Relatório Final de Iniciação Científica

Uma Ferramenta de Software para a Predição de Desempenho de Workflows Científicos

Aluno: Lucas Magno
Bolsista PIBIC do CNPq
Instituto de Física (IF)

Orientadora: Kelly Rosa Braghetto
Departamento de Ciência da Computação (DCC)
Instituto de Matemática e Estatística (IME)

Universidade de São Paulo

Junho de 2014

Resumo

Este documento descreve as atividades realizadas durante o período de julho de 2013 a junho de 2014 no âmbito do projeto de iniciação científica do aluno Lucas Magno, número USP 7994983, orientado pela Profa. Dra. Kelly Rosa Braghetto e financiado por uma bolsa PIBIC/CNPq.

O objetivo principal do projeto foi desenvolver uma ferramenta de software para a conversão automática de modelos de *workflows* em modelos estocásticos na álgebra de processos *PEPA - Performance Evaluation Process Algebra*. A partir desses modelos estocásticos, é possível extrair predições de desempenho de *workflows*.

Sumário

1 Introdução

Inicialmente desenvolvidos para automatizar processos industriais e empresariais, os *workflows* se popularizaram e passaram a ser usados na modelagem e automatização de experimentos científicos em diversas áreas da ciência. Um *workflow científico* é a descrição completa ou parcial de um experimento científico em termo de suas atividades, controles de fluxo e dependência de dados [12].

1.1 Representação de *Workflows* Científicos

Há várias maneiras de se representar um *workflow científico*, mas entre elas se destacam [13]:

- *Grafos direcionados*: uma das formas mais comuns e simples de representação de *workflows*, permitem sua visualização gráfica e facilitam sua descrição através de modelos gráficos. Num grafo, os nós representam as atividades de um experimento científico e, as arestas, as dependências entre essas.
- Unified Modeling Language *(UML)*: linguagem padrão para modelagem de *software* orientado a objetos, tendo como um de seus recursos o diagrama de atividades, que pode ser utilizado para descrever as dependências entre atividades e, portanto, *workflows*.
- *Redes de Petri*: muito utilizadas para modelar comportamento concorrente em sistemas distribuídos discretos, podem ser interpretadas como um caso particular de grafos direcionados, diferindo destes por possuírem dois tipos de nós: lugares e transições. As arestas sempre conectam dois tipos diferentes de nós, tornando o grafo bipartido.
- *Álgebras de Processos*: podem ser entendidas como um estudo do comportamento de sistemas paralelos ou distribuídos por meio de uma abordagem algébrica, que permite verificações, análises algébricas e aperfeiçoamento de processos por meio de transformações. Sendo assim, diferentemente dos exemplos anteriores, não possuem representação gráfica, somente textual.

Os mecanismos de representação listados acima são usados para criar modelos que especificam a ordem de execução das atividades dos *workflows*. Mas as redes de Petri e as álgebras de processos são linguagens formais (a semântica de seus construtores é definida de forma não ambígua). Por isso, elas permitem também que se verifique propriedades qualitativas ou quantitativas dos modelos de *workflow*, como, por exemplo, a existência de *deadlocks* (pontos de impasse).

No entanto, somente grafos direcionados e álgebras de processo serão utilizados neste trabalho.

1.2 Composição de *Workflows* Científicos

Um *workflow* pode ser composto de diversos elementos, mas os relevantes neste projeto são atividades, que representam atividades reais de um experimento, e estruturas para descrever o fluxo de controle, que definem a ordem de execução das atividades dentro do *workflow*. Temos como exemplos de estruturas *sequência*, *paralelismo*, *escolha* e *sincronização*.

As estruturas de controle de fluxo são definidas nos modelos de workflows por meio de operadores. Os modelos de workflows considerados neste trabalho utilizam os seguintes operadores:

- *AND* (paralelismo/sincronização): todos os ramos são executados
- *XOR* (escolha exclusiva/junção): apenas um ramo é executado
- *OR* (escolha múltipla/junção): um ou mais ramos são executados

Os ramos abertos por um operador (chamado operador de divisão) terminam em um único operador do mesmo tipo (chamado operador de junção), cuja única função é sincronizar o final dos ramos executados.

1.3 Análise de Desempenho

É comum em experimentos científicos a manipulação de enormes quantidades de dados e processos muito demorados, o que estimula o cientista a aperfeiçoar o experimento antes de sua execução, pois esta pode demandar muitos recursos e tempo. Daí a necessidade da análise do desempenho de um *workflow*, que pode ser feita através de três métodos [11]:

- *Medição:* consiste na execução do *workflow* uma quantidade estatisticamente relevante de vezes e então no cálculo dos tempos médios de interesse. Logo, só pode ser aplicada a sistemas já implementados, não tendo caráter preditivo;
- *Simulação:* baseada em modelos matemáticos cuja solução é dada por um programa que simula o comportamento modelado;
- *Modelagem analítica:* também baseada em modelos, mas analisa numericamente determinados aspectos de interesse em um sistema.

Neste projeto, foi usada a modelagem analítica, por ser preditiva, rápida e não muito difícil de se implementar, embora menos precisa que a medição. Tanto as redes de Petri quanto as álgebras de processos são formalismos que possuem extensões estocásticas e que, portanto, podem ser usados no método modelagem analítica.

Como álgebra de processos estocástica foi escolhida a PEPA, *Performance Evaluation Process Algebra* [6], porque o uso desse formalismo ainda não foi profundamente explorado para a análise de desempenho preditiva de *workflows* científicos.

2 Objetivos e Contribuições

Uma desvantagem da modelagem analítica usando PEPA é a necessidade da descrição do *workflow* em uma linguagem de modelagem estocástica e utilização de programas específicos para a análise, exigindo do usuário um certo nível de conhecimento sobre álgebras de processo. No entanto, *workflows* científicos são utilizados em diversas áreas da ciência que não necessitam de um grande aprofundamento em computação, o que pode inviabilizar a aplicação deste método.

O objetivo do trabalho foi facilitar a extração de predições de desempenho para *workflows* científicos. Para isso, foi desenvolvida uma ferramenta de software que gera modelos estocásticos em *PEPA* e sua solução numérica a partir de modelos de *workflows*. Os modelos de *workflows* usados como entrada para a ferramenta são descritos textualmente na forma de um grafo dirigido - uma representação simples e que pode ser usada com facilidade por usuários não especialistas. Além do modelo em *PEPA* e sua solução, a ferramenta também gera uma representação gráfica do modelo de *workflow*, que permite que o usuário possa verificá-lo mais facilmente.

3 Metodologia

Para que possua um modelo correspondente em *PEPA*, um modelo de *workflow* precisa ser bem estruturado e não possuir ambiguidades semânticas. Por essa razão, neste trabalho consideramos apenas modelos de *workflow* que apresentam somente um ponto de entrada e um ponto de saída, têm sua estrutura em forma de "blocos" e não apresentam ciclos, ou laços, o que permite uma implementação mais simples.

3.1 O Programa

Para automatizar o processo de predição de desempenho, foi implementado um programa que realiza as seguintes etapas:

1. Lê como entrada uma descrição textual de um *workflow*
2. Gera uma estrutura de dados baseada em grafo na memória representando o *workflow*
3. Gera uma visualização do *workflow* de entrada
4. Gera um modelo analítico (estocástico) do *workflow*
5. Solução numérica do modelo analítico e extração dos índices de desempenho

Para tanto, foi escolhida a linguagem *Python*, por flexibilidade, facilidade de aprendizado e grande número de bibliotecas auxiliares.

Na etapa 1, foi definida uma gramática simples baseada na linguagem DOT [3] e se utilizou os analisadores léxico e sintático disponíveis na biblioteca PLY, *Python Lex-Yacc* [7] (com dependência na biblioteca *pyParsing* [8]), para efetuar sua leitura.

A implementação da etapa 2 foi feita inicialmente com o auxílio da biblioteca *python-graph* [10], que permite a criação e manipulação de diversos tipos de grafos por meio de classes. Entretanto, mais recentemente, o programa foi reescrito para não depender dessa biblioteca, definindo explicitamente, portanto, as classes de manipulação de nós, arestas e *workflows*. A implementação customizada permitiu maior flexibilidade e clareza de código, além de melhorar a performance e diminuir as dependências. No entanto, a etapa 3, que era realizada pela *python-graph*, teve que ser manualmente implementada, criando a descrição do grafo em linguagem DOT e sua visualização gráfica a partir da biblioteca *Graphviz* [4].

Na etapa 4, a partir da estrutura de dados definida na etapa 2, é criado o modelo analítico do grafo, na linguagem PEPA. Sua implementação não exigiu o uso de nenhuma biblioteca específica e funcionou como esperado nos casos testados, mas, devido a sua complexidade, mais testes serão necessários para confirmar sua eficiência.

A solução numérica do modelo analítico e a extração de índices de desempenho foram realizadas através da biblioteca *pyPEPA* [9], uma implementação recente da *PEPA* em *Python*.

3.2 Nota Técnica

O projeto foi desenvolvido e testado sob sistema operacional linux e *Python* 2.7. Seu código fonte, bem como exemplos de *workflow* de entrada e suas representações em *PEPA* geradas podem ser conferidos na página do projeto [2].

3.2.1 Dependências

Em um sistema linux baseado em *Debian*, os seguintes pacotes devem ser instalados além da instalação padrão do *python* 2.7:

- python-ply (Biblioteca Python Lex-Yacc)
- libgv-python (Biblioteca Graphviz)
- python-pyparsing (Biblioteca pyParsing)
- python-numpy (Biblioteca NumPy)
- python-scipy (Biblioteca SciPy)

3.2.2 Execução

O programa deve ser executado a partir de um terminal utilizando os seguintes comandos:

```
1 $ python script.py input1 input2 input3
```

Onde $ representa que se está num terminal, *script.py* é o arquivo que contém o programa, e *input1*, *input2* e *input3* são arquivos de entrada contendo descrições textuais de *workflows* na linguagem definida neste projeto. Não há limite para a quantidade de arquivos de entrada.

O programa então processa os *workflows* e, caso não haja erros, imprime para a tela:

```
1 workflow1 was sucessfully processed! :D
2 Output files were created.
```

E cria os seguintes arquivos de saída:

- workflow1.dot
- workflow1.pdf
- workflow1.pepa
- workflow1_solution

No caso de ocorrerem erros no processamento de um *workflow*, a seguinte mensagem será impressa para a tela:

```
1 There was an error while processing workflow1. :(
2 Traceback was logged to "workflow1_traceback".
```

Onde *workflow1_traceback* é um arquivo contendo a mensagem de erro originalmente produzida, que não foi impressa para a tela para não a poluir. Também podem ser criados alguns arquivos de saída, dependendo do local do código onde o erro ocorreu.

Deve-se notar que o processamento de cada *workflow* é feito de forma independente, ou seja, essas mensagens serão impressas para cada *workflow* processado e a ocorrência de erro em um processamente não interfere em nada no próximo.

4 Resultados

O programa, então, executa todos os passos desde a leitura da descrição textual do *workflow* à criação do modelo analítico em PEPA. Para demonstrar as saídas do programa, serão utilizados exemplos de um mesmo experimento.

4.1 Descrição Textual do *Workflow*

Baseada em linguagem DOT, foi definida uma linguagem simples para a descrição textual de *workflows*. Como a intenção dessa linguagem é permitir apenas a descrição de grafos que representem experimentos científicos, e não de qualquer tipo de grafo, ela é muito mais concisa que a linguagem DOT. Os detalhes desta linguagem serão discutidos mais adiante.

```
1  graph           : digraph [ ID ] '{' stmt_list '}'
2  stmt_list       : stmt ';' [ stmt_list ]
3  stmt            : node [ "->" edge_list ]
4  node            : ID [ node_attr ]
5  node_attr       : number | operator
6  edge_list       : edge [',' edge_list ]
7  edge            : [ edge_prob ] ID
8  edge_prob       : '[' number ']'
9  number          : DIGIT* ['.' DIGIT* ]
10 operator        : "AND" | "XOR" | "OR"
```

Código 1: Gramática da linguagem de descrição textual de *workflows*

```
1  digraph {
2          a               -> b;
3          b               -> and1;
4          and1 [and]      -> e, xor1;
5          xor1 [xor]      -> [0.15] c, [0.85] d;
6          e [0.5]         -> and2;
7          c               -> xor2;
8          d               -> xor2;
9          xor2            -> and2;
10         and2            -> f;
11 }
```

Código 2: Exemplo de descrição textual de um *workflow* na linguagem definida

4.2 Estrutura de Dados Baseada em Grafo

Utilizando classes em *python*, foi possível implementar diretamente os nós (*nodes*), arestas (*edges*) e até mesmo o próprio *workflow*, bem como diversos *métodos* pertencentes a essas classes que facilitam sua manipulação. Embora suas descrições sejam simples, não serão exibidos seus códigos, para manter a clareza do texto, e sim representações textuais que demonstram os aspectos relevantes da implementação.

Nó

```
Node:
  Name: xor1
  Type: XOR
  Rate: None
  Predecessors: ['and1']
  Sucessors:    ['c', 'd']
```

Código 3: Representação textual de um objeto *Node()*

Aresta

```
Edge:
  Tail: xor1
  Head: c
  Prob: 0.15
```

Código 4: Representação textual de um objeto *Edge()*

Workflow

```
Workflow:
  Name: workflow1
  Nodes: ['a', 'xor1', 'c', 'b', 'e', 'd', 'f', 'xor2', 'and1', 'and2']
  Edges: ['and1 -> e', 'and2 -> f', 'xor2 -> and2', 'e -> and2',
          'c -> xor2', 'xor1 -> d', 'and1 -> xor1', 'b -> and1',
          'a -> b', 'xor1 -> c', 'd -> xor2']
```

Código 5: Representação textual de um objeto *Workflow*

4.3 Visualização do *Workflow*

A partir do grafo em memória do *workflow*, o programa gera arquivos contendo o código equivalente na linguagem DOT e um *pdf* com a visualização do mesmo.

```
digraph workflow1 {
a [shape=box,label=a];
xor1 [shape=diamond,label=xor1,
      color=green];
c [shape=box,label=c];
b [shape=box,label=b];
e [shape=box,label=e];
d [shape=box,label=d];
f [shape=box,label=f];
xor2 [shape=diamond,label=xor2,
      color=green];
and1 [shape=diamond,label=and1,
      color=blue];
and2 [shape=diamond,label=and2,
      color=blue];
and1 -> e;
and2 -> f;
c -> xor2;
e -> and2;
xor2 -> and2;
xor1 -> d [label="0.85"];
and1 -> xor1;
b -> and1;
a -> b;
xor1 -> c [label="0.15"];
d -> xor2;
}
```

Código 6: Exemplo de descrição do *Workflow* em linguagem DOT

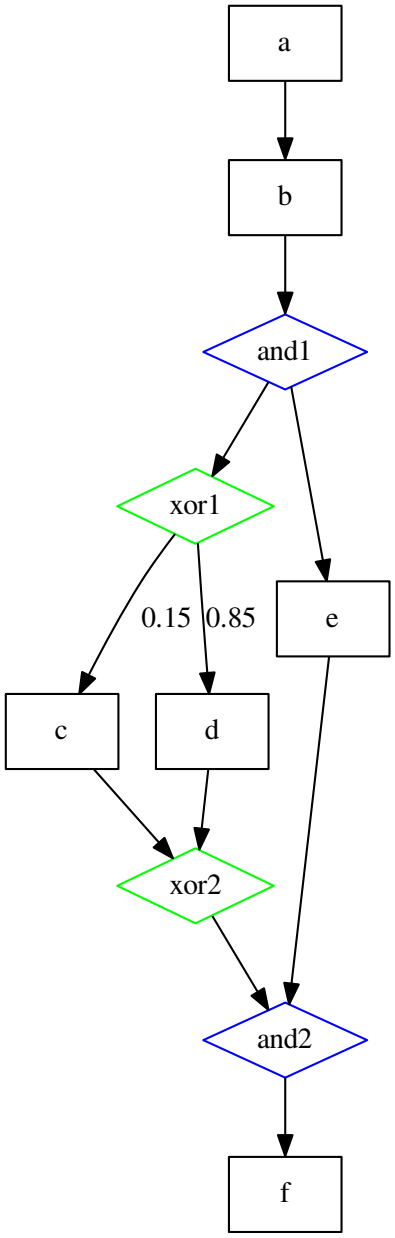

Figura 1: Exemplo de visualização do *workflow* criada a partir do código em DOT

4.4 Descrição em PEPA

Tendo o *workflow* já representado num objeto *Workflow*, o programa então percorre todos seus nós, gerando sua descrição em PEPA e, finalmente, gravando essa descrição em um arquivo *.pepa*.

```
1  r_a = 1.0;
2  r_b = 1.0;
3  r_e = 0.5;
4  r_c = 1.0;
5  r_d = 1.0;
6  r_f = 1.0;
7
8  r_AND = 100.0;
9  r_XOR = 100.0;
10 r_OR  = 100.0;
11
12 prob_xor1_c = 0.15;
13 prob_xor1_d = 0.85;
14
15 r_xor1_c = prob_xor1_c * r_XOR;
16 r_xor1_d = prob_xor1_d * r_XOR;
17
18 P = (a, r_a) . (b, r_b) . (and1, r_AND) . (and2, r_AND) . (f, r_f) . P;
19
20 P_and1_e = (and1, r_AND) . (e, r_e) . (and2, r_AND) . P_and1_e;
21 P_and1_xor1 = (and1, r_AND) . P_xor1;
22 P_xor1_c = (c, r_c) . P_xor2;
23 P_xor1_d = (d, r_d) . P_xor2;
24 P_xor1 = (xor1, r_xor1_c) . P_xor1_c + (xor1, r_xor1_d) . P_xor1_d;
25 P_xor2 = (xor2, r_XOR) . (and2, r_AND) . P_and1_xor1;
26
27 P <and1, and2> (P_and1_e <and1, and2> P_and1_xor1)
```

Código 7: Descrição em PEPA do *workflow*.

5 Análises

5.1 Descrição Textual do *Workflow*

Como dito anteriormente, a gramática de descrição do *workflow* definida é baseada na linguagem DOT, pois esta é amplamente utilizada na representação de grafos em geral. Porém, essa mesma generalidade implica em maior complexidade sintática, por isso a necessidade da definição de uma nova linguagem, capaz de lidar com as formas mais comuns de *workflows* científicos, mas ainda assim minimalista. Portanto, definimos uma linguagem com as seguintes regras (ver o exemplo do Código 2 para maior esclarecimento):

- Toda descrição textual de *workflow* deve ser iniciada com a palavra "digraph", para manter similaridade com a linguagem DOT.
- A seguir, pode aparecer o nome do *workflow*, que será usado para nomear os arquivos de saída. *Workflows* sem nome serão nomeados sequencialmente. Ex: *workflow1*, *workflow2*, *workflow3*, ...
- A descrição do *workflow* propriamente dita fica entre chaves.
- Espaços e tabulações são ignorados.
- Todas as linhas da descrição terminam com ponto e vírgula (";").
- Não há declaração explícita de nós e arestas, esta é feita de forma implícita.
- Nomes de nós e arestas devem ser alfanuméricos e podem conter o símbolo *underline* "_", mas devem iniciar por uma letra minúscula.
- Cada linha representa uma ou mais arestas, utilizando o símbolo "->" (seta) para separar o nó de partida de seus nós de destino.
- À esquerda da seta são declarados os nós de partida com seus respectivos atributos após seu nome e entre colchetes: um número, caso seja uma atividade, representando sua taxa de execução, ou uma palavra (OR, XOR ou AND), caso seja um operador, representando seu tipo. Notar que não é necessário explicitar o tipo do operador caso este esteja fechando um bloco (no exemplo: XOR2 e AND2), o programa atribuirá automaticamente seu tipo. Cada linha só pode conter um nó de partida e seu respectivo atributo.
- À direita são declaradas os nós de destino e seus respectivos atributos, separados por vírgula. Os atributos são números representando a probabilidade daquele caminho (aresta) ser tomado pelos dados e deve estar entre colchetes e antes do nome do nó. Aqui há uma criação de nós ainda não declarados, mas sem atributos, que serão adicionados posteriormente quando estes forem declarados como nós de partida.
- Toda linha deve conter a seta, isto é, não se pode declarar somente um nó em uma linha. No entanto, na declaração de arestas, o nó de destino é criado automaticamente. Isto implica no fato de que nós sem arestas partindo deles não podem ter atributos, exigindo em alguns casos a utilização de um nó especial para simbolizar o final do *workflow*.

5.2 Analisadores Léxico e Sintático

Uma vez decidido partir de uma descrição textual do *workflow*, é necessário que o programa seja capaz de ler e interpretar o texto dado. Logo, é necessário o uso de analisadores léxicos e sintáticos, que têm exatamente essa função.

O analisador léxico, ou *lexer*, quebra o texto em pequenos fragmentos, ou *tokens*, seguindo regras definidas pelo usuário. A seguir, passa esses *tokens* ao analisador sintático, ou *parser*, que, também a partir de regras, interpreta o papel de cada *token* na sintaxe geral em relação aos anteriores, executando uma ação específica a cada *token* novo. Apesar de não serem muito simples de se implementar, os analisadores permitem uma grande flexibilidade na definição da gramática do texto.

Escolheu-se, então, utilizar o Lex, *A Lexical Analyzer Generator*, e o Yacc, *Yet Another Compiler-Compiler* [5], geradores de analisadores léxicos e sintáticos, respectivamente. Apesar de antigos (1975 e 1970, respectivamente), ainda são amplamente utilizados através de reimplementações e frequentemente juntos. O que é exemplificado pela biblioteca PLY, *Python Lex-Yacc*, uma implementação recente (2001) inteiramente em *Python*, a qual busca entregar toda a funcionalidade do Lex/Yacc somada a uma extensa verificação de erro.

Portanto, ao utilizar a biblioteca PLY, é possível definir uma sintaxe abstrata para o *workflow* independente das outras partes do programa e, ao mesmo tempo, construir o grafo representante em tempo real, isto é, ao longo da leitura do texto.

5.3 Estrutura de Dados Baseada em Grafo

A escolha de se representar o *workflow* por meio de uma estrutura de dados em memória, em forma de grafo, se dá pela generalidade dessa estrutura, que não depende de nenhuma linguagem, facilitando sua manipulação e eventuais traduções para linguagens específicas.

Por o projeto ser em *Python*, inicialmente se imaginou necessária uma biblioteca desta linguagem que trabalhasse com grafos de uma maneira leve e flexível. Foi encontrada, então, a *python-graph*, que parecia preencher esses requisitos e oferecia um grande número de algoritmos úteis ao se lidar com grafos. No entanto, logo ficou claro que faltavam algumas funções simples e frequentemente era necessário modificar o código da biblioteca. Como isto não era uma boa opção (o programa deve rodar igualmente em qualquer sistema), decidiu-se abandonar o uso desta biblioteca e se implementar um código equivalente no próprio programa.

Utilizando apenas três classes (*Node*, *Edge* e *Workflow*) e alguns algoritmos, foi possível manipular os *workflows* de maneira clara, utilizando uma linguagem de alto nível, e flexível, permitindo ajustar a estrutura de cada classe conforme necessário.

A seguir uma breve descrição de cada classe, bem como suas componentes. Para melhor ilustração, ver os Códigos 3, 4 e 5.

5.3.1 Classe *Node*

Classe que representa um nó, composta por:

- Atributos:
  - *name*: *string* contendo o nome do nó
  - *type*: *string* contendo o tipo do nó ("ACT", "OR", "XOR" ou "AND")
  - *rate*: variável contendo a taxa de execução do nó (número ou *None*)
  - *predecessors*: lista contendo todos os nós antecessores a este no *workflow*
  - *sucessors*: lista contendo todos os nós sucessores a este no *workflow*

5.3.2 Classe *Edge*

Classe que representa uma aresta, composta por:

- Atributos:
  - *tail*: objeto do tipo *Node* contendo o nó de partida da aresta
  - *head*: objeto do tipo *Node* contendo o nó de chegada da aresta
  - *prob*: variável contendo a probabilidade daquela aresta ser percorrida (número)

5.3.3 Classe *Workflow*

A classe mais complexa das três. Representa um *workflow* completo e, além de atributos, contém vários *métodos*. É composta por:

- Atributos:
  - *name*: *string* contendo o nome do *workflow*
  - *nodes*: dicionário contendo todos os objetos do tipo *Node* que pertencem ao *workflow* indexados por seus respectivos nomes (variável *name*)
  - *edges*: dicionário contendo todos os objetos do tipo *Edge* que pertencem ao *workflow* indexados por uma tupla na forma *"(nome do nó de partida, nome do nó de chegada)"*
  - *pepa*: dicionário contendo listas de *strings* que depois serão unidas para formar uma única *string* (a descrição em *PEPA* do *workflow*)
- Métodos:
  - *add_node*: função que adiciona o nó especificado ao *workflow*, atualizando o dicionário *nodes*
  - *add_edge*: função que adiciona a aresta especificada ao *workflow*, atualizando o dicionário *edges* e as listas *predecessors* e *sucessors* dos nós involvidos
  - *has_node*: função que retorna se um nó especificado pertence ao *workflow* (retorna *True* ou *False*)
  - *has_edge*: função que retorna se uma aresta especificada pertence ao *workflow* (retorna *True* ou *False*)
  - *root_node*: função que retorna o nó pertencente ao *workflow* que não tem nenhum antecessor. Presume-se que só exista um nó com essa característica no *workflow* (nó "raiz").

5.4 Linguagem *DOT*

Antes realizada automaticamente pela biblioteca *python-graph*, a tradução do *workflow* para linguagem *DOT* teve que ser manualmente implementada. A função responsável por isso (*write*) percorre o *workflow* e cria sua descrição em *DOT* simultaneamente.

Uma vez obtida tal descrição, cria-se sua representação gráfica, através da biblioteca *Graphviz*, e ambas são salvas em arquivos de saída. Essas etapas são executadas pela função *dot_pdf*, como pode ser visto em seu código:

```
1  def dot_pdf(wkf):
2    # Descrever o workflow em linguagem DOT, gerar sua visualização
3    # e salvar ambas em arquivos de saída
4
5    # Obter string com a descrição do workflow em linguagem DOT
6    dot = write(wkf)
7
8    # Escrevê-la num arquivo de saída
9    with open(wkf.name + '.dot', 'w') as f:
10     f.write(dot)
11
12   # Renderizar workflow baseado em sua descrição em DOT
13   # e salvar em um arquivo pdf
14   gvv = gv.readstring(dot)
15   gv.layout(gvv,'dot')
16   gv.render(gvv,'pdf', wkf.name + '.pdf')
```

Código 8: Função *dot_pdf com comentários traduzidos*

Neste projeto foi adotado o formato *pdf*, por sua qualidade, versatilidade e facilidade de inclusão em documentos. Também foram adotadas algumas convenções para facilitar a visualização do *workflow*, como pode ser verificado no Código 6 e na Figura 1, que são:

- Atividades são representadas por retângulos
- Operadores são representados por losangos com as seguintes cores:
  - *Azul* para operadores do tipo *AND*
  - *Vermelho* para operadores do tipo *OR*
  - *Verde* para operadores do tipo *XOR*
- Nome de cada componente (atividade ou operador) dentro de seu símbolo
- Setas para representar relações de precedência entre componentes
- Um número (entre 0 e 1) ao lado de cada seta representando a probabilidade daquele "ramo" do *workflow* ser executado. Caso seja 1 (o padrão), ele é omitido

Apesar de algumas restrições, como a ausência de atributos dos nós nas imagens (Figura 1) e não distinguir operadores que dividem a linha de fluxo dos que fazem sua junção, ainda é vantajosa a utilização da linguagem *DOT*, pois suas funções são relativamente simples de implementar e dão ao usuário a possibilidade de verificar se a estrutura em memória corresponde a seu *workflow* original.

5.5 Modelagem Analítica

Uma vez que já existe o grafo na memória, só resta sua tradução para um modelo analítico e então efetuar a análise numérica. Foi escolhido, então, o modelo de álgebra de processos estocástica, por apresentar vantagens em relação a outros modelos, dentre as quais as mais importantes são [1]:

- *Composicionalidade:* a habilidade de modelar um sistema como a interação de subsistemas.
- *Formalismo:* dar um significado preciso para todos os termos na linguagem.
- *Abstração:* a habilidade de construir modelos complexos a partir de componentes detalhadas, desconsiderando os detalhes quando apropriado.

Dentre as linguagens disponíveis, foi usada a *PEPA*, uma álgebra de processos estocásticos bem desenvolvida e que conta com várias ferramentas de apoio, como um complemento para o ambiente integrado de desenvolvimento *Eclipse* e, recentemente, uma biblioteca para a linguagem *Python*, a *pyPEPA*.

Essa biblioteca é inicialmente utilizada para calcular as probabilidades no regime estacionário de cada um dos estados possíveis do *workflow* descrito no modelo em *PEPA*. A partir destas probabilidades, a *pyPEPA* consegue fornecer o rendimento (*throughput*) das atividades do *workflow* e também a taxa de utilização de seus componentes.

A *PEPA*, no entanto, é uma linguagem muito genérica, desenvolvida para ser capaz de definir inúmeros tipos de processo. Isto a torna complexa, impondo dificuldades na conversão de modelos de *workflow*. Por este motivo, a ferramente desenvolvida neste projeto converte apenas *workflows* que contenham os operadores *AND* ou *XOR*, mas pode ser estendida para suportar o operador *OR*, mesmo este não tendo implementação direta em *PEPA*, através de uma combinação dos outros dois.

Mesmo assim, a estrutura de um modelo em *PEPA* pode ser resumida em (ver Código 7):

1. Declaração das constantes que definem as taxas de execução das atividades e dos operadores, bem como as probabilidades dos ramos. No formato:

   "*constante* = *expressão*;"

   Onde *expressão* pode ser qualquer equação matemática simples envolvendo números e/ou *constantes* previamente definidas.

2. Declaração das equações que definem os componentes do *workflow* no formato:

   "*nome da equação* = *componentes*;"

   Onde os componentes podem ser:

   - Outras equações
   - *Ações*: são descritas no formato:
     "(*nome*, *constante*)"
     Deve-se notar que as ações aqui representam qualquer nó do *workflow*, não havendo distinção entre as atividades e operadores utilizados durante o programa.
   - *Operadores*: usados para controlar o fluxo do *workflow*, são sempre aplicados entre duas ações ou equações. São:
     - *Sequência*: representada por um *ponto* ("."), indica que uma ação ou equação é executada logo após outra.
     - *Escolha*: representada por uma soma ("+"), indica que apenas uma das opções será executada.

3. Definição do *workflow* completo, por meios dos paralelismos (indicados pelo operador "||") e sincronizações (indicadas pelo operador "<ação1,ação2,ação3...>", que sincroniza duas equações nas ações especificadas). Esta definição deve conter pelo menos a equação mestre (geralmente denotada por "P") e não deve terminar em *ponto-e-vírgula* (";").

6 Conclusões

Neste projeto de iniciação científica, foi desenvolvida uma ferramenta de software que converte de forma automática modelos de *workflow* em modelos estocásticos e, a partir destes últimos, extrai medidas de desempenho dos *workflows*, facilitando o uso de ferramentas de predição por usuários não-especialistas.

Os modelos de *workflow* recebidos como entrada para o programa são definidos por meio de uma notação simples que permite descrever os modelos mais comuns de experimentos científicos.

Para a geração dos modelos estocásticos, usou-se a linguagem *PEPA*, capaz de descrever uma grande variedade de processos, embora tenha uma sintaxe complexa. A análise numérica desses modelos foi feita por meio da biblioteca *pyPEPA*, uma implementação recente de *PEPA* em *Python*.

A escolha de *Python* como linguagem de implementação do programa se provou bastante favorável, já que as estruturas de dados e bibliotecas existentes nessa linguagem agilizaram o desenvolvimento das funcionalidades primárias do programa.

As etapas realizadas durante o projeto podem ser resumidas em:

- O estudo da linguagem *Python* e suas bibliotecas utilizadas
- O estudo dos conceitos relacionados ao tema do projeto, como "*workflows* científicos", "linguagens de modelagem de *workflows*" e "modelos estocásticos", como a PEPA
- Criação de uma linguagem textual simples para a descrição de *workflows*
- Criação de um *lexer* e um *parser* para a leitura da descrição de um *workflow*
- Criação de uma estrutura de dados baseada em grafo em memória que represente um *workflow*
- Gerar arquivos contendo o código em linguagem DOT que representa o grafo em memória e sua visualização
- Criação de um algoritmo para a conversão de um grafo de *workflow* para um modelo analítico em *PEPA*
- Solução do modelo analítico em *PEPA* e extração dos índices de desempenho através da biblioteca *pyPEPA*

O programa pode ser acessado na página do projeto [2].

6.1 Dificuldades Encontradas no Projeto

A complexidade da sintaxe da *PEPA*, entretanto, dificultou a conversão dos modelos de *workflow*, pois exigiu o tratamento de diversas especificidades desta. Além disso, a implementação recente da *pyPEPA* apresentou diversos obstáculos durante o projeto, como sua sintaxe muito restritiva (ela ainda não suporta completamente a sintaxe da *PEPA*) e sua documentação escassa.

Por estes motivos, as etapas relativas a essas ferramentas ocuparam grande parte do cronograma do projeto, não sendo possível, portanto, realizar outras funcionalidades inicialmente sugeridas neste trabalho, mas que ainda são interessantes de se implementar, listadas a seguir.

6.2 Trabalhos Futuros

- Gerar modelos analíticos de *workflows* que contenham o operador *OR*
- Definição de uma linguagem para descrição dos recursos que podem ser utilizados por um *workflow*
- Incorporação de informações sobre recursos nos modelos analíticos
- Permitir a descrição de *workflows* cuja estrutura não seja "em blocos"

Referências

[1] *About Performance Evaluation Process Algebra*. http://www.dcs.ed.ac.uk/pepa/about/, [Online; acessado em 24 de junho de 2014].

[2] *Código fonte da ferramenta de software desenvolvida, exemplos de workflow de entrada e seus respectivos modelos em PEPA*. www.ime.usp.br/~kellyrb/ic/#lucas, [Online; acessado em 24 de junho de 2014].

[3] *The DOT Language | Graphviz - Graph Visualization Software*. http://www.graphviz.org/content/dot-language, [Online; acessado em 24 de junho de 2014].

[4] *Graphviz | Graphviz - Graph Visualization Software*. http://www.graphviz.org/, [Online; acessado em 24 de junho de 2014].

[5] *The LEX & YACC Page*. http://dinosaur.compilertools.net/, [Online; acessado em 24 de junho de 2014].

[6] *PEPA - Performance Evaluation Process Algebra*. http://www.dcs.ed.ac.uk/pepa/, [Online; acessado em 24 de junho de 2014].

[7] *PLY (Python Lex-Yacc)*. http://www.dabeaz.com/ply/, [Online; acessado em 24 de junho de 2014].

[8] *pyparsing - home*. http://pyparsing.wikispaces.com/, [Online; acessado em 24 de junho de 2014].

[9] *pypepa - Python toolset for PEPA*. https://github.com/tdi/pyPEPA, [Online; acessado em 24 de junho de 2014].

[10] *python-graph - A library for working with graphs in Python - Google Project Hosting*. http://code.google.com/p/python-graph/, [Online; acessado em 24 de junho de 2014].

[11] Braghetto, K. R.: *Técnicas de Modelagem para a Análise de Desempenho de Processos de Negócio*. Tese de Doutoramento, Instituto de Matemática e Estatística da Universidade de São Paulo, 2011.

[12] Gadelha, L. M. R.: *Gerência de Proveniência em Workflows Científicos Paralelos e Distribuídos*. Tese de Doutoramento, Universidade Federal do Rio de Janeiro, 2012.

[13] Ogasawara, E. S.: *Uma Abordagem Algébrica para Workflows Científicos com Dados em Larga Escala*. Tese de Doutoramento, Universidade Federal do Rio de Janeiro, 2011.