UNIVERSIDADE FEDERAL DO PAMPA – UNIPAMPA

WILLIAN DOMINGUES COELHO

EDITOR DE GRAMÁTICAS DE GRAFOS ORIENTADOS A OBJETO

Bagé
2015

Willian Domingues Coelho

Editor de Gramáticas de Grafos Orientados a Objeto

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Computação da Universidade Federal do Pampa, como requisito para obtenção do Título de Bacharel em Engenharia de Computação.

Orientador: Prof.ª Dr.ª Ana Paula Lüdtke Ferreira

Bagé
2015

Willian Domingues Coelho

Editor de Gramáticas de Grafos Orientados a Objeto

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Computação da Universidade Federal do Pampa, como requisito para obtenção do Título de Bacharel em Engenharia de Computação.

Trabalho de Conclusão de Curso defendido em: 31 de janeiro de 2015.
Banca examinadora:

_____

Prof.ª Dr.ª Ana Paula Lüdtke Ferreira
Orientador
Engenharia de Computação/Campus Bagé – UNIPAMPA

_____

Prof. MSc. Carlos Michel Betemps
Engenharia de Computação/Campus Bagé – UNIPAMPA

_____

Prof.ª MSc. Sandra Dutra Piovesan
Engenharia de Computação/Campus Bagé – UNIPAMPA

AGRADECIMENTOS

Aos meus pais, por sempre me incentivarem a manter o foco nos estudos. À minha namorada, por todo o apoio e todos os momentos de alegria que tem me proporcionado nos últimos anos. À minha orientadora, por todo o empenho e dedicação para que este trabalho fosse realizado da melhor forma possível.

RESUMO

Com o crescimento da demanda por softwares cada vez mais complexos e com prazos de entrega cada vez menores, todo procedimento de otimização no desenvolvimento de software que possa acelerar o processo, mantendo a qualidade, deve ser estudado. Com isso, o conceito de Model-Driven Engineering (MDE) surge como uma opção para automatização do processo de desenvolvimento de software, pois consiste na geração (semi) automatizada de código a partir de especificações de alto nível. Este trabalho apresenta a implementação de uma ferramenta que permite a especificação de sistemas por meio de gramáticas de grafos, gramáticas de grafos tipados e gramáticas de grafos orientados a objeto. Tal ferramenta permite criar e editar gramáticas de grafos de forma interativa, além de permitir a exportação das gramáticas construídas em um formato que pode ser utilizado para as etapas de geração de código do conceito de MDE.

Palavras-chave: gramáticas de grafos orientados a objeto, MDE, editor de gramáticas de grafos.

ABSTRACT

With the growth market demand for increasingly complex software systems and with shorter delivery deadlines, all the optimizations procedures that may speed up the process, keeping the quality ought to be studied. Therewith, the concept of Model-Driven Engineering (MDE) arises like an option for software automation development process, because it is a (semi) automatic code generation from high-level specifications. This paper presents an implementation of a tool that allows the specification of systems through graph grammars, typed graph grammars and object-oriented graph grammars. Such a tool allows create and edit graph grammars iteratively, in addition to afford the grammars exportation constructed in a format that may be used for the MDE's steps of code generation.

Keywords: object-oriented graph grammars, MDE, graph grammars editor.

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

API – Application Programming Interface

DSMLs – Domain-Specific Modeling Languages

GXL – Graph eXchange Language

IDE – Integrated Development Environment

JGraph – Java Graph

MDE – Model-Driven Engineering

XML – eXtensible Markup Language

OO – Orientação a Objeto

OOGXL – Object-Oriented Graph eXchange Language

QoS – Quality of Service

SPIN – Simple Promela Interpreter

UML – Unified Modeling Language

SUMÁRIO

1. INTRODUÇÃO

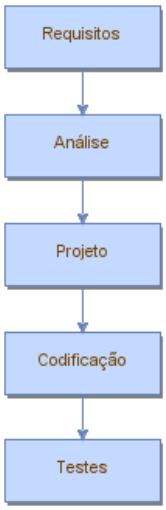
Apesar da intensa evolução das linguagens de programação, dos modelos de desenvolvimento de software e ambientes integrados de desenvolvimento (IDE), os principais erros de programação pouco se alteraram com o passar dos anos. Ainda que erros possam ser introduzidos em qualquer fase de um projeto de desenvolvimento de software, é a etapa de codificação que apresenta maior concentração de introdução de erros em relação às demais etapas (Humphrey, 1995). A codificação é um processo que, na indústria de software, dá pouca margem de criatividade ao programador, tratando-se de um processo longo e manual. Assim, o programador está sujeito a cometer erros de codificação, resultando em gasto de tempo e esforço de correção nos procedimentos posteriores de teste de unidade e de teste de integração.

Devido à quantidade de bibliotecas disponíveis, constantes mudanças nas Interfaces de Programação de Aplicativos (API), surgimento de novas plataformas e demanda por rapidez e qualidade na entrega de produtos ao mercado, a indústria de software torna-se cada vez mais complexa de gerenciar. Dentro desta realidade, não é aceitável que os desenvolvedores demorem muito tempo para familiarizar-se com o uso de padrões e APIs de uma determinada linguagem de programação, o que é necessário para que um produto de qualidade seja produzido. Ademais, o processo de encontrar soluções via software não se restringe à programação.

O projeto do sistema é genérico e quase sempre independe da arquitetura específica de implementação e das linguagens de programação utilizadas, como mostra a Figura 1. A elicitação de requisitos visa obter do usuário as necessidades do sistema em termos de usabilidade e regras do negócio. No processo de análise e de projeto o produto é efetivamente concebido em termos de funcionalidade e estrutura. Contudo, nesta fase ainda não temos relação com uma arquitetura específica. Nomeadamente, o modelo de dados usualmente é descrito via diagramas de entidade-relacionamento e o modelo funcional é descrito por meio de alguma representação gráfica textual em alto nível (digrama de fluxo de dados, diagramas de sequência, etc.). Somente no processo de codificação que linguagens e ferramentas especificas passam a ser usadas. Os processos de codificação e teste são então os processos menos criativos (visto que somente

implementam aquilo que já foi projetado e previamente definido) e, por essa razão, são mais passíveis de automatização.

Figura 1- Etapas de desenvolvimento de software



Fonte: Arquivo Pessoal

Embora a Figura 1 apresente a estrutura básica do desenvolvimento de software, as metodologias de desenvolvimento efetivamente usadas não são sequenciais como é mostrado [Sommerville, 2011]. Requisitos são modificados ou precisam ser detalhados ao longo do processo de desenvolvimento e mesmo após a entrega do software ao cliente ou ao mercado [Pressman, 2006]. A alteração dos requisitos naturalmente gera mudanças em todas as fases subsequentes. Alterações na tecnologia usada (arquiteturas, linguagens de programação, métodos de especificação) também têm impacto no projeto e demais fases do processo de desenvolvimento. Em Sommerville (2011), por exemplo, diversas metodologias são apresentadas e descritas, caracterizando frequentes idas e vindas, usualmente gerando retrabalho.

Uma possível solução para estes problemas é a geração automática de código a partir de uma especificação de mais alto nível. Dentre as alternativas disponíveis para gerar código automaticamente, o conceito de *Model-Driven Engineering* (MDE) trabalha a partir de modelos abstratos, usualmente baseados em linguagens visuais de especificação, que são facilmente compreensíveis, inclusive por pessoas sem conhecimento na área de computação. Note-se que linguagens visuais de especificação são de grande valia na validação de sistemas com o usuário, que somente quer entender as interfaces para verificar se seus desejos estão sendo atendidos. A validação é uma fase

essencial do desenvolvimento de software e frequentemente gera alteração em requisitos e análise do sistema.

O MDE baseia-se na construção de modelos, os quais são abstrações dos sistemas de software em alto nível, e que, após sucessivos refinamentos, geram o código descrito pelo modelo inicial. Refinamento é denominado o processo de traduzir uma especificação abstrata em uma outra especificação mais concreta (ou mais próxima da sua implementação real). Esses refinamentos preferencialmente ocorrem da maneira mais automática possível, com garantia de que as traduções dos modelos estejam corretas em relação à semântica desejada para o sistema. Este modo de produzir software minimiza, ou mesmo elimina, a necessidade de conhecimento sobre uma determinada linguagem, torna o processo de desenvolvimento mais rápido e eficaz, evitando erros comuns de programação, assegura a coerência entre as implementações e os requisitos de qualidade descritos nos modelos e minimiza o retrabalho no caso de mudança nos requisitos do sistema.

Para que a tradução de uma especificação em outra seja garantidamente correta, é preciso que tanto a linguagem de origem quanto a linguagem de destino tenham sintaxe e semântica formal. No mínimo, as linguagens precisam ter sintaxe formal para que a tradução (usualmente dirigida por sintaxe) possa ser realizada (AHO, 2008).

Umas das maneiras de utilizar os conceitos de MDE é a construção de modelos abstratos por meio de um editor de linguagens visuais. A linguagem visual foco deste trabalho são as *gramáticas de grafos*. Gramáticas de grafos definem um modelo computacional, com concorrência verdadeira e totalmente baseada em grafos. Diagramas de toda sorte podem ser formalizados como grafos, constituindo um modelo formal para a semântica (significado) destes diagramas. Adicionalmente, como podem ser expressos como objetos gráficos que especificam relacionamentos, facilitam a comunicação com o usuário não especialista.

Dado que o paradigma dominante do desenvolvimento de software hoje é a orientação a objetos (PRESSMAN, 2006), o foco deste trabalho são as *gramáticas de grafos orientados a objeto*. Gramáticas de grafos orientados a objeto (GGOO) foram propostas por Ferreira (2005) com o objetivo de incorporar os conceitos de orientação objeto (encapsulamento de dados, oclusão da informação, herança e polimorfismo) na abordagem algébrica para gramática de grafos proposta por Löwe (1991).

O uso de modelos visuais de especificação facilita a construção do sistema e ainda permite uma melhor comunicação entre desenvolvedores e diversos públicos, leigos ou

não. O processo de validação de requisitos, com o usuário de um sistema, pode beneficiar deste tipo de especificação.

Para que um sistema possa ser descrito de forma visual por uma determinada aplicação, de forma prática e atrativa ao usuário, a ferramenta deve conter recursos visuais que auxiliem no processo de especificação. Mas, mais do que isso, essas especificações devem ser salvas em formatos que permitam a integração com outras ferramentas. A linguagem OOGXL (Object-Oriented Graph eXchange Language), proposta em Moreira (2007), será usada neste trabalho com esse fim. A linguagem OOGXL é uma extensão da linguagem de especificação de grafos GXL (Graph eXchange Language); esta última, por sua vez, é baseada na linguagem XML (eXchange Markup Language). A linguagem OOGXL permite a representação de grafos com características de orientação a objetos. Essas gramáticas geradas constituem a primeira etapa do conceito de MDE, a construção de modelos. De posse destes arquivos, é possível gerar código automaticamente por meio de traduções das gramáticas geradas. Em Moreira (2007) é implementada uma tradução de OOGXL para a linguagem de entrada do verificador formal SPIN (*Simple Promela Interpreter*), PROMELA. Dessa forma, modelos construídos com gramáticas de grafos orientados a objeto podem ser formalmente verificados e ter propriedades de sua execução (ausência de *deadlock* ou de postergação indefinida, por exemplo) garantidas. Essa aplicação esclarece o propósito deste trabalho, visto que até o momento não existe editor que gere gramáticas nesta linguagem. Dessa forma, o escopo deste trabalho está restrito à fase de modelagem conceitual de sistemas. O produto gerado, na linguagem OOGXL, pode então ser usado em outros trabalhos que visem posteriores refinamentos, testes ou verificação.

1.1 Objetivos

O objetivo deste trabalho é desenvolver uma ferramenta que permita a criação, edição e exportação de modelos de sistemas especificados como gramáticas de grafos, gramática de grafos tipados ou gramáticas de grafos orientados a objeto para diferentes formatos de arquivo.

Dentro dos objetivos específicos temos os seguintes requisitos de construção:

- Apresentar uma interface amigável para o usuário, dentro dos princípios de interação humano-computador existentes;
- Permitir a criação de gramáticas de grafos tipados e de grafos tipos, que garantem a possibilidade de colocar informações sobre tipagem de objetos, facilitando a migração para um modelo computacional;
- Permitir que tipos possam ser especificados por meio de gráficos, em formatos usuais de representação de imagens (gif, png, jpeg, etc.);
- Permitir a criação e edição do grafo tipo, do grafo inicial e do conjunto de regras de gramática, com operadores que façam sentido do ponto de vista do usuário e da própria especificação;
- Exportar as gramáticas de grafos especificadas em formato OOGXL, com a garantia de que a tradução do modelo visual para a linguagem textual está correta.

1.2 Organização do trabalho

Este trabalho está organizado como se segue: no Capítulo 2 são apresentados os aspectos teóricos que foram estudados para o desenvolvimento do trabalho. São abordados temas como MDE, grafos, gramáticas de grafos e trabalhos correlatos. O Capítulo 3 descreve a metodologia e as ferramentas que foram utilizadas para o desenvolvimento deste trabalho. Já o Capítulo 4, apresenta o desenvolvimento do Editor de Gramáticas de Grafos, detalhando os requisitos, especificação e implementação do protótipo. Por fim, o Capítulo 5 apresenta as conclusões finais.

2. REFERENCIAL TEÓRICO

2.1 Model-Driven Engineering

MDE é um arcabouço teórico focado na construção de modelos abstratos de sistemas. Quando implementado em software, esses modelos são construídos em níveis mais altos do que códigos de linguagens de programação, diagramas de fluxo de dados ou outras representações que estejam próximas de codificação de sistemas. Segundo Schmidt (2006), tecnologias MDE combinam *Domain-Specific Modeling Languages* (DSMLs) e ferramentas de transformação e geração de artefatos.

DSMLs são linguagens de especificação, usualmente visuais, que formalizam a estrutura da aplicação, comportamento e requisitos dentro de domínios de aplicação específicos. São descritas por meio do uso de metamodelos, que definem as relações entre os conceitos de um domínio e especificam precisamente as restrições associadas com estes conceitos de domínio (Schmidt, 2006).

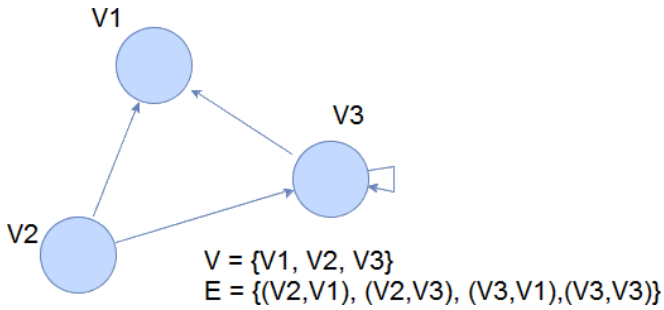
Ferramentas de transformação e geração analisam aspectos dos modelos e, com isso, podem sintetizar vários tipos de artefatos, como código fonte, simulação de entradas ou representações de modelos alternativos.

Esta capacidade de produzir código a partir de modelos ajuda a garantir a consistência entre a implementação e os requisitos funcionais descritos pelo modelo inicial. Este processo de transformação automática é conhecido como “correto por construção”, que contraria o modo de desenvolvimento de software convencional “construir por correção”, que é manual e propenso a erros.

2.2 Grafos

Um grafo (RABUSKE, 1992) é um par ordenado G = (V, E) onde V é um conjunto de vértices ou nodos e E $\subseteq$ V x V é uma relação binária sobre V, que determina um conjunto de arcos. Um grafo consiste de elementos e de uma relação bem definida sobre esses elementos conforme mostra a Figura 2.

Figura 2 - Exemplo de grafo



Fonte: Arquivo Pessoal

Grafos dirigidos são grafos em que as arestas têm direção (como no exemplo da Figura 2), são denominados dígrafos e cada aresta é definida por um par ordenado de vértices, origem e destino (RABUSKE, 1992).

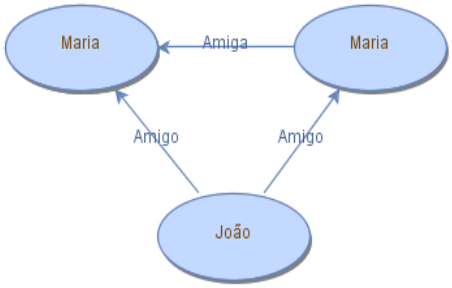
Especificar sistemas por meio de grafos possibilita obter uma semântica bem definida, por serem grafos estruturas matemáticas. Virtualmente qualquer tipo de diagrama pode ser especificado por algum tipo de grafo. Assim, é possível construir representações visuais que podem ajudar a melhorar a compreensão do problema em si. Contudo, a utilização de grafos definidos desta forma não é suficiente para especificação de modelos mais complexos da realidade, onde temos objetos de diferentes tipos com diferentes tipos de relações entre eles. Desta forma, se grafos vão ser usados para representar sistemas computacionais, temos que utilizar definições que permitam o enriquecimento do detalhamento de grafos, tais como grafos rotulados, hipergrafos e grafos tipados.

Grafos rotulados (RABUSKE, 1992) permitem que tanto os nodos quanto os arcos recebam uma informação adicional, denominada rótulo. Esse rótulo diferencia nodos e arcos com uma informação adicional que se queira representar e usualmente, nodos e arcos possuem somente um rótulo. Um grafo rotulado é uma tupla G = {V, E, Lv, Le, src, tar, labv, labe} onde:

- V é um conjunto de vértices;
- E é um conjunto de arcos;
- Lv é um conjunto de rótulos de nodos;
- Le é um conjunto de rótulos de arcos;

- src: E→V é a função de origem;
- tar: E→V é a função de destino;
- labv: V → Lv é a função de rotulação de vértices;
- labe: E→Le é a função de rotulação de arcos.

Figura 3- Exemplo de grafo rotulado



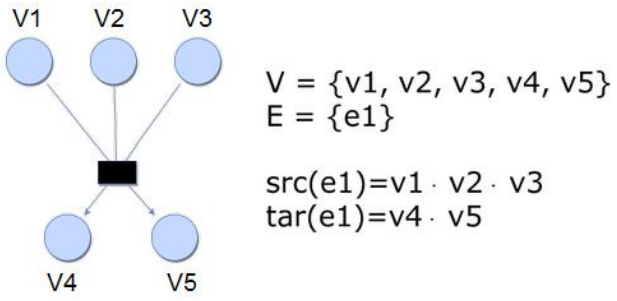
Fonte: Arquivo Pessoal

Para fins de melhor visualização, os nomes dos nodos e arestas foram omitidos na Figura 3, restando somente nodos, arestas e seus respectivos rótulos.

Hipergrafos (RABUSKE, 1992) são uma generalização do conceito de grafo, onde as arestas podem ligar quaisquer quantidades de vértices. Sua definição formal é dada por G = (V, E, src, tar), onde:

- V é um conjunto de vértices;
- E é um conjunto de hiperarcos;
- src: E→V* é a função de origem;
- tar: E→V* é a função de destino.

Figura 4- Exemplo de hipergrafo



Fonte: Arquivo Pessoal

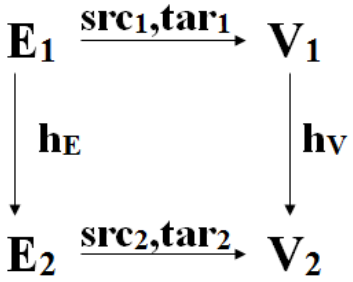
Grafos tipados (RABUSKE, 1992) são tais que cada vértice e aresta representa um tipo distinto de uma especificação. Cada grafo que faz parte de uma gramática que possui grafo tipo só pode ter vértices e arestas de tipos especificados. Um grafo tipado pode ser definido como uma tripla GT = {G, t, T}, G é um grafo tipado sobre T e t: G→T é um morfismo total de grafos, sendo um morfismo constituído por um par de funções que mapeiam os elementos dos dois grafos de forma que a origem e o destino dos arcos sejam preservadas. Um morfismo de grafos rotulados deve também garantir que os rótulos sejam preservados; um morfismo de grafos tipados deve também preservar os tipos. A definição formal de morfismo é dada por:

- dois grafos rotulados $G_1 = \{V_1, E_1, src_1, tar_1\}$ e $G_2 = \{V_2, E_2, src_2, tar_2\}$.

Um morfismo de grafos h: G_1->G_2 é um par de funções $\{h_V, h_E\}$, onde:

- h_V: V_1-> V_2 e h_E: E_1-> E_2, e onde $h_V \cdot src_1 = src_2 \cdot h_E$ e $h_V \cdot tar_1 = tar_2 \cdot h_E$.

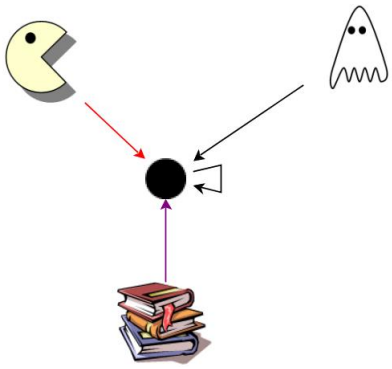
Figura 5- Definição formal de morfismo



Fonte: Arquivo Pessoal

A Figura 6 ilustra um grafo tipo criado pelo editor desenvolvido neste trabalho. Este grafo tipo serve para identificar tipos diferentes de nodos ou arestas. Neste exemplo os tipos são: fantasma, pacman, tabuleiro, aresta do pacman para o tabuleiro, aresta do fantasma para o tabuleiro, aresta entre nodos do tabuleiro. Assim, deve existir um morfismo de tipagem entre qualquer grafo da gramática e o grafo tipo, de forma que só os nodos e arestas que pertencem ao grafo tipo podem aparecer.

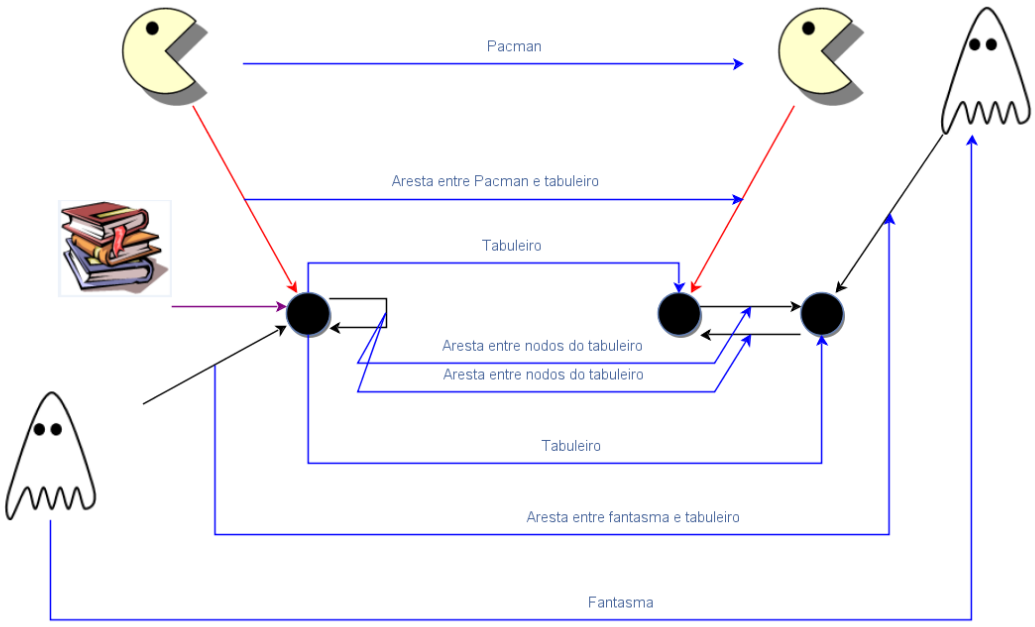
Figura 6- Exemplo de grafo tipo para o jogo Pacman



Fonte: Arquivo Pessoal

Na Figura 7 é apresentado um exemplo de um grafo que pertence a mesma gramática do grafo tipo da Figura 6. Neste exemplo é possível observar que todos os elementos mais a direita, que correspondem ao grafo da gramática, podem ser mapeados pelo grafo tipo, que corresponde aos elementos mais a esquerda. Esse mapeamento é representado na Figura 7 por um arco rotulado, contendo a descrição do nodo ou aresta que é mapeado do grafo tipo. Pode-se observar ainda que nem todos os elementos do grafo tipo (esquerda) estão presentes no grafo tipado (direita), porém todos os elementos do grafo tipado (direita) devem estar presentes no grafo tipo (esquerda).

Figura 7- Exemplo de mapeamento do grafo tipo



Fonte: Arquivo Pessoal

2.3 Gramáticas de Grafos

Grafos são formais e intuitivos ao mesmo tempo, por isso surgem como uma boa solução para a formalização de problemas. Com isso, podem ser utilizados na representação de estruturas de sistemas e também descrever seu comportamento, caso seja utilizado um formalismo capaz de modelar a evolução ao longo do tempo. E como são descritos por linguagem de especificação formal, podem ser traduzidos para outras linguagens formais, como linguagens de programação e linguagens de entrada para verificadores de modelos. Através dessas traduções, muito do processo de codificação e testes pode ser realizado de forma totalmente automática evitando erros, acelerando o processo de desenvolvimento e mantendo a qualidade do software.

Gramáticas de grafos constituem um modelo formal de computação baseado em regras de transformação de grafos (FERREIRA; RIBEIRO, 2006). Como suas estruturas são descritas formalmente, se expressam de forma não ambígua e descrevem exatamente o que se quer dizer. Diversos tipos de gramáticas de grafos podem ser construídas, dependendo do tipo de grafo usado para construção e da estrutura de suas regras. Uma gramática de grafos é uma generalização de gramáticas de Chomsky (MARTIN, 1996), transformando grafos em vez de sequências de símbolos. Da mesma forma, uma gramática de grafos é composta de regras com um lado esquerdo e um lado direito.

A definição formal de uma gramática de grafos tipada é dada por uma tupla GG = (I, P, T) onde:

- I é o grafo inicial da gramática;
- P é um conjunto finito de regras;
- T é o grafo tipo da gramática.

A construção de uma gramática de grafos dá-se por meio da criação de um grafo inicial e de regras da gramática. Assim como a execução de uma gramática inicia com o grafo inicial, que é modificado através de repetidas aplicações de regras. Esse processo de aplicação de regras é denominado derivação. Para que se possa descrever a execução de uma gramática de grafos são necessários elementos que descrevam o comportamento dinâmico das estruturas. Isso é realizado através das regras da gramática.

Uma regra de gramática de grafos é uma tupla p = {L, r, R} onde:

- L e R são grafos tipados sobre T;
- r: L→R é um morfismo parcial de grafos tipados.

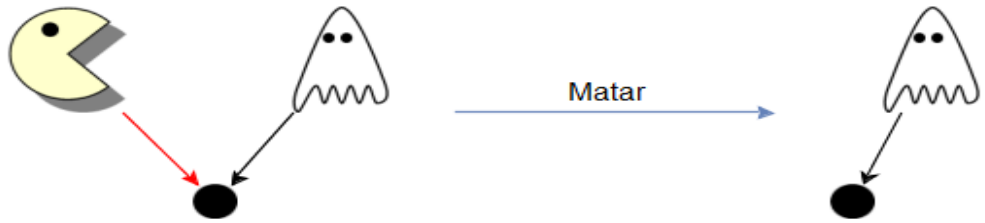
As regras de gramáticas são compostas pelo grafo L, pelo grafo R e por p, morfismo parcial de grafos. Um morfismo parcial é um sub-grafo que mapeia vértices e arestas de L para R de forma compatível, ou seja, cada vez que uma aresta *aL* contida no grafo L for mapeada para uma aresta *aR* do grafo R, então o vértice origem e o vértice destino de *aL* devem ser os mesmos em *aR* (GUMS, 2009). L e R representam o lado esquerdo e o lado direito da regra, respectivamente, e o processo de derivação de uma determinada regra, ou seja, a substituição da ocorrência (lado esquerdo) pelo lado direito é descrita como um morfismo de grafos entre o lado esquerdo e o grafo que representa o sistema.

Dada uma regra r: L -> R,

- Os elementos que pertencem ao domínio de r (arcos e nodos) são os elementos preservados pela regra;
- Os elementos que não pertencem ao domínio de r são os elementos apagados pela regra;
- E os elementos que não pertencem à imagem de r no grafo R são os elementos inseridos pela regra.

A Figura 8 exemplifica uma regra de gramática utilizada para especificar um movimento do jogo Pacman. Quando o pacman se encontra com um fantasma, neste caso, deve ser eliminado. É possível observar que o nodo, a aresta e o fantasma são os elementos preservados pela regra, enquanto o pacman e sua aresta são elementos apagados. Não há elementos criados por esta regra exemplo.

Figura 8- Exemplo de regra



Fonte: Arquivo Pessoal

No editor de gramáticas de grafos desenvolvido neste trabalho, é possível construir três tipos distintos de gramáticas:

I. Gramáticas de grafos;
II. Gramáticas de grafos tipados;
III. Gramáticas de grafos orientados a objeto.

As gramáticas de grafos não possuem restrições, ou seja, seu grafo inicial pode possuir qualquer tipo de elemento e suas regras podem inserir, deletar ou preservar quaisquer elementos. Isso não ocorre nas gramáticas de grafos tipados. Neste caso, primeiro é preciso especificar um grafo tipo, que irá conter todos os tipos de vértices e arestas que estarão disponíveis na gramática para construção do grafo inicial e regras. Se não for possível mapear todos os vértices e arestas presentes em uma determinada regra ou grafo inicial para o grafo tipo, então o grafo inicial ou regra é considerado inválido e não pertence a gramática de grafos tipados. As gramáticas de grafos orientados a objeto, por sua vez, acrescentam características de orientação a objetos sobre as gramáticas de grafos tipados e, com isso, permitem modelar de forma intuitiva sistemas orientados a objeto. Isso é possível considerando objetos ou classes como vértices e atributos e mensagens como arestas (MOREIRA, 2007).

2.4 Trabalhos Relacionados

Esta seção contém uma análise sobre três ferramentas que possuem funções de edição de grafos, destacando suas principais funcionalidades e limitações.

2.4.1 GenGED

A ferramenta de metamodelagem GenGED é destinada a definição e manipulação de linguagens visuais de forma rápida e de fácil utilização. Para implementação de formalismos visuais, o GenGED possui integração com uma ferramenta de *layout* chamada Parcon. Isso permite que a tarefa de posicionamento e reposicionamento ocorra sem a necessidade de codificação. Contudo, a utilização do Parcon inviabiliza seu uso, pois apenas os binários estão disponíveis e estes só funcionam em versões muito antigas do Linux e Solaris, desta forma o GenGED não pode ser testado.

O GenGED pode ser descrito por meio de seus três principais componentes, o editor de alfabetos, o editor de regras de gramáticas e o editor de simulação e animação. Como este trabalho se propõe a criar um editor de gramáticas, que tem como objetivo a edição de regras de gramáticas de grafos, as funcionalidades do GenGED de simulação e animação não foram analisadas.

No editor de alfabetos são especificados os símbolos e as relações, dado um formalismo específico. Segundo Bardohl (1998), o processo de especificação de alfabetos é realizado por meio de outros três editores. O primeiro, *Graphical Object Editor*, é responsável por definir símbolos visuais a cada entidade do formalismo. No segundo editor, *TypiEditor* as entidades do formalismo a ser criado são atribuídas a um símbolo visual definido no Editor Gráfico de objetos ou a um espaço reservado não-visual. Além disso, atributos visuais, tais como strings para os nomes, são instanciados, embora eles não estão ligados a qualquer entidade neste momento. Por fim, no *ConEditor* deve ser especificado a atual relação entre as entidades, incluindo qual atributo pertence a qual entidade.

Já o editor de regras, *Grammar Rules Editor*, é responsável pela especificação de regras de gramática para o formalismo. Para criação de regras de gramáticas, o GenGED oferece 3 regras básicas, que são construídas de acordo com o alfabeto, e são denominadas respectivamente *insert*, *deletion* e *attribute modification*. A partir destas regras básicas é possível construir regras mais complexas sem grandes dificuldades.

O formato de exportação de arquivos do GenGED é o svg que pode ser utilizado em outras ferramentas que suportem o arquivo.

2.4.2 Draw.io

Draw.io é uma ferramenta de desenho de diagramas online, gratuita e dos mesmos desenvolvedores das bibliotecas JGraph. Ela apresenta uma interface intuitiva e diversas variedades de componentes para criação de diagramas. Diagramas foram criados e exportados em diversos formatos para que pudesse ser feito uma análise das principais funcionalidades da ferramenta.

Uma de suas principais características é apresentar componentes especiais para criação de diagramas UML, como caixas de texto e tabelas, assim como componentes e ícones presentes em sistemas operacionais modernos como o Android e IOS, que são sistemas para dispositivos móveis amplamente utilizados. Vale destacar que por ser executada direto no browser do usuário, não requer instalação e pode ser utilizada em qualquer sistema operacional, contudo só pode ser utilizada se uma conexão com a internet estiver disponível.

Com relação à importação e exportação de arquivos, o Draw.io se mostra repleto de possibilidades. Ao iniciar a ferramenta, ou seja, acessar o endereço *web*, é apresentado ao usuário uma janela em que deve ser escolhido o local de armazenamento dos diagramas. Dentre as opções de armazenamento disponíveis estão as ferramentas de armazenamento em nuvem Dropbox e Google Drive, bem como as opções de salvar no browser e no computador. Escolhido o local onde serão armazenados os diagramas e criado um novo diagrama, os diagramas criados podem ser exportados em diversos formatos como png, xml, gif, pdf, svg, jpg ou com a opção de exportação avançada que permite a exportação nos mesmo formatos, porém com formatação de tamanho e cor de fundo.

2.4.3 Grafio

Grafio é um aplicativo para *tablets* e *smartphones* destinado a criação de diagramas de forma rápida e eficaz. Ele conta com inúmeras funções aliadas ao *touchscreen* dos dispositivos atuais, tornando-se uma ferramenta poderosa para criação de diagramas. No próprio aplicativo, o usuário pode desenhar ou utilizar objetos existentes como base para formação do seu diagrama. Contando com um sistema de reconhecimento de formatos, ou seja, ao usuário desenhar algo parecido com um círculo,

o aplicativo substitui o desenho por um círculo perfeito, isso ocorre para as outras formas geométricas e para outros componentes padrão do aplicativo.

O Grafio permite notações de áudio em seus objetos, podendo ser utilizado para descrever o objeto ou para qualquer outra função desejada. Assim como é possível transformar o diagrama em uma apresentação de vídeo facilmente, esse vídeo de apresentação é criado conforme a ordem de criação do documento. Cada novo objeto gera um novo frame e todos as notas de áudio podem ser incluídas.

Com relação a importação de arquivos, o aplicativo que foi testado em um iPad de quarta geração, possuía três formas de importação. A primeira opção para importar diagramas se dá através do iTunes, software da Apple responsável pela interface entre iPad e PC, responsável por gerenciamento de arquivos e aplicativos. Além do iTunes, os sistemas de armazenamento em nuvem Dropbox e Box também podem ser utilizados para importação de trabalhos anteriores.

Quanto a exportação de arquivos, o aplicativo oferece diversos formatos e modos de exportação. Os formatos em que os projetos podem ser exportados são PDF, PNG, jPeg, MOV no caso de geração de vídeo e, por fim, um formato especial da própria aplicação, Grafio Document (iddz). Escolhido o formato é possível enviar o arquivo por e-mail, fazer upload nas aplicações de armazenamento em nuvem Dropbox e Box, copiar para o iTunes ou no caso de exportação no formato de imagem ou vídeo, ainda é possível armazenar no dispositivo ou publicá-lo nas redes sociais Facebook e Twitter.

3. METODOLOGIA

Para a implementação inicial do editor foi necessária a utilização de ferramentas de apoio. Este capítulo apresenta as ferramentas utilizadas e as etapas seguidas.

3.1 Etapas de execução

Etapa 1: levantamento bibliográfico

Nesta etapa foi realizado um estudo de materiais bibliográficos que abordassem MDE, grafos, gramáticas de grafos e desenvolvimento de software. Também foi realizada uma busca por editores de gramáticas de grafos já existentes.

Etapa 2: análise de editores de gramáticas

Esta etapa se efetivou a partir da análise das ferramentas encontradas na etapa anterior. Dentre os diversos editores de diagramas e ferramentas para editar gramáticas de grafos encontrados, foram analisados apenas os que apresentavam características mais semelhantes aos objetivos deste trabalho. O objetivo é a observação de principais funcionalidades como tipo de gramática, tipo de regras, *layout*, limitações e formato de armazenamento e/ou exportação.

Etapa 3: levantamento de requisitos

Durante esta etapa foram levantados requisitos funcionais e não funcionais para que a ferramenta fosse desenvolvida. Como referência, foram utilizados os resultados da etapa anterior que previa observar funcionalidades de ferramentas existentes em relação à criação e edição de gramáticas de grafos.

Etapa 4: ambiente de desenvolvimento do protótipo

Desenvolver uma aplicação com interface visual atrativa pode ser uma tarefa árdua se não forem utilizadas as ferramentas corretas. Durante esta etapa algumas IDEs

foram analisadas, verificando seu suporte para desenvolvimento de interface. Paralelamente, foi realizada uma busca por bibliotecas abertas que auxiliassem no desenvolvimento de aplicações que manipulassem grafos.

Etapa 5: desenvolvimento do protótipo

A etapa de desenvolvimento do protótipo primeiramente se deu através da criação de uma estrutura básica que permitia a criação de grafos. Em seguida foram implementadas funcionalidades padrão de qualquer tipo de editor como iniciar, salvar e carregar projetos. A função de salvar grafos foi o próximo recurso a ser implantado, permitindo que grafos fossem salvos em formatos como png e xml, possibilitando que pudessem ser carregados e editados posteriormente. No final desta etapa, os métodos de criação e edição de grafos foram modificados para diferenciar os três tipos de gramáticas que deveriam ser suportados, liberando ou bloqueando recursos.

Etapa 6: desenvolvimento de função para exportação de gramáticas em OOGXL

Por fim, a última etapa consistiu no desenvolvimento de um método que realiza a exportação de gramáticas de grafos orientados a objeto para a linguagem OOGXL.

3.2 Ferramentas utilizadas

Nesta seção é apresentada uma descrição das ferramentas utilizadas no auxilio da construção do editor de gramáticas.

3.2.1 NetBeans

O NetBeans (versão 8.0) é um IDE gratuito e de código aberto. Foi utilizado para a implementação do editor proposto neste trabalho por possuir um grande número de bibliotecas, módulos e APIs que facilitam o desenvolvimento de aplicações. Este IDE, além de possuir uma vasta documentação em português, permite uma rápida prototipação de janelas e proporciona o desenvolvimento de interface gráfica em alto nível, permitindo

uma visualização imediata do visual da aplicação, sem a necessidade de programação manual, evitando erros e acelerando o processo de desenvolvimento.

3.2.2. JGraph

Para que seja possível criar gramáticas de grafos, é necessário primeiro poder desenhar grafos. Os grafos devem poder ter seus componentes criados, removidos e remanejados em tempo real. A implementação de um painel que suporte todas essas funções básicas de grafos, como o desenho das arestas, nodos, ligações entre nodos que devem ser reconstruídas a cada vez que um nodo é remanejado, pode ser realizada de forma simples e rápida por meio da biblioteca JGraph. Esta biblioteca, que possui funcionalidades voltadas a desenho de grafos e código aberto, foi desenvolvida na linguagem Java e permite criar nodos e vértices, bem como inserir ligações entre componentes, adicionar rótulos, reposicionar elementos mantendo as ligações existentes e outras funcionalidades mais avançadas.

4. DESENVOLVIMENTO

4.1 Introdução

Para representarmos um sistema como modelo visual, mais precisamente como uma gramática de grafos, de forma intuitiva e atrativa ao usuário, se faz necessário o desenvolvimento de uma interface simplificada, porém ao mesmo tempo repleta de recursos que permitam a criação de gramáticas. Buscando atender a esses requisitos básicos, foi necessário realizar definições e análises, que possibilitaram o desenvolvimento do protótipo.

O protótipo visa uma solução para a especificação de sistemas em alto nível através de gramáticas de grafos, gramáticas de grafos tipados e gramáticas de grafos orientados a objeto, onde não só a edição de gramáticas será possível, como também a exportação em um formato específico para validações e geração de código de forma automática. A linguagem escolhida para o desenvolvimento deste protótipo foi o JAVA, por ter suporte para os mais diversos sistemas operacionais e contar com bibliotecas que facilitam a criação e manejo de grafos, da mesma maneira que facilitam a manipulação de arquivos XML.

Para que o protótipo pudesse ser implementado, foi realizado um levantamento de requisitos que resultou em uma lista de requisitos funcionais e não funcionais. Os requisitos que a ferramenta deve atender estão listados a seguir:

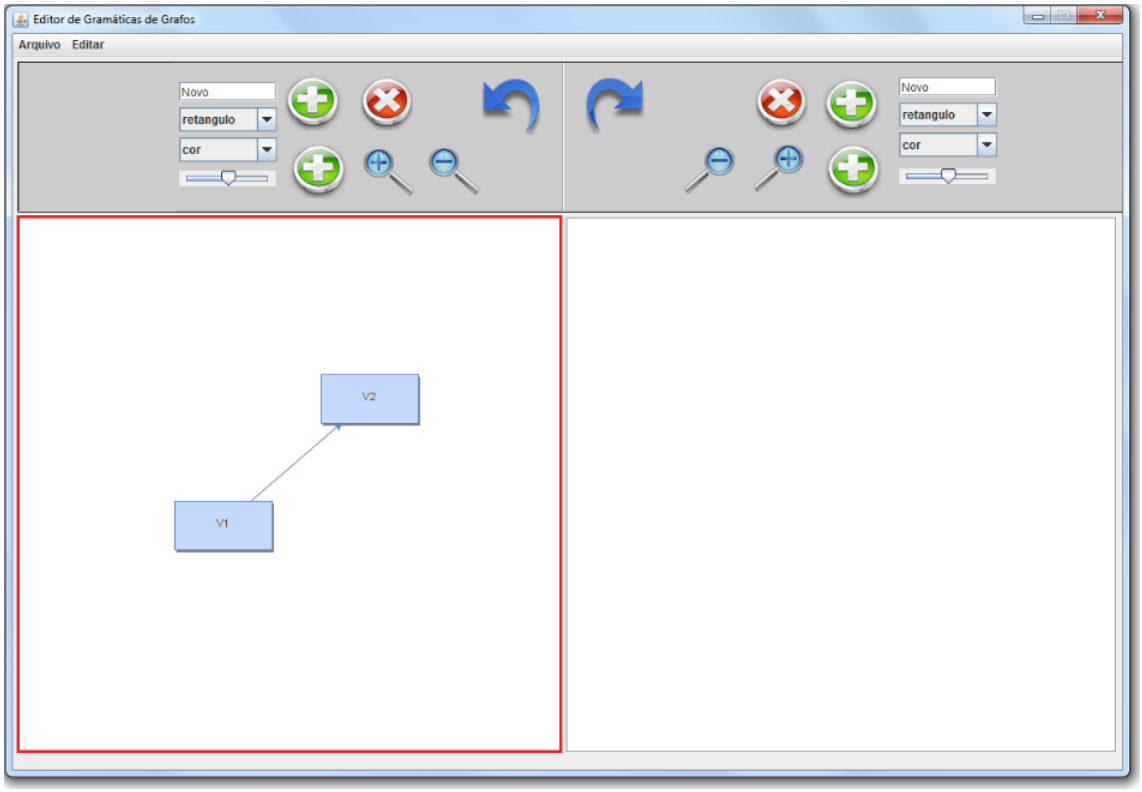
- O editor deve permitir ao usuário a criação de vértices;
- O editor deve permitir ao usuário a criação de arestas;
- O editor deve permitir ao usuário a criação de um grafo inicial;
- O editor deve permitir a criação de gramáticas de grafos tipados e de grafos tipos;
- O editor deve permitir que tipos possam ser especificados por meio de gráficos, em formatos usuais de representação de imagens;
- O editor deve permitir ao usuário a criação de regras de gramática;
- O editor deve permitir a exportação de arquivos no formato OOGXL;
- Apresentar interface amigável de utilização.

4.2 Protótipo Inicial

A implementação do editor teve como passo inicial um estudo sobre a biblioteca JGraph. Esse estudo foi realizado através da análise de exemplos disponíveis no site oficial da biblioteca e por meio de leitura da documentação da biblioteca. Testes iniciais foram realizados com códigos exemplos de JGraph até que o projeto do editor fosse iniciado.

O início do projeto se deu por meio da criação de uma classe *JFrame* na IDE NetBeans e a partir desta classe java, foram dados os primeiros passos no desenvolvimento. O primeiro componente a ser inserido na IDE foi um painel especial da biblioteca JGraph, ilustrado pela Figura 9, onde os grafos são exibidos.

Figura 9- Painel tipo mxGraphComponent

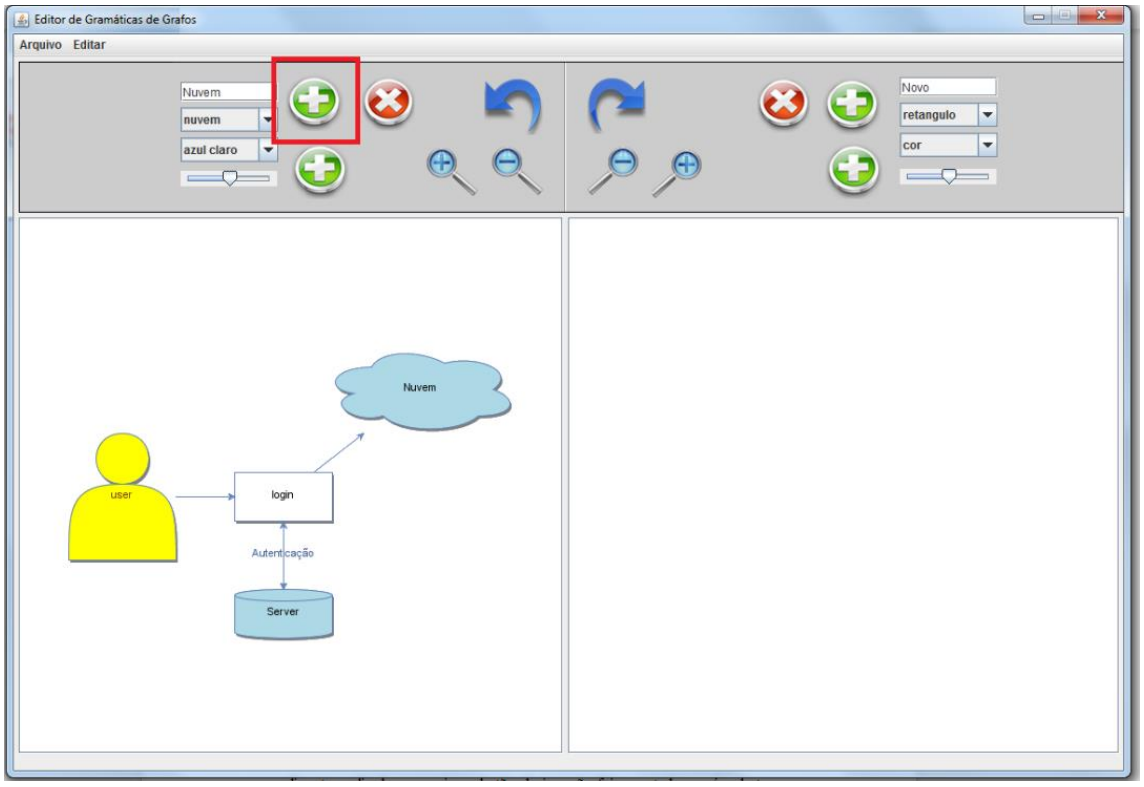


Fonte: Arquivo Pessoal

Na Figura 9 existem dois painéis do tipo *mxGraphComponent*, para ilustração ficar clara, apenas o da esquerda, que é responsável por representar o lado esquerdo de uma determinada regra, foi destacado.

Após ser possível, por meio de modificações em linha de código, inserir um elemento no painel de grafos, foi adicionado um botão (padrão da IDE), conforme mostra a Figura 10, para que uma função genérica de inserção no painel fosse chamada no momento em que o botão fosse pressionado. Os componentes que eram adicionados ao painel naquele momento eram somente nodos, visto que para inserir uma aresta basta selecionar um nodo origem e clicar em um nodo destino. Essas funcionalidades como inserção de arestas e rotulação de nodos e arestas são nativas da biblioteca *JGraph* e não precisam ser implementadas. Já para rotular um grafo ou uma aresta basta clicar duas vezes sobre o elemento. O método de inserção de nodos foi posteriormente transferida para uma outra classe java a fim de manter o código limpo e organizado.

Figura 10- Inserir novo nodo

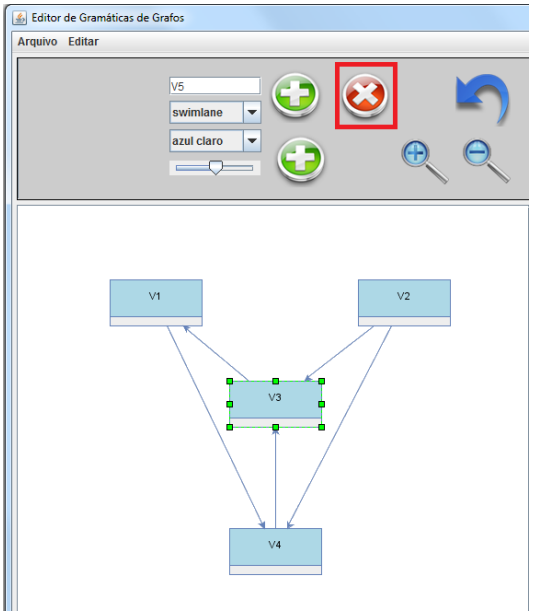


Fonte: Pessoal

Com o método de inserção funcionando de forma genérica, foi implementada (também por linha de código) uma função de remoção de componentes para que fossem realizados testes. Após isso, com o código de exclusão já funcionando corretamente, o mesmo procedimento realizado para criar o botão de inserção foi executado, porém, desta vez, o método chamado pelo clique do botão foi o de exclusão de elementos. A função

foi implementada de maneira que, quando um elemento for ser excluído, basta selecionar esse elemento (sendo ele nodo ou aresta) e clicar no botão de exclusão. Se um nodo for excluído, todos as arestas que partem ou apontam para este nodo são deletadas juntamente com o mesmo. As Figuras 11 e 12 exemplificam uma exclusão de um nodo no editor.

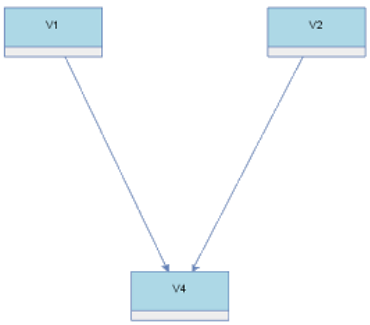
Figura 11- Deletar nodo



Fonte: Arquivo Pessoal

Após selecionar o nodo do centro da imagem (V3) e clicar na opção deletar, ou simplesmente usar a tecla delete do teclado, o elemento é excluído juntamente com as arestas que o tem como origem ou destino, conforme mostra a Figura 12. Para melhor visualização, apenas a metade esquerda do editor é mostrada nas Figuras 11 e 12.

Figura 12- Elemento excluído



Fonte: Arquivo Pessoal

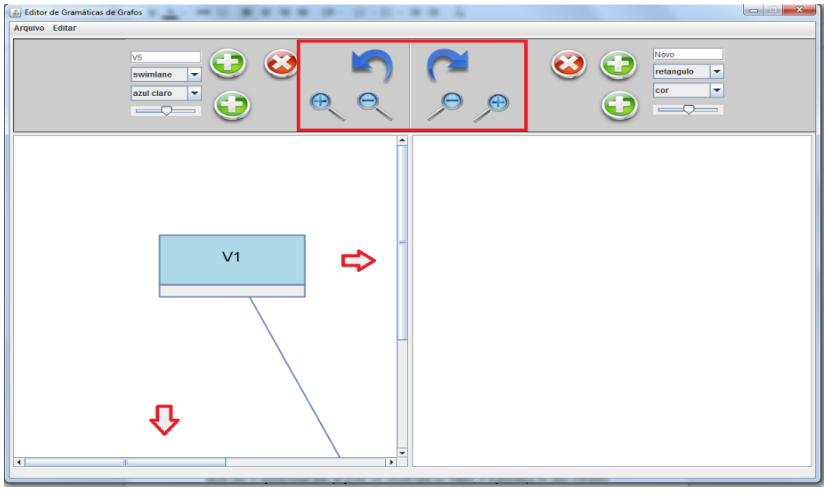
Com funcionalidades de inserção e exclusão de nodos e arestas operando corretamente, tornou-se possível o desenho de grafos no painel de grafos principal. Deste ponto em diante, o editor começou a ter sua interface desenvolvida, para que pudesse ser atrativo e funcional para qualquer usuário. Somando-se a isso, um novo painel de desenho de grafos foi inserido e colocado à direita do painel já implementado, esse painel tem como objetivo representar o lado direto das regras criadas. Nesse momento, foram criados os mesmos botões de inserção e exclusão de elementos, até que ambos os painéis pudessem ser editados de forma igual.

A partir do momento em que se pôde editar e criar o lado direito e esquerdo de uma regra de gramática, já era possível identificar um esboço de editor de gramáticas de grafos em funcionamento. Assim, por meio dos estudos de trabalhos correlatos, foi possível identificar características essenciais a esse tipo de ferramenta, como as funcionalidades desfazer, refazer, aumentar zoom no painel, diminuir zoom no painel e inserir e excluir nodos por atalhos do teclado. Para implementar essas funcionalidades iniciou-se um novo estudo mais profundo sobre a biblioteca JGraph.

Utilizando uma combinação de funções dessa biblioteca, foram implementadas funcionalidades que visavam facilitar a especificação de sistemas pela ferramenta desenvolvida. Primeiramente, foram criados dois botões para atenderem as funções de desfazer e refazer. Um fato a ser observado é que o desfazer e refazer segue a ordem de todos os remanejamentos de nodos, arestas, criação e exclusão de nodos e arestas, bem como rotulações dos dois painéis de grafos, esquerdo e direito. Assim como, se o primeiro procedimento for criar um nodo no lado esquerdo, seguido de um nodo do lado direito, o método desfazer eliminará, quando acionada pela primeira vez, o nodo no painel esquerdo e, se pressionada novamente, o nodo no painel da direita. Estes dois (desfazer e refazer) são os únicos botões que aplicam ações em ambos os painéis, como mostra a Figura 13.

Sabe-se que conforme um grafo vai sendo desenhado, o espaço necessário para continuar o desenho aumenta, por isso foram implementadas as funções de aumentar e diminuir zoom, assim como a adição de um sistema de rolagem nas bordas de cada painel de grafos. Ademais, por questões de melhor interação com o usuário, existem dois botões de cada tipo de zoom, um destinado ao painel esquerdo e outro ao direito. Uma adição futura a ferramenta será a opção de abrir uma nova janela em que apenas um lado da regra seja editado por vez, visando facilitar a edição de regras.

Figura 13- Desfazer, refazer, aumentar e diminuir zoom



Fonte: Arquivo Pessoal

Após a inserção desses botões, e com um ambiente de especificação de gramáticas funcional, foi realizada uma bateria de testes, procurando por *bugs* e eventuais falhas que pudessem causar erros durante a edição de regras de uma gramática. Concluídos esses testes, foram observados alguns problemas causados por erros pontuais de programação. Esses erros foram removidos, possibilitando o avanço do desenvolvimento do editor, que neste momento estava com suas funcionalidades básicas completas em relação à edição de grafos.

Com o mesmo propósito de tornar a ferramenta mais interessante, foram adicionados campos selecionáveis que pudessem agregar mais informações aos nodos no momento de sua inserção. Esses novos recursos possibilitaram criar um nodo com diversos formatos, cores, tamanhos, e rótulo previamente selecionados pelo usuário, conforme mostra a Figura 14.

Figura 14- Campos de criação de novo nodo

Fonte: Arquivo Pessoal

A próxima etapa de desenvolvimento do protótipo consistiu na criação de ícones mais adequados para cada função, alteração de cores, tamanhos e remanejamentos de

botões visando tornar o visual da ferramenta mais amigável. Junto com a mudança de visual foi desenvolvido um método que adapta o tamanho das janelas e redistribui os componentes do editor, fazendo com que ele se ajuste a resolução da tela do usuário. Além disso, um novo recurso de inserção de nodos foi adicionado, que conta com uma função especial que permite ao usuário escolher uma imagem disponível em seu computador (em formato PNG ou JPEG) e utilizá-la como se fosse um dos tipos padrão disponíveis para seleção. Uma vez que o usuário adiciona um novo elemento a partir deste modo de inserção, a imagem utilizada é automaticamente salva pelo editor e passa a fica disponível na lista de formatos de vértices. Na Figura 14, mais especificamente no segundo campo, será possível selecionar o novo componente sempre que o usuário desejar.

4.3 Adição de Características de Editor

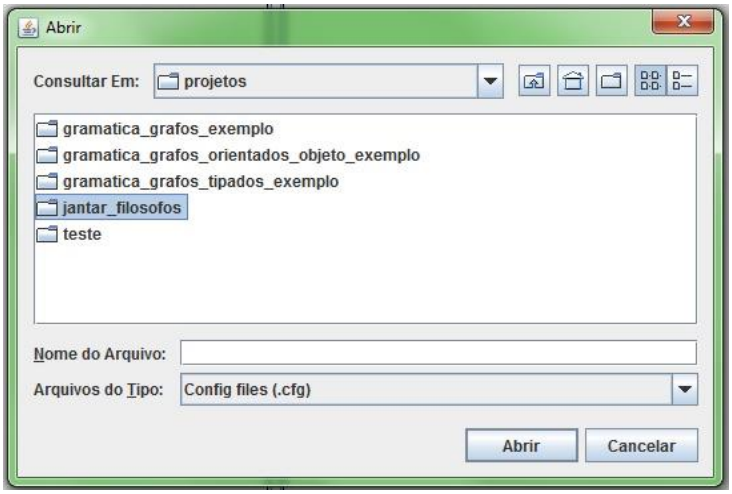
Apesar do editor ter recursos suficientes para criar e editar regras de gramáticas de grafos, ainda não era possível criar projetos de gramáticas de grafos propriamente ditos, por isso a etapa seguinte resultou na inserção de um menu de opções, no qual foram inseridas funcionalidades características de um editor, como mostra a Figura 15.

Figura 15- Menu e barra de ferramentas do editor

Fonte: Arquivo Pessoal

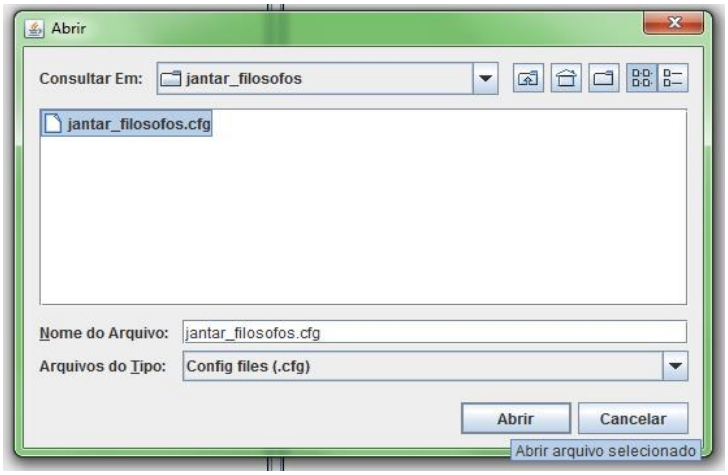
O primeiro recurso desenvolvido foi a opção **Novo Projeto**, que consistiu na implementação de um método que cria um diretório específico para o novo projeto e, abaixo deste diretório, um arquivo (de formato cfg) que contém informações sobre a configuração do projeto. Este mesmo arquivo cfg é utilizado para abrir projetos. Quando a opção **Abrir Projeto** é acionada, um *jFileChooser* será prototipado permitindo ao usuário navegar em seus diretórios e selecionar o projeto desejado conforme mostram as Figuras 16 e 17.

Figura 16- Abrir projeto



Fonte: Arquivo Pessoal

Figura 17- Arquivo cfg para o projeto jantar_filosofos



Fonte: Arquivo Pessoal

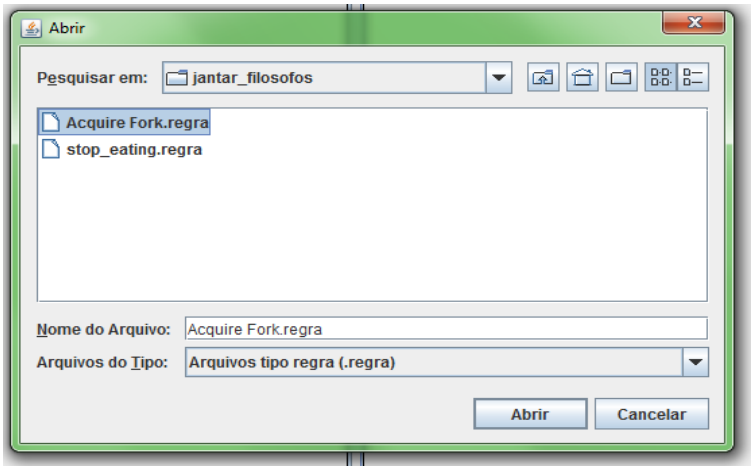
O diretório que for criado em novo projeto ou selecionado em abrir projeto irá conter todos os arquivos relacionados a gramática em questão. Em sequência foram implementadas as opções **Fechar** e **Sair**, também no menu arquivo, que fecham o projeto aberto e fecham o editor, respectivamente. Com esses novos recursos era possível abrir, criar e fechar projetos, características básicas de qualquer tipo de editor atual.

4.4 Criação de Gramáticas de Grafos

No capítulo 2 foi visto que gramáticas de grafos são compostas por um grafo inicial e um conjunto de regras, portanto o editor precisa ter capacidade de salvar e restaurar os grafos. Como o editor já possui todos os métodos necessários para criar e editar nodos e arestas, foram desenvolvidos dois novos métodos de manipulação de arquivos. O primeiro método permitia salvar o conteúdo de um *mxGraphComponent* (componente que contém o lado esquerdo ou direito da regra) em um arquivo XML. Já o segundo, permite que seja realizada leitura de um arquivo XML e este tenha seu conteúdo inserido em um *mxGraphComponent*.

Com esses métodos de manipulação de arquivos, foram criadas as opções **Carregar regra** e **Salvar** regra, presentes no menu arquivo. A opção salvar regra fará com que sejam criados no diretório do projeto os arquivos NomeDaRegra_esq.xml, NomeDaRegra_dir.xml e NomeDaRegra.regra. O arquivo NomeDaRegra.regra possui o caminho dos dois XMLs que descrevem os grafos do lado direito e do lado esquerdo da regra e deve ser selecionado quando o usuário desejar carregar uma regra já construída, de forma semelhante ao processo de abrir projetos. A Figura 18 mostra a janela prototipada quando o usuário clica na opção **Carregar regra**, note que apenas arquivos de extensão REGRA são visíveis.

Figura 18- Regras salvas no projeto jantar_filosofos



Fonte: Arquivo Pessoal

Este modo de salvar uma determinada regra, em dois arquivos e com um terceiro como referência a importação de regras, permite que os lados de regras sejam reaproveitados

futuramente com a opção Importar XML, presente no menu Importar. Além disso, armazenar os lados separadamente facilita a escrita e leitura dos arquivos, simplificando o processo de armazenamento e a codificação necessária.

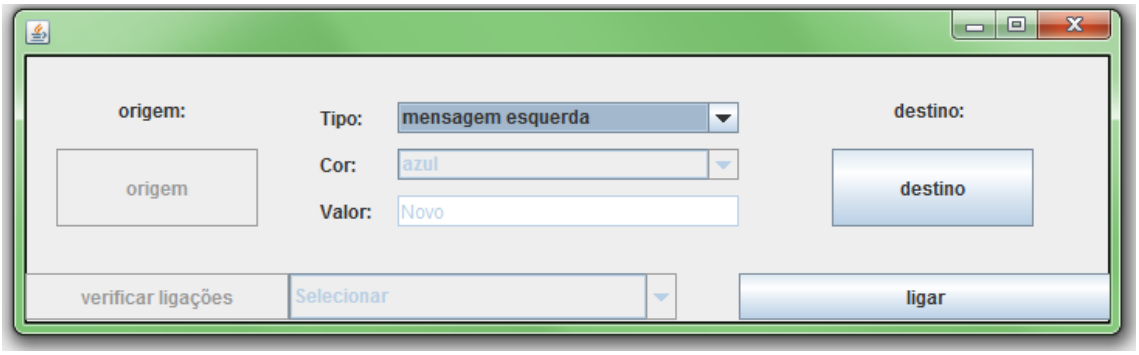
A construção do grafo inicial foi implementada de forma semelhante à criação e edição de regras. Foram criadas opções no menu arquivo em que o usuário pode carregar, construir ou salvar um novo grafo inicial. O arquivo que armazena o grafo inicial tem nome fixo de Grafo_inicial.xml e não deve ser alterado. Este arquivo também pode ser encontrado na pasta projetos, dentro dos diretórios de cada projeto. Vale destacar que, quando uma nova gramática de grafos é construída o arquivo Grafo_inicial.xml é criado, pois a função cria grafo inicial é disparada automaticamente. Isto significa que o usuário pode construir o grafo inicial logo que criar uma gramática, porém se desejar construir as regras primeiro, basta iniciar uma nova regra e o grafo tipo será salvo como um grafo vazio.

4.5 Construções de Gramáticas de Grafos Tipados e Orientados a Objeto

O editor possui recursos para criar e editar grafos iniciais e regras. Isso é suficiente para construir gramáticas de grafos, porém ainda não é o bastante para gramáticas de grafos tipados e orientados a objeto. Essas gramáticas são diferenciadas, pois necessitam de uma informação extra, o grafo tipo, que realiza tipagem dos elementos da gramática. Este grafo, apresentado na sessão 2.2 deste trabalho, define quais são os tipos de vértices e arestas disponíveis e, portanto, limita a construção do grafo inicial e regras da gramática. Quando um novo projeto de gramáticas de grafos tipados ou orientados a objeto é construído, automaticamente é iniciado um novo grafo tipo para o usuário editar. Este grafo tipo não possui restrições, qualquer tipo de vértice pode ser inserido e qualquer tipo de ligação disponível para gramáticas de grafos pode ser efetuada. Contudo, existem algumas diferenças entre gramáticas de grafos tipados e orientados a objeto. Podemos considerar que gramáticas de grafos orientados a objeto são uma extensão das gramáticas de grafos tipados. Em um grafo tipo das gramáticas de grafos orientados a objeto podem ser criados vértices especiais que representam trocas de mensagens entre processos e ligações especiais que caracterizam herança de classes. Para adicionar mensagens o usuário deve clicar na opção inserir nova ligação e buscar por mensagem, selecionar o

nodo que receberá a mensagem e informar qual será seu valor, conforme mostra a Figura 19. Note que existem dois tipos de mensagem, mensagem esquerda e mensagem direita, porém essa diferença se resume apenas ao layout da mensagem. Para ligar uma mensagem do lado esquerdo de um nodo seleciona-se mensagem esquerda, do contrário mensagem direita.

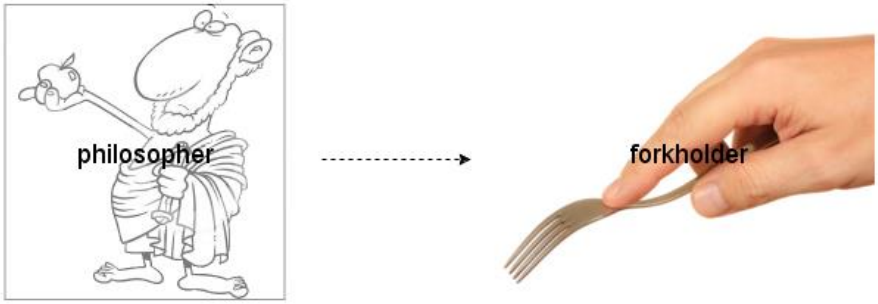
Figura 19- Inserir nova mensagem



Fonte: Arquivo Pessoal

Para representar características de herança foi criado um tipo único de seta que deve ser selecionada na janela de inserir novas ligações. Ao selecionar a opção pontilhada, no campo tipo, os campos cor e valor serão desabilitados, restando ao usuário selecionar origem, destino e clicar em ligar. Essa seta tem coloração preta, é tracejada e não possui rotulação conforme mostra o exemplo da Figura 20.

Figura 20- *Philosopher* herdando *forkholder*



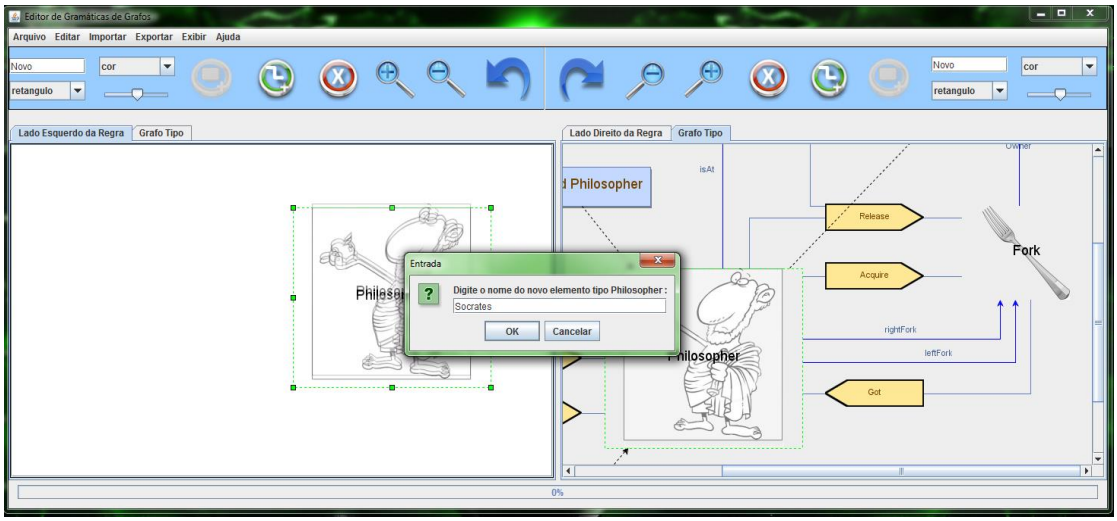
Fonte: Arquivo Pessoal

Construído o grafo tipo, o usuário pode iniciar a construção de regras ou do grafo inicial. Para gramáticas de grafos orientados a objeto e grafos tipados o editor não permite que o

usuário adicione vértices da forma convencional, deste modo os botões de inserção serão desabilitados. Isso ocorre devido as limitações impostas pelo grafo tipo, que determina os elementos que podem existir ou não e quais são as ligações entre eles. Para adicionar elementos em regras e no grafo inicial de gramáticas que possuem grafo tipo, o editor apresenta um novo modo de inserção que permite apenas a importação de elementos diretamente do grafo tipo. Este método de importação precisa ser rápido e prático ao usuário, deste modo foram realizados ajustes na interface do editor, resultando na inserção de abas nos painéis que representam os lados das regras. Ambos os lados possuem uma aba que contém o grafo que representa o lado da regra e outra com o grafo tipo. Deste modo o usuário pode acessar o grafo tipo a qualquer momento e tudo que precisa fazer para adicionar um componente a um lado da regra ou grafo inicial é arrastá-lo do grafo.

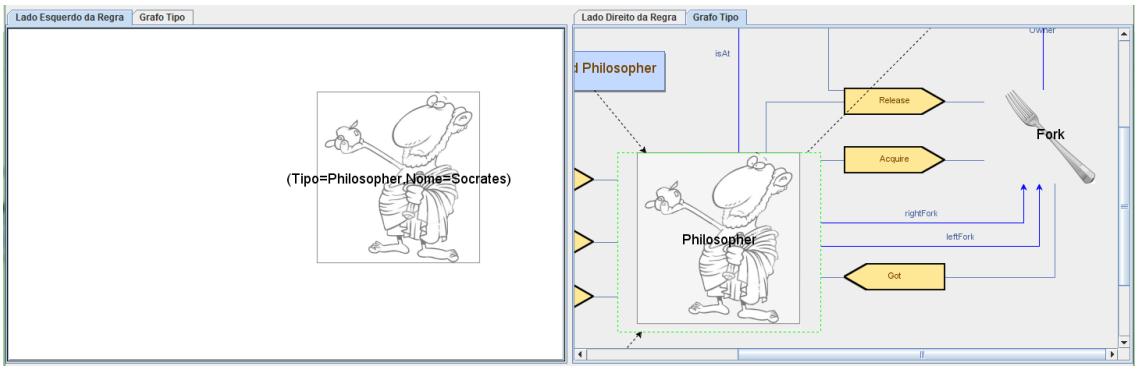
As Figuras 21 e 22 demonstram o processo de importação de nodos e arestas. Note que quando um elemento é arrastado do grafo tipo para um dos lados da regra, imediatamente uma janela é prototipada solicitando que seja informado o nome do componente. Deste modo podemos distinguir componentes de mesmo tipo nas regras, sem perder informações sobre a tipagem dos elementos.

Figura 21- Importando elemento tipo *Philosopher* do grafo tipo para lado esquerdo da regra



Fonte: Arquivo Pessoal

Figura 22- Elemento tipo *Philosopher* importado e com nome definido



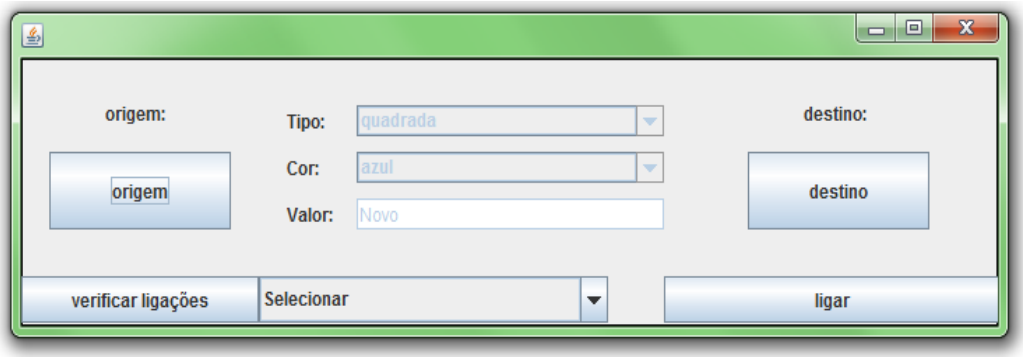
Fonte: Arquivo Pessoal

A importação de componentes não se resume a vértices e nem se limita a um elemento por vez. Podem ser selecionados vários vértices e arestas, resultando em uma importação múltiplas de elementos. Essa seleção múltipla é dada por cliques em elementos enquanto é mantida pressionada a tecla *control* ou por prototipação de um *rubber band selection box*. Esse processo de importação é ainda mais importante quando se trata de construção de regras de gramáticas de grafos orientados a objeto. Para que o tradutor do editor funcione corretamente deve-se respeitar o modo de construir as regras de gramáticas:

- Primeiro deve-se construir o lado esquerdo da regra. Essa construção deve ser feita por meio da importação de elementos do grafo tipo. As arestas podem ser construídas (como será mostrado logo abaixo) ou importadas do grafo tipo. Para importar uma aresta será preciso selecionar aresta, origem e destino. O lado esquerdo servirá como referência para os elementos preservados pela regra;
- Todos os elementos presentes no lado direito da regra, que o usuário deseje preservar, devem ser importados do lado esquerdo da regra, incluindo suas ligações. Deste modo os elementos que estão presentes no lado esquerdo e não estão presentes no lado direito serão considerados elementos eliminados;
- Os elementos que forem importados do grafo tipo para o lado direito ou as arestas construídas serão considerados elementos inseridos pela regra.

O usuário pode construir arestas a partir dos vértices importados do grafo tipo. Ao clicar na opção inserir nova ligação, será prototipada a janela que trata a criação de arestas, conforme mostra a Figura 23.

Figura 23- Janela de criação de arestas

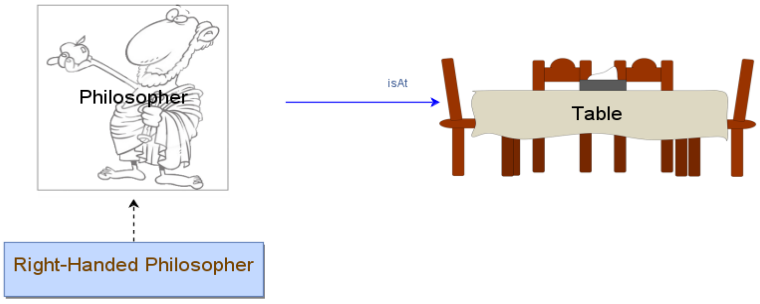


Fonte: Arquivo Pessoal

Note que agora os campos de inserção de informações estão desabilitados. Isto ocorre porque as arestas possíveis já estão definidas no grafo tipo. Observe ainda que na Figura 23 está disponível a opção verificar ligações, essa opção só pode ser selecionada em gramáticas de grafos orientados a objeto e grafos tipados durante construção de regras ou grafo inicial. A função de verificar ligação consulta quais ligações estão disponíveis entre a origem e destino selecionados pelo usuário.

Se existirem ligações entre estes elementos elas serão listadas ao lado do botão de verificação, restando ao usuário apenas selecionar a desejada e clicar em ligar. Nas gramáticas de grafos orientados a objeto a função de verificação deve, inclusive, verificar quais ligações que não estão no grafo tipo porém são possíveis graças a herança de classes. Essa verificação é realizada recursivamente e garante que se alguma das classes que são herdadas possuírem ligações com o destino ou origem selecionados, essas ligações estarão disponíveis na caixa de seleção de ligação. As Figuras 24 e 25 exemplificam este último tipo de ligação mencionado.

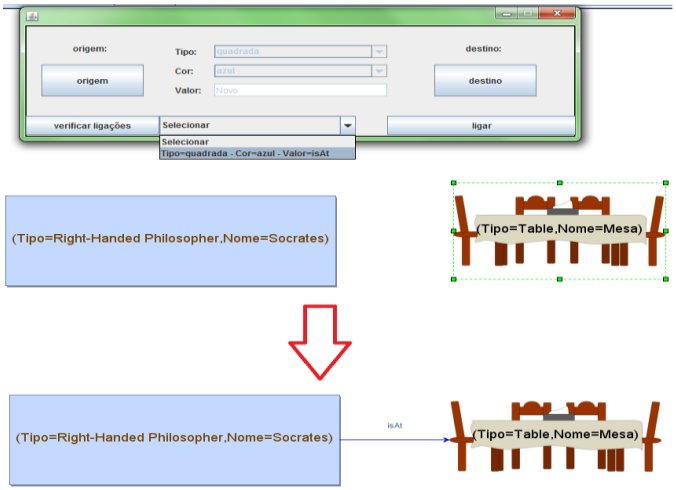
Figura 24- *Right-Handed Philosopher* herda *Philosopher*



Fonte: Arquivo Pessoal

A Figura 24 possui um exemplo de grafo tipo em que um nodo tipo *Right-Handed Philosopher* herda o tipo *Philosopher*. O tipo *Philosopher*, por sua vez, possui uma ligação azul de rótulo *isAt* com o tipo *Table*. Isso implica que ao selecionar *Right-Handed Philosopher* como origem e *Table* como destino, a função verificar ligação deve apresentar a ligação *isAt*, conforme mostra a Figura 25.

Figura 25- Ligação entre Socrates e Mesa é possível devido a herança do tipo *Philosopher*



Fonte: Arquivo Pessoal

4.6 Exportação para OOGXL

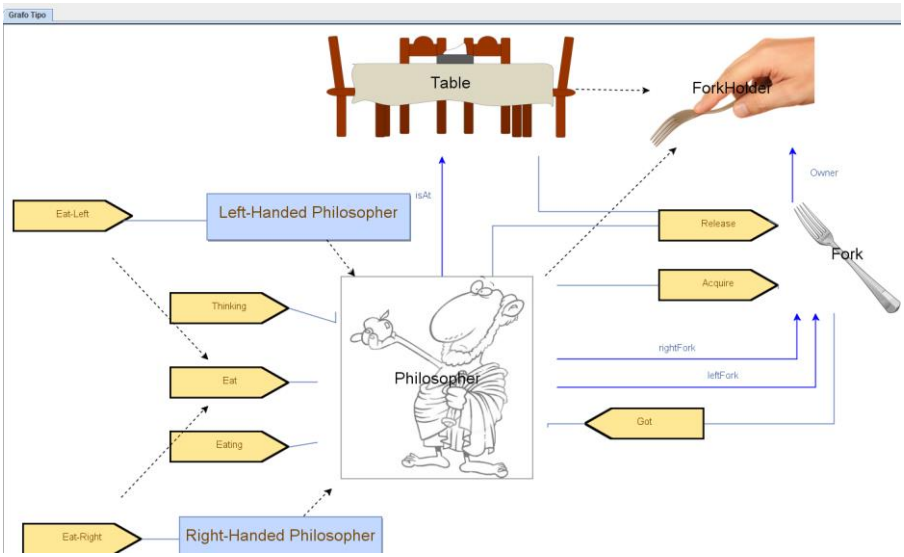
Apesar do editor desenvolvido neste trabalho possibilitar ao usuário três opções de construção de gramáticas, as gramáticas de grafos orientados a objeto são o foco principal. A partir deste tipo de gramática é possível realizar a tradução para a linguagem OOGXL. Conforme previsto nos objetivos deste trabalho, o editor apresenta um recurso para que as gramáticas de grafos orientados a objeto que possuam grafo tipo, grafo inicial e uma ou mais regras, possam ser traduzidas para OOGXL. Para exemplificar como ocorre o processo de tradução, será utilizada nas sessões subsequentes até o final do capítulo, uma gramática de grafos orientados a objeto que descreve o problema do *Jantar dos Filósofos* (TANENBAUM, 1992). Os métodos de estruturação dos componentes em OOGXL utilizados para realizar esta tradução são definidos em Moreira (2007).

A tradução de gramáticas é disparada na classe Main e é realizada na classe ExportarOOGXL, cujo parâmetro se resume ao nome do projeto que será exportado. Para que o método retorne corretamente a tradução, é necessário que o projeto informado possua grafo tipo, grafo inicial e pelo menos uma regra. O primeiro passo do método de tradução consiste em verificar se esses arquivos existem, se existirem, inicia-se a tradução, caso contrário, o usuário é informado sobre o que está faltando.

4.6.1 Grafo Tipo ou Grafo de Classes

O processo de tradução realizado na classe ExportarOOGXL é iniciado com a tradução do grafo tipo (ou grafo de classes). A Figura 26 apresenta um exemplo de grafo tipo, construído no editor, que faz parte de uma gramática de grafos orientados a objeto que descreve o problema do Jantar dos Filósofos. O primeiro passo para realizar a tradução do grafo tipo é carregar o conteúdo do arquivo Grafo_Tipo.xml e separar a informação em vértices e arestas. Essas informações obtidas são armazenadas em duas listas dinâmicas, uma contendo a lista de vértices e a outra a lista de arestas. Porém, isso não é o bastante para iniciar o processo de tradução, pois vértices comuns e mensagens apresentam diferenças significativas e, por isso, precisam ser armazenados separadamente. Então, um algoritmo seletor faz uma verificação para determinar quais dos elementos presentes na lista de vértices são do tipo mensagem e os armazena em uma terceira lista, a lista de mensagens.

Figura 26- Grafo tipo para o problema Jantar dos filósofos



Fonte: Arquivo Pessoal

Neste momento existem três listas que armazenam quais são as mensagens, nodos e arestas do grafo tipo. Já o próximo passo é remover informações desnecessárias para a tradução OOGXL, como os dados referentes ao estilo dos vértices (tamanho, cor, imagem e localização no componente) que não serão armazenados. Desta forma, somente informações úteis à tradução, como seus identificadores, rótulos, origem e destino de ligações, são mantidas. Com as listas contendo apenas informações relevantes é possível realizar a próxima etapa de tradução, que consiste em preparar vértices e mensagens para serem armazenados. Na lista de arestas existem ligações que descrevem a quem pertencem as mensagens e quais heranças estão presentes no grafo. Essas informações não serão armazenadas como arestas no arquivo final, mas sim nas próprias mensagens e vértices. Na Listagem de Código 1 estão sendo armazenados os nodos Philosopher e Fork, respectivamente.

Listagem de Código 1: Armazenando nodo *Philosopher*

```
0       <node id="Philosopher" parent="#ForkHolder">
1               <type xlink:href="#Philosopher">
2       </node>
3
4        <node id="Fork">
5               <type xlink:href="#Fork">
6       </node>
```

A estrutura básica para armazenar um nodo inicia-se com a tag *node* e seu respectivo parâmetro id (que podem ser observados na linha 0), e que deve receber um identificador único para a classe, que designa seu nome. Na linha 1 a tag *type* é utilizada para determinar o rótulo da classe através da parametrização de xlink:href. Em modelagem das classes do grafo de classes, o valor atribuído ao parâmetro id deve ser o mesmo valor atribuído ao parâmetro xlink:href precedido pelo símbolo #. A descrição das classes, que são representadas pelos nodos *Philosopher* e *Fork*, são semelhantes, contudo a classe *Philosopher* está relacionada com outra classe através da relação de herança. Podemos perceber então que na linha 0 existe um parâmetro adicional nomeado *parent* que não existe na linha 4. Esse parâmetro estabelece a relação entre a classe atual e a classe cujo nome é indicado pelo parâmetro de *parent*, que deve ser precedido de #. Isso significa que no exemplo da Listagem 1 a classe Philosopher estende a classe *ForkHolder*.

Definido o modo de como estruturar nodos, precisamos varrer a lista de arestas e identificar quais nodos herdam classes e quais não herdam. Isso se dá verificando quais

ids da lista de nodos estão presentes no campo *source* da lista de ligações pontilhadas e retornando o id presente no campo *target*. Nesse processo atualizamos as informações na lista de nodos, que agora serão reescritos com a estrutura correta e com o parâmetro parent quando for o caso, removendo então as ligações de heranças da lista de arestas.

Após a reestruturação das informações contidas na lista de vértices, restam ainda as ligações e as heranças das mensagens na lista de arestas. Portanto, devemos estruturar as mensagens do grafo para que possuam a estrutura padrão do OOGX e, ao mesmo tempo, remover as últimas ligações que não correspondem a atributos na lista de arestas. A Listagem de Código 2 apresenta como devem ser armazenadas as mensagens *Release* e *Eat-Left.*

Listagem de Código 2: Armazenando mensagens *Release* e *Eat-Left.*

```
0     <rel id="Release" isdirected="true">
1           <type xlink:href="root-class-model.gxl.xml#Message"/>
2           <relend target="Fork"
3                 startorder="1" direction=" out "/>
4           <relend target="Philosopher"
5                 startorder="2" direction=" in "/>
6           <relend target="Table"
7                 startorder="3" direction=" in "/>
8      </rel>
9
10    <rel id="Eat-Left" isdirected="true" override="Eat">
11          <type xlink:href="root-class-model.gxl.xml#Message"/>
12          <relend target="Left-Handed Philosopher"
13                startorder="1" direction=" out "/>
14    </rel>
```

A estrutura das mensagens inicia em *<rel>* e pelo seu respectivo parâmetro *id*, que pode ser observado nas linhas 0 e 10, e que deve conter um valor único. O segundo parâmetro *isdirected* vem por padrão contendo o valor verdadeiro, pois todas as mensagens são direcionadas. Assim como os nodos, as mensagens podem herdar outras mensagens e, por isso, na linha 10 o terceiro e último parâmetro *override* foi introduzido. Como a mensagem *Eat-Left* herda a mensagem *Eat,* o parâmetro *override* deve receber o valor *Eat*. Conforme mostra o grafo tipo da Figura 26, a mensagem *Eat-Left* está relacionada a classe *Left-Handed Philosopher*. A *tag <rel>* também é utilizada para representar atributos, como veremos em seguida, porém as relações de mensagens são efetivamente reconhecidas como mensagens através do valor /root-classmodel.gxl.xml#Message. Este valor é atribuído ao parâmetro xlink:href (linhas 1 e 11) de *type*, em que rootclass-

model.gxl.xml é o nome do arquivo onde os tipos de hiperarcos estão presentes, e *#Message* é o tipo do hiperarco que corresponde as mensagens. As relações entre classes e mensagens são descritas em <*relend*> em seus parâmetros *target*, *startorder* e *direction*. Onde *target* é a classe que a mensagem se relaciona, *startorder* define a ordem das ligações e *direction* informa se ligação parte da mensagem para uma classe (*out)* ou de uma classe para a mensagem(*in*).

Realizada a reestruturação das informações de classes e mensagens, resta na lista de arestas apenas os atributos que serão de fato armazenados. Na Listagem de código 3 temos como exemplo o atributo *isAt* formatado corretamente para OOGXL.

Listagem de Código 3: Armazenando atributo *isAt*

```
0   <rel id="isAt" isdirected="true">
1       <type xlink:href="
2           root-class-model.gxl.xml#Attribute"/>
3       <relend target="Philosopher"
4           startorder="1" direction="in"/>
5       <relend target="Table"
6           startorder="2" direction="out"/>
7   </rel>
```

Podemos perceber que atributos e mensagens possuem uma estrutura muito semelhante, sendo diferenciados apenas por *xlink:href* em *type*. Neste caso, o link aponta para o mesmo arquivo, root-class-model.gxl.xml, porém indica que o elemento em questão é um hiperarco *Attribute*. Assim como em mensagens a *tag* <*rel*> deve possuir um *id* único e, por padrão, todos os atributos são direcionados, portanto, o parâmetro *isderected* tem valor *true*. A *tag* <*relend*> é utilizada para expressar origem e destino do atributo, tendo como parâmetros *target*, *startorder* e *direction*. O parâmetro *target* informa qual elemento é origem quando *direction* possui valor *in* e *startorder* é 1, e qual elemento é destino quando *direction* possui valor *out* e *startorder* é 2.

Devidamente traduzidas as classes, mensagens e atributos, podemos utilizar a estrutura presente na Listagem de Código 4 para construir o arquivo que armazenará o grafo tipo traduzido para OOGXL.

Listagem de Código 4: Estrutura do grafo de classes da gramática de grafos orientados a objeto descrito através da linguagem OOGXL

```
0 <?xml version ="1.0" encoding="UTF-8"?>
```

```
1 <!DOCTYPE gxl SYSTEM "dtd/gxl-1.0 ">
2
3 <gxl xmlns:xlink="http://www.w3.org/1999/xlink">
4                   <graph id="jantar_filosofosClassModel">
5                   < nodos>
6                   < relações de atributo>
7                   < relações de mensagem>
8         </graph>
9</gxl>
```

Todo grafo de classes descrito através de OOGXL deve seguir a estrutura apresentada em Listagem de Código 4 (Moreira, 2007). A informação entre aspas da linha1: dtd/gxl-1.0.dtd consiste no arquivo DTD que define a sintaxe do código da linguagem GXL. Podendo consistir em uma referência para um arquivo local, ou em uma URL que aponta para um endereço remoto. Na linha 3 encontra-se a declaração do início do bloco de código OOGXL. Já nas linhas 4 e 8 são apresentados os delimitadores para um grafo, todas as estruturas do grafo precisam estar entre esses delimitadores. A tag *graph* da linha 4 possui um parâmetro obrigatório chamado id, que é setado com o nome do projeto concatenado com *ClassModel*. Os símbolos presentes nas linhas 5-7 são substituídos pelas listas de nodos, atributos e mensagens, respectivamente.

Esses dados são então inseridos no arquivo de saída de acordo com a estrutura de grafos de classes de gramáticas de grafos orientados a objeto descritos através da linguagem OOGXL. A Listagem de Código 5 contém o grafo tipo completamente traduzido para OOGXL.

Listagem de Código 5: Grafo tipo da gramática de grafos orientados a objeto que representa o problema jantar dos filósofos completamente traduzido para OOGXL

```
0 <?xml version ="1.0" encoding="UTF-8"?>
1 <!DOCTYPE gxl SYSTEM "dtd/gxl-1.0.dtd">
2
3 <gxl xmlns:xlink="http://www.w3.org/1999/xlink">
4          <graph id="jantar_filosofosClassModel">
5         <!-- NODES -->
6         <node id="Philosopher" parent="#ForkHolder">
7                   <type xlink:href="#Philosopher">
8         </node>
9
10        <node id="Table" parent="#ForkHolder">
11                  <type xlink:href="#Table">
12        </node>
13
14         <node id="Fork">
```

```
15              <type xlink:href="#Fork">
16      </node>
17
18      <node id="ForkHolder">
19              <type xlink:href="#ForkHolder">
20      </node>
21
22      <node id="Right-Handed Philosopher" parent="#Philosopher">
23              <type xlink:href="#Right-Handed Philosopher">
24      </node>
25
26      <node id="Left-Handed Philosopher" parent="#Philosopher">
27              <type xlink:href="#Left-Handed Philosopher">
28      </node>
29
30      <!-- ATTRIBUTES -->
31      <rel id="rightFork" isdirected="true">
32              <type xlink:href="
33                      root-class-model.gxl.xml#Attribute"/>
34              <relend target="Philosopher"
35                      startorder="1" direction="in"/>
36              <relend target="Fork"
37                      startorder="2" direction="out"/>
38      </rel>
39
40      <rel id="leftFork" isdirected="true">
41              <type xlink:href="
42                      root-class-model.gxl.xml#Attribute"/>
43              <relend target="Philosopher"
44                      startorder="1" direction="in"/>
45              <relend target="Fork"
46                      startorder="2" direction="out"/>
47      </rel>
48
49      <rel id="isAt" isdirected="true">
50              <type xlink:href="
51                      root-class-model.gxl.xml#Attribute"/>
52              <relend target="Philosopher"
53                      startorder="1" direction="in"/>
54              <relend target="Table"
55                      startorder="2" direction="out"/>
56      </rel>
57
58
59      <rel id="Owner" isdirected="true">
60              <type xlink:href="
61                      root-class-model.gxl.xml#Attribute"/>
62              <relend target="Fork"
63                      startorder="1" direction="in"/>
64              <relend target="ForkHolder"
```

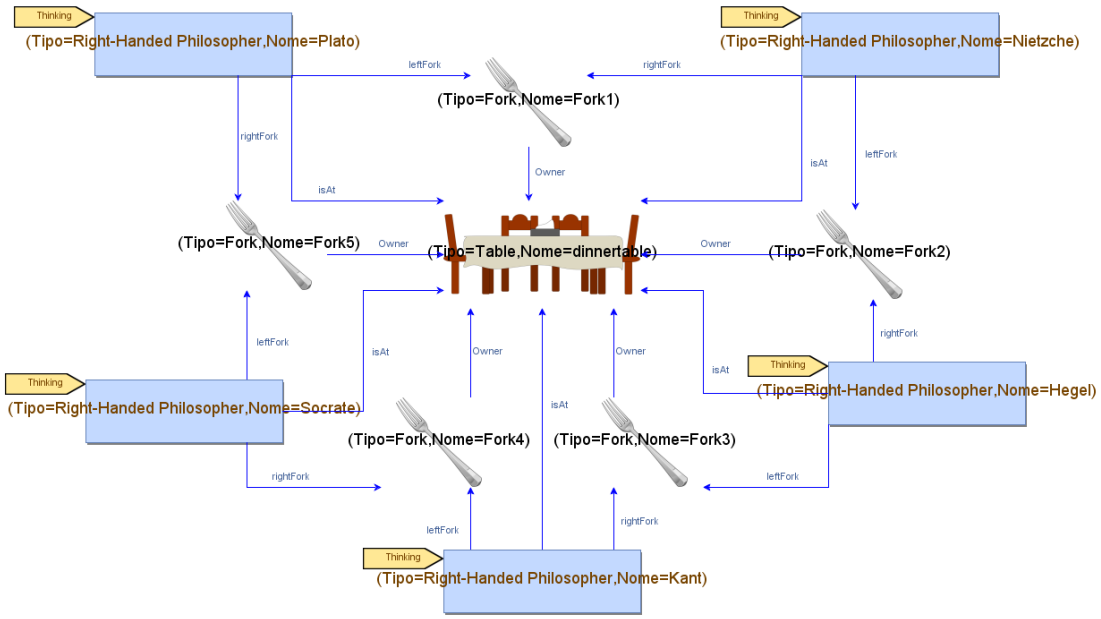
```
65                      startorder="2" direction="out"/>
66      </rel>
67
68      <!-- MESSAGES -->
69      <rel id="Release" isdirected="true">
70              <type xlink:href="root-class-model.gxl.xml#Message"/>
71              <relend target="Fork"
72                      startorder="1" direction=" out "/>
73              <relend target="Philosopher"
74                      startorder="2" direction=" in "/>
75              <relend target="Table"
76                      startorder="3" direction=" in "/>
77       </rel>
78
79      <rel id="Eat-Left" isdirected="true" override="Eat">
80              <type xlink:href="root-class-model.gxl.xml#Message"/>
81              <relend target="Left-Handed Philosopher"
82                      startorder="1" direction=" out "/>
83      </rel>
84
85      <rel id="Thinking" isdirected="true">
86              <type xlink:href="root-class-model.gxl.xml#Message"/>
87              <relend target="Philosopher"
88                      startorder="1" direction=" out "/>
89      </rel>
90
91      <rel id="Eating" isdirected="true">
92              <type xlink:href="root-class-model.gxl.xml#Message"/>
93              <relend target="Philosopher"
94                      startorder="1" direction=" out "/>
95      </rel>
96
97      <rel id="Eat" isdirected="true">
98              <type xlink:href="root-class-model.gxl.xml#Message"/>
99              <relend target="Philosopher"
100                     startorder="1" direction=" out "/>
101     </rel>
102
103     <rel id="Got" isdirected="true">
104             <type xlink:href="root-class-model.gxl.xml#Message"/>
105             <relend target="Philosopher"
106                     startorder="1" direction=" out "/>
107             <relend target="Fork"
108                     startorder="2" direction=" in "/>
110     </rel>
111
112      <rel id="Acquire" isdirected="true">
113             <type xlink:href="root-class-model.gxl.xml#Message"/>
114             <relend target="Fork"
115                     startorder="1" direction=" out "/>
```

```
116             <relend target="Philosopher"
117                     startorder="2" direction=" in "/>
118     </rel>
119
120     <rel id="Eat-Right" isdirected="true" override="Eat">
121             <type xlink:href="root-class-model.gxl.xml#Message"/>
122             <relend target="Right-Handed Philosopher"
123                     startorder="1" direction=" out "/>
124      </rel>
125
126
127 </graph>
128 </gxl>
```

4.6.2 Grafo Inicial

O segundo passo no processo de exportação de uma gramática de grafos orientados a objeto consiste em traduzir o grafo inicial. A Figura 27 apresenta o grafo inicial da gramática de grafos orientados a objeto que descreve o problema do jantar dos filósofos. Esse grafo será utilizado para exemplificar o processo de tradução de um grafo inicial construído no editor para a linguagem OOGXL.

Figura 27- Grafo inicial da gramática de grafos que descreve o problema Jantar dos Filósofos



Fonte: Arquivo Pessoal

A tradução de um grafo inicial construído no editor ocorre de modo semelhante à tradução de grafos tipos da sessão anterior. Ao carregar o arquivo XML que contém o grafo inicial, nomeado grafo_inicial.xml, todos os elementos devem ser separados como vértices ou arestas e armazenados na lista de arestas e vértices. Da mesma forma que no grafo tipo, ocorre uma varredura que determina quais tipos de vértices são mensagens e quais são nodos, separando-os em duas listas. Após essa varredura, existem três listas: lista de nodos, lista de arestas e lista de mensagens. Como não é preciso representar características de herança no grafo inicial, podemos começar a estruturar os nodos imediatamente, conforme os exemplos da Listagem de Código 6.

Listagem de Código 6: Construção OOGXL que descreve objetos do grafo inicial da gramática de grafos orientados a objeto.

```
0<node id ="Kant">
1       <type xlink:href="jantar_filosofosClassModel.xml#Right-Handed
Philosopher"/>
2</node>
3
4<node id ="Fork2">
5       <type xlink:href="jantar_filosofosClassModel.xml#Fork"/>
6</node>
```

Para armazenarmos um nodo de grafo inicial em formato OOGXL, utilizamos a mesma construção *node* presente na tradução do grafo tipo, com o parâmetro *id* recebendo o nome do objeto. A construção *type* (linhas 1 e 5), que no grafo de classes foi utilizada para rotular uma classe, aqui é utilizada para tipar um objeto. Isso ocorre através de uma referência para uma classe do grafo de classes descrita pelo parâmetro xlink:href. Este, por sua vez, consiste no nome do arquivo que descreve o grafo de classes, o separador #, e o nome da classe usada para tipar o objeto.

O próximo passo na tradução do grafo inicial consiste em reestruturar as relações de mensagens. Para isso é realizada uma varredura na lista de arestas, com o objetivo de relacionar mensagens com seus respectivos nodos. Esse conteúdo analisado da lista de arestas é removido conforme as mensagens são reestruturadas, restando somente as relações de atributos válidos na lista de arestas. A Listagem de Código 7 apresenta a estrutura necessária para representar a mensagem *thinking* do nodo tipo *Righ-Handed Philosopher* de nome *Kant*.

Listagem de Código 7: Construção OOGXL que descreve relações de mensagem do grafo inicial da gramática de grafos orientados a objeto.

```
0    <rel id ="Thinking_Kant" isdirected="true">
1        <type xlink:href="jantar_filosofosClassModel.xml#Thinking"/>
2        <relend target="Kant"
3                startorder="1" direction="out" />
4    </rel>
```

Para armazenarmos uma relação de mensagem utilizamos a mesma construção *rel* presente na tradução do grafo tipo, com o parâmetro *id* recebendo o nome do objeto. A construção *type*, mais uma vez, é utilizada para tipar o objeto através do parâmetro xlink:href. Por fim, as construções *relend* indicam as ligações da mensagem, tendo como parâmetros *target*, *startorder* e *direction*. Neste caso, *target* contém o nome do objeto que relaciona com a mensagem, *startorder* determina a ordem das ligações e *direction* indica se a ligação parte da mensagem para o objeto (*out)* ou do objeto para a mensagem (*in*).

A última etapa de tradução do grafo inicial consiste em realizar a tradução das relações de atributo. Na Listagem de Código 8 pode ser observado um exemplo desse tipo de relação.

Listagem de Código 8: Construção OOGXL que descreve relações de atributo do grafo inicial da gramática de grafos orientados a objeto.

```
0    <rel id="rightFork_Kant" isdirected="true">
1        <type xlink:href="
2            jantar_filosofosClassModel.xml#rightFork"/>
3        <relend target="Kant"
4            startorder="1" direction ="in"/>
5        <relend target ="Fork3"
6            startorder="2" direction="out"/>
7    </rel>
```

Todo grafo inicial descrito através de OOGXL deve seguir a estrutura apresentada em Listagem de Código 8 (Moreira, 2007). Na Listagem de Código 8 podemos observar que a tradução de uma relação de atributo de um grafo inicial ocorre exatamente do mesmo modo que na tradução de grafos tipos. A construção *rel* possui parâmetros *id*, que tem seu valor atribuído a partir do rótulo da ligação, e *isdirected*, que é sempre verdadeiro, pois todas as relações de atributo são direcionadas. A construção *type* define o tipo de ligação, enquanto as construções *relend* descrevem quais são os nodos de origem e destino, de

acordo com o parâmetro *direction*. Nas linhas 3 e 4 da Listagem de Código 8, observamos que a relação *rightFork* possui origem em *Kant*, pois *target* está setado com este valor e *direction* tem valor *in*, enquanto que nas linhas 5 e 6 o destino é definido *Fork*, já que o parâmetro *direction* tem valor *out*.

Com as listas de nodos, atributos e mensagens contendo todos os elementos traduzidos para OOGXL, podemos armazenar o grafo inicial traduzido com base na estrutura padrão apresentada pela Listagem de Código 9.

Listagem de Código 9: Estrutura do grafo inicial da gramática de grafos orientados a objeto descrito através da linguagem OOGXL.

```
0<?xml version ="1.0" encoding="UTF-8"?>
1 <!DOCTYPE gxl SYSTEM "gxl-1.0.dtd">
2
3<gxl xmlns:xlink="http://www.w3.org/1999/xlink">
4       <graph id="jantar_filosofosInstanceModel">
5               < objetos>
6               < relações de atributo>
7               < relações de mensagem>
8       </graph>
9 </gx>
```

Listagem de Código 10: Grafo inicial do problema jantar dos filósofos completamente traduzido para OOGXL.

```
0<?xml version ="1.0" encoding="UTF-8"?>
1 <!DOCTYPE gxl SYSTEM "gxl-1.0.dtd">
2
3<gxl xmlns:xlink="http://www.w3.org/1999/xlink">
4        <graph id="jantar_filosofosInstanceModel">
5        <!-- NODES -->
6       <node id ="dinnertable">
7               <type xlink:href="jantar_filosofosClassModel.xml#Table"/>
8       </node>
9
10      <node id ="Kant">
11              <type xlink:href="jantar_filosofosClassModel.xml#Right-
Handed Philosopher"/>
12      </node>
13
14      <node id ="Fork2">
15              <type xlink:href="jantar_filosofosClassModel.xml#Fork"/>
16      </node>
17
18      <node id ="Fork1">
19              <type xlink:href="jantar_filosofosClassModel.xml#Fork"/>
20      </node>
```

```
21
22      <node id ="Fork5">
23              <type xlink:href="jantar_filosofosClassModel.xml#Fork"/>
24      </node>
25
26      <node id ="Fork4">
27              <type xlink:href="jantar_filosofosClassModel.xml#Fork"/>
28      </node>
29
30      <node id ="Fork3">
31              <type xlink:href="jantar_filosofosClassModel.xml#Fork"/>
32      </node>
33
34      <node id ="Hegel">
35              <type xlink:href="jantar_filosofosClassModel.xml#Right-
Handed Philosopher"/>
36      </node>
37
38      <node id ="Nietzche">
39              <type xlink:href="jantar_filosofosClassModel.xml#Right-
Handed Philosopher"/>
40      </node>
41
42      <node id ="Plato">
43              <type xlink:href="jantar_filosofosClassModel.xml#Right-
Handed Philosopher"/>
44      </node>
45
46      <node id ="Socrate">
47              <type xlink:href="jantar_filosofosClassModel.xml#Right-
Handed Philosopher"/>
48      </node>
49
50        <!-- ATTRIBUTES -->
51      <rel id="Owner_Fork4" isdirected="true">
52              <type xlink:href="
53                      jantar_filosofosClassModel.xml#Owner"/>
54              <relend target="Fork4"
55                      startorder="1" direction ="in"/>
56              <relend target ="dinnertable"
57                      startorder="2" direction="out"/>
58      </rel>
59
60      <rel id="Owner_Fork3" isdirected="true">
61              <type xlink:href="
62                      jantar_filosofosClassModel.xml#Owner"/>
63              <relend target="Fork3"
64                      startorder="1" direction ="in"/>
65              <relend target ="dinnertable"
66                      startorder="2" direction="out"/>
```

```
67      </rel>
68
69      <rel id="Owner_Fork2" isdirected="true">
70              <type xlink:href="
71                      jantar_filosofosClassModel.xml#Owner"/>
72              <relend target="Fork2"
73                      startorder="1" direction ="in"/>
74              <relend target ="dinnertable"
75                      startorder="2" direction="out"/>
76      </rel>
77
78      <rel id="Owner_Fork1" isdirected="true">
79              <type xlink:href="
80                      jantar_filosofosClassModel.xml#Owner"/>
81              <relend target="Fork1"
82                      startorder="1" direction ="in"/>
83              <relend target ="dinnertable"
84                      startorder="2" direction="out"/>
85      </rel>
86
87      <rel id="Owner_Fork5" isdirected="true">
88              <type xlink:href="
89                      jantar_filosofosClassModel.xml#Owner"/>
90              <relend target="Fork5"
91                      startorder="1" direction ="in"/>
92              <relend target ="dinnertable"
93                      startorder="2" direction="out"/>
94      </rel>
95
96      <rel id="isAt_Plato" isdirected="true">
97              <type xlink:href="
98                      jantar_filosofosClassModel.xml#isAt"/>
99              <relend target="Plato"
100                     startorder="1" direction ="in"/>
101             <relend target ="dinnertable"
102                     startorder="2" direction="out"/>
103     </rel>
104
105     <rel id="isAt_Socrate" isdirected="true">
106             <type xlink:href="
107                     jantar_filosofosClassModel.xml#isAt"/>
108             <relend target="Socrate"
109                     startorder="1" direction ="in"/>
110             <relend target ="dinnertable"
111                     startorder="2" direction="out"/>
112     </rel>
113
114     <rel id="isAt_Kant" isdirected="true">
115             <type xlink:href="
116                     jantar_filosofosClassModel.xml#isAt"/>
```

```
117             <relend target="Kant"
118                     startorder="1" direction ="in"/>
119             <relend target ="dinnertable"
120                     startorder="2" direction="out"/>
121     </rel>
122
123     <rel id="isAt_Nietzche" isdirected="true">
124             <type xlink:href="
125                     jantar_filosofosClassModel.xml#isAt"/>
126             <relend target="Nietzche"
127                     startorder="1" direction ="in"/>
128             <relend target ="dinnertable"
129                     startorder="2" direction="out"/>
130     </rel>
131
132     <rel id="isAt_Hegel" isdirected="true">
133             <type xlink:href="
134                     jantar_filosofosClassModel.xml#isAt"/>
135             <relend target="Hegel"
136                     startorder="1" direction ="in"/>
137             <relend target ="dinnertable"
138                     startorder="2" direction="out"/>
139     </rel>
140
141     <rel id="leftFork_Kant" isdirected="true">
142             <type xlink:href="
143                     jantar_filosofosClassModel.xml#leftFork"/>
144             <relend target="Kant"
145                     startorder="1" direction ="in"/>
146             <relend target ="Fork4"
147                     startorder="2" direction="out"/>
148     </rel>
149
150     <rel id="rightFork_Kant" isdirected="true">
151             <type xlink:href="
152                     jantar_filosofosClassModel.xml#rightFork"/>
153             <relend target="Kant"
154                     startorder="1" direction ="in"/>
155             <relend target ="Fork3"
156                     startorder="2" direction="out"/>
157     </rel>
158
159     <rel id="leftFork_Socrate" isdirected="true">
160             <type xlink:href="
161                     jantar_filosofosClassModel.xml#leftFork"/>
162             <relend target="Socrate"
163                     startorder="1" direction ="in"/>
164             <relend target ="Fork5"
165                     startorder="2" direction="out"/>
166     </rel>
```

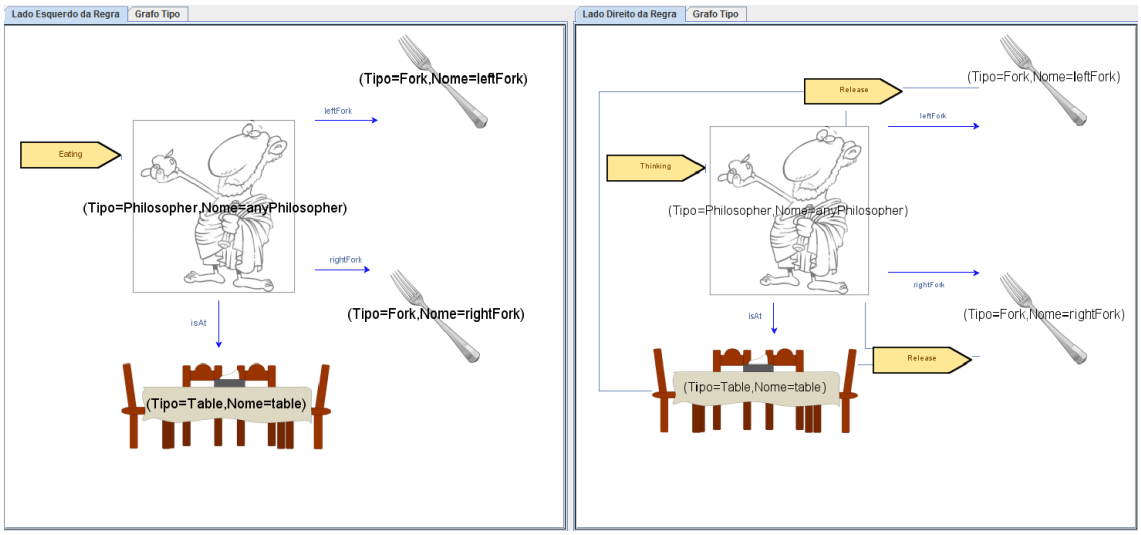
```
167
168     <rel id="rightFork_Socrate" isdirected="true">
169             <type xlink:href="
170                     jantar_filosofosClassModel.xml#rightFork"/>
171             <relend target="Socrate"
172                     startorder="1" direction ="in"/>
173             <relend target ="Fork4"
174             startorder="2" direction="out"/>
175     </rel>
176
177     <rel id="leftFork_Hegel" isdirected="true">
178             <type xlink:href="
179                     jantar_filosofosClassModel.xml#leftFork"/>
180             <relend target="Hegel"
181                     startorder="1" direction ="in"/>
182             <relend target ="Fork3"
183                     startorder="2" direction="out"/>
184     </rel>
185
186     <rel id="rightFork_Hegel" isdirected="true">
187             <type xlink:href="
188                     jantar_filosofosClassModel.xml#rightFork"/>
189             <relend target="Hegel"
190                     startorder="1" direction ="in"/>
191             <relend target ="Fork2"
192             startorder="2" direction="out"/>
193     </rel>
194
195     <rel id="rightFork_Plato" isdirected="true">
196             <type xlink:href="
197                     jantar_filosofosClassModel.xml#rightFork"/>
198             <relend target="Plato"
199                     startorder="1" direction ="in"/>
200             <relend target ="Fork5"
201                     startorder="2" direction="out"/>
202     </rel>
203
204     <rel id="leftFork_Plato" isdirected="true">
205             <type xlink:href="
206                     jantar_filosofosClassModel.xml#leftFork"/>
207             <relend target="Plato"
208                     startorder="1" direction ="in"/>
209             <relend target ="Fork1"
201             startorder="2" direction="out"/>
202     </rel>
203
204     <rel id="leftFork_Nietzche" isdirected="true">
205             <type xlink:href="
206                     jantar_filosofosClassModel.xml#leftFork"/>
207             <relend target="Nietzche"
```

```
208                     startorder="1" direction ="in"/>
209             <relend target ="Fork2"
210                     startorder="2" direction="out"/>
211     </rel>
212
213     <rel id="rightFork_Nietzche" isdirected="true">
214             <type xlink:href="
215                     jantar_filosofosClassModel.xml#rightFork"/>
216             <relend target="Nietzche"
217                     startorder="1" direction ="in"/>
218             <relend target ="Fork1"
219                     startorder="2" direction="out"/>
220     </rel>
221
222   <!-- MESSAGES -->
223     <rel id ="Thinking_Socrate" isdirected="true">
224             <type xlink:href="jantar_filosofosClassModel.xml#Thinking"/>
225             <relend target="Socrate"
226                     startorder="1" direction="out" />
227     </rel>
228
229     <rel id ="Thinking_Plato" isdirected="true">
230             <type xlink:href="jantar_filosofosClassModel.xml#Thinking"/>
231             <relend target="Plato"
232                     startorder="1" direction="out" />
233     </rel>
234
235     <rel id ="Thinking_Kant" isdirected="true">
236             <type xlink:href="jantar_filosofosClassModel.xml#Thinking"/>
237             <relend target="Kant"
238                             startorder="1" direction="out" />
239     </rel>
240
241     <rel id ="Thinking_Nietzche" isdirected="true">
242             <type xlink:href="jantar_filosofosClassModel.xml#Thinking"/>
243             <relend target="Nietzche"
244                     startorder="1" direction="out" />
245     </rel>
246
247     <rel id ="Thinking_Hegel" isdirected="true">
248             <type xlink:href="jantar_filosofosClassModel.xml#Thinking"/>
249             <relend target="Hegel"
250                     startorder="1" direction="out" />
251     </rel>
252
253     </graph>
254</gxl>
```

4.6.3 Regras da gramática

O terceiro e último passo no processo de exportação de uma gramática de grafos orientados a objeto consiste em traduzir o conjunto de regras construídas. Para realizar esse procedimento, o editor busca no diretório do projeto todos os arquivos tipo regra, faz uma lista destes arquivos e traduz uma a uma, armazenando essas regras descritas em OOGXL no arquivo de saída. A Figura 28 apresenta uma das regras da gramática de grafos orientados a objeto que descreve o problema do jantar dos filósofos. Esse grafo será utilizado para exemplificar o processo de tradução de uma regra construída no editor para a linguagem OOGXL.

Figura 28- Regra *Stop Eating*



Fonte: Arquivo Pessoal

Para armazenar a regra *stop_eating* em OOGXL devemos seguir a estrutura presente na Listagem de Código 11 (Moreira, 2007). Nesta estrutura a construção *rule* caracteriza o início de uma descrição de regra, e, o parâmetro *name*, por sua vez, nomeia a regra.

Listagem de Código 11: Estrutura OOGXL para regra *Stop Eating*.

```
0<rule name="stop_eating">
1       <preserved>
2               < Todos os elementos preservados pela regra>
3       </preserved>
4
5       <deleted>
```

```
6               <Todos os elementos eliminados pela regra>
7       </deleted>
8
9       <created>
10              <Todos os elementos inseridos pela regra>
11      </created>
```

Conforme visto no capítulo 2, as regras de gramáticas de grafos definem quais elementos serão preservados, eliminados e inseridos. Esses elementos devem ser estruturados no formato OOGXL e posicionados entre as construções *preserved*, *deleted* ou *created*.

Para determinar quais são os elementos preservados, eliminados e inseridos, ambos os arquivos XML que armazenam os lados da regra são acessados. Como foi realizado nas traduções das sessões anteriores, os elementos são divididos em nodos e arestas. De posse da lista de elementos do lado esquerdo e do lado direito da regra, é executado um procedimento que compara os *ids* dos elementos. Durante a construção de regras, todo elemento que é arrastado do lado esquerdo para o lado direito da regra tem seu *id* preservado, caracterizando assim um elemento preservado pela regra. Uma nova lista é criada, a de elementos preservados, e cada vez que um nodo ou aresta presente em ambos os lados da regra for encontrado pela comparação de ids, ele é armazenado nesta lista. Para verificar quais elementos foram eliminados, basta comparar o lado esquerdo da regra com a lista de elementos preservados, todos que não estiverem nesta lista de preservados devem ser inseridos na nova lista de elementos excluídos. O método que verifica os elementos inseridos ocorre de modo semelhante, onde é realizada uma comparação entre elementos do lado direito da regra com a lista de preservados, todos que não estiverem na lista de preservados devem ser inseridos na nova lista de elementos inseridos pela regra.

A Listagem de Código 12 apresenta dois dos elementos que serão preservados pela regra *stop_eating*, construída no editor. Todo o código OOGXL delimitado por <*preserved*> e <*preserved*> deve obrigatoriamente ser iniciado pela tag <*graph*> presente na linha 1 (Moreira, 2007). O parâmetro *id* recebe "*graph_"* concatenado com o nome da regra e é único até o final do arquivo de regras, uma vez que o editor não permite ao usuário criar duas regras de mesmo nome.

Listagem de Código 12: Estrutura OOGXL para elementos preservados pela regra *stop_eating.*

```
0<preserved>
```

```
1   <graph id="graph_stop_eating">
2       <node id="table_stop_eating">
3                   <type xlink:href="jantar_filosofos_ClassModelOOGXL.xml#Table"/>
4       </node>
5
6       <rel id="isAt_stop_eating" isdirected="true">
7               <type xlink:href="jantar_filosofos_ClassModelOOGXL.xml#isAt"/>
8               <relend target="anyPhilosopher_stop_eating"
9                               startorder="1" direction="in"/>
10              <relend target="table_stop_eating"
11                              startorder="2" direction="out"/>
12      </rel>
13
14  </graph>
15</preserved>
```

Cada construção *node* representa um nodo do grafo que será preservado pela regra e que será identificado pelo seu parâmetro *id*. O *id* de um nodo preservado é composto pelo nome do nodo concatenado com o nome da regra. Além disso, por meio da construção *type*, o tipo do nodo é especificado através do parâmetro *xlink:href*. Esse parâmetro é composto pelo caminho para o arquivo que contém a tipagem de elementos da regra e pela concatenação de # com o tipo do nodo.

Cada construção *rel* pode representar uma relação de atributo ou uma relação de mensagem. Nas linhas 6 a 12 da Listagem de Código 12 podemos observar como é estruturada uma relação de atributo. O parâmetro *id* contém um valor único, recebendo a concatenação do rótulo do atributo com o nome da regra. A construção *type*, mais uma vez, serve para especificar a tipagem da relação de atributo. Por fim, as construções *relend* determinam a origem e o destino, por meio dos parâmetros *target*, que contêm o nome do nodo concatenado com o nome da regra, e *direction*, que determina se é origem (*in*) ou destino (*out*).

A estrutura para representar nodos e arestas como elementos inseridos e eliminados não se altera. A Listagem de Código 13 apresenta a remoção da mensagem *eating* e a inserção da mensagem *release* (*rightFork*).

Listagem de Código 13: Estrutura OOGXL para alguns dos elementos eliminados e inseridos pela regra *stop_eating*.

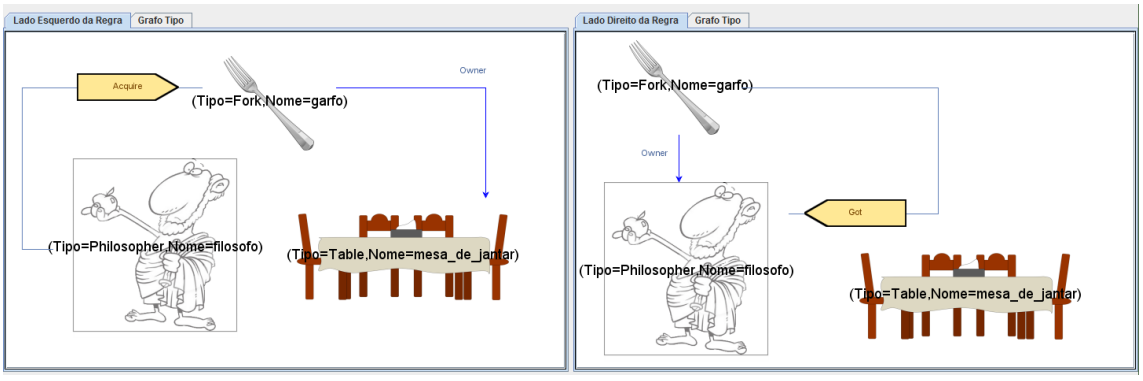
```
0 <deleted>
1       <rel id="anyPhilosopher_Eating_stop_eating" isdirected="true">
2               <type xlink:href="jantar_filosofos_ClassModelOOGXL.xml#Eating"/>
3               <relend target="anyPhilosopher_stop_eating"
```

```
4                                   startorder="1" direction="out"/>
5               </rel>
6 </deleted>
7 <created>
8       <rel id="rightFork_Release_stop_eating" isdirected="true">
9               <type xlink:href="jantar_filosofos_ClassModelOOGXL.xml#Release"/>
10              <relend target="rightFork_stop_eating"
11                      startorder="1" direction="out"/>
12              <relend target="anyPhilosopher_stop_eating"
13                      startorder="2" direction="in"/>
14              <relend target="table_stop_eating"
15                      startorder="3" direction="in"/>
16      </rel>
17</created>
```

Assim como a estrutura que representa mensagens em OOGXL pouco se altera em relação a que representa relações de atributo. A única diferença entre essas representações é dada no momento da concatenação do parâmetro *id*, que, neste caso é composto pelo rótulo do nodo destino com o rótulo da mensagem e o nome da regra, como pode ser observado nas linhas 1 e 8 da Listagem de Código 13. As mensagens se diferem de atributos através da informação de tipagem, que é dada pelo parâmetro *type*.

A Listagem de Código 14 apresenta a tradução completa da regra *stop_eating* e, para fins de exemplificação de como seria a estrutura para múltiplas regras, também da regra representada na Figura 29, nomeada *Acquire Fork*.

Figura 29- Regra *Acquire Fork*



Fonte: Arquivo Pessoal

Listagem de Código 14: Estrutura OOGXL que armazena duas regras da gramática jantar_filosofos.

```
0       <?xml version ="1.0" encoding="UTF-8"?>
1       <!DOCTYPE gxl SYSTEM "gtxl.dtd">
```

```
2
3       <gtxl>
4       <gts approach="spo">
5
6       <rule name="Acquire Fork">
7       <preserved>
8       <graph id="graph_Acquire Fork">
9       <node id="filosofo_Acquire Fork">
10                      <type
xlink:href="jantar_filosofos_ClassModelOOGXL.xml#Philosopher"/>
11              </node>
12      <node id="garfo_Acquire Fork">
13                      <type
xlink:href="jantar_filosofos_ClassModelOOGXL.xml#Fork"/>
14              </node>
15      <node id="mesa_de_jantar_Acquire Fork">
16                      <type
xlink:href="jantar_filosofos_ClassModelOOGXL.xml#Table"/>
17              </node>
18      </graph>
19      </preserved>
20      <deleted>
21      <rel id="garfo_Acquire_Acquire Fork" isdirected="true">
22              <type xlink:href="jantar_filosofos_ClassModelOOGXL.xml#Acquire"/>
23              <relend target="garfo_Acquire Fork"
24                      startorder="1" direction="out"/>
25              <relend target="filosofo_Acquire Fork"
26                      startorder="2" direction="in"/>
27      </rel>
28      <rel id="Owner_Acquire Fork" isdirected="true">
29                      <type
xlink:href="jantar_filosofos_ClassModelOOGXL.xml#Owner"/>
30                              <relend target="garfo_Acquire Fork"
31                                      startorder="1" direction="in"/>
32                              <relend target="mesa_de_jantar_Acquire Fork"
33                                      startorder="2" direction="out"/>
34              </rel>
35      </deleted>
36      <created>
37      <rel id="filosofo_Got_Acquire Fork" isdirected="true">
38              <type xlink:href="jantar_filosofos_ClassModelOOGXL.xml#Got"/>
39              <relend target="filosofo_Acquire Fork"
40                      startorder="1" direction="out"/>
41              <relend target="garfo_Acquire Fork"
42                      startorder="2" direction="in"/>
43      </rel>
44      <rel id="Owner_Acquire Fork" isdirected="true">
45                      <type
xlink:href="jantar_filosofos_ClassModelOOGXL.xml#Owner"/>
46                              <relend target="garfo_Acquire Fork"
```

```
47                                      startorder="1" direction="in"/>
48                              <relend target="filosofo_Acquire Fork"
49                                      startorder="2" direction="out"/>
50              </rel>
51      </created>
52      </rule>
53      <rule name="stop_eating">
54      <preserved>
55      <graph id="graph_stop_eating">
56      <node id="table_stop_eating">
57                      <type
xlink:href="jantar_filosofos_ClassModelOOGXL.xml#Table"/>
58              </node>
59      <node id="anyPhilosopher_stop_eating">
60                      <type
xlink:href="jantar_filosofos_ClassModelOOGXL.xml#Philosopher"/>
61              </node>
62      <node id="leftFork_stop_eating">
63                      <type
xlink:href="jantar_filosofos_ClassModelOOGXL.xml#Fork"/>
64              </node>
65      <node id="rightFork_stop_eating">
66                      <type
xlink:href="jantar_filosofos_ClassModelOOGXL.xml#Fork"/>
67              </node>
68      <rel id="isAt_stop_eating" isdirected="true">
69                      <type
xlink:href="jantar_filosofos_ClassModelOOGXL.xml#isAt"/>
70                              <relend target="anyPhilosopher_stop_eating"
71                                      startorder="1" direction="in"/>
72                              <relend target="table_stop_eating"
73                                      startorder="2" direction="out"/>
74              </rel>
75      <rel id="leftFork_stop_eating" isdirected="true">
76                      <type
xlink:href="jantar_filosofos_ClassModelOOGXL.xml#leftFork"/>
77                              <relend target="anyPhilosopher_stop_eating"
78                                      startorder="1" direction="in"/>
79                              <relend target="leftFork_stop_eating"
80                                      startorder="2" direction="out"/>
81              </rel>
82      <rel id="rightFork_stop_eating" isdirected="true">
83                      <type
xlink:href="jantar_filosofos_ClassModelOOGXL.xml#rightFork"/>
84                              <relend target="anyPhilosopher_stop_eating"
85                                      startorder="1" direction="in"/>
86                              <relend target="rightFork_stop_eating"
87                                      startorder="2" direction="out"/>
88              </rel>
89      </graph>
```

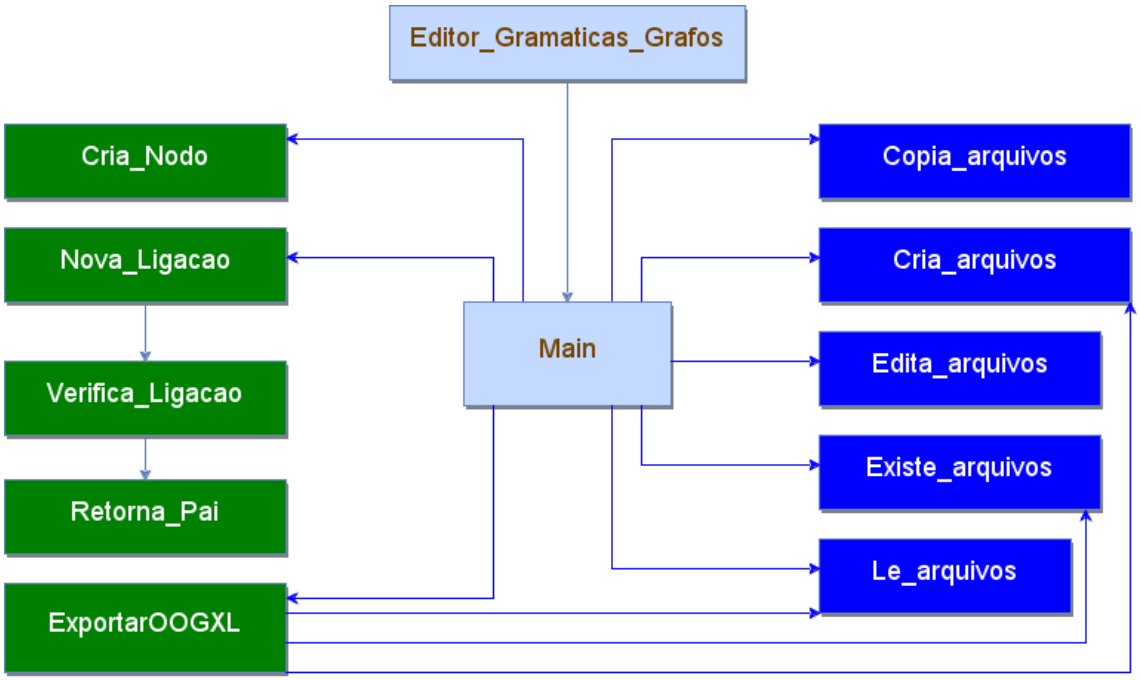
```
90      </preserved>
91      <deleted>
92      <rel id="anyPhilosopher_Eating_stop_eating" isdirected="true">
93              <type xlink:href="jantar_filosofos_ClassModelOOGXL.xml#Eating"/>
94              <relend target="anyPhilosopher_stop_eating"
95                      startorder="1" direction="out"/>
96      </rel>
97      </deleted>
98      <created>
99      <rel id="anyPhilosopher_Thinking_stop_eating" isdirected="true">
100             <type
xlink:href="jantar_filosofos_ClassModelOOGXL.xml#Thinking"/>
101             <relend target="anyPhilosopher_stop_eating"
102                     startorder="1" direction="out"/>
103     </rel>
104     <rel id="leftFork_Release_stop_eating" isdirected="true">
105             <type xlink:href="jantar_filosofos_ClassModelOOGXL.xml#Release"/>
106             <relend target="leftFork_stop_eating"
107                     startorder="1" direction="out"/>
108             <relend target="anyPhilosopher_stop_eating"
109                     startorder="2" direction="in"/>
110             <relend target="table_stop_eating"
111                     startorder="3" direction="in"/>
112     </rel>
113     <rel id="rightFork_Release_stop_eating" isdirected="true">
114             <type xlink:href="jantar_filosofos_ClassModelOOGXL.xml#Release"/>
115             <relend target="rightFork_stop_eating"
116                     startorder="1" direction="out"/>
117             <relend target="anyPhilosopher_stop_eating"
118                     startorder="2" direction="in"/>
119             <relend target="table_stop_eating"
120                     startorder="3" direction="in"/>
121     </rel>
122     </created>
123     </rule>
124     </gts>
125     </gxl>
```

Nas linhas 0 a 4 está o cabeçalho padrão para representar regras de gramáticas de grafos orientados a objeto em OOGXL (Moreira, 2007). As regras dentro do arquivo devem ser separadas por construções *<rule>* e *</rule>*, sendo diferenciadas pelo parâmetro *name*, que recebe o nome da regra descrita. As demais regras que compõem o problema do *Jantar dos Filósofos* podem ser encontradas em Moreira (2007).

4.7 Visão Geral do Editor

Uma visão geral da arquitetura do editor pode ser visualizada na Figura 30. A execução do editor se inicia na classe Editor_Gramaticas_Grafos, que tem como única função instanciar e iniciar a execução da classe Main. A classe Main é a principal classe do editor, nela está implementada toda interface gráfica do editor, com exceção da janela de inserir ligações, que foi implementada na classe Nova_Ligacao. Além disso, estão localizados em Main os principais métodos para desenho e edição de grafos.

Figura 30- Visão geral da arquitetura do editor



Fonte: Arquivo Pessoal

As classes a esquerda de *Main*, na Figura 30, são utilizadas para manipulação de componentes de gramáticas de grafos:

- A classe *Nova_Ligacao* coleta origem, destino e demais informações de uma nova aresta e as retorna a classe *Main*, que realiza a inserção. Quando se trata de uma gramática de grafos orientados a objeto ou de grafos tipados, utiliza as classes *Retorna_Pai* e *Verifica_Ligacao* para verificar quais arestas podem ser inseridas de acordo com o grafo tipo;
- A classe *Cria_Nodo* recebe como parâmetros o grafo e as características do novo nodo que deve ser inserido no grafo e realiza a inserção;

- A Classe *ExportarOOGXL* realiza a tradução de gramáticas de grafos orientados a objeto para OOGXL. O parâmetro desta classe se resume ao nome do projeto.

As classes a direita de *Main*, na Figura 30, são utilizadas para manipulação de arquivos:

- A classe *Copia_arquivos* é utilizada para copiar imagens, que serão utilizadas para tipar nodos em gramáticas de algum diretório do usuário para o diretório de imagens do editor;
- A classe *Cria_arquivos* é utilizada para criar diretórios, arquivos TXT e XML;
- A classe *Edita_arquivos* é utilizada para escrever informações nos diversos arquivos criados pelo editor;
- A classe *Le_arquivos* realiza a leitura de arquivos criados ou importados para o editor;
- A classe Existe_arquivos é utilizada para verificar se um determinado arquivo existe.

5. CONCLUSÃO

Inicialmente, foi realizado um estudo sobre os principais temas que foram abordados neste trabalho, como MDE, grafos, gramáticas de grafos e metodologias de desenvolvimento de software. Depois de realizado este estudo inicial, foram feitas diversas pesquisas com o objetivo de verificar quais ferramentas já estavam disponíveis e quais suas principais funcionalidades. Dentre as ferramentas encontradas, foram escolhidas as que possuíam maior similaridade com o trabalho em questão e então foi realizada a análise para verificar as funcionalidades das mesmas. Embora existissem algumas ferramentas com propósitos semelhantes, na maioria das vezes apenas arquivos binários estavam disponíveis e, por se tratarem de softwares antigos ou de código fechado, dificilmente funcionavam nas plataformas atuais. Com os resultados da análise, foram levantados os principais requisitos que o editor deveria possuir.

Já de posse dessa lista de requisitos, foi iniciado o processo de desenvolvimento da ferramenta por meio da IDE NetBeans e com a utilização da biblioteca JGraph. Logo após, um protótipo inicial foi construído para realizar o desenho de grafos. Esse protótipo inicial permitia a criação de vértices e arestas, além de permitir mudança de rotulações e exclusão de elementos. Depois de ser possível o desenho de grafos em formato de regras de gramáticas, isto é, contendo um lado esquerdo e direito, foram desenvolvidas funcionalidades que facilitam o trabalho de desenho de grafos. A partir da implementação dessas funcionalidades, como *zoom in* e *zoom out*, no componente pai dos grafos, e funções de desfazer e refazer, a ferramenta começou a parecer com um editor. O último passo de desenvolvimento do protótipo inicial consistiu na elaboração de um método que permitisse a inserção de componentes por meios gráficos, ou seja, o usuário pudesse inserir um novo tipo de nodo selecionando uma imagem disponível em seu computador em formatos como PNG, JPG e GIF.

Esse protótipo foi então aprimorado para possuir características comuns de editores modernos e amplamente utilizados. Primeiramente, foram desenvolvidas funções que permitiam salvar e restaurar um grafo a partir de um arquivo XML. Também foi desenvolvido um método que exportava o grafo desenhado como uma imagem PNG. Já o próximo passo foi inserir no editor um menu com opções conhecidas, que estão presentes nos editores de texto e imagens modernos. Este menu permite que até mesmo novos usuários saibam intuitivamente onde encontrar os recursos mais comuns, já que a

localização de recursos, os atalhos e a interface visual lhe são familiares. Como para salvar um arquivo, onde é provável que os usuários busquem pelo menu *ARQUIVO*, pois a grande maioria dos softwares atualmente mantém as opções *salvar* e *salvar como* neste local. Assim, continuando o aprimoramento do editor, foram desenvolvidos métodos específicos para os três tipos de gramáticas propostas nos objetivos. Como as gramáticas de grafos comuns não possuem restrições para inserção de componentes, esses métodos, em sua maioria, estavam relacionados a tratamentos especiais que as gramáticas de grafos tipados e as gramáticas de grafos orientados a objeto necessitam. Esses tratamentos especiais consistiam em permitir ao usuário criar um grafo tipo, que realiza a tipagem dos elementos da gramática e, portanto, impõe restrições durante a criação de um grafo inicial ou de regras da gramática. A solução encontrada foi permitir que regras e grafos iniciais tivessem componentes importados diretamente do grafo tipo, de modo que os nodos fossem literalmente arrastados de um grafo para outro. Porém, para a construção das regras de gramáticas de grafos orientados a objeto ainda era preciso que fosse desenvolvido um método que apresentasse as possíveis ligações entre nodos, levando em conta as características de herança presentes em OO. Deste modo, foi desenvolvido um método recursivo que retornava todos os pais de um determinado par de nodos (origem e destino) e então era verificado todos os tipos de ligações possíveis entre esses componentes. Se alguma ligação fosse encontrada, sendo ela diretamente entre os nodos ou devido a característica de herança, ela era inserida na lista de ligações possíveis, e podia, portanto, ser selecionada pelo usuário, garantindo assim que apenas ligações permitidas pelo grafo tipo fossem inseridas em regras e grafos iniciais. Além disso, as gramáticas de grafos orientados a objeto ainda apresentavam um tipo de nodo especial para a representação de mensagens. Por este motivo, foi inserida uma opção especial na janela de inserir ligações, que permite ao usuário inserir um tipo único que representa mensagens.

A parte final do trabalho se deu através da construção de um método que realiza a exportação das gramáticas de grafos orientados a objeto construídas no editor. Esta funcionalidade é a principal contribuição deste trabalho, visto que, durante o período de desenvolvimento deste trabalho, não existia nenhuma ferramenta que realizasse traduções para OOGXL, sendo assim, toda gramática deveria ser traduzida manualmente. A exportação consiste em traduzir as gramáticas geradas de modo que possam ser descritas em linguagem OOGXL. O primeiro passo foi realizar um estudo para verificar exatamente como fazer a tradução e, com isso, desenvolver um algoritmo que estruturasse

corretamente todo o tipo de elemento presente nas gramáticas no formato padrão de OOGXL. A partir desse estudo concluiu-se que o processo de tradução se inicia com o grafo tipo, pois o arquivo de saída, traduzido para OOGXL, serve como referência para os arquivos que representam o grafo inicial e o conjunto de regras que são traduzidos posteriormente nesta ordem e armazenados no diretório da gramática.

Por fim, para fins de validação do sistema, foi criada uma gramática de grafos orientados a objeto que representa o problema do *Jantar dos Filósofos* (TANENBAUM, 1992), utilizando todos os recursos propostos pelo editor de forma rápida e objetiva. Concluída a construção da gramática, foi realizada a exportação para OOGXl. Os arquivos de saída dessa gramática foram comparados com a tradução da mesma gramática, realizada em Moreira (2007). Assim, concluiu-se que os resultados obtidos foram satisfatórios, à medida que os arquivos de saída corresponderam à tradução da mesma gramática para OOGXL realizada em Moreira (2007), portanto, consideram-se atingidos os objetivos definidos para este trabalho.

5.1 Trabalhos Futuros

Apesar dos objetivos propostos para esse trabalho terem sido atingidos, existem dois pontos que podem ser aprimorados no editor, adicionando formatos novos de exportação e realizando uma integração de ferramentas.

O primeiro ponto está relacionado a métodos de exportação das gramáticas geradas. Uma possível extensão do editor seria desenvolver um método de exportação de gramáticas de grafos e de gramáticas de grafos tipados para linguagem GXL, ampliando assim a usabilidade do editor.

O segundo ponto seria uma integração entre este editor de gramáticas de grafos orientados a objeto e o tradutor de OOGXL para PROMELA, implementado em Moreira (2007). Dessa forma, modelos construídos com gramáticas de grafos orientados a objeto podem ser formalmente verificados e ter propriedades de sua execução (ausência de *deadlock* ou de postergação indefinida, por exemplo) garantidas.

REFERÊNCIAS

AHO, A.; SETHI, R.; LAM, M. **Compiladores – Princípios, Técnicas e Ferramentas.** Pearson Addison Wesley, 2008.

BARDOHL, R. A **Generic Graphical Editor for Visual Languages based on Algebraic Graph Grammars**, Proc. IEEE Symposium on Visual Languages (VL'98), Sept. 1998, Halifax, Canada.

FERREIRA, A. P. L. **Object-Oriented Graph Grammars**. Tese (Tese de Doutorado) — Universidade Federal do Rio Grande do Sul, Porto Alegre, RS, Setembro 2005.

FERREIRA, A. P. L.; RIBEIRO, L. **Jornadas de Atualização em Informática** Capítulo 8 - Programação Orientada a Objetos com Grafos, 2006.

FERREIRA, A. P. L.; RIBEIRO, L. **Towards object-oriented graphs and grammars**. In: IFIP TC6/WG6.1 INTERNATIONAL CONFERENCE ON FORMAL METHODS FOR OPEN OBJECT-BASED DISTRIBUTED SYSTEMS. [S.l.: s.n.], 2003.

HOPCROFT, J. E. **Formal languages and their relation to automata.** Reading: Addison-Wesley, 1969.

HUMPHREY, W. S. **A Discipline for Software Engineering**. Addison-Wesley Professional. First edition, 1995.

KENT, S. **Model Driven Engineering. In proceedings of IFM2002**, volume 2355 of LNCS. Springer-Verlag, 2002.

GUMS, F. **Verificador de propriedades em Gramática de Grafos**. Monografia. — Universidade Regional de Blumenau, Blumenau, SC, 2009.

LÖWE, M. **Extended Algebraic Graph Transformation**. PhD thesis — Technischen Universität Berlin, Berlin, Feb 1991.

RABUSKE, M. A. **Introdução à Teoria dos Grafos** Editora da UFSC, 1992.

MARTIN, J. C. **Introduction to Languages and the Theory of Computation**. 2nd. ed. [S.l.]: WCB/McGraw-Hill, 1996.

MOREIRA, C. E. S. **Automatização da Tradução de Especificações de Gramáticas de Grafos Orientados a Objetos para PROMELA**. Monografia. — Universidade do Vale do Rio dos Sinos, São Leopoldo, RS, Novembro 2007.

PRESSMAN, R. S. **Engenharia de software**. 6ª ed. Porto Alegre: Bookman, 2006.

SCHMIDT, D. C. **Guest Editor's Introduction: Model-Driven Engineering**. IEEE Computer, 39(2): fevereiro 2006.

SOMMERVILLE, I. **Engenharia de Software** 9ª. ed. São Paulo: Pearson Education do Brasil 2011.

TANENBAUM, A. S. **Modern Operating Systems**. Upper Saddle River, NJ: Simon & Schuster, 1992. 727p.