rightarrow\; \langle G, \rho, I, O\rangle}$$

Para a remoção do subgrafo filho da outra expressão seleciona-se para remoção o conjunto de vértices atingíveis a partir do vértice w' e que não são alcançáveis a partir de w. Na Figura 19 os vértices V' selecionados para remoção da expressão não escolhida seriam E, F, G, H e o próprio vértice $w' : E_2$.

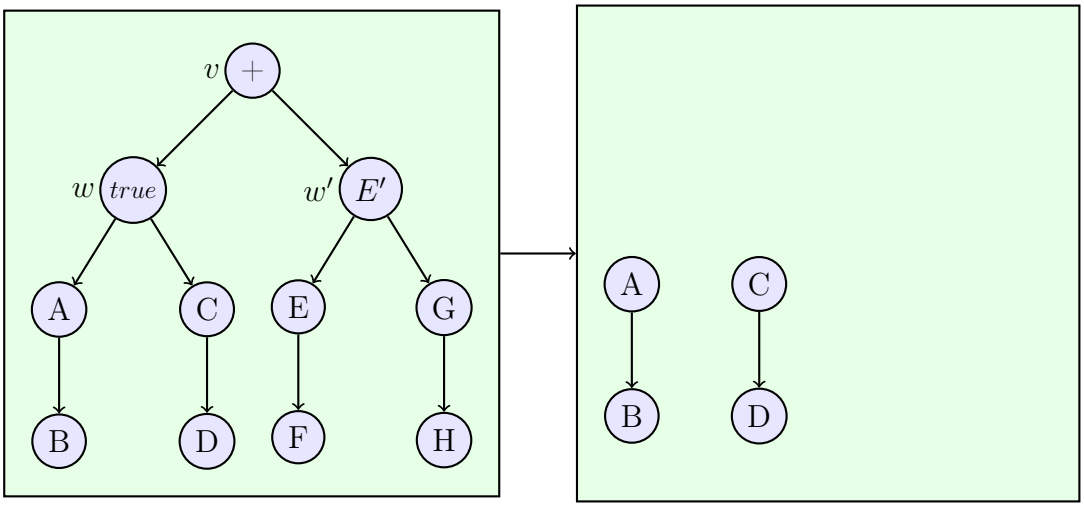


Figura 19: Grafo de uma escolha não determinística com uma expressão verdadeira.

A regra da escolha não determinística para o caso das duas expressões terem sido avaliadas para **false** é apresentada a seguir. Neste caso remove-se do grafo: o vértice +; os vértices das expressões falsas e os subgrafos filhos de cada uma das expressões.

$$\frac{\begin{array}{c} indegree_c(v) = 0, \qquad indegree_d(w) = 0, \qquad indegree_d(w') = 0, \\ Eval_\rho(E) = \textbf{false}, \qquad Eval_\rho(E') = \textbf{false}, \\ V' = \{u \in V \mid w \mapsto_c^* u \wedge w' \not\mapsto_c^* u\}, \qquad V'' = \{u \in V \mid w' \mapsto_c^* u \wedge w \not\mapsto_c^* u\}, \\ G = \langle V, A_c, A_d\rangle \setminus (V' \cup V'' \cup \{v\}) \end{array}}{\langle\langle V[v : +, w : E, w' : E'], A_c[v \mapsto w, v \mapsto w'], A_d\rangle, \rho, I, O\rangle \;\rightarrow\; \langle G, \rho, I, O\rangle}$$

Para a remoção dos subgrafos filhos de cada expressão remove-se do grafo o conjunto de vértices V' e V'', onde V' possui os vértices atingíveis a partir de w que não são alcançáveis pelo vértice w' correspondendo ao subgrafo com raiz em w e o conjunto de vértices V'' que possui os vértices atingíveis a partir de w' mas que não são alcançáveis pelo vértice w, correspondendo ao subgrafo com raiz em w'.

A Figura 20 ilustra a aplicação desta regra em um grafo com duas expressões falsas.

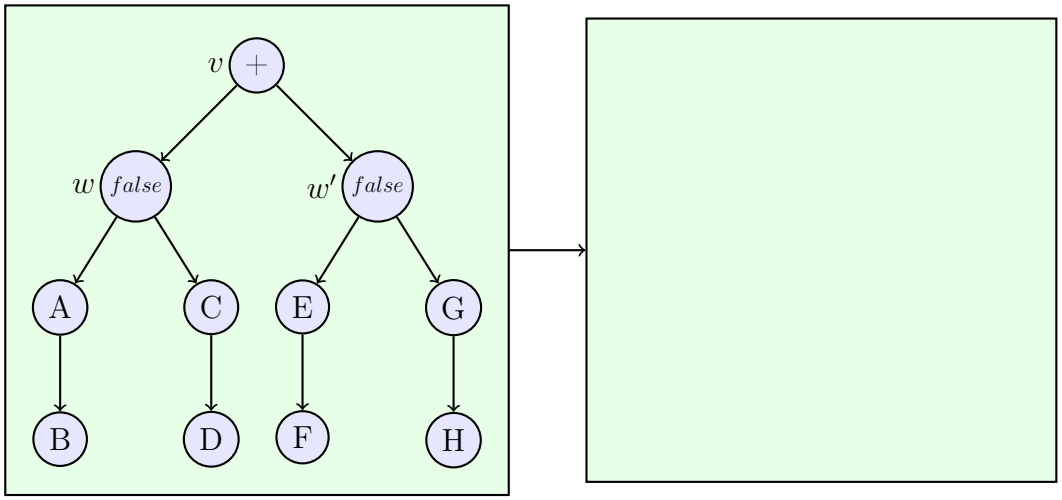


Figura 20: Grafo de uma escolha não determinística com todas as expressões falsas.

3.1.4.4 Unfolding

Esta regra deve ser aplicada quando o próximo vértice a ser avaliado é um vértice $\mu.P$, indicando que trata-se de uma estrutura de repetição $unfolding$. Quando esta regra é aplicada, todas as ocorrências do vértice $unfold$ são expandidas para uma cópia do subgrafo com raiz em $\mu.P$.

Nesta regra W é o conjunto de vértices w representando todos os vértices $unfold$ atingíveis a partir do vértice v ($\mu.P$) utilizando as arestas de controle que não são atingíveis por outros vértices $\mu.P$ dentro de $\mathcal{T}[\![P]\!]$. G^w é uma cópia do grafo v, com novos vértices e novas arestas de controle e dependência, como é demonstrado a seguir.

Dado $\mathcal{T}[\![P]\!] = \langle V^P, A_c^P, A_d^P \rangle$, G^w é definido como:

$$G^w = \langle V^P \cup \{w : \mu.P\}, A_c^P \cup \{w \mapsto r | r \in roots(V^P)\}, A_d^P \rangle$$

onde $roots(G)$ representa a função que retorna o conjunto de vértices raízes de um grafo $G = \langle V, A_c, A_d \rangle$ e é definida como $roots(G) = \{v \in V | indegree_c(v) = 0\}$.

O novo grafo G' é formado pela substituição de todos os vértices $unfold$ do conjunto W por cópias G^w do grafo com raiz em $\mu.P$. Esta substituição corresponde a criação de uma nova iteração do *loop unfolding*, como ilustra a Figura 21.

$$\frac{\begin{array}{c} indegree_c(v) = 0 \\ W = \{w \mid w : unfold \in V \wedge v \mapsto_c^* w \wedge \nexists u : \mu.P'.\, v \mapsto_c^* u \wedge u \mapsto_c^* w\} \\ G' = \langle V - W, A_c, A_d \rangle \cup \bigcup_{w \in W} G^w \end{array}}{\langle \langle V[v : \mu.P], A_c, A_d \rangle, \rho, I, O \rangle \rightarrow \langle G', \rho, I, O \rangle}$$

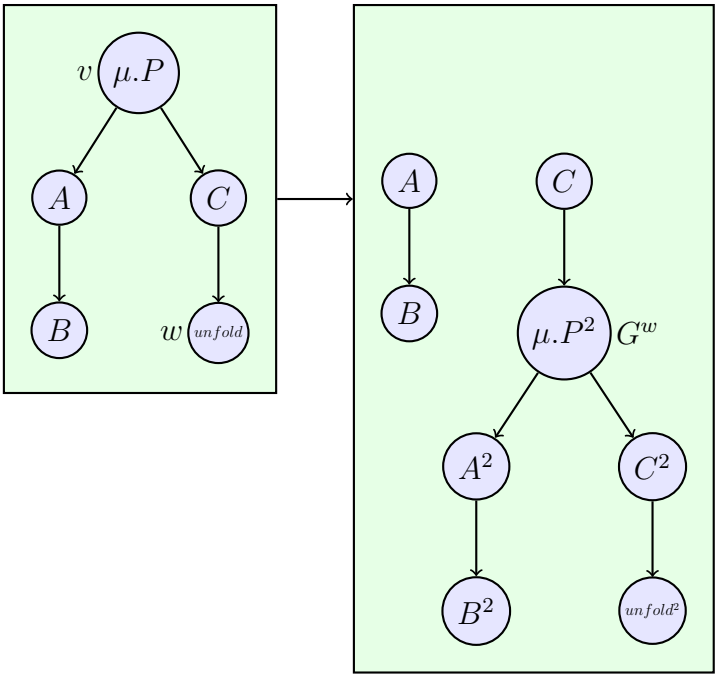


Figura 21: Grafo de unfolding.

3.2 Pony

Pony [53] é um sistema para gerenciamento de fluxos de trabalho para Hadoop baseado em BPEL. Este sistema se destaca por utilizar um agendador de fluxos de trabalho. O agendador utilizado por Pony permite priorizar a execução de operações pertencentes a fluxos de trabalho dentro do Hadoop.

Os fluxos de trabalho utilizados por Pony são modelados como grafos direcionados (ou seja, grafos de fluxos de trabalho) e são posteriormente mapeados para processos BPEL.

Como BPEL realiza orquestrações utilizando apenas serviços web, um serviço *wrapper* (utilizado para encapsular tarefas Hadoop) foi proposto para converter tarefas do Hadoop

para serviços web. Para cada tipo de tarefa do Hadoop há um serviço *wrapper* correspondente que fica responsável por realizar envio de dados, execução e obtenção de resposta da tarefa do Hadoop.

O agendador de tarefas do Hadoop tem como função alocar recursos (*containers*[1]) para tarefas em execução. Por padrão, Hadoop utiliza o agendador de tarefas FIFO (*First In First Out*) , que aloca recursos para tarefas na ordem em que elas chegam ao *cluster*. Dois outros agendadores de tarefa que podem ser utilizados são *Capacity* e *Fair* [75].

No entanto, estes agendadores não tem condições de determinar se uma tarefa pertence ou não a um fluxo de trabalho. Desta forma, quando fluxos de trabalho são enviados para execução, suas tarefas concorrem por recursos junto com tarefas que não fazem parte do fluxo de trabalho, fazendo com que tarefas pertencentes a um fluxo de trabalho esperem por recursos para que possam ser executadas. A espera por recursos acaba retardando a execução dos fluxos de trabalho.

Para resolver este problema, Pony propôs a criação de um agendador de tarefas Hadoop que trabalha de forma colaborativa com BPEL para permitir a execução de fluxos de trabalho de forma eficiente. Desta forma, o Hadoop pode dar prioridade à execução de tarefas pertencentes à fluxos de trabalho, acelerando assim sua execução.

Como pode ser visto na Figura 22, Pony é composto por um "*Hadoop Workflow Designer*" que fica responsável por criar fluxos de trabalho de forma visual e converter estes grafos para BPEL para que o fluxo de trabalho possa ser executado pelo "*Hadoop Workflow Execution Engine*". O "*Execution Engine*" cuidará de executar o fluxo de trabalho BPEL e quando uma tarefa Hadoop for executada, o serviço *wrapper* correspondente irá realizar comunicação com o *cluster* Hadoop, executar sua tarefa e retornar resultados para o "*Execution Engine*". Os serviços *wrapper* se comunicam com o "Pony Job Scheduler" (o agendador de tarefas), e este último trata de dar prioridades às tarefas Hadoop que pertencem a fluxos de trabalho, permitindo que o fluxo seja executado mais rapidamente.

- permitir a chamada de operações de multiplos dominios a partir do cluster hadoop -

3.3 Conclusões do Capítulo

Neste capítulo foram mostrados trabalhos relacionados à proposta desta dissertação. Na seção 3.1 foi apresentada a máquina de redução de grafos PEWS-AM que permite a cri-

[1]Container é a unidade computacional mínima dentro do Hadoop, que equivale aos recursos, tais como CPU e memória, necessários para executar uma tarefa em um nodo do *cluster*.

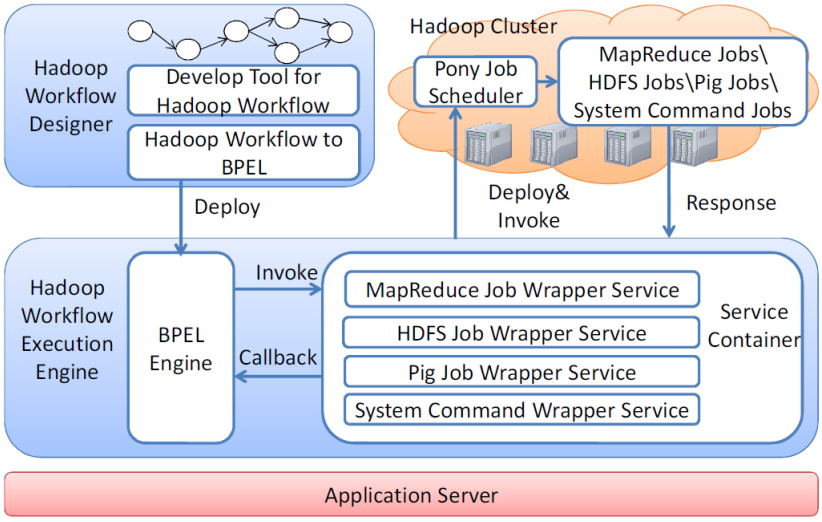


Figura 22: Arquitetura de Pony [53].

ação de composições de serviços web utilizando a linguagem PEWS. Foram apresentados os construtores da linguagem; a arquitetura utilizada pela máquina; o conjunto de regras de tradução que transforma código PEWS em grafos e o conjunto de regras de redução que através de sucessivas aplicações transformam os grafos e executam as chamadas de serviços web da composição.

A experiência obtida na definição e implementação da máquina PEWS-AM foi utilizada neste trabalho, permitindo propor uma máquina de redução de grafos extensível e com uma estrutura de grafos mais simplificada.

A seção 3.2 tratou de comentar sobre a ferramenta Pony, que realiza fluxos de trabalho Hadoop, utilizando BPEL para executar os fluxos de trabalho em conjunto com um agendador de tarefas. Sendo este o trabalho mais próximo relacionado à proposta.

No próximo capítulo serão apresentados os principais conceitos relacionados à máquina MiniBPEL-AM.

4 A Máquina de Redução de Grafos MiniBPEL-AM

Este capítulo apresenta de forma geral a máquina MiniBPEL-AM. Esta máquina permite a execução de fluxos de trabalho com operações mistas, isto é, operações como chamadas de serviços web, tarefas do hadoop, funções definidas pelo usuário ou qualquer outra operação que o usuário queira chamar, utilizando para isto construtores similares aos utilizados em linguagens que permitem composições de serviços web baseadas em *workflows* (fluxos de trabalho), tais como BPEL e PEWS.

A máquina PEWS-AM foi utilizada como base para a criação da máquina MiniBPEL-AM. A presente proposta representa uma evolução da máquina PEWS-AM. A nossa proposta se diferencia de PEWS-AM em dois aspectos principais: (1) O grafo gerado pela função de tradução foi redefinido e simplificado e (2) foram adicionadas características de extensibilidade que permitem o uso da máquina em conjunto com outras tecnologias. Em particular, nosso trabalho demonstra essa extensibilidade mediante a adição de uma interface para o processamento de grandes volumes de dados usando Hadoop.

A máquina MiniBPEL-AM permite a execução de fluxos de trabalho, assim como o sistema de gerenciamento de fluxos de trabalho Pony apresentado na seção 3.2 do capítulo de trabalhos relacionados 3. O que diferencia as duas ferramentas é o foco: Pony tem como foco executar múltiplos fluxos de trabalho de forma eficiente, enquanto MiniBPEL-AM procura permitir que o usuário possa criar novas operações para executar em um mesmo fluxo de trabalho.

A sintaxe da linguagem MiniBPEL é detalhada na seção 4.1; a arquitetura da máquina que será vista na seção 4.2; o conjunto de regras de tradução e redução, que são utilizados pela máquina serão vistos nas seções 4.3 e 4.4 respectivamente. Será apresentado na seção 4.5 um exemplo de composição utilizando a linguagem MiniBPEL e por fim as conclusões são apresentadas na seção 4.6.

4.1 Construtores da Linguagem

Os construtores da linguagem MiniBPEL são quase todos os mesmos utilizados pela máquina PEWS-AM, com exceção do construtor de repetição, onde foi removido o *unfolding/unfold* e foi inserido *while*.

As regras da gramática a seguir definem a linguagem MiniBPEL:

$$E ::= \text{id} \mid \text{n} \mid \text{t} \mid \text{s} \mid E_1 + E_2 \mid E_1 - E_2 \mid E_1 * E_2 \mid E_1/E_2 \mid -E_1 \mid E_1 \text{ and } E_2 \mid E_1 \text{ or } E_2 \mid \text{not } E_1 \mid E_1 == E_2 \mid E_1 < E_2 \mid E_1 \leq E_2 \mid E_1 > E_2 \mid E_1 \geq E_2 \mid (\ E_1)$$

$$P ::= S(\overline{E};\overline{X}) \mid P_1||P_2 \mid P_1;P_2 \mid [E_1]P_1 + [E_2]P_2 \mid \text{if } E \text{ then } P \mid \textit{while } E \textit{ do } P$$

Assim como na máquina PEWS-AM o símbolo não terminal E descreve expressões aritméticas e booleanas, permitindo o uso de identificadores, literais, *strings*, operadores lógicos, aritméticos e relacionais mais comuns. Da mesma forma o símbolo não terminal P define as chamadas de serviços web, chamadas de tarefas de *big data* ou outras operações e regras de composição utilizadas na linguagem. A maioria das regras P se comporta de forma similar ao que foi definido para a máquina PEWS-AM, com exceção das seguintes regras:

- $S(\overline{E};\overline{X})$: corresponde a chamada de uma operação, onde S é o identificador da operação, $\overline{E}$ é uma lista de expressões E que servirão como parâmetros de entrada para a operação e $\overline{X}$ é uma lista de identificadores *id* que guardarão valores resultantes da execução da operação. A operação pode ser um serviço web, uma tarefa de *big data* ou uma operação definida pelo usuário. A diferença em relação a PEWS-AM é que as chamadas podem, agora, além de invocar serviços web, invocar operações de *big data* ou operações definidas pelo usuário. Esta regra separa as listas $\overline{E}$ e $\overline{X}$ utilizando ponto e virgula (;), o que difere da regra PEWS que utiliza uma virgula para separar estas listas. Esta mudança torna mais claro onde a lista de parametros termina e onde a lista de identificadores inicia;
- *while E do P*: regra para repetição. Quando esta regra é avaliada o trecho de programa é reescrito como $if\ E\ then\ (P;\ while\ E\ do\ P)$, correspondendo a um novo *loop* de repetição do *while*.

Na máquina MiniBPEL-AM são utilizadas apenas restrições de fluxo de controle. Estas restrições são impostas pelos construtores da linguagem. Por exemplo, caso exista

uma restrição de controle de um programa P_1 até um programa P_2 isto indica que o programa P_1 deve ser executado antes do programa P_2.

As operações de *big data* que serão implementadas neste trabalho são:

- $mapReduce(e; x)$, que espera como parâmetro de entrada e, um arquivo do tipo .jar, que possua as funções *Map* e *Reduce* implementadas;
- $hdfs(e; x)$, que recebe como parâmetro de entrada um conjunto de caracteres correspondendo a um comando hdfs;
- $pig(e; x)$, que espera como parâmetro de entrada um *script* pig;
- $hive(e; x)$, que aceita como parâmetro de entrada um *script* hive.

Todas estas operações de *big data* retornam uma variável x que indica se a operação foi bem sucedida ou não.

4.2 Máquina de Redução de Grafos

Um programa ao ser executado pela máquina de redução de grafos MiniBPEL-AM é inicialmente traduzido para um grafo. Esta tarefa é feita utilizando um *Parser* que aplica um conjunto de regras de tradução $\mathcal{T}[\![P]\!]$. De posse do grafo, a máquina de redução aplica sucessivas regras de redução, até que não seja mais possível aplicar outras regras. A arquitetura da máquina pode ser observada na Figura 23.

Quando há a necessidade de se realizar uma chamada de operação, uma tupla (o, v, l) é escrita no *buffer* de saída O da máquina, fazendo com que o *gerenciador de operações* realize a operação o e retorne o resultado da chamada no *buffer* de entrada I.

Durante o período em que o *gerenciador de operações* está trabalhando, a máquina de redução pode executar novas reduções em vértices candidatos ou apenas esperar o retorno do *gerenciador de operações*, caso não existam regras para serem aplicadas no momento.

A máquina de redução de grafos proposta é uma extensão da arquitetura da máquina de redução apresentada na seção 3.1.2. As modificações que serão feitas irão alterar principalmente o componente Gerenciador de Serviços de PEWS-AM (Figura 8), o qual será substituído por um componente extensível chamado de *gerenciador de operações* (Figura 23). O *gerenciador de operações* tem a função de ler o *buffer* de saída da máquina de

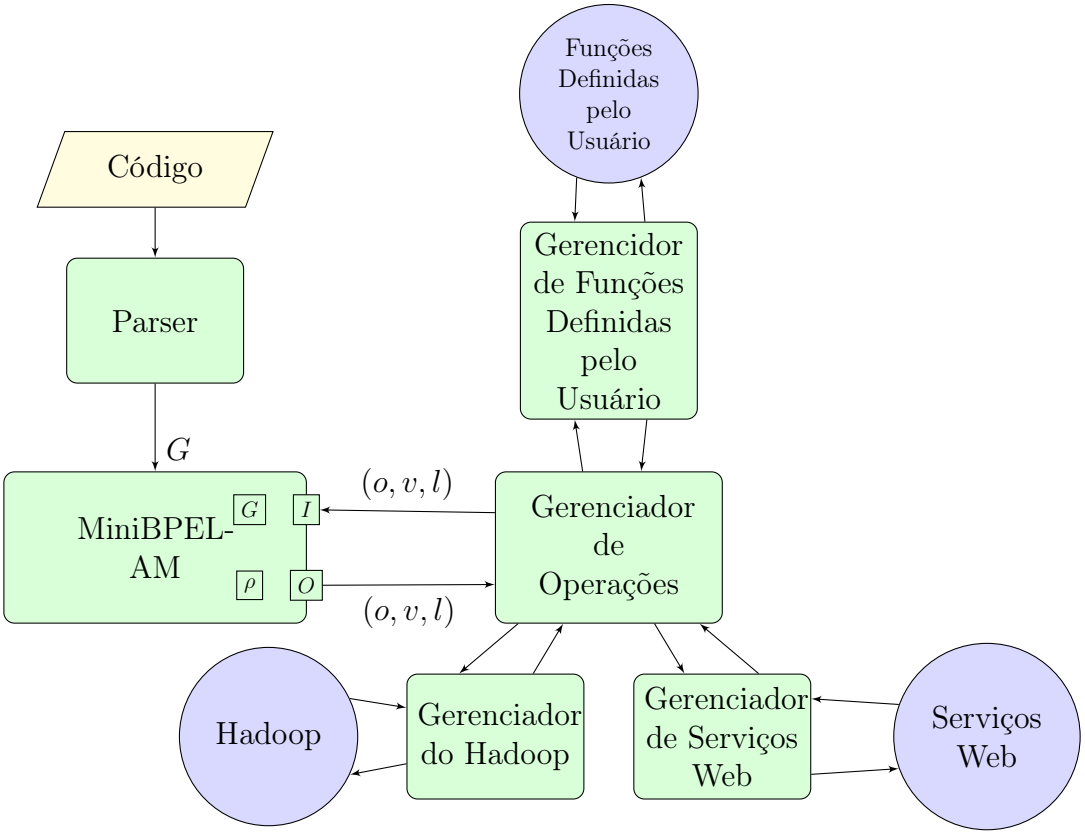


Figura 23: Arquitetura da máquina de redução MiniBPEL-AM.

redução de grafos, realizar uma operação e escrever no *buffer* de entrada da máquina o resultado da operação chamada.

Na máquina proposta o *gerenciador de operações* possuirá sub-gerenciadores para poder tratar cada tipo de operação que seja chamada pela máquina de redução de grafos. Por exemplo, um sub-gerenciador permitirá tratar chamadas a serviços web. Outro sub-gerenciador irá gerenciar a chamada de operações de *big data* mais específicas, como o caso de operações de tarefas do Hadoop, ou outro sistema que possua operações de *big data*, como ilustra a Figura 24.

Desta forma o sub-gerenciador do Hadoop ficará responsável pela chamada das operações *mapReduce*, $hdfs$, *pig* e *hive* citada na seção anterior. O sub-gerenciador de serviços web mostrado na Figura 24 corresponde ao gerenciador de serviços web da máquina PEWS-AM, tendo como responsabilidade realizar chamadas a serviços web.

Novas operações podem ser chamadas através da utilização de um sub-gerenciador de funções definidas pelo usuário. Novos sub-gerenciadores podem ser utilizados para a chamada de operações não relacionadas a chamada de serviços web ou tarefas do Hadoop. Por exemplo, novos sub-gerenciadores podem ser definidos especificamente para imple-

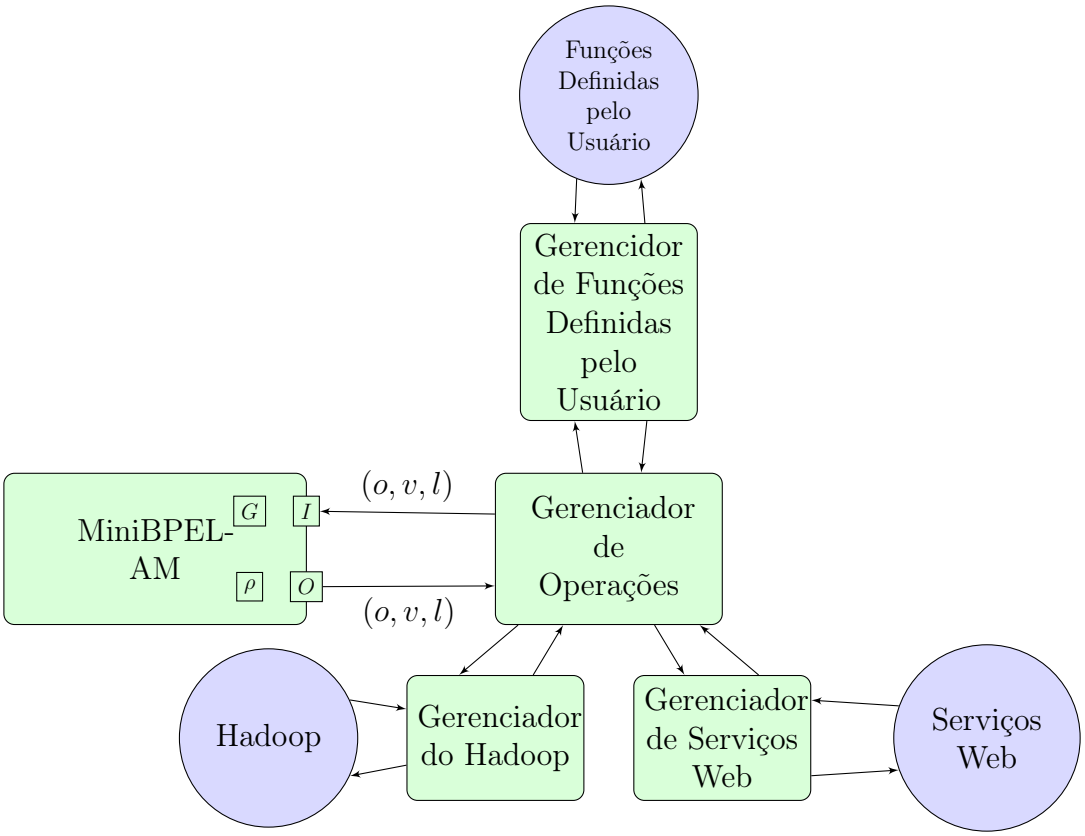


Figura 24: Comunicação do *núcleo de redução* com o *gerenciador de operações* na máquina MiniBPEL-AM.

mentar chamadas a RPC/RMI, CORBA ou qualquer outra tecnologia que implemente operações, as quais poderão ser integradas ao fluxo de trabalho definido pelo programa MiniBPEL-AM.

A máquina de redução utiliza como configuração inicial uma 4-tupla $\langle \mathcal{T}[\![P]\!], \emptyset, \epsilon, \epsilon \rangle$, onde $\mathcal{T}[\![P]\!]$ é o grafo gerado a partir das regras de tradução apresentadas na seção 4.3; $\emptyset$ indica que o ambiente de variáveis ρ começa vazio; e por fim os *buffers* de entrada e saída estão vazios (ϵ).

4.3 Regras de Tradução

Nesta seção as regras de tradução da máquina MiniBPEL serão descritas. A função $\mathcal{T}$ é definida para transformar programas P em grafos.

Grafos são representados como pares $\langle V, A \rangle$. A função $\mathcal{T}$ atribui um nome único a cada vértice do conjunto V. O conjunto de arestas de controle A representa as restrições de fluxo de controle impostas pelos construtores da linguagem.

O número de arestas que chegam e saem de um nodo do grafo são utilizadas pelas regras de tradução. Estas informações são disponibilizadas pelas funções definidas a seguir:

- $indegree(v)$ informa o número de arestas de controle que chegam ao nó v;
- $outdegree(v)$ que informa o número de arestas de controle que saem de um nó v;

Os nodos de maior relevância para a maioria das regras serão os nodos que tenham $indegree = 0$ (nodos de entrada) e $outdegree = 0$ (nodos de saída).

4.3.1 Chamada de Operação

Dada uma chamada de operação $S(\overline{E}; \overline{X})$, na qual S representa o nome de uma operação, $\overline{E}$ representa uma lista de expressões de entrada e $\overline{X}$ uma lista de identificadores de saída, a aplicação da regra de tradução $\mathcal{T}[\![S(\overline{E}; \overline{X})]\!]$ é definida a seguir:

$$\mathcal{T}[\![S(\overline{E}; \overline{X})]\!] = \langle \{w : S!\overline{E}, w' : S?\overline{X}\},\ \{w \mapsto w'\} \rangle$$

São criados os nós w, w' e uma aresta ligando os dois nodos. O nó w representa a chamada da operação S utilizando a lista $\overline{E}$ de expressões como parâmetros de entrada.

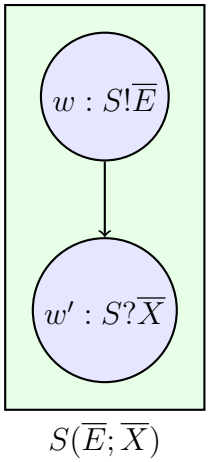


Figura 25: Grafo da chamada de operação.

Enquanto o nó w' representa o retorno da operação. Quando a operação S terminar sua execução os valores de retorno serão associados a lista de variáveis de $\overline{X}$ no nó w'. A aresta ligando os nós w e w' define que a chamada do serviço (w) deve ocorrer antes do retorno (w'), como ilustrado na Figura 25.

Ponto e virgula é utilizado para separar as listas de expressões de entrada e de identificadores de saída. O que deixa mais claro onde começa cada lista, quando comparamos com a chamada de operações da máquina PEWS-AM que utiliza virgula para separar estas listas.

4.3.2 Composição em Paralelo

Dados dois programas P_1 e P_2 escritos na linguagem MiniBPEL, a regra $\mathcal{T}[\![P_1||P_2]\!]$ define a regra de tradução para a execução em paralelo de dois programas P_1 e P_2. A composição é definida como a união dos vértices e arestas dos grafos que representam os programas P_1 e P_2:

$$\begin{aligned}&\mathcal{T}[\![P_1\|P_2]\!] = \langle V_1 \cup V_2,\ A_1 \cup A_2\rangle\\ &\quad\textbf{where}\\ &\qquad\langle V_1, A_1\rangle = \mathcal{T}[\![P_1]\!], \quad \langle V_2, A_2\rangle = \mathcal{T}[\![P_2]\!],\end{aligned}$$

Na Figura 26 é mostrado o grafo resultante da aplicação da regra de tradução da composição em paralelo $\mathcal{T}[\![P_1||P_2]\!]$ para dois programas P_1 e P_2.

A composição em paralelo da máquina MiniBPEL-AM é mais simples que a composição em paralelo da máquina PEWS-AM por não utilizar as arestas de dependência de

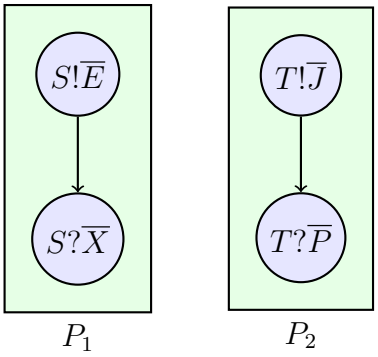


Figura 26: Grafo de uma composição em paralelo.

dados, consumindo assim menos recursos para armazenar o grafo em memória.

4.3.3 Composição Seqüencial

Dados dois programas P_1 e P_2, onde cada um destes programas representa um programa MiniBPEL válido, a regra de tradução $\mathcal{T}[\![P_1; P_2]\!]$ é definida por:

$$\mathcal{T}[\![P_1; P_2]\!] = \langle V_1 \cup V_2,\ A_1 \cup A_2 \cup A' \rangle$$
where
$$\langle V_1, A_1 \rangle = \mathcal{T}[\![P_1]\!],$$
$$\langle V_2, A_2 \rangle = \mathcal{T}[\![P_2]\!],$$
$$A' = \{w \mapsto w' | w \in V_1, outdegree(w) = 0, w' \in V_2, indegree(w') = 0\}$$

A aplicação da regra de tradução de uma composição sequencial é representada pela união dos vértices dos grafos dos dois programas. O conjunto de arestas resultantes é formado pela união das arestas A_1 e A_2 mais A'. Onde A' é o conjunto de arestas que partem de todos os vértices de P_1 que possuem $outdegree = 0$ com destino a todos os vértices de P_2 que possuem $indegree = 0$.

A composição sequencial é mais simples na máquina MiniBPEL-AM quando comparada com a máquina PEWS-AM por dois fatores: i) não faz uso de arestas de dependência de dados e ii) por possuir um número menor de arestas de controle.

O número menor de arestas de controle é alcançado criando arestas apenas dos nodos de saída de um programa P_1 para os nodos de entrada de um programa P_2 (Figura 27), enquanto que na máquina PEWS-AM a composição sequencial cria arestas de todos os nodos do programa P_1 para os nodos de entrada do programa P_2, o que acaba criando

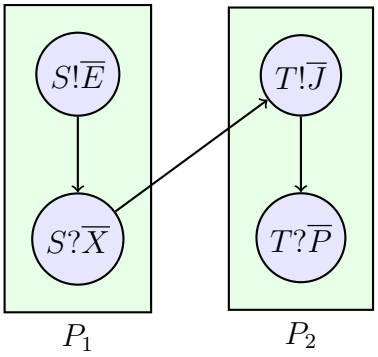


Figura 27: Grafo de uma composição seqüencial.

um grande número de arestas quando o programa P_1 possui muitos nodos.

4.3.4 Escolha Não Determinística

Dado um programa "$[E_1]P_1+[E_2]P_2$", onde P_1 e P_2 são programas MiniBPEL e E_1 e E_2 são as respectivas guardas de cada programa MiniBPEL, representa uma escolha não determinística.

A aplicação desta regra cria um grafo, onde cada expressão possui arestas de controle apontando para seus respectivos programas e um vértice adicional +, que possui arestas para cada uma das expressões a serem avaliadas.

A regra para tradução da escolha não determinística define que o conjunto de vértices do grafo gerado é composto pela união dos vértices V_1, V_2 e V', onde V' representa os vértices + e vértices para cada expressão a ser avaliada.

$$\begin{aligned}
&\mathcal{T}[\![[E_1]P_1+[E_2]P_2]\!] = \langle V_1 \cup V_2 \cup V',\ A_1 \cup A_2 \cup A' \rangle \\
&\quad \textbf{where} \\
&\qquad \langle V_1,\ A_1 \rangle = \mathcal{T}[\![P_1]\!], \quad \langle V_2,\ A_2 \rangle = \mathcal{T}[\![P_2]\!], \\
&\qquad V' = \{v : +, v_1 : E_1, v_2 : E_2\}, \\
&\qquad A' = \{v \mapsto v_1, v \mapsto v_2\} \cup \Delta_1 \cup \Delta_2, \\
&\qquad \Delta_1 = \{v_1 \mapsto w \mid w \in V_1, indegree(w) = 0\}, \\
&\qquad \Delta_2 = \{v_2 \mapsto w \mid w \in V_2, indegree(w) = 0\}
\end{aligned}$$

O conjunto de arestas de controle é formado pelas arestas A_1, A_2 e A', onde A' é o conjunto de arestas que partem do vértice + para cada uma das expressões em união com o conjunto de arestas que partem de cada uma das expressões para os vértices com $indegree = 0$ de seus respectivos programas.

A Figura 28 mostra o grafo resultante da execução da regra de tradução para escolha não determinística $\mathcal{T}[\![[E_1]((A;B)||(C;D))+[E_2]((E;F)||(G;H))]\!]$.

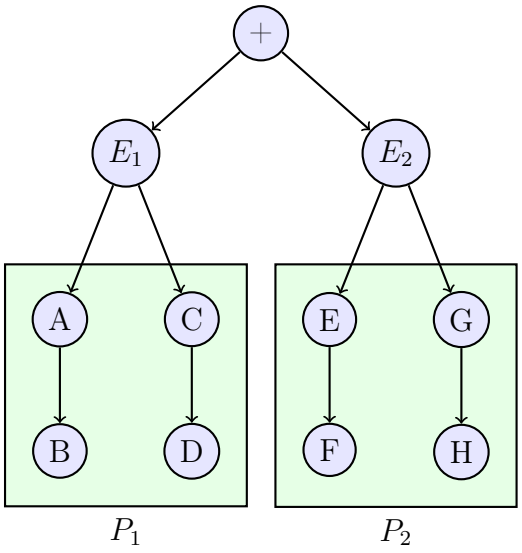


Figura 28: Grafo de uma escolha não determinística.

4.3.5 Condicional

A regra de tradução de um programa condicional é definida de forma similar ao modo como a regra de escolha não determinística, com a diferença de que na regra do condicional há apenas uma expressão a ser avaliada. A segunda expressão, por não existir é definida como um nodo **false** sem nodos filhos.

A regra de tradução do condicional define que o conjunto de vértices do grafo gerado é formado pelos vértices V do programa P em união com um vértice $+$, um vértice representando a expressão e um vértice representando o valor **false**.

$$
\begin{aligned}
&\mathcal{T}[\![\textbf{if } E \textbf{ then } P]\!] = \langle V \cup V',\ A \cup A' \rangle \\
&\quad \textbf{where} \\
&\qquad \langle V,\ A \rangle = \mathcal{T}[\![P]\!], \\
&\qquad V' = \{v : +, v_1 : E, v_2 : \textbf{false}\}, \\
&\qquad A' = \{v \mapsto v_1,\ v \mapsto v_2\} \cup \Delta, \\
&\qquad \Delta = \{v_1 \mapsto w \mid w \in V, indegree(w) = 0\}
\end{aligned}
$$

As arestas de controle são formadas pelas arestas A do programa P em união com V' e A', onde V' é o conjunto com as arestas de controle que ligam o vértice $+$ aos vértices que representam a expressão a ser avaliada e o valor **false**, e A' são as arestas que partem

da expressão com destino a todos os nós com $indegree = 0$ do programa P.

O grafo da Figura 29 é o resultado da aplicação da regra de tradução do condicional $\mathcal{T}[\![\textbf{if } E \textbf{ then } ((A;B)||(C;D))]\!]$.

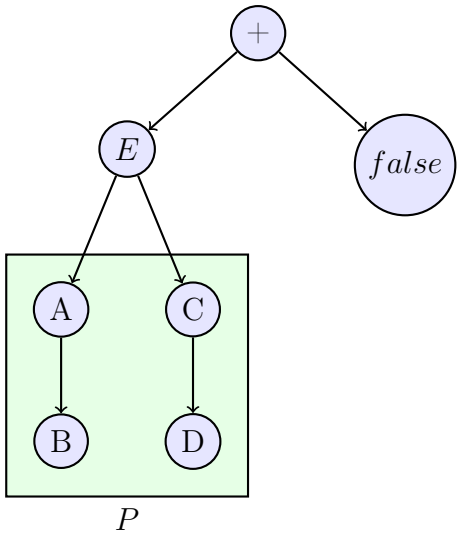


Figura 29: Grafo de um condicional.

4.3.6 While

O trecho de código "**while** E **do** P", onde E é uma expressão a ser avaliada e P é um programa MiniBPEL, representa uma estrutura de repetição *while*.

A regra de tradução *while*, quando aplicada, cria um grafo com um único nodo $\mathcal{V}$. Este nodo possui uma cópia da expressão a ser avaliada e do programa a ser executado posteriormente pelas regras de redução. A expressão E e o programa P são anotados neste nó para que possam ser avaliados durante a aplicação das regras de redução. Para representar estas anotações o vértice $\mathcal{V}$ será escrito como $\mathcal{V}E.P$ no grafo. A regra não cria arestas de controle, uma vez que apenas um vértice foi criado.

$$\mathcal{T}[\![\textbf{while } E \textbf{ do } P]\!] = \langle \{v : \mathcal{V}E.P,\ \emptyset\} \rangle$$

Esta regra de tradução cria o grafo apresentado na Figura 30.

Figura 30: Grafo do while.

Esta regra de tradução para estruturas de repetição é mais simples que a regra utilizada pela máquina PEWS-AM, que opta por traduzir o grafo do programa P, deixando o grafo da estrutura de repetição expandido e ocupando espaço mesmo que ele não vá ser avaliado. Já o grafo da estrutura de repetição da máquina MiniBPEL-AM só é expandido quando é avaliado pelas regras de redução, mantendo o número de nodos e arestas menores, quando comparado com a máquina PEWS-AM.

4.4 Regras de Redução

As regras de redução utilizadas pela máquina de redução são apresentadas nesta seção. O processo de avaliação ocorre sempre que algum determinado vértice se torna candidato a avaliação. Para que um vértice se torne candidato a avaliação ele precisa atender as restrições de cada regra e ser um dos seguintes vértices: $S!\overline{E}$ (chamada de operações), $S?\overline{X}$ (retorno de operações), $+$ (usado em escolhas não determinísticas e em condicionais) e $\mathcal{V}$ (usado pelo **while**).

O processo de execução da máquina MiniBPEL-AM é definida pela relação de avaliação:

$$\langle G, \rho, I, O\rangle \rightarrow \langle G', \rho', I', O'\rangle$$

onde a partir de um estado inicial $\langle G, \rho, I, O\rangle$, após um passo na avaliação, utilizando as regras desta subseção, chega-se ao estado $\langle G', \rho', I', O'\rangle$.

Para um conjunto H, tal que $x \notin H$, escreve-se $H[x]$, com o significado $H \cup \{x\}$.

As configurações da máquina MiniBPEL-AM são similares às configurações da máquina PEWS-AM mostradas na subseção 3.1.4. Isto é, assim como na máquina PEWS-AM são formadas por um grafo, um ambiente de variáveis, e *buffers* de entrada e saída.

As regras de redução são apresentadas a seguir:

4.4.1 Chamada de Operação

$$\frac{\begin{array}{c} indegree(w) = 0, \qquad Vars(\overline{E}) \subseteq Dom(\rho), \\ \overline{a} = Eval_{\rho}(\overline{E}), \quad A' = A - \{w \mapsto v \mid v \in V\}, \\ G = \langle V[w : S!\overline{E},\ w' : S?\overline{X}], A[w \mapsto w'] \rangle \end{array}}{\langle G, \rho, I, O \rangle \ \rightarrow \ \langle \langle V[w' : S?\overline{X}], A' \rangle, \rho, I, O[(S, w', \overline{a})] \rangle}$$

A regra de redução para a chamada de operações espera que: o grafo possua um vértice w do tipo $S!\overline{E}$ com $indegree = 0$; que as variáveis pertencentes ao conjunto de expressões de entrada $\overline{E}$ já tenham sido atribuídas, ou seja, pertençam ao ambiente ρ; que a lista de expressões de entrada $\overline{E}$ possa ser avaliada numa lista de valores $\overline{a}$, utilizando as variáveis do ambiente ρ; um vértice w' do tipo $S?\overline{X}$ e uma aresta de controle partindo de w com destino a w'. Em resumo, existe um vértice w pronto para ser avaliado que representa uma chamada de operação, como ilustrado no lado esquerdo da Figura 31.

O processo de avaliação consiste nos seguintes passos: *a*) escrever no *buffer* de saída O uma tripla $(S, w', \overline{a})$ com o nome da operação S a ser chamada, o vértice w' ao qual a máquina deve retornar o resultado e uma lista $\overline{a}$ de valores avaliados a partir da lista de expressões $\overline{E}$ que serão utilizados como parâmetros de entrada da operação S; *b*) remover todas as arestas de controle que partem de w; e *c*) remover o vértice w do grafo.

O resultado da avaliação desta regra é mostrado na Figura 31, ficando o vértice w' e uma tripla no *buffer* de saída.

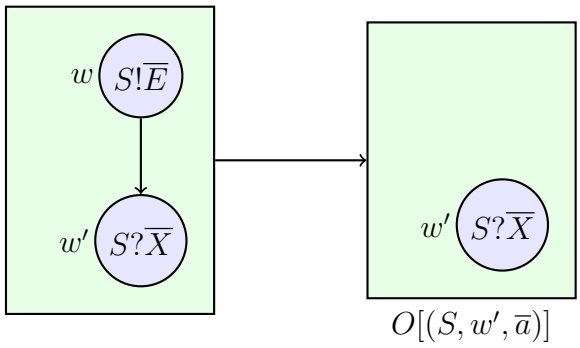


Figura 31: Grafo da chamada de uma operação.

Com a tripla $(S, w', \overline{a})$ no *buffer* de saída O, o *gerenciador de operações* mostrado na Figura 23, irá fazer uma chamada à operação S usando como parâmetro a lista $\overline{a}$.

Quando a operação chamada retornar seu resultado, o *gerenciador de operações* irá escrever uma nova tripla no *buffer* de entrada I da máquina. Esta nova tripla $(S, w', \overline{r})$

que é escrita no *buffer* de entrada da máquina é formada utilizando o nome da operação S que foi chamada, o vértice w' ao qual o resultado deve ser retornado e uma lista $\overline{r}$ com os valores retornados pela chamada da operação.

A partir deste ponto a máquina MiniBPEL-AM tem como avaliar o vértice de retorno de operação, que não podia ser avaliado antes porque não exista uma tripla no *buffer* de entrada com seu nome.

4.4.2 Retorno de Operação

$$\frac{\begin{array}{c} \mathit{indegree}(w) = 0, \qquad \overline{X} = (x_1, \ldots, x_k), \qquad \overline{r} = (v_1, \ldots, v_k), \\ \rho' = \rho \cup \{(x_i, v_i) \mid 1 \leq i \leq k\}, \quad A' = A - \{w \mapsto v \mid v \in V\} \end{array}}{\langle\langle V[w : S?\overline{X}], A\rangle, \rho, I[(S, w, \overline{r})], O\rangle \;\rightarrow\; \langle\langle V, A'\rangle, \rho', I, O\rangle}$$

A regra para retorno de operação espera que o grafo possua um vértice w do tipo $S?\overline{X}$, que possua $\mathit{indegree} = 0$ e uma entrada no *buffer* de entrada $I[(S, w, \overline{r})]$ que indique que existe um resultado de chamada de operação para o vértice w. Ou seja, existe um vértice de retorno de operação pronto para ser avaliado.

A avaliação associa cada um dos valores retornados $\overline{r}$ com cada uma das variáveis $\overline{X}$ no ambiente ρ. A tripla $(S, w, \overline{r})$ é removida do *buffer* de entrada I. O vértice w é removido do grafo, bem como todas as arestas de controle e de dependência de dados que partem de w. O resultado da avaliação desta regra pode ser visto na Figura 32.

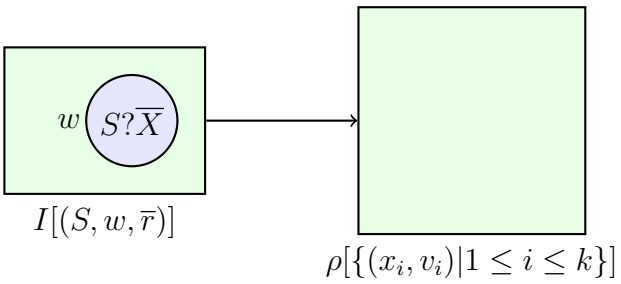


Figura 32: Grafo do retorno de uma operação.

4.4.3 Escolha Não Determinística

Uma escolha não determinística é identificada pela máquina quando o próximo vértice a ser avaliado é um vértice $+$.

Para o caso de escolha não determinística são utilizadas duas regras de avaliação: (i)

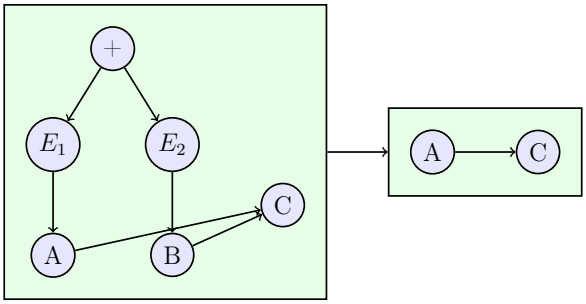


Figura 33: Redução de $([E_1]A + [E_2]B); C$ em $A; C$.

uma para quando uma das expressões avaliadas possui valor **true**; e (ii) as duas expressões avaliadas possuem valor **false**. Quando existem duas expressões com valor verdadeiro a primeira regra é utilizada e seleciona-se uma das expressões ao acaso.

A regra a seguir é usada quando uma das expressões da escolha não determinística avaliada possui valor **true**. Neste caso não é necessário verificar as outras expressões da escolha não determinística, removendo assim: o subgrafo filho da outra expressão; o vértice +; e os vértices das expressões, como ilustrado na Figura 34.

$$\frac{\begin{array}{c} indegree(v) = 0, \qquad Vars(E) \subseteq Dom(\rho), \qquad Eval_\rho(E) = \textbf{true}, \\ V' = \{u \in V \mid w' \mapsto^* u \wedge w \not\mapsto^* u\}, \qquad G = \langle V \setminus V', A\rangle \end{array}}{\langle\langle V[v : +, w : E, w' : E'], A[v \mapsto w, v \mapsto w']\rangle, \rho, I, O\rangle \;\rightarrow\; \langle G, \rho, I, O\rangle}$$

Para a remoção do subgrafo filho da outra expressão selecionamos para remoção o conjunto de vértices atingíveis a partir do vértice w' e que não são alcançáveis a partir de w.

Por exemplo, considere o programa $([E_1]A + [E_2]B); C$ em que a escolha entre duas operações (A e B) é seguida por uma operação C. No caso da guarda da operação A ser verdadeira, todos os vértices alcançáveis a partir do vértice E_2 e que não são alcançáveis a partir de E_1 serão apagados do grafo original, incluindo os vértices E_1 e +, como pode ser visto na Figura 33. Como resultado resta apenas um grafo onde os vértices A e C estão conectados por uma aresta de controle.

Na Figura 34 os vértices V' selecionados para remoção da expressão não escolhida seriam E, F, G, H e o próprio vértice $w' : E'$.

Caso a escolha não determinística possua duas expressões avaliadas em **false**, remove-se do grafo: o vértice +; os vértices das expressões falsas; e os subgrafos filhos de cada uma das expressões.

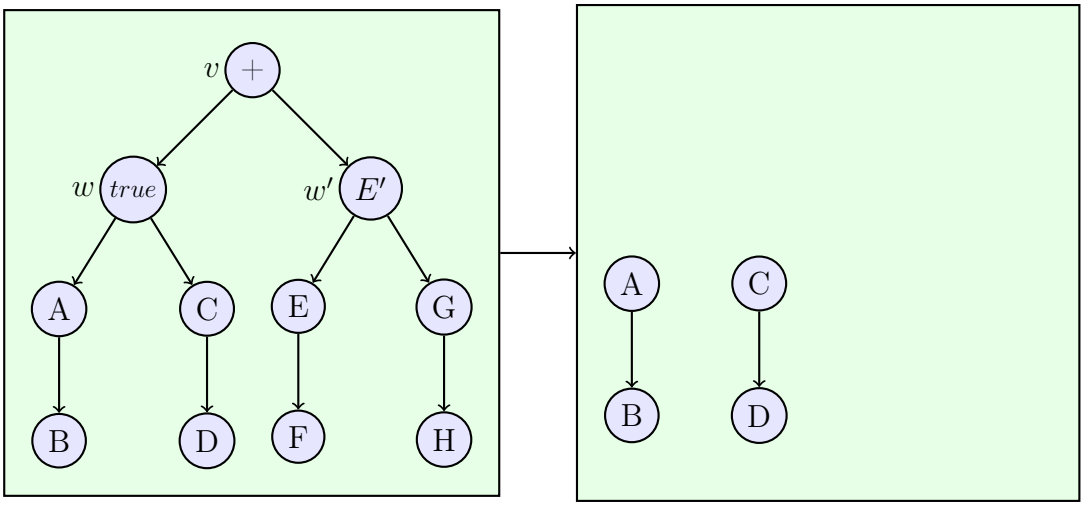


Figura 34: Grafo de uma escolha não determinística com uma expressão verdadeira.

$$\frac{\begin{array}{c} indegree(v) = 0, \qquad Vars(E) \cup Vars(E') \subseteq Dom(\rho), \qquad Eval_\rho(E) = Eval_\rho(E') = \textbf{false}, \\ V' = \{u \in V \mid w \mapsto^* u \wedge w' \not\mapsto^* u\}, \qquad V'' = \{u \in V \mid w' \mapsto^* u \wedge w \not\mapsto^* u\}, \\ G = \langle V \setminus (V' \cup V''), A \rangle \end{array}}{\langle \langle V[v : +, w : E, w' : E'], A[v \mapsto w, v \mapsto w'] \rangle, \rho, I, O \rangle \;\rightarrow\; \langle G, \rho, I, O \rangle}$$

Para a remoção dos subgrafos filhos de cada expressão remove-se do grafo o conjunto de vértices V' e V'', onde V' possui os vértices atingíveis a partir de w que não são alcançáveis pelo vértice w' correspondendo ao subgrafo com raiz em w e o conjunto de vértices V'' que possui os vértices atingíveis a partir de w' mas que não são alcançáveis pelo vértice w, correspondendo ao subgrafo com raiz em w'.

A Figura 35 ilustra a aplicação desta regra em um grafo com duas expressões falsas.

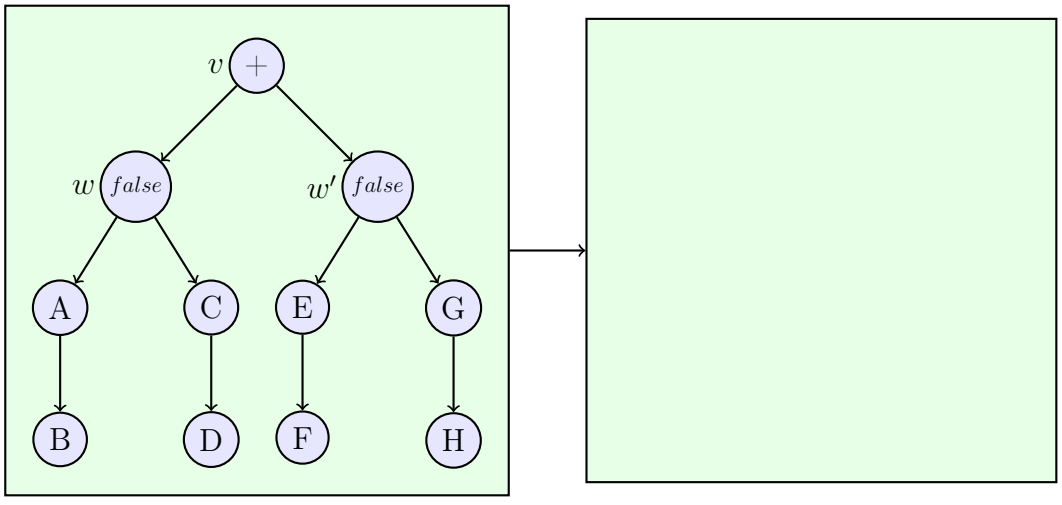


Figura 35: Grafo de uma escolha não determinística com todas as expressões falsas.

4.4.4 While

Esta regra deve ser aplicada quando o próximo vértice a ser avaliado é um vértice $\mathcal{V}E.P$, indicando que estamos tratando uma estrutura de repetição **while**.

A regra então expande o vértice para o grafo de uma estrutura condicional, em que caso a expressão E seja verdadeira permitirá a execução do grafo traduzido do programa P em seqüência com um novo nó $\mathcal{V}E.P$. A regra faz a seguinte transformação:

$$\textbf{while } E \textbf{ do } P \rightarrow \textbf{if } E \textbf{ then } (P; \textbf{while } E \textbf{ do } P)$$

Esta regra é aplicada quando o vértice $v : \mathcal{V}E.P$ possui $indegree = 0$. Neste caso o programa P é traduzido para um grafo $\langle V', A' \rangle = \mathcal{T}[\![P]\!]$ que fará parte do corpo do grafo $\langle V'', A'' \rangle$, correspondente ao grafo de um condicional.

$$\frac{\begin{array}{c} indegree(v) = 0, \qquad \langle V', A' \rangle = \mathcal{T}[\![P]\!], \\ V'' = V[v : \mathcal{V}E.P] \cup \{p_v : +, w_v : E, w'_v : false\} \cup V', \\ A'' = A \cup A' \cup \{p_v \mapsto w_v, p_v \mapsto w'_v\} \cup R \cup Q, \\ R = \{w_v \mapsto u | u \in V', indegree(u) = 0\}, \\ Q = \{u \mapsto v | u \in V' \wedge outdegree(u) = 0\} \end{array}}{\langle \langle V[v : \mathcal{V}E.P], A \rangle, \rho, I, O \rangle \rightarrow \langle \langle V'', A'' \rangle, \rho, I, O \rangle}$$

O conjunto de vértices V'' possui os vértices iniciais V em união com o conjunto de vértices V' traduzidos de P e os vértices $+$, E e **false** correspondentes ao condicional. O conjunto de arestas A'' é formado por: arestas iniciais A; arestas provenientes da tradução do programa P; arestas que ligam a expressão E a todos os vértices com $indegree = 0$ do programa P; e arestas que partem dos vértices do programa P com $outdegree = 0$ com destino ao vértice $v : \mathcal{V}E.P$. O resultado da aplicação desta regra de redução pode ser visto na Figura 36.

Observe que cada ocorrência de $\mathcal{T}[\![P]\!]$ gera um novo grafo, com novos nodos e arestas, da mesma forma que na regra do $unfolding$ da máquina PEWS-AM.

É importante notar que esta regra usa a técnica de tradução dinâmica (ou tradução *just-in-time*) [76]. O uso desta forma de tradução possibilita a simplificação tanto da especificação quanto da implementação da regra de redução do comando *while*.

Os vértices *while* no grafo só são expandidos no momento de sua avaliação. Enquanto a estrutura de repetição utilizada pela máquina PEWS-AM já inicia expandida. Esta

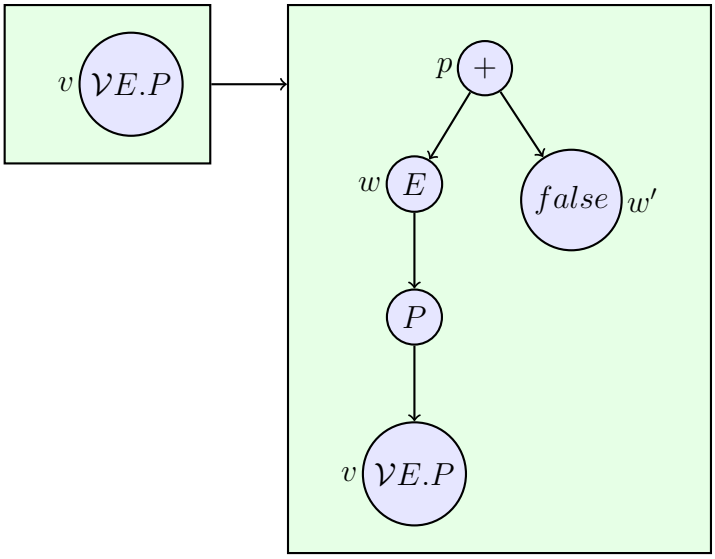


Figura 36: Grafo do while.

diferença nas estruturas de repetição das duas máquinas faz com que os grafos gerados pela máquina MiniBPEL-AM, tenham uma quantidade menor de nodos.

4.5 Exemplo de Composição

Nesta seção um exemplo será montado utilizando todos os construtores da linguagem MiniBPEL para criar uma composição de serviços e tarefas de *big data*. Neste exemplo uma organização busca saber se os seus consumidores estão aceitando ou rejeitando um determinado produto. Esta verificação é feita todos os dias e um relatório é exibido em um site protegido para posterior conferência. O fluxo de trabalho que representa este exemplo implementado pela máquina proposta pode ser visto na Figura 38.

A composição é formada pelos seguintes serviços web:

- `wait`: aguarda pela quantidade de tempo especificada como parâmetro;
- `collectData`: junta informações de diversas fontes e as armazena dentro do *cluster* do Hadoop na pasta especificada como parâmetro;
- `checkData`: verifica se há novos dados armazenados no *cluster* do Hadoop desde sua ultima execução e se existir alguma modificação o segundo parâmetro retorna verdadeiro. Caso contrário, **false** é retornado;
- `tweet`: envia um alerta para o perfil da aplicação na rede social Twitter;

- `publish`: faz a publicação dos dados processados no site privado da organização.

As seguintes chamadas de tarefa MapReduce fazem parte da composição:

- `mapreduce(‘‘normalize.jar’’; normalized)`: esta chamada MapReduce faz uma normalização dos dados colhidos. Prepara os dados colhidos para classificação. Esta tarefa remove mensagens duplicadas, corrige erros de codificação mais comuns, remove conteúdo gerado por programas de terceiros, dentre outras atividades. O parâmetro `normalized` retorna **true** caso a normalização tenha ocorrido sem problemas e **false** caso contrário;
- `mapreduce(‘‘classify.jar’’; classified)`: esta tarefa realiza a classificação dos dados com base em uma lista de palavras positivas e negativas. Cada mensagem recebe uma classificação positiva ou negativa com base nesta lista de palavras. O parâmetro `classified` retorna **true** caso a classificação tenha sido concluída e **false** caso contrário.

O código do programa MiniBPEL proposto neste exemplo é exibido na Figura 37:

```
while true do (
    (wait("1 day"); collectData("/dir/"))
    ||
    (checkData("/dir/"; haveData);
    (if haveData then
        mapreduce("normalize.jar"; normalized);
        [normalized] (
            mapreduce("classify.jar"; classified);
            tweet("last day classification: ok");
            publish("/dir/")
        ) +
        [not normalized] tweet("last day classification:
   error")
    );
    wait("1 day"))
)
```

Figura 37: Programa MiniBPEL para o problema proposto.

Como esta composição deve ser executada todos os dias, uma estrutura de repetição em *loop* infinito é utilizada. Dentro do *loop* duas ações são tomadas: a) a coleta dos dados e b) o processamento dos dados.

A coleta dos dados é representada pelo sub-programa:

```
wait("1 day"); collectData("/dir/")
```

enquanto que o processamento dos dados é representado pelo código ilustrado pelo código:

```
checkData("/dir/"; haveData);
(if haveData == true then
    mapreduce("normalize.jar"; normalized);
    [normalized] (
        mapreduce("classify.jar"; classified);
        tweet("last day classification: ok");
        publish("/dir/")
    ) +
    [not normalized] tweet("last day classification:
error")
 );
 wait("1 day")
```

Na coleta dos dados o programa aguarda por um dia, através do serviço web `wait`, e depois executa o serviço de coleta de dados. Já no processamento dos dados, o serviço `checkData` verifica se existem dados novos para serem processados. Caso existam dados a serem processados uma tarefa MapReduce é chamada para normalizar os dados e prepará-los para classificação. Se esta normalização ocorreu sem problemas a classificação é feita, uma mensagem é enviada para o twitter da aplicação e os dados são publicados para posterior consulta. Se ocorreu algum erro com a normalização uma mensagem é enviada para o twitter da aplicação informando que um erro ocorreu. Por fim a fase de processamento aguarda por mais um dia.

Utilizando as regras de tradução propostas, se cria o seguinte grafo:

onde E representa **true** e P representa todo o corpo do *while*. Neste caso a máquina utilizará a regra de redução *while* para reduzir em um passo o grafo, criando o grafo mostrado na Figura 38.

Para facilitar a visualização do grafo, as operações *wait*, *checkData*, *collectData*, *mapreduce*, *tweet* e *publish* estão sendo representadas com apenas um vértice. Em um grafo

normal cada uma destas operações teria um vértice de chamada e um de retorno ligado por uma aresta de controle.

4.6 Conclusões do Capítulo

Neste capítulo pôde ser vista a similaridade da máquina proposta neste trabalho e a PEWS-AM: O trabalho proposto revisita o de Aguiar [73], resultando em uma proposta ao mesmo tempo simplificada e mais abrangente do trabalho apresentado por ele.

A extensibilidade da máquina é o resultado de generalizar a arquitetura, através da substituição do gerenciador de serviços web por um *gerenciador de operações* que permite a utilização de operações de diversos domínios.

A eliminação das arestas de dependência de dados permite a representação do grafo de forma sucinta e gastando menos recursos, o que facilita a aplicação de certas regras. No entanto o trabalho de verificar se uma variável foi ou não declarada antes de ser utilizada continua sendo feito, em PEWS-AM com a utilização das arestas de dependência de dados, e na máquina MiniBPEL-AM através de uma condição $Vars(X) \in Dom(\rho)$ que deve ser satisfeita dentro de determinadas regras.

Outra mudança relevante com relação a máquina PEWS-AM foi feita no ambiente de variáveis ρ utilizado pelas máquinas. A máquina PEWS-AM cria um novo ambiente de variáveis para cada iteração de uma estrutura de repetição, enquanto a máquina MiniBPEL-AM continua utilizando o mesmo ambiente de variáveis para todas as iterações. Esta mudança faz com que menos recursos sejam necessários para armazenar os valores das variáveis, uma vez que temos apenas um único ambiente de variáveis.

No próximo capítulo a implementação da máquina MiniBPEL-AM será detalhada.

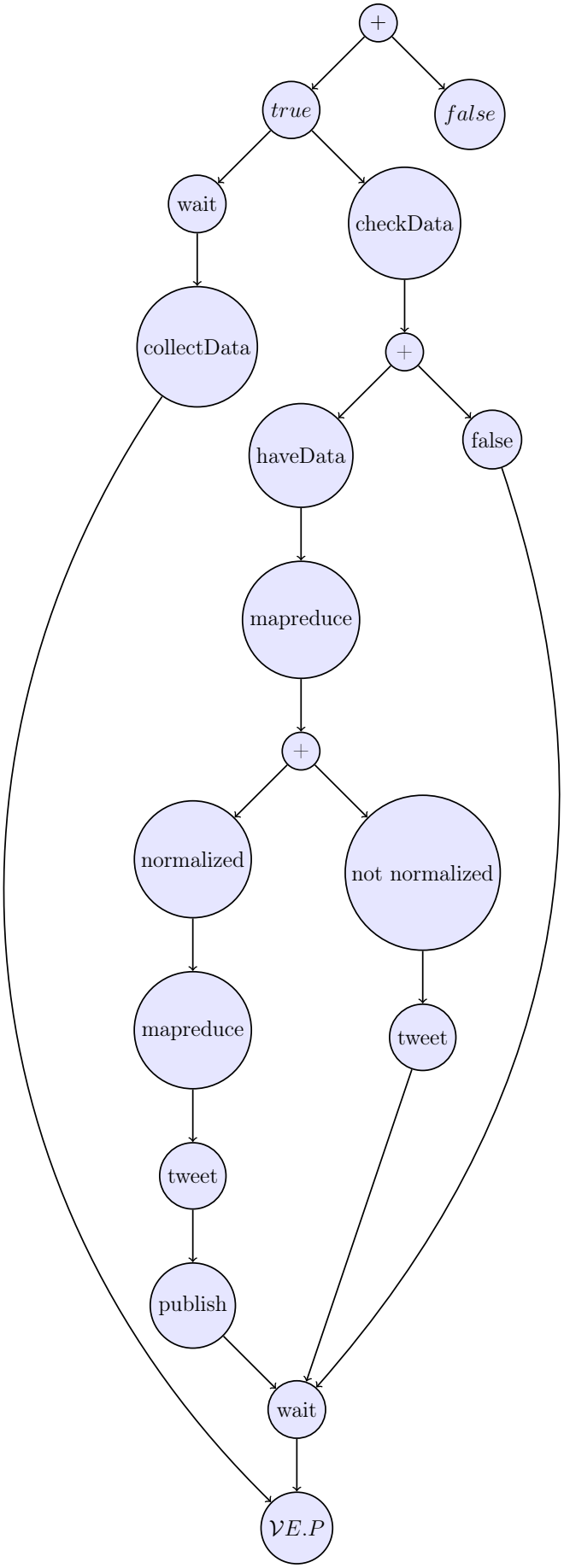


Figura 38: Grafo do exemplo após aplicação da regra de redução while.

5 Implementação

Neste capítulo será feito o detalhamento de como a máquina MiniBPEL-AM foi implementada. Os principais componentes da máquina serão apresentados e detalhados, tais como o *parser*, o *núcleo de reduções* e o *gerenciador de operações*. Será descrito como a extensibilidade proporcionada pela máquina foi alcançada e os principais componentes utilizados para alcançar esta extensibilidade.

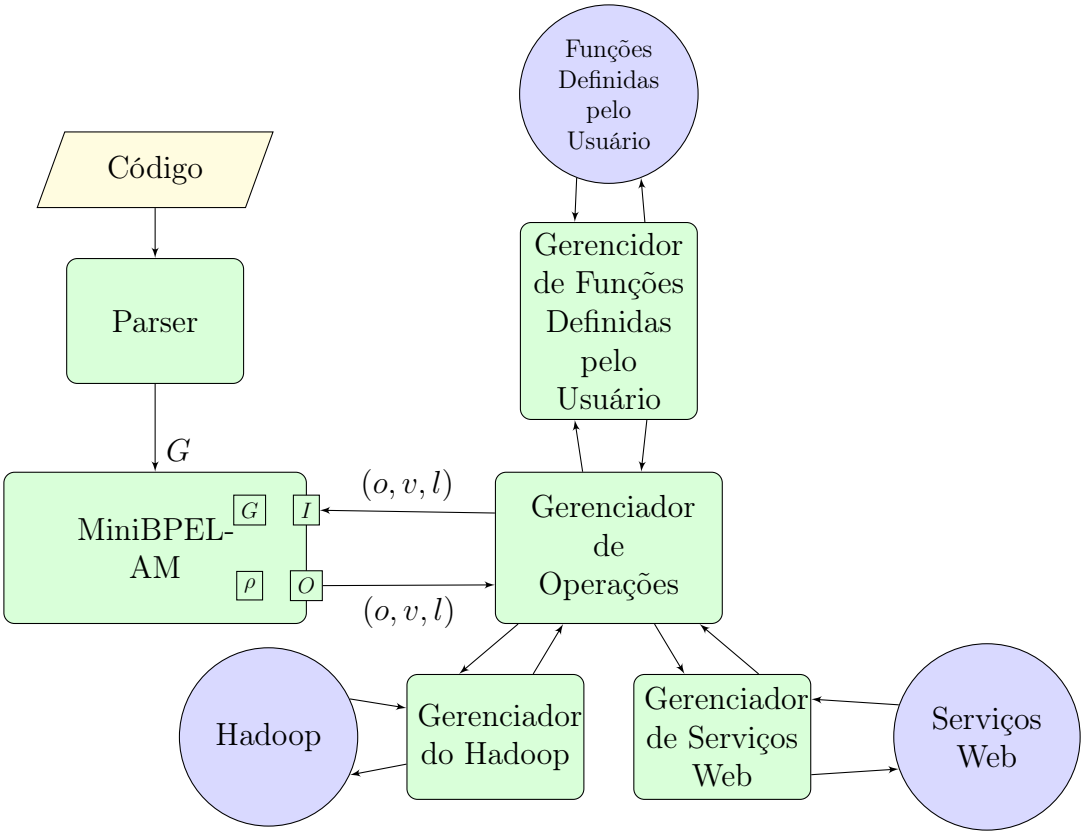


Figura 39: Arquitetura da máquina MiniBPEL-AM.

Como ilustrado pela Figura 39 os principais componentes da máquina MiniBPEL-AM são o *parser*, o *núcleo de redução* e o *gerenciador de operações*. Para a execução de um fluxo de trabalho, um código MiniBPEL deve ser escrito utilizando as regras apresentadas na gramática da seção 4.1. Este código é traduzido no *parser* para um grafo direcionado. O

grafo é então utilizado pelo *núcleo de redução* para executar o fluxo de trabalho. Tuplas são escritas no *buffer* de saída do *núcleo de reduções* quando se faz necessário realizar chamada de operações. Estas tuplas são lidas pelo *gerenciador de operações* que fica responsável por realizar a chamada das operações e escrever o resultado destas chamadas no *buffer* de entrada do *núcleo de reduções*. Quando uma tupla é encontrada no *buffer* de entrada, o *núcleo de reduções* avalia o vértice associado a tupla e segue a execução do fluxo de trabalho. O *núcleo de reduções* segue aplicando regras de redução até não haver mais vértices que possam ser avaliados.

A máquina MiniBPEL-AM foi implementada utilizando a linguagem Python [77]. O *parser* da máquina que tem como responsabilidade transformar os códigos escritos na linguagem MiniBPEL para grafos através da aplicação das regras de tradução apresentadas na seção 4.3 foi implementado com o módulo PLY (*Python Lex-Yacc*) [78]. O módulo PLY utilizado é uma implementação Python do Yacc (*Yet Another Compiler - Compiler*) [79] (gerador de *parsers* para a linguagem C).

O grafo gerado pelo *parser* possui um conjunto de vértices e arestas que são armazenados por uma estrutura de grafo direcionado provido pelo módulo NetworkX [80]. Este módulo facilita a criação e manipulação de grafos e funções tais como '*indegree*' e '*outdegree*' já se encontram implementadas neste módulo.

Assim que instanciada, a máquina MiniBPEL-AM cria:

- um objeto grafo do módulo NetworkX;
- um ambiente de variáveis rho;
- um objeto 'reductionRules' que permite executar as regras de redução;
- um objeto '*parser*' que transforma o código fonte em grafos respeitando a gramática definida na seção 4.1 e as regras de tradução da seção 4.3;
- *buffers* de entrada, saída e de erros;
- uma thread 'operationManager' que implementa o *gerenciador de operações*.

Como ilustrado na Figura 40, o construtor da máquina MiniBPEL-AM ainda faz o *parsing* do código fonte, e armazenando seu resultado no objeto grafo criado nas primeiras linhas do construtor.

```
def __init__(self, code):
    self.graph = nx.MultiDiGraph()
    self.rho = dict()
    self.reductionRules = ReductionRules(self)
    self.parser = parser

    self.inputBuffer = []
    self.outputBuffer = []
    self.errorBuffer = []

    #cria a thread do gerenciador de operações
    self.operationManager = OperationManager(self)
    self.operationManager.start()

    self.graph = self.parser.parse(code)
```

Figura 40: Construtor da máquina MiniBPEL-AM.

Cada vértice criado pelo *parser* possui um nome e uma lista de atributos no formato chave/valor. Todos os vértices possuem um atributo chamado '*type*' que é utilizado pelas regras de redução para identificar qual é o tipo de nodo que está sendo analisado e aplicar a regra de redução correspondente. O restante dos atributos do vértice é especifico de acordo com o seu tipo, por exemplo, um vértice do tipo '*operation_call*' terá além do atributo '*type*' os atributos '*operation*' e '*expressions*', um vértice do tipo '*operation_response*' terá os atributos '*operation*' e '*variables*'; já um vértice do tipo '+' não terá nenhum atributo adicional.

As arestas são formadas por tuplas de nomes de vértices. Isto é, dado dois vértices com nomes 'a' e 'b' uma aresta entre estes dois vértices é criada informando uma tupla (a,b).

Os tipos de vértices que o *parser* pode criar são: '*operation_call*' para chamada de operações; '*operation_response*' para retorno de operações; '*plus*' para escolha não determinística, apresentado neste trabalho como '+'; '*expression*' para expressões aritméticas ou booleanas; e '*while*' para estrutura de repetição.

Além do atributo '*type*' cada vértice possui um conjunto de atributos específicos que irão ser utilizados na fase de avaliação das regras de redução. Cada vértice '*operation_call*' possui adicionalmente os atributos '*operation*' que representa o nome da operação que está sendo chamada e o atributo '*expressions*' que informa quais as expressões que deverão ser avaliadas e passadas como parâmetros para a chamada da operação. Vértices do tipo '*operation_response*' possuem os atributos '*operation*' que informa que qual operação está aguardando que sua execução seja concluída e '*variables*' que guarda uma lista de

variáveis para onde os valores de retorno da execução da operação devem ser armazenados. Vértices '*plus*' não possuem atributos extras. Cada vértice do tipo '*expression*' possui como atributo adicional '*value*' que guarda um texto da expressão que ainda não foi avaliada. Vértices '*while*' possuem atributos '*expression*' que guarda o texto de uma expressão que será utilizada como critério de parada para a estrutura de repetição e '*production*' que guarda o corpo da estrutura de repetição que será replicado sempre que um novo *loop* for iniciado.

Uma vez que o programa fonte foi transformado em um uma instancia de grafo, o *núcleo de redução* da máquina MiniBPEL-AM escolhe um dos vértices raízes, isto é, vértices que não possuem arcos entrantes (*indegree* do vértice com valor zero), e aplica a regra de redução correspondente. Quando a aplicação da regra de redução é concluída o *núcleo de redução* espera por um segundo e verifica se ainda existem vértices raízes para serem executados e verifica se o *gerenciador de operações* retornou algum erro. O *núcleo de redução* seleciona um novo vértice raiz para avaliação caso existam vértices raízes a serem executados e não existam erros retornados pelo *gerenciador de operações*. A execução do *núcleo de redução* é interrompida caso não existam mais vértices raízes a serem executados ou caso algum erro tenha sido retornado pelo *gerenciador de operações*. O método `run` mostrado na figura 41 representa o *núcleo de redução* da máquina MiniBPEL-AM.

```
def run(self):
    while (len(self.get_root_nodes()) > 0) and (not self.errorBuffer):
        one_root = self.get_one_root_node()
        if self.graph.node[one_root]['type'] == 'operation_call':
            self.reductionRules.operation_call(one_root)
        elif self.graph.node[one_root]['type'] == 'operation_response':
            self.reductionRules.operation_response(one_root)
        elif self.graph.node[one_root]['type'] == 'plus':
            self.graph = self.reductionRules.guarded_choices(self.graph, one_root)
        elif self.graph.node[one_root]['type'] == 'while':
            self.graph = self.reductionRules.whiledo(self.graph, one_root)
        time.sleep(1)
    self.operationManager.stop = True
```

Figura 41: Método principal do *núcleo de redução*.

O método '`self.get_root_nodes()`' utilizado no *núcleo de redução* retorna uma lista dos vértices raízes, isto é, vértices com *indegree* com valor zero. Enquanto o método '`self.get_one_root_node()`' retorna aleatoriamente um dos vértices raízes. Desta forma a cada nova iteração o *núcleo de redução* seleciona um vértice raiz de forma não determinística.

Para cada tipo de vértice raiz existe uma regra de redução associada. As regras de

redução verificam se o vértice possui os pre requisitos para ser avaliado e modifica o grafo de acordo com as regras mostradas na seção 4.4.

Quando uma chamada de operação é realizada, a lista de expressões que o vértice possui é avaliada, uma tupla é escrita no *buffer* de saída da máquina e o vértice do tipo '*operation_call*' é removido do grafo. Neste ponto a máquina procura outro vértice raiz para avaliar, caso não exista nenhuma outra raiz além do vértice de retorno de operação para ser avaliado, então a máquia 'espera' que o *gerenciador de operações* retorne o resultado da chamada da operação. A 'espera' está implementada na forma de uma espera ociosa[1] com um temporizador, isto é, o *núcleo de redução* fica em *loop* checando a cada segundo se os vértices raízes do tipo '*operation_response*' já podem ser avaliados. Um temporizador de um segundo foi escolhido para diminuir o gasto de CPU enquanto se espera uma resposta do *gerenciador de operações*.

Caso o *gerenciador de operações* encontre algum erro ao realizar uma chamada de operação, ele pode re-executar a chamada até três vezes antes de retornar uma tupla para a máquina MiniBPEL-AM informando o erro, através do *buffer* de entrada da máquina. A máquina por sua vez, ao receber uma tupla de erro, trata de encerrar a execução do fluxo de trabalho.

Quando o *gerenciador de operações* termina de executar uma operação, uma entrada é escrita no *buffer* de entrada da máquina, permitindo que a máquina possa avaliar o vértice de retorno de operação que aguardava a execução da operação. A regra de redução de retorno de operação verifica se existe uma tupla no *buffer* de entrada com o nome de algum vértice raiz do tipo '*operation_response*' ainda não avaliado. Se houver algum vértice de retorno de operação aguardando retorno então a lista de valores retornado pela execução da operação é associada à lista de variáveis no ambiente *rho* e o vértice de retorno de operação é removido do grafo.

O *gerenciador de operações* verifica periodicamente o *buffer* de saída da máquina MiniBPEL-AM. Esta verificação periódica foi implementada na forma de uma espera ociosa com temporizador de um segundo, assim como no *núcleo de redução*. Isto é, o Gerenciador inicia um *loop* e a cada iteração ele verifica se há alguma tupla no *buffer* de saída da máquina aguardando um segundo antes de iniciar a próxima iteração do *loop*. Quando o *gerenciador de operações* encontra uma tupla no *buffer* de saída esta tupla é encaminhada para um de seus sub-gerenciadores com base no nome da operação. As operações possuem prefixos especiais para que o *gerenciador de operações* possa encaminhar

[1] Testar continuamente uma váriavel até que algum valor apareça. [81]

a operação para um sub-gerenciador que possa processa-la corretamente. Por exemplo:

$$ws.weather.getTemp(\text{“}teresina\text{”}, \text{“}brazil\text{”}; T)$$

O trecho acima é um código MiniBPEL válido. A operação getTemp possui um prefixo inicial ‘*ws*’ e um prefixo intermediário ‘*weather*’. O prefixo inicial é utilizado pelo *gerenciador de operações* para identificar qual sub-gerenciador deve processar a operação. Já o prefixo intermediário será utilizado pelo sub-gerenciador que trata de serviços web para chamar a operação ‘*getTemp*’ no serviço ‘*weather*’.

Um arquivo de configurações deve ser fornecido para informar quais prefixos estarão associados aos sub-gerenciadores. Um exemplo deste arquivo de configuração pode ser visto na Figura 42.

```
SETTINGS = {
    'webservice': {
        'module': 'mbpel.om.ws.webservice.WebServiceManager',
        'prefix': 'ws',
        'services': {
            'weather': {
                'wsdl': 'http://marcioalves.com.br:8080/Weather/services/Weather?wsdl',
                'operations': ['getTemp']
            },
        },
    },
    'hadoop': {
        'module': 'mbpel.om.hd.hadoop.HadoopManager',
        'prefix': 'hd',
        'clusters': {
            'vm': {
                'host': '127.0.0.1',
                'port': '2222',
                'user': 'root',
                'pass': 'hadoop',
                'operations': ['pig', 'ssh', 'hdfs'],
            },
        },
    },
}
```

Figura 42: Exemplo de arquivo de configuração.

Este arquivo possui apenas a definição de um dicionário chamado SETTINGS. Cada entrada neste dicionário deve possuir um identificador único que será associado ao módulo de um sub-gerenciador. Na Figura 42 temos dois sub-gerenciadores: ‘webservice’ para tratamento de serviços web e ‘hadoop’ para tratamento de tarefas de *big data*. Cada

sub-gerenciador possui por sua vez um outro dicionário associado.

O sub-gerenciador 'webservice' possui os campos 'module', 'prefix' e 'services'. O campo 'module' informa o local no disco onde o código do sub-gerenciador está armazenado. Enquanto o campo 'prefix' informa qual o prefixo inicial deve ser utilizado no código fonte MiniBPEL para encaminhar operações para este sub-gerenciador. Por fim o campo 'services' é especifico do sub-gerenciador, isto é, será utilizado apenas por ele para executar suas funções. Neste caso o campo 'services' informa ao sub-gerenciador dicionários de serviços web contendo o endereço do arquivo WSDL [19] associado a aquele serviço, um cliente para o serviço web é criado com o módulo SOAPpy [82] e com a ajuda deste cliente operações poderão ser chamadas pela máquina MiniBPEL-AM.

Enquanto o sub-gerenciador 'hadoop' possui os campos 'module', 'prefix' e 'clusters'. Os campos 'prefix' e 'clusters' tem a mesma função mostrada no sub-gerenciador de serviços web. Já o campo 'clusters' é especifico deste sub-gerenciador, tendo como função informar os dados de acesso para conexão via SSH (*Secure Shell*) ao *cluster* Hadoop e a lista de operações permitidas para execução.

Os campos 'module' e 'prefix' são utilizados pelo *gerenciador de operações* e portanto são obrigatórios, enquanto os demais campos são utilizados apenas pelos sub-gerenciadores.

Com base no arquivo de configurações o *gerenciador de operações* sabe para qual sub-gerenciador deve enviar a chamada de operação que está no *buffer* de saída da máquina MiniBPEL. Cada sub-gerenciador toma as ações necessárias para a chamada da operação e caso a operação tenha sido concluída o resultado é escrito no *buffer* de entrada da máquina MiniBPEL. Caso algum erro tenha sido encontrado durante a execução da operação o *gerenciador de operações* envia uma mensagem para a máquina MiniBPEL-AM encerrar a execução do fluxo de trabalho.

O *gerenciador de operações* neste trabalho possui implementados os sub-gerenciadores para tratamento de serviços web, operações de *big data* e funções definidas pelo usuário. Um arquivo de configurações utilizando todos os três sub-gerenciadores implementados pode ser visto na Figura 63 do Apêndice A.

O sub-gerenciador que trata dos serviços web permite a chamada de métodos de qualquer serviço web, desde que seja informado o arquivo WSDL correspondente ao serviço web. O sub-gerenciador cria um serviço web genérico com base nas informações do arquivo WSDL e permite a chamada das operações deste serviço. O sub-gerenciador de operações

de *big data* permite a chamada de tarefas do Hadoop. Para isto é necessário informar uma conexão SSH para uma das máquinas onde o Hadoop está instalado. O sub-gerenciador de *big data* permite a chamada de operações de HDFS, MapReduce e Pig Latin. Existe ainda um sub-gerenciador que permite a utilização de funções definidas pelo usuário. Para isto este sub-gerenciador precisa apenas da localização do arquivo que contem as funções e da lista de funções que serão utilizadas na composição.

O *gerenciador de operações* foi implementado desta forma visando favorecer a extensibilidade, isto é, para facilitar a criação de novos sub-gerenciadores definidos pelo usuário, sendo este um dos objetivos principais do nosso trabalho. Caso seja necessário utilizar uma operação não suportada pelos sub-gerenciadores já implementados, o usuário pode implementar seu próprio sub-gerenciador e informar seu prefixo e localização no arquivo de configurações. Por exemplo, um sub-gerenciador poderia ser implementado para realizar uma consulta SQL (*Structured Query Language*) em um banco de dados relacional durante a execução de um fluxo de trabalho.

6 Estudos de Casos

Neste capítulo serão apresentados três estudos de caso para demonstrar o funcionamento da máquina MiniBPEL-AM em diversos cenários. Para cada estudo de caso, será feita uma composição utilizando os principais construtores da linguagem MiniBPEL. O primeiro estudo de caso abordará composições apenas no domínio dos serviços web. O segundo estudo de caso criará uma composição no domínio de tarefas de *big data*. Enquanto o último estudo de caso criará uma composição que utiliza operações de serviços web, *big data* e funções definidas pelo usuário.

6.1 Estudo de Caso 1

Neste primeiro estudo de caso propõe-se a criação de uma composição para obter a média da temperatura de três cidades (Teresina, Natal e Fortaleza). Caso a temperatura média seja maior que 25°C ou menor que 10°C, uma mensagem é enviada para o Twitter alertando que está muito quente ou muito frio. A verificação de temperatura e postagem no Twitter é feita a cada minuto.

Para a realização desta composição foram utilizados os seguintes serviços web e suas operações:

- *Delay*: disponibiliza a operação *waitTime*, que espera a quantidade de minutos passada por parâmetro antes de encerrar sua execução;
- *Storage*: permite utilizar as operações *initVar*, *load* e *store* e fica responsável por armazenar variáveis e seus valores. *initVar* cria uma variável vazia persistente, *load* busca o valor de uma variável persistente e *store* armazena um valor em uma variável persistente;
- *Twitter*: permite o envio de mensagens para um perfil registrado na rede social *Twitter*, utilizando a operação *tweet*;

- *Weather*: possui a operação *getTemp*, que permite obter a temperatura de uma cidade, recebendo como parâmetros o nome da cidade e o país ao qual ela pertence.

Foi utilizado ainda o arquivo local de configurações apresentado na Figura 55 do Apêndice A. No arquivo de configurações é informado que será utilizado o sub-gerenciador de serviços web. É informado que será utilizado o prefixo 'ws' para a chamada de serviços web e os serviços web *delay*, *storage*, *twitter* e *weather* são listados, informando seus respectivos arquivos WSDL e as operações de cada serviço que serão utilizadas.

Um programa MiniBPEL ilustrado na Figura 43 foi escrito para executar o estudo de caso proposto.

```
ws.storage.initVar("media";);
while true do (
    ( ws.weather.getTemp("teresina","brazil"; T)
      || ws.weather.getTemp("natal","brazil"; N)
      || ws.weather.getTemp("fortaleza","brazil"; F)
    );
    ws.storage.store("media", (T+N+F)/3; );
    ws.delay.waitTime(2;)
)||
while true do (
    ws.storage.load("media"; MEDIA);
    (if ((MEDIA < 10) or (MEDIA > 25)) then
        ([MEDIA < 10] ws.twitter.tweet(MEDIA+": a temperatura esta muito
    baixa";)
        + [MEDIA > 25] ws.twitter.tweet(MEDIA+": a temperatura esta muito
     alta";))
    );
    ws.delay.waitTime(1;)
)
```

Figura 43: Código fonte MiniBPEL para o estudo de caso 1.

Duas estruturas de repetição *while* são utilizadas, uma para calcular a média das temperaturas e outra para publicar alertas caso a média esteja fora de um intervalo predefinido. A primeira linha do programa cuida de criar uma variável persistente chamada "media". Esta variável persistente será utilizada pela primeira estrutura de repetição para guardar o valor da média e será verificada pela segunda estrutura de repetição para publicação deste valor.

A primeira linha do corpo do primeiro *while* executa a chamada da operação *getTemp* em paralelo, para obter a temperatura das cidades brasileiras de Teresina, Natal e Fortaleza. Como mostrado no código da Figura 43, a operação *getTemp* possui dois prefixos. O prefixo inicial indicando que a operação deve ser tratada pelo gerenciador de serviços web

e o prefixo intermediário informando qual dos serviços cadastrados no arquivo de configurações possui a operação que está sendo executada. Observe que as operações podem receber listas de parâmetros de entrada separados por vírgula e retornar listas de variáveis de saída também separados por vírgula. Já a separação entre os parâmetros de entrada e os parâmetros de saída é feita utilizando ponto e vírgula. As listas podem ser omitidas no caso da operação não necessitar de parâmetros de entrada ou das variáveis de saída, como é o caso da última operação $waitTime(1;)$ que não precisa de uma variável de saída para ser executada.

Quando as operações $getTemp$ terminarem suas execuções, as temperaturas das três cidades serão armazenadas nas variáveis T, N e F. Em seguida a operação *store* armazena a média da temperatura das três cidades em um armazenamento persistente "media". O serviço $waitTime(1;)$ é então chamado, para que o primeiro *while* aguarde um minuto antes de realizar novas medições de temperatura.

A segunda estrutura de repetição *while* inicia chamando $waitTime(1;)$. Como as duas estruturas de repetição estão sendo executadas em paralelo, é necessário chamar $waitTime(1;)$ no início deste segundo *while* para que o primeiro *while* tenha tempo de calcular a média das temperaturas. Após esperar um minuto, a média é carregada para a variável MEDIA através da operação *load*.

A próxima operação é um comando if que verifica se o valor de MEDIA está abaixo de 10°C ou acima de 25°C. Se uma das expressões do if for verdadeira, utiliza-se uma escolha não determinística para verificar se a média das temperaturas está fora do intervalo de 10 a 25 graus Celcius. Caso a temperatura esteja abaixo de 10°C então uma mensagem é enviada pela operação *tweet* informando que o tempo está muito frio. Caso a temperatura esteja acima de 25°C então outra mensagem é enviada pela operação *tweet* informando que o tempo está muito quente.

As Figuras 56 e 57, do Apêndice A mostram respectivamente o estado inicial do grafo e o estado do grafo após a avaliação das estruturas de repetição.

Por fim, a Figura 44 mostra o resultado da execução da composição acima. Demonstrando que a máquina MiniBPEL-AM é capaz de realizar composições de serviços web e que todas as primitivas mostradas na seção 4.1 do capítulo que explica o funcionamento da máquina MiniBPEL-AM funcionam como esperado.

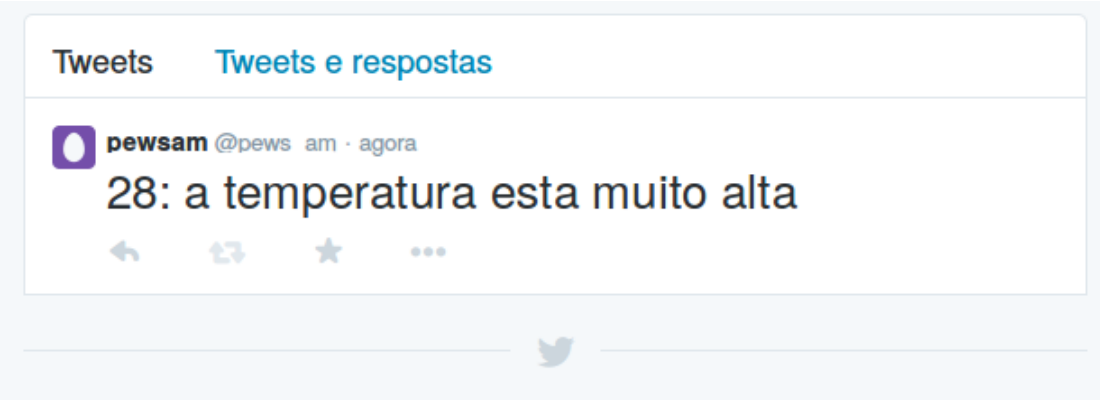


Figura 44: Execução da composição do estudo de caso 1.

6.2 Estudo de Caso 2

Para o estudo de caso 2 foram utilizadas apenas operações de *big data*, mais especificamente, operações do Hadoop. Neste estudo de caso foram coletados 3 gigabytes de dados da rede social Twitter na forma de mensagens que possuem a palavra "dilma" em seu conteúdo, durante o primeiro e segundo turno das eleições para escolha do presidente da república em 2014. Deseja-se limpar estes dados e contar quantas mensagens positivas e negativas a candidata Dilma Rousseff recebeu durante todo o período eleitoral.

As mensagens foram capturadas utilizando um *script* python, que capturava as mensagens enviadas para o Twitter que continham a palavra chave "dilma". Foram criadas duas pastas, uma para armazenar as mensagens do primeiro turno (Figura 45) e para guardar as mensagens do segundo turno. Cada arquivo possui duas etiquetas: data no formato ano/mês/dia e horário no formato hora/minuto/segundo, informando a data e hora da criação daquele arquivo. A etiqueta de data serve para agrupar as mensagens que foram publicadas naquele dia, isto é, um arquivo de nome 'tweets-20140618-114213.csv' conterá as mensagens publicadas em 18/06/2014.

Como uma mensagem do Twitter possui uma grande quantidade de campos, foram extraídos apenas os campos *id*, *username*, *date*, *lang*, *geo*, *client* e *msg* de cada mensagem (Figura 46). Os campos selecionados são separados por ponto e vírgula.

O Campo *id* é o identificador único de uma mensagem do Twitter; *username* informa o apelido do usuário que está enviando a mensagem; *date* mostra a data de publicação da mensagem; *lang* tenta informar a linguagem da mensagem; *geo* tenta informar a latitude e longitude de onde a mensagem surgiu; *client* mostra de qual cliente do Twitter a mensagem foi enviada; e *msg* é o campo que possui a mensagem enviada.

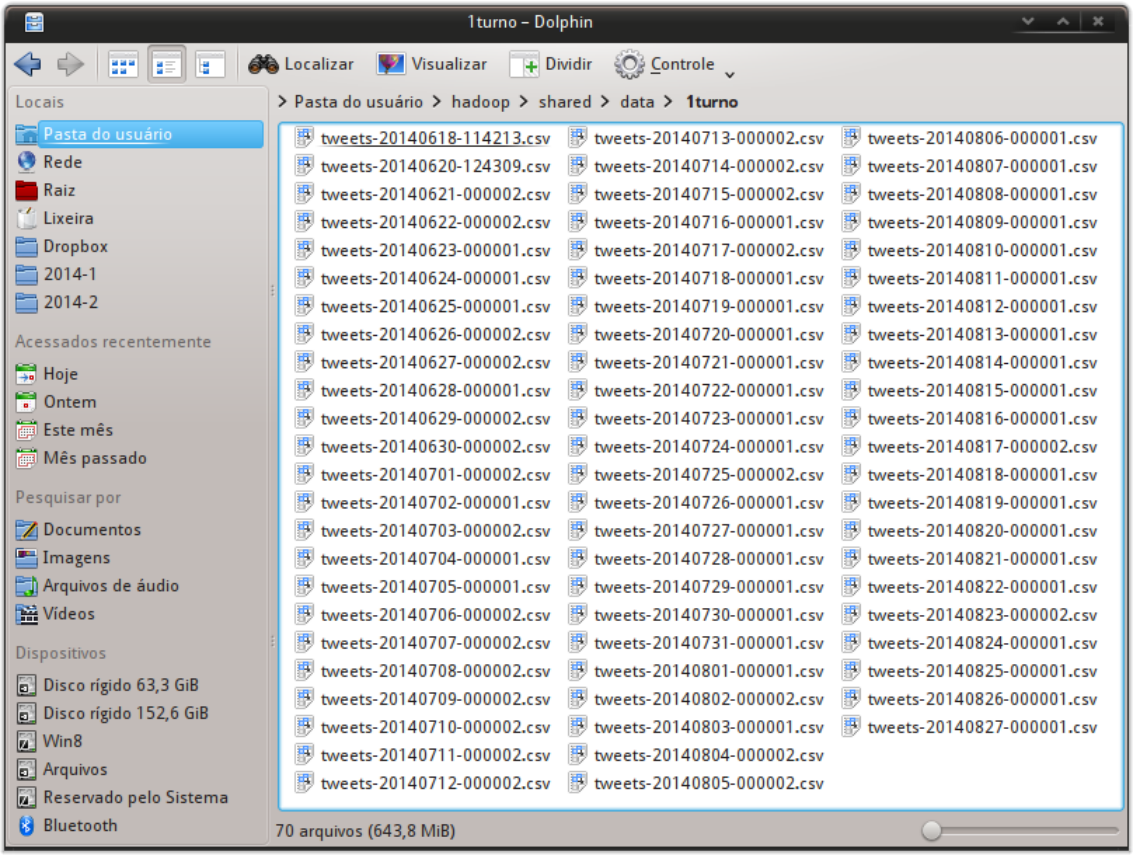


Figura 45: Mensagens extraídas da rede social Twitter.

```
1  ...
2  480013011516919808,[redacted],2014-06-20 15:43:14,pt,,Twitter Web Client,"RT
   [redacted] Nas rede sociais, a presidenta #DilmaéaFavorita dos
   internautas. [redacted]"
3  480013022396940288,[redacted],2014-06-20 15:43:17,pt,,Twitter Web
   Client,"Porque [redacted] apoia os xingamentos à Dilma: Pobre viajando de
   avião, que horror!!! [redacted]"
4  480013031351795712,[redacted],2014-06-20 15:43:19,pt,,Twitter Web
   Client,"Gente que acha PT legal pq curte ""Dilma Bolada"" mas não sabe uma
   virgula sobre Decreto 82.43 e ""Controle da Midia"". Não encham o saco!"
5  ...
```

Figura 46: Formato das mensagens extraídas da rede social Twitter.

Para execução das operações do Hadoop foi utilizada uma máquina virtual[1], disponibilizada pela Hortonworks que possui um ambiente pronto para uso de ferramentas como MapReduce, pig, hdfs, oozie, dentre outras.

O arquivo de configurações mostrado na Figura 58 do Apêndice A foi utilizado nesta composição. Neste arquivo é informado que o sub-gerenciador do Hadoop será utilizado e que o prefixo 'hd' está associado a este sub-gerenciador. Para que o sub-gerenciador funcione corretamente, pelo menos um *cluster* Hadoop deve ser informado e seu respectivo identificador. Neste estudo de caso o *cluster* é identificado como '*vm*'. Os dados para acesso via ssh são informados e o conjunto de operações permitidas para execução são descritas.

Na composição deste estudo de caso foram utilizados as seguintes operações disponibilizadas pelo sub-gerenciador Hadoop:

- ssh: executa comandos remotos em uma máquina que faz parte do *cluster* Hadoop;
- hdfs: realiza operações passadas por parâmetro no sistema de arquivos distribuído do Hadoop;
- pig: executa um *script* Pig no *cluster* Hadoop.

O programa MiniBPEL mostrado na Figura 47 foi utilizado para executar o estudo de caso 2.

```
hd.vm.ssh("/media/sf_shared/minibpel/delete_dirs.sh");
hd.vm.hdfs("mkdir /user/hue/mbpel/minibpel/input");
( hd.vm.hdfs("put /media/sf_shared/data/1turno /user/hue/mbpel/minibpel/
    input/");
  hd.vm.hdfs("put /media/sf_shared/data/2turno /user/hue/mbpel/minibpel/
    input/")
);
hd.vm.pig("/media/sf_shared/minibpel/clean.pig");
hd.vm.pig("/media/sf_shared/minibpel/sentiment.pig")
```

Figura 47: Código fonte MiniBPEL para o estudo de caso 2.

O objetivo desta composição é realizar uma análise de sentimento, com base em parâmetros positivos e negativos nas mensagens retiradas do Twitter. As duas primeiras operações da composição servem para preparar a estrutura de diretórios para o envio de dados, limpeza e execução da análise. Na primeira operação 'hd.vm.ssh' o *script* da Figura 59, Apêndice A, é executado para remover diretórios com mensagens e resultados

[1]http://br.hortonworks.com/products/hortonworks-sandbox/

de execuções anteriores. Na segunda linha uma operação hdfs é executada para criar um diretório 'input' no sistema de arquivos distribuído do Hadoop. Logo em seguida duas operações hdfs são executadas em paralelo para coletar as mensagens que estão armazenadas no disco local da máquina virtual e enviá-las para a pasta input criada anteriormente do sistema de arquivos distribuído do Hadoop. Ao término do envio das mensagens, o *script* de limpeza da Figura 60, Apêndice A, escrito em Pig Latin é executado no Hadoop. Este *script* Pig lê as mensagens armazenadas na pasta input, cria um novo conjunto de mensagens na pasta '/user/hue/mbpel/minibpel/output_clean_tweets' no sistema de arquivos distribuído, salvando apenas os campos *id*, *username*, *date* e *msg* de cada mensagem. Por fim o *script* Pig da Figura 61, Apêndice A, é executado para realizar a análise de sentimento.

O *script* 'sentiment.pig' lê as mensagens salvas na fase de limpeza e carrega o dicionário da Figura 62, Apêndice A, de palavras positivas e negativas. Para cada mensagem é atribuída uma pontuação com base na quantidade de palavras positivas ou negativas que a mensagem possui. São contadas quantas mensagens foram classificadas como positivas ou negativas. O resultado da contagem é armazenado na pasta '/user/hue/mbpel/minibpel/output_tweets_sentiment' do sistema de arquivos distribuído.

O dicionário da Figura 62, Apêndice A, foi construído a partir de uma análise manual das palavras mais frequentes do conjunto de mensagens analisado. Foi executada uma tarefa MapReduce para a contagem das palavras mais frequentes e deste resultado foram escolhidas 20 palavras positivas e 20 negativas.

O resultado da execução da composição pode ser visto na Figura 48. Observa-se que a candidata Dilma, com base no dicionário de palavras positivas e negativas utilizado, recebeu um total de 705.246 mensagens positivas e 349.410 mensagens negativas.

```
[root@sandbox ~]# hdfs dfs -cat /user/hue/mbpel/minibpel/output_tweets_sentiment/part-r-00000
705246  349410
[root@sandbox ~]#
```

Figura 48: Resultado da execução da composição do estudo de caso 2.

A execução do estudo de caso demonstra que a máquina MiniBPEL-AM é capaz de executar composições com tarefas de *big data*.

6.3 Estudo de Caso 3

Neste terceiro estudo de caso utilizou-se operações mistas, isto é, foi criado um fluxo de trabalho com chamadas de: serviços web, tarefas do Hadoop e funções definidas pelo usuário. Foi utilizado o mesmo conjunto de dados apresentado no estudo de caso 2 (6.2), ou seja, 3 gigabytes de mensagens da rede social Twitter que possuíam em seu texto a palavra "dilma".

Desta vez procura-se saber a quantidade de mensagens positivas ou negativas que a candidata Dilma Rousseff recebeu por dia.

No fluxo de trabalho deste estudo de caso foram utilizadas as seguintes operações:

- Serviços web
  - Delay: permite utilizar a operação *waitTime*, que espera uma quantidade de minutos antes de encerrar sua execução;
  - Twitter: disponibiliza o uso da operação *tweet*, que envia mensagens para um perfil na rede social Twitter;
- Hadoop
  - ssh: executa comandos remotos em uma máquina conectada ao *cluster* Hadoop;
  - hdfs: realiza operações no sistema de arquivos distribuído do Hadoop;
  - pig: executa um *script* Pig no *cluster* Hadoop;
- Funções Definidas pelo Usuário
  - Tempo: este módulo permite utilizar as operações: *nextDate* que retorna o dia seguinte ao dia passado por parâmetro (entradas e saídas são passadas no formato ANO-MES-DIA) e $defragDate$ que extrai o ano e o mês de uma data que está no formato ANO-MES-DIA;
  - LocalStorage: disponibiliza o uso da operação $filterDataSource$, que busca no diretório onde as mensagens do Twitter estão armazenadas, apenas as mensagens da data especificada no parâmetro, salvando estas mensagens no diretório "/media/sf_shared/ec03/input/" para posterior envio ao *cluster* Hadoop;
  - Tratamento: permite utilizar a operação $extractValues$, que retorna dois números de uma string no formato "NUMERO <TABULAÇÃO> NUMERO";

– Couch: este módulo permite chamar a operação *save*, que guarda a data, número de mensagens positivas e número de mensagens negativas no banco de dados não relacional CouchDB [83]

O programa MiniBPEL mostrado na Figura 49 foi utilizado para executar o estudo de caso 3 e seu arquivo de configuração pode ser visto na Figura 63 do Apêndice A.

```
my.tempo.nextDate("2014-09-07";DATE);
my.tempo.defragDate(DATE; YEAR, MONTH);
while (YEAR==2014) and (MONTH >= 06 and MONTH < 11) do (
  hd.vm.ssh("/media/sf_shared/ec03/prepare_dirs.sh";);
  my.localstorage.filterDataSource(DATE;);
  hd.vm.hdfs("put /media/sf_shared/ec03/input /user/hue/mbpel/ec03/";);
  hd.vm.pig("/media/sf_shared/ec03/clean.pig";);
  hd.vm.pig("/media/sf_shared/ec03/sentiment.pig";);
  hd.vm.hdfs("cat /user/hue/mbpel/ec03/output_sentiment/part-r-00000";CAT);
  my.tratamento.extractValues(CAT; F, A);
  (if ((F > 100) and (A > 100)) then
      ([F-A > 0]ws.twitter.tweet(DATE+": Public stay friendly to product!";)
        +[F-A < 0]ws.twitter.tweet(DATE+": Public stay against product!";)
      );
      ws.twitter.tweet(DATE + ": Friendly: " + F + ", Against: " + A;)
       || my.couch.save(DATE, F, A;)
  );
  ( my.tempo.nextDate(DATE; DATE);
    my.tempo.defragDate(DATE; YEAR, MONTH)
  ) || ws.delay.waitTime(1;)
)
```

Figura 49: Código fonte MiniBPEL para o estudo de caso 3.

Este fluxo de trabalho tem por objetivo realizar uma análise de sentimento, com base em parâmetros positivos e negativos nas mensagens retiradas do Twitter. Porém esta análise será feita por dia, diferentemente da análise do estudo de caso 2, que realizava a análise de todas as mensagens. Os resultados de cada dia são então enviados para o Twitter e salvos no CouchDB para consultas posteriores.

A primeira operação do fluxo de trabalho, *my.tempo.nextDate* serve para inicializar a variável DATE com uma data no formato ANO-MES-DIA. Para esta chamada o resultado da variável DATE será "2014-06-21". Logo em seguida a operação *my.tempo.defragDate* extrai da data o ano e o mês da variável DATE para as variáveis YEAR e MONTH respectivamente.

A variável DATE é utilizada para busca de arquivos e publicação de resultados, enquanto as variáveis YEAR e MONTH são utilizadas para limitar o período de análise.

Cada iteração da estrutura de repetição *while* irá realizar a análise de sentimento

para uma dia especifico, dentro do intervalo de 06/2014 até 10/2014. A estrutura de repetição é encerrada quando o dia 01/11/2014 tentar ser analisado. Logo após o início do *while* a operação *hd.vm.ssh* é aplicada, executando o *script* SSH 64 que cuida de remover mensagens e pastas resultantes de execuções anteriores desta composição.

A próxima operação $my.localstorage.filterDataSource1$ busca nos arquivos de mensagens do twitter quais arquivos possuem a data especificada por DATE e trata de mover estes arquivos para o diretório "/media/sf_shared/ec03/input/".

As próximas quatro operações tratarão de: (i) enviar as mensagens da pasta "/media/sf_-shared/ec03/input/" para o sistema de arquivos distribuídos do hadoop; (ii) limpar as mensagens; (iii) executar a análise de sentimento; e (iv) carregar o resultado da análise na variável CAT.

A operação $my.tratamento.extractValues$ serve para tratar o resultado da análise de sentimento (CAT), extraindo os valores de mensagens positivas (F) e negativas (A).

Um condicional if é utilizado para verificar se os valores de F e A estão acima de 100, isto é, se há pelo menos 200 mensagens analisadas. Caso haja mais de 200 mensagens, verifica-se através de uma escolha se há um número maior de mensagens positivas ou negativas. Se houverem mais mensagens positivas ($F - A > 0$), uma mensagem é enviada para o Twitter informando que o público está favorável. Caso contrário ($F - A < 0$), uma mensagem é enviada informando que naquele dia o público está contra.

Após a escolha não determinística ter sido executada, uma mensagem adicional é enviada ao Twitter informando os números de mensagens positivas e negativas. Em paralelo, os dados são enviados para o CouchDB através da chamada da operação *my.couch.save*, uma vez que estas duas operações não são interdependentes.

Por fim o fluxo de trabalho executa as operações $my.tempo.nextDate$ e $my.tempo.defragDate$ em paralelo com a operação $ws.delay.waitTime$. Isto é, as operações $my.tempo.nextDate$ e $my.tempo.defragDate$ atualizam os valores das variáveis DATE, YEAR e MONTH, enquanto é feita uma chamada ao serviço $ws.delay.waitTime$ que faz o fluxo de trabalho aguardar um minuto antes de iniciar uma nova iteração, realizando todo o processo novamente para o dia subsequente.

O resultado da execução da composição pode ser visto na Figura 50

As Figuras 67 e 68, do Apêndice A mostram respectivamente o estado inicial do grafo e o estado do grafo após a avaliação da estrutura de repetição.

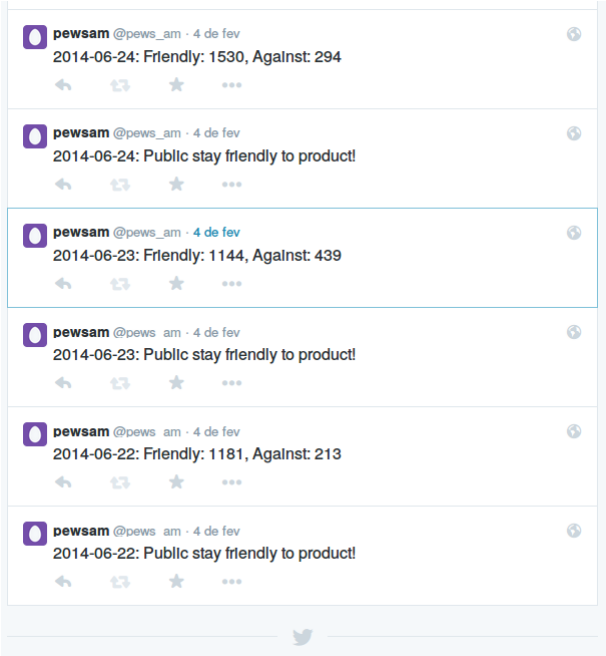


Figura 50: Resultado da execução da composição do estudo de caso 3.

A Figura 51 mostra um gráfico com a porcentagem de mensagens positivas e negativas da candidata Dilma, correspondente ao período de 22/06/2014 a 24/08/2014. Estes dados, obtidos a partir do CouchDB, mostram que a quantidade de mensagens positivas se mantêm acima de 50% durante quase todo o período. A quantidade de mensagens positivas só fica significativamente baixa em duas ocasiões: no período entre 06/07/2014 a 13/07/2014, correspondente a derrota da seleção brasileira para a Alemanha; e no período de 10/08/2014 a 17/08/2014, correspondente a morte do candidato Eduardo Campos.

A máquina MiniBPEL-AM mostrou ser capaz de executar fluxos de trabalho mistos envolvendo chamadas de serviços web, tarefas do Hadoop e funções definidas pelo usuário de forma simples e eficiente. Facilitando a criação de fluxos de trabalho mais complexos e de forma simplificada, uma vez que o fluxo de trabalho foi especificado utilizando poucas linhas de código.

6.4 Análise dos Estudos de Caso

No estudo de caso 1 será feita a medição de alguns parâmetros estáticos e dinâmicos obtidos a partir do código da máquina MiniBPEL durante a execução da composição. Estes parâmetros serão utilizados para avaliação da máquina MiniBPEL-AM com relação as máquinas PEWS-AM e PEWS-RT.

No estudo de caso 2 a máquina MiniBPEL-AM será comparada com o sistema de

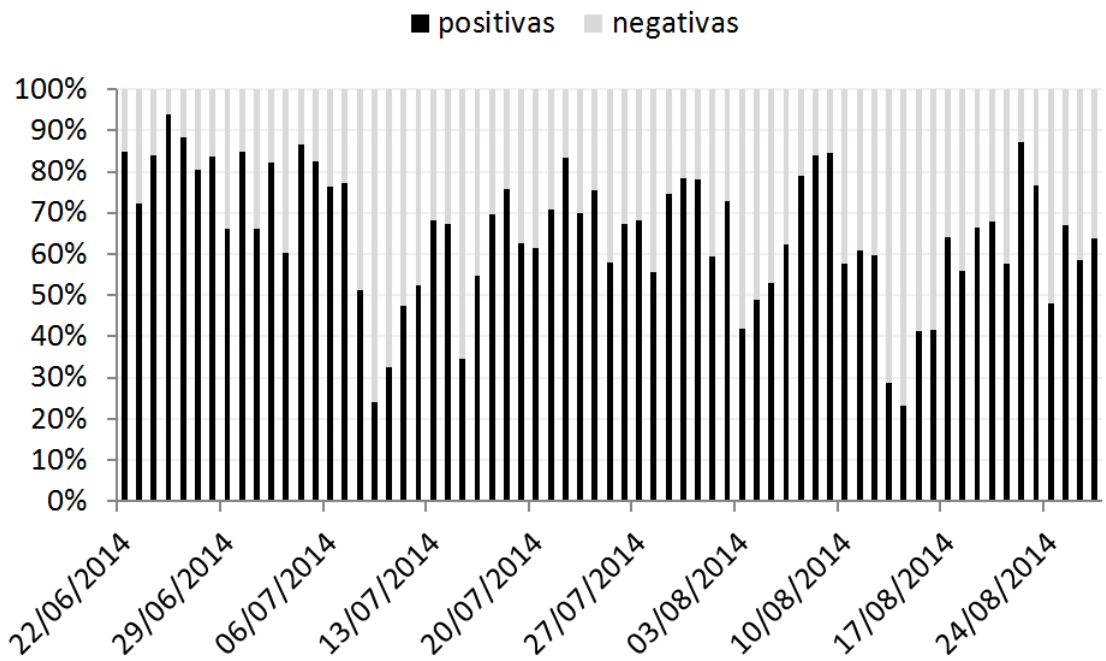


Figura 51: Gráfico de porcentagem de mensagens positivas e negativas do estudo de caso 3.

gerenciamento de fluxos de trabalho Oozie, que permite a criação de composições para Hadoop.

A análise do estudo de caso 3 irá mostrar as estatísticas estáticas e dinâmicas obtidas a partir do código da máquina MiniBPEL-AM durante a execução do fluxo de trabalho.

6.4.1 Estudo de Caso 1

Para a análise do estudo de caso 1, na máquina MiniBPEL-AM foram feitas duas análises: (i) uma análise dinâmica de parâmetros do grafo gerado pela máquina; (ii) uma análise do código de alguns parâmetros estáticos relacionados ao código fonte da implementação.

Nesta primeira análise a máquina MiniBPEL-AM foi comparada com a máquina PEWS-AM. Como mostrado no Capítulo 3.1, a máquina PEWS-AM propõe um modelo abstrato para a execução de composições de serviços web em uma variante da linguagem PEWS. A máquina PEWS-AM possui um conjunto de regras de tradução que transformam programas escritos em PEWS para grafos que são modificados pela máquina PEWS-AM, utilizando um conjunto de regras de redução. A aplicação das regras de redução corresponde à execução da composição.

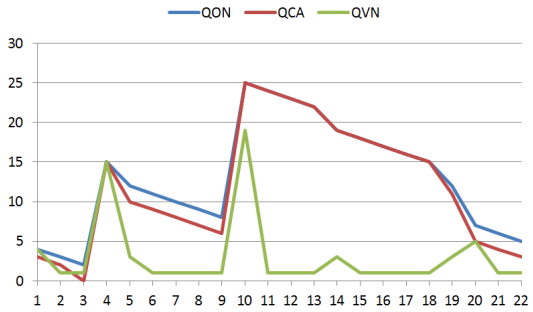
Para a análise dinâmica do grafo gerado pela máquina foram escolhidos os seguintes

parâmetros de comparação:

- QON: Quantidade de nodos do grafo que são alcançáveis a partir das raízes;
- QCA: Quantidade de arestas de controle do grafo que está sendo reduzido;
- QVN: Quantidade de nodos visitados representa o número de vértices que a máquina manipula durante uma redução.

Os valores destes parâmetros foram obtidos a partir da execução do estudo de caso 1 apresentado na seção 6.1 pelas máquinas MiniBPEL-AM e PEWS-AM. Como pode ser visto no código da Figura 43, este exemplo executa duas estruturas de repetição em paralelo indefinidamente. Como o estudo de caso 1 é executado indefinidamente, os dados foram coletados até a conclusão de cada estrutura de repetição. A Figura 52 mostra os valores obtidos dos parâmetros escolhidos para as duas máquinas de redução. As colunas destas figuras representam a quantidade de reduções executadas pelas máquinas.

(a) MiniBPEL-AM



(b) PEWS-AM

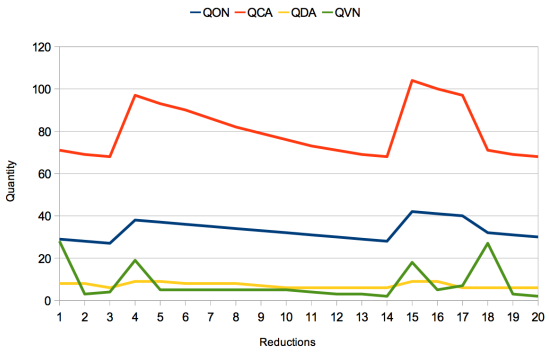


Figura 52: Análise dinâmica dos grafos gerados no estudo de caso 1.

A medida que os passos de redução são aplicados nas duas máquinas, a quantidade de nodos e arestas decresce, ocasionado pela aplicação das regras de chamada e retorno de operações das duas máquinas. Para as duas máquinas só há aumento no número de nodos

(QON) quando estruturas de repetição são avaliadas, o que ocorre nos passos de redução 4 e 15 na máquina PEWS-AM e nos passos de execução 4 e 10 da máquina MiniBPEL-AM. Quando estruturas de repetição são avaliadas pelas duas máquinas, o nodo da estrutura é substituído pelo grafo correspondente ao seu corpo, aumentando a quantidade de nodos (QON) do grafo, a quantidade de arestas de controle (QCA) e a quantidade de nodos visitados (QVN).

A diferença entre o número de passos de redução entre as máquinas avaliadas na Figura 52 se deve ao modo como as duas máquinas realizam a redução de suas estruturas de repetição. A máquina MiniBPEL-AM utiliza um passo de redução para transformar o grafo de um programa "$while\ E\ do\ P$" em um grafo "$if\ E\ then\ (P; while\ E\ do\ P)$", e outro passo de redução para avaliar o condicional if. Isto é, a máquina MiniBPEL-AM precisa utilizar dois passos de redução para começar a executar o corpo de uma estrutura de repetição, enquanto a máquina PEWS-AM realiza esta tarefa com apenas um passo de redução.

No passo de redução inicial todos os nodos dos grafos são visitados afim de identificar os nodos raízes. O número de nodos iniciais é menor na máquina MiniBPEL-AM quando comparado com a máquina PEWS-AM porque a estrutura de repetição utilizada pela máquina MiniBPEL-AM (*while*) não é expandida até o momento de sua avaliação. Enquanto a estrutura de repetição da máquina PEWS-AM (*unfolding*/*unfold*) se mantêm expandida mesmo que não venha a ser avaliada. Esta característica faz com que grafos com estruturas de repetição tenham uma quantidade menor de nodos (QON) na máquina MiniBPEL-AM quando comparada com a máquina PEWS-AM.

A linha QDA mostrada na Figura 52b é referente a quantidade de arestas de dependência de dados. As duas máquinas possuem arestas de controle, no entanto só a máquina PEWS-AM possui arestas de dependência de dados. Portanto, só será comentado a diferença entre a quantidade de arestas de controle.

A máquina MiniBPEL-AM possui uma quantidade menor de arestas de controle (QCA) quando comparada com a máquina PEWS-AM. Esta diferença se deve principalmente à regra de tradução das composições sequenciais de cada máquina. Segundo a subseção 3.1.3.3, em uma composição $P_1; P_2$, a máquina PEWS-AM cria uma aresta de controle de cada vértice de um programa P_1 para os vértices raízes do programa P_2. Este problema se agrava se a quantidade de vértices do programa P_1 for grande. Para esta mesma composição a máquina MiniBPEL-AM ao aplicar sua regra (4.3.3) cria arestas de controle apenas dos vértices com $outdegree = 0$ do programa P_1 para os vértices raízes

do programa P_2. Diminuindo assim, a quantidade de arestas de controle de dados criadas e mantendo o controle de execução do grafo.

A quantidade de nodos visitados (QVN) nas duas máquinas possuem comportamento similar, havendo aumento quando as regras das estruturas de repetição das duas máquinas são aplicadas.

Com relação aos parâmetros dinâmicos relacionados ao grafo gerado pelas máquinas, pode-se perceber que os grafos gerados pela máquina MiniBPEL-AM possuem menos nodos e arestas para a representação de um mesmo programa quando comparado com a máquina PEWS-AM.

Para a comparação de parâmetros de análise do código, foram comparadas as máquinas MiniBPEL-AM, PEWS-AM e PEWS-RT [84]. Esta ultima é a implementação de um sistema de tempo de execução escrito em C# para a linguagem PEWS, que faz uso da infraestrutura do *Microsoft Workflow Foundation* [85].

Na análise do código foram escolhidos como parâmetros de comparação [86]:

- LDC: Linhas de código, que conta a quantidade de linhas de código geradas pelos compiladores das máquinas para realizar a composição, sem considerar linhas em branco e comentários;
- QMC: Quantidade de métodos criados, que informa quantos métodos foram criados para execução da composição;
- NDA: Número de atributos, que conta a quantidade de atributos criados para execução da composição;
- AEO: Acoplamento entre objetos, que conta a quantidade de classes externas que uma classe está utilizando;
- QCF: Quantidade de chamadas de funções, que informa a quantidade de vezes que os métodos foram chamados.

A Tabela 1 mostra os valores destes parâmetros de comparação para o estudo de caso 1 executado pela máquina MiniBPEL-AM em comparação com o mesmo estudo de caso executado pelas máquinas PEWS-AM e PEWS-RT.

A tabela acima mostra que a máquina implementada neste trabalho obteve melhores resultados para quase todas as métricas quando comparada com as demais máquinas, para o estudo de caso 1.

Parâmetros	MiniBPEL-AM	PEWS-AM	PEWS-RT
LDC	345	617	670
QMC	29	42	126
NDA	15	102	69
AEO	13	30	119
QCF	332	1730	49

Tabela 1: Resultados dos parâmetros de comparação para o estudo de caso 1.

Com relação à quantidade de linhas de código (LDC), percebe-se que a máquina MiniBPEL-AM possui cerca de 50% menos linhas de código quando comparada com PEWS-AM e PEWS-RT. A quantidade reduzida de linhas se deve à diferença nas linguagens utilizadas, uma vez que programas escritos em Python tendem a ter uma quantidade de linhas menor quando comparados com um programa que executa as mesmas operações escrito em Java ou C#.

A quantidade de métodos criados (QMC), o número de atributos (NDA) e o acoplamento entre objetos (AEO) foi menor na máquina MiniBPEL-AM em comparação com PEWS-AM e PEWS-RT. A máquina MiniBPEL-AM obteve melhores resultados quando comparada com a máquina PEWS-AM nestes últimos parâmetros porque PEWS-AM optou por construir uma estrutura de grafo onde cada nodo é um objeto, enquanto a máquina MiniBPEL-AM optou por utilizar uma estrutura de grafo mais simples, onde cada nodo é uma tupla com nome e atributos.

A máquina MiniBPEL-AM possui uma quantidade de chamadas de funções (QCF) muito menor em relação a máquina PEWS-AM, no entanto maior quando comparada com PEWS-RT. As máquinas PEWS-AM e MiniBPEL-AM tendem a ter uma quantidade maior de chamadas de funções por conta de seu modo de funcionamento, onde as duas máquinas aplicam sucessivas regras de redução até que novas regras não possam mais ser executadas. O número de chamadas de função da máquina MiniBPEL-AM é considerado alto, porque o *núcleo de redução* utiliza uma espera ociosa com temporizador, que faz com que a máquina espere (ou seja, pare de realizar chamadas de funções) por um segundo antes de procurar outro vértice para ser avaliado. Este tempo geralmente é suficiente para o *gerenciador de operações* poder realizar a chamada do serviço web e retornar com um resultado. Mesmo com um temporizador, a espera ociosa ainda traz muitos gastos com chamadas de funções desnecessárias. Espera-se substituir este mecanismo por uma espera mais eficiente (semáforos ou *locks* [81]) em versões futuras da máquina MiniBPEL-AM.

Em resumo, a máquina MiniBPEL-AM obteve melhores resultados para as métricas avaliadas por conta da estrutura utilizada para representação de grafos e por possuir

nodos mais simples. A implementação da máquina MiniBPEL-AM em Python também trouxe benefícios, uma vez que os códigos escritos nesta linguagem tendem a ser menores e mais compactos. A máquina proposta obteve melhores resultados para todas as métricas tetadas, quando comparada com a máquina PEWS-AM. Já na comparação com PEWS-RT, só não obteve melhores quantidades de chamadas de função.

Apesar de não listado como parâmetro, vale lembrar que PEWS-AM e PEWS-RT executam apenas fluxos de trabalho envolvendo serviços web, enquanto a máquina MiniBPEL-AM permite a execução de outras operações além dos serviços web.

6.4.2 Estudo de Caso 2

O sistema de gerenciamento de fluxos de trabalho Oozie [51] foi escolhido para ser comparado com a máquina MiniBPEL-AM no estudo de caso 2. Oozie foi escolhido por permitir a criação de composições de tarefas do Hadoop, utilizar fluxos de trabalho para elaborar a composição, permitir a utilização de muitas operações do Hadoop (hdfs, map-reduce, pig,...) e por já vir instalado na máquina virtual da Hortonworks.

O fluxo de trabalho criado pelo Oozie para o estudo de caso 2 pode ser visto na Lista A.1 do Apêndice A.

Como não foi possível obter os parâmetros utilizados na seção 6.4.1 anterior, foram escolhidos para comparação os parâmetros mostrados na Tabela 2. Os resultados destes parâmetros para o Oozie, obtidos das fontes [36, 87, 88] foram comparados com os resultados da máquina MiniBPEL-AM.

Parâmetros	**MiniBPEL-AM**	**Oozie**
Requer um Servlet/container JSP	Não	Sim
Acompanhamento da execução do fluxo de trabalho	Console	Página web
Agendamento da execução de fluxos de trabalho	Não	Sim
Modelo de execução	Linha de Comando	Daemon
Tarefas Hadoop	5+	5
Suporte a estruturas condicionais	+ e if	conditional
Suporte a estruturas de repetição	While	Não
Linguagem de descrição do fluxo de trabalho	MiniBPEL	XML
Número de linhas de código do programa fonte	Poucas	Muitas
Extensibilidade	Muita	Pouca

Tabela 2: Parâmetros de comparação para o estudo de caso 2.

Oozie funciona como um serviço instalado no *cluster* Hadoop. Por ser implementado como uma aplicação web Java, Oozie **Requer um Servlet/container JSP** para ser

executado. Como MiniBPEL-AM foi implementada como uma ferramenta de linha de comando, não há a necessidade de se utilizar Servlets ou containers JSP.

Por conta desta característica Oozie pode ser considerada uma ferramenta para criação de fluxos de trabalho do lado do servidor, enquanto MiniBPEL-AM é uma ferramenta do lado do cliente.

As duas ferramentas disponibilizam meios de acompanhar o andamento da execução do fluxo de trabalho. Este acompanhamento serve para que o usuário possa saber se o fluxo de trabalho está sendo executado como esperado. O modo de **Acompanhamento da execução do fluxo de trabalho** é o próprio console na máquina MiniBPEL-AM, equanto o Oozie pode utilizar tanto o console quanto a interface web.

Oozie permite o **Agendamento da execução de fluxos de trabalho** por meio de *coordinators*. MiniBPEL-AM não possui uma primitiva para agendamento de fluxos de trabalho, porém este comportamento pode ser facilmente implementado pelo usuário através da criação de uma função definida por ele.

O **Modelo de execução** de MiniBPEL-AM é via linha de comando, enquanto Oozie é executado como um *daemon*.

Oozie permite a execução das **Tarefas Hadoop**: mapreduce, pig, hdfs[2], hive e scoop. Já MiniBPEL-AM permite a utilização através do sub-gerenciador Hadoop de tarefas mapreduce, pig, hdfs[3] e hive. A máquina MiniBPEL-AM permite ainda a implementação de tarefas que não estejam nesta lista. O conjunto de operações disponibilizado pelas duas ferramentas é suficiente para implementação da maioria dos fluxos de trabalho, uma vez que as operações mais utilizadas são normalmente mapreduce, pig e hdfs.

Com relação a primitivas de controle, Oozie tem **Suporte a estruturas condicionais**, na figura de uma ação condicional. Já MiniBPEL-AM possui duas estruturas condicionais: escolha não determinística e o condicional *if*. No caso do **Suporte a estruturas de repetição**, MiniBPEL-AM permite a utilização da primitiva *while* enquanto Oozie não possui suporte a tais estruturas. As duas ferramentas possuem estruturas condicionais, no entanto MiniBPEL-AM permite um melhor controle do fluxo de trabalho por permitir o uso de duas estruturas e possuir a estrutura de repetição *while*.

Linguagem de descrição do fluxo de trabalho se refere a qual linguagem deve ser utilizada para criar os fluxos de trabalho. Oozie utiliza XML, enquanto MiniBPEL-AM

[2]Um conjunto limitado de operações
[3]Todas as operações

utiliza a linguagem MiniBPEL definida neste trabalho. Como XML é uma linguagem que faz uso de muitas marcações, o **Número de linhas de código do programa fonte** acaba sendo maior do que em programas escritos na linguagem MiniBPEL. Por exemplo, para o estudo de caso 2, o número de linhas de código gerado para a linguagem MiniBPEL foi de 7 linhas, enquanto a mesma composição ficou com 60 linhas de código em XML para Oozie.

Por fim o parâmetro de **Extensibilidade** informa o quão fácil é implementar operações que não estão disponíveis na ferramenta. A criação de novas operações na ferramenta Oozie é mais complicada que na máquina MiniBPEL-AM. Para Oozie, caso um usuário necessite utilizar alguma operação não suportada, deve-se tentar adaptar a operação para um programa feito em java, um *script* shell ou um *script* ssh. No caso da máquina MiniBPEL-AM operações podem ser diretamente implementadas pelo usuário na forma de sub-gerenciadores escritos em linguagem python. Os sub-gerenciadores definidos pelo usuário são automaticamente importados pela máquina MiniBPEL-AM e suas operações ficam disponíveis para utilização sem a necessidade de alteração do código da máquina. Operações mais simples podem ser inseridas na máquina MiniBPEL-AM através do sub-gerenciador de funções definidas pelo usuário já incluído na máquina.

De forma geral as duas ferramentas possibilitam a criação de fluxos de trabalho para operações de *big data*. A máquina MiniBPEL-AM implementada neste trabalho em comparação com Oozie, mostrou que pode executar fluxos de trabalho com menor número de linhas no programa fonte, possui maior facilidade para criação de novas operações e um número maior de primitivas de controle do fluxo de trabalho. Para casos onde o usuário está procurando uma ferramenta para criação de fluxos de trabalho com operações de *big data*, com foco na simplicidade de implementação, maior controle do fluxo e maior extensibilidade, a máquina MiniBPEL-AM é aconselhada quando comparada com Oozie.

6.4.3 Estudo de Caso 3

Serão apresentados nesta subseção os resultados da análise dinâmica de parâmetros do grafo gerado pela máquina e uma análise de parâmetros relacionados ao código fonte da implementação, medidos durante a execução do programa 49 do estudo de caso 3.

Este estudo de caso executa operações de chamadas de serviços web, operações do Hadoop e funções definidas pelo usuário. Como não foi possível executar este estudo de caso inteiramente com PEWS-AM, PEWS-RT ou Oozie, apenas será comentado os valores obtidos pelas métricas já apresentadas nas subseções anteriores.

Foram coletados os valores para os parâmetros de avaliação dinâmica do grafo gerado durante a execução do estudo de caso 3. Os valores coletados são referentes a uma execução completa da estrutura de repetição *while*. Estes valores da quantidade de nodos (QON), quantidade de arestas de controle (QCA) e quantidade de nodos visitados (QVN) podem ser observados na Figura 53.

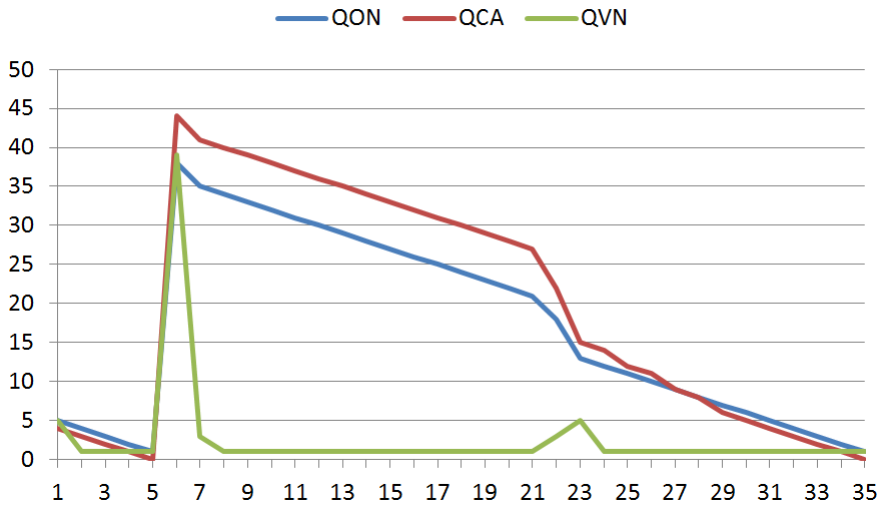


Figura 53: Análise dinâmica dos grafos gerados no estudo de caso 3.

O eixo horizontal desta figura representa o número de regras de redução aplicadas. Assim como no estudo de caso 1, o grafo gerado pela máquina MiniBPEL-AM começa com uma quantidade pequena de nodos. Uma vez que o vértice *while* é avaliado, no passo de redução 6, novos nodos e arestas são criados.

O número de nodos (QON) e arestas (QCA) vai diminuindo a medida que chamadas de operações vão sendo executadas, o que pode ser visto entre o passo de redução 7 até o passo 21 e do passo 24 até 35.

Quando a regra de escolha não determinística é aplicada, ocorre uma diminuição da quantidade de nodos (QON) e arestas (QCA) bem como um aumento na quantidade de nodos visitados (QVN), o que pode ser visto no passo de redução 7 e 23.

No passo de redução 35 o grafo está pronto para chamada da regra de redução *while*, equivalente ao passo de redução 5.

Os parâmetros dinâmicos analisados mostraram que o número de nodos (QON) e arestas (QCA) é equivalente na máquina MiniBPEL-AM, e que o número de nodos visitados (QVN) só cresce quando a máquina está aplicando regras de escolha não determinística ou a regra de redução do *while*.

A Tabela 3 mostra os valores obtidos das métricas estáticas e dinâmicas: linhas de código (LDC), quantidade de métodos criados (QMC), número de atributos (NDA), acoplamento entre objetos (AEO) e quantidade de chamadas de função QCF relacionadas ao código da máquina MiniBPEL-AM.

Parâmetros	**MiniBPEL-AM**
LDC	346
QMC	29
NDA	15
AEO	13
QCF	1722

Tabela 3: Resultados dos parâmetros de comparação para o estudo de caso 3.

As métricas de número de linhas de código (LDC), quantidade de métodos criados (QMC), número de atributos (NDA) e acoplamento entre objetos (AEO) não sofreram alteração em relação aos valores obtidos na análise do estudo de caso 1. No entanto a quantidade de chamadas de função (QCF) obteve o valor de 1722.

Este alto valor do número de chamadas de função se deve ao mecanismo de espera adotado pela máquina, que foi implementado na forma de espera ociosa com temporizador. A espera implementada no *núcleo de redução* da máquina aguarda um segundo antes de procurar um novo nodo para avaliar.

Uma chamada de operação é traduzida pela máquina MiniBPEL-AM em dois vértices: um vértice de chamada de operação e um de retorno de operação. Quando o vértice de chamada de operação é avaliado, este vértice é removido do grafo e uma chamada desta operação é enviada ao *gerenciador de operações*, via *buffer* de saída da máquina, ficando apenas o vértice de retorno de operação no grafo. Neste ponto a máquina espera um segundo e tenta avaliar um novo vértice, voltando ao início da estrutura de repetição *while* mostrado na Figura 41 do capítulo de implementação 5. Se não houver outros vértices para serem avaliados, além do vértice de retorno de operação que está esperando o retorno do *gerenciador de operações*, a máquina seguirá esperando um segundo e verificando novamente se o vértice já pode ser avaliado. Esta verificação segue até que o *gerenciador de operações* retorne uma tupla para o *buffer* de entrada da máquina. Na próxima verificação a máquina irá aplicar a regra de retorno de operação.

Esta abordagem favorece a chamada de serviços web e funções definidas pelo usuário, que possuem tempos de respostas relativamente baixos, fazendo com que a quantidade de chamada de funções seja baixo. No entanto operações mais demoradas, como operações de *big data*, fazem com que o *núcleo de reduções* realize uma grande quantidade de chamadas

de função, uma vez que ele estará verificando a cada segundo se o retorno de operação pode ser avaliado.

A Figura 54 mostra a evolução da quantidade de chamadas de função para o estudo de caso 3.

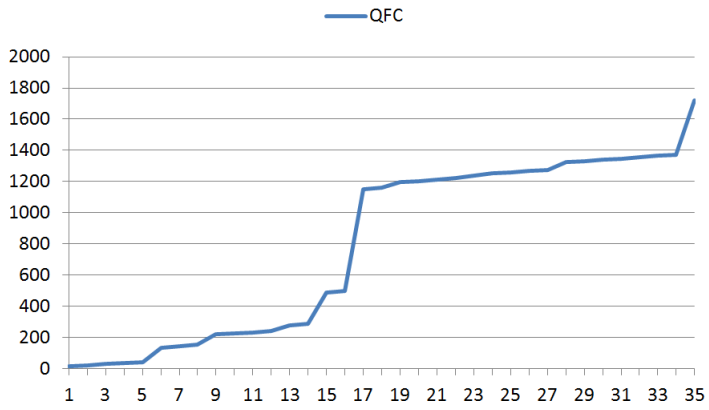


Figura 54: Quantidade de chamadas de função para o estudo de caso 3.

Como pode ser observado na Figura 54, a quantidade de chamadas de função cresce lentamente a cada passo de redução. Com exceção dos passos de redução 15 e 17 correspondentes às chamadas das operações Pig que executam os *scripts* de limpeza (*clean.pig*) e análise de sentimentos (*sentiment.pig*). O passo de redução 35 também mostra um aumento do número de chamadas de função, por conta da operação *ws.delay.waitTime* que é executada neste passo.

Quando o *script clean.pig* é executado no passo de redução 15 a máquina já registra cerca de 250 chamadas de função. Ao término da execução do *script*, a máquina está registrando cerca de 500 chamadas de operação, isto é, a máquina executou aproximadamente 250 chamadas de função enquanto esperava pelo retorno do *script clean.pig*.

O *script sentiment.pig* no passo de redução 17, gera um número aproximado de 600 chamadas de função, enquanto a operação *ws.delay.waitTime* no passo 35 gera aproximadamente 350 chamadas de função.

Espera-se resolver este problemas em versões futuras da máquina através da substituição do mecanismo de espera ociosa com temporizador por uma abordagem que faça uso de semáforos ou *locks* [81].

Como resultado foi possível observar que a máquina MiniBPEL-AM ao executar o estudo de caso 3, criou um número equivalente de nodos e arestas. A cada passo de redução este número de nodos e arestas vai sendo reduzido, sendo que operações reduzem

estes números mais lentamente e escolhas não determinísticas diminuem estes valores mais rapidamente. O número de nodos e arestas só aumenta quando uma regra *while* é aplicada.

Foi mostrado também que operações mais demoradas tendem a aumentar a quantidade de chamadas de operação, por conta do mecanismo de espera ociosa adotado pela máquina MiniBPEL-AM.

A execução do estudo de caso demonstra que a máquina é capaz de realizar fluxos de trabalhos compostos, utilizando chamadas de serviços web, operações do Hadoop e funções definidas pelo usuário.

6.5 Conclusões do Capítulo

Neste capítulo foram apresentados três estudos de caso, demonstrando o funcionamento da máquina MiniBPEL-AM em três diferentes domínios: serviços web, tarefas Hadoop e composições mistas.

No estudo de caso 1 todos os construtores definidos em 4.1 foram utilizados para a execução de um fluxo de trabalho envolvendo apenas serviços web. A máquina MiniBPEL-AM provou construir grafos mais compactos quando comparada com a máquina PEWS-AM e demonstrou que possui um código mais simples quando comparada com as máquinas PEWS-AM e PEWS-RT.

O estudo de caso 2 demonstrou que a máquina proposta é comparável ao sistema Oozie para executar fluxos de trabalho com tarefas do Hadoop. A máquina MiniBPEL-AM possui ainda algumas vantagens, tais como: utilizar menos linhas de código para descrever um fluxo de trabalho e poder ser executada localmente, isto é, fora do cluster Hadoop.

A máquina MiniBPEL-AM permitiu executar operações de diferentes domínios no estudo de caso 3. Foi possível ainda criar uma operação para comunicação com bancos de dados. Tarefas que não puderam ser executadas pelas ferramentas dos estudos de caso 1 e 2, demonstrando suas principais características: a execução de operações de múltiplos domínios em um mesmo fluxo de trabalho e a possibilidade de se criar e utilizar novas operações em um fluxo de trabalho.

No capítulo seguinte serão apresentadas as conclusões gerais do trabalho.

7 Conclusões

Neste trabalho foram apresentados os conceitos de composição de serviços web, ferramentas e linguagens utilizadas para sua criação. Linguagens como PEWS e BPEL utilizam fluxos de trabalho para representar composições de serviços, porém com o advento de novas tecnologias faz-se necessário criar composições mais complexas e que permitam utilizar outras operações além de serviços web. O principal objetivo deste trabalho foi resolver este problema.

Foi proposta a linguagem MiniBPEL, semelhante a PEWS e BPEL, e a máquina MiniBPEL-AM para sua execução. A gramática da linguagem MiniBPEL foi detalhada e definida, bem como conjuntos de regras de tradução, que transformam código escrito em MiniBPEL para grafos, e regras de redução que aplicam transformações nos grafos enquanto executam as operações definidas no fluxo de trabalho.

A máquina MiniBPEL-AM foi implementada de forma a permitir a realização de fluxos de trabalho com operações de diferentes domínios e possibilitar a implementação de novas operações sem modificar o código da própria máquina. Um *núcleo de redução* aplica sucessivas regras de redução e quando chamadas de operações precisam ser feitas, mensagens são enviadas para um *gerenciador de operações*, que se encarrega de identificar o domínio da operação e realizá-la de acordo com as necessidades do domínio. Ao concluir a operação, o *gerenciador de operações* envia uma mensagem para o *núcleo de reduções* da máquina MiniBPEL-AM, para que este possa continuar a execução do fluxo de trabalho.

Estudos de caso foram propostos para comprovar o funcionamento da linguagem e máquina proposta. A máquina MiniBPEL-AM executou todos os estudos de caso respeitando as regras de tradução e redução definidas no capítulo 4.

O estudo de caso 1 foi analisado utilizando as métricas número de linhas de código, quantidade de métodos criados, número de atributos, acoplamento entre objetos e quantidade de chamadas de funções para o código fonte utilizado para a execução. Os grafos gerados pelas execuções também foram analisados utilizando as métricas: de quantidade

de nodos, arestas e vértices visitados.

No estudo de caso 1 a máquina MiniBPEL-AM foi comparada com as máquinas PEWS-AM e PEWS-RT no domínio de execução de fluxos de trabalho envolvendo serviços web. A máquina MiniBPEL-AM, neste primeiro estudo de caso, criou grafos com menores números de nodos e arestas quando comparados com a máquina PEWS-AM. A máquina proposta obteve menores valores em quase todos os parâmetros de análise de código quando comparada com as máquinas PEWS-AM e PEWS-RT. A quantidade de chamadas de funções obtidas pela máquina MiniBPEL-AM foi maior que o de PEWS-RT e menor que o de PEWS-AM.

A máquina MiniBPEL-AM foi comparada com Oozie no estudo de caso 2, uma vez que este estudo de caso exige a chamada de operações do Hadoop. No estudo de caso 2 a máquina proposta foi comparada com a ferramenta Oozie, mostrando ser comparável a ferramentas consolidadas no mercado. A máquina MiniBPEL-AM criou um número menor de linhas de código para criação da composição do estudo de caso 2. Possui maior facilidade para criação de novas operações e tem um número maior de primitivas de controle do fluxo quando comparada com Oozie.

O estudo de caso 3 exige a utilização de operações mistas, envolvendo a chamada de serviços web, operações do Hadoop e funções definidas pelo usuário, de forma que não foi possível comparar a máquina MiniBPEL-AM com nenhuma das outras ferramentas. Foram coletadas as mesmas estatísticas mostradas no estudo de caso 1. No terceiro estudo de caso a máquina MiniBPEL-AM executou uma composição mista, contendo chamadas de serviços web, tarefas do Hadoop e funções definidas pelo usuário, demonstrando que a máquina consegue executar operações de diferentes domínios em uma mesma composição. Neste estudo de caso também foi possível observar a utilização de novas funcionalidades, como o envio de dados para o CouchDB, que não estava no leque de operações iniciais oferecidas pela máquina MiniBPEL-AM.

7.1 Principais contribuições

A revisão na gramática da linguagem PEWS, das regras de tradução, redução e arquitetura da máquina de redução de grafos PEWS-AM, permitiu criar a máquina MiniBPEL-AM: uma expansão que possibilita a implementação e utilização de novas operações e que torna possível que operações de diferentes domínios possam fazer parte de um mesmo fluxo de trabalho.

A linguagem MiniBPEL apresentada, representa linguagens de fluxo de trabalho de forma geral, como PEWS e BPEL, possuindo construtores tais como if, $while$, escolha não determinística, composição sequencial, composição em paralelo e chamada de operações. A linguagem se mostrou expressiva o suficiente para realizar fluxos de trabalho e permitiu comparações com ferramentas do domínio dos serviços web, tarefas do Hadoop e composições mistas.

A máquina MiniBPEL-AM que executa a linguagem proposta, utiliza um conjunto de regras de tradução e redução para executar fluxos de trabalho. Sua implementação mostrou que a máquina é capaz de executar composições envolvendo serviços web, tarefas do Hadoop e funções definidas pelo usuário de forma eficiente, com grafos de tamanho reduzido e com menor custo computacional. A extensibilidade é alcançada através da generalização do operador de "chamada de operações" e da utilização de sub-gerenciadores que executam as operações específicas de cada domínio.

7.2 Limitações

Durante a execução dos estudos de caso notou-se que operações que demoram mais tempo para ser completadas, causam o aumento da quantidade de chamadas de funções no *núcleo de reduções* da máquina. Este problema é causado pelo mecanismo de espera ociosa com temporizador utilizado pelo *núcleo de redução* da máquina, que faz com que a máquina verifique se a operação já foi concluída a cada segundo, causando o aumento no número de chamadas de funções. A troca do mecanismo de espera ociosa com temporizador por uma abordagem mais eficiente, como semáforos ou *locks*, resolverá este problema em versões futuras da máquina.

Outro problema durante à execução dos estudos de caso foi referente a utilização da base de dados de mensagens do Twitter. Para este trabalho foram criados dois conjuntos de dados formados por mensagens do Twitter durante o período de 06/2014 até 11/2014:

- mensagens do Twitter contendo 7 campos que possuíam em seu texto a palavra 'dilma', resultando em um conjunto de mensagens com aproximadamente 3 gigabytes de dados para todo o período analisado;
- mensagens do Twitter contendo cerca de 100 campos que possuíam em seu texto palavras como: 'eleicoes2014', 'debate2014', 'dilma', 'aecio', 'marina', dentre outras, resultando em um conjunto de aproximadamente 45 gigabytes de dados para o

primeiro turno e outros 45 gigabytes para o segundo turno.

Pretendia-se utilizar o conjunto de dados maior para executar o estudo de caso 3. Isto não foi possível porque a máquina virtual da Hortonworks, que permite o uso do Hadoop, possui um espaço de armazenamento limitado de aproximadamente 44 gigabytes, o que impossibilitou a utilização destes dados, mesmo compactados. Esta limitação deverá ser eliminada em versões futuras, as quais não deverão usar a máquina da Hortonworks.

7.3 Trabalhos Futuros

Para a atual versão da máquina MiniBPEL-AM, alguns trabalhos futuros podem ser citados para melhoria de seu funcionamento e utilização.

Implementação de uma ferramenta visual para a criação de grafos representando o fluxo de trabalho que deve ser executado. Esta ferramenta teria de transformar o grafo que representa o fluxo de trabalho em um programa MiniBPEL. Facilitando assim, a visualização do fluxo de trabalho criado.

A versão atual da máquina realiza chamadas de operações de forma síncrona. O que impede que chamadas de operações em uma via (*one way*) [19] possam ser implementadas. Para permitir que operações em uma via possam ser realizadas, é necessário implementar chamadas de operações assíncronas e um conjunto de regras de tradução especial para tratar este tipo de chamada de operações.

Outra ideia seria o aumento da integração da máquina proposta com o Hadoop, através da implementação de um escalonador de operações dentro do Hadoop. Uma vez que a versão atual da máquina apenas envia operação por operação para o Hadoop. Este escalonador permitiria o envio de uma parte do grafo (possuindo apenas operações de *big data*) para o Hadoop e cuidaria de executar as operações do fluxo de trabalho com uma maior prioridade. Isto é, as operações que fazem parte de um fluxo de trabalho deverão ser executadas antes das demais operações. Desta maneira se busca um melhor aproveitamento dos recursos do *cluster* Hadoop, enquanto se prioriza a execução de operações que fazem parte de um fluxo de trabalho.

Espera-se que este trabalho possa servir como fonte para projetos de pesquisa, ajudando a expandir o conhecimento adquirido e colaborando para a criação de trabalhos futuros.

Referências

[1] Thiagarajan Ravichandran and Marcus A Rothenberger. Software reuse strategies and component markets. *Communications of the ACM*, 46(8):109–114, 2003.

[2] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web services*. Springer, 2004.

[3] Chris Peltz. Web services orchestration and choreography. *Computer*, 36(10):46–52, 2003.

[4] Rania Khalaf, Nirmal Mukhi, and Sanjiva Weerawarana. Service-oriented composition in bpel4ws. In *WWW (Alternate Paper Tracks)*, pages 27–28, 2003.

[5] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[6] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[7] Apache Software Foundation. Apache pig, October 2013. Disponível em: <`http://pig.apache.org/`>. Acesso em 01. Abril 2014.

[8] George F Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed systems: concepts and design*. pearson education, 2005.

[9] Apache Software Foundation. Hadoop powered by, February 2014. Disponível em: <`http://wiki.apache.org/hadoop/PoweredBy`>. Acesso em 31. Março 2014.

[10] Apache Software Foundation. Giraph, January 2014. Disponível em: <`http://giraph.apache.org/`>. Acesso em 04. Junho 2014.

[11] Apache Software Foundation. Apache hive, March 2014. Disponível em: <`http://hive.apache.org/`>. Acesso em 01. Abril 2014.

[12] Paul Zikopoulos, Chris Eaton, et al. *Understanding big data: Analytics for enterprise class hadoop and streaming data*. McGraw-Hill Osborne Media, 2011.

[13] James Manyika, Michael Chui, Brad Brown, Jacques Bughin, Richard Dobbs, Charles Roxburgh, and Angela H Byers. Big data: The next frontier for innovation, competition, and productivity. 2011.

[14] Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, et al. Business process execution language for web services, 2003.

[15] Cheikh Ba, Marcos Aurélio Carrero, Mirian Halfeld Ferrari Alves, and Martin A Musicante. Pews: A new language for building web service interfaces. *J. UCS*, 11(7):1215–1233, 2005.

[16] Aphrodite Tsalgatidou and Thomi Pilioura. An overview of standards and related technology in web services. *Distributed and Parallel Databases*, 12(2-3):135–162, 2002.

[17] Anura Guruge. *Web services: theory and practice*. Digital Press, 2004.

[18] Hugo Haas and Allen Brown. Web services glossary. *W3C Working Group Note (11 February 2004)*, 2004.

[19] Erik Christensen, Francisco Curbera, Greg Meredith, Sanjiva Weerawarana, et al. Web services description language (wsdl) 1.1, 2001.

[20] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple object access protocol (soap) 1.1, 2000.

[21] Tim Berners-Lee and Dan Connolly. Hypertext markup language-2.0. Technical report, RFC 1866, November, 1995.

[22] Tim Bray, Jean Paoli, C Michael Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (xml). *World Wide Web Consortium Recommendation REC-xml-19980210. http://www. w3. org/TR/1998/REC-xml-19980210*, 1998.

[23] Manish Agarwal. *Synthesizing autonomic compositions in grid environment*. PhD thesis, Rutgers, The State University of New Jersey, 2003.

[24] Marcelo Guerra Hahn, Regina Motz, Alberto Pardo, and Martin A Musicante. Formal semantics and expressiveness of a web service composition language. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1667–1673. ACM, 2013.

[25] Assaf Arkin et al. Business process modeling language. *BPMI. org*, 2002.

[26] Assaf Arkin, Sid Askary, Scott Fordin, Wolfgang Jekeli, Kohsuke Kawaguchi, David Orchard, Stefano Pogliani, Karsten Riemer, Susan Struble, Pal Takacsi-Nagy, et al. Web service choreography interface (wsci) 1.0. *Standards proposal by BEA Systems, Intalio, SAP, and Sun Microsystems*, 2002.

[27] Frank Leymann et al. Web services flow language (wsfl 1.0), 2001.

[28] Satish Thatte. Xlang: Web services for business process design. *Microsoft Corporation*, 2001, 2001.

[29] Ja Ja Joseph. An introduction to parallel algorithms, 1992.

[30] Matthew Felice Pace. Bsp vs mapreduce. *arXiv preprint arXiv:1203.2081*, 2012.

[31] Michael T Goodrich, Nodari Sitchinava, and Qin Zhang. Sorting, searching, and simulation in the mapreduce framework. In *Algorithms and Computation*, pages 374–383. Springer, 2011.

[32] Amazon Web Services LLC. Amazon elastic mapreduce, September 2014. Disponível em: <http://aws.amazon.com/elasticmapreduce/>. Acesso em 05. Setembro 2014.

[33] Microsoft Corporation. Daytona., September 2014. Disponível em: <http://research.microsoft.com/en-us/projects/daytona/default.aspx>. Acesso em 05. Setembro 2014.

[34] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: Abstraction and reuse of object-oriented design*. Springer, 1993.

[35] Mohamed Fayad and Douglas C Schmidt. Object-oriented application frameworks. *Communications of the ACM*, 40(10):32–38, 1997.

[36] Tom White. *Hadoop: The Definitive Guide: The Definitive Guide*. O'Reilly Media, 2009.

[37] Apache Software Foundation. Apache hadoop, October 2013. Disponível em: <http://hadoop.apache.org/>. Acesso em 01. Abril 2014.

[38] Apache Software Foundation. Apache software foundation projects, April 2014. Disponível em: <http://apache.org/foundation/>. Acesso em 01. Abril 2014.

[39] Dhruba Borthakur. The hadoop distributed file system: Architecture and design. *Hadoop Project Website*, 11:21, 2007.

[40] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[41] Apache Software Foundation. Tez, 2014. Disponível em: <http://tez.incubator.apache.org/>. Acesso em 04. Junho 2014.

[42] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.

[43] Apache Software Foundation. Storm, 2013. Disponível em: <http://storm.incubator.apache.org/>. Acesso em 04. Junho 2014.

[44] Hortonworks. Hadoop yarn, 2012. Disponível em: <http://br.hortonworks.com/hadoop/yarn/>. Acesso em 05. Junho 2014.

[45] Eric Sammer. *Hadoop operations*. "O'Reilly Media, Inc.", 2012.

[46] Cascading, 2014. Disponível em: <http://www.cascading.org/>. Acesso em 02. Junho 2014.

[47] Vadim Zaliva and Vladimir Orlov. Hamake: A data flow approach to data processing in hadoop. In *CLOSER*, pages 457–461, 2012.

[48] Azkaban, 2014. Disponível em: <http://azkaban.github.io/azkaban2/docs/2.5/>. Acesso em 03. Junho 2014.

[49] Roshan Sumbaly, Jay Kreps, and Sam Shah. The big data ecosystem at linkedin. In *Proceedings of the 2013 international conference on Management of data*, pages 1125–1134. ACM, 2013.

[50] Chen Zhang and Hans De Sterck. Cloudwf: A computational workflow system for clouds based on hadoop. In *Cloud Computing*, pages 393–404. Springer, 2009.

[51] Apache Software Foundation. Apache oozie, 2014. Disponível em: <`http://oozie.apache.org/`>. Acesso em 02. Junho 2014.

[52] Mohammad Islam, Angelo K Huang, Mohamed Battisha, Michelle Chiang, Santhosh Srinivasan, Craig Peters, Andreas Neumann, and Alejandro Abdelnur. Oozie: towards a scalable workflow management system for hadoop. In *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*, page 4. ACM, 2012.

[53] Jie Liu, Qiyuan Li, Feng Zhu, Jun Wei, and Dan Ye. Building an efficient hadoop workflow engine using bpel. In *Current Trends in Web Engineering*, pages 281–292. Springer, 2013.

[54] Christopher P Wadsworth. *Semantics and Pragmatics of the Lambda-Calculus*. PhD thesis, University of Oxford, 1971.

[55] Peter Henderson and James H Morris Jr. A lazy evaluator. In *Proceedings of the 3rd ACM SIGACT-SIGPLAN symposium on Principles on programming languages*, pages 95–103. ACM, 1976.

[56] Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys (CSUR)*, 21(3):359–411, 1989.

[57] Keith Cooper and Linda Torczon. *Engineering a compiler*. Elsevier, 2011.

[58] W Appel Andrew and P Jens. Modern compiler implementation in java, 2002.

[59] Simon L Peyton Jones. *The implementation of functional programming languages (prentice-hall international series in computer science)*. Prentice-Hall, Inc., 1987.

[60] TJW Clarke, PJS Gladstone, CD MacLean, and AC Norman. Skim-the s, k, i reduction machine. In *Proceedings of the 1980 ACM conference on LISP and functional programming*, pages 128–135. ACM, 1980.

[61] David A Turner. A new implementation technique for applicative languages. *Software: Practice and Experience*, 9(1):31–49, 1979.

[62] Richard B Kieburtz. The g-machine: A fast, graph-reduction evaluator. In *Functional Programming Languages and Computer Architecture*, pages 400–413. Springer, 1985.

[63] Geoffrey L Burn, Simon L Peyton Jones, and John D Robson. The spineless g-machine. In *Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 244–258. ACM, 1988.

[64] Thomas Johnsson. *Efficient compilation of lazy evaluation*, volume 19. ACM, 1984.

[65] Lennart Augustsson. A compiler for lazy ml. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 218–227. ACM, 1984.

[66] Simon L Peyton Jones. Implementing lazy functional languages on stock hardware: the spineless tagless g-machine version 2.5, 1993.

[67] Thomas Johnsson. Efficient compilation of lazy evaluation. *ACM SIGPLAN Notices*, 39(4):125–138, 2004.

[68] Simon L Peyton Jones, Chris Clack, Jon Salkild, and Mark Hardie. Grip—a high-performance architecture for parallel graph reduction. In *Functional Programming Languages and Computer Architecture*, pages 98–112. Springer, 1987.

[69] John Darlington and Mike Reeve. Alice a multi-processor reduction machine for the parallel evaluation cf applicative languages. In *Proceedings of the 1981 conference on Functional programming languages and computer architecture*, pages 65–76. ACM, 1981.

[70] Lennart Augustsson and Thomas Johnsson. Parallel graph reduction with the (v, g)-machine. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 202–213. ACM, 1989.

[71] Mark Scheevel. Norma: a graph reduction processor. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 212–219. ACM, 1986.

[72] Martín A Musicante and Rafael D Lins. Gm-c: a graph multi-combinator machine. *Microprocessing and Microprogramming*, 31(1):81–84, 1991.

[73] Daniel Aguiar da Silva Carvalho et al. Uma máquina de redução de grafos para serviços web. Dissertação de mestrado, Universidade Federal do Rio Grande do Norte, 2012.

[74] Daniel A S Carvalho, Martin A Musicante, Alberto Pardo, and Umberto S Costa. A Graph Reduction Machine for Web Service Compositions. unpublished manuscript, 2014.

[75] Sameer Wadkar, Madhu Siddalingaiah, and Jason Venner. *Pro Apache Hadoop*. Apress, 2014.

[76] Alfred V Aho. *Compilers: Principles, Techniques and Tools (for Anna University), 2/e*. Pearson Education India, 2003.

[77] Python Software Foundation. Python, January 2015. Disponível em: <`https://www.python.org/`>. Acesso em 16. Janeiro 2015.

[78] David M Beazley. Ply, python lex-yacc (2001), January 2015. Disponível em: <`http://www.dabeaz.com/ply`>. Acesso em 16. Janeiro 2015.

[79] Stephen C Johnson. Yacc: yet another compiler-compiler. *Bell Laboratories*, 1986.

[80] Aric Hagberg, Dan Schult, and Pieter Swart. Networkx, January 2015. Disponível em: <`http://networkx.github.io/`>. Acesso em 16. Janeiro 2015.

[81] Andrew S Tanenbaum. *Modern operating systems*, volume 3. Pearson Education, 2009.

[82] Cayce Ullman, Brian Matthews, Gregory R. Warnes, Makina Corpus, and Mathieu Le Marec. Soappy, January 2015. Disponível em: <`https://pypi.python.org/pypi/SOAPpy`>. Acesso em 27. Janeiro 2015.

[83] J Chris Anderson, Jan Lehnardt, and Noah Slater. *CouchDB: the definitive guide*. "O'Reilly Media, Inc.", 2010.

[84] Handerson Bezerra Medeiros. Pews-rt: um sistema de tempo de execução para a linguagem pews. Master's thesis, Universidade Federal do Rio Grande do Norte, 2013.

[85] Kenn Scribner. *Microsoft Windows Workflow Foundation: Step by Step*. Microsoft Press, 2007.

[86] Tássia AV Freitas, Thaís V Batista, Flávia C Delicato, and Paulo F Pires. Metrics for evaluation of aspect-oriented middleware. In *Software Engineering, 2009. SBES'09. XXIII Brazilian Symposium on*, pages 73–82. IEEE, 2009.

[87] Alex Holmes. *Hadoop in Practice*. Manning Publications Co., 2012.

[88] Vladimir Orlov. Hamake, January 2015. Disponível em: <`https://code.google.com/p/hamake/wiki/HamakeComparisonWithOtherWorkflowEngines`>. Acesso em 25. Janeiro 2015.

APÊNDICE A – Códigos fonte utilizados nos estudos de caso

Neste apêndice serão listados os códigos fonte auxiliares utilizados para execução dos estudos de caso.

A.1 Estudo de caso 1

```
SETTINGS = {
    'webservice': {
        'module': 'mbpel.om.ws.webservice.WebServiceManager',
        'prefix': 'ws',
        'services': {
            'delay': {
                'wsdl': 'http://marcioalves.com.br:8080/Delay/services/Delay?wsdl',
                'operations': ['waitTime'],
            },
            'storage': {
                'wsdl': 'http://marcioalves.com.br:8080/Storage/services/Storage?wsdl',
                'operations': ['initVar', 'load', 'store'],
            },
            'twitter': {
                'wsdl': 'http://marcioalves.com.br:8080/Twitter/services/TwitterService?wsdl',
                'operations': ['tweet'],
            },
            'weather': {
                'wsdl': 'http://marcioalves.com.br:8080/Weather/services/Weather?wsdl',
                'operations': ['getTemp']
            },
        },
    },
}
```

Figura 55: Arquivo de configuração para o estudo de caso 1.

O grafo inicial gerado pela execução do programa do estudo de caso 1 é mostrado na Figura 56.

Apos a avaliação da chamada de operação, do retorno de operação e dos dois whiles, o grafo toma a forma mostrada na Figura 57.

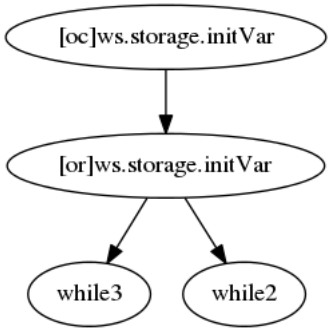


Figura 56: Grafo inicial para o estudo de caso 1.

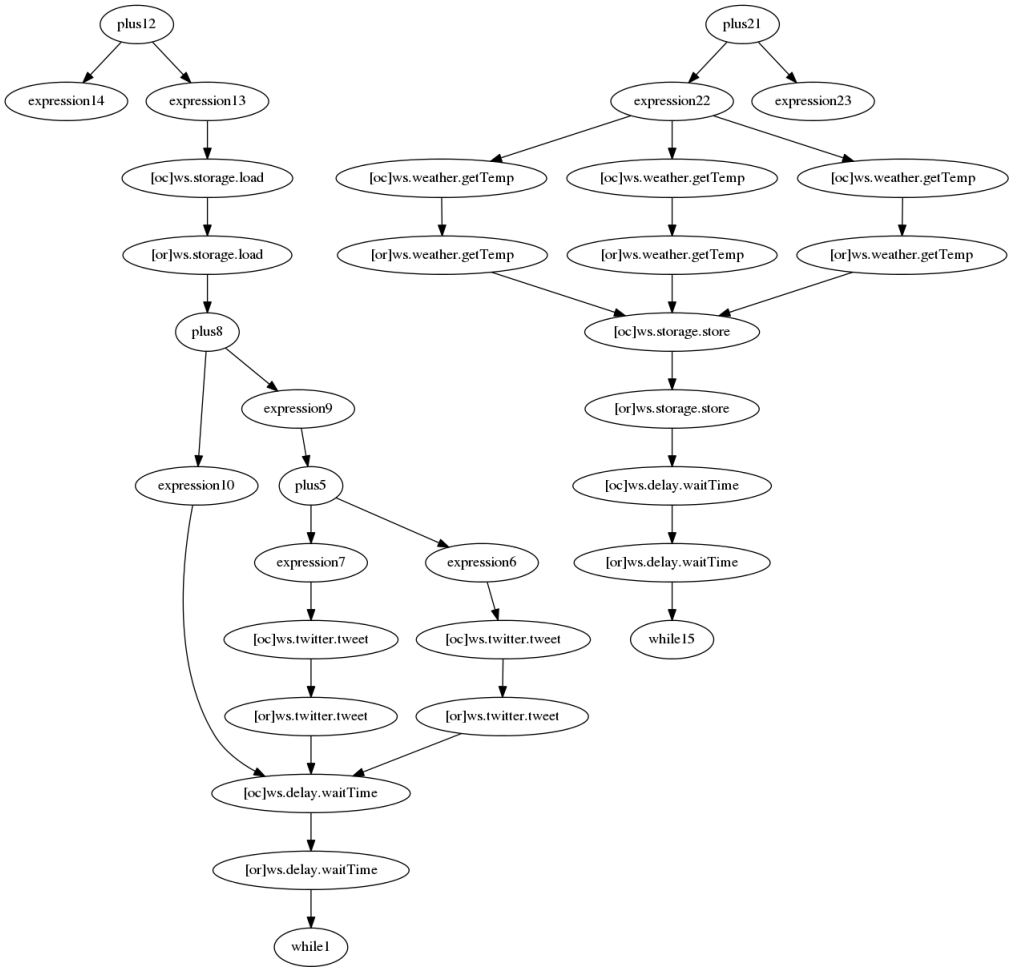


Figura 57: Grafo apos avaliação dos dois whiles para o estudo de caso 1.

A.2 Estudo de caso 2

```
SETTINGS = {
    'hadoop': {
        'module': 'mbpel.om.hd.hadoop.HadoopManager',
        'prefix': 'hd',
        'clusters': {
            'vm': {
                'host': '127.0.0.1',
                'port': '2222',
                'user': 'root',
                'pass': 'hadoop',
                'operations': ['pig', 'ssh', 'hdfs'],
            },
        },
    },
}
```

Figura 58: Arquivo de configuração para o estudo de caso 2.

```
#!/bin/bash

hdfs dfs -rm -R /user/hue/mbpel/minibpel/output_clean_tweets
hdfs dfs -rm -R /user/hue/mbpel/minibpel/output_tweets_sentiment
hdfs dfs -rm -R /user/hue/mbpel/minibpel/input
```

Figura 59: Script para remoção de pastas utilizadas em execuções anteriores.

```
tweets = LOAD '/user/hue/mbpel/minibpel/input/' using
PigStorage(',') as (
    id:   chararray,
    name: chararray,
    date: chararray,
    lang: chararray,
    geo:  chararray,
    from: chararray,
    msg:  chararray
);
b = foreach tweets generate id, name, date, msg;
store b into '/user/hue/mbpel/minibpel/output_clean_tweets';
```

Figura 60: Script Pig para limpeza de campos não utilizados nesta composição.

Listing A.1: Fluxo de trabalho Oozie para o estudo de caso 2.

```
1 <workflow-app name="ec02" xmlns="uri:oozie:workflow:0.4">
2     <start to="delete_dirs"/>
3     <action name="delete_dirs">
4         <fs>
5             <delete path='${nameNode}/user/hue/mbpel/oozie/
                  output_clean_tweets'/>
```

```
tweets = load '/user/hue/mbpel/minibpel/output_clean_tweets' using PigStorage('\t') as
(id:long,name:chararray,date:chararray,text:chararray);

dictionary = load '/user/hue/mbpel/minibpel/dictionary.txt' using PigStorage('\t')  AS
(word:chararray,polarity:chararray);

twords = foreach tweets generate id, FLATTEN( TOKENIZE(text) ) AS word;

tsentiment = join twords by word left outer, dictionary by word using 'replicated';

wscore = foreach tsentiment generate twords::id as id, (CASE dictionary::polarity WHEN 'positive' THEN 1
WHEN 'negative' THEN -1 else 0 END) as score;

tgroup = group wscore by id;

tscore = foreach tgroup generate group as id, SUM(wscore.score)  as final;

tclassify = foreach tscore generate id, ( (final > 0)? 1 : 0 ) as positive, ( (final < 0)? 1 : 0 ) as
negative;

sgroup = group tclassify all;

sentiment = foreach sgroup generate SUM( tclassify.positive ) as tpositive, SUM ( tclassify.negative ) as
tnegative;

store sentiment into '/user/hue/mbpel/minibpel/output_tweets_sentiment';
```

Figura 61: Script Pig que realiza a análise de sentimento.

```
...
impeachment	negative
vitória	positive
FRAUDE	negative
PRESA	negative
eleita	positive
desconfie	negative
#RIPDilma	negative
reeleita	positive
competência	positive
Impeachment	negative
golpe	negative
dívidas	negative
Bom	positive
déficit	negative
crescente	positive
desconfiança	negative
...
```

Figura 62: Dicionário de palavras positivas e negativas.

```
6               <delete path='${nameNode}/user/hue/mbpel/oozie/
                    output_tweets_sentiment'/>
7               <delete path='${nameNode}/user/hue/mbpel/oozie/input'/>
8           </fs>
9           <ok to="create_dirs"/>
10          <error to="kill"/>
11      </action>
12      <action name="create_dirs">
13          <fs>
14              <mkdir path='${nameNode}/user/hue/mbpel/oozie/input'/>
15          </fs>
16          <ok to="fork-17"/>
17          <error to="kill"/>
18      </action>
19      <fork name="fork-17">
20          <path start="twitter_data_01" />
21          <path start="twitter_data_02" />
22      </fork>
23      <action name="twitter_data_01">
24          <fs>
25              <move source='${nameNode}/user/hue/mbpel/oozie/data/1turno
                    ' target='${nameNode}/user/hue/mbpel/oozie/input'/>
26          </fs>
27          <ok to="join-18"/>
28          <error to="kill"/>
29      </action>
30      <action name="twitter_data_02">
31          <fs>
32              <move source='${nameNode}/user/hue/mbpel/oozie/data/2turno
                    ' target='${nameNode}/user/hue/mbpel/oozie/input'/>
33          </fs>
34          <ok to="join-18"/>
35          <error to="kill"/>
36      </action>
37      <join name="join-18" to="clean"/>
38      <action name="clean">
39          <pig>
40              <job-tracker>${jobTracker}</job-tracker>
41              <name-node>${nameNode}</name-node>
42              <script>/user/hue/mbpel/oozie/clean.pig</script>
43          </pig>
44          <ok to="analyse"/>
45          <error to="kill"/>
```

```
46      </action>
47      <action name="analyse">
48          <pig>
49              <job-tracker>${jobTracker}</job-tracker>
50              <name-node>${nameNode}</name-node>
51              <script>/user/hue/mbpel/oozie/sentiment.pig</script>
52          </pig>
53          <ok to="end"/>
54          <error to="kill"/>
55      </action>
56      <kill name="kill">
57          <message>Action failed, error message[${wf:errorMessage(
                wf:lastErrorNode())}]</message>
58      </kill>
59      <end name="end"/>
60  </workflow-app>
```

A.3 Estudo de caso 3

O grafo inicial gerado pela execução do programa do estudo de caso 3 é mostrado na Figura 67.

Apos a avaliação das chamadas de operação, dos retornos de operação e do while, o grafo toma a forma mostrada na Figura 68.

```
SETTINGS = {
    'webservice': {
        'module': 'mbpel.om.ws.webservice.WebServiceManager',
        'prefix': 'ws',
        'services': {
            'delay': {
                'wsdl': 'http://marcioalves.com.br:8080/Delay/services/Delay?wsdl',
                'operations': ['wait'],
            },
            'twitter': {
                'wsdl': 'http://marcioalves.com.br:8080/Twitter/services/TwitterService?wsdl',
                'operations': ['tweet'],
            },
        },
    },
    'hadoop': {
        'module': 'mbpel.om.hd.hadoop.HadoopManager',
        'prefix': 'hd',
        'clusters': {
            'vm': {
                'host': '127.0.0.1',
                'port': '2222',
                'user': 'root',
                'pass': 'hadoop',
                'operations': ['mapreduce'],
            },
        },
    },
    'user': {
        'module': 'mbpel.om.udf.userdefined.UserDefinedFunctionsManager',
        'prefix': 'my',
        'user_modules': {
            'tempo': {
                'module': 'modules.tempo',
                'operations': ['nextDate']
            },
            'tratamento': {
                'module': 'modules.tratamento',
                'operations': ['extractValues']
            },
            'localstorage': {
                'module': 'modules.localstorage',
                'operations': ['filterDataSource1']
            },
            'couch': {
                'module': 'modules.couch',
                'operations': ['save']
            }
        }
    },
}
```

Figura 63: Arquivo de configuração para o estudo de caso 3.

```
#!/bin/bash

hdfs dfs -rm -R /user/hue/mbpel/ec03/output_clean
hdfs dfs -rm -R /user/hue/mbpel/ec03/output_sentiment
hdfs dfs -rm -R /user/hue/mbpel/ec03/input
```

Figura 64: Script para remoção de pastas utilizadas em execuções anteriores.

```
tweets = LOAD '/user/hue/mbpel/ec03/input/' using
PigStorage(',') as (
    id:   chararray,
    name: chararray,
    date: chararray,
    lang: chararray,
    geo:  chararray,
    from: chararray,
    msg:  chararray
);
b = foreach tweets generate id, name, date, msg;
store b into '/user/hue/mbpel/ec03/output_clean';
```

Figura 65: Script Pig para limpeza de campos não utilizados nesta composição.

```
tweets = load '/user/hue/mbpel/ec03/output_clean' using PigStorage('\t') as
(id:long,name:chararray,date:chararray,text:chararray);

dictionary = load '/user/hue/mbpel/ec03/dictionary.txt' using PigStorage('\t')  AS
(word:chararray,polarity:chararray);

twords = foreach tweets generate id, FLATTEN( TOKENIZE(text) ) AS word;

tsentiment = join twords by word left outer, dictionary by word using 'replicated';

wscore = foreach tsentiment generate twords::id as id, (CASE dictionary::polarity WHEN 'positive' THEN 1
WHEN 'negative' THEN -1 else 0 END) as score;

tgroup = group wscore by id;

tscore = foreach tgroup generate group as id, SUM(wscore.score)  as final;

tclassify = foreach tscore generate id, ( (final > 0)? 1 : 0 ) as positive, ( (final < 0)? 1 : 0 ) as
negative;

sgroup = group tclassify all;

sentiment = foreach sgroup generate SUM( tclassify.positive ) as tpositive, SUM ( tclassify.negative )
as tnegative;

store sentiment into '/user/hue/mbpel/ec03/output_sentiment';
```

Figura 66: Script Pig que realiza a análise de sentimento.

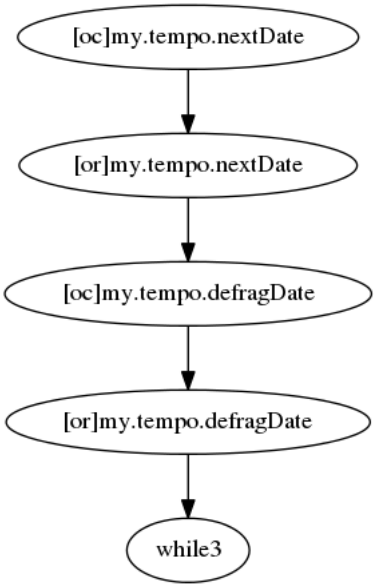


Figura 67: Grafo inicial para o estudo de caso 3.

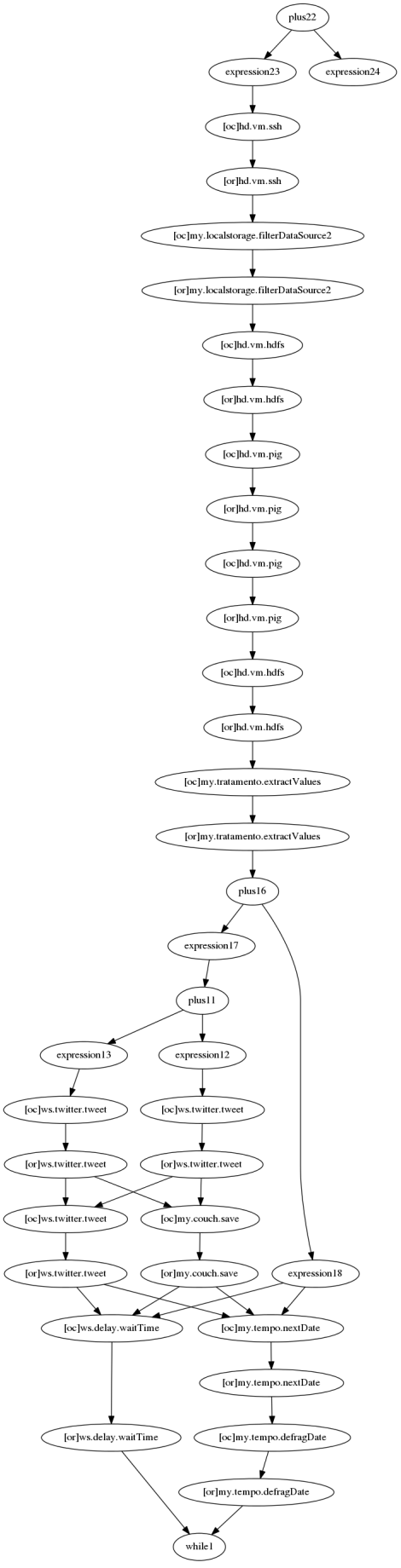


Figura 68: Grafo apos avaliação do while para o estudo de caso 3.