UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO – BACHARELADO

VERIFICADOR DE PROPRIEDADES EM GRAMÁTICA DE GRAFOS

FERNANDA GUMS

BLUMENAU
2009

2009/1-06

FERNANDA GUMS

VERIFICADOR DE PROPRIEDADES EM GRAMÁTICA DE GRAFOS

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciências da Computação — Bacharelado.

Profª. Joyce Martins, Mestre – Orientadora

BLUMENAU
2009

2009/1-06

VERIFICADOR DE PROPRIEDADES EM GRAMÁTICA DE GRAFOS

Por

FERNANDA GUMS

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

Presidente:	Profª. Joyce Martins, Mestre – Orientadora, FURB
Membro:	Prof. Dalton Solano dos Reis, Mestre – FURB
Membro:	Prof. José Roque Voltolini da Silva – FURB

Blumenau, 07 de julho de 2009

Aos que trilham pelos caminhos do conhecimento.

AGRADECIMENTOS

A Deus pela especificação e implementação do programa da vida.

Aos meus pais pela boa utilização do programa criado por Deus, instanciando um objeto do tipo Pessoa com atributos e métodos totalmente diferentes de todos os outros objetos residentes por aqui. Pelo cuidado, carinho, amor e acesso à educação.

Ao meu namorado Fernando, pelo carinho e paciência que teve nos momentos de desespero e agonia. Também pelo incentivo, motivação e pelos puxões de orelha pra que eu não me perdesse no caminho.

À Joyce, minha orientadora, pelo suporte, manutenção e pelo incentivo de atirar tiro no escuro, no que se refere ao tema, e por cima conseguir acertar o alvo.

Aos amigos, que fizeram parte da minha vida nestes últimos cinco anos, no processo da construção do conhecimento. Obrigada pelos encontros, risadas, aflições, discussões e principalmente por aquelas histórias sem pé nem cabeça que nos faziam fugir da realidade e nos matavam de rir.

Aos Professores, pelo incentivo a busca do conhecimento.

Ao Edward Murphy, que esteve presente desde o início do desenvolvimento do trabalho (apesar de ter tirado férias em alguns momentos), fazendo-me lembrar que "Se há mais de um modo de se fazer um trabalho, e um desses modos resultará em desastre, então alguém o fará".

Por fim, *Ein Prosit*!

Quem sabe concentrar-se numa coisa e insistir nela como único objetivo, obtém, ao fim e ao cabo, a capacidade de fazer qualquer coisa.

Mahatma Gandhi

RESUMO

Este trabalho apresenta a especificação e a implementação do Java *Verification Graph Grammar* (JVGG) para a criação de uma gramática de grafos de forma interativa e a verificação da alcançabilidade, aplicabilidade e conflito, que são propriedades de gramáticas de grafos. Para tornar a criação da gramática interativa, fez-se necessária a utilização da biblioteca Java *Graph* (JGraph) e da biblioteca Java *Graph Type* (JGraphT).

Palavras-chave: Gramática de grafos. Especificação formal. Verificação de propriedades.

ABSTRACT

This work presents the specification and implementation of Java Verification Graph Grammar (JVGG) to create a graph grammar of the form interactive and verification reachability, applicability and potential conflict, which are properties of graph grammar. To make the creation of interactive grammar, it was necessary to use the Java Graph library (JGraph) and Java Graph library Type (JGraphT).

Key-words: Graph grammars. Formal specification. Model checking.

LISTA DE ILUSTRAÇÕES

LISTA DE SIGLAS

AGG – *Attributed Graph Grammar*

DPO – *Double PushOut*

GPL – *General Public License*

GrGen – *Graph Rewrite GENerator*

GROOVE – *GRaphs for Object-Oriented Verification*

GXL - *Graph eXchange Language*

JGraph – Java *Graph*

JGraphT – Java *Graph Type*

JVGG – Java *Verification Graph Grammar*

RF – Requisito Funcional

RNF – Requisito Não Funcional

SPO – *Single PushOut*

UC – Casos de Uso

UML – *Unified Modeling Language*

XML – *eXtensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO

O desenvolvimento de sistemas, concorrentes ou não, tem sido uma tarefa árdua, tornando necessária a especificação de forma completa e livre de erros. Visando auxiliar este processo, existem métodos formais que são utilizados para construir modelos que descrevem o comportamento do sistema. Segundo Ribeiro (2000, p. 3), a especificação deve ser compacta, precisa e sem ambigüidade, ou seja, feita em uma linguagem com sintaxe e semântica precisamente definidas, utilizando conceitos matemáticos. Estes conceitos são importantes para especificar sistemas, pois permitem afirmar se "um sistema computacional apresenta uma certa propriedade ou satisfaz sua especificação" (DÉHARBE et al., 2000, p. 31). A especificação pode ser total ou parcial: é total quando descreve todo o comportamento desejado do sistema e é parcial quando descreve algumas propriedades que são necessárias para que o sistema possa operar. A especificação parcial é também conhecida como verificação de propriedades.

Como exemplo de método formal pode-se citar as gramáticas de grafos, que vêm sendo estudadas desde os anos 70 e têm sido aplicadas em campos da Ciência da Computação que requerem modelos de grafos dinâmicos (grafos que podem sofrer transformações ou manipulações em sua estrutura). Este formalismo consiste em um grafo inicial, que simboliza quais vértices e arestas estão presentes quando ele é iniciado, e um conjunto de regras, que representam os possíveis mapeamentos (morfismos) que podem modificar o estado do sistema. As gramáticas de grafos permitem a especificação de linguagens formais, reconhecimento de padrões, construção de compiladores, modelagem de sistemas concorrentes e distribuídos, projetos de banco de dados e modelagem de sistemas biológicos (RUSSO, 2003, p. 11).

Diante do exposto, propõe-se o desenvolvimento de uma ferramenta para especificação formal de sistemas utilizando gramáticas de grafos, onde será possível modelar de forma visual, o grafo inicial e o conjunto de regras que definem a gramática. Para que a modelagem seja feita de forma intuitiva, faz-se necessária a utilização da biblioteca gráfica Java *Graph* (JGraph). Após a definição da gramática, será possível verificar: a alcançabilidade, que indica quais vértices são alcançáveis ou não; a aplicabilidade, que identifica quais regras são aplicadas no sistema e quais nunca serão aplicadas; e o conflito, que encontra situações em que, concorrentemente, duas regras tentam modificar um objeto ao mesmo tempo.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é disponibilizar uma ferramenta para verificar propriedades em gramáticas de grafos.

São objetivos específicos do trabalho:

a) permitir a criação do grafo inicial (estado inicial) e do conjunto de regras da gramática;
b) verificar as propriedades alcançabilidade, aplicabilidade e conflito potencial.

1.2 ESTRUTURA DO TRABALHO

O presente trabalho está organizado em três capítulos intitulados respectivamente como: fundamentação teórica, desenvolvimento e conclusões.

O capítulo 2 apresenta os aspectos teóricos estudados para o desenvolvimento do trabalho. São abordados temas como grafos, gramáticas de grafos e propriedades existentes neste formalismo. Também são relacionados alguns trabalhos correlatos.

No capítulo 3 é descrito como foi realizado o desenvolvimento deste trabalho, detalhando os requisitos do protótipo, a especificação e a implementação. São apresentados os resultados encontrados com a finalização do trabalho.

Por fim, o capítulo 4 traz conclusões deste trabalho, bem como alguns aspectos que ficaram em aberto, servindo de sugestões para futuras extensões.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta inicialmente o conceito de grafos. Em seguida aborda conceitos do método de especificação formal conhecido como gramática de grafos e apresenta duas abordagens que são bastante difundidas com este formalismo. Na sequência são relacionadas três propriedades que podem ser verificadas em modelos elaborados utilizando gramáticas de grafos. Por fim, são apresentados trabalhos correlatos que auxiliaram no entendimento e no conhecimento de representações feitas utilizando gramática de grafos.

2.1 GRAFOS

Um grafo, de acordo com Szwarcfiter (1984, p. 35), é um conjunto finito não vazio de vértices e um conjunto de arestas, onde cada aresta é definida por um par não ordenado de vértices pertencentes ao conjunto de vértices do grafo (Figura 1(1)). É uma estrutura algébrica composta por pontos (vértices) e linhas (arestas), utilizada para explicar situações complexas de forma compacta, visual e clara. Uma outra representação conhecida na teoria dos grafos são os grafos dirigidos, também chamados de dígrafos (Figura 1(2)), que utilizam o conceito básico de grafos com a diferença que cada aresta é definida por um par ordenado de vértices, ou seja, a aresta possui origem e destino.

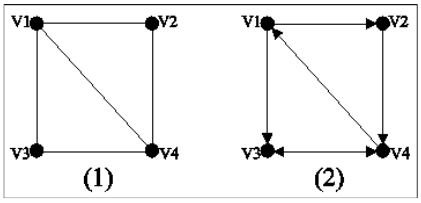


Figura 1 - Grafo e dígrafo

Existem duas vantagens em se especificar sistemas com grafos:

a) a possibilidade de obter uma semântica bem definida, por serem estruturas matemáticas;
b) a construção de uma representação visual, possibilitando uma especificação que melhore a compreensão por pessoas que desconheçam este recurso.

A utilização de grafos, simples ou dígrafos, pode dificultar o entendimento de modelos complexos. Para isso, Russo (2003, p. 12) sugere a utilização de conceitos que dêem mais

informações aos grafos, tais como:

a) hipergrafo, que é uma generalização de um grafo, onde as arestas podem ter vários destinos e origens. É definido formalmente por $H = (V, E)$, onde V é um conjunto de vértices do grafo e E é um conjunto de arestas que é composto por subconjuntos não vazios de V;
b) grafo com atributos, onde vértices e arestas recebem valores (pertencentes à álgebra), permitindo que sejam executadas operações sobre estes atributos;
c) grafo tipo, que é um grafo onde cada vértice e cada aresta representam um tipo distinto de vértice/aresta de uma especificação, ou seja, cada grafo que representa um estado do sistema só pode ter vértices e arestas do tipo especificado;
d) grafo rotulado, onde cada vértice e/ou cada aresta recebem um rótulo de um alfabeto Σ que contém os rótulos que podem ser utilizados.

2.2 GRAMÁTICA DE GRAFOS

Conforme Russo (2003, p. 9), gramática de grafos é um método de especificação formal onde os estados do sistema são representados por estruturas algébricas e o comportamento do mesmo é descrito pela mudança de estados. Uma gramática de grafos é composta por:

a) uma representação do conjunto de rótulos permitidos no sistema;
b) um grafo inicial, que representa o estado inicial do sistema;
c) um conjunto de regras, que define as possíveis mudanças que podem ocorrer no estado do sistema.

Dado um grafo inicial e um conjunto de regras, tem-se a noção da linguagem gerada, obtida pela aplicação das regras no grafo inicial. As regras são formadas por grafos e denotam a substituição de um grafo (lado esquerdo - *L*) por outro (lado direito - *R*). O lado esquerdo define o que deve estar presente em algum estado para que a regra seja aplicada, já o lado direito mostra o resultado da aplicação da regra.

Para especificar uma gramática de grafos, segundo Ribeiro (2000, p. 6), é necessário definir a sintaxe e a semântica da linguagem utilizada. Para que a sintaxe seja definida, é necessária a criação de estruturas baseadas em grafos (regras, grafo inicial) que são utilizadas para representar a especificação e para ditar como que as estruturas podem ser modificadas. Já

para a definição da semântica é necessário saber como as modificações podem acontecer a partir da aplicação das regras criadas.

Existem algumas abordagens de representação e transformação das gramáticas de grafos. Cada abordagem define uma forma distinta para especificar sintaxe e semântica. A seguir serão apresentadas duas destas abordagens.

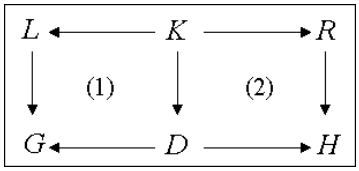
2.2.1 Abordagens de representação e transformação das gramáticas de grafos

Há muitas abordagens para representar as gramáticas de grafos e fazer suas transformações, sendo que uma delas é a abordagem algébrica (RIBEIRO, 2000, p. 6), que utiliza noções da teoria das categorias para definir, transformar e aplicar as regras. Ela foi criada na *Technical University of Berlin* e recebe este nome porque grafos são álgebras. As transformações são dadas por construções algébricas, conhecidas como *pushout*, colagem ou soma amalgamada, que permitem aplicar conceitos de álgebra e teoria das categorias, fornecendo provas complexas. Esta abordagem descreve gramáticas de grafos usando morfismos e colagens, ou seja, permite que um sub-grafo possa ser substituído por outro. A abordagem algébrica é dividida em algumas sub-abordagens, sendo que as mais conhecidas são a *Double-PushOut* (DPO) e a *Single-PushOut* (SPO).

A DPO foi a primeira abordagem algébrica desenvolvida. Seu nome vem da operação algébrica usada para definir o passo de derivação, que consiste na reescrita de grafos por duas construções de colagens na categoria dos grafos e morfismos totais (CORRADINI et al., 1997, p. 171). As regras na abordagem DPO são definidas por um par de morfismos $L \xleftarrow{l} K \xrightarrow{r} R$, onde *L* é o lado esquerdo da regra; *R* o lado direito; *K* é um grafo de interface[1] e *l* e *r* são os morfismos totais. A Figura 2 apresenta como são realizados os passos de derivação na abordagem DPO. A derivação acontece quando se tem uma regra que se deseja aplicar em um grafo G. Para que a regra seja aplicada é necessário que exista uma ocorrência de *L* em *G* (que é o grafo do estado atual). Caso a condição anterior tenha sido satisfeita, o próximo passo é excluir do grafo *G* todos os itens de *L* que não fazem parte da interface *K*, gerando o grafo *D* (Figura 2 (1)). Finalmente, faz-se a colagem de *R* com *D*, criando os itens de *R* que não estão em *K*, gerando o grafo *H* (Figura 2 (2)), que é o resultado

[1] Um grafo de interface contém os itens (vértices/arestas) que pertencem a *L* e que são preservados em *R*.

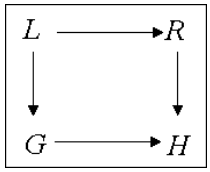
do passo de derivação da regra.



Fonte: adaptado de Corradini (1997, p. 171).

Figura 2 – Passo de derivação da abordagem DPO

Ao contrário da DPO, para SPO o passo de derivação é definido em uma única operação de colagem na categoria dos grafos e morfismos parciais de grafos (EHRIG et al., 1997, p. 249). Uma regra $L \xrightarrow{p} R$ consiste de dois grafos, onde *L* é o lado esquerdo da regra; *R* o lado direito e *p* é o morfismo parcial entre eles. De acordo com a Figura 3, o passo de derivação acontece quando há um grafo *G* em que se deseja aplicar uma regra. O lado *L* da regra define o estado que *G* precisa estar para que a regra seja aplicada e *R* representa o que será alterado caso aconteça o passo de derivação e *H* seja gerado.



Fonte: adaptado de Corradini (1997, p. 171).

Figura 3 – Passo de derivação da abordagem SPO

O que diferencia as abordagens DPO e SPO é a forma que cada uma das abordagens se comporta para resolver situações problemáticas. A DPO não permite, em alguns casos, a derivação do grafo, já na SPO derivações sempre são permitidas e a exclusão tem prioridade sobre a preservação de vértices. De acordo com Russo (2003, p. 21), a abordagem DPO pode ser traduzida de forma viável para a SPO.

Os conceitos relacionados à sintaxe e à semântica apresentados na sequência estão baseados na abordagem algébrica SPO.

2.2.2 Sintaxe

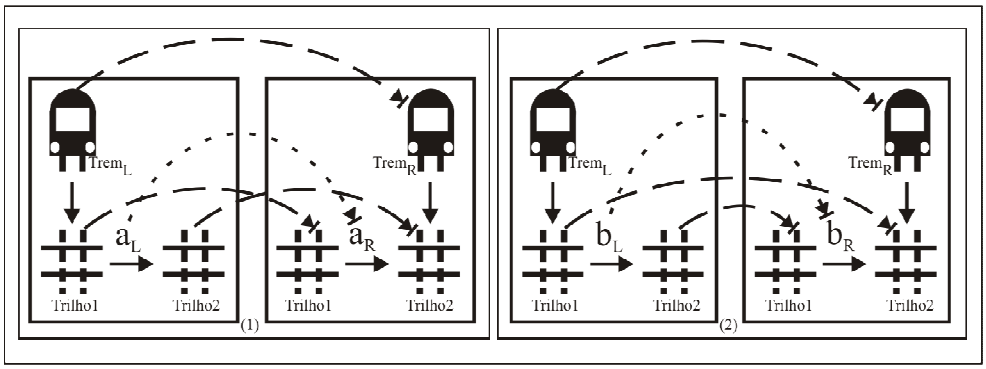
Para definir a sintaxe na gramática de grafos é necessária a construção das regras que especificam as mudanças de estado que podem ocorrer a partir do grafo inicial. As regras são compostas pelo grafo *L*, pelo grafo *R* e por *p*, morfismo parcial de grafos, que é responsável pela compatibilidade do mapeamento de vértices e arestas de *L*, em vértices e arestas em *R*.

De acordo com Dotti, Pasini e Santos (2004, p. 88), uma regra denotada por $r: L \rightarrow R$

especifica a mudança de estado do sistema, que ocorre da seguinte forma:

a) itens em *L* devem estar presentes no estado atual para que a regra possa ser aplicada;
b) itens em *L* que não têm uma imagem em *R* são removidos;
c) itens em *L* que são mapeados para *R* são preservados;
d) itens em *R* que não têm uma pré-imagem em *L* são criados.

As regras podem ser consideradas consistentes quando suas relações ou mapeamentos forem expressos através de morfismos, que podem ser totais ou parciais. O morfismo total é a coleção de todos os grafos que preservam a sua estrutura[2] (vértices e arestas). 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 a_L contida no grafo *L* for mapeada para uma aresta a_R do grafo *R*, então o vértice origem e o vértice destino de a_L devem ser os mesmos em a_R. A Figura 4 apresenta um exemplo onde um trem pode trocar de trilho. Este exemplo será utilizado para explicar o morfismo e o não-morfismo de grafos. A Figura 4(1) mostra que o $Trem_L$ está mapeado para o $Trem_R$, porém em outro vértice. O vértice $Trilho_{1L}$ está mapeado para o $Trilho_{1R}$, assim como o $Trilho_{2L}$ para o $Trilho_{2R}$ e a aresta a_L para a aresta a_R, o que o torna um exemplo de morfismo, onde todos os vértices e arestas estão compatíveis. Na Figura 4(2), o $Trem_L$ está mapeado para o $Trem_R$, porém visualmente em outro vértice. Observando o mapeamento dos trilhos, são encontrados problemas, tais como: o $Trilho_{1L}$ estar mapeado para o $Trilho_{2R}$ e o $Trilho_{2L}$ estar mapeado para o $Trilho_{1R}$. É como se o trem não tivesse em nenhum momento saído do lugar, apenas tendo sido trocado de posição visualmente.



Fonte: adaptado de Dehárbe (2000, p. 19).

Figura 4 – Morfismo e não-morfismo de grafos

[2] A estrutura de um grafo é preservada quando "para qualquer aresta mapeada [...], se verificarmos qual é seu nodo de origem no grafo de origem do mapeamento e depois de mapearmos este nodo, temos que chegar exatamente no mesmo lugar que chegaríamos se primeiro mapeássemos a aresta e [...] verificássemos qual é o nodo de origem da aresta mapeada no grafo destino." (FERREIRA; RIBEIRO, 2006, p. 403)

2.2.3 Semântica

A semântica de uma gramática de grafos indica seu comportamento. Este comportamento é observado a partir da aplicação de regras nos grafos que representam seu estado. Desta forma, a semântica é dada pelo conjunto de grafos deriváveis a partir do grafo inicial. A aplicação de regras ou transição de estado é modelada por um morfismo total de grafos, o que significa que só é possível aplicar a regra se no estado atual do sistema existir um mapeamento para a regra.

Para explicar a aplicação de regras e outros conceitos, é apresentado um exemplo de especificação completa utilizando gramática de grafos. O exemplo, adaptado de Déharbe et al. (2000, p. 16), descreve o funcionamento de uma ferrovia bem simples. A ferrovia possui três tipos de componentes (Figura 5): trens que se comportam independentemente, trilhos e desvios com duas posições.

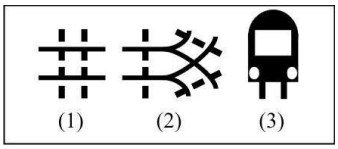


Figura 5 – Rótulos da especificação da ferrovia

A ferrovia deve se comportar da seguinte forma:

a) cada trem deve ocupar uma posição diferente na malha ferroviária;
b) os trens se movimentam alternadamente pela malha ferroviária;
c) os trens definem para qual posição devem ir;
d) quando o trem estiver em um desvio e o desvio estiver conectado ao caminho que o trem deve seguir, ele segue, senão aguarda até que o desvio se altere para o outro caminho.

Para especificar a ferrovia utilizando gramática de grafos, primeiramente é necessário definir o conjunto de rótulos Σ:

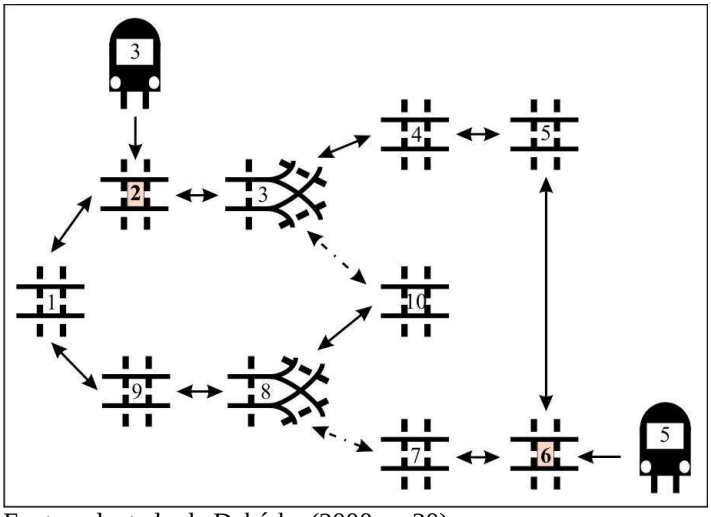
a) o símbolo da Figura 5(1) representa um trilho da ferrovia. Quando este símbolo aparece no grafo inicial ou em regras, ele possui um identificador que é um número no interior do trilho. Se este número for com o fundo transparente significa que o trecho está livre, se ele tiver alguma coloração, significa que está ocupado por um trem;
b) os desvios da ferrovia são representados conforme mostra a Figura 5(2). No grafo inicial ou em regras este símbolo quando aparece com arestas tracejadas indicam

que o desvio não está conectado, já as arestas cheias indicam que o caminho está conectado ao desvio;

c) os trens são representados pela Figura 5(3). Este símbolo no grafo inicial ou em regras aparece com uma aresta indicando a sua posição na ferrovia e o número dentro da cabine do trem indica qual é o próximo trecho que ele deve ir.

As arestas que ligam os trilhos representam o sentido nos quais os trens podem se movimentar. Os números que fazem parte da especificação são identificadores do conjunto de rótulos.

Um estado inicial possível para uma ferrovia está representado na Figura 6, onde existem oito trilhos, dois desvios e dois trens. Pode-se notar que o `trilho2` não está com sua coloração transparente, o que indica que está ocupado por um `trem`, que deve ir para o `desvio3` na sequência. Da mesma forma o `trilho6`, está ocupado por um `trem` que deve ir para o `trilho5`.



Fonte: adaptado de Dehárbe (2000, p. 20).

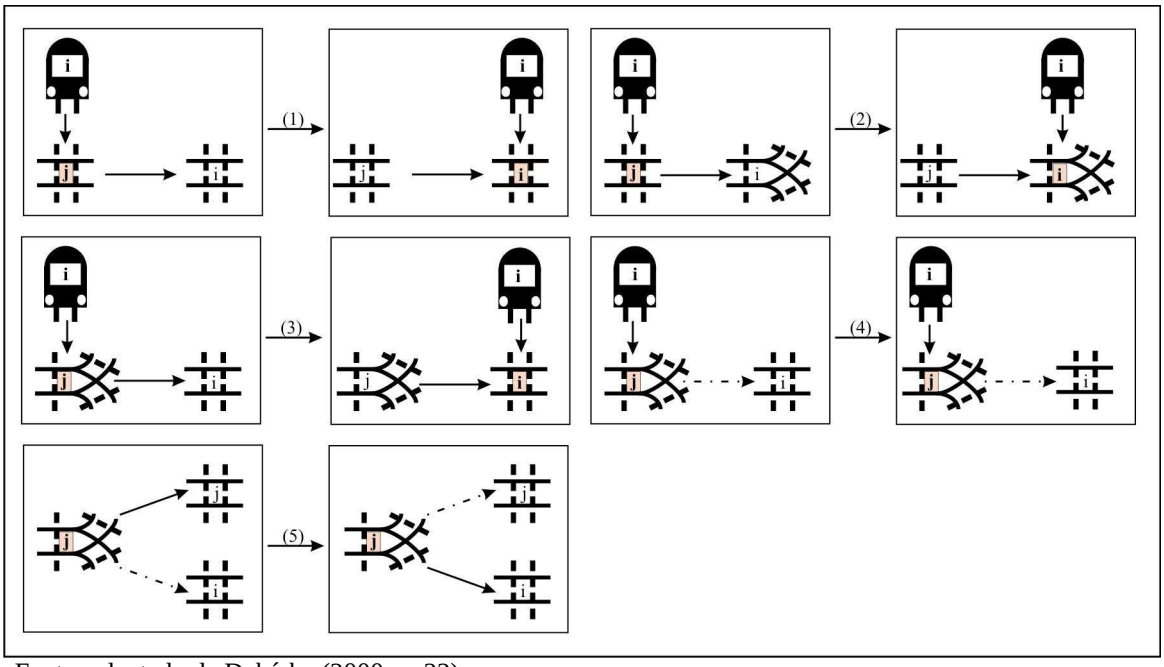
Figura 6 – Estado inicial da ferrovia

O funcionamento da ferrovia é modelado através das regras para funcionamento do `trem`, descritas no Quadro 1 e ilustradas na Figura 7. O `trem` possui quatro regras: `move`, `entraDesvio`, `saiDesvio` e `aguardaDesvio`. O trilho é controlado pela regra `mudaDesvio`.

REGRA	CONDIÇÃO	COMPORTAMENTO
`move` Figura 7 (1)	O `trem` está em um `trilhoj` que tem como vizinho o `trilhoi` livre e o trem quer se movimentar para o `trilhoi`.	O movimento pode ocorrer, ficando o `trilhoi` com fundo colorido, o `trilhoj` transparente e o `trem` no `trilhoi`.
`entraDesvio` Figura 7 (2)	O `trem` está em um `trilhoj` que tem como vizinho um `desvioi` livre e o `trem` quer se movimentar para o `desvioi`.	Análogo a regra `move`.
`saiDesvio` Figura 7 (3)	O `trem` está em um `devioj` que está conectado ao `trilhoi`. Este trilho está livre e o `trem` quer se movimentar para o `trilhoi`.	Análogo a regra `move`.
`aguardaDesvio` Figura 7 (4)	O `trem` está em um `desvioj` que não está conectado ao `trilhoi`, para o qual o `trem` quer se movimentar.	O `trem` permanece na mesma posição e aguarda até que o `desvioj` seja alterado.
`mudaDesvio` Figura 7 (5)	O `desvioj` está conectado ao `trilhoj` e o `desvioj` deve se conectar ao `trilhoi`.	O `desvioj` desconecta-se do `trilhoj` e conecta-se ao `trilhoi`.

Fonte: adaptado de Dehárbe (2000, p. 21).

Quadro 1 – Regras do trem



Fonte: adaptado de Dehárbe (2000, p. 22).

Figura 7 – Regras para o funcionamento do `trem`

Um exemplo de aplicação da regra `saiDesvio` pode ser visto na Figura 8. Seja G o grafo que define o estado atual do sistema. A aplicação da regra `saiDesvio` pode ocorrer quando existir um mapeamento de *L* em G, permitindo que o morfismo seja realizado, gerando um novo grafo (*H*) baseado em *R*. Pode-se observar que o `trem` que está sobre o `desvio3` torna aplicável a regra `saiDesvio`, pois existe um trem em um desvio e este desvio está conectado a um trilho da malha ferroviária, satisfazendo a condição do lado *L* da regra. Sendo assim, os itens (vértices/arestas) que estão em *L* e não estão em *R* são removidos. Os

itens de *R* que não estão em *L* são criados, gerando um novo grafo conhecido como *H*. Neste grafo *H* o `trem` não está mais no `desvio3`, mas sim no `trilho4`, finalizando a aplicação da regra.

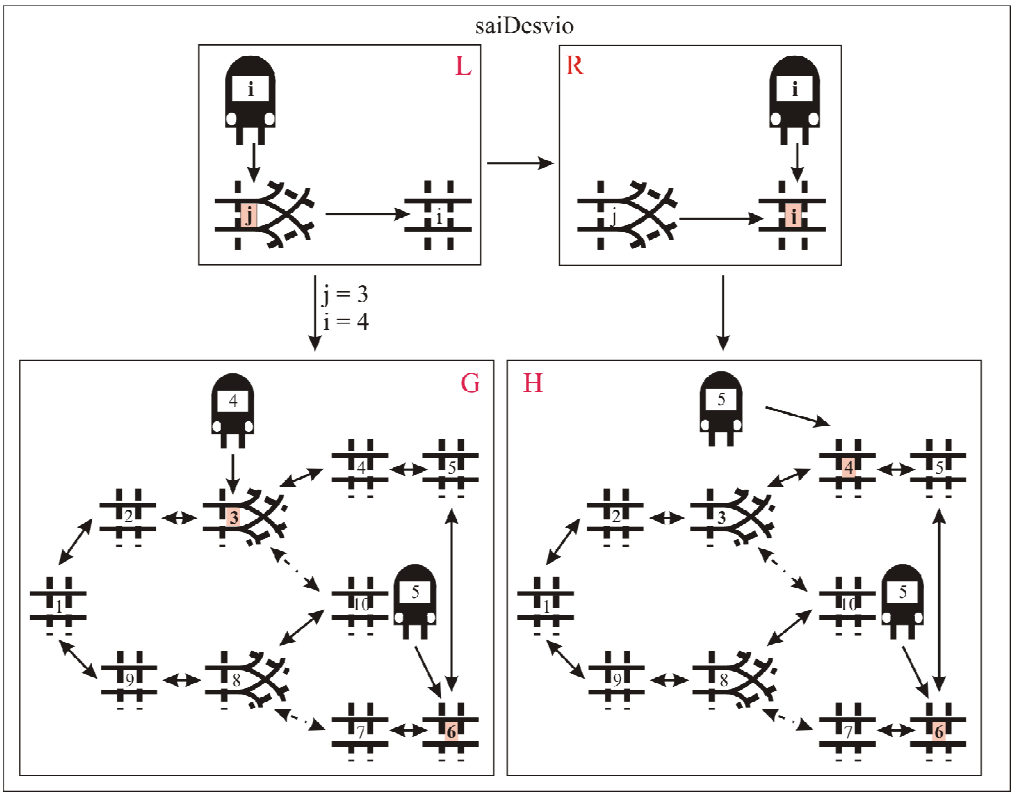
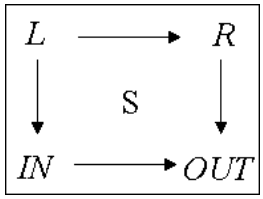


Figura 8 – Aplicação da regra `saiDesvio`

Quando acontece a aplicação das regras, o grafo resultante desta aplicação serve de entrada para a aplicação da regra seguinte, sendo essa ocorrência chamada de passo de derivação (*S*). Conforme Ribeiro (2000, p. 9), o resultado da aplicação de uma regra *r: L* ⟶ *R* em um grafo é obtido através dos seguintes passos, ilustrados na Figura 9:

a) adicionar ao grafo (*IN*) tudo que foi criado pela regra (itens que fazem parte de *R*, mas não de *L*);
b) remover do grafo resultante do passo acima tudo que deve ser eliminado pela regra (itens que fazem parte de *L* e não de *R*);
c) remover as arestas pendentes, caso haja arestas conectadas a vértices eliminados, mesmo que isso não esteja definido na regra (grafo *OUT*).



Fonte: adaptado de Ehrig et al. (1997, p. 256).

Figura 9 – Passo de derivação correspondente a um *pushout*

As regras podem ser aplicadas sequencialmente ou paralelamente. A derivação sequencial é uma sequência de passos de derivação que podem ser obtidos a partir do grafo inicial e utilizando as regras da gramática. Pode-se observar na Figura 10 que o grafo de saída de um passo serve como entrada para o próximo passo. Ao aplicar a regra `entraDesvio` no grafo inicial *G1*(*IN*), outro grafo *G2*(*OUT*) é gerado. Este mesmo grafo *G2*(*OUT*) serve como entrada *G2*(*IN*) para a aplicação da regra `saiDesvio`, gerando um novo grafo de saída *G3*(*OUT*) e assim sucessivamente até não existirem mais regras que possam ser aplicadas e a execução é encerrada.

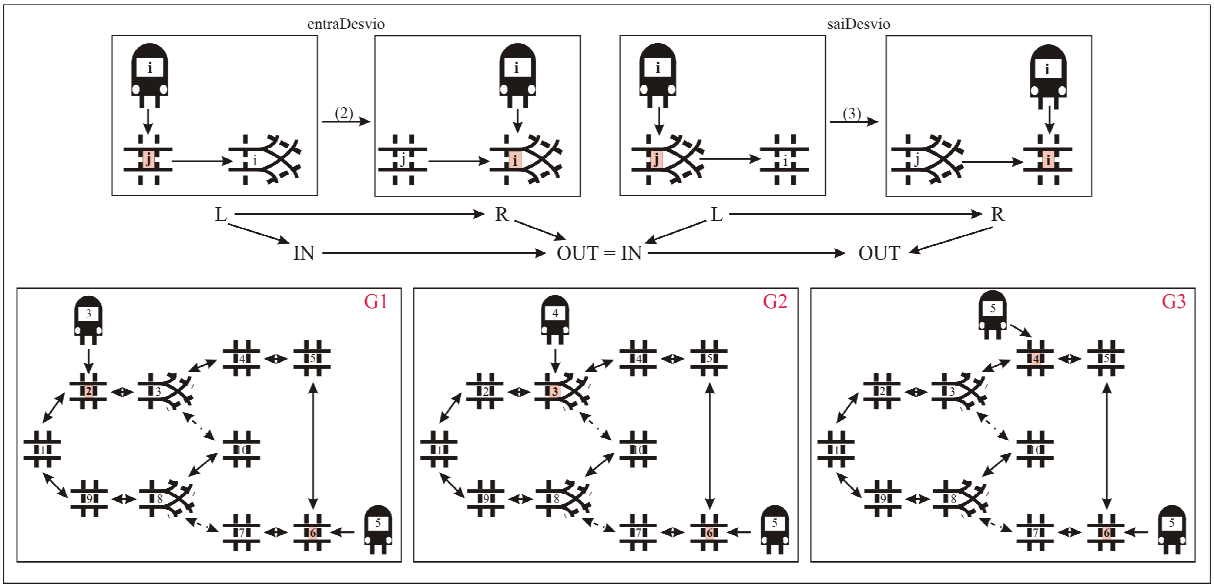


Figura 10 – Derivação sequencial

Já a derivação paralela só pode acontecer caso as regras não estejam em conflito, ou seja, não possuam acesso à escrita (remoção) em um mesmo item, formando um conjunto consistente de regras. As regras `move` e `saiDesvio` são aplicadas em paralelo em um mesmo estado do sistema na Figura 11[3]. A regra `saiDesvio` pode ser aplicada no `trem` que está no `desvio3` e a regra `move` no `trem` que está no `trilho6`. Como resultado da aplicação das regras, o `trem` do `trilho6` irá para o `trilho7` e o `trem` do `desvio3` irá para o `trilho4`. Por não estarem acessando itens (vértices/arestas) iguais, conseguem ter as regras aplicadas paralelamente em *Gini*, produzindo como resultado o grafo que é a união entre *G1* e *G2* (*Gpar*).

[3] A regra `saiDesvio` está com o lado *L* e o lado *R* da regra invertidos na imagem, por questão de melhor disposição dos objetos na figura.

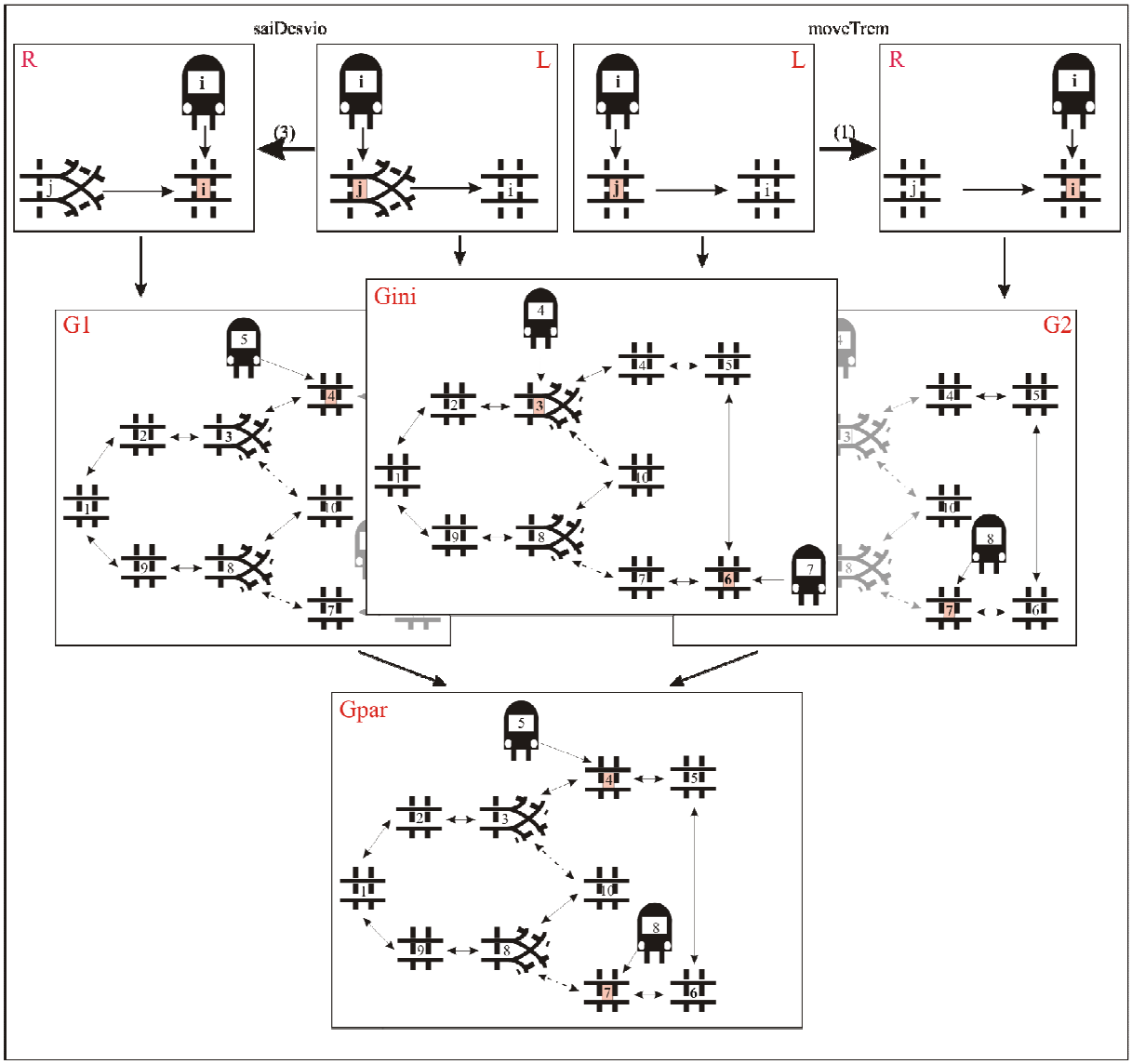


Figura 11 – Derivação paralela

As regras apresentadas na Figura 7 sinalizam os estados necessários que o grafo inicial precisa estar para que as regras possam ser aplicadas. Além destas regras simples, existe um tipo de criação de regra conhecida como condições negativas. Estas condições negativas são regras que indicam o que *não* pode acontecer para que a regra seja aplicada. Neste caso, a representação de uma condição negativa dá-se por um corte no lado esquerdo da regra, informando o que não deve estar presente para que a regra seja aplicada. De acordo com Ehrig et al. (1997, p. 278), regras que utilizam condições negativas podem ser consideradas mais inteligentes e consistentes do que as regras já apresentadas. A Figura 12 apresenta a regra (`nMove`) que move um `trem`, alterada para utilizar a condição negativa. A regra `nMove` simboliza que o `tremi` só pode trocar de `trilho` caso não haja outro `trem` ocupando o `trilho` para o qual ele deseja ir, o que pode definitivamente impedir que dois trens ocupem o mesmo trilho.

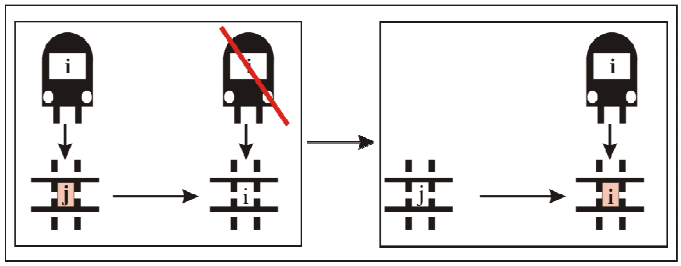


Figura 12 – Exemplo de condição negativa

2.3 PROPRIEDADES DE GRAMÁTICA DE GRAFOS

Propriedades de gramática de grafos referem-se ao comportamento da mesma, que é provida por meios de análise das computações do sistema especificado. Desta forma, é possível identificar: quais vértices são alcançáveis, dado um conjunto de regras e o grafo inicial; quais regras são aplicáveis para que os estados do sistema possam ser alterados e; quais destas regras que são consideradas aplicáveis podem ou não ser aplicadas em paralelo (permitindo a identificação do conflito entre as regras).

2.3.1 Alcançabilidade

A alcançabilidade é a propriedade que indica a possibilidade de atingir um determinado vértice, a partir de um estado inicial. Dada uma gramática *GG = (GI, N)*, onde *GI* é o grafo inicial e *N* o conjunto de regras, tem-se o conjunto de vértices alcançáveis em *GI* denotado por *VerticeAlcancavelGI*, definido no Quadro 2.

$VerticeAlcancavelGI = \{(vi,vf) \mid vi \in GI \wedge (vf \in GI \vee vf \in N) \wedge$ para cada vértice pertencente ao caminho entre *vi* e *vf* , existe uma regra aplicável$\}$
sendo:
vi = vértice inicial
vf = vértice final

Quadro 2 – Definição de vértice alcançável

Pode-se observar na Figura 13, que o vértice `trilho1` não é alcançável com a aplicação de nenhuma das regras já descritas anteriormente na Figura 7. Esta propriedade é importante porque se existir algum vértice não alcançável, significa que ele pode ser retirado da gramática por ser inútil.

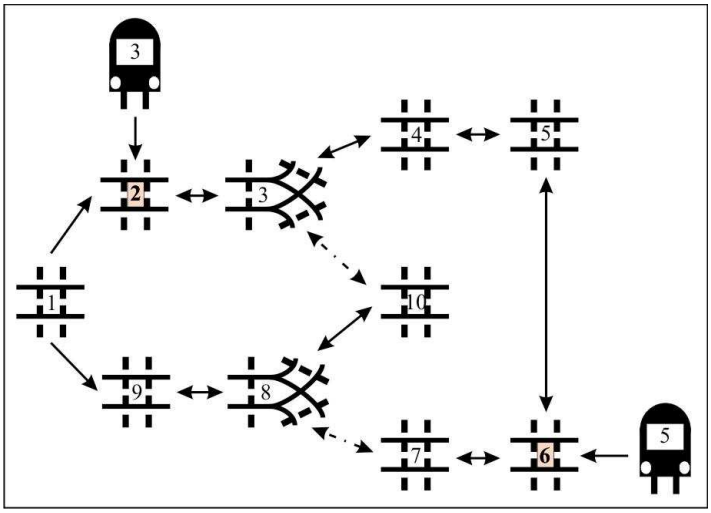


Figura 13 – Não alcançabilidade do `trilho1`

2.3.2 Aplicabilidade

A aplicabilidade é um comportamento baseado na aplicação de regras, estando relacionada com outras regras e com a semântica adotada. De acordo com Loreto (2001, p. 133), para que uma regra seja considerada aplicável, é necessário que exista pelo menos uma ocorrência para a regra, ou seja, que haja no mínimo um sub-grafo (no estado atual) que corresponda ao lado esquerdo da regra, de forma que a condição da regra possa ser satisfeita. Como uma regra pode ser aplicada de várias formas, o conjunto de regras aplicáveis é composto de pares, contendo uma regra e uma ocorrência da mesma. Para identificar a aplicabilidade, dada uma gramática *GG = (GI, N)*, onde *GI* é o grafo inicial e *N* o conjunto de regras, tem-se o conjunto de regras aplicáveis em *GI* denotado por *RegrasAplicaveisGI*, definido no Quadro 3.

$$RegrasAplicaveisGI = \{(r,m) \mid r: L \rightarrow R \in N \wedge m: L \rightarrow GI\}$$

sendo:
- r = regra
- m = ocorrência da regra
- L = lado esquerdo da regra
- R = lado direito da regra

Quadro 3 – Definição de regras aplicáveis

Caso haja alguma regra que não seja aplicada, pode ser removida da gramática. Se o estado do sistema relativo à Figura 14 fosse o estado inicial do sistema da ferrovia, algumas regras poderiam ser eliminadas por nunca serem aplicadas, como por exemplo, `entraDesvio`, `saiDesvio`, `mudaDesvio` e `aguardaDesvio`.

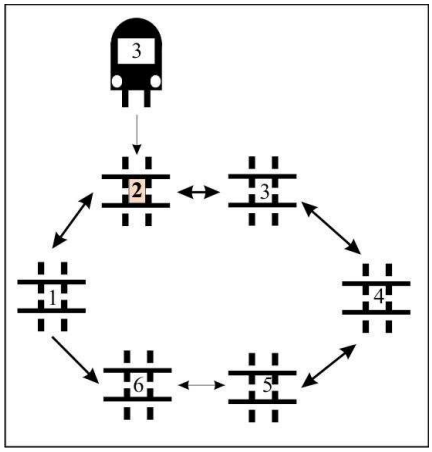


Figura 14 – Estado do sistema em que regras, da especificação da ferrovia, não seriam aplicadas

2.3.3 Conflito

Em grande parte dos sistemas, regras podem ser executadas simultaneamente (em paralelo), mas para que isso possa acontecer é preciso identificar se as regras não estão em conflito. O conflito entre as regras é caracterizado pelo tipo de acesso que as regras possuem em relação aos vértices e às arestas (SULZBACH, 2002, p. 23). Juntamente com a aplicabilidade, o conflito permite afirmar que a especificação possui um conjunto consistente de regras.

Existem dois tipos de conflito: o potencial e o real. Para identificar quais regras podem ter conflitos potenciais, é necessária uma análise sobre todas as regras definidas na gramática, antes da execução em tempo real do sistema. Já para identificar conflitos reais é necessária uma análise não somente nas regras, mas também no estado inicial/atual, porque depende das informações que são obtidas em tempo de execução. Através do conflito potencial é possível identificar os pares de regras que podem estar ou que nunca estarão em conflito real durante a execução do sistema. Mas por estarem em conflito potencial, não significa que haverá um conflito real, pois durante a execução do sistema pode acontecer de nunca serem executadas em paralelo. A vantagem de se identificar as regras que possuem conflito potencial é que quando existirem, pode-se reescrever a regra ou identificar exatamente porque o conflito aconteceu, caso haja uma execução. Outra vantagem é ser possível afirmar a ausência de conflito real, independente do estado do sistema.

Para identificar o conflito potencial, dada uma gramática $GG = (GI, N)$ e duas regras r_1 e r_2, onde GI é um grafo e N é o conjunto de regras, tem-se o conflito potencial entre r_1 e r_2,

denotado por *conflitoPotencial*, definido no Quadro 4.

conflitoPotencial = {$((r_1,r_2),(m_1,m_2)) \mid r_1: L_1 \rightarrow R_1 \in N \wedge r_2: L_2 \rightarrow R_2 \in N \wedge m_1: L_1 \rightarrow GI \wedge m_2: L_2 \rightarrow GI \wedge m_2': L_2 \rightarrow G_1 \wedge m_1': L_1 \rightarrow G_2 \wedge m_1'$ e m_2' não são morfismos totais}
sendo:
r_1 = regra
r_2 = regra
m_1 = morfismo do lado *L* de r_1 em *GI*
m_2 = morfismo do lado *L* de r_2 em *GI*
G_1 = grafo resultante do morfismo m_1
G_2 = grafo resultante do morfismo m_2
m_1' = morfismo do lado *L* de r_1 em G_2
m_2' = morfismo do lado *L* de r_2 em G_1

Quadro 4 – Definição de conflito potencial

De forma mais simples, aplica-se as regras r_1 e r_2 em *GI* através dos morfismos m_1 e m_2, o que ocasiona a geração de dois novos grafos G_1 e G_2. Tenta-se aplicar r_2 em G_1 com o morfismo m_1' e r_1 em G_2 com o morfismo m_2'. Caso não exista possibilidade de uma das regras serem aplicadas, é identificado o conflito potencial (Figura 15).

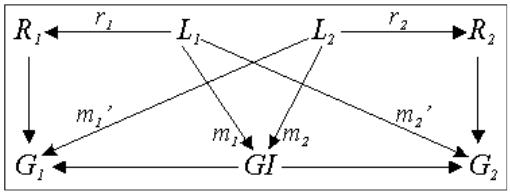


Figura 15 – Definição de conflito potencial

Já para identificar o conflito real (Figura 16), dadas duas regras aplicáveis (r_1,m_1) e (r_2,m_2) em um grafo *G*, representando o estado atual do sistema, tem-se o conflito real, denotado por *conflitoReal*, definido no Quadro 5.

conflitoReal = {$((r_1,r_2),(m_1,m_2)) \mid r_1: L_1 \rightarrow R_1 \in N \wedge r_2: L_2 \rightarrow R_2 \in N \wedge m_1: L_1 \rightarrow G \wedge m_2: L_2 \rightarrow G \wedge m_2': L_2 \rightarrow G_1 \wedge m_1': L_1 \rightarrow G_2 \wedge m_1'$ e m_2' não são morfismos totais}
sendo:
r_1 = regra aplicável
r_2 = regra aplicável
m_1 = morfismo do lado *L* de r_1 em *G*
m_2 = morfismo do lado *L* de r_2 em *G*
G = grafo correspondente ao estado atual do sistema
G_1 = grafo resultante do morfismo m_1
G_2 = grafo resultante do morfismo m_2
m_1' = morfismo do lado *L* de r_1 em G_2
m_2' = morfismo do lado *L* de r_2 em G_1

Quadro 5 – Definição de conflito real

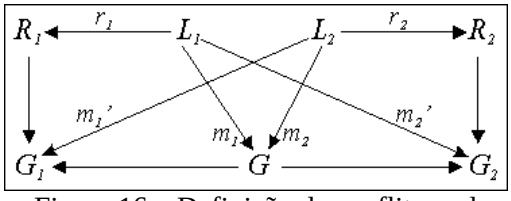


Figura 16 – Definição de conflito real

De acordo com a definição acima, uma regra estará em conflito real, quando as duas

regras aplicáveis, r_1 e r_2, forem aplicadas em G (pelos morfismos m_1 e m_2) e os grafos G_1 e G_2 não permitirem a aplicação inversa das regras (com os morfismos m_1' e m_2') nestes novos grafos gerados e os morfismos m_1' e m_2' não forem totais.

Se o sistema estivesse sendo executado na situação da Figura 17 e o `trem4` solicitasse a regra `mudaDesvio` e o `trem3` solicitasse a regra `entraDesvio` de forma paralela, haveria um conflito real entre as regras `entraDesvio` e `mudaDesvio`, pois ambas regras estariam tentando acessar (e uma modificar) a mesma aresta.

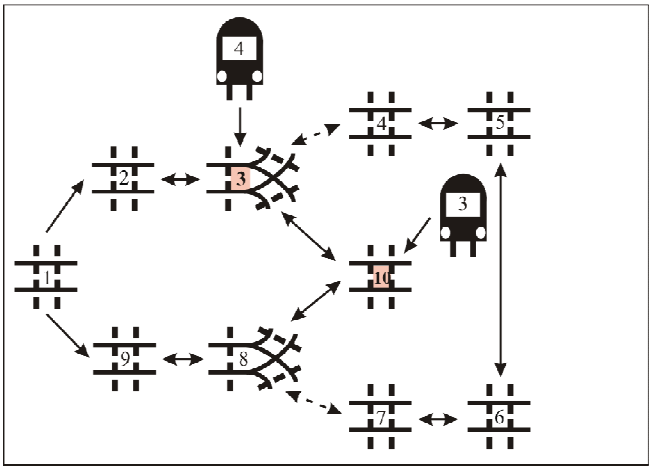


Figura 17 – Conflito entre as regras `mudaDesvio` e `entraDesvio`

2.4 TRABALHOS CORRELATOS

Existem ferramentas que oferecem suporte à criação da especificação de gramáticas de grafos, tais como: *Attributed Graph Grammar* (AGG), *Graph Rewrite GENerator* (GrGen) e *GRaphs for Object-Oriented VErification* (GROOVE).

O AGG é um ambiente de desenvolvimento para sistemas de transformação de grafos com atributos. Seu desenvolvimento foi iniciado pela *Technical University of Berlin* no ano de 1997 e tem sido mantido até hoje. O ambiente é composto por editor, interpretador e depurador para a transformação de grafos com atributos e é desenvolvido na linguagem Java.

De acordo com Taentzer et al. (2008), o AGG é uma ferramenta para editar grafos dirigidos, tipados e com atributos, com o objetivo de definir gramáticas de grafos como entrada para o componente de transformação de grafos do sistema, usando a abordagem algébrica baseada na SPO. A interface do AGG (Figura 18) é composta por uma barra de ferramentas (Figura 18(1)); duas janelas para edição de regras (Figura 18(2)); uma janela para

editar o grafo inicial e o grafo tipado (Figura 18(3)); uma janela para apresentar o grafo tipado das regras (Figura 18(4)); uma janela para editar o nome das gramáticas de grafos, grafos iniciais e regras (Figura 18(5)); e duas áreas para cadastro de vértices e arestas que podem ser coloridos (Figura 18(6)). A ferramenta possui também um analisador de conflito em um par de regras e um analisador para aplicabilidade de uma sequência de regras (Figura 18(7)).

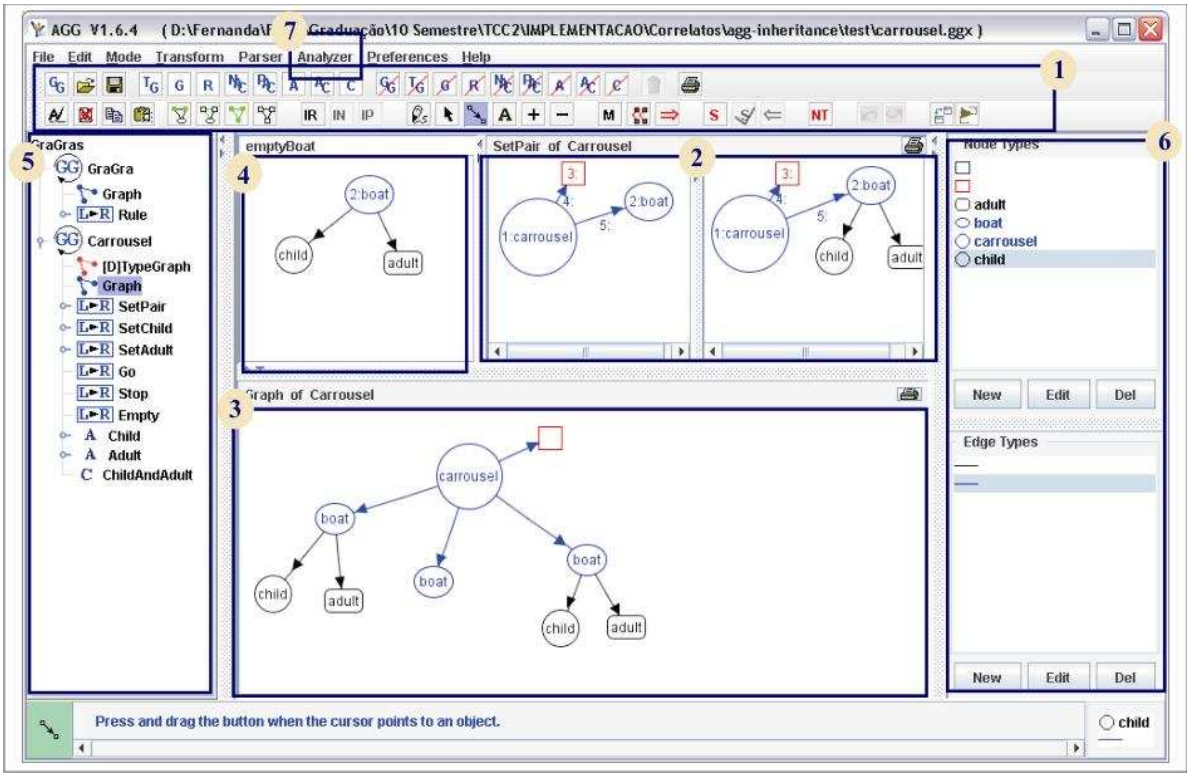


Figura 18 – Tela principal do AGG v1.6.4

O GrGen é uma ferramenta *opensource*, licenciada pela *General Public License* (GPL), de transformação de grafos (seguindo a abordagem algébrica SPO) que utiliza o conceito de grafos tipados e atributos para definir vértices e arestas (GEIß et al., 2006, p. 396). O processo de reescrita de grafos é dividido em quatro etapas: representar um grafo para o modelo, procurar um padrão no grafo, realizar aplicação da regra e finalmente escolher uma nova regra para aplicação. Geiß et al. (2008) disponibiliza uma biblioteca chamada `LibGr`, que permite a criação de regras na forma de algoritmos, que podem ser codificadas utilizando a linguagem .NET (Figura 19(1)). O GrGen possui um visualizador gráfico conhecido como `yComp` (Figura 19) que permite a criação visual de: vértices e arestas tipados; vértices e arestas com atributos (Figura 19(2)); arestas que podem ou não ser dirigidas e multigrafos. Este visualizador não possui o código aberto e, segundo Geiß et al. (2008), só pode ser utilizado para fins não comerciais.

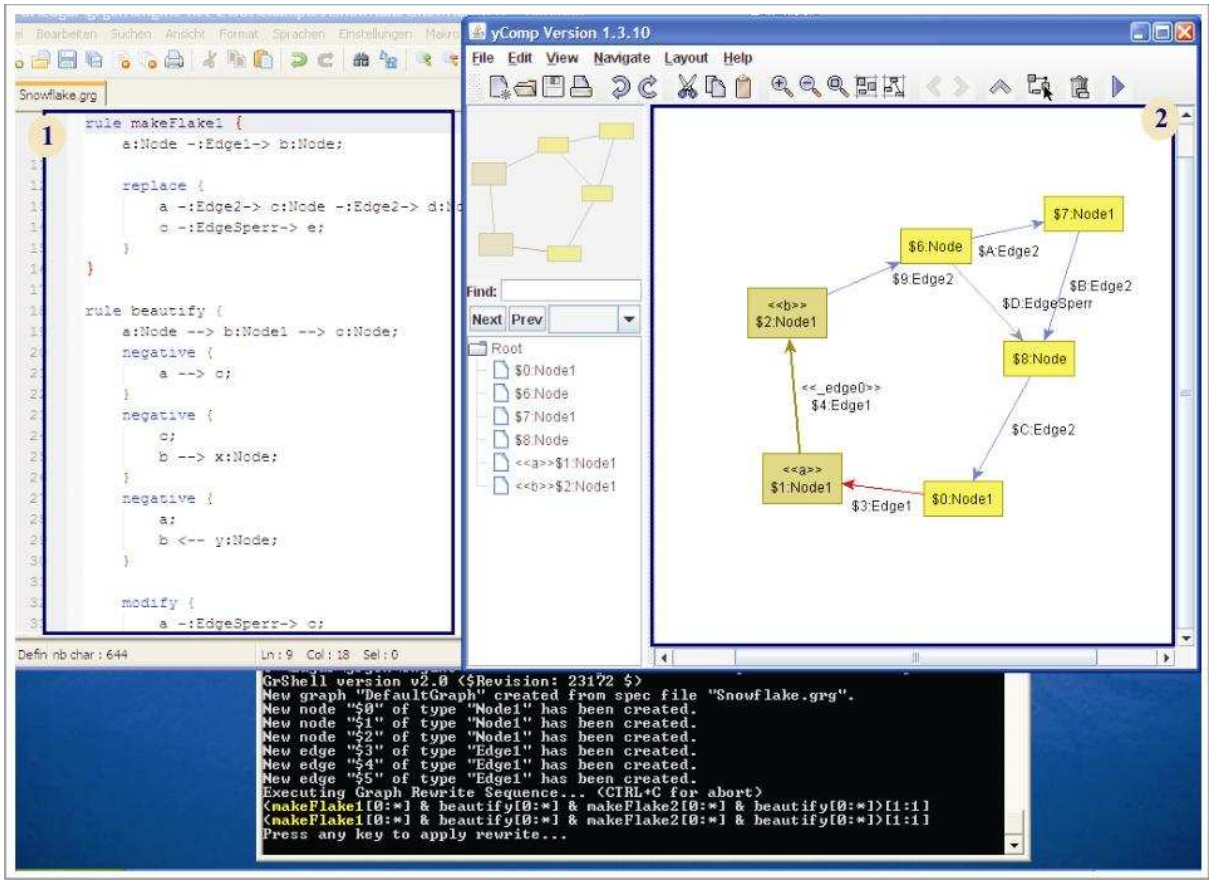


Figura 19 – Tela principal do GrGen v2.1.2

O GROOVE é um projeto desenvolvido pela *University of Twente* iniciado em 2002, que permanece em desenvolvimento até os dias atuais. É desenvolvido na linguagem Java, utilizando algumas bibliotecas de suporte como Castor, Xerces e Java *Graph* (JGraph). A abordagem algébrica SPO é utilizada para a representação e transformação de grafos. De acordo com Rensink et al. (2009), a ferramenta é composta por: um editor gráfico para a criação de grafos e regras, que faz a transformação das regras utilizando o formato *Graph eXchange Language* (GXL); e um simulador de gramáticas de grafos capaz de gerar as transições entre as regras da gramática a partir de modelos definidos em *eXtensible Markup Language* (XML). Para representar o grafo é utilizado o conceito de vértices e arestas rotuladas ou com atributos. A interface permite a criação do grafo inicial e a criação de regras (Figura 20).

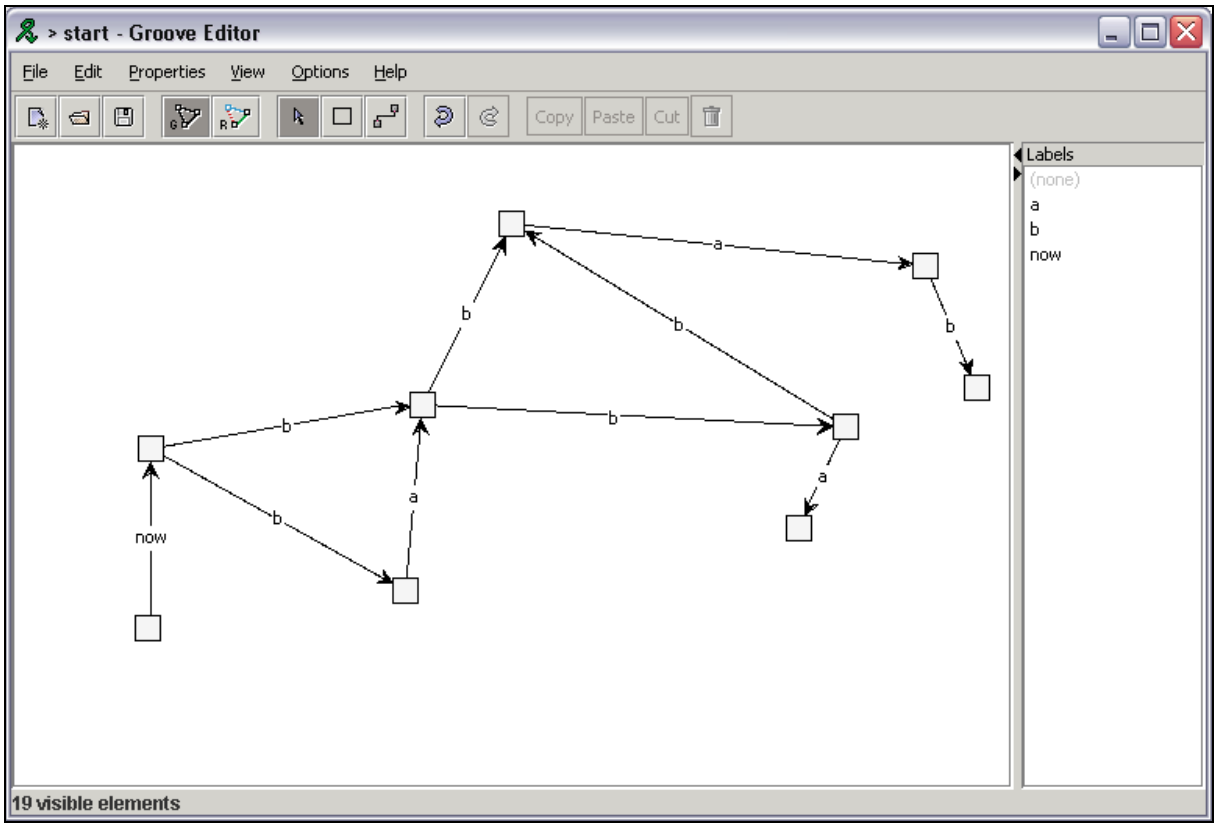


Figura 20 – Tela principal do GROOVE v3.2.1

De forma mais clara pode-se observar algumas das características dos trabalhos correlatos no Quadro 6.

	AGG	GrGen	GROOVE
criação / edição de regras	✓	✓	✓
criação / edição do grafo inicial	✓	✓	✓
criação / edição de vértices / arestas rotulados	×	×	✓
criação / edição de vértices / arestas tipados	✓	✓	×
criação / edição de vértices / arestas com atributos	✓	✓	✓
verificação de conflito	✓	×	×
verificação de aplicabilidade	✓	×	×
verificação de alcançabilidade	×	×	×
abordagem	SPO	SPO	SPO

Quadro 6 – Comparativo entre as características presentes nos trabalhos correlatos

3 DESENVOLVIMENTO

As seções seguintes descrevem a especificação, implementação e a operacionalidade do Java *Verification Graph Grammar* (JVGG). A operacionalidade é apresentada de acordo com o exemplo da ferrovia, descrito na fundamentação teórica. Por fim, são indicados os resultados obtidos com este trabalho.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

Um verificador de propriedades para gramática de grafos necessita de um módulo em que o usuário possa inserir a gramática a ser verificada. Este módulo deve permitir a criação de vértices, arestas, regras e o grafo inicial, seja através de uma interface específica ou de arquivo texto. Isso significa que somente após o usuário ter informado todos os dados (vértices, regras, grafo inicial) para o módulo citado acima é que ele poderá iniciar a verificação das propriedades de alcançabilidade, aplicabilidade e conflito parcial, tendo ao final um retorno da verificação da propriedade escolhida.

Na sequência são apresentados os Requisitos Funcionais (RF) e Requisitos Não Funcionais (RNF) que são atendidos pelo JVGG. No Quadro 7 podem ser observados os RF e no Quadro 8 os RNF.

Requisitos Funcionais (RF)	**Caso de Uso (UC)**
RF01: o sistema deve permitir ao usuário a criação de vértices	UC01, UC02
RF02: o sistema deve permitir ao usuário a criação de arestas	UC01, UC02
RF03: o sistema deve permitir ao usuário a criação do grafo inicial	UC02
RF04: o sistema deve permitir ao usuário a criação das regras da gramática	UC01
RF05: o sistema deve permitir ao usuário a escolha da propriedade a ser verificada	UC03
RF06: o sistema deve emitir um relatório com resultado da verificação da propriedade selecionada	UC03

Quadro 7 – Requisitos funcionais

Requisitos Não Funcionais (RNF)
RNF01: utilizar a abordagem algébrica *Single PushOut*
RNF02: ser implementado utilizando a linguagem de programação Java
RNF03: utilizar a biblioteca Java *Graph* (JGraph) para a criação de forma visual da gramática
RNF04: utilizar a biblioteca Java *Graph Type* (JGraphT) para manipulação de grafos

Quadro 8 – Requisitos não-funcionais

3.2 ESPECIFICAÇÃO

Na sequência é apresentada a especificação do JVGG, que foi modelada na ferramenta *Entreprise Architect*. Foram utilizados conceitos da orientação a objetos e da *Unified Modeling Language* (UML) para a criação dos diagramas de casos de uso, classe e de sequência.

3.2.1 Diagrama de casos de uso

O JVGG possui três casos de uso: `Criar regra`, `Criar grafo inicial` e `Verificar propriedades`, como pode ser observado na Figura 21. Existe uma sequência obrigatória para os casos de uso: primeiro deve-se criar o grafo inicial e as regras (não importando a ordem desta criação), na sequência é permitido fazer a verificação de propriedades.

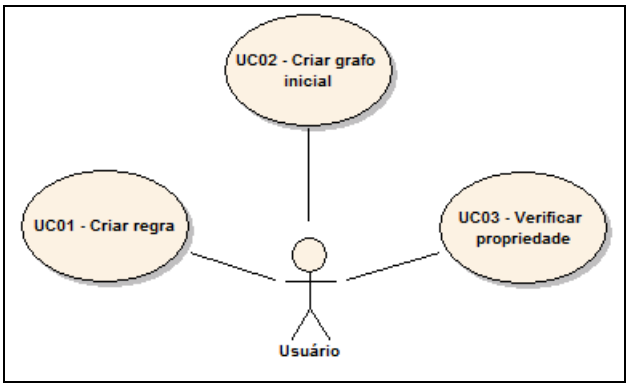


Figura 21 – Diagrama de casos de uso da ferramenta

O primeiro caso de uso (Quadro 9), designado `Criar regra`, é o caso que apresenta quais são as ações realizadas pelo usuário e pelo JVGG para criar a regra. Este caso de uso, além do cenário principal, contém dois cenários alternativos e três cenários de exceção informando possíveis erros encontrados durante a criação.

UC01 – Criar regra: permite a criação de regras da gramática de grafos	
Pré-condições	Ter vértices adicionados na gramática.
Cenário principal	01) O usuário clica no ícone “Adicionar regra” ou na opção “Regra” do menu “Gramática”. 02) O JVGG solicita o preenchimento dos dados. 03) O usuário preenche os dados e confirma. 04) O JVGG adiciona a nova regra. 05) O usuário abre o nó da regra criada e escolhe o lado L ou o lado R da regra. 06) O JVGG apresenta a área para a inserção de vértices e arestas. 07) O usuário clica no ícone “Adicionar vértice” ou na opção “Vértice” do menu “Gramática”. 08) O JVGG solicita o preenchimento dos dados. 09) O usuário preenche os dados e confirma. 10) Após ter criado todos os vértices necessários, o usuário clica no ícone “Adicionar aresta” ou na opção “Aresta” do menu “Gramática”. 11) O JVGG solicita o preenchimento dos dados. 12) O usuário preenche os dados e confirma. 13) Retorna para o passo 05.
Fluxo alternativo 01	No passo 05, o usuário escolhe o mesmo lado que havia escolhido anteriormente: 05.01) O JVGG apresenta o grafo correspondente ao lado selecionado. 05.02) Retorna para o passo 06.
Fluxo alternativo 02	No passo 05, o usuário já escolheu os dois lados e preencheu as informações: 05.01) A regra está pronta para ser utilizada.
Exceção 01	No passo 03, caso os dados solicitados não tenham sido preenchidos, é apresentada ao usuário uma mensagem solicitando que os dados sejam preenchidos.
Exceção 02	No passo 09, caso os dados solicitados não tenham sido preenchidos, é apresentada ao usuário uma mensagem solicitando que os dados sejam preenchidos.
Exceção 03	No passo 12, caso os dados solicitados não tenham sido preenchidos, é apresentada ao usuário uma mensagem solicitando que os dados sejam preenchidos.
Pós-condições	A regra deve ter sido criada.

Quadro 9 – Detalhamento do caso de uso `Criar regra`

O segundo caso de uso (Quadro 10), denominado `Criar grafo inicial`, descreve os passos necessários para que o grafo inicial seja criado. Ele possui, além do cenário principal, dois cenários de exceção informando possíveis erros encontrados durante a criação.

UC02 – Criar grafo inicial: permite a criação do grafo inicial da gramática.	
Pré-condições	Ter vértices adicionados na gramática.
Cenário principal	01) O usuário clica no nó da árvore “GrafoIni” ou no ícone “Adicionar grafo inicial” ou na opção “Grafo inicial” do menu “Gramática”. 02) O JVGG apresenta a área para a inserção de vértices e arestas. 03) O usuário clica no ícone “Adicionar vértice” ou na opção “Vértice” do menu “Gramática”. 04) O JVGG solicita o preenchimento dos dados. 05) O usuário preenche os dados e confirma. 06) Após ter criado todos os vértices necessários, o usuário clica no ícone “Adicionar aresta” ou na opção “Aresta” do menu “Gramática”. 07) O JVGG solicita o preenchimento dos dados. 08) O usuário preenche os dados e confirma.
Exceção 01	No passo 05, caso os dados solicitados não tenham sido preenchidos, é apresentada ao usuário uma mensagem solicitando que os dados sejam preenchidos.
Exceção 02	No passo 08, caso os dados solicitados não tenham sido preenchidos, é apresentada ao usuário uma mensagem solicitando que os dados sejam preenchidos.
Pós-condições	O grafo inicial deve ter sido criado.

Quadro 10 – Detalhamento do caso de uso `Criar grafo inicial`

O terceiro caso de uso (Quadro 11), designado `Verificar propriedade`, descreve o processo para realizar a verificação das propriedades: alcançabilidade, aplicabilidade e conflito. Ele possui, além do cenário principal, sete cenários alternativos e três cenários de exceção informando possíveis erros encontrados durante a verificação.

UC03 – Verificar propriedade: permite a verificação das propriedades disponíveis pelo JVGG na gramática criada pelo usuário.	
Pré-condições	Ter regras e o grafo inicial criado.
Cenário principal	01) O usuário clica na opção “Todos” do menu “Verificação”. 02) O JVGG apresenta uma tela com as propriedades disponíveis para verificação. 03) O usuário opta por verificar alcançabilidade, aplicabilidade, conflito potencial ou todas elas.
Fluxo alternativo 01	No passo 01, o usuário clica no ícone “Verificar alcançabilidade” ou na opção “Alcançabilidade” do menu “Verificação”: 01.01) O JVGG solicita o preenchimento dos dados. 01.02) O usuário preenche os dados e confirma.
Fluxo alternativo 02	No passo 01, o usuário clica no ícone “Verificar aplicabilidade” ou na opção “Aplicabilidade” do menu “Verificação”: 01.01) O JVGG solicita o preenchimento dos dados. 01.02) O usuário preenche os dados e confirma.
Fluxo alternativo 03	No passo 01, o usuário clica no ícone “Verificar conflito potencial” ou na opção “Conflito” do menu “Verificação”: 01.01) O JVGG solicita o preenchimento dos dados. 01.02) O usuário preenche os dados e confirma.
Fluxo alternativo 04	No passo 03, o usuário opta por verificar a alcançabilidade: 03.01) O usuário clica na aba “Alcançabilidade”. 03.02) O JVGG solicita o preenchimento dos dados. 03.03) O usuário preenche os dados, clica na aba “Verificação” e confirma.
Fluxo alternativo 05	No passo 03, o usuário opta por verificar a aplicabilidade: 03.01) O usuário clica na aba “Aplicabilidade”. 03.02) O JVGG solicita o preenchimento dos dados. 03.03) O usuário preenche os dados, clica na aba “Verificação” e confirma.
Fluxo alternativo 06	No passo 03, o usuário opta por verificar o conflito potencial: 03.01) O usuário clica na aba “Conflito”. 03.02) O JVGG solicita o preenchimento dos dados. 03.03) O usuário preenche os dados, clica na aba “Verificação” e confirma.
Fluxo alternativo 07	No passo 03, o usuário opta por verificar todas as propriedades: 03.01) O usuário clica na aba “Alcançabilidade”. 03.02) O JVGG solicita o preenchimento dos dados. 03.03) O usuário preenche os dados, clica na aba “Aplicabilidade”. 03.04) O JVGG solicita o preenchimento dos dados. 03.05) O usuário preenche os dados, clica na aba “Conflito”. 03.06) O JVGG solicita o preenchimento dos dados. 03.07) O usuário preenche os dados, clica na aba “Verificação” e confirma.
Exceção 01	Nos fluxos alternativos 01,02 e 03, passo 01.02, caso os dados solicitados não tenham sido preenchidos, é apresentada ao usuário uma mensagem solicitando que os dados sejam preenchidos.
Exceção 02	Nos fluxos alternativos 04, 05 e 06, passo 03.03, caso os dados solicitados não tenham sido preenchidos, é apresentada ao usuário uma mensagem solicitando que os dados sejam preenchidos.
Exceção 03	No fluxo alternativo 07, passo 03.07, caso os dados solicitados não tenham sido preenchidos, é apresentada ao usuário uma mensagem solicitando que os dados sejam preenchidos.
Pós-condições	Resultado da verificação das propriedades selecionadas no *console* do JVGG.

Quadro 11 - Detalhamento do caso de uso `Verificar propriedade`

3.2.2 Diagrama de classes

O diagrama de classes fornece uma visão de como as classes estão estruturadas e relacionadas. De forma a facilitar a estruturação e a relação entre elas, são apresentados quatro diagramas de classes.

O primeiro diagrama, na Figura 22, fornece a visão geral do JVGG, mostrando todas as classes e seus relacionamentos. As classes têm cores diferentes, para que seja possível identificar os pacotes que foram criados. O pacote `view` (de coloração cinza) contém todas as classes responsáveis pela interação com o usuário. As classes de coloração azul pertencem ao pacote `model`, mais especificamente ao sub-pacote `data` que diz respeito à estrutura de dados. Outro sub-pacote pertencente ao pacote `model` é o `verification`, que contém as classes que fazem as verificações das propriedades nas gramáticas, que estão na cor verde. A classe em amarelo pertence ao pacote `control`, que tem por finalidade fazer a conversação entre o pacote `model` e o pacote `view`.

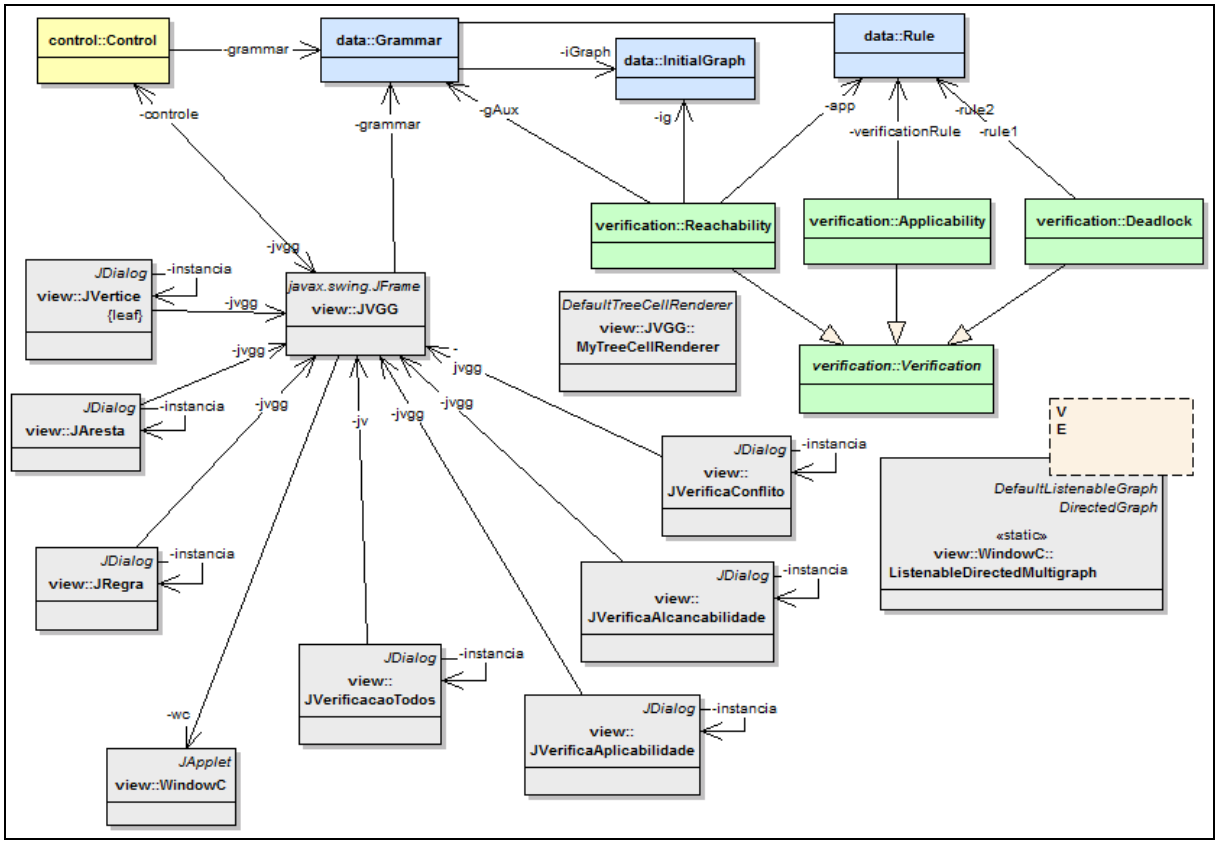


Figura 22 – Diagrama de classes do JVGG

O pacote `view` contém as classes de interação com o usuário e que são executadas durante a utilização do JVGG (Figura 23). A classe principal deste pacote é a classe `JVGG` que

é responsável por centralizar todas as janelas e também por fazer a requisição à classe `Control` quando é necessário inserir alguma informação ou quando precisa de alguma informação do modelo de dados. Outra classe importante é a `WindowC`, onde é utilizada a biblioteca JGraph, tornando-a portanto uma cópia da estrutura da gramática de grafos, só que na forma gráfica.

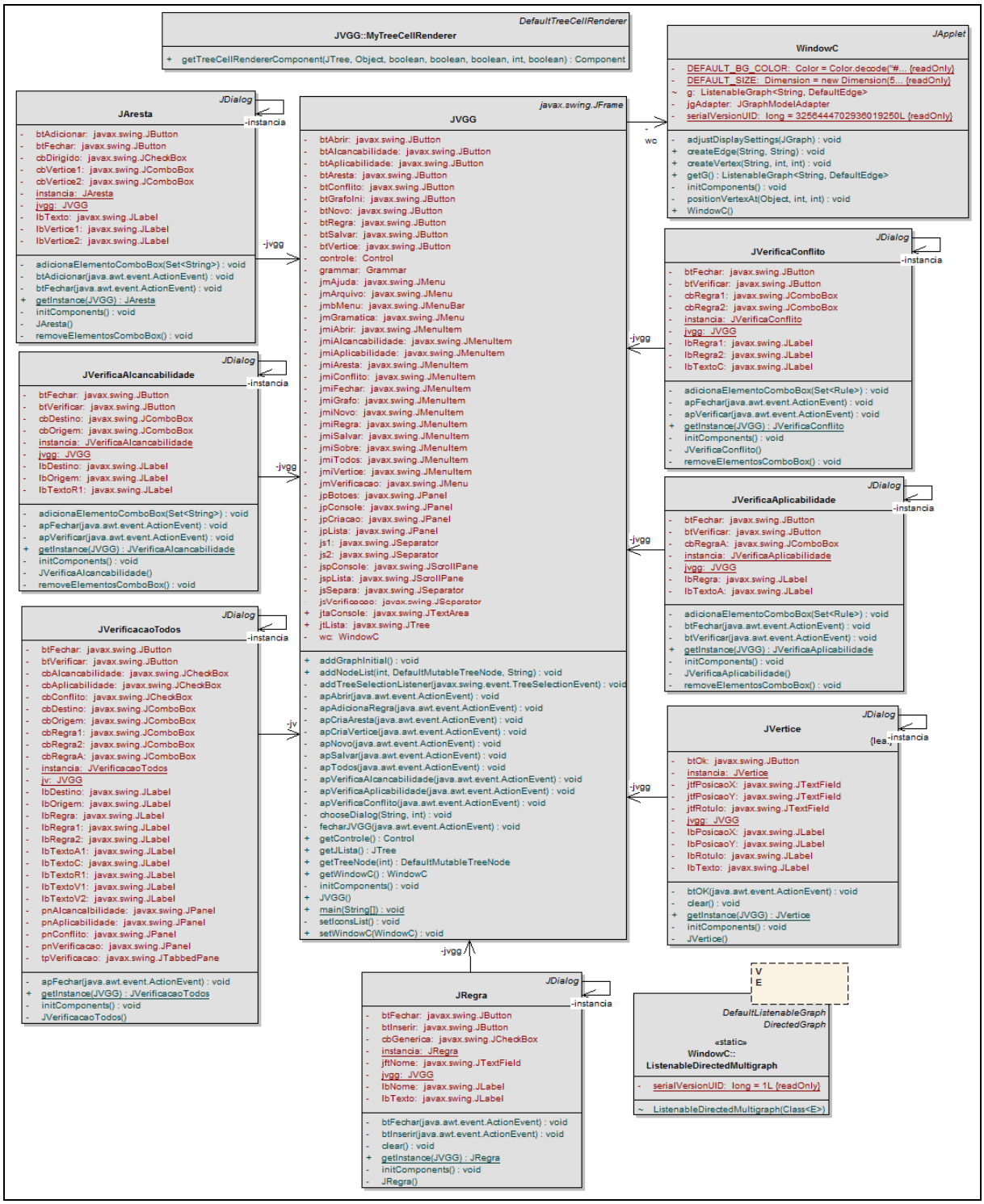


Figura 23 – Diagrama de classes do pacote `view`

A estrutura de dados e os algoritmos de verificação de propriedades estão no pacote

`model`, como mostra a Figura 24. Como já citado anteriormente, este pacote está dividido em outros dois sub-pacotes: `data` e `verification`. O sub-pacote `data` armazena as informações da gramática: conjunto de vértices, conjunto de regras e grafo inicial. As classes do pacote `data` utilizam tipos de dados (`Graph` e `DefaultEdge`) da biblioteca JGraphT.

No sub-pacote `verification` é que ficam as classes responsáveis pela verificação das propriedades de alcançabilidade, aplicabilidade e conflito potencial nas gramáticas de grafo. A JGraphT também é utilizada no sub-pacote `verification`, quando é realizada a verificação da alcançabilidade, através do algoritmo de Dijsktra no método `dijkstraWay(String, String, InitialGraph)`. O detalhamento dos principais atributos e métodos do pacote `model` encontra-se no apêndice A.

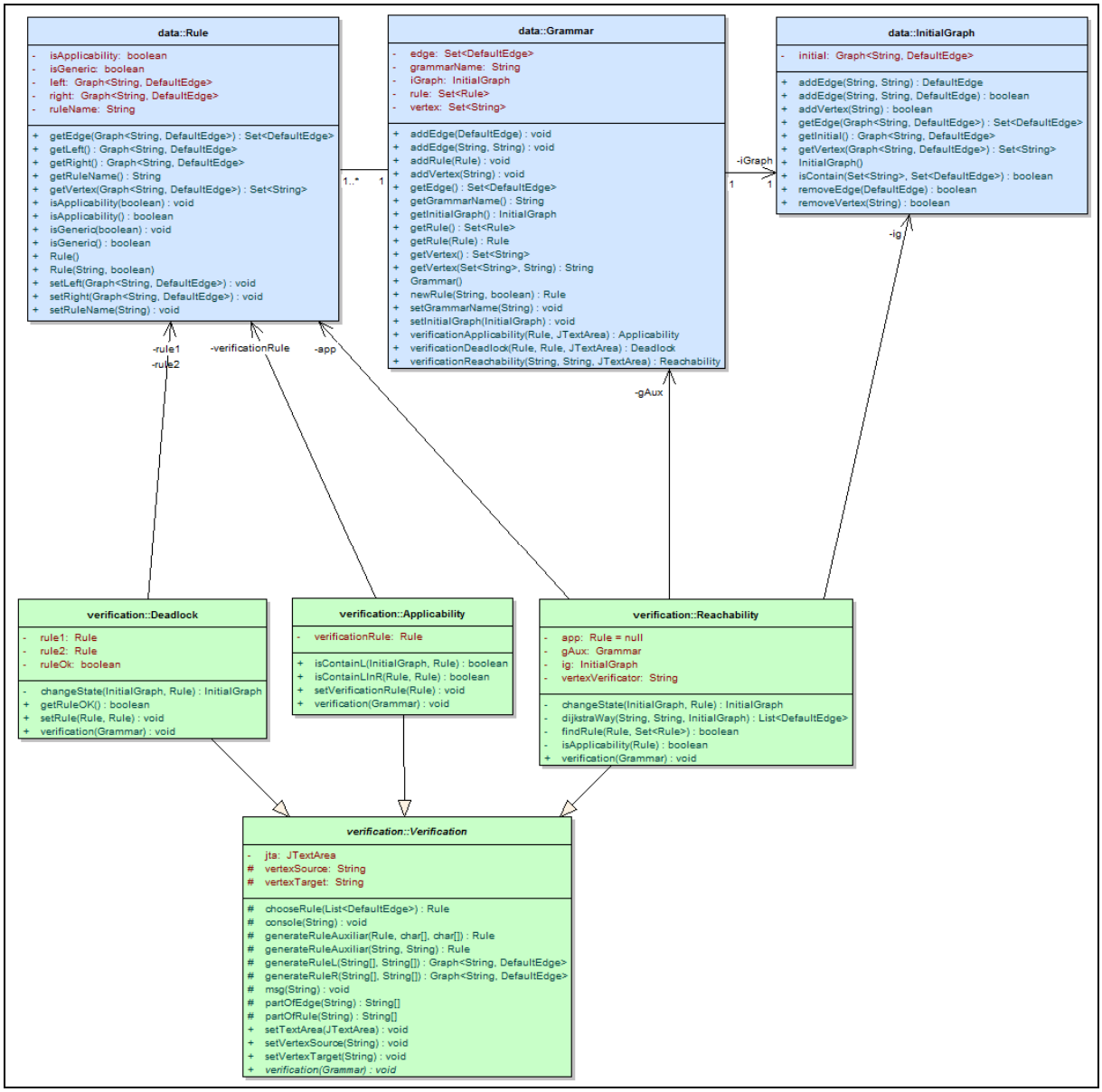


Figura 24 – Diagrama de classes do pacote `model`

A classe `Control.java`, apresentada na Figura 25, é a única classe pertencente ao

pacote `control`. É ela que faz a conversação entre os pacotes `view` e `model`.

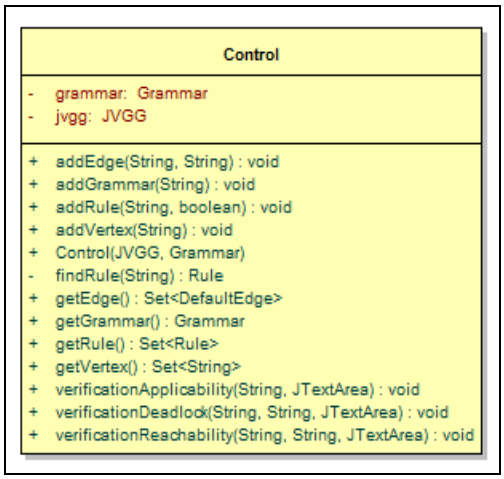


Figura 25 – Diagrama de classes do pacote `control`

3.2.3 Diagrama de sequência

Esta seção apresenta os diagramas de sequência que representam o conjunto de passos que o programa executa para realizar determinada tarefa, com base nas ações do usuário.

A Figura 26 apresenta a sequência de chamada de métodos realizada pelo sistema para que o usuário crie regras na gramática. Este diagrama corresponde ao caso de uso `Criar regra`.

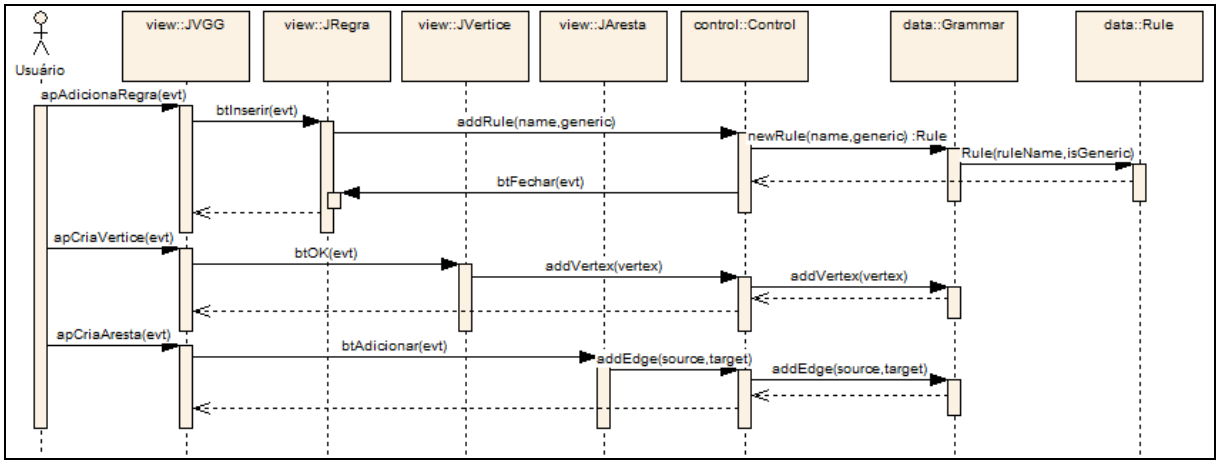


Figura 26 – Diagrama de sequência para a criação de regras

O diagrama correspondente ao caso de uso `Criar grafo inicial` é apresentado na Figura 27, que mostra quais os passos a serem executados para que o grafo inicial seja criado.

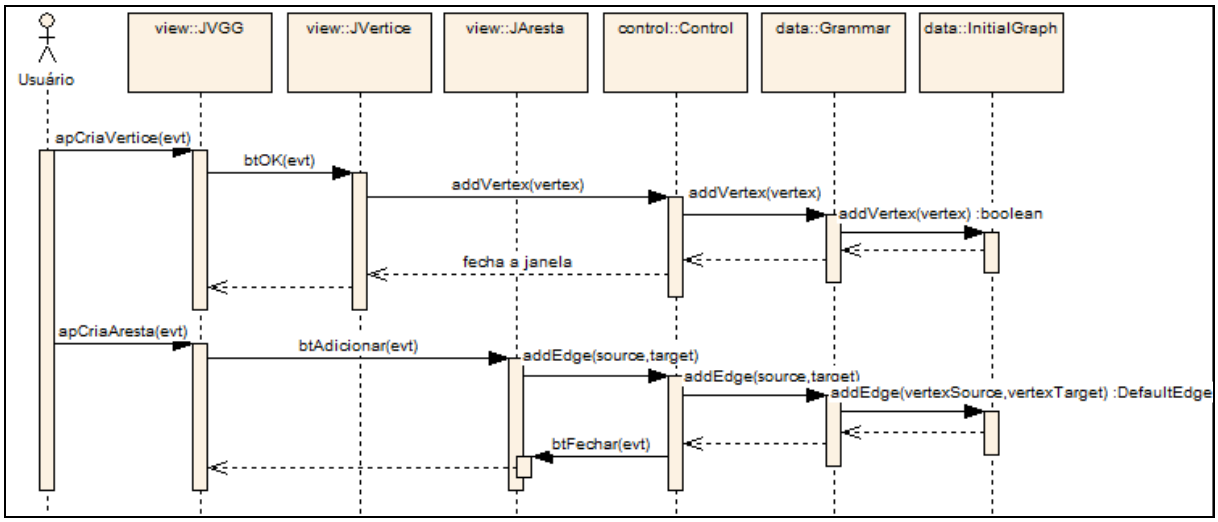


Figura 27 – Diagrama de sequência para a criação do grafo inicial

Para que o caso de uso `Verificar propriedade` possa ser melhor visualizado foram criados três diagramas de sequência, cada qual para uma propriedade. Na Figura 28 é possível visualizar a sequência de chamadas de métodos para que seja realizada a verificação da alcançabilidade.

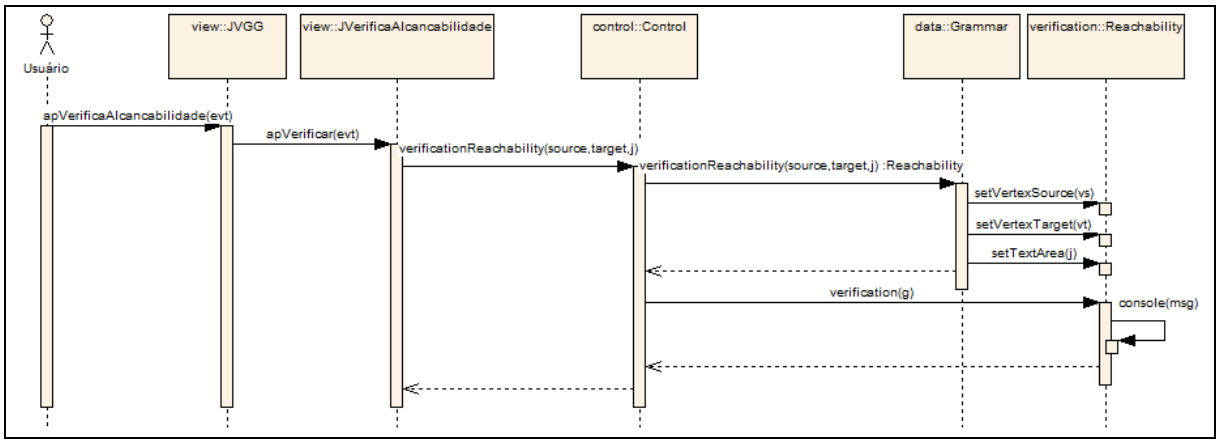


Figura 28 – Diagrama de sequência para a verificação da alcançabilidade

O conjunto de métodos chamados após a solicitação do usuário para verificar a aplicabilidade pode ser observada na Figura 29.

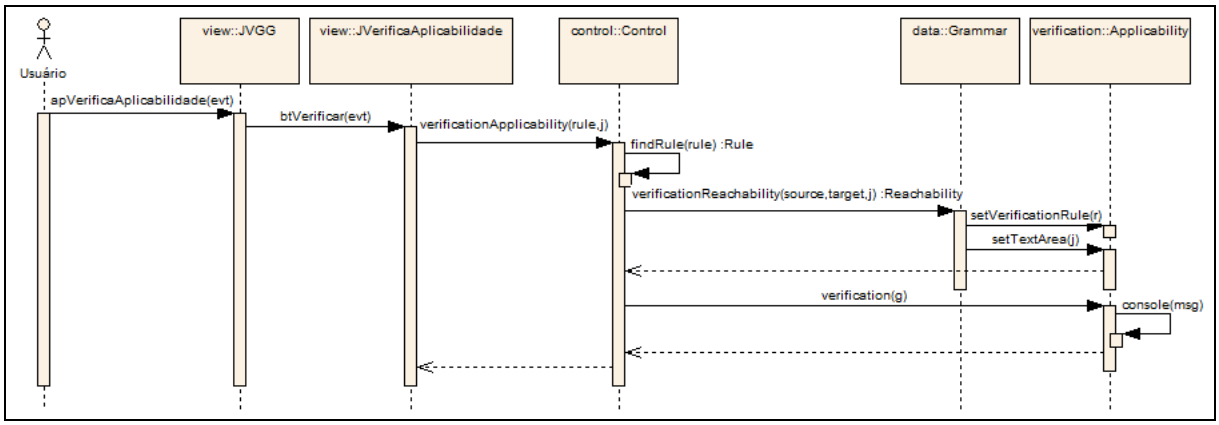


Figura 29 – Diagrama de sequência para a verificação da aplicabilidade

A verificação do conflito, na Figura 30, mostra quais ações são realizadas pelo JVGG até que o algoritmo tenha sido finalizado. Mais detalhes das chamadas e dos algoritmos de verificação são apresentados na seção 3.3.2 que trata do desenvolvimento do JVGG.

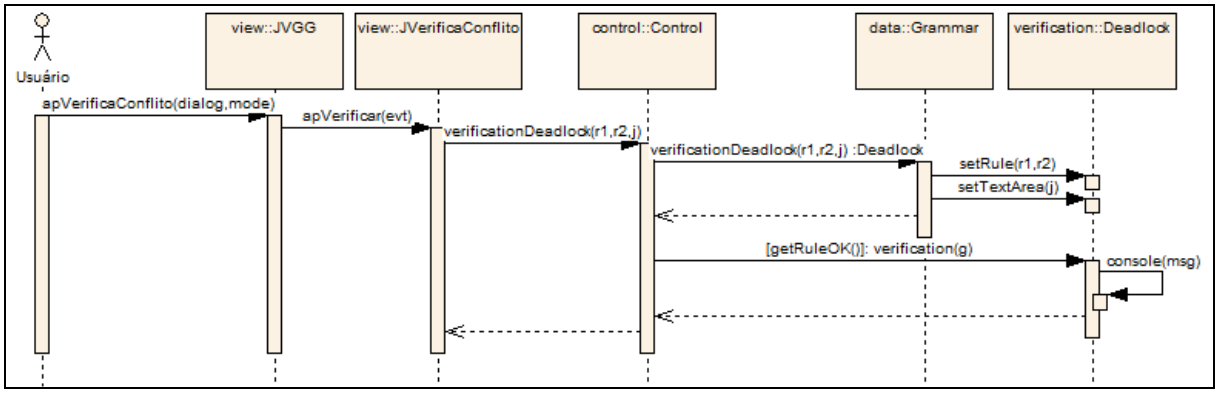


Figura 30 – Diagrama de sequência para a verificação do conflito potencial

3.3 IMPLEMENTAÇÃO

Nesta seção são apresentadas informações sobre as técnicas e ferramentas utilizadas para a implementação do JVGG, bem como o processo de implementação.

3.3.1 Técnicas e ferramentas utilizadas

O JVGG foi implementado na linguagem Java, seguindo o paradigma da orientação a objetos. Para a codificação foram utilizados dois ambientes de desenvolvimento (Eclipse e NetBeans) e duas bibliotecas (JGraphT e JGraph).

O Eclipse (versão 3.4.2) foi utilizado para a implementação de toda a camada de modelo e de controle. Já o NetBeans (versão 6.5.1) foi utilizado na implementação da camada de visão, por permitir uma ágil prototipação das janelas.

A JGraphT é uma biblioteca de código aberto, desenvolvida em Java que disponibiliza implementação de modelos matemáticos e de algoritmos. De acordo com Naveh (2009), esta biblioteca suporta vários tipos de grafos, como: dirigidos, não dirigidos, rotulados, multigrafos, entre outros. Ela foi projetada para suportar sistemas que necessitam de alta performance e que sejam de grande escala, permitindo portanto a manipulação de grafos com

milhões de vértices e arestas. No JVGG, a JGraphT foi utilizada tanto para auxiliar a criação da estrutura de dados, como também para implementar o algoritmo de verificação da alcançabilidade, onde utilizou-se o algoritmo de caminhamento em grafos proposto por Dijkstra.

A biblioteca JGraph possui funcionalidades voltadas ao desenho de grafos e é de código aberto, desenvolvida na linguagem Java, sendo totalmente compatível com as versões 1.4 ou superior. Através de sua utilização é possível criar, visualizar e interagir com grafos. Segundo Benson (2009, p. 7), além de ser utilizada para modelar grafos, é utilizada para outras finalidades, entre elas criação de: diagramas da UML, fluxogramas, esquemas de rede, circuitos eletrônicos, entre outros. Na implementação do JVGG, a JGraph foi utilizada para permitir criação, visualização e interação com as gramáticas de grafos. A JGraphT e a JGraph apesar de suas finalidades serem diferentes, são complementares, podendo ser utilizadas em conjunto (NAVEH, 2009).

3.3.2 Desenvolvimento do JVGG

O desenvolvimento do JVGG foi dividido em duas etapas: a criação da gramática e a verificação das propriedades. A seguir são apresentados trechos de código fonte ilustrando as etapas citadas acima.

3.3.2.1 Etapa 1: criação da gramática

Para que uma gramática seja completa, é necessário ter vértices (simbolizando os rótulos), regras e grafo inicial. Por este motivo, quando é iniciada a execução do JVGG, é carregada uma área disponível para a criação dos vértices que serão utilizados na especificação das regras e do grafo inicial.

A criação de vértices é realizada com a associação de um vértice à gramática através da chamada do método `addVertex(String vertex)` da classe `Grammar`. Este método armazena o rótulo do vértice no atributo `vertex`, do tipo `Set<String>`, que representa o conjunto de vértices da gramática.

Após a associação de vértices à gramática, é possível iniciar a criação do grafo inicial

ou das regras. A criação do grafo inicial dá-se pela criação de arestas entre os vértices que estão disponíveis para a utilização da gramática. Antes de adicionar arestas, é necessário informar à classe `InitialGraph` quais são os vértices da gramática que serão utilizados na criação do grafo inicial. Esta associação é feita através do método `addVertex(String vertex)` pertecente à classe `InitialGraph`. As arestas só podem ser criadas após informar no mínimo dois vértices, pois um vértice não pode estar conectado por uma aresta com ele mesmo. A partir deste momento, torna-se possível adicionar arestas à gramática utilizando o método `addEdge(String vertexSource, String vertexTarget)`, que pertence à classe `InitialGraph`. Tendo os vértices relacionados e as arestas entre os vértices criadas, a criação do grafo inicial é concluída.

As regras são criadas de forma análoga à criação do grafo inicial. O que as difere é que as regras contêm dois grafos, um que representa o lado L e outro que representa o lado R da regra. Para a criação da regra é solicitado ao usuário que dê um nome para a regra e informe se ela é genérica ou não. Se ela for genérica, significa que pode ser utilizada em outras situações em que os seus valores sejam diferentes, facilitando sua criação, pois o usuário não precisará digitar várias vezes a mesma regra com valores dos vértices diferentes. A partir das informações anteriores, a regra é criada invocando o construtor `Rule(String ruleName, boolean isGeneric)` que além de associar as informações passadas como parâmetro, cria os lados L e R da regra instanciando dois objetos de `SimpleDirectedGraph`, que é uma classe pertencente a biblioteca JGraphT que cria grafos dirigidos simples. Para associar vértices e arestas as lados L e R da regra, os métodos `addVertex(String arg0)` e `addEdge(String arg0, String arg1)`, que pertencem à classe `Graph` da biblioteca JGraphT, são invocados pelo respectivo lado que deseja realizar a associação.

Com a gramática completa (vértices, regras e grafo inicial), torna-se possível realizar a verificação das propriedades de alcançabilidade, aplicabilidade e conflito potencial, que são descritas na próxima seção.

3.3.2.2 Etapa 2: verificação das propriedades

Com a utilização do JVGG é possível verificar três propriedades existentes nas gramáticas de grafos, sendo elas: alcançabilidade, aplicabilidade e conflito potencial.

A implementação dos algoritmos foi realizada em três classes: `Reachability`,

`Applicability` e `Deadlock`, sendo que elas são uma especialização da classe `Verification`. A classe `Verification` possui métodos e atributos que podem ser utilizados pelas classes que a estendem. Possui também um método abstrato `verification(Grammar g)` onde as classes especializadas implementam o respectivo algoritmo de verificação.

A alcançabilidade é a propriedade que indica a possibilidade de atingir um determinado vértice, a partir de um estado inicial. Para isso, foi implementada a classe `Reachability`, que além de outros métodos, tem a implementação do algoritmo de alcançabilidade no método `verification(Grammar g)` (Quadro 12). A idéia básica do algoritmo é: dado um vértice origem (`vertexSource`) e um vértice destino (`vertexTarget`), verificar se existe um caminho no grafo inicial que possa ser percorrido entre eles através do método `dijkstraWay(String vertexSource, String vertexTarget, InitialGraph ig)` (linha 04) que instancia um objeto de `DijkstraShortestPath<String, DefaultEdge>`, classe que contém o algoritmo de Dijkstra e que pertence à biblioteca JGraphT. Com o caminho retornado pela execução do método `dijkstraWay`, são selecionadas regras para tentar aplicar (linha 07-09), durante todo o percurso até chegar no `vertexTarget` e identificar que o vértice é alcançável (linha 15-19). Caso não exista regra para aplicar (linha 20-25) ou não exista um caminho que possa ser traçado por Dijkstra (linha 26-28), é identificado que o vértice não é alcançável, podendo ser retirado da gramática por não ser utilizado.

```
01 public void verification(Grammar g) {
02  this.gAux = g;
03  //Monta o caminho utilizando o algoritmo proposto por Dijkstra
04  List<DefaultEdge> de = dijkstraWay(this.vertexSource,this.vertexTarget,
gAux.getInitialGraph());
05  if (de != null) {
06    //Escolhe uma regra pra aplicar
07    Rule r = chooseRule(de);
08    //Se ela pertence ao conjunto de regras
09    if (findRule(r, gAux.getRule())) {
10      //Se ela é aplicável
11      if (isApplicability(r)) {
12        //Cria um grafo inicial auxiliar a partir da aplicação da regra
13        this.ig = changeState(this.ig, app);
14        gAux.setInitialGraph(ig);
15        if (!this.vertexTarget.equals(vertexVerificator)) {
16          verification(gAux);
17        } else {
18          console("É alcançável.");
19        }
20       } else {
21        console("Regra não pode ser aplicada.");
22       }
23    } else {
24      console("Não alcançável: Sem regras suficientes para aplicar.");
25    }
26  } else {
27    console("Não alcançável: Sem caminho para percorrer.");
28  }
29 }
```

Quadro 12 – Método `verification(Grammar g)` implementado na classe `Reachability`

Na classe `Applicability` está codificado o algoritmo que verifica se uma regra é aplicável, ou seja, se existe pelo menos um sub-grafo no estado atual do grafo que corresponda ao lado esquerdo da regra. O algoritmo é executado para uma regra específica. Para isso, ele inicia perguntando se em algum momento a regra já foi verificada (linha 03), consequentemente se já foi tida como aplicável. Se a resposta for verdadeira, o algoritmo é finalizado indicando que a regra é aplicável (linha 04), caso contrário dá-se início a verificação. Existem dois estados de regras no JVGG, as que são genéricas e as que são fixas. Quando uma regra é fixa, basta verificar se existe uma ocorrência do lado L da regra no grafo inicial para identificar se ela é aplicável ou não (linhas 77-81). Já quando ela é genérica, é necessário gerar todas as situações de aplicação possível para dar uma resposta (linhas 7-76). Então o primeiro vértice da regra genérica do lado L é escolhido (linhas 20-24) e é realizada uma busca para encontrar todos os vértices que iniciam com o identificador do vértice (letra). Na sequência são buscados todos os adjacentes dos vértices encontrados (linhas 26-30). Para cada valor de rótulo (número) dos adjacentes, é gerada uma nova regra com base no valor do primeiro vértice e do vértice adjacente, ou seja, gera-se uma regra fixa auxiliar (linhas 56-66). Com as regras geradas, tenta-se aplicar a regra verificando se existe uma ocorrência do lado L da regra no grafo inicial. Se existir, ela é aplicável, senão não é aplicável (linhas 68-72). Este algoritmo apresenta todas as situações possíveis, salientando que se em ao menos uma situação a regra é aplicada, ela já é considerada aplicável. No Quadro 13 é possível observar o código fonte referente à verificação da alcançabilidade, que está codificado no método `verification(Grammar g)` da classe `Applicability`.

```
01 public void verification(Grammar g) {
02 //Verifica se já tem alguma informação armazenada no atributo de aplicabilidade
da regra
03 if (this.verificationRule.isApplicability() == true) {
04   console("É Aplicável!");
05 } else {
06 // Verifica se a regra é genérica
07 if (this.verificationRule.isGeneric() == true) {
08   String[] valores = {"0","0","0","0","0",”0",”0"};
09   String[] auxilio = {"0","0","0","0","0","0","0","0","0","0"};
10   String[] posicao = { "X", "Y", "Z", "A", "B", "C", "D" };
11   ArrayList<String> setVertexChoice = new ArrayList<String>();
12   // Armazena o conjunto de vértices existentes
13   Set<DefaultEdge> sVer =
this.verificationRule.getEdge(this.verificationRule.getLeft());
14   // Separa os vértices
15   String[] sEdArr = partOfRule(sVer.toString());
16   String[] sVerArr = partOfEdge(sEdArr[0]);
17   String[] sVertexArr = partOfRule(g.getInitialGraph().getVertex(
g.getInitialGraph().getInitial()).toString());
18   String procurado = "";
19   for (int i = 0; i < sVertexArr.length; i++) {
20     //Encontra a primeira ocorrência de um vértice do lado L para começar a
substituição
```

```
21     if (sVertexArr[i].trim().startsWith(sVerArr[0].substring(0, 1))) {
22       procurado = sVerArr[0].substring(0, 1);
23       setVertexChoice.add(sVertexArr[i].trim());
24      }
25   }
26   for (String vertex : setVertexChoice) {
27     int f = 0, a = 0;
28      valores[f] = vertex.substring(1);
29    // Encontra os adjacentes do vértice escolhido
30    Set<DefaultEdge> adjacentes =
g.getInitialGraph().getInitial().edgesOf(vertex);
31    // Para cada adjacente
32    for (DefaultEdge de : adjacentes) {
33      String[] sVerAdj = partOfEdge(de.toString());
34      // Encontra os adjacentes do "adjacente"
35       Set<DefaultEdge> adjacentesAdj =
g.getInitialGraph().getInitial().edgesOf(sVerAdj[1].trim());
36      // Para cada um deles
37      for (DefaultEdge de2 : adjacentesAdj) {
38        String[] sVerAdj2 = partOfEdge(de2.toString());
39        if (!(sVerAdj2[0].trim().substring(1).equals(valores[f].trim())) &&
procurado.equals(sVerAdj2[0].trim().substring(0,1))) {
40          f++;
41          valores[f] = sVerAdj2[0].substring(1).trim();
42           break;
43         } else {
44          if(!(sVerAdj2[1].trim().substring(1).equals(valores[f].trim()) )&&
procurado.equals(sVerAdj2[0].trim().substring(0,1))) {
45            f++;
46            valores[f] = sVerAdj2[1].substring(1).trim();
47             break;
48           } else {
49            if(!valores[f].contains(sVerAdj2[0].trim().substring(1))){
50               auxilio[a] = sVerAdj2[0].trim().substring(1);
51             } else {
52               if(!valores[f].contains(sVerAdj2[1].trim().substring(1))){
53                 auxilio[a] = sVerAdj2[1].trim().substring(1);
54                 a++;
55               }}}}}
56      String newRuleL= verificationRule.getLeft().edgeSet().toString();
57      String newRuleR= verificationRule.getRight().edgeSet().toString();
58      String newRuleL2;
59      String newRuleR2;
60      for (int b = 0; !valores[b].equals("0"); b++) {
61        newRuleL = newRuleL.replace(posicao[b], valores[b]);
62        newRuleR = newRuleR.replace(posicao[b], valores[b]);
63        for (int c = 0; !auxilio[c].equals("0"); c++) {
64          newRuleL2 = newRuleL.replace(posicao[b+1], auxilio[c]);
65          newRuleR2 = newRuleR.replace(posicao[b+1], auxilio[c]);
66          Rule r = generateRuleAuxiliar(newRuleL2.trim(), newRuleR2.trim());
67          console(r.getLeft().toString() + " - "+r.getRight(). toString());
68          if (isContainL(g.getInitialGraph(), r)) {
69            console("É Aplicável.");
70           } else {
71            console("Não é aplicável.");
72          }
73        }
74      }
75    }}
76 } else {
77  if (isContainL(g.getInitialGraph(), this.verificationRule)){
78    console("É Aplicável.");
79  } else {
80     console("Não aplicável.");
81  }}}}
```

Quadro 13 - Método `verification(Grammar g)` implementado na classe `Applicability`

Para identificar o conflito potencial é necessária uma análise sobre duas regras

envolvidas na verificação. O algoritmo para verificar esta propriedade está na classe `Deadlock`, para a qual são informadas duas regras: `rule1` e `rule2` em que se deseja fazer a verificação. A verificação dá-se pela aplicação de `rule1` e `rule2` no grafo inicial `g`, na qual gera dois outros grafos auxiliares: `G1`, que é gerado pela aplicação de `rule1` em `g` (linha 03), e `G2`, que é resultante da aplicação de `rule2` em `g` (linha 05). Com o `G1` e o `G2` gerados, tenta-se aplicar as regras inversamente, ou seja, aplica-se `rule1` em `G2` e `rule2` em `G1` gerando `morphG1` (linha 08) e `morphG2` (linha 10), respectivamente. Caso os grafos `G1` e `morphG2` sejam iguais e os grafos `G2` e `morphG1` sejam também iguais (linha 12), não existe conflito potencial (linha 13). Já se eles forem diferentes, indica que há conflito potencial entre as regras (linha 15). O código fonte que faz a verificação do conflito potencial pode ser visto no Quadro 14, onde é apresentado o método `verification(Grammar g)` da classe `Deadlock`.

```
01 public void verification(Grammar g) {
02   InitialGraph G1, G2;
03   G1 = changeState(g.getInitialGraph(), rule1);
04   String g1 = G1.getInitial().toString();
05   G2 = changeState(g.getInitialGraph(), rule2);
06   String g2 = G2.getInitial().toString();
07   InitialGraph morphG1, morphG2;
08   morphG1 = changeState(G1, rule2);
09   String m1 = morphG1.getInitial().toString();
10   morphG2 = changeState(G2, rule1);
11   String m2 = morphG2.getInitial().toString();
12   if(m1.equals(g2) && m2.equals(g1)){
13     console("Sem conflito potencial entre as regras "+ rule1.getRuleName()+ " e
"+ rule2.getRuleName());
14   } else {
15     console("Conflito potencial entre as regras "+ rule1.getRuleName()+ " e "+
rule2.getRuleName());
16   }
17 }
```

Quadro 14 - Método `verification(Grammar g)` implementado na classe `Deadlock`

3.3.3 Operacionalidade da implementação

A operacionalidade do JVGG é apresentada nesta seção utilizando o exemplo da ferrovia, descrito na página 20, porém com algumas limitações indicadas no Quadro 15.

Como foi especificado	Como é modelado
Trem: ...o número dentro da cabine do trem indica qual é o próximo trecho que ele deve ir...	O identificador do trilho é o próprio rótulo do vértice, que no caso dos trens indicará apenas onde ele se encontra e não para onde ele irá.
Regras: ... quatro regras: ... O trilho é controlado pela regra `mudaDesvio`.	Existem duas regras fixas: `mudaDesviod3` e `mudaDesviod8`, as outras continuam como estão.
Trilho: ...possui um identificador que é um número no interior do trilho. Se este número for com o fundo transparente significa que está livre, se ele tiver alguma coloração, significa estar ocupado por um trem...	O identificador do trilho é o próprio rótulo do vértice e não existe diferenciação de cores. Portanto, não há como saber se o trilho está ocupado.
Desvios: ...arestas tracejadas indicam que o desvio não está conectado...	Não há representação de arestas tracejadas para o desvio que não está conectado. Para resolver esta situação, foram criadas as regras fixas `mudaDesviod3` e `mudaDesviod8`.

Quadro 15 – Limitações no JVGG para o exemplo da ferrovia

Ao executar o JVGG, o usuário irá encontrar a tela apresentada na Figura 31, onde existem quatro áreas distintas: a de menus e botões (Figura 31(1)), a que contém os elementos da gramática (Figura 31(2)), a de criação da gramática (Figura 31(3)) e a de visualização dos resultados dos algoritmos (Figura 31(4)), que foi nomeada de *console*.

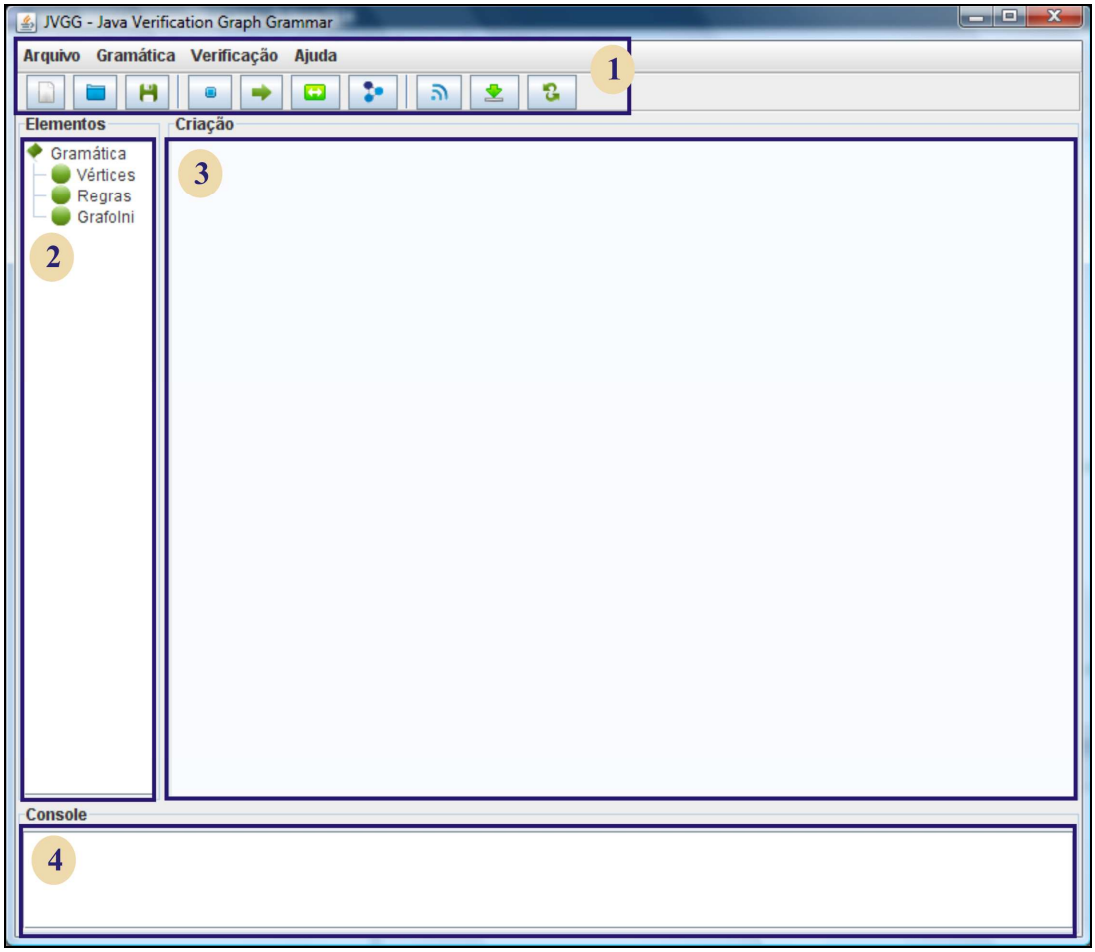


Figura 31 – Tela inicial do JVGG

A área de menus e botões contém quase todas as mesmas funcionalidades. Nos menus

é que existem algumas opções a mais, como: verificar todas as propriedades e sobre o JVGG. Os botões podem ser vistos na Figura 32. Da esquerda para a direita, tem-se: o primeiro botão permite a criação de um novo arquivo, o segundo a abertura de uma gramática já existente, o terceiro salva a gramática atual, o quarto botão representa a criação de vértices, o quinto a criação de arestas, o sexto a criação de regras, o sétimo a criação de grafo inicial e os três últimos botões servem para fazer a verificação de alcançabilidade, aplicabilidade e conflito potencial, respectivamente. Observa-se que o segundo e o terceiro botão estão desabilitados.

Figura 32 – Botões de acesso rápido do JVGG

A área que contém os elementos da gramática serve como um norteador para a criação da gramática e para que o usuário possa visualizar rapidamente suas ações. Ela é criada em estrutura de árvore para facilitar a visualização dos elementos que são divididos em três categorias: vértices, regras e grafo inicial. O elemento vértice vai conter os vértices pertencentes à gramática, o elemento regras por sua vez vai conter o nome das regras da gramática com seus respectivos lados *L* e *R* e o elemento grafo inicial vai conter o grafo inicial da gramática (que é único).

A área identificada por criação (Figura 31(3)) é onde é possível criar, visualizar e manipular o grafo inicial e as regras da gramática. Esta área permite que o usuário interaja com os vértices e arestas durante a criação dos elementos, que é facilitada pela representação visual, a qual fornece um melhor entendimento da gramática.

A área de *console* foi reservada para que os resultados das verificações das propriedades da gramática apareçam na forma de texto. Ao verificar uma propriedade são exibidas informações que podem conduzir o usuário no procedimento que foi realizado, emitindo ao final um parecer referente à propriedade verificada.

Para iniciar a criação dos elementos da gramática, primeiramente é necessário criar os vértices que serão utilizados na gramática. Para isso, o usuário deve clicar no elemento “Vértices”, indicando sua intenção em informar os vértices da gramática. Após isso, ele pode escolhe clicar no botão “Adicionar vértice” ou então acessar a opção “Vértices”, do menu “Gramática”. Executando uma das duas ações é possível informar o nome do rótulo (que deve ser composto por uma letra e um número) e a sua posição (X, Y) na área de criação (Figura 33). Se o usuário clicar no botão “Inserir”, ele poderá visualizar o vértice adicionado na área de criação. Já a remoção de um vértice se dá ao selecionar o vértice na área de criação e pressionar a tecla *delete*.

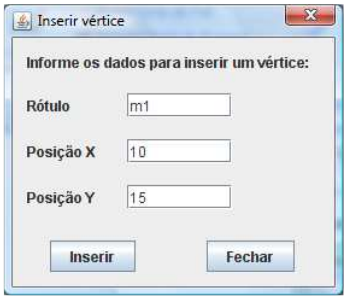


Figura 33 – Inserção de vértice `m1`

Após a inserção de todos os vértices que podem ser utilizados na gramática, o usuário pode optar por inserir o grafo inicial ou regras. Caso o usuário opte por inserir o grafo inicial, ele deve clicar no elemento “GrafoIni” e realizar o mesmo procedimento citado anteriormente para adicionar vértices. Com a criação dos vértices também no grafo inicial, pode-se iniciar a criação de arestas, que o usuário pode optar por clicar no botão “Adicionar aresta” ou na opção “Aresta” do menu “Gramática”. Executando uma das duas opções, é possível indicar o vértice origem e o vértice destino da aresta e também selecionar se é desejado que se crie uma aresta inversa, ou seja, do destino para a origem (Figura 34). Ao clicar no botão “Adicionar”, a aresta é inserida na área de criação. A remoção de uma aresta acontece quando o usuário seleciona a aresta na área de criação e pressionar a tecla *delete*.

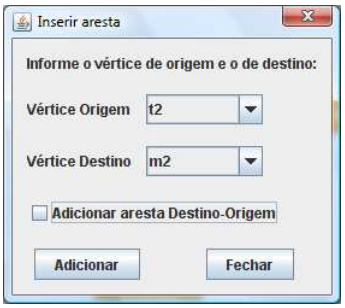


Figura 34 – Inserção de aresta `t2:m2`

Com a criação dos vértices e arestas, o grafo inicial termina de ser criado (Figura 35), podendo avançar para a criação das regras da gramática. É possível notar que na área de criação da Figura 35 as arestas estão com referências, como por exemplo (t2:m2). Estas referências são fornecidas pela biblioteca JGraph, que visam auxiliar na identificação das arestas, indicando sua origem e seu destino.

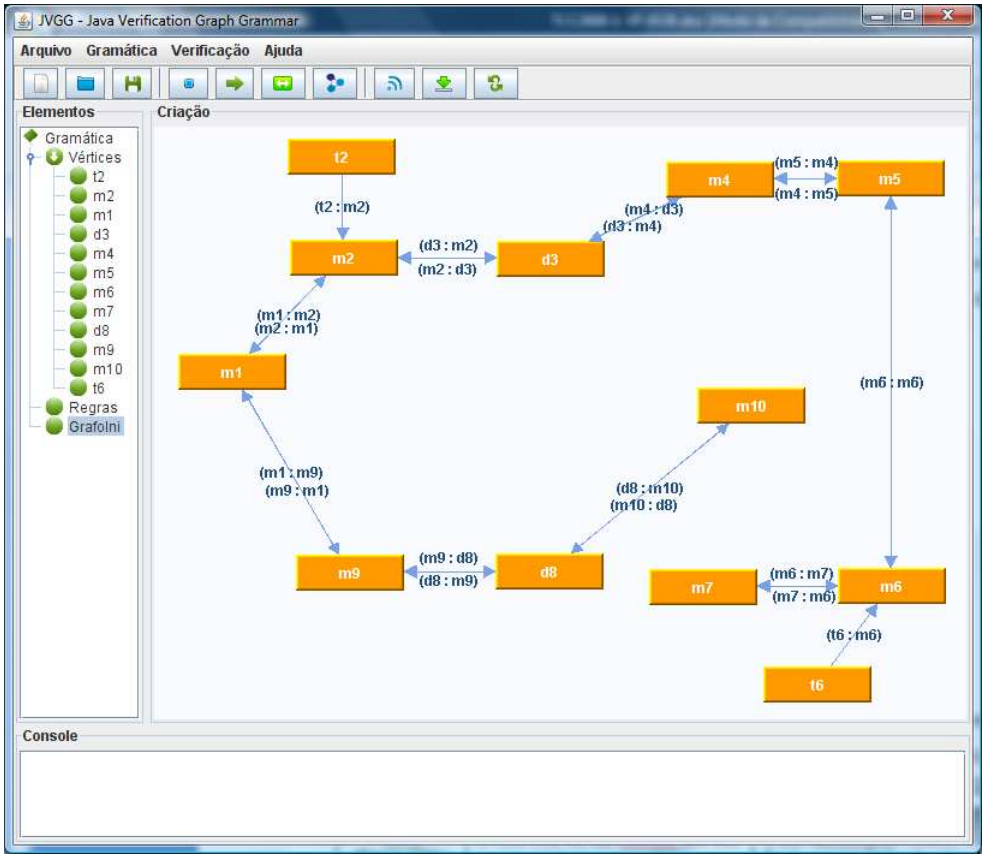


Figura 35 – Grafo inicial do exemplo ferrovia

Para iniciar a criação das regras é necessário que o usuário clique no botão “Criar regra” ou na opção “Regra” do menu “Gramática”. Executando uma das duas opções, é aberta uma janela para que o usuário informe o nome da regra e indique se ela é genérica ou não, como apresentado na Figura 36. Ao clicar no botão “Inserir”, a regra aparece na área de elementos, no nó regras, onde haverá outro nó com o nome da regra criada e dois nós filhos, chamados *L* e *R*, onde deverão ser inseridos os grafos referentes aos lados respectivos da regra. Para excluir uma regra é necessário clicar sobre o seu nome na área de elementos e pressionar a tecla “delete”.

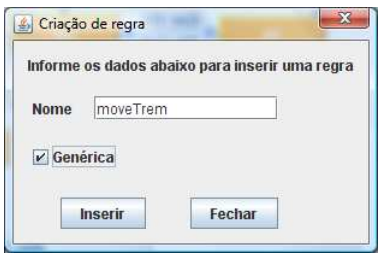


Figura 36 – Criação da regra `moveTrem`

Para inserir os grafos no lado *L* e *R* da regra criada é necessário que o usuário clique sobre o lado no qual deseja inserir os vértices e arestas, e realizar os mesmos procedimentos citados anteriormente para a criação de vértices e arestas, levando em conta que se a regra for genérica, o vértice deve ser uma variável, composta pela letra que identifica o nó e por um

identificador variável (*X*, *Y*, *Z*, *A*, *B*, *C* ou *D*). A regra criada é apresentada na Figura 37.

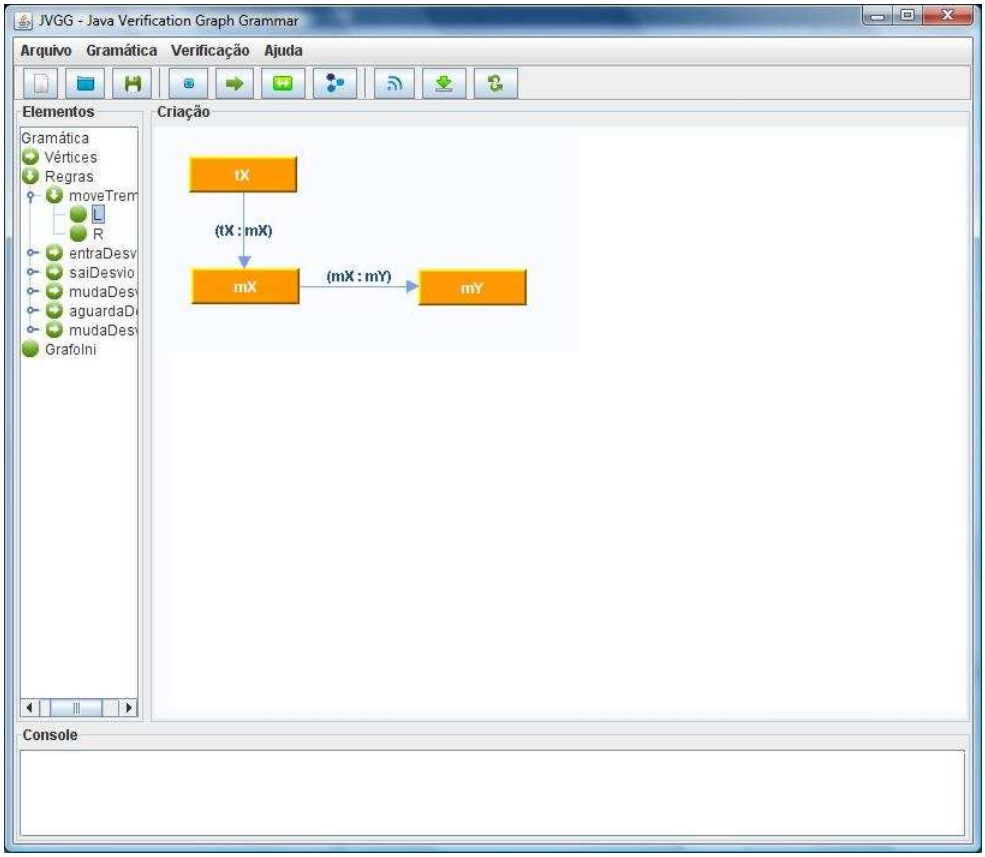


Figura 37 – Criação do lado L da regra `moveTrem`

Tendo o grafo inicial e o conjunto de regras, é possível iniciar a verificação de propriedades. Para verificar as propriedades, existem duas opções: verificar todas as propriedades de uma vez só ou verificar cada uma separadamente. A verificação de todas as propriedades de uma só vez acontece quando o usuário clica na opção “Todos” do menu “Verificação”, indicando quais as propriedades que deseja verificar. Após isso, é necessário que sejam preenchidas as informações referentes ao algoritmo selecionado, nas abas respectivas (Figura 38). Com as informações preenchidas (referente às propriedades selecionadas), o usuário clica no botão “Verificar” e o JVGG inicia a verificação das propriedades, emitindo os resultados da verificação no *console* da janela principal.

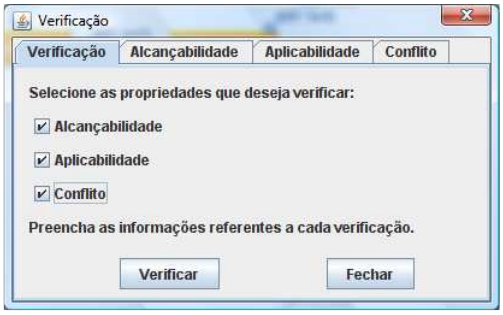


Figura 38 – Janela para a verificação de mais que uma propriedade em conjunto

Para verificar as propriedades separadamente, o usuário pode clicar no botão referente

à propriedade ou ir ao menu “Verificação” e escolher a propriedade desejada. Se o usuário escolher a verificação da alcançabilidade, o JVGG solicitará que seja informado o vértice origem e o vértice destino para fazer a verificação(Figura 39).

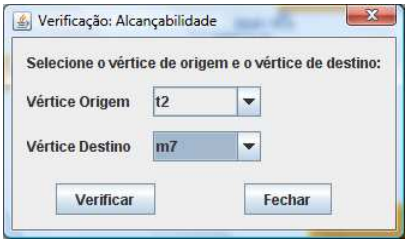


Figura 39 – Verificação da alcançabilidade entre o vértice `t2` e `m7`

Quando o usuário clicar no botão “Verificar”, a verificação será iniciada e o resultado da verificação (Quadro 16) poderá ser visto no *console* do JVGG. O resultado mostra para cada iteração o caminho encontrado pela execução do algoritmo de Dijkstra, a necessidade de adicionar ou remover algum vértice e aresta (aplicação de uma regra) e como ficou o grafo gerado depois da aplicação. Ao final das iterações indica se o vértice é ou não alcançável.

```
Alcançabilidade

Dijkstra = [(t2 : m2), (m2 : d3), (d3 : m4), (m4 : m5), (m5 : m6), (m6 : m7)]
remove -(t2 : m2)
adiciona +(t3 : d3)
([m1, m2, m4, m5, m6, m7, m9, m10, d3, d8, t6, t3], [(t6,m6), (m1,m2), (m2,m1),
(m2,d3), (t3,d3), (d3,m2), (d3,m4), (m4,d3), (m5,m4), (m4,m5), (m5,m6), (m6,m5),
(m6,m7), (m7,m6), (d8,m10), (m10,d8), (d8,m9), (m9,d8), (m9,m1), (m1,m9)])

Dijkstra = [(t3 : d3), (d3 : m4), (m4 : m5), (m5 : m6), (m6 : m7)]
remove -(t3 : d3)
adiciona +(t4 : m4)
([m1, m2, m4, m5, m6, m7, m9, m10, d3, d8, t6, t4], [(t6,m6), (m1,m2), (m2,m1),
(m2,d3), (d3,m2), (d3,m4), (m4,d3), (m5,m4), (m4,m5), (m5,m6), (m6,m5), (m6,m7),
(m7,m6), (d8,m10), (m10,d8), (d8,m9), (m9,d8), (m9,m1), (m1,m9), (t4,m4)])
Dijkstra = [(t4 : m4), (m4 : m5), (m5 : m6), (m6 : m7)]
remove -(t4 : m4)
adiciona +(t5 : m5)
([m1, m2, m4, m5, m6, m7, m9, m10, d3, d8, t6, t5], [(t6,m6), (m1,m2), (m2,m1),
(m2,d3), (d3,m2), (d3,m4), (m4,d3), (m5,m4), (m4,m5), (m5,m6), (m6,m5), (m6,m7),
(m7,m6), (d8,m10), (m10,d8), (d8,m9), (m9,d8), (m9,m1), (m1,m9), (t5,m5)])

Dijkstra = [(t5 : m5), (m5 : m6), (m6 : m7)]
remove -(t5 : m5)
adiciona +(t6 : m6)
([m1, m2, m4, m5, m6, m7, m9, m10, d3, d8, t6], [(t6,m6), (m1,m2), (m2,m1),
(m2,d3), (d3,m2), (d3,m4), (m4,d3), (m5,m4), (m4,m5), (m5,m6), (m6,m5), (m6,m7),
(m7,m6), (d8,m10), (m10,d8), (d8,m9), (m9,d8), (m9,m1), (m1,m9)])

Dijkstra = [(t6 : m6), (m6 : m7)]
remove -(t6 : m6)
adiciona +(t7 : m7)
([m1, m2, m4, m5, m6, m7, m9, m10, d3, d8, t7], [(m1,m2), (m2,m1), (m2,d3),
(d3,m2), (d3,m4), (m4,d3), (m5,m4), (m4,m5), (m5,m6), (m6,m5), (m6,m7), (m7,m6),
(d8,m10), (m10,d8), (d8,m9), (m9,d8), (m9,m1), (m1,m9), (t7,m7)])

É alcançável!
```

Quadro 16 – Resultado da verificação da alcançabilidade entre o vértice `t2` e `m7`

A Figura 40 mostra a janela que solicita ao usuário que informe qual a regra que deseja

verificar a aplicabilidade.

Figura 40 – Verificação da aplicabilidade da regra `entraDesvio`

Quando o usuário clicar no botão “Verificar”, o algoritmo é executado e pode-se ver o resultado (Quadro 17) da verificação no *console*. O resultado apresenta o nome da regra que está sendo verificada, quais são os vértices e arestas pertencentes ao lado *L* e *R* da regra, se a regra é genérica e, para cada possível tentativa de aplicação, mostra se é aplicável ou não. No caso da regra `entraDesvio`, ela é considerada aplicável porque na segunda tentativa foi informado que ela conseguiu ser aplicada.

```
Aplicabilidade

Nome da Regra - entraDesvio
L - ([tX, mX, dY], [(tX,mX), (mX,dY)])
R - ([tY, mX, dY], [(tY,dY), (mX,dY)])
isGeneric - true

([t2, m2, d1], [(t2,m2), (m2,d1)]) - ([t1, d1, m2], [(t1,d1), (m2,d1)])
Não é aplicável!

([t2, m2, d3], [(t2,m2), (m2,d3)]) - ([t3, d3, m2], [(t3,d3), (m2,d3)])
é Aplicavel!

([t6, m6, d5], [(t6,m6), (m6,d5)]) - ([t5, d5, m6], [(t5,d5), (m6,d5)])
Não é aplicável!

([t6, m6, d7], [(t6,m6), (m6,d7)]) - ([t7, d7, m6], [(t7,d7), (m6,d7)])
Não é aplicável!

([t3, m3, d2], [(t3,m3), (m3,d2)]) - ([t2, d2, m3], [(t2,d2), (m3,d2)])
Não é aplicável!

([t3, m3, d4], [(t3,m3), (m3,d4)]) - ([t4, d4, m3], [(t4,d4), (m3,d4)])
Não é aplicável!
```

Quadro 17 – Resultado da verificação da aplicabilidade da regra `entraDesvio`

Para identificar o conflito potencial, o JVGG solicita que sejam informadas duas regras. Para iniciar a verificação, o usuário deve pressionar o botão “Verificar” (Figura 41). No *console* aparece o resultado da verificação apenas indicando se está em conflito ou não (Quadro 18).

Figura 41 - Verificação do conflito potencial entre as regras `saiDesvio` e `mudaDesviod3`

```
Conflito potencial
Conflito potencial entre as regras saiDesvioFixo e mudaDesviod3
```

Quadro 18 – Resultado da verificação do conflito potencial entre as regras `saiDesvio` e `alteraDesvio`

3.4 RESULTADOS E DISCUSSÃO

Os resultados encontrados com o término do trabalho são satisfatórios, pois com a utilização do JVGG é possível criar gramáticas e verificar as propriedades de alcançabilidade, aplicabilidade e conflito potencial, portanto alcançando o objetivo inicial.

No Quadro 19 é apresentado o comparativo das características do JVGG com os trabalhos correlatos. Pode-se observar que o JVGG não contempla a criação / edição de vértices / arestas tipados ou com atributos. Este seria um próximo estágio para que se tenham gramáticas mais robustas. No quesito verificação, o JVGG pode ser considerado mais completo por ter opção para verificação de mais propriedades do que os outros trabalhos.

Outro fato interessante no comparativo de características é que todos os trabalhos utilizam a abordagem SPO. Uma justificativa deve-se a fácil representação / transformação, se comparada com a DPO, e a ser uma abordagem mais recente, que contempla itens que não são contemplados em outras abordagens, podendo citar a não restrição de aplicação de regras.

	AGG	**GrGen**	**GROOVE**	**JVGG**
criação / edição de regras	✓	✓	✓	✓
criação / edição do grafo inicial	✓	✓	✓	✓
criação / edição de vértices / arestas rotulados	×	×	✓	✓
criação / edição de vértices / arestas tipados	✓	✓	×	×
criação / edição de vértices / arestas com atributos	✓	✓	✓	×
verificação de conflito	✓	×	×	✓
verificação de aplicabilidade	✓	×	×	✓
verificação de alcançabilidade	×	×	×	✓
abordagem	SPO	SPO	SPO	SPO

Quadro 19 – Características do JVGG e trabalhos correlatos

4 CONCLUSÕES

No início do desenvolvimento do JVGG foram elencadas funcionalidades que ele deveria ter como: permitir criar vértices, arestas, grafo inicial e regras, para que a funcionalidade principal deste trabalho pudesse ser implementada, que é a verificação das propriedades de alcançabilidade, aplicabilidade e conflito potencial. Ao término pode-se observar que as funcionalidades foram criadas, atingindo os objetivos.

Uma dificuldade encontrada foi à busca de materiais didáticos que abordassem o formalismo da gramática de grafos, mas especificamente sobre verificação de propriedades. Para resolver este problema, foi necessária uma visita à biblioteca da Universidade Federal do Rio Grande do Sul que tem materiais base sobre gramática de grafos, uma vez que é uma das linhas de pesquisa desenvolvidas nesta instituição. Com base nestes materiais encontrados, foi possível decidir pela utilização da abordagem SPO em vez da DPO.

A escolha da abordagem SPO deu-se por não permitir a aplicação de regras em algumas situações. O motivo foi a facilidade de representação das regras e pela economia das transformações nos grafos e das estruturas que teriam que ser criadas. Outro fator levado em consideração é que grande parte dos sistemas que utilizam gramática de grafos são baseados na abordagem SPO para realizar a representação e transformação em grafos.

Houve também dificuldade em relação à verificação das propriedades (alcançabilidade, aplicabilidade e conflito). Não foi encontrado nenhum material didático que sugerisse a implementação dos algoritmos. Por esse motivo, as verificações encontradas no presente trabalho foram implementadas apenas utilizando a teoria das gramáticas de grafos.

A utilização das bibliotecas JGraph e JGraphT auxiliaram no desenvolvimento do JVGG, principalmente no que diz respeito a parte gráfica, que seria bem mais complexa de implementar caso não tivesse utilizado a JGraph. A JGraphT na parte de estrutura de dados facilitou a criação dos grafos, permitiu a utilização do algoritmo de Dijkstra e mostrou-se bastante completa para manipulação de grafos, podendo ser bastante útil no caso da implementação de algumas sugestões de extensão.

No entanto, existem limitações no JVGG como: o engessamento dos rótulos dos vértices, obrigando que o rótulo seja composto de uma letra e um número; a alcançabilidade pode ser apenas verificada no grafo inicial e; a ausência de verificação do conflito real.

4.1 EXTENSÕES

Existem pontos que podem ser agregados ou melhorados no JVGG. Como sugestão pode-se citar:

a) permitir transformações em grafos em tempo real utilizando a SPO, de modo que o usuário possa verificar como as transformações vão acontecendo durante a execução dos algoritmos;
b) adicionar novos algoritmos de verificação, como por exemplo a verificação do limite, ou seja, o tamanho do grafo máximo e mínimo que se pode ter em uma gramática de grafos;
c) adicionar ao JVGG as gramáticas de grafos orientadas a objetos;
d) permitir a criação de grafos tipados e grafos com atributos;
e) permitir a criação de regras com condições negativas;
f) melhorar a interface, permitindo abrir e salvar arquivos com gramáticas de grafo.

REFERÊNCIAS BIBLIOGRÁFICAS

BENSON, David. **JGraph and JGraph Layout Pro**: user manual. Northampton, 2009. Disponível em: <http://www.jgraph.com/pub/jgraphmanual.pdf>. Acesso em: 28 maio 2009.

CORRADINI, Andrea et al. Algebraic approaches to graph transformation. Part I: basic concepts and double pushout approach. In: ROZENBERG, Grzegorz. **Handbook of graph grammars and computing by graph transformation:** foundations. River Edge: World Scientific Publishing Co., 1997. p. 163-245.

DÉHARBE, David et al. Introdução a métodos formais: especificação, semântica e verificação de sistemas concorrentes. **Revista de Informática Teórica e Aplicada**, Porto Alegre, v. vii, n. 1, p. 7-48, set. 2000.

DOTTI, Fernando L.; PASINI, Fábio; SANTOS, Osmar M. Uma metodologia para verificação de sistemas parciais modelados na gramática de grafos baseada em objetos. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, 18., 2004, Brasília. **Anais...** Porto Alegre: Sociedade Brasileira de Computação, 2004. p. 86-101.

EHRIG, Hartmut et al. Algebraic approaches to graph transformation. Part II: single pushout approach and comparison with double pushout approach. In: ROZENBERG, Grzegorz. **Handbook of graph grammars and computing by graph transformation:** foundations. River Edge: World Scientific Publishing Co., 1997. p. 247-312.

FERREIRA, Ana P. L.; RIBEIRO, Leila. Programação orientada a objeto com grafos. In: BREITMAN, Karin; ANIDO, Ricardo. **Atualizações em informática**. Rio de Janeiro: Editora PUC Rio, 2006. p. 387-452.

GEIß et al. GrGen: a fast SPO-based graph rewriting tool. In: GRAPH TRANSFORMATION: INTERNATIONAL CONFERENCE, 3., 2006, Rio Grande do Norte. **Proceedings...** New York: Springer-Verlag, 2006. p. 383-397.

_____. **GrGen.NET**. Karlsruhe, 2008. Disponível em: <http://www.info.uni-karlsruhe.de/software/grgen>. Acesso em: 17 set. 2008.

LORETO, Aline B.; TOSCANI, Laira V.; RIBEIRO, Leila. Decidibilidade e tratabilidade de problemas em gramática de grafos orientada a objetos. In: WORKSHOP DE MÉTODOS FORMAIS, 4., 2001, Rio de Janeiro. **Anais...** Rio de Janeiro: SBC, 2001. p. 129-140.

NAVEH, Barak. **Welcome to JGraph**: a free Java graph library. [S.l.], 2009. Disponível em: < http://www.jgrapht.org/>. Acesso em: 28 maio 2009.

RENSINK, Arend et al. **GROOVE - GRaphs for Object-Oriented VErification:** documentation. Enschede, 2009. Disponível em: <http://groove.cs.utwente.nl/documentation/>. Acesso em: 16 jan. 2009.

RIBEIRO, Leila. Métodos formais de especificação: gramáticas de grafos. In: ESCOLA DE INFORMÁTICA DA SBC-SUL, 8., 2000, Ijuí. **Anais…** Porto Alegre: Editora da UFRGS, 2000. p. 1-33.

RUSSO, Cristina C. **Gramática de grafos para modelagem de crescimento do tumor glioblastoma multiforme**. 2003. 51 f. Projeto de Diplomação (Bacharelado em Ciências da Computação) – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

SULZBACH, Sirlei I. **Um estudo da computabilidade e da tratabilidade dos problemas de verificação em gramáticas de grafos**. 2002. 40 f. Trabalho Individual I (Mestrado em Computação) – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

SZWARCFITER, Jayme L. **Grafos e algoritmos computacionais**. Rio de Janeiro: Campus, 1984.

TAENTZER, Gabi et al. **A brief description of AGG**. Berlin, 2008. Disponível em: <http://tfs.cs.tu-berlin.de/agg/docu.html>. Acesso em: 31 ago. 2008.

APÊNDICE A – Principais atributos e métodos das classes do pacote `model`

No Quadro 20 é apresentado o detalhamento dos principais atributos e métodos da classe `Grammar.java`.

Classe: `Grammar.java`	
Atributo	**Descrição**
`vertex`	Conjunto de vértices (rótulos) da gramática.
`rule`	Conjunto de regras da gramática.
`iGraph`	Grafo inicial da gramática.
Método	**Descrição**
`verificationReachability`	Informa os parâmetros para a classe `Reachability` poder realizar a verificação da alcançabilidade.
`verificationApplicability`	Informa os parâmetros para a classe `Applicability` poder realizar a verificação da aplicabilidade.
`verificationDeadlock`	Informa os parâmetros para a classe `Deadlock` poder realizar a verificação do conflito potencial.

Quadro 20 – Detalhamento dos principais atributos e métodos da classe `Grammar.java`

No Quadro 21 é apresentado o detalhamento dos principais atributos e métodos da classe `InitialGraph.java`.

Classe: `InitialGraph.java`	
Atributo	**Descrição**
`initial`	Grafo inicial da gramática do tipo `Graph<String, DefaultEdge>` pertencente a biblioteca JGraphT.
Método	**Descrição**
`isContain`	Verifica se o conjunto de vértices e o conjunto de arestas passado como parâmetro estão contidos no grafo inicial.

Quadro 21 – Detalhamento do principal atributo e método da classe `InitialGraph.java`

No Quadro 22 é apresentado o detalhamento dos principais atributos e métodos da classe `Rule.java`.

Classe: `Rule.java`	
Atributo	**Descrição**
`left`	Lado *L* da regra.
`right`	Lado *R* da regra.
`isGeneric`	Indica se a regra é genérica ou não.
`isApplicability`	Indica se a regra é aplicável ou não.

Quadro 22 – Detalhamento dos principais atributos da classe `Rule.java`

No Quadro 23 é apresentado o detalhamento dos principais atributos e métodos da classe `Verification.java`.

Classe: `Verification.java`	
Atributo	**Descrição**
`vertexSource`	Armazena o vértice origem.
`vertexTarget`	Armazena o vértice destino.
Método	**Descrição**
`generateRuleAuxiliar`	Gera a regra auxiliar.
`generateRuleL`	Gera o lado *L* de uma regra auxiliar.
`generateRuleR`	Gera o lado *R* de uma regra auxiliar.
`chooseRule`	Escolhe uma regra para aplicar no grafo inicial.

Quadro 23 – Detalhamento dos principais atributos e métodos da classe `Verification.java`

No Quadro 24 é apresentado o detalhamento dos principais atributos e métodos da classe `Reachability.java`.

Classe: `Reachability.java`	
Atributo	**Descrição**
`vertexVerificator`	Vértice auxiliar.
`ig`	Grafo inicial auxiliar.
Método	**Descrição**
`verification`	Método principal da classe que realiza a verificação da alcançabilidade.
`dijkstraWay`	Cria um caminho entre o vértice origem e o vértice destino executando o algoritmo de Dijkstra. Para executar Dijkstra utiliza a classe `DijkstraShortestPath` da biblioteca JGraphT.
`findRule`	Verifica se determinada regra existe na lista de regras da gramática.
`changeState`	Muda o estado do grafo inicial.

Quadro 24 – Detalhamento dos principais atributos e métodos da classe `Reachability.java`

No Quadro 25 é apresentado o detalhamento dos principais atributos e métodos da classe `Applicability.java`.

Classe: `Applicability.java`	
Atributo	**Descrição**
`verificationRule`	Vértice auxiliar.
Método	**Descrição**
`verification`	Método principal da classe que realiza a verificação da aplicabilidade.
`isContainL`	Verifica se o lado *L* de uma regra está contido no grafo inicial.
`isContainLInR`	Verifica se o lado *L* de uma regra está contido no lado *R* de outra regra.

Quadro 25 – Detalhamento dos principais atributos e métodos da classe `Applicability.java`

No Quadro 26 é apresentado o detalhamento dos principais atributos e métodos da classe `Deadlock.java`.

Classe: `Deadlock.java`	
Atributo	**Descrição**
`rule1`	Regra para comparar com a `rule2`.
`rule2`	Regra para comparar com a `rule1`.
Método	**Descrição**
`verification`	Método principal da classe que realiza a verificação do conflito potencial.
`changeState`	Muda o estado do grafo inicial.

Quadro 26 – Detalhamento dos principais atributos e métodos da classe `Deadlock.java`