DEMÓSTENES SANTOS DE SENA

AGRAPHS: DEFINIÇÃO, IMPLEMENTAÇÃO E SUAS FERRAMENTAS

Natal

Julho de 2006

DEMÓSTENES SANTOS DE SENA

AGRAPHS: DEFINIÇÃO, IMPLEMENTAÇÃO E SUAS FERRAMENTAS

Trabalho apresentado ao curso de mestrado em Sistemas e Computação como requisito parcial à obtenção do grau de mestre em Sistemas e Computação.

Universidade Federal do Rio Grande do Norte.

Orientadora: Profa. Dra. Anamaria M. Moreira
Co-orientador: Prof. Dr. David B. P. Deharbe

Natal, 2006

Divisão de Serviços Técnicos

Catalogação da Publicação na Fonte. UFRN / Biblioteca Central Zila Mamede

Sena, Demóstenes Santos de.
AGraphs : definição, implementação e suas ferramentas / Demóstenes Santos de Sena. – Natal, RN, 2006.
109 f. : il.

Orientadora : Anamaria M. Moreira.
Co-orientador : David B. P. Deharbe.

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

1. Linguagens de programação (Informática) – Dissertação. 2. Estrutura de dados – Dissertação. 3. Representação e transferência de dados - Dissertação. I. Moreira, Anamaria M. II. Deharbe, David B. P. III. Título.

RN/UF/BCZM CDU 004.43(043.3)

DEMÓSTENES SANTOS DE SENA

AGRAPHS: DEFINIÇÃO, IMPLEMENTAÇÃO E SUAS FERRAMENTAS

Esta dissertação foi julgada adequada para a obtenção do título de Mestre em Sistemas e Computação (MSc.) área de concentração em Engenharia de Software, e aprovada em sua forma final pelo Programa de Pós-Graduação em Sistemas e Computação da Universidade Federal do Rio Grande do Norte.

Anamaria Martins Moreira
Orientadora

Regivan Nunes Santiago
Coordenador do Programa

Banca Examinadora:

Profa. Dra. Anamaria M. Moreira
Universidade Federal do Rio Grande do Norte

Prof. Dr. David B. P. Déharbe
Universidade Federal do Rio Grande do Norte

Prof. Dr. Martin A. Musicante
Universidade Federal do Rio Grande do Norte

Prof. Dr. Roberto Ierusalimschy
Pontifícia Universidade Católica do Rio de Janeiro

AGRADECIMENTOS

Agradeço inicialmente, às pessoas que me concederam o dom da vida e me propiciaram um ambiente para o meu desenvolvimento pessoal. Aos meus pais toda a gratidão pelos ensinamentos, exemplos e carinho a mim concedidos.

Agradeço a minha orientadora, Anamaria Martins Moreira, com toda a sua paciência em organizar as reuniões e desprendimento ao utilizar o seu escasso tempo para a minha formação, desde os meus tempos de graduação.

Aos meus companheiros de mestrado, por compartilhar do ambiente do "laboratório de mestrado, frio e impessoal"[Camila04], e todos os obstáculos superados. Agradeço especialmente à Camila, que esteve sempre ao meu lado, e colaborou para a revisão desta dissertação e à Macilon com suas colaborações técnicas e sua personalidade hilariante.

Agradeço aos professores Martin Musicante e Roberto Ierusalimschy, que participaram da banca e fizeram diversas contribuições ao trabalho, juntamente com o prof. David Déharbe, a quem agradeço especialmente pela sua co-orientação.

Finalmente, agradeço também a todos que participaram da minha vida durante o período em que permaneci no DIMAp, professores, alunos, secretários, e demais funcionários.

"...terminado o jogo, o peão e o rei voltam para a mesma caixa."
autor desconhecido

Resumo

Programas manipulam informações. Entretanto, as informações são essencialmente abstratas e precisam ser representadas, normalmente por estruturas de dados, permitindo a sua manipulação.

Esse trabalho apresenta os AGraphs, um formato de representação e transferência de dados que usa grafos direcionados tipados que permitem a simulação de hiperarestas e de grafos hierárquicos. Associado ao formato AGraphs existe uma biblioteca de manipulação com uma interface simples de ser usada, mas dependente da linguagem. O formato AGraphs foi usado de maneira *ad-hoc* como formato de representação em algumas ferramentas desenvolvidas na UFRN, e, com a possibilidade de uso em outras aplicações, tornou-se necessária uma definição precisa e o desenvolvimento de ferramentas de suporte. A definição precisa e as ferramentas foram desenvolvidas e são descritas neste trabalho.

Finalizando, comparações do formato AGraphs com outros formatos de representação e transferência de dados (ATerms, GDL, GraphML, GraX, GXL e XML) são realizadas. O principal objetivo destas comparações é obter as características significantes e em que conceitos o formato e a biblioteca AGraphs deve amadurecer.

Palavras-chave: estruturas de dados; linguagens de programação; formatos de representação e transferência de dados;

Abstract

Programs manipulate information. However, information is abstract in nature and needs to be represented, usually by data structures, making it possible to be manipulated.

This work presents the AGraphs, a representation and exchange format of the data that uses typed directed graphs with a simulation of hyperedges and hierarchical graphs. Associated to the AGraphs format there is a manipulation library with a simple programming interface, tailored to the language being represented. The AGraphs format in ad-hoc manner was used as representation format in tools developed at UFRN, and, to make it more usable in other tools, an accurate description and the development of support tools was necessary. These accurate description and tools have been developed and are described in this work.

This work compares the AGraphs format with other representation and exchange formats (e.g ATerms, GDL, GraphML, GraX, GXL and XML). The main objective this comparison is to capture important characteristics and where the AGraphs concepts can still evolve.

Keywords: data structures; programming languages; representation and exchange formats:

Lista de Figuras

3.1 Arquitetura do FERUS. 27
3.2 Exemplo de AGraph para uma especificação 31
3.3 Funções da biblioteca AGraphs fornecidas aos usuários 32
3.4 Diagrama de atividades para a função de criação de um nó 33
3.5 Diagrama UML de classes para esquemas AGraph 35
3.6 Estrutura para o nó definição de unidade CASL. 38
3.7 Exemplos de declarações de construtores de nós. 40
3.8 Exemplo do formato textual AGraph 41
3.9 Relação entre a organização do formato textual AGraph e do formato codificado (BGF) 42
3.10 Exemplo de transferência de informações usando AGraphs. 43

4.1 Exemplo de definição SDF2 da sintaxe de uma linguagem de especificação fictícia. 52
4.2 Processo de construção da biblioteca AGraph a partir da sintaxe concreta. 53
4.3 Exemplo de mapeamento da definição da sintaxe, via definições abstratas, para a geração do código. 54
4.4 Exemplo de mapeamento de uma regra SDF2, da figura 4.1, para um esquema AGraph. 56

B.1 *Diagrama UML de Casos de Uso* contendo as funções das bibliotecas AGraphs. 94
B.2 *Diagrama UML de Classes* descrevendo os elementos que compõem os esquemas AGraphs. 95
B.3 *Diagrama UML de Estados* do AGraph e de *Atividades* da função de inicialização da biblioteca. 96
B.4 *Diagrama UML de Atividades* da função de criação de nós (`make`). . . . 97
B.5 *Diagramas UML de Atividade* das funções acessores (`set` e `get`). 98
B.6 *Diagrama UML de Atividades* da função de leitura de uma unidade (`read`). 99
B.7 *Diagrama UML de Atividades* da função de escrita de uma unidade (`write`). 100
B.8 *Diagrama UML de Atividades* da função de compactação (`compress`). . 101
B.9 *Diagrama UML de Atividades* da função de descompactação (`decompress`). 102
B.10 *Diagramas UML de Sequência* das funções de inicialização (`init`) e criação de um nó (`make`). 103
B.11 *Diagramas UML de Sequência* das funções acessores (`set` e `get`). . . . 104
B.12 *Diagrama UML de Sequência* da função de leitura de uma unidade (`read`).105

B.13 *Diagrama UML de Sequência* da função de escrita de uma unidade (`write`). 106
B.14 *Diagrama UML de Sequência* da função de compactação de uma unidade (`compress`). 107
B.15 *Diagrama UML de Sequência* da função de descompactação de uma unidade (`decompress`). 108

Lista de Tabelas

2.1 Propriedades características e alguns formatos [13]. 21
2.2 Propriedades características e alguns formatos [13] (cont.). 22
2.3 Classificação de alguns formatos de transferência de acordo com seus esquemas [13]. 23
2.4 Vantages e desvantagens relacionadas aos padrões de esquemas [13]. . . . 24
2.5 Padrões de formatos X Características de um formato de transferência padrão [13]. 25

4.1 Legenda para as tabelas 4.2 e 4.3. 61
4.2 Análise do método de codificação binária 62
4.3 Análise do método de codificação binária (cont.) 63

Sumário

1 Introdução **13**
1.1 Estrutura da dissertação . . . 14

2 Representação e transferência de dados **15**
2.1 Propriedades características dos formatos de transferência . . . 17
2.1.1 Sintaxe abstrata . . . 17
2.1.2 Níveis de abstração . . . 17
2.1.3 Codificação . . . 18
2.1.4 Mecanismo de transferência . . . 18
2.1.5 Tipos de esquemas . . . 19
2.1.6 Exemplos de classificação . . . 19
2.2 Propriedades dos esquemas . . . 23
2.2.1 Definição do esquema . . . 23
2.2.2 Localização do esquema . . . 23
2.2.3 Exemplos de classificação . . . 23
2.3 Qualidades de formatos de transferência . . . 23

3 AGraphs **26**
3.1 Aplicação . . . 26
3.1.1 FERUS . . . 26
3.2 Especificação . . . 28
3.2.1 Estrutura de dados AGraph . . . 29
3.2.2 Interface fornecida pela biblioteca AGraphs . . . 31
3.2.3 Esquemas AGraphs . . . 35
3.3 Implementação . . . 37
3.3.1 Implementação dos nós do grafo . . . 38
3.3.2 Dependência de unidades . . . 39
3.3.3 Interface da biblioteca AGraph . . . 39
3.3.4 Formatos persistentes AGraph . . . 40
3.4 AGraph como formato de transferência . . . 42
3.4.1 Classificação dos AGraphs . . . 43

4 Ferramentas **45**
4.1 Gerador automático . . . 45
4.1.1 Geração customizada . . . 45
4.1.2 Geração a partir da descrição da sintaxe . . . 51
4.1.3 Mapeamento automático . . . 54
4.2 Codificação Binária do formato AGraph . . . 58
4.2.1 Descrição do método . . . 59

4.2.2 API da compactação/descompactação 60
4.2.3 Resultados 61

5 Trabalhos relacionados **65**
5.1 *Annotated Terms* - ATerms 65
5.1.1 Operações 66
5.1.2 Implementação 68
5.1.3 ATerms x AGraphs 69
5.2 Graph Description Language - GDL 70
5.3 GraphML 71
5.4 GraX 73
5.5 Graph eXchange Language - GXL 74
5.6 XML 75

6 Conclusão **76**

A Especificação formal - *B* **82**
A.1 Especificação estática 82
A.2 Especificação dinâmica 86

B Especificação gráfica - Diagramas UML **94**

C Linguagem de entrada para o gerador **109**

Capítulo 1

Introdução

Programas manipulam informações, sendo que estas são abstratas por natureza e precisam ser representadas, normalmente usando estruturas de dados, permitindo a sua manipulação.

Além da representação, algumas aplicações compartilham informações com outras ferramentas. Este compartilhamento é motivado principalmente pela cooperação entre ferramentas que desempenham tarefas diferentes. Tradicionalmente, cada software define seus formatos de representação de maneira ad-hoc e fechada. Devido aos problemas que a abordagem tradicional de representar e transferir dados possui, uma nova tendência surge, esta nova tendência visa a padronização, abertura e interoperabilidade da representação e da transferência dos dados. Consequentemente, padrões de estruturas de representação e de métodos de transferência foram desenvolvidos, como por exemplo: ATerms [25], GraphML [4], GraX [10], GXL [28] e XML [7].

Neste trabalho, descreve-se AGraph, um formato de representação e transferência de dados que usa grafos tipados, direcionados, que utiliza uma simulação de hiperarestas e de grafos hierárquicos. Associado ao formato AGraph existe uma biblioteca de funções que possibilita a manipulação da sua estrutura de dados.

Inicialmente, AGraphs foi usado nas duas instâncias da ferramenta FERUS [11] (FERUS-CASL [15] e FERUS-ELAN [18]) e no *model checker CV* [9] para uma sub-linguagem de VHDL.

O formato AGraphs e a sua biblioteca podem ser aplicados a várias linguagens de programação. Com o uso do formato e da biblioteca AGraphs torna-se necessária a definição precisa dos conceitos associados ao formato AGraphs e o desenvolvimento de ferramentas de apoio, sendo estes os principais objetivos deste trabalho.

Neste trabalho, os AGraphs (formato e sua biblioteca) são definidos usando três categorias de descrições, denominadas: especificação textual, formal e a gráfica.

A especificação textual é fornecida na seção 3.2. Esta especificação descreve a estrutura de dados, a biblioteca de manipulação, e as informações fornecidas pelo esquema de um AGraph textualmente. A especificação textual é auxiliada por trechos da especificação formal e diagramas da especificação gráfica.

Na especificação formal usa-se a linguagem de especificação B, construindo *máquinas* para decrever a estrutura de dados, *estática* e *dinamicamente*, dos AGraphs.

A descrição estática da estrutura de dados é a descrição das relações dos elementos que compõem os AGraphs, ou seja, a sua organização. Por exemplo a relação entre nós de diferentes tipos.

A descrição dinâmica da estrutura de dados é a organização comportamental dos

AGraphs aplicados às funções de manipulação. Por exemplo, o comportamento da estrutura de dados a inserção de um novo nó ao AGraph.

UML é a linguagem usada na descrição gráfica dos AGraphs. Utilizam-se vários diagramas UML para obter os objetivos de (1) expor o processo das funções que compõem as bibliotecas AGraphs (diagramas de *casos de uso*, de *atividades* e de *sequências*) e (2) descrever a formação do *esquema* dos grafos representados pelo formato AGraph (diagrama de *classes* e *objetos*).

Como apoio à aplicação do formato AGraphs, é necessário o provimento de ferramentas de suporte. Neste trabalho, foram desenvolvidas duas ferramentas de suporte: (a) um gerador automático de AGraphs, e (b) um método de compactação da representação.

A ferramenta de geração é necessária devido à dependência que o formato AGraphs possui com relação à linguagem representada. O método de compactação foi desenvolvido para amenizar o crescimento do tamanho da representação textual, criando uma representação binária compactada.

1.1 Estrutura da dissertação

Este trabalho inicia com a contextualização (capítulo 2), fornecendo a motivação do uso dos formatos de representação e transferência de dados e suas propriedades significativas e alguns conceitos importantes para o desenvolvimento deste trabalho.

No capítulo 3, o formato AGraph e a sua biblioteca de manipulação são descritos. Inicia-se com (1) a exposição do FERUS, uma das ferramentas que forneceu a motivação do desenvolvimento de AGraph como formato de representação de dados (seção 3.1), seguida (2) pela descrição textual do formato (seção 3.2), e (3) pela descrição da estrutura de dados AGraphs e da sua biblioteca de manipulação na seção 3.3, e, finalizado com (4) a descrição de proposta de uso de AGraph como formato de transferência de dados, fornecendo as principais características do problema e de como utilizar o formato AGraph nesta perspectiva de aplicação (seção 3.4).

No capítulo 4, as ferramentas de suporte desenvolvidas são descritas. Na seção 4.1 é fornecida a descrição do gerador automático de bibliotecas AGraphs e as suas duas modalidades de geração: (1) geração *customizada* (seção 4.1.1), e, (2) geração a partir da sintaxe (seção 4.1.2). Na seção 4.2, o método de compactação, o formato binário BGF, a `API`, e uma comparação entre formato BGF e o *.gzip* são fornecidos.

Vários formatos de representação e transferência de dados existem na literatura. No capítulo 5, alguns formatos são apresentados e comparados com o formato AGraphs. O principal objetivo da comparação é verificar as caraterísticas significantes e em que o formato AGraphs deverá amadurecer.

E finalmente, no capítulo 6, as conclusões obtidas das atividades realizadas neste trabalho e os trabalhos futuros são fornecidos.

Nos anexos, a especificação formal completa em B (apêndice A), todos os diagramas UML construídos (apêndice B), e a sintaxe da linguagem de entrada do gerador automático de bibliotecas AGraphs na modalidade *customizada* (apêndice C) são fornecidos.

Capítulo 2

Representação e transferência de dados

Programas manipulam dados. Entretanto, os dados são essencialmente abstratos e precisam ser representados por estruturas de dados, permitindo a sua manipulação.

Listas, árvores e os grafos são as estruturas de dados mais utilizadas para a representação de dados. Esta preferência ocorre por dois principais motivos: (1) são estruturas complexas, o que acarreta em uma maior abrangência dos dados possíveis de serem representados, e, (2) possuem algoritmos de manipulação simples com bons desempenhos.

Além da representação, em algumas aplicações o compartilhamento dos dados entre ferramentas é necessário, motivado principalmente pela cooperação entre ferramentas que desempenham tarefas diferentes e/ou que possuem funções complementares. Como exemplo, existem ambientes integrados de engenharia reversa ou ferramentas *CASE* (e.g. CaseStudio, ErWin, Visio e Together).

A definição da estrutura que representa internamente os dados e de como a mesma é disponibilizada a terceiros (transferência dos dados) depende de diversos fatores. Tradicionalmente, nas transferências de dados, cada software define seus formatos de representação de maneira ad-hoc e fechada. Assim, a descrição do formato de representação e o próprio dado mantido estão embutidas, dificultando, por exemplo, a manutenabilidade do software.

Devido aos problemas que a abordagem tradicional de representar e transferir dados possui, uma nova tendência surgiu. A nova tendência visa a padronização, abertura e interoperabilidade da representação em memória e da transferência de dados. Consequentemente, padrões de estruturas de representação e de métodos de transferência foram desenvolvidos. Dificilmente uma técnica abrange todos os requisitos de software de modo geral. Assim, estes padrões e métodos investem em perspectivas diversas, possibilitanto uma grande abrangência de aplicações exploradas.

Vários trabalhos foram desenvolvidos com o intuito de classificar e comparar as diversas estruturas e os métodos de transferência de dados, tendo dois principais objetivos: definir padrões e apoiar escolhas. Um desses trabalhos [13] é base para este capítulo, e, no contexto do trabalho, este é bastante referenciado.

Na transferência de dados entre ferramentas existem três principais técnicas, usando: (1) uma representação intermediária (*IR - Intermediate Representations*), (2) uma API (*Application Programming Interface*), e, (3) formatos de transferência. Estas técnicas serão descritas a seguir.

Representação intermediária A representação intermediária é uma técnica de compartilhamento de dados usada há bastante tempo. Inicialmente, esta técnica foi desenvolvida,

e ainda é utilizada, no compartilhamento de dados entre os módulos de um compilador.

No contexto dos compiladores, os dados são analisados e representados em uma estrutura de dados, antes de realizar a geração do código de máquina. Esta estrutura é a representação intermediária e pode ser usada fora do contexto dos compiladores.

Vários problemas surgiram do uso da representação intermediária fora do contexto de compiladores: (1) não existe uma representação intermediária padrão, o que implica que para cada aplicação, desenvolva-se uma representação intermediária que melhor se adapta a sua aplicação, (2) as representações intermediárias não são independentes das linguagens, para cada linguagem representada uma nova representação intermediária é necessária que restringe o domínio de aplicações, e, (3) os dados representados possuem abstração baixa (e.g. expressões de linguagens), também restringindo o seu uso do ponto de vista do domínio de atuação dos dados representados.

API - *Application Programming Interface* API é uma técnica de transferência de dados em que uma aplicação disponibiliza os dados através de um conjunto de funções remotas, permitindo o acesso dinâmico.

Existem três principais modos de API, usando: (1) bibliotecas, (2) protocolo de comunicação ou (3) hibrído, que une as características dos dois primeiros.

No modo *biblioteca*, a aplicação possui uma versão encapsulada em outra aplicação, permitindo o acesso aos dados pela execução de funções na versão encapsulada. A principal vantagem é o acesso direto aos dados, não precisando de intermediadores que aumentam a complexidade do processo de transferência, e a desvantagem é a implementação: algumas aplicações não são organizadas como uma biblioteca, o que dificulta o uso deste modo de API.

Protocolo de comunicação é o outro modo de transferência de dados com API. Este modo é construído sobre uma arquitetura de comunicação (e.g. *peer-to-peer* ou *cliente-servidor*). API é composta por um conjunto de operações sobre os dados de modo remoto, em uma arquitetura de comunicação. A principal vantagem é a compatibilidade entre plataformas diferentes. A dificuldade em garantir a sincronização entre os dados é a principal desvantagem dessa categoria de API.

Formatos de transferência A utilização de formatos de transferência é uma proposta de solução para a transferência de dados entre ferramentas, onde os dados são transmitidos por um arquivo em um formato aberto, legível e independente da linguagem representada. A escolha ou definição do formato é um acordo entre as possíveis ferramentas participantes da integração ou uma adequação à proposta a ser desenvolvida. Normalmente, um formato de transferência usa um arquivo como objeto da transferência, sendo diferente da representação intermediária por ser independente da linguagem de programação usada no software.

A principal vantagem no uso dos formatos com relação às outras técnicas de transferência é a portabilidade e a facilidade de análise. Um formato de transferência deve ser legível e independente da linguagem.

O *overhead* de processamento gerado pelo armazenamento e recuperação é a principal desvantagem desta técnica. Para construir o arquivo em um formato é necessário a análise da estrutura de dados em memória, tal estrutura mantém os dados na ferramenta. Para recuperar os dados é necessária a análise do arquivo em um formato para construir a estrutura de dados em memória.

Os formatos são definidos pela sua sintaxe e semântica. A sintaxe é a maneira como os dados são fornecidas. A semântica define os dados que são armazenados em cada construção no formato. A definição da semântica é denominado *esquema* do formato.

Nesta seção, estuda-se mais aprofundadamente os formatos de transferência, iniciando na seção 2.1, onde descreve-se as propriedades que caracterizam um formato de transferência segundo [13]. A classificação dos esquemas com relação à sua definição, *explícita* ou *implícita*, e as respectivas vantagens e desvantagens são descritas na seção 2.2. Finalizando, na seção 2.3, as qualidades desejáveis em um formato de transferência são apresentadas.

2.1 Propriedades características dos formatos de transferência

Ao longo do tempo, vários formatos foram desenvolvidos para aplicações específicas ou com intuito de tornarem-se padrões na área. Em [13] foram definidas algumas características que permitem a classificação de uma diversidade de formatos. As principais características são revisadas a seguir.

2.1.1 Sintaxe abstrata

A estrutura de dados utilizada na representação dos dados, denominada *sintaxe abstrata*, é uma importante característica nos formatos. Define-se três principais categorias:

1. **Dados estruturados** Os dados são representados em uma estrutura própria. Esta estrutura não é árvore e nem grafo, normalmente é simples e formada por elementos atômicos (e.g. *tags*).

2. **Árvores** Os dados são representados usando somente árvores.

3. **Grafos** Os dados são representados usando grafos, incluindo árvores. Este modo de estruturação possui as seguintes características:

   - Tipado: existe uma classificação para as arestas e os nós.
   - Atribuído: os elementos do grafo possuem valores associados (atribuição), os quais são valores de natureza primitiva (e.g. conjunto de caracteres e inteiro) ou estruturada (e.g. listas de inteiros).
   - Com Herança: os elementos do grafo podem herdar propriedades (atributos) de outros elementos do mesmo.
   - Hierárquico: característica associada aos grafos que possuem grafos aninhados como atributos dos seus elementos.

2.1.2 Níveis de abstração

Os dados representados por um formato são classificados em relação aos seus níveis de detalhes na representação, denominado nível de abstração. Dependendo do nível de abstração do dado mantido, um formato de transferência é classificado como:

1. **Baixo** O formato com nível de abstração baixo representa expressões em uma linguagem. Neste nível de abstração, usa-se árvores (**AST** - *Abstract Syntax Tree*) ou grafos (**ASG** - *Abstract Syntax Graph*) para manter os dados.

2. **Médio** O formato com este nível de abstração representa os dados relativos aos procedimentos e suas relações em uma linguagem.

3. **Alto** O formato com este nível de abstração representa as relações entre os módulos, classes e pacotes em uma linguagem.

2.1.3 Codificação

Segundo o tipo de codificação, temos a seguinte classificação:

1. **Textual** Os dados são representados em texto e estruturados usando *tags* ou *labels*. Normalmente, este formato é legível, permitindo a manipulação por editores de texto simples. Como consequência dessa legibilidade, os arquivos neste tipo de formato de transferência são ineficientes em espaço de representação.

2. **Binária** Na codificação binária, os dados são: (a) compactados, convertidos da representação textual para a binária, ou, (b) mapeados diretamente da estrutura interna para a representação binária. Frequentemente, o tamanho da representação na codificação binária é menor que na codificação textual. Entretanto, quando a compactação e a descompactação são necessárias, o esforço para realizá-las pode comprometer um melhor desempenho na transferência.

2.1.4 Mecanismo de transferência

O mecanismo de transferência classifica o canal de comunicação entre as ferramentas. Esta característica é fracamente relacionada ao formato em si. As modalidades do mecanismo de transferência são:

1. **Arquivo** Método antigo para a transferência de dados. Atualmente, ainda é bastante utilizado em ferramentas situadas localmente. Este mecanismo possui uma implementação simples,

2. **Fluxo de dados estruturados** A transferência é realizada via um meio de comunicação em rede (e.g. intranet). Os dados são empacotados e enviados para a ferramenta situada remotamente. Existem vários formatos desenvolvidos para este tipo de mecanismo. Pode-se citar como exemplo HTML (*HyperText Markup Language*) [19] e XML (*eXtensible Markup Language*) [7], que são os formatos mais difundidos e servem como base para outros formatos,

3. **DIF** (*Direct Inter-Tool Funcionality*) As ferramentas comunicam-se entre si diretamente por meio de uma interface, ou compartilhamento de memória de dados, ou mesmo pelo padrão cliente/servidor.

2.1.5 Tipos de esquemas

Segundo o esquema, um formato é classificado como:

1. **Fixo** O esquema é imutável. Em outras palavras, a semântica de cada construção é sempre a mesma, normalmente definida implicitamente no formato.

2. **Mutável** A semântica do formato varia de acordo com uma descrição fornecida explícitamente. Este tipo de esquema permite uma maior flexibilidade ao formato.

2.1.6 Exemplos de classificação

Nesta seção é exposta uma tabela com a classificação de alguns formatos. Esta tabela está organizada de forma a cobrir as propriedades descritas nas seções anteriores. Os formatos classificados são:

- *Annotated Terms* (ATerms) [25] é um formato de representação e transferência baseado em árvores que possui uma biblioteca de funções para manipulá-las. Este formato é descrito em detalhes na seção 5.1.

- *Common Object-based Re-engineering Unified Model* (CORUM) [30] e a sua extensão, CORUM II [14], são ambientes para integração de ferramentas de engenharia reversa. A versão original, CORUM, integra ferramentas que trabalham a nível de código (nível baixo de abstração). CORUM II foi desenvolvido com mais funcionalidades, permitindo a integração de ferramentas que trabalham a qualquer nível de abstração.

- *FAMOOS Information Exchange Model* (FAMIX) [24] é o formato de representação e transferência usado no projeto *FAMOOS* da *University of Beme*. Este projeto suporta a engenharia reversa de sistemas orientados a objetos desenvolvidos em Ada, C++, JAVA e Smaltalk.

- GraX [10] é um formato baseado em um modelo de grafo conhecido como TGraphs. Os grafos baseados nos TGraphs possuem as seguintes características: (1) direcionados, (2) tipados, (3) com atributos nos nós e arestas, (4) as arestas são entidades de primeira classe e são ordenadas. Na seção 5.4, o formato GraX será abordado em maiores detalhes.

- *Graph eXchange Language* (GXL) [28] é um formato capaz de representar todos os níveis de abstração e foi desenvolvido a partir das características de vários formatos. Na seção 5.5, o formato GXL é descrito em maiores detalhes.

- *PROgramming with Graph Rewriting Systems* (PROGRES) [21] é um ambiente que integra um conjunto de ferramentas que apoiam os desenvolvedores a construir, analisar, compilar e verificar especificações para sistemas de reescrita com grafos.

- *Rigi Standard Format* (RSF) [16, 29] é o formato usado na ferramenta *Rigi*. *Rigi* é uma ferramenta de modelagem que suporta a análise de programas e documentos da perspectiva de engenharia reversa.

- *Tuple-Attribute* (TA) [12] expressa grafos que representam programas. O formato TA é baseado na distinção entre entidades e relações, onde as entidades são representadas por nós e as relações são as arestas.

As tabelas 2.1 e 2.2 fornecem uma classificação para os formatos descritos acima usando as propriedades fornecidas nas seções anteriores.

Tabela 2.1: Propriedades características e alguns formatos [13].

		Sintaxe abstrata					**Nível de abstração**		
	Dados estruturados	Árvores	G. tipado.	G. atribuído	G. herdado	G. hierárquico	baixo	médio	alto
ATerms	x	◇					x		
CORUM						x	x		
CORUM II						x	x	x	x
FAMIX	x						x	x	
GraX			x	x	x		x	x	x
GXL			x	x	x	x	x	x	x
PROGRES			x	x	x	x	x	x	x
RSF			x	x			◇	x	x
TA			x	x	x		x	x	x

x - Indica a presença da propriedade no formato.
◇ - Indica que a propriedade é atendida de alguma perspectiva.

Tabela 2.2: Propriedades características e alguns formatos [13] (cont.).

	Tipo de codificação		Mecanism. de Transf.			Tipo de esquema	
	textual	binário	arquivo	fluxo de texto estrut.	Direct. inter-tool	fixo	mutável
ATerms	X	X	X		X	X	
CORUM	X				X	X	
CORUM II	X				X	X	
FAMIX	X			X			X
GraX	X			X			X
GXL	X			X			X
PROGRES	X		X				X
RSF	X		X			X	
TA	X		X				X

X - Indica a presença da propriedade no formato.
◇ - Indica que a propriedade é atendida de alguma perspectiva.

	Implícito	**Explícito**
Interno	CORUM CORUM II	ATerms PROGRES
Externo	RSF	FAMIX[2] GraX[2] GXL[2] TA[1]

Tabela 2.3: Classificação de alguns formatos de transferência de acordo com seus esquemas [13].

2.2 Propriedades dos esquemas

Os esquemas podem ser classificados com relação a duas características: (a) definição, e (b) localização.

2.2.1 Definição do esquema

Os esquemas em um formato podem ser definidos implícita ou explicitamente. Quando:

1. **Implícito** O esquema é definido pelo contexto da aplicação, e,

2. **Explícito** O esquema é definido por alguma especificação.

2.2.2 Localização do esquema

A localização de um esquema é interna ou externa à ferramenta. Sendo:

1. **Interno** O esquema é parte da aplicação. Esquemas classificados neste tipo de esquema são esquemas fixos, necessariamente, ou ,

2. **Externo** O esquema é objeto na transferência entre aplicações. Um esquema definido externamente pode ser fixo ou mutável.

2.2.3 Exemplos de classificação

Nesta seção, os formatos apresentados na seção 2.1.6 são classificados segundo as propriedades de seus esquemas, propriedades essas apresentadas nas seções anteriores. A tabela 2.3 expõe esta classificação.

Em consequência da classificação dos esquemas, são obtidos quatro padrões para formatos de transferência. A Tabela 2.4 sumariza as principais vantagens e desvantagens desses padrões.

2.3 Qualidades de formatos de transferência

Vários autores propuseram características ideais para um formato de transferência padrão. Nesta seção, enumeram-se as características propostas por St-Denis, Schauer e Keller [23]. Eles propuseram 13 características tendo como base suas experiências com os formatos de transferência.

[1]O arquivo que descreve o esquema é transferido simultaneamente com os dados.

[2]O arquivo que descreve o esquema é transferido após os dados.

	Vantagens	Desvantagens
Definição Implícita	• Bom desempenho	• Não extensível • Difícil documentação • Difícil verificação de uma entrada
Definição Explícita	• Extensível • Fácil compreensão • Bem documentado • Fácil verificação de uma entrada	• Baixa desempenho • Implementação complexa
Localização interna	• Bom desempenho	• Difícil manutenção entre duas ou mais ferramentas
Localização externa	• Fácil gerenciamento entre duas ou mais ferramentas	• Difícil verificação da consistência dos dados com o esquema

Tabela 2.4: Vantages e desvantagens relacionadas aos padrões de esquemas [13].

1. **Transparência** Nenhuma alteração no significado dos dados no formato é causada pela utilização de codificadores ou decodificadores.
2. **Escalabilidade** O formato é utilizado para qualquer tamanho de representação.
3. **Simplicidade** Fácil utilização, permitindo descrever sem muita complexidade, fácil entendimento e manutenção.
4. **Neutralidade** É independente de uma ferramenta específica, permitindo o uso para outras ferramentas.
5. **Formalidade** Formalmente bem definido, evitanto interpretações erradas.
6. **Flexibilidade** Permite a utilização de vários tipos de ferramentas, linguagens e sintaxe para os dados e esquemas.
7. **Evolutibilidade** Facilmente adaptável às necessidades futuras.
8. **Popularidade** Fácil adaptação ao formato.
9. **Completude** Todo dado necessário à transferência está inclusa no formato, evitando acessos aos dados consequentes.
10. **Esquema identidade** É a capacidade que um formato possui de converter dados em uma instância de esquema para outra instância preservando a sua identidade.
11. **Reuso de soluções** Capacidade que um formato possui em permitir a reutilização de soluções comprovadas de implementação.
12. **Legibilidade** Fácil entendimento.

	Padrões de formatos de comunicação			
	Implícito/ Interno	**Explícito/ Interno**	**Implícito/ Externo**	**Explícito/ Externo**
Transparência	□	□	□	□
Escalabilidade	χ	$\surd$	χ	$\surd$
Simplicidade	χ	$\surd$	$\surd\surd$	$\surd$
Neutralidade	χ	χ	χ	$\surd\surd$
Formalidade	χ	$\surd$	χ	$\surd\surd$
Flexibilidade	χ	$\surd$	χ	$\surd\surd$
Evolvabilidade	χ	χ	χ	$\surd\surd$
Popularidade	□	□	□	□
Completude	□	□	□	□
Esquema identidade	χ	$\surd\surd$	χ	$\surd\surd$
Reuso	χ	χ	χ	$\surd$
Legibilidade	χ	$\surd$	χ	$\surd$
Integridade	□	□	□	□

"□" indica que nada se conclui sobre um padrão satisfazer ou não a propriedade
"χ" indica que o padrão não satisfaz a propriedade
"$\surd$" indica que o padrão satisfaz a propriedade
"$\surd\surd$" indica que o padrão satisfaz a propriedade e que esta é uma característica significante no formato.

Tabela 2.5: Padrões de formatos X Características de um formato de transferência padrão [13].

13. **Integridade** Uso de mecanismos especiais que asseguram a transferência dos dados, evitando erros.

A Tabela 2.5 expõe as características que são atendidas pelos padrões de formatos de transferência.

Analisando as aceitações das propriedades pelos padrões, Dean Jim em [13] verificou que o padrão mais adequado a um formato de transferência padrão é o externo explícito.

Capítulo 3

AGraphs

3.1 Aplicação

AGraphs são utilizados na representação de linguagens (programação, especificação, etc.). Inicialmente, AGraphs foram utilizados na implementação das duas instâncias da ferramenta de apoio ao desenvolvimento de especificações algébricas, FERUS [15, 18], e no *model checker* CV [9] para uma sublinguagem de VHDL.

Na seção seguinte, descrevem-se superficialmente as características de uma das ferramentas que serviu como motivação para o desenvolvimento deste trabalho.

3.1.1 FERUS

FERUS é uma ferramenta com interface gráfica que fornece ao usuário um ambiente de criação, manipulação e prototipação de unidades em uma linguagem de especificação algébrica. Na instância FERUS-CASL, as unidades manipuladas são escritas em CASL [1, 5], e na instância FERUS-ELAN, as unidades são escritas em ELAN [3, 2].

No desenvolvimento de uma especificação com FERUS, é possível, utilizar editores de texto integrados ao ambiente e/ou operadores de transformação, incluindo a generalização [17]. Em ambos os casos, o usuário é assistido por : (a) um analisador que verifica a corretude sintática e semântica da especificação criada; e, (b) uma ferramenta de prototipagem que permite a execução de uma especificação. Na prototipagem, a especificação executada deve respeitar algumas regras de executabilidade.

Funcionalidades

Como citado acima, FERUS permite a criação, manipulação e prototipação de unidades em uma linguagem. Estas funcionalidades são descritas abaixo:

- **Edição**: Alteração manual de unidades usando um editor. O editor utilizado é uma escolha do usuário, podendo ser um editor padrão ou externo ao ambiente.
- **Compilação/Decompilação**: A compilação é um procedimento que permite ao usuário: (a) certificar-se da corretude sintática e semântica das suas especificações, e, (b) preparar um componente de especificação para ser executado ou manipulado pelos operadores de transformação. O processo de compilação cria uma representação interna (também denominado formato interno) do componente. A decompilação é um procedimento dual da compilação, que obtem a especificação textual

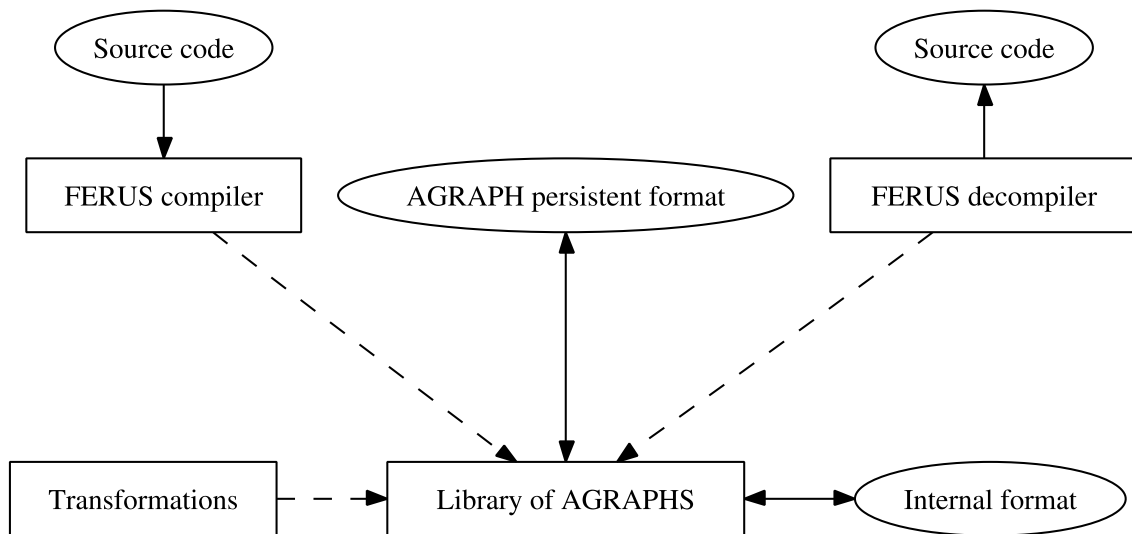


Figura 3.1: Arquitetura do FERUS.

a partir do formato interno. A biblioteca AGraphs é usada na implementação do formato interno da ferramenta FERUS.

- **Prototipação de especificações**: A prototipação é a execução dos componentes de uma especificação. Normalmente, usa-se a prototipação com o objetivo de observar os resultados das operações de uma especificação.

Além destas existem outras funcionalidades, denominadas operações de transformação. Estas operações manipulam o formato interno que representa uma especificação. As operações de transformação são:

- **Renomagem**: Renomeia símbolos declarados em um componente. Normalmente, a renomagem é utilizada com o objetivo de adaptar componentes a outros contextos de uso.

- **Generalização**: Abstrai características de um componente conservando parcialmente a sua semântica. Em outras palavras, o componente fica mais abrangente na sua funcionalidade. Por exemplo, componentes de um modelo específico, após a generalização, descrevem uma classe de modelos, aumentando o nível de abstração e o potencial de reutilização destes componentes.

- **Instanciação**: Aumenta o grau de especialização de um componente. Este procedimento é dual à generalização.

- **Extensão**: Enriquece modelos através da declaração de novos símbolos e/ou da especialização da interpretação de símbolos declarados anteriormente.

- **Redução** Elimina ou esconde partes de componentes de uma especificação.

Arquitetura

FERUS possui uma arquitetura modular. A Figura 3.1 expõe a relação entre os principais módulos do FERUS e a descrição de cada módulo é fornecida abaixo:

- **Biblioteca AGraph** Disponibiliza um conjunto de funções de criação e manipulação do formato interno. O formato interno é um grafo denominado AGraph que mantém informações sobre uma unidade. Na Figura 3.1, os módulos que acessam as funções da biblioteca AGraph estão ligados a este módulo por setas tracejadas indicando a invocação das funções desta.

- **Compilador FERUS** Traduz componentes em linguagem para o formato interno FERUS. O compilador interage com a biblioteca AGraph criando e mantendo o grafo correspondente a uma unidade. Posteriormente, uma unidade compilada é armazenada em um repositório de unidades compiladas.

- **Decompilador FERUS** Reconstrói os componentes de uma unidade contida no repositório de unidades compiladas. As unidades compiladas estão no formato persistente AGraph (veja seção 3.3.4). Com o objetivo de analisar o grafo e reconstruir o componente correspondente, o decompilador usa as funções da biblioteca AGraph.

- **Transformações** Contém as operações de renomagem, generalização, instanciação, extensão e redução. Semelhante aos módulos de compilação e decompilação, este módulo acessa as funções da biblioteca AGraph para implementar as suas operações.

3.2 Especificação

As bibliotecas implementadas usando o conceito AGraph possuem a estrutura de dados e os perfis de interface semelhantes. Estes conceitos comuns compõem o formato interno de representação AGraph e o seu ambiente de manipulação, os quais podem representar várias linguagens de programação. Estes conceitos e as descrições das características do AGraph são fornecidos nas seções seguintes.

O objetivo principal das especificações é descrever os conceitos que permeiam AGraphs e estruturas de representação e manipulação (bibliotecas). Com este objetivo foram desenvolvidas especificações em diferentes categorias de especificações: (a) uma especificação formal, usando a linguagem de especificação *B*, (b) uma especificação gráfica, usando diagramas UML, e, (c) uma especificação textual, descrição textual fornecida nas seções (3.2.1 e 3.2.2).

A especificação formal em *B* foi desenvolvida para descrever a estrutura AGraph em memória, definindo a estrutura antes e depois do uso das funções de manipulação, garantindo um conjunto de características que devem ser verificadas.

A razão pela escolha da linguagem de especificação *B* deve-se ao uso habitual dos desenvolvedores, propiciando um melhor aproveitamento dos recursos que esta linguagem oferece e principalmente pelo conjunto de ferramentas disponibilizadas para auxiliar na construção da especificação.

A especificação gráfica em UML visa a descrição da biblioteca, fornecendo o conjunto de funções que compõe a biblioteca AGraph e seus comportamentos. Cinco categorias de diagramas UML foram usadas nas especificações, sendo:

- o *diagrama de casos de uso* enumera as funções que compõem as bibliotecas AGraphs e as suas relações com as outras funções.

- o *diagrama de classes* descreve as relações entre os elementos da unidade, as categorias de tipos de nó e seus componentes (atributos, arestas e hiperarestas) nos esquemas AGraphs. Esta descrição é geral para todos os esquemas, ou seja, descreve a organização dos elementos que compõem um esquema.

- o *diagrama de objetos* fornece uma instância do diagrama de *Classes*. Diferente do diagrama de classes, um diagrama de objetos fornece a relação dos elementos que compõem um esquema para uma específica linguagem representada.

- os *diagramas sequenciais e de atividades* descrevem o comportamento das funções que compõem a biblioteca AGraph. Nos diagramas sequenciais, o comportamento observado é a interação entre os módulos que participam desta função e a sequência de mensagens trocadas. Nos diagramas de atividades, a sequência de atividades e seus requisitos e resultados são os comportamentos observados.

- o *diagrama de estados* foi criado para descrever os possíveis estados que a estrutura de dados AGraph pode assumir.

As diferentes categorias de especificações se completam. Assim, características não observadas em uma categoria de especificação são fornecidas em pelo menos uma das outras categorias de especificações.

As especificações formal e gráficas são fornecidas completamente nos apêndices A e B, respectivamente.

Nesta seção, descreve-se os conceitos AGraphs usando a descrição textual como base, usando a especificação gráfica ou formal como descrição auxiliar.

3.2.1 Estrutura de dados AGraph

Muitas linguagens de programação modernas possuem o conceito de unidade compilada com as seguintes características: (a) uma unidade pode ser referenciada por outras unidades, e, (b) uma unidade pode ser compilada (ou analisada) independentemente das suas unidades que esta faz referência, entretanto, garantindo que as unidades referenciadas foram compiladas antes.

Em algumas linguagens, o conjunto de unidades pode compor as bibliotecas de unidades. De modo geral, algumas unidades podem referenciar as unidades de outras bibliotecas.

No formato AGraph, cada unidade é representada por um conjunto de nós interconectados, onde cada nó mantém informações sobre um elemento da sintaxe da linguagem desta unidade. Logo, existem diversos tipos de nó, um para cada não-terminal na gramática da linguagem. Por exemplo, um nó do tipo *SortDef* na biblioteca AGraph para CASL representa a definição de um sorte (tipo de dados no contexto de especificações algébricas).

Cada tipo de nó possui uma assinatura, composta por um conjunto de *atributos* tipados e identificados, *arestas* e *hiperarestas*.

Define-se *atributo* como um valor atômico não-estruturado que mantém uma informação sobre o nó. Por exemplo, os nós de definição de unidades CASL possuem o atributo *aIdentifier*, o qual armazena o identificador desta especificação.

As *arestas* apontam para outros nós, definindo grafos tipados arbitrariamente complexos, permitindo realizar a expansão das regras na sintaxe abstrata da linguagem. Uma aresta é usada para representar uma relação unidirecional entre dois nós. Por exemplo, a relação "uma variável possui um tipo" pode ser representada por uma aresta de um nó que armazena informações sobre uma variável para um nó que armazena informações sobre um tipo (constante inteira, ponto flutuante, etc.).

Na relação entre um nó e um conjunto de nós usam-se as *hiperarestas* que são arestas que apontam para uma lista de nós. Por exemplo, em uma unidade de uma linguagem podem existir várias declarações de *sortes*, assim, existe uma hiperaresta da declaração da unidade para o conjunto de declaração de *sortes*.

Um AGraph não possui um elemento "concreto" hiperaresta. Uma hiperaresta é um elemento conceitual e modelado, em AGraphs, usando uma lista encadeada. Logo, o nó origem possui uma aresta para o nó cabeça da lista encadeada e cada nó da lista encadeada aponta para um dos nós que pertencem ao conjunto destino.

A especificação denominada estática AGraph descreve as relações dos elementos que compõem os AGraphs. Por exemplo a relação entre nós de diferentes tipos. Logo, na especificação estática em B (apêndice A.1), o seguinte trecho, que pertence à máquina `NodeDescription`, define os elementos que compõem um nó.

```
node: NAME +-> (POW(attributeNames) *
                POW(edgeNames) *
                POW(hyperedgeNames)) &
nodeNames: dom(node)
```

`node` é a função que relaciona um identificador de nó (`nodeNames`) à estrutura de um nó. Um nó é formado por um conjunto de atributos, arestas e hiperarestas, representadas, respectivamente, por `attributeNames`, `edgeNames` e `hyperedgeNames`. Para melhorar a legibilidade da descrição, os elementos que compõe um nó são referenciados pelos seus identificadores (e.g. `attributeNames`). As informações sobre estes elementos são obtidos por funções definidas para cada elemento. Por exemplo, a função de recuperação de informações a partir do identificador de um atributo é:

```
attribute: NAME +-> TYPENAMES &
attributeNames = dom(attribute)
```

A função `atribute` associa um identificador de atributo (`NAME`) ao tipo dos valores que o atributo pode assumir (`TYPENAMES`). `attributeNames` é um subconjunto de `NAME`, sendo os identificadores dos atributos definidos. `TYPENAMES` é o conjunto de tipos pré-definidos ou derivados que um atributo pode assumir.

Semelhante modo de recuperar informações é usado sobre as arestas e hiperarestas que compõe um nó. Como mostra o trecho a seguir.

```
edge: NAME +-> POW1(nodeNames) &
edgeNames = dom(edge) &

hyperedge: NAME +-> (nodeNames * POW1(nodeNames)) &
hyperedgeNames = dom(hyperedge)
```

Uma restrinção imposta pela implementação é que os identificadores de um atributo, aresta, hiperaresta ou de um nó nunca sejam iguais. Esta condição é definida abaixo

```
attributeNames /\ edgeNames /\ hyperedgeNames /\ nodeNames = {}
```

Como pode ser observado, a conectividade entre os nós é determinada pelas funções `edge` e `hyperedge`. Ambas as funções associam um identificador de elemento (`edgeNames` e `hyperedgeNames`) a um conjuntos de identificadores de nó (`nodeNames`).

Exemplo 3.1 *Como exemplo de uma estrutura de dados AGraph, usa-se o trecho de código abaixo que fornece uma unidade em uma linguagem fictícia.*

```
module Nat
 sorts
   nat
 ops
   zero: nat,
   succ: nat -> nat
end
```

Esta unidade é identificada por `Nat` *e possui a declaração do tipo (sorte)* `nat` *e de dois operadores,* `zero` *e* `succ`*, ambos com imagem em* `nat` *e* `succ` *com domínio em* `nat`.

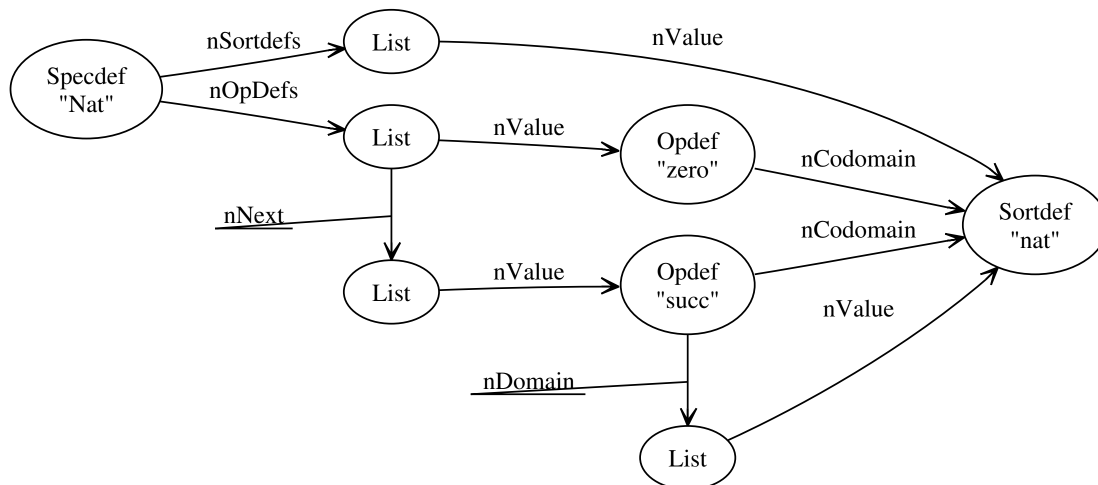


Figura 3.2: Exemplo de AGraph para uma especificação

A Figura 3.2 expõe um possível AGraph que representa a unidade `Nat`*. A figura contém um conjunto de nós interligados com o nó* `Specdef` *identificado por* `Nat` *sendo o nó raiz. Este nó está conectado a dois nós do tipo lista pelas arestas* `nSortdefs` *e* `nOpdefs`*. O nó apontado por* `nSortdefs` *é o único na lista de declarações de sortes e aponta (*`nValue`*) para o nó de declaração do sorte* `Sortdef` *identificado por* `nat`*. O nó apontado por* `nOpdefs` *é o primeiro nó na lista de declaração de operadores e possui duas arestas,* `nValue` *aponta para o nó declaração de operador a ele associado* `Opdef` *(*`zero`*), e* `nNext` *aponta para o próximo nó na lista que por sua vez a aresta* `nValue` *aponta para a segunda declaração de operadores,* `Opdef`*, identificada por* `succ`*. As duas declarações de operadores possuem imagem (*`nCodomain`*) em* `nat` *(*`Sortdef`*), e o operador* `succ` *possui domínio um sorte* `nat` *(*`nDomain`*)* .

3.2.2 Interface fornecida pela biblioteca AGraphs

A interface fornecida pela biblioteca AGraph (AGraph API) permite a manipulação do grafo que representa uma expressão da linguagem alvo a partir do uso de funções, escondendo os detalhes da estrutura do grafo.

O diagrama de *Casos de Uso* da Figura 3.3 expõe as funções que um usuário pode acessar em uma biblioteca AGraph. Na figura, o ator *User* representa o usuário, e as elipses são as funções que estão contidas na biblioteca AGraph (*AGraph Library*).

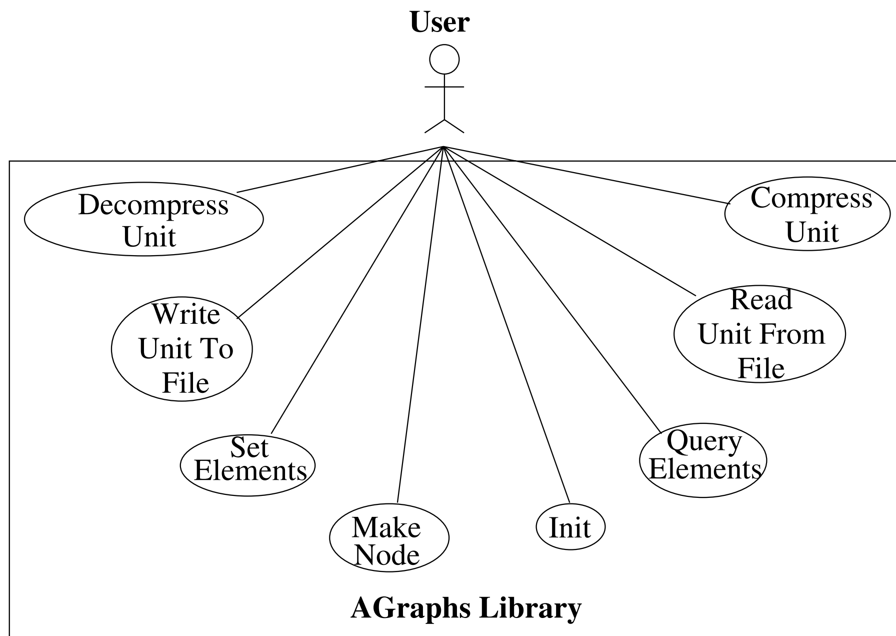


Figura 3.3: Funções da biblioteca AGraphs fornecidas aos usuários

As funções fornecidas pela API são:

- Função de inicialização (`Init`) da biblioteca que faz a alocação inicial do grafo e das estruturas auxiliares. A inicialização é o primeiro procedimento que o usuário executa antes das outras funções.

- Construtores, funções *make* (`Make Node`), para cada tipo de nó, os quais alocam memória e atribuem valores iniciais.

- Acessores para recuperar, funções de consulta *query* (`Query Elements`), e funções de atribuição, funções *set* (`Set Elements`), valores aos atributos, arestas e hiperarestas. Para cada atributo, aresta ou hiperaresta de cada tipo de nó existe uma função *query* e uma função *set* associadas.

- Funções de leitura (`Read Unit From File`) e escrita (`Write Unit To File`). A função de leitura recupera uma unidade a partir da representação persistente armazenada em disco. Em sentido contrário, a função de escrita armazena uma unidade em disco, mantendo os atributos e as relações entre os nós.

- Função de compactação (resp. descompactação) cria um arquivo no formato binário (resp. textual) a partir de um arquivo no formato textual (resp. binário). Na figura 3.3, a função de compactação (resp. descompactação) é representada pela elipse `Compress Unit` (resp. `Decompress Unit`).

Nos diagramas UML usados na descrição gráfica, o diagrama de sequência e de atividades descrevem o comportamento das funções da biblioteca de manipulação do AGraph.

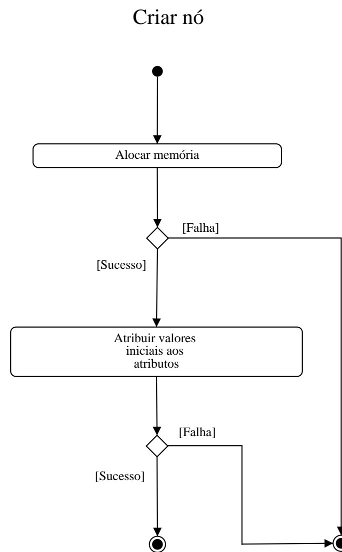


Figura 3.4: Diagrama de atividades para a função de criação de um nó

Para ilustrar o comportamento da função de criação (`make`), a figura 3.4 contém o diagrama de atividades.

Neste diagrama, a primeira atividade realizada é a alocação do nó (`Alocar memória`). Se a alocação foi bem sucedida a tarefa de atribuição dos valores iniciais dos atributos (`Atribuir valores iniciais aos atributos`) é realizada. Se alguma das etapas da função de criação de um nó obtiver falha, a função é finalizada.

De outra perspectiva, a especificação formal em B também descreve o comportamento das funções da biblioteca AGraph. Ilustra-se essa afirmação expondo o trecho da especificação dinâmica em B da função de criação de um nó.

```
01. make(Ats, Eds, Hyps, pnN) =
02.  PRE
03.   pnN : nodeNames & /* corresponde a um tipo de nó existente */
04
05.   /* Verifica se o tipo dos parâmetros correspondem aos atributos.*/
06.   Ats <: (attributeNames * ENUMVALUES) &
07.
08.   /* Verifica se os atributos podem ser associados ao nó.*/
09.   !a . (a : dom(Ats) =>
10.         (a : nodeattributeNames(pnN) &
11.          Ats(a) : Types(attribute(a)))) &
12.
13.   /* Verifica se o tipo dos parâmetros correspondem às arestas.*/
14.   Eds <: (edgeNames * nodeId) &
15.
16.   /* Verifica se as arestas podem ser associadas ao nó.*/
17.   !e . (e : dom(Eds) =>
18.         (e : nodeedgeNames(pnN) &
19.          nodeInstanceName(Eds(e)) : edge(e))) &
20.
```

```
21.    /*
22.     * Verifica se o tipo dos parâmetros
23.     * correspondem às arestas multivaloradas.
24.     */
25.    Hyps <: (hyperedgeNames * nodeId) &
26.
27.    /*
28.     * Verifica se as arestas multivaloradas
29.     * podem ser associadas ao nó.
30.     */
31.    !h . (h : dom(Hyps) =>
32.          (h : nodehyperedgeNames(pnN) &
33.           nodeInstanceName(Hyps(h)) = hyperedge(h)))
34.
35.  THEN
36.   ANY pos WHERE
37.    pos : NAT1 &
38.    pos /: nodeId
39.   THEN
40.    nodeInstanceName := nodeInstanceName \/ {pos |-> pnN} ||
41.    nodeInstanceAttributes := nodeInstanceAttributes \/ {pos |-> Ats} ||
42.    nodeInstanceEdges := nodeInstanceEdges \/ {pos |-> Eds} ||
43.    nodeInstanceHyperedges := nodeInstanceHyperedges \/ {pos |-> Hyps}
44.   END
45.  END;
```

A função de criação (`make`) recebe um conjunto de atributos (`Ats`), arestas (`Eds`) e hiperarestas (`hyps`) que serão associadas aos seus componentes após a locação (associação inicial). O outro parâmetro da função fornece o identificador do tipo de nó a ser criado (`pnN`).

Como pré-requisito para a execução da função de criação, cada elemento que pertença aos conjuntos de atributos, arestas e hiperarestas passados como parâmetros devem possuir uma associação na estrutura do nó a ser criado e o identificador do nó (`pnN`) deve existir pertencer a um identificador de nó válido (linha 03). Estes pré-requisitos são verificados no bloco `PRE...THEN`. Por exemplo, para verificar se todos os atributos do conjunto de atributos parâmetros possuem correspondentes na estrutura do nó, existe a seguinte verificação:

```
09.   !a . (a : dom(Ats) =>
10.         (a : nodeattributeNames(pnN) &
11.          Ats(a) : Types(attribute(a)))) &
```

Assim, para todo atributo `a` (linha 09), este deve pertencer ao conjunto de tipos de atributos válidos para o tipo de nó a ser criado (linha 10 e 11).

Verificado todos os pré-requisitos, a escolha de um identificador único para o nó é realizada,

```
36.   ANY pos WHERE
37.    pos : NAT1 &
38.    pos /: nodeId
39.   THEN
```

Este trecho obtém um identificador único (`pos`), representando a alocação em memória e a atribuição de um endereço que é único. `nodeId` é o conjunto de identificadores dos nós criados (alocados).

E finalmente, as associações são efetuadas,

```
40.   nodeInstanceName := nodeInstanceName \/ {pos |-> pnN} ||
41.   nodeInstanceAttributes := nodeInstanceAttributes \/ {pos |-> Ats} ||
42.   nodeInstanceEdges := nodeInstanceEdges \/ {pos |-> Eds} ||
43.   nodeInstanceHyperedges := nodeInstanceHyperedges \/ {pos |-> Hyps}
```

atualizando as relações: (1) identificador do nó e seu tipo (linha 40), (2) os atributos associados ao nó (linha 41), (3) as arestas associadas ao nó (linha 42), e (3) as hiperarestas associadas ao nó (linha 43).

Comparando os diagramas UML e a especificação em B, o primeiro abstrai as subtarefas realizadas em cada tarefa e a organização da estrutura de dados AGraphs, e no segundo, são observados. Em contrapartida, os diagramas UML são mais simples de compreender do que as especificações B.

3.2.3 Esquemas AGraphs

Como citado na seção 3.2.1, o conceito AGraph é geral, mas os nós e seus perfis (atributos, arestas e hiperarestas) são específicos para a linguagem representada. Para cada linguagem podem existir várias estruturas AGraph diferentes capazes de manter os mesmos dados. Estas estruturas AGraph definem uma classe de grafos e são descritas por um *esquema*.

Na figura 3.5, o diagrama UML de Classes fornece as informações necessárias para a descrição dos componentes de um esquema AGraph e as suas relações. Os componentes que formam um esquema são: uma unidade (`Unit`), diversas categorias de tipos de nó (`Node`, `RootNode`, `Import` e `List`), e seus componentes (`Datatype`, `Attribute`, `Edge` e `Hyperedge`), que são descritos a seguir.

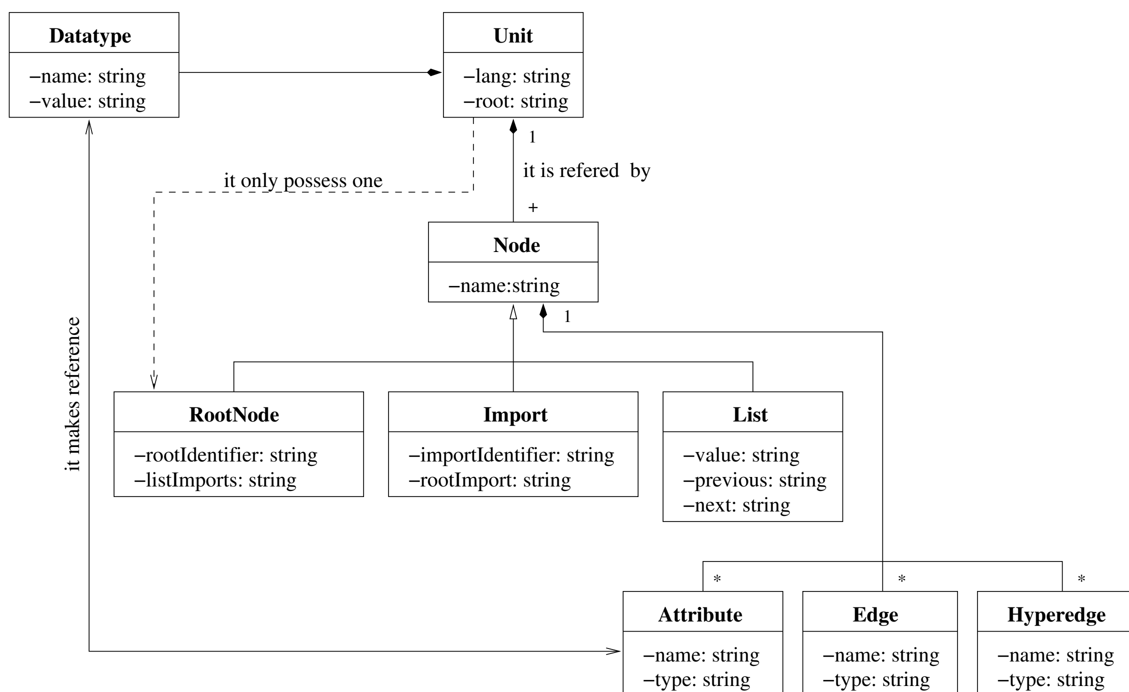


Figura 3.5: Diagrama UML de classes para esquemas AGraph

Cada AGraph representa uma unidade (`Unit`) para uma linguagem alvo específica, e contém somente um nó raiz. O esquema de uma unidade define o identificador da linguagem (`lang`) e o identificador da classe dos nós raiz (`root`).

Um identificador de uma linguagem é uma etiqueta, que serve para nomear as funções da API do AGraph correspondente. Esta é uma das convenções adotadas nas bibliotecas AGraphs e refletem na descrição do esquema AGraph. Na seção 3.3, as convenções são descritas com maiores detalhes.

Uma unidade é formada por várias categorias de nós, existindo três categorias especiais de nós: a categoria dos nós raiz, importação e lista. Todas as categorias de nós possuem um identificador (name) e possuem um conjunto de componentes. Os componentes de uma classe de nó são:

- Atríbutos (Attribute): Mantêm informações de tipos primitivos (por exemplo, inteiro e string) ou tipos enumerados definidos pelo usuário (Datatype). Cada atributo de nó possui um identificador (name) e tipo (type) definidos no esquema.

- Arestas (Edge): Apontam para outros nós no grafo. Um componente aresta é definido por seu identificador (name) e por um conjunto de identificadores de tipo de nó (type). Este conjunto de identificadores indica os tipos de nó para as quais aresta pode apontar.

- Hiperarestas (Hyperedge): Semelhantes às arestas, as hiperarestas apontam para outros nós, entretanto, uma aresta aponta para um nó, enquanto que uma hiperaresta aponta para vários nós usando um nó do tipo lista como ligação. Assim, um componente hiperaresta é definido por seu identificador (name) e pelo tipo de nós lista que essa hiperaresta utiliza (type). A restrição sobre os tipos de nós apontados por uma hiperaresta é a restrição correspondente ao tipo de nós lista usado.

Cada unidade possui uma etiqueta que a identifica permitindo que outras unidades a referenciem, e, analogamente, esta unidade pode fazer referência a outras unidades, assim, um nó do tipo raiz possui um identificador (rootIdentifier) e uma hiperaresta (listImports) para os nós raiz das unidades importadas.

Por motivo de implementação, o acesso às raízes das unidades importadas não é direto, assim, foi definida uma categoria de nó importação. Na descrição de nós importação é necessário informar o seu identificador (importIdentifier) e a aresta que aponta para o nó raiz da unidade importada (rootImport). O identificador do nó importação armazena o mesmo valor do identificador da unidade importada.

Possibilitando a associação múltipla, AGraphs usa nós especiais do tipo lista que simula listas encadeadas, aqui denominadas hiperarestas. Cada nó do tipo lista aponta para um nó alvo na associação e para o seu antecessor e sucessor na lista. Assim, um nó lista possui três arestas: duas arestas ligando-o a outros nós na lista, para o nó anterior (previous) e para o seguinte (next), e a terceira aresta aponta para o nó alvo (value).

Na descrição de um tipo de nó lista é necessário informar um identificador para este tipo e o(s) tipo(s) dos nós alvos. A declaração das arestas que apontam para o nó anterior e posterior na lista não é necessária, visto que estes são do mesmo tipo da lista definida.

As características do esquema de AGraph também são observadas na especificação em B. A máquina UnitMeta, pertencente a especificação estática, contém a descrição das informações dos esquemas AGraph. Esta máquina é mostrada abaixo.

```
01. MACHINE
02.  UnitMeta(language_name, STRINGPARAM)
03.
04. CONSTRAINTS
```

```
05.  language_name : STRINGPARAM
06.
07. SEES
08.  Datatype, NodeDescription
09.
10. CONSTANTS
11.  rootnode, imports, lists
12.
13. PROPERTIES
14.  rootnode: node &
15.  imports: POW(node) &
16.  lists: POW(node) &
17.
18.  (lists = {} => hyperedge = {}) &
19.  dom(ran(hyperedge)) <: dom(lists) &
20.
21.  rootnode /: imports &
22.  rootnode /: lists &
23.  imports /\ lists = {} &
24.  (lists = {} => imports = {})
25. END
```

`UnitMeta` recebe dois parâmetros: `language_name` é a etiqueta da linguagem do esquema, e `STRINGPARAM` é o conjunto de conjuntos de caracteres (strings) possíveis para a etiqueta da linguagem.

As máquinas `Datatype` e `NodeDescription` contêm, respectivamente, as definições dos tipos pré-definidos e definidos pelo usuário, e a organização dos nós.

Três nós especiais são definidos: `rootnode` que é o nó raiz (linha 14), `imports` que é o conjunto de tipos de nós importação (linha 15) e `lists` que é o conjunto de tipos de nós lista (linha 16).

Algumas propriedades são definidas da especificação anterior:

1. Como a hiperaresta utiliza nós do tipo lista para efetuar a associação entre os nós, se, no esquema, não foi definido nenhum nó lista, não devem existir hiperarestas(linha 18 e 19).

2. A importação utiliza hiperarestas, se não existe hiperarestas, não existe nó importação. Como consequência da propriedade anterior, então, se não foram definidos nós lista, não existem nós importação (linha 24).

3. O nó raiz não pertence a outro tipo especial de nó, nem ao conjunto de nós importação (linha 21) e nem ao conjunto de nó lista (linha 22). A mesma definição é usada entre os conjuntos de nós importação e nós lista, nenhum nó importação é um nó lista e vice-versa (linha 23).

3.3 Implementação

Nesta seção, descrevem-se as implementações das bibliotecas AGraphs, expondo as características comuns, independentemente da linguagem representada.

Nas implementações das bibliotecas AGraphs (seção 3.1) foi definido um conjunto de convenções. Estas convenções são adotadas para todas as bibliotecas AGraphs e as principais são as seguintes:

Na linguagem C:

```
 struct {
    caslRef position;
    void * aToolInfo;
    caslKind aKind;
    caslRef nRoot;
    unsigned aLine;
    char * aIdentifier;
    caslRef nSpecImports;
    caslRef nSortDefs;
    caslRef nOperatorDefs;
    caslRef nVariableDefs;
    caslRef nAxiomDefs;
} SpecDef;
```

Na linguagem JAVA:

```
public class SpecDef {
/* Declaração dos atributos */
  caslRef position;
  Object aToolInfo;
  caslKind aKind;
  caslRef nRoot;
  unsigned aLine;
  String aIdentifier;
  caslRef nSpecImports;
  caslRef nSortDefs;
  caslRef nOperatorDefs;
  caslRef nVariableDefs;
  caslRef nAxiomDefs;
  /* Definição dos métodos */
  ...
}
```

Figura 3.6: Estrutura para o nó definição de unidade CASL.

- os atributos são identificados por um “a” no início do nome do elemento. Por exemplo, `aNumber` é um atributo.
- as arestas e hiperarestas são identificadas por um “n” no início do nome do elemento. Por exemplo, `nSortdef` é uma aresta ou hiperaresta.
- todas as funções da biblioteca iniciam com a etiqueta da linguagem representada. Por exemplo, a função `caslQueryaIdentifier` é uma função `query` da biblioteca AGraph para uma linguagem com etiqueta `casl`.

Não é obrigatório o respeito a essas convenções, contudo, elas facilitam o entendimento da estrutura de dados AGraph e são significantes no desenvolvimento de novas implementações usando o gerador automático, como pode ser observado na seção 4.1.

3.3.1 Implementação dos nós do grafo

A figura 3.6 é um exemplo de estrutura de nó. Este nó corresponde à definição de uma unidade em CASL implementada em C e em JAVA.

Os quatro primeiros componentes das estruturas são obrigatórios para todos os nós que formam o grafo, independentemente da linguagem representada e da linguagem de implementação. Os outros componentes (atributos, arestas e hiperarestas) mantêm informações específicas da linguagem representada (neste exemplo é CASL). Os componentes obrigatórios são descritos a seguir:

- `Position`: mantém a referência para o nó na representação interna.
- `aToolInfo`: atributo ponteiro genérico, fornecido para manter informações temporárias. A informação não é armazenada em disco quando o grafo é escrito.
- `aKind`: armazena o tipo do nó.
- `nRoot`: aresta que aponta para o nó raiz da unidade ao qual o nó pertence. Para cada unidade existe somente um nó raiz.

O campo Position e todas as outras referências a nós (arestas e hiperarestas) são do tipo langRef, onde lang é a etiqueta da linguagem representada.

Nas duas implementações, C e JAVA, seja lang a etiqueta da linguagem representada, todos os tipos de nós são derivados do tipo de nó langNode, sendo que na implementação C, simula-se a derivação usando a união (union) de estruturas (struct), onde cada struct define um tipo de nó. Em JAVA, a derivação é implementada usando a herança disponibilizada pela própria linguagem.

3.3.2 Dependência de unidades

Diferentemente de unidades independentes, que possuem grafos fortemente interconectados, unidades que importam outras unidades possuem nós que apontam para subgrafos das unidades importadas.

Devido à dependência que ocorre entre a unidade importadora e as unidades importadas, antes que a representação interna AGraph da unidade importadora seja construída em memória, por conveniência para a implementação, é necessária a construção das representações internas das unidades importadas.

Assim, a representação interna é composta por duas tabelas: a primeira contém os nós que compõem à unidade representada, e a segunda contém o conjunto das unidades importadas.

3.3.3 Interface da biblioteca AGraph

Na seção 3.2.2, são descritas as finalidades das funções da API das bibliotecas AGraph. Aqui, são fornecidos exemplos para cada operação aproximando-as das características de implementação das estruturas dos nós.

1. langInit(), inicializa as tabelas que compõem a representação interna, permitindo a criação e manipulação dos nós.

2. Exemplo da declaração do construtor para o tipo de nós Specdef nas linguagens C e JAVA é mostrado na figura 3.7. O tipo dos nós SpecDef representa uma definição de unidades em CASL.

3. Uma função de entrada (resp. saída) para leitura (resp. escrita) de uma unidade a partir de (resp. para) um arquivo no formato textual AGraph (3.3.4). O exemplo

   ```
   caslNode caslReadSpec(char * specName);
   ```

   é a declaração da função de leitura de uma unidade a partir do arquivo cujo o nome é fornecido por specName, construindo a representação interna e armazenando em caslNode. Observe que a linguagem representada possui a etiqueta casl, por isso caslNode e caslReadSpec.

4. Para cada componente (atributo, aresta, ou hiperaresta) de cada tipo de nó existem as funções *query* e *set* correspondentes. Em C, a função *query* possui um nó como parâmetro e retorna o valor atual do correspondente componente. Em JAVA, o nó alvo é um objeto e a função *query* é um método da classe deste nó, retornando o valor atual do componente correspondente. As funções *set*, na linguagem C, possuem

Implementação C:

```
caslNode caslMakeSpecDef
 ( unsigned aLine,
   char*paIdentifier,
   caslNode pnSpecImports,
   caslNode pnSortDefs,
   caslNode pnOperatorDefs,
   caslNode pnVariableDefs,
   caslNode pnAxiomDefs,
   caslNode pnRoot);
```

Implementação JAVA:

```
public void caslMakeSpecDef
 ( unsigned aLine,
   String paIdentifier,
   caslNode pnSpecImports,
   caslNode pnSortDefs,
   caslNode pnOperatorDefs,
   caslNode pnVariableDefs,
   caslNode pnAxiomDefs,
   caslNode pnRoot);
```

Figura 3.7: Exemplos de declarações de construtores de nós.

dois parâmetros: o nó alvo e o novo valor para o componente correspondente. Na linguagem JAVA, o nó alvo é um objeto e a função *set* é um método de sua classe com um parâmetro: o novo valor para o componente correspondente. O resultado da função *set* é a atribuição do novo valor ao componente correspondente no nó alvo. Exemplos de declarações de funções de acesso:

Em C:

```
caslNode caslQuerySpecImports(caslNode n);
void caslSetSpecImports(caslNode n, caslNode new);
```

Em JAVA:

```
public caslNode QuerySpecImports();
public void caslSetSpecImports(caslNode new);
```

3.3.4 Formatos persistentes AGraph

A codificação do formato no modo textual e no binário são os formatos persistentes fornecidos pelos AGraphs.

Ambos os modos de codificação são construídos para representar os dados de um AGraph persistente. A textual é construída a partir da aplicação da função de biblioteca AGraph recebendo como parâmetro uma unidade em memória, e a binária é formada a partir dos métodos de compactação (seção 4.2.1).

A seguir, descreve-se as modalidades de formatos persistentes disponíveis para AGraphs.

Formato textual AGraph Um arquivo no formato textual AGraph é formado por um conjunto de constantes inteiras e `strings` agrupadas que compõem o cabeçalho e as descrições de nós. A descrição de nó mantém informações sobre um nó e é restrita ao tipo de nó descrito. As informações mantidas em uma descrição de nó variam de acordo com o esquema usado na definição da estrutura do AGraph.

Diferente das descrições dos nós, a organização do cabeçalho é fixa e contém as seguintes informações:

- informações relacionadas à versão da representação da unidade e da biblioteca AGraph utilizada. Estas informações são representadas por constantes inteiras de 16 bits e indicam: (a) a instância da biblioteca AGraph utilizada para construir a representação, (b) a versão da instância da biblioteca AGraph, e, (c) a data da última modificação no arquivo.

- a quantidade de unidades importadas indiretamente[1] e os seus identificadores,
- a quantidade de unidades importadas diretamente e os seus identificadores,
- a quantidade de descrições de nós,
- a localização da descrição do nó raiz da representação no formato persistente.

Exemplo 3.2 *A Figura 3.8 fornece um exemplo do formato textual AGraph.*

```
linha. código

01. <131972 100 1112387421>
02. 1
03. Elem
04. 2
05. Nat
06. String
07. 3
08. 1
09. <0 1 4 NatElem 0 0 0 3 0 0 0 0 0 0 >
10. <2 0 1 4 natElem >
11. <10 0 2 0 0 0 0 >
```

Figura 3.8: Exemplo do formato textual AGraph

Nesta figura, a primeira linha contém informações sobre a versão da biblioteca. A linha 02 contém a quantidade de unidades importadas (1), as linhas seguintes possuem os identificadores das unidades importadas (`Elem`*). A quarta linha indica a quantidade de unidades importadas indiretamente (2), e os identificadores das unidades importadas indiretamente (*`Nat` *e* `String`*). A constante inteira da linha 07 e da linha 08 indicam respectivamente a quantidade de nós da unidade (3) e a posição da descrição do nó raiz (1), respectivamente. As linhas que seguem, da nona linha até o final do arquivo, contêm as descrições dos nós, cada linha é uma descrição de nó.*

BGF - Binary AGraph Format Um formato textual AGraph é dividido no cabeçalho e nas descrições de nó. O cabeçalho não possui um padrão que favoreça a compactação, diferentemente das descrições dos nós. Logo, as informações do cabeçalho são apenas codificadas em binário, pertencendo ao cabeçalho do arquivo no BGF, e as descrições dos nós são compactados e codificados.

A Figura 3.9 relaciona a estrutura de um arquivo no formato textual AGraph e a estrutura do arquivo correspondente no formato binário AGraph (BGF).

Nem todas as descrições dos nós são compactadas dessa maneira, existe pelo menos uma descrição que é um nó referência não compactado. Todas as descrições não compactadas, incluindo as descrições que não são referênciadas, são convertidas em binário e colocadas no cabeçalho do arquivo em BGF. O resto do arquivo é formado da compactação das descrições dos nós (seção 4.2.1).

É importante salientar que as constantes inteiras das descrições, que aparecem muito nas descrições pela própria natureza do arquivo, possuem uma otimização e não são meramente codificadas para binário. Para otimizar o espaço de representação, associa-se alguns bits à descrição codificada do nó e/ou antes de cada constante inteira [2].

[1] Uma unidade é importada indiretamente quando existe uma unidade importada que a importa.

[2] No cabeçalho do arquivo, alguns valores inteiros também sofrem esta otimização.

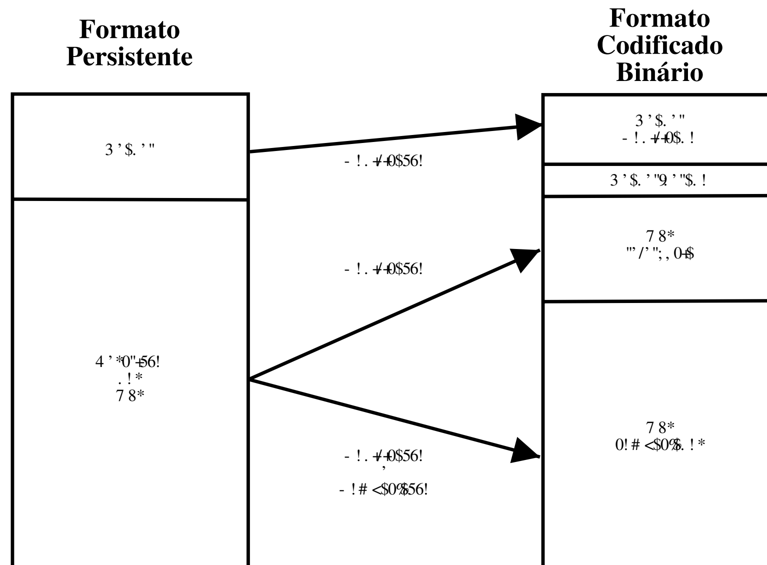


Figura 3.9: Relação entre a organização do formato textual AGraph e do formato codificado (BGF)

Esta otimização é regida pelas seguintes regras:

- se todas as constantes inteiras da descrição do nó são representadas pela mesma capacidade de bits, então, somente uma indicação da capacidade necessária é colocada no início da descrição do nó codificado,

- caso contrário, no início da descrição do nó é colocada uma indicação informando que antes de cada constante inteira existe uma indicação da capacidade necessária para esta constante inteira.

A indicação da capacidade necessária de representação da constante inteira é composta por no máximo 2 (dois) bits.

Na implementação do método de compactação testes foram realizados, comprovando que o benefício devido ao uso dos bits de determinação do limite das constantes inteiras aumenta com o tamanho do arquivo, o que compensa o *overhead* gerado.

3.4 AGraph como formato de transferência

AGraph possui dois formatos persistentes: o formato textual e o binário, os quais mantêm a integridade das representações das informações armazenadas no formato interno. O compartilhamento dos arquivos no formato persistente permite o uso de AGraph como formato de transferência.

O objeto da transferência é um arquivo no formato persistente contendo a representação persistente de uma unidade. Entretanto, para ocorrer a transferência, é necessária a compatibilidade semântica entre os formatos persistentes das ferramentas. Para existir a compatibilidade semântica, as bibliotecas AGraphs usadas devem possuir esquemas idênticos, garantindo que os formatos persistentes das bibliotecas AGraphs envolvidas sejam compatíveis. Logo, um arquivo construído por uma biblioteca pode ser lido pela outra biblioteca AGraph e vice-versa.

A figura 3.10 expõe um exemplo de processo de transferência de informações usando AGraph como formato de transferência entre as ferramentas A e B.

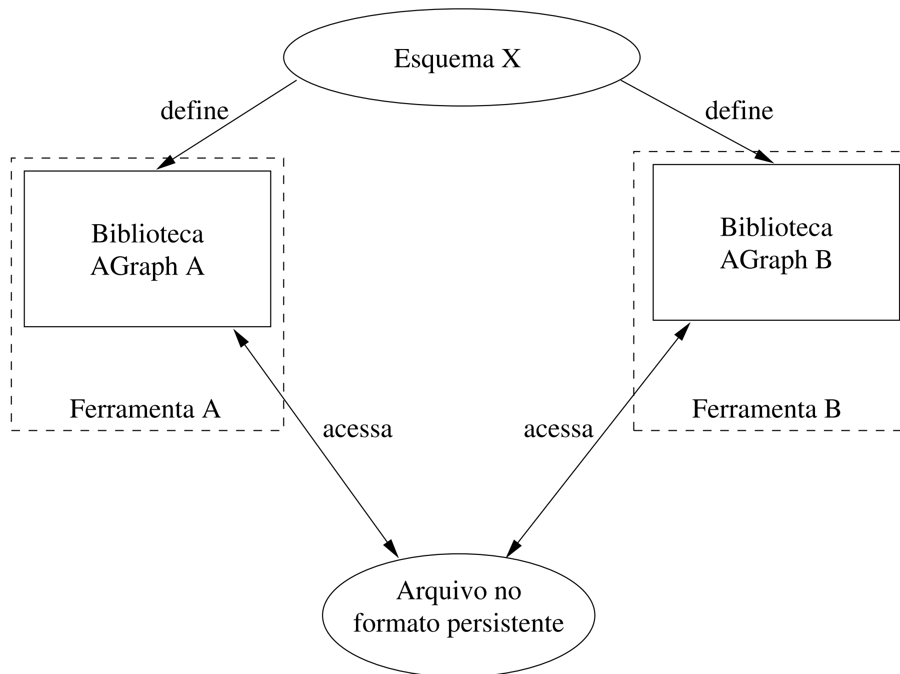


Figura 3.10: Exemplo de transferência de informações usando AGraphs.

Na figura, um arquivo no formato persistente pode ser construído e acessado pelas bibliotecas AGraph A e B. Esta transferência de dados ocorre devido a compatibildade semântica entre as bibliotecas que foram definidas pelo mesmo esquema, Esquema X.

Uma consideração importante é que a compatibilidade semântica pode ocorrer, inclusive, entre ferramentas que foram implementadas em linguagens diferentes. Utilizando o exemplo anterior, a biblioteca A pode ser implementada em C, e a biblioteca B implementada em JAVA, mas isto não impede a compatibilidade semântica.

Na seção seguinte, uma classificação para AGraph como formato de transferência é fornecida.

3.4.1 Classificação dos AGraphs

Segundo as características descritas em [13] e revisadas na seção 2.1, AGraph como formato de transferência de dados possui as seguintes classificações:

- **Sintaxe abstrata**: A sintaxe abstrata de AGraph é grafo. Estes grafos possuem nós tipados e atribuídos. A sintaxe abstrata possui uma simulação de grafos hierárquicos, e sem o conceito de herança. A definição de grafos hierárquicos é associada à definição de grafos aninhados. Um grafo é aninhado quando sua raiz é referenciada por um vértice de outro grafo. Logo, pode-se associar uma relação entre os grafos aninhados, onde os grafos mais externos possuem maior nível hierárquico. A classe de nós importação permite a implementação da simulação do conceito de hierárquica entre grafos. Os grafos importados possuem menor nível hieráquico do que os grafos importadores.

Esta simulação de hierarquia entre grafos é da perspectiva da estrutura de dados. Entretanto, relacionado ao contexto da representação de programas (ou especificações), uma importação é a indicação do reuso de componentes de outros programas (ou especificações).

- **Nível de abstração**: Um grafo mantém informações e as relações entre estas. Em qualquer nível de abstração (alto, médio ou baixo), as informações e suas relações são representadas. Portanto, o nível de abstração de AGraph não é restrito, mas depende do modo como as informações são modeladas, ou seja, depende do esquema.

- **Codificação**: AGraphs disponibiliza as duas maneiras de codificação: formato textual e binário. A função de escrever unidade fornecida pela API AGraph cria um arquivo no formato textual para a unidade alvo. Para criar um arquivo no formato binário é realizada a compactação binária (veja seção 4.2) sobre o arquivo correspondente no formato textual.

- **Mecanismo de transferência**: O desenvolvimento da biblioteca AGraph não visava a transferência de informações entre aplicações. Por este motivo, o mecanismo de transferência é o mais simples, usando arquivos. A princípio, não existe nenhum impedimento para o desenvolvimento de outros mecanismos, visto que o mecanismo de transferência não é intrínseco ao formato.

- **Tipo de Esquemas**: Uma implementação do formato AGraph possui esquema fixo. Entretanto, uma técnica de geração automática de AGraphs (veja seção 4.1) foi desenvolvida, e cria a API para uma linguagem representada, possibilitando uma diversidade de esquemas.

Segundo a classificação dos formatos de transferência em [13], pode-se classificar o esquema AGraph como: (a) *implícito*, o esquema é definido pelo contexto da ferramenta que usa a biblioteca AGraph correspondente, e (b) *interno*, o esquema é parte integrante da biblioteca AGraph.

Entretanto, existe a geração automática que usa o esquema de um AGraph para construir a biblioteca correspondente. Assim, com a geração automática classifica-se os AGraphs como um formato que possui um esquema *explícito* e *externo*, devido à existência da descrição do esquema (definição explícita) que é fornecido ao gerador automático (localização externa), ou seja, externo à biblioteca AGraph.

Capítulo 4

Ferramentas

Neste capítulo, as ferramentas de suporte aos AGraphs são descritas. Inicia-se com as descrições das modalidades de geração e dos módulos implementados no gerador automático, na seção 4.1. Em seguida, na seção 4.2, o método de compactação é fornecido.

4.1 Gerador automático

O uso das funções da API AGraph é intuitivo, permitindo uma fácil programação. Esta qualidade provoca, como consequência, a dependência da biblioteca de manipulação com relação a linguagem representada. Logo, para cada linguagem representada é necessária a implementação da biblioteca AGraph correspondente.

A implementação de uma nova instância da biblioteca é uma tarefa árdua, contudo, possui um padrão de formação. Este padrão de formação é fortemente relacionado à estrutura do grafo, ou seja, associado ao esquema do grafo.

Para eliminar o esforço de desenvolvimento de uma instância de biblioteca AGraph para cada linguagem representada foi desenvolvida uma ferramenta de geração automática de bibliotecas AGraphs.

A ferramenta possibilita a geração das bibliotecas usando duas modalidades:

- baseada na descrição do esquema AGraph usando uma especificação customizada; ou,
- diretamente da definição da sintaxe da linguagem representada.

Nas seções 4.1.1 e 4.1.2 são descritas as modalidades de geração customizada e a geração a partir da sintaxe, respectivamente.

4.1.1 Geração customizada

O conjunto de funções para qualquer biblioteca AGraph segue o padrão descrito na seção 3.3.3, sendo que as implementações diferenciam entre si devido à estrutura do grafo.

A primeira modalidade de geração usa a descrição da estrutura do grafo e algumas informações adicionais para gerar a biblioteca AGraph correspondente. Estas informações, a estrutura do grafo e as informações adicionais são fornecidas por uma sublinguagem XML simples, definido exclusivamente para o gerador e denominado *formato customizado*.

A descrição do formato customizado é fornecida na próxima seção. No exemplo 4.1, um esquema para uma sublinguagem da linguagem ELAN [3, 2] é descrita no formato customizado.

Especificação do formato customizado

O formato customizado é composto por elementos XML, identificadas por *tags*. Cada *tag* possui informações correspondentes aos elementos que compõe os esquemas AGraphs, elementos estes definidos na seção 3.2.3. Existem 9 categorias de *tags*: `Unit`, `DataType`, `Attribute`, `Edge`, `Hyperedge`, `Node`, `RootNode`, `Import` e `List`. Nesta seção são descritas as informações que compõem cada categoria de *tags*.

O elemento identificado pela *tag* `Unit` delimita a especificação do esquema AGraph, e é único em cada arquivo de descrição. Este elemento possui dois parâmetros: `lang` e `root`. `lang` é um identificador da linguagem representada, e `root` é o identificador da categoria dos nós raiz.

Como exemplo, o trecho a seguir expõe uma atribuição aos parâmetros. Ao parâmetro `lang` , é associada a etiqueta “Elan”, e ao parâmetro `root` é associado o identificador “ModuleDef”.

```
<Unit lang="Elan" root="ModuleDef">
  ...
</Unit>
```

O elemento `Unit` é formado por um conjunto de outros elementos identificados pelas *tags*: `DataTypes`, `RootNode`, `Import`, `List`, e `Node`. Os primeiros elementos definidos em `Unit` são os `DataTypes`, seguidos por somente um elemento `RootNode` e sequências de `Import`, `List`, e `Node`.

Cada elemento `DataType` declara um tipo de dados denominado enumerado. Este tipo de dado é a associação de um conjunto de valores possíveis a um identificador de tipo. Os valores possíveis são listados no parâmetro `value` e o identificador no parâmetro `name`. No exemplo

```
<DataType name="Color" value="Black|Red" />
```

o elemento `DataType` define um tipo identificado por `Color`, cujo os atributos assumem os valores `Black` ou `Red`.

Observe que os valores no parâmetro `value` são separados por um barra vertical (“`|`”). Na descrição do esquema AGraph, quando um parâmetro é associado à vários valores, estes são separados pelo caractere “`|`”.

Os elementos que descrevem as categorias de nó são elementos compostos formados por elementos simples que definem os atributos, as arestas e as hiperarestas daquela categoria de nós.

Um atributo é definido pelo elemento identificado pela *tag* `Attribute`. O elemento `Attribute` associa um nome a um tipo pré-definido no gerador ou enumerado, onde o nome é definido pelo parâmetro `name` e o tipo pelo parâmetro `type`.

Os tipos pré-definidos foram definidos a partir das linguagens em que o gerador foi implementado (C e JAVA). Na implementação atual do gerador, os valores possíveis para o parâmetro `type` dos elementos da categoria `Attribute` são `int` (constante inteira),

`string` (conjunto de caracteres) e o identificador (parâmetro `name`) de algum elemento `DataType`.

No exemplo

```
<Attribute name="aIdentifier" type="string"/>
```

o atributo declarado é identificado pelo nome `aIdentifier` e é do tipo `string`.

O elemento `Edge` declara uma aresta e possui dois parâmetros. O primeiro parâmetro, `name`, é o identificador da aresta, e o segundo parâmetro, `type`, informa as categorias possíveis do nó alvo.

O exemplo

```
<Edge name="nLeftExpression" type="Variable|Operation"/>
```

declara uma aresta identificada por `nLeftExpression`. Esta aresta apontará para nós da categoria identificada por `Variable` ou `Operation`.

As hiperarestas são declaradas pelo elementos `Hyperedge`. Os elementos `Hyperedge` possuem dois parâmetros: `name` é o identificador da hiperaresta, e, `type` é um identificador de alguma categoria de nós do tipo lista, descritos a seguir. Como exemplo, o elemento

```
<Hyperedge name="nImports" type="oneList" />
```

declara uma hiperaresta identificada por `nImports`, e `oneList` é a categoria dos nós apontados por esta.

Um AGraph possui três categorias de nós especiais, como descrito na seção 3.2.3.

A fim de simular uma lista encadeada de nós e permitir a declaração de hiperarestas, existe o elemento identificado pela *tag* `List`. O elemento `List` possui quatro parâmetros: (1) o identificador da categoria informado pelo `name`, (2) a aresta que aponta para o valor corrente, `value`, (3) a hiperaresta para o próximo nó na lista, `next`, e, (4) a hiperaresta para o nó anterior na lista, `previous`.

No exemplo

```
<List name="oneList"
      next="nNext"
      previous="nPrevious"
      value="nValue" >
  <Edge name="nValue" type="elanNode" />
</List>
```

existe a definição de um tipo de nó lista que apontará para os nós do tipo `elanNode`. Assume-se que o uso do identificador `langNode`, para uma linguagem com etiqueta `lang`, permite que o nó alvo da aresta (ou hiperaresta) seja qualquer tipo de nó definido no esquema. No exemplo anterior, o identificador para qualquer nó definido é `elanNode`.

Em toda declaração do tipo de nó lista, não é necessária a declaração das hiperarestas `next` e `previous`, respectivamente, `nNext` e `nPrevious`. Esta ausÊncia de declaração destes elementos ocorre devido aos perfis destas hiperarestas serem implicitamente declarados, alterando somente os seus identificadores entre definições de listas.

A categoria dos nós importação é outra categoria especial, sendo definida pelo elemento identificado pela *tag* `Import`. Este elemento possui três parâmetros: `name` é o identificador do tipo de nó, `importIdentifier` indica o atributo que mantém o identificador da unidade importada, e, `rootIdentifier` indica a aresta que aponta para o nó raiz do grafo correspondendo à unidade importada.

No exemplo

```
<Import name="Includes"
        importIdentifier="aIdentifier"
        rootIdentifier="nRootImport">
  <Attribute name="aIdentifier" type="string" />
  <Edge name="nRootImport" type="ModuleDef" />
</Import>
```

um tipo de nó importação identificado por `Includes` é declarado. Os nós do tipo `Includes` possuem o atributo `aIdentifier` que mantém o identificador da unidade importada, e a aresta `nRootImport` que aponta para o nó raiz da unidade importada, que, no exemplo anterior, deve ser um nó do tipo `ModuleDef`.

A categoria de nós raiz é a terceira categoria de nós especiais. `RootNode` é a *tag* do elemento que define um tipo de nó desta categoria. Este elemento possui três parâmetros: (a) `name`, identificador do tipo de nó, (b) `rootIdentifier`, nome do atributo que mantém o identificador do nó raiz, e, (c) `listImports`, nome da hiperaresta que apontará para os nós raiz das unidades importadas. No exemplo

```
<RootNode name="ModuleDef"
          rootIdentifier="aIdentifier"
          listImports="nIncludes">
  <Attribute name="aIdentifier" type="string" />
  <Hyperedge name="nIncludes" type="IncludesList" />
  ...
</RootNode>
```

o tipo de nó lista usado para a importação deverá possuir uma aresta que aponte para nós da categoria importação. Por exemplo, na declaração do elemento `IncludesList`, deve existir uma aresta que aponte para nós da categoria nós importação. Se esta aresta não existir, a descrição não está bem formada e as informações a respeito das importações não serão mantidas.

Para a definição de outras categorias que não pertençam às categorias de nós especiais existe o elemento `Node`. O exemplo

```
<Node name="AxiomDef">
  <Attribute name="aNumber" type="int" />
  <Edge name="nLeftExpression" type="Variable|Operation" />
  <Edge name="nRightExpression" type="Variable|Operation" />
</Node>
```

define um tipo de nó denominado `AxiomDef` que possui um atributo `aNumber` do tipo `int`, e duas arestas, `nLeftExpression` e `nRightExpression`, que podem apontar para nós do tipo `Variable` ou `Operation`.

O *resultado da geração* é uma biblioteca para o AGraphs descrito por um arquivo no formato customizado. Esta biblioteca gerada segue o padrão descrito na seção 3.3. Logo:

- na biblioteca existe uma função de inicialização, leitura e escrita de uma unidade.
- para cada tipo de nó é gerada uma função de construção de um nó (`make`).
- para cada atributo, aresta e hiperaresta uma função `set` e `query`.

Além dessas funções, existem outras funções, as funções auxiliares. As funções auxiliares não podem ser executadas pelos usuários. Como exemplo de funções auxiliares, pode-se citar as funções de leitura e escritas de um nó, que não podem ser acessadas pelo usuário e são chamadas na execução da função de leitura e escrita de uma unidade.

A distribuição das estruturas e funções nos módulos que compõem a biblioteca depende da linguagem de programação usada na implementação: C ou JAVA. Na descrição a seguir, supõe-se que a etiqueta da linguagem representada seja `lang`.

Em C, as declarações das estruturas de dados e das funções principais estão contidas no arquivo *lang.h*, as definições das estruturas de dados e dos tipos enumerados se localizam em *langInt.h* e as implementações das funções principais e suas funções auxiliares são agrupadas em três arquivos diferentes: (a) *make.c* para as funções `make` para todos os tipos de nó, (b) *set.c* para as funções `set`, e (c) *query.c* para as funções `query` de todos os atributos, arestas e hiperarestas.

Existem outros arquivos:

- *write.c/read.c*: com as funções de escrita/leitura para cada nó.
- *translate.c*: com algumas funções auxiliares executadas na leitura e escrita de unidades.
- *manager.c*: contém as funções de inicialização da biblioteca e leitura (resp. escrita) de uma unidade.

JAVA é uma linguagem de programação orientada a objetos. Consequentemente, a disposição dos arquivos que contém a estrutura de dados e as funções da biblioteca AGraph é diferente da implementação em C.

Em JAVA, existe:

- uma classe para gerenciar a biblioteca. Esta classe é responsável pela inicialização, leitura e escrita de unidades.
- uma classe para cada tipo de nó. Uma classe de um tipo de nó contém informações sobre os seus nós e a implementação das funções de criação, e das funções `query` e `set` para cada atributo, aresta e hiperaresta deste tipo de nó.
- uma classe para cada definição de tipo enumerado, descrevendo os possíveis valores que os elementos deste tipo podem assumir.

Exemplo 4.1 *Fornece-se um exemplo completo de geração automática usando o modo de geração customizado para uma sublinguagem ELAN. Antes de iniciar-se as definições dos tipos de nós do AGraph, é necessário uma descrição superficial desta sublinguagem ELAN.*

Uma unidade desta sublinguagem ELAN é dividida em três partes: uma denominada `export`*, outra denominada* `hidden` *e um conjunto de definições de axiomas.*

`export` *e* `hidden` *são compostas por um conjunto de definições de* `sorts`*, operadores prefixados e variáveis. As variavéis são utilizadas nas definições de axiomas.*

Os axiomas são expressões incondicionais formadas por aplicação de operadores (operações), e são necessárias para determinar as características desses operadores.

A importação é permitida nesta sublinguagem. Os elementos declarados na parte `export` *podem ser utilizados por outras unidades, ao contrário com o que ocorre na parte* `hidden`.

Existem várias possibilidades de descrições de um esquema AGraph para uma linguagem. Uma possível descrição do esquema AGraph para esta sublinguagem ELAN inicia com a tag `Unit`*, indicando o identificador da linguagem (*`elan`*), e o identificador da classe de nós raiz (*`ModuleDef`*).*

```
<Unit lang="elan" root="ModuleDef">
```

A classe de nós `ModuleDef` *contém o identificador* `string` *(*`aIdentifier`*), uma lista de unidades importadas (*`nImports`*), uma referência para os itens exportados (resp. importados) em* `nExports` *(resp.* `nHiddens`*), e uma lista de definições de axiomas (*`nAxiomDefs`*).*

```
<RootNode name="ModuleDef"
          rootIdentifier="aIdentifier"
          listImports="nImports">
  <Attribute name="aIdentifier" type="string" />
  <Hyperedge name="nImports" type="oneList" />
  <Edge name="nExports" type="DefSection" />
  <Edge name="nHiddens" type="DefSection" />
  <Hyperedge name="nAxiomDefs" type="oneList" />
</RootNode>
```

Conjuntos de sortes (`nSortDefs`*), operadores (*`nOperatorDefs`*) e variáveis (*`nVariableDefs`*) são declarados nas seções* `exports` *e* `hiddens` *(*`DefSection`*). Declarações de sortes (*`SortDef`*) possuem um identificador (*`aIdentifier`*); declarações de operadores (*`OperatorDef`*) possuem um identificador (*`aIdentifier`*), um domínio (*`nDomain`*) e uma imagem (*`nCodomain`*); e, as declarações de variáveis (*`VariableDef`*) contém um identificador (*`aIdentifier`*) e um sorte (*`nSort`*).*

```
<Node name="DefSection">
  <Hyperedge name="nSortDefs" type="oneList" />
  <Hyperedge name="nOperatorDefs" type="oneList" />
  <Hyperedge name="nVariableDefs" type="oneList" />
</Node>
<Node name="SortDef">
  <Attribute name="aIdentifier" type="string" />
</Node>
<Node name="OperatorDef">
  <Attribute name="aIdentifier" type="string" />
  <Hyperedge name="nDomain" type="oneList" />
  <Edge name="nCodomain" type="SortDef" />
</Node>
<Node name="VariableDef">
  <Attribute name="aIdentifier" type="string" />
  <Edge name="nSort" type="SortDef" />
</Node>
```

Existem as definições de axiomas incondicionais (`AxiomDef`*), cujas expressões (*`nLeftExpression` *e* `nRightExpression`*) são operações recursivamente compostas por variáveis e/ou operações (*`VariableInst` *e* `Operation`*).* `VariableInst` *são as instâncias das variáveis definidas (*`nVariable`*), enquanto,* `Operation` *são as aplicações dos operadores (*`nOperator`*) a uma lista de argumentos (*`nArguments`*).*

```
<Node name="AxiomDef">
  <Attribute name="aNumber" type="int" />
  <Edge name="nLeftExpression"
        type="VariableInst|Operation" />
  <Edge name="nRightExpression"
        type="VariableInst|Operation" />
</Node>
<Node name="Operation">
  <Edge name="nOperator"
        type="OperatorDef" />
  <Hyperedge name="nArguments"
             type="oneList" />
</Node>
<Node name="VariableInst">
  <Edge name="nVariable"
        type="VariableDef" />
</Node>
```

Com AGraphs, diferentes mecanismos de importação são possíveis com a definição de diferentes classes de nós importação. Neste exemplo, definimos um tipo de importação (`Import`*), com um identificador (*`aIdentifier`*) e uma aresta para o nó raiz da unidade importada (*`nRootImport`*).*

```
<Import name="Includes"
        importIdentifier="aIdentifier"
        rootImport="nRootImport">
  <Attribute name="aIdentifier" type="string" />
  <Edge name="nRootImport" type="ModuleDef" />
</Import>
```

Múltiplas classes de nós listas podem ser definidas. Contudo, defini-se uma classe de nós lista básica (`oneList`*), contendo uma aresta para qualquer classe de nós definidos (*`nValue`*).*

```
 <List name="oneList"
       next="nNext"
       previous="nPrevious"
       value="nValue">
  <Edge name="nValue" type="elanNode" />
 </List>
</Unit>
```

Observe que na linguagem representada pelo esquema AGraph, é possível a importação. Logo, é necessária a definição de pelo menos uma classe de nós lista, cujo identificador é parâmetro na tag `RootNode`.

Neste exemplo, a tag `Datatype` *não foi utilizada. Mas se necessário, um tipo de dados seria definido. Por exemplo, a seguir, existe a definição de um tipo, identificado por* `aKind` *podendo assumir dois valores:* `Loose` *ou* `Complex`.

```
<DataType name="aKind" value="Loose|Complex" />
```

4.1.2 Geração a partir da descrição da sintaxe

A outra modalidade de geração automática é baseada na conexão direta entre a sintaxe concreta e o esquema AGraph correspondente da linguagem alvo. A sintaxe concreta é descrita em um formalismo específico para a descrições de linguagens, SDF2 [27, 26]. Na seção seguir, SDF2 é brevemente descrita.

SDF2

SDF2 é um formalismo de definição de sintaxe que permite a definição modular, com as construções léxicas e sintáticas integradas.

Os arquivos na linguagem SDF2 são formados por três partes:

- `export` declaração dos sortes usados nas regras.
- `lexical syntax` definição dos elementos léxicos.
- `context-free syntax` conjunto de regras para construção da sintaxe.

```
modules Modules

exports
 sorts Module
lexical syntax
 [A-Z][a-zA-Z\-\_]+ -> ModuleName
 [a-z]+ -> SortName
 [\ \t\n] -> LAYOUT
context-free syntax
 "module" Identifier:ModuleName
 "sorts" SortDefs:{SortDef ","}* ";"
 "imports" ImportDefs:{ImportDef ","}* ";"
 "end"                                                  -> Module

 Identifier:SortName                                    -> SortDef

 Identifier:ModuleName            -> ImportDef {const("import")}
```

Figura 4.1: Exemplo de definição SDF2 da sintaxe de uma linguagem de especificação fictícia.

A figura 4.1 contém um exemplo do formalismo SDF2. Neste exemplo, define-se a sintaxe da linguagem de especificação fictícia denominada `Modules`.

No exemplo, a linguagem fictícia `Modules` é formada por um conjunto de identificadores de especificações importadas e de declaração de sortes.

Descrição

Esta modalidade permite:

- o uso da sintaxe concreta como documentação para o formato AGraph,
- desenvolvimento rápido de novas instâncias de bibliotecas AGraphs, e,
- eficiência nas atualizações na biblioteca provocada por atualizações na linguagem.

Para a conexão direta entre a sintaxe concreta e o esquema do grafo, foi definido um mapeamento. Este mapeamento é descrito na seção 4.1.3.

A implementação desta modalidade de geração usa ferramentas de mapeamento. Estas ferramentas mapeiam as definições da sintaxe concreta para um tipo de dados abstrato [8],

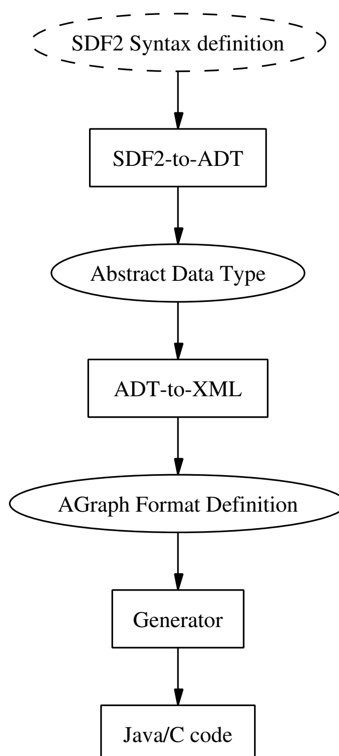


Figura 4.2: Processo de construção da biblioteca AGraph a partir da sintaxe concreta.

e, a partir deste tipo de dados, é construído o esquema AGraph no formato descrito na seção 4.1.1.

A figura 4.2 contém o processo de construção da biblioteca AGraph a partir da sintaxe concreta de uma linguagem.

A entrada para o gerador é a descrição da sintaxe concreta em SDF2 (SDF2 Syntax definition). A partir da sintaxe, um arquivo contendo a sintaxe abstrata é construído. Esta sintaxe abstrata é descrita por um tipo de dados abstratos, denominado *Abstract Data Type Definition* (ADT). Os arquivos no formatos ADT são construídos pela ferramenta Sdf2toAdt que pertence ao conjunto de ferramentas de suporte disponibilizado para o formato SDF2.

Em seguida, o arquivo ADT é traduzido para o formato de definição do AGraph (AGraph Definition Format), denominado formato *customizado* (seção 4.1.1). Esta tradução ocorre de uma estrutura orientada à árvore (ADT) para uma orientada à grafos (esquema AGraph). E, finalmente, a definição AGraph no formato *customizado* é usada para construir a biblioteca seguindo as regras de geração customizada descritas na seção 4.1.1.

Exemplo 4.2 *Na figura 4.3, é fornecido o diagrama de um exemplo de aplicação da geração automática a partir da sintaxe. A figura expõe o processo de construção de uma representação AGraph para uma unidade de entrada.*

Os arquivos de entrada são delimitados por linhas tracejadas. A definição da sintaxe da linguagem alvo é uma entrada, por exemplo a sintaxe de CASL, e a outra entrada é uma unidade na linguagem alvo, uma especificação em CASL, por exemplo. A partir da sintaxe, um parser para as unidades da linguagem alvo é construído. A especificação é analisada pelo parser, construído automaticamente, resultando em um arquivo com a

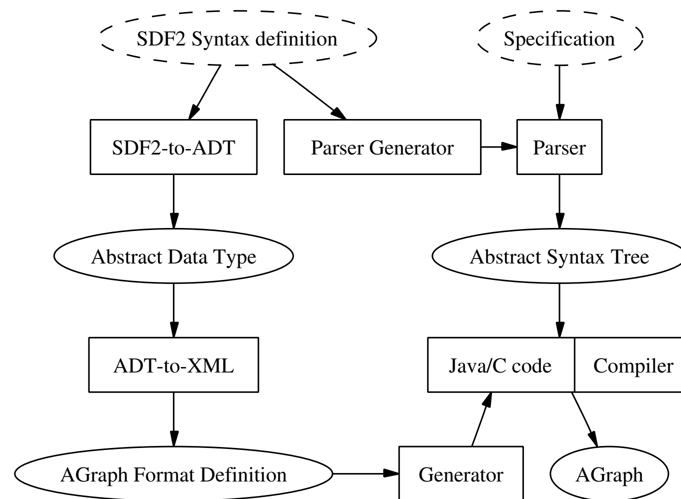


Figura 4.3: Exemplo de mapeamento da definição da sintaxe, via definições abstratas, para a geração do código.

sintaxe abstrata da especificação CASL no formato árvore (Abstract Syntax Tree).

Finalizando o processo, o compilador, implementado com as funções da biblioteca AGraph que foi gerada automaticamente, cria a representação AGraph a partir da sintaxe abstrata.

Na próxima seção, os padrões de mapeamento são definidos.

4.1.3 Mapeamento automático

Em uma especificação SDF2, somente o conjunto de regras (`context-free syntax`) é significante para o mapeamento. Cada regra define um tipo de nó no esquema AGraph, declarando os seus atributos, arestas e hiperarestas.

Devido à rica expressividade do formalismo SDF2 versus a simplicidade do formato ADT e as informações necessárias para um esquema AGraph, foram definidas as seguintes restrições sobre a descrição SDF2 original:

- o nome do arquivo define a etiqueta para a biblioteca da linguagem.
- o identificador `name` é palavra reservada.
- todos os elementos definidos em uma regra (atributos, arestas e hiperarestas) devem possuir um identificador fornecido antes dos *dois pontos* (“`:`”).
- `Number` é uma palavra reservada e na construção de uma regra significa a declaração de um atributo do tipo inteiro.
- a primeira regra sempre define um tipo de nó raiz, `RootNode`.
- todo identificador do nó raiz e do nó importação deve ser `Identifier`.
- a regra que define um tipo de nó importação é seguida pela expressão:

```
{const("import")}
```

Exemplo 4.3 *Para auxiliar na descrição dos padrões de mapeamento automático, é usada a primeira regra do SDF2 da figura 4.1, identificada por* `Module`*. O exemplo de mapeamento está localizado na figura 4.4*

Nesta figura, as informações conectadas são relacionadas. Por exemplo, no trecho `SortDefs:(SortDef ",")*`*, existe a declaração de uma hiperaresta identificada por* `SortDefs`*, pois, a formação* `(SortDef ",")*` *associa o tipo de nó definido a um conjunto de nós do tipo* `SortDef` *separados por vírgula (“*`,`*”), o que caracteriza uma hiperaresta (associação de um nó à vários nós).*

Em outro trecho, a construção `Identifier:ModuleName` *declara um atributo do tipo* `string`*, pois* `ModuleName` *não é o nome de uma regra no arquivo SDF2, sendo o nome deste atributo* `aIdentifier`*, onde o caracter “*`a`*” é devido ao fato de ser um atributo, e* `Identifier` *pelo identificador da formação.*

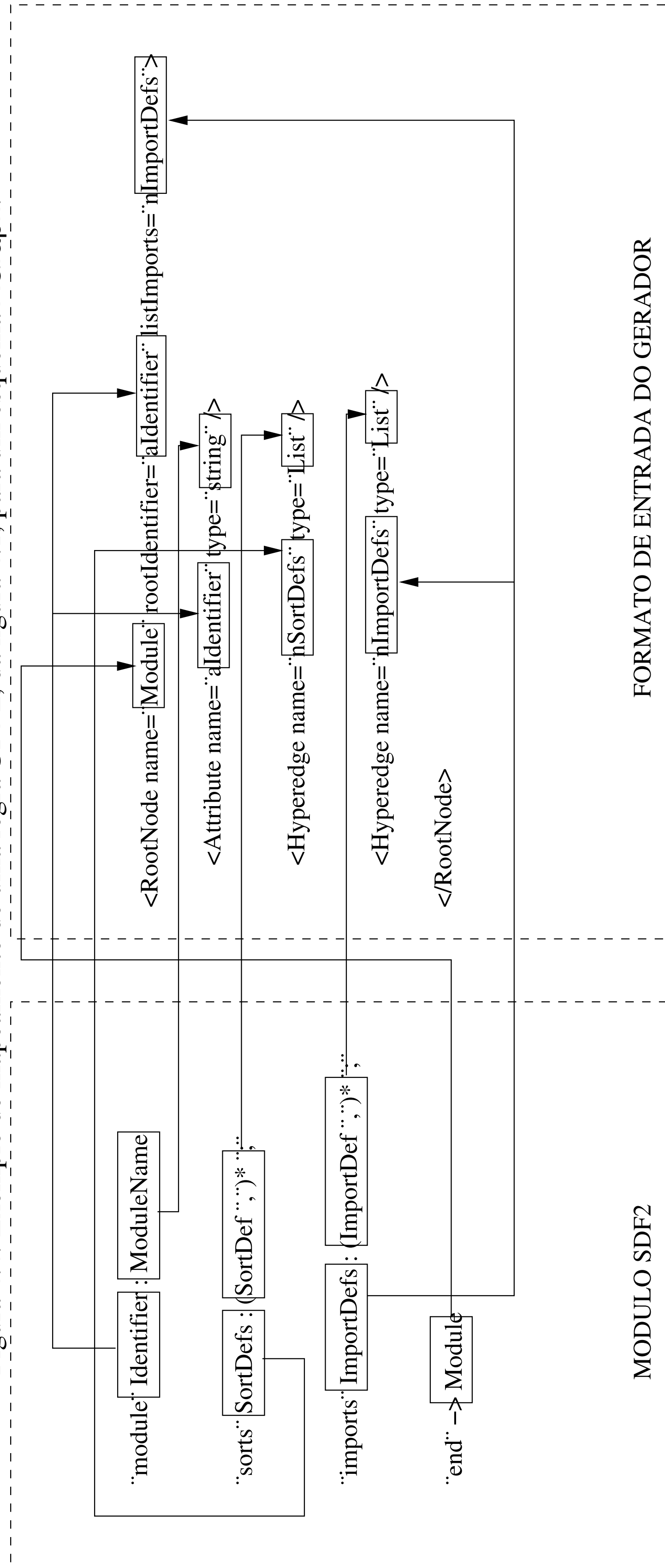


Figura 4.4: Exemplo de mapeamento de uma regra SDF2, da figura 4.1, para um esquema AGraph.

Deste exemplo, pode se observar que uma hiperaresta é declarada pelo seguinte padrão SDF2:

```
{<tipo_do_nó> [<separador>]}*
```

onde: `<tipo_do_nó>` é o identificador de um tipo declarado de nó, e, `<separador>` é um caractere, o qual é opcional.

Quando um padrão de declaração de hiperaresta ocorre pela primeira vez em qualquer regra SDF2, um tipo de nó lista é criado no esquema AGraph. Este tipo lista é identificado por `oneList` e possui a seguinte estrutura:

```
<List name="oneList"
      next="nNext"
      previous="nPrevious"
      value="nValue">
  <Edge name="nValue" type="ModulesNode" />
</List>
```

O valor `ModulesNode` do parâmetro `type` informa que a aresta `nValue` apontará para qualquer tipo de nó.

Neste exemplo, não existe a declaração de um atributo inteiro, mas o valor `Number` como tipo para um elemento em uma regra informa que o atributo é do tipo constante inteira. Como exemplo, a parte de uma regra SDF2 que contém

```
... aNumber:Number ... -> Numerical
```

declara um atributo do tipo constante inteira em `Numerical`. Logo, no arquivo de definição AGraph é gerado o código a seguir:

```
<Node name="Numerical">
  ...
  <Attribute name="aNumber" type="int" />
  ...
</Node>
```

Exemplo 4.4 *A seguir é fornecida a descrição da sintaxe de uma linguagem fictícia, e, posteriormente, a respectiva descrição da estrutura AGraph no formato permitido à geração customizada, com alguns breves comentários.*

```
modules Modules

exports
 sorts Module SortDef ImportDef
lexical syntax
 [A-Z][a-zA-Z\-\_]+ -> ModuleName
 [a-z]+ -> SortName
 [\ \t\n] -> LAYOUT

context-free syntax
 "module" Identifier:ModuleName
 "sorts" SortDefs:{SortDef ","}* ";"
 "imports" ImportDefs:{ImportDef ","}* ";"
 "end"                                       -> Module

 Identifier:SortName                         -> SortDef

 Identifier:ModuleName    -> ImportDef {const("import")}
```

Na parte `exports` *na seção* `sorts`*, estão contidas as denominações dos tipos de nós que serão geradas. Na seção* `lexical syntax` *define-se um elemento,* `LAYOUT`*, que serve para indicar os caracteres especiais, sem significado para o gerador mas indicado no manual de SDF2. Todas estas informações são ignoradas pelo gerador, mas são importantes como documentação da linguagem representada.*

```
<Unit lang="lang" root="Module">

<RootNode name="Module" rootIdentifier="aIdentifier" >
  <Attribute name="aIdentifier" type="string" />
  <Hyperedge name="nSortDefs" type="oneList" />
  <Hyperedge name="nImportDefs" type="oneList" />
</RootNode>

<Import name="ImportDef"
        importIdentifier="aidentifier"
        rootIdentifier="nRootImport">
  <Attribute name="aIdentifier" type="string" />
</Import>

<Node name="SortDef">
  <Attribute name="aIdentifier" type="string" />
</Node>

<List name="oneList"
      value="nValue"
      previous="nPrevious"
      next="nNext">
  <Hyperedge name="nNext" type="oneList" />
  <Hyperedge name="nPrevious" type="oneList" />
  <Edge name="nValue" type="langNode" />
</List>

</Unit>
```

Comparando as descrições, observa-se que uma regra que define um tipo de nó da categoria `Import` *finaliza com* `{cons("import")}`*. Na descrição do formato de entrada da geração customizada, o identificador do atributo que mantém o identificador da unidade importada (*`rootIdentifier="nRootImport"`*) e os* `Hyperedges` *dos tipos de nós* `List` *(*`nPrevious` *e* `nNext`*) são declarados automaticamente, sendo elementos padrões da estrutura* customizada.

4.2 Codificação Binária do formato AGraph

Considerando que o formato AGraph é usado na transferência e no armazenamento de dados, os arquivos na representação textual deverão possuir tamanhos aceitáveis, apesar que o formato textual AGraph é relativamente conciso se comparado, por exemplo, com o formato textual ATerm.

O tamanho dos arquivos manipulados podem influenciar no desempenho de uma aplicação que usa a biblioteca AGraphs. Para amenizar as consequências de possíveis problemas com o tamanho das descrições em arquivo, investiu-se na compactação dos arquivos no formato textual.

Um método de codificação binária para o formato textual AGraph foi implementado. Para obter um melhor desempenho, a implementação do método de codificação é associada ao esquema AGraph, assim, as funções de compactação (resp. descompactação) pertencem à biblioteca AGraph correspondente.

Na seção 4.2.1, o método de compactação é descrito seguido pelas descrições das funções de compactação/descompactação (seção 4.2.2).

4.2.1 Descrição do método

O formato persistente textual contém várias descrições de nós (veja seção 3.3.4) que diferem em alguns valores entre si. Partindo desta afirmação, a compactação é realizada entre as descrições de nós que possuem valores semelhantes, a fim de reduzir a quantidade de informações repetidas.

O principal objetivo da compactação é representar as constantes (inteiro ou `string`) diferentes, e evitar representar as constantes repetidas nas descrições envolvidas.

No processo de compactação, existe uma iteração para compactar cada descrição de nó que compõe o arquivo. Para cada descrição de nó poderá existir uma descrição de nó referência.

Uma descrição do nó referência é obtida da comparação entre a descrição a compactar e as outras descrições de nó. A descrição que possui menor quantidade de constantes diferentes é a descrição do nó referência.

Como exemplo, sejam duas descrições dos nós a seguir:

```
<1 10 3 add 0 1 0 2>
<1 10 3 add 0 2 0 4>
```

Com o primeiro servindo de nó referência, a compactação tem como resultado:

```
<1 10 3 add 0 1 0 2>
1 00000101 2 4
```

Como a primeira é a descrição do nó referência, então, esta é escrita no formato textual. A segunda descrição é codificada com relação à descrição do nó referência, cuja posição no arquivo é indicada pela primeira constante na descrição codificada (1). O valor binário que segue indica quais constantes diferem entre as descrições. Para cada constante existe um bit indicando se o valor desta difere ou não. Se o bit tem valor 1, então o valor difere, e não difere caso contrário (valor 0). No exemplo, a sexta e a oitava constantes da descrição do nó diferem. Em seguida, os valores que diferem são fornecidos, listados em ordem, da esquerda para direita.

Observa-se que para saber quantas e os tipos das constantes que compõem uma descrição de nó é necessário conhecer o tipo do nó descrito. Por este motivo, a compactação é uma funcionalidade associada ao esquema AGraph.

Generalizando, a compactação é aplicada entre várias descrições de nós. Obtendo observações sobre o método de codificação:

- um nó referência é escolhido pela comparação entre a descrição do nó a ser compactado e as outras descrições dos nós do mesmo tipo. Esta comparação gera um *overhead* que pode prejudicar o desempenho da compactação.

- nós diferentes podem possuir um mesmo nó referência.

- um nó referência pode ter um nó referência, podendo ser compactado, aumentando a taxa de compactação. Este fato não é claro, mas normalmente ocorre, como observa-se no exemplo a seguir:

```
<3 0 1 0 2 1 6>
<3 0 1 0 2 1 7>
<3 1 1 0 2 1 7>
```

  A primeira descrição de nó é referência para a segunda (diferença de 1 (uma) constante), e a segunda é referência para a terceira (diferença de 1 (uma) constante). A primeira descrição de nó não é a melhor referência para a terceira, visto que a diferença são de 2 (duas) constantes. Consequentemente, a primeira descrição de nó é referência, e não será compactada, a segunda é uma descrição de nó referência, e será compactada, e a terceira não é referência e será compactada. Obtendo-se:

```
<3 0 1 0 2 1 6>
1 00001 7
2 10000 1
```

- um nó não é compactado se não possui um nó referência associado, o qual deve conter pelo menos uma constante igual.
- na atual versão da compactação, as descrições de nó que possuem alguma `string` como elemento não são compactadas. Esta é uma tarefa para trabalhos futuros.
- adicionalmente à aplicação da compactação descrita acima, todas as informações são codificadas em binário melhorando o desempenho obtido da compactação.

Concluindo, a compactação aplicada a alguns arquivos no formato textual não garante a redução do tamanho desses. Entretanto, as unidades da maioria das linguagens de programação (ou especificação) possuem alguns elementos do mesmo tipo (por exemplo, variáveis ou operações), o que possibilita descrições semelhantes dos nós, viabilizando a aplicação da compactação, garantindo que o tamanho da representação binária seja menor que o tamanho da representação textual.

4.2.2 API da compactação/descompactação

Existem duas funções relacionadas a codificação binária: (a) compactação (`Compress`), e,(b) descompactação (`Decompress`).

A compactação é realizada em um arquivo no formato textual AGraph que representa uma unidade. O identificador da unidade é o parâmetro para a função `Compress`.

Esta função verifica se o arquivo associado ao identificador parâmetro está no diretório especificado, e então, realiza a compactação, obtendo como resultado um arquivo em BGF, com extensão `.bgf`. Caso o arquivo de entrada não seja bem formado, uma indicação de erro é retornada, informando que não foi possível realizar a compactação.

A função `Decompress` possui como parâmetro o nome do arquivo contendo a codificação binária AGraph da representação de uma unidade. Se for verificada a existência deste arquivo, e se nenhum problema de leitura do arquivo for detectado, um arquivo no formato textual AGraph correspondendo à unidade codificada é gerado.

Sigla	Significado
.casl	Formato CASL
.agraph	Formato persistente textual AGraph
.bgf	Formato BGF
.gz	Formato GZIP
.bgf.gz	Formato BGF_GZIP
B/F	Relação entre os formatos .bgf e o formato persistente textual AGraph
G/F	Relação entre os formatos .gz e o formato persistente textual AGraph
BG/B	Relação entre os formatos .bgf.gz e .bgf
BG/G	Relação entre os formatos .bgz.gz e .gz

Tabela 4.1: Legenda para as tabelas 4.2 e 4.3.

4.2.3 Resultados

Para verificar o desempenho do método de compactação, foram realizados vários testes com arquivos representando especificações exemplos na linguagem CASL. A ferramenta de compactação do formato AGraphs (seção 4.2) foi comparada com a ferramenta de compactação GNU `gzip`[1], e estas foram combinadas, proporcionando uma comparação entre as diversas combinações de arquivos gerados.

A tabela 4.1 fornece a legenda para os diversos formatos criados com a combinação das aplicações das ferramentas de compactação do formato textual AGraphs e gzip.

O formato denominado BGF_GZIP é a aplicação da ferramenta gzip nos arquivos do formato BGF. Os arquivos com o formato .gz são gerados da aplicação da ferramenta `gzip` nos arquivos no formato persistente textual AGraphs.

Nas tabelas 4.2 e 4.3, os tamanhos dos vários formatos para cada exemplo são fornecidos, assim como as relações entre eles.

[1]Esta ferramenta implementa o algoritmo de codificação Lempel-Ziv (LZ77) [31, 32].

Tabela 4.2: Análise do método de codificação binária

Arquivo	.casl	.agraph	.bgf	.gz	.bgf.gz	**B/F**	**G/F**	**BG/F**	**BG/B**	**BG/G**
	Bytes (B)					**Porcento (%)**				
Big10	2,8k	12000	3600	2800	3200	30	23,33	26,67	88,89	114,29
Big20	11k	45000	15000	10000	12000	33,33	22,22	26,67	80	120
Big30	24k	102000	34000	23000	27000	33,33	22,55	26,47	79,41	117,39
Big40	43k	180000	60000	41000	46000	33,33	22,78	25,56	76,67	112,2
BnatTree	326	988	304	360	346	30,77	36,44	35,02	113,82	96,11
Bool	557	1900	476	542	499	25,05	28,53	26,26	104,83	92,07
Boolean	594	2100	532	613	561	25,33	29,19	26,71	105,45	91,52
Btree2	363	983	304	354	341	30,93	36,01	34,69	112,17	96,33
Btree2th	255	799	252	300	294	31,54	37,55	36,8	116,67	98
Btree	417	1600	464	530	503	29	33,13	31,44	108,41	94,91
Btreesynt	366	988	308	360	351	31,17	36,44	35,53	113,96	97,5
Character	35	105	64	108	96	60,95	102,86	91,43	150	88,89
Char	61	331	120	158	146	36,25	47,73	44,11	121,67	92,41
Commut.	122	455	164	219	203	36,04	48,13	44,62	123,78	92,69
Compac	130	576	184	228	216	31,94	39,58	37,5	117,39	94,74
Elem	30	100	56	96	84	56	96	84	150	87,5
ElemList	104	280	120	165	157	42,86	58,93	56,07	130,83	95,15
FDT1	83	304	124	175	155	40,79	57,57	50,99	125	88,57
GenerateN.	257	377	148	205	187	39,26	54,38	49,6	126,35	91,22
Importada	38	108	64	112	99	59,26	103,7	91,67	154,69	88,39
List	340	1400	396	461	431	28,29	32,93	30,79	108,84	93,49
ListEq	352	1600	460	519	497	28,75	32,44	31,06	108,04	95,76
NatBool2	193	504	192	235	226	38,1	46,63	44,84	117,71	96,17
NatBool	399	1800	468	534	500	26	29,67	27,78	106,84	93,63
Nat	425	1400	368	429	394	26,29	30,64	28,14	107,07	91,84
NatEq	251	1000	300	355	336	30	35,5	33,6	112	94,65
NatList	215	798	260	309	301	32,58	38,72	37,72	115,77	97,41

Tabela 4.3: Análise do método de codificação binária (cont.)

Arquivo	.casl	.agraph	.bgf	.gz	.bgf.gz	**B/F**	**G/F**	**BG/F**	**BG/B**	**BG/G**
	Bytes (B)					**Porcento (%)**				
Natth	175	560	180	221	211	32,14	39,46	37,68	117,22	95,48
Num	69	212	92	133	118	43,4	62,74	55,66	128,26	88,72
Sem	197	628	196	245	222	31,21	39,01	35,35	113,27	90,61
SortVar	77	161	92	126	125	57,14	78,26	77,64	135,87	99,21
SortVarImp	147	318	148	181	185	46,54	56,92	58,18	125	102,21
String	105	350	136	182	172	38,86	52	49,14	126,47	94,51
Natural	135	480	164	207	195	34,17	43,13	40,63	118,9	94,2
Natural1	34	104	60	105	93	57,69	100,96	89,42	155	88,57
Natural2	135	481	164	208	199	34,1	43,24	41,37	121,34	95,67
Natural3	93	305	120	167	152	39,34	54,75	49,84	126,67	91,02
Natural4	58	169	80	125	115	47,34	73,96	68,05	143,75	92
Natural5	125	376	140	189	176	37,23	50,27	46,81	125,71	93,12
Natural6	178	773	236	285	273	30,53	36,87	35,32	115,68	95,79
Principal	175	450	184	223	223	40,89	49,56	49,56	121,2	100
Stack1	69	155	88	123	122	56,77	79,35	78,71	138,64	99,19
Stack2	227	742	256	297	291	34,5	40,03	39,22	113,67	97,98
Stack3	148	365	152	189	187	41,64	51,78	51,23	123,03	98,94
Stack4	182	540	200	233	238	37,04	43,15	44,07	119	102,15
StringFDT	116	353	140	190	180	39,66	53,82	50,99	128,57	94,74
Synt	53	228	96	144	122	42,11	63,16	53,51	127,08	84,72
Total_Order	402	1600	424	485	464	26,5	30,31	29	109,43	95,67
Test2	224	764	256	282	290	33,51	36,91	37,96	113,28	102,84
Test3	375	1470	420	441	455	28,57	30	30,95	108,33	103,17
Test4	651	2984	732	789	735	24,53	26,44	24,63	100,41	93,16
Test5	1135	5876	1392	1416	1336	23,69	24,1	22,74	95,98	94,35
Test6	2022	11396	2892	2641	2580	25,38	23,17	22,64	89,21	97,69
Test7	3631	21922	5696	4995	4831	25,98	22,79	22,04	84,81	96,72

Observando os resultados das tabelas anteriores, pode-se concluir que o desempenho do método não é suficientemente satisfatório, visto que para arquivos .agraphs menores, o tamanho do arquivo .bgf é menor que o tamanho do arquivo .gzip, entretanto, para arquivos .agraphs maiores esta relação se inverte.

Apesar desse desempenho, a possibilidade de uso de um método próprio de compactação do formato é fornecida, e, estudos sobre um possível aperfeiçoamento do método, melhorando o desempenho da compactação, serão realizadas em trabalhos futuros.

Capítulo 5

Trabalhos relacionados

Nesta seção, são expostos superficialmente alguns formatos, fornecendo suas principais características e uma comparação com o formato AGraph. Destes formatos, é concedido maior destaque ao ATerms, formato que serviu como uma motivação para o desenvolvimento do formato AGraphs.

Estas comparações são realizadas sobre formatos que possuem (ATerms, GraphML, GraX, GXL e XML) e não possuem (GDL) propostas semelhantes ao formato AGraphs (representação e transferência de dados). Das comparações com os formatos que possuem a mesma proposta, observa-se as propriedades no formato comparado e se estas podem ser representadas no formato AGraph.

Com os resultados obtidos destas comparações, pode-se verificar quais propriedades são importantes e se devem (ou podem) ser representadas no formato AGraphs.

5.1 *Annotated Terms* - **ATerms**

Annotated Terms (ou simplesmente ATerms) ([25]) é um formato de representação e transferência baseado em árvore. Este formato serve como representação interna em ferramentas que utilizam a biblioteca ATerm, sendo também utilizado na transferência de dados entre ferramentas.

O formato ATerm e sua biblioteca possuem as seguintes características:

- Aberto: Independente da plataforma de software ou hardware.
- Simples: A interface da biblioteca que manipula o formato necessita de poucas funções.
- Eficiente: As operações sobre a estrutura são efetuadas eficientemente.
- Conciso: O espaço de representação em memória das estruturas de dados explora o compartilhamento.
- Independente da linguagem: As estruturas podem ser criadas e manipuladas em qualquer linguagem de programação.
- Anotações: As anotações são informações não-funcionais na representação, normalmente usadas para estender a capacidade de representação desta estrutura em aplicações específicas.

As árvores constituindo um ATerm são denominadas termos e são representadas por aplicações de funções, definindo a estrutura de dados abstratos ATerms. Os ATerms utilizam 7 categorias de tipos primitivos:

- *INT*: Números inteiros com 32 bits ou 64 bits.
- *REAL*: Números reais com 64 bits de representação.
- *BLOB - Binary LOng data oBject*: Tipo binário que possui o tamanho da representação indicado no próprio elemento, possibilitanto uma representação binária grande.
- *APPL*: Aplicação de função é caracterizada por um símbolo de função e uma lista de ATerms correspondendo aos argumentos da função.
- *LIST*: Uma lista contendo zero ou mais termos é um ATerm válido.
- *PLACEHOLDER*: Um identificador de tipo utilizado para definir padrões para construção ou casamento de termos.
- Outro tipo é o par <*label, annotation*>. Este tipo não é um tipo identificado, pois, trata-se de uma lista de termos associando termos, onde, um dos termos é uma anotação.

Para cada tipo acima citado, exceto o último, existem construtores. Para o par <*label, annotation*>, o construtor é denominado *construtor de anotações*, e associa uma anotação (informação transparente) a um ATerm.

5.1.1 Operações

As operações dos ATerms são classificadas em dois níveis: (a) interface nível um, e (b) interface nível dois.

As funções da interface nível um manipulam estruturas simples, ATerms na forma primitiva. As operações deste nível, apresentados a seguir, são divididas em três categorias: (1) *criação/casamento de padrões*, (2) *leitura/escrita*, e (3) *Anotações*.

As operações da interface nível dois manipulam estruturas complexas, por exemplo, um conjunto de ATerms. Internamente, estas funções são compostas pela combinação das funções da interface nível um.

Criação/Casamento de padrões dos ATerms

A interface nível um da biblioteca ATerm utiliza o paradigma *criação/casamento de padrões*:

- *Criação* - Um ATerm é construído a partir de um padrão e um conjunto de regras de definição (composição). A função `ATerm ATmake(String p, ATerm a1, ..., ATerm an)` cria um termo seguindo um padrão `p`. Este padrão é analisado, e, dependendo do PLACEHOLDER contido em uma determinada posição, um ATerm é obtido dos argumentos (`a1` à `an`). Se alguma incompatibilidade entre o padrão e os argumentos ocorrer uma mensagem de erro é lançada e a operação é abortada.

- *Casamento de padrões* - Os termos que compõe um ATerm são obtidos a partir da comparação deste com um padrão de termos. Um ATerm é decomposto seguindo um padrão de formação de termos (decomposição). Na função `ATbool ATmatch(ATerm t, String p, ATerm * a1, ..., ATerm * an)` o termo `t` é comparado com o padrão `p`, ocorrendo um casamento, os termos que compõem `t` são associados aos argumentos da operação (`a1` à `a2`). Esta associação segue o padrão fornecido em `p`. Ocorrendo algum erro na execução do *casamento dos padrões*, uma mensagem de erro é enviada e a execução abortada. Caso contrário, se o casamento ocorrer, então `true` será o valor retornado, senão, `false`.

Como funções auxiliares existem:

- `Boolean ATisEqual(ATerm t1, ATerm t2)`: Verifica se os dois termos são iguais, realizando uma comparação nos subtermos, igualmente nas anotações.
- `Integer ATgetType(ATerm t)`: Retorna um inteiro correspondente ao tipo do termo `t`.

É importante observar que a estrutura de dados ATerms não possibilita a atualização destrutiva, ou seja, um termo após criado não é passível de alterações.

Leitura/Escrita dos ATerms

Existem dois modos de armazenamento em disco disponíveis para os ATerms: textual e binário.

O modo textual é legível, mas, o compartilhamento em mémoria é perdido. Por este motivo, é uma representação ineficiente em espaço.

Por outro lado, o modo binário é eficiente em espaço e mantém o compartilhamento dos termos, mantendo-se conciso. *BAF (Binary ATerm Format)* é a denominação dos arquivos neste formato.

As operações nesta categoria são:

- `ATerm ATreadFromString(String s)`
`ATerm ATreadFromTextFile(File f)`
`ATerm ATreadFromBinaryFile(File f)`: Estas funções lêem um termo de uma entrada e retorna um ponteiro da representação em memória do ATerm.
- `Boolean ATwriteToTextFile(ATerm t, File f)`
`Boolean ATwriteToBinaryFile(ATerm t, File f)`
`String ATwriteToString(ATerm t)`: Funções de armazenamento em disco da representação interna de um termo. As duas primeiras funções retornam `true` caso a escrita obtenha sucesso, e na terceira função, o termo escrito é retornado na *string*.

Anotações

Existem três funções de criação/manipulação de anotações:

- `ATerm ATsetAnnotation(ATerm t, ATerm l, ATerm a)`: Associa o par *label-anotação*(`l, a`) ao termo `t`, e retorna o termo representando essa associação.
- `ATerm ATgetAnnotation(ATerm t, ATerm l)`: Recupera o termo associado ao *label* `l` no termo `t`.
- `ATerm ATremoveAnnotation(ATerm t, ATerm l)`: Remove a anotação associada ao *label* `l` no termo `t`, e retorna o termo resultado.

5.1.2 Implementação

A implementação da biblioteca ATerms garante as propriedades citadas na introdução sobre o formato ATerm. Para o cumprimento destas propriedades, algumas características na implementação são observadas. Estas características são: (1) Máximo compartilhamento, (2) Coleta de lixo, (3) Codificação dos termos, e (4) BAF - *Binary ATerm Format*.

Máximo compartilhamento

Em termos, é muito comum ocorrer sub-termos semelhantes. Beneficiando-se desta característica, a biblioteca ATerm cria um novo nó apenas quando este não existe na estrutura em memória.

Devido à implementação do máximo compartilhamento, a estrutura de dados ATerms não permite a atualização destrutiva, sendo uma representação puramente funcional, logo, um termo após criado não poderá ser modificado.

Coleta de lixo

A coleta de lixo é utilizada na remoção de estruturas sem referência na memória, liberando espaço para a alocação de novas estruturas. *Mark-sweep collection* [22] é a técnica de coleta de lixo na biblioteca ATerms.

Uma exceção na aplicação tradicional da técnica *mark-sweep* é causada pelo uso da operação *ATprotect*. Esta operação permite que o usuário proteja um termo sem referências da desalocação na coleta de lixo, permitindo, por exemplo, a declaração de termos temporários.

Codificação dos termos

Uma característica importante dos ATerms é a eficiência em tempo e espaço. Parte desta eficiência é atribuída à sua codificação em memória. Na codificação, a biblioteca ATerms assume que o tamanho da palavra da máquina seja 8 bits, e os termos são codificados em duas os mais palavras. A codificação é construída em baixo nível.

BAF - *Binary ATerm Format*

No armazenado em disco, obviamente, um termo não é referenciado por seu endereço em mémoria. A solução foi atribuir identificadores para cada símbolo de função e para cada sub-termo, onde um sub-termo é um símbolo de função com argumentos. Um arquivo no formato binário ATerm (BAF) contém duas tabelas: (a) uma tabela com símbolos

de funções e os respectivos identificadores, e, (b) uma tabela de termos que associa um identificador a cada sub-termo.

Num arquivo BAF, os ATerms são escritos em ordem prefixada, por exemplo, para escrever uma função aplicação, primeiro escreve-se o identificador do símbolo da função, seguido pelos identificadores dos argumentos (sub-termos). Quando um termo sem identificador é escrito, cria-se um índice na tabela de termos para este termo, e, então, escreve-se o identificador dos seus sub-termos, garantindo que um termo é escrito uma única vez, preservando o compartilhamento.

5.1.3 ATerms x AGraphs

Apesar de possuirem sintaxes abstratas diferentes, AGraphs possui a sintaxe baseada em grafos, e os ATerms em árvores (termos), apresenta-se aqui uma comparação entre eles.

Assim como ATerms, AGraphs possui dois modos de armazenamento (*codificação*). Nos AGraphs o armazenamento textual e binário mantém um compartilhamento que não é máximo, diferente dos ATerms que o modo binário garante o máximo compartilhamento.

AGraphs, igualmente aos ATerms, foi proposto para a representação de programas usando nível de abstração baixo. A princípio, AGraphs não possue nenhuma restrição para representar programas usando outros níveis de abstração.

O *mecanismo de transferência* dos dados não é pertinente ao formato, a princípio, mas alguns formatos são projetados para serem usados com um determinado mecanismo. Os ATerms foi desenvolvido para ser usado no mecanismo do tipo arquivo, assim como os AGraphs. Adicionalmente, ATerms também possibilitam usar a transferência direta entre ferramentas (*direct inter-tool*).

Com relação aos *tipos de esquemas*, AGraphs e ATerms possuem os esquemas classificados como fixos. Entretanto, os AGraphs permitem a modificação do esquema usando o *gerador automático*.

Do ponto de vista das bibliotecas dos formatos, a *compatibilidade de tipos* entre ATerms e AGraphs é possível. Para cada tipo primitivo que um ATerm represente, este tipo é capaz de ser representado em AGraph, ou usando o tipo primitivo correspondente ou definindo um novo tipo de nó. Pode-se citar como exemplo o tipo primitivo ATerm *aplicação*, que é derivado das propriedades das árvores, e são representados em grafos. Por exemplo, a *aplicação* `func(p1, p2, ..., pn)`, pode ser representada em AGraphs como um nó do tipo *aplicação*, com um atributo string `nomeApl` (i.e. `func`) e uma lista de nós do tipo *param* (i.e. `p1, p2, ...,pn`) que armazena informações sobre os parâmetros da *aplicação*.

Essa maneira de representar os tipos de outros formatos que não possuem tipos correspondentes em AGraphs, definindo um novo tipo de nó, é usado em outras ocasiões nas comparações desta seção.

Semelhante ao ATerms, AGraphs foi desenvolvido com o conceito de *anotações*. Em AGraphs, esta capacidade ainda não é representada persistentemente (campo `aToolInfo` na estrutura de um nó, ver seção 3.3). Como trabalho futuro, a inserção desta característica e o desenvolvimento de funções para a sua manipulação serão realizadas.

O *máximo compartilhamento* em memória dos ATerms é obtido pela codificação na representação em memória e refletida na representação persistente binária. Nos AGraphs, o compartilhamento é obtido pela própria estrutura dos grafos, ou seja, a representação de

um elemento é uma referência a este.

Como consequência da implementação, AGraphs manipulam os grafos com atualização destrutiva. Ou seja, em qualquer momento um grafo pode ser modificado com eficiência. Diferente do que ocorre com os ATerms que a atualização nos termos é não-destrutiva.

5.2 Graph Description Language - GDL

É um formato puramente textual que descreve grafos, subgrafos, nós e arestas para aplicações de visualização de grafos [20]. Este especifica detalhes da aparência dos atributos de um grafo. Por exemplo, tamanho (*size*), identificadores (*label*) e cores (*colors*) são atributos para os nós do grafo. O trecho de pseudo-código, a seguir, mostra as principais informações de uma especificação no formato GDL.

```
graph: {
  < atributos do grafo >
  < lista de nós ou subgrafos >
  < lista de arestas >
}
```

A seguir, as definições de nós e arestas.

```
node: {
  title: < identificador do nó >
  < atributos do nó >
}

edge: {
  source: < identificador do nó fonte >
  target: < identificador do nó alvo >
  < atributos da aresta >
}
```

Um grafo (`graph`) é definido por seus atributos (`< atributos do grafo >`), um conjunto de descrições de nós (`< lista de nós ou subgrafos >`), e uma lista de descrições de arestas (`< lista de arestas >`).

A descrição de um nó (`node`) é formada por um identificador (`title`), e os atributos do nó (`< atributos do nó >`). Os elementos que compõem a descrição da aresta (`edge`) são: identificador do nó fonte (`source`), idenificador do nó destino (`target`) e um conjunto de atributos (`< atributos da aresta >`).

Os subgrafos são grafos definidos em nós. Os atributos dos nós e arestas são pré-definidos pelo formato, assim como as arestas especiais (`backedge, nearedge e bentedge`) que possuem as mesmas definições das arestas simples (`edge`).

Observa-se que este formato não permite a definição de outros atributos para os tipos de elementos, ou seja, somente os elementos pré-definidos pelo formato GDL podem ser usados, restringindo os domínios das aplicações com este formato.

Exemplo 5.1 *O código GDL a seguir fornece a descrição de um grafo (*`graph`*) com dois nós (*`node`*), com denominações (*`title`*) “*`a`*” e “*`b`*”. Uma aresta (*`edge`*) é fornecida, tendo como nó fonte (atributo* `source`*) o nó identificado por “*`a`*”, e o nó destino é identificado por “*`b`*”. Esta aresta possui um atributo,* `color`*, com o valor* `red`.

```
graph: {
  node: { title:  "a" }
  node: { title:  "b" }
  edge: { source: "a" target: "b" color: red }
}
```

Construindo uma relação com AGraphs, cada nó GDL terá um nó correspondente no grafo AGraph, assim como os seus atributos. Como AGraphs não permite arestas atribuídas, as arestas GDL seriam representadas por nós, onde o nó fonte teria uma aresta AGraph para este nó que representa a aresta GDL, e este teria uma aresta AGraph para o nó destino, permitindo que as arestas sejam tipadas. Esta é uma solução para representar arestas tipadas em AGraphs, ocorrendo citações a esta solução nas comparações com outros formatos nesta seção.

Uma possível solução para a definição de subgrafos em AGraphs usa nós importação, onde os grafos mais internos seriam as unidades importadas e as mais externas as unidades importadoras. Esta solução permite a modularidade de unidades, o que não ocorre com GDL.

5.3 GraphML

É uma linguagem que descreve as propriedades estruturais de grafos e um mecanismo extensível e flexível para incluir informações específicas às aplicações [4] usando XML como formato de descrição. Este descreve grafos direcionados, adirecionados e mistos, adicionalmente com o conceito de hipergrafos e grafos hierárquicos.

Toda a descrição está contida no elemento `graphml`, sendo formado por definições de grafo (elemento `graph`), novos atributos (elemento `key`), nós (elemento `node`), arestas (elemento `edge`) e hiperarestas (elemento `hyperedge`).

O elemento `key` permite a definição de um novo atributo identificado pelo valor de `id`. Para definir este novo atributo é necessário informar o tipo dos elementos que podem possuí-lo (`for`), um nome (`attr.name`) e um tipo (`attr.type`). Para usar os elementos definidos em `key` é necessário declarar um elemento `data` de algum nó ou aresta, informando qual o elemento `key` que está sendo utilizado (parâmetro `key`).

Na definição de um grafo (`graph`) é necessário informar um identificador (`id`) e alguns parâmetros do grafo, por exemplo, o tipo das arestas (parâmetro `edgedefault`), se direcionadas ou não. Um grafo é formado por um conjunto de nós (`node`), arestas (`edge`) e hiperarestas (`hyperedge`), e adicionalmente, por definições de outros grafos, modelando grafos hierárquicos.

Um *nó* (`node`) é declarado pelo identificador (`id`) e por outros parâmetros e por atributos adicionais. Uma *aresta* (`edge`) é declarada informando um identificador (`id`), o nó fonte (parâmetro `source`) e o nó destino (parâmetro `target`). GraphML permite a declaração de arestas com atributos, assim, o elemento `edge` pode ser um elemento XML complexo. Uma *hiperaresta* (`hyperedge`) define a associação entre um conjunto de outros nós. O elemento `hyperedge` é complexo, formado por elemento simples `ednpoint` que indicam os nós alvos da associação (parâmetro `node`).

Exemplo 5.2 *A seguir, fornece-se um exemplo que descreve características importantes no formato* GraphML.

```
<graphml>
```

```
 <key id="d0" for="node" attr.name="color" attr.type="string">
  <default>yellow</default>
 </key>

 <key id="d1" for="edge" attr.name="weight" attr.type="double"/>

 <graph id="G" edgedefault="directed">
  <node id="n0"/>
  <edge source="n0" target="n2"/>
  <node id="n1"/>
  <node id="n2"/>
  ...
  <node id="n5">
   <data key="d0">turquoise</data>
  </node>
  ...
  <edge id="e2" source="n1" target="n3">
   <data key="d1">2.0</data>
  </edge>
  ...
  <hyperedge>
   <endpoint node="n6" />
   <endpoint node="n7" />
   <endpoint node="n8" />
  </hyperedge>
 </graph>
  ...
</graphml>
```

Neste exemplo, existe a definição de um arquivo no formato GraphML*. Na descrição do elemento* `graphml` *existe a definição de dois atributos: (1)* `color`*, que é identificado por* `d0`*, é do tipo* `string` *e é aplicado aos elementos* `node`*, e, (2)* `weight`*, que é identificado por* `d1`*, é do tipo* `double` *e é aplicado aos elementos* `edge`*. Observe que podem existir valores iniciais para estes novos atributos. No exemplo, para o atributo* `color` *o valor inicial é* `yellow`*, e para o atributo* `weight` *não existe valor inicial.*

Neste trecho, define-se um grafo com arestas direcionadas, sendo esta característica indicada pelo valor `directed` *atribuído ao parâmetro* `edgedefault`.

Existe a declaração de três nós (`n0`*,* `n1` *e* `n2`*) com o atributo* `color` *padrão e outro que possui o valor* `turquoise` *para o atributo* `color`.

Existem duas arestas declaradas, sendo uma arestas sem atributo entre os nós `n0` *e* `n2`*. A outra aresta é identificada por* `e2`*, liga os nós* `n1` *e* `n3` *e o atributo* `weight` *possui valor* `2.0`.

A única hiperaresta declarada associa interliga os nós `n6`*,* `n7` *e* `n8`.

As principais diferenças nas estruturas dos grafos entre GraphML e AGraphs são as arestas atribuídas e adirecionais. Com relação às arestas atribuídas a solução de representação é semelhante à aplicada ao formato GDL. Em AGraphs, as arestas são direcionais, contudo, para cada aresta adirecional GraphML existirá um par de arestas direcionais AGraph correspondente, sendo uma aresta do nó fonte para o nó destino, e outra aresta do nó destino para o nó fonte.

5.4 GraX

Formato de comunicação, formalmente baseado em **TGraphs** [10], o qual define uma classe abrangente de grafos, tendo XML como a notação utilizada para representar esta classe.

A sintaxe abstrata deste formato é definida por um grafo direcionado, tipado, atribuído e ordenado (**TGraphs**), o qual permite múltiplas arestas e *loops*.

Um arquivo no formato GraX é formato pela descrição de um grafo (elemento `grax`) e o respectivo esquema (parâmetro `schema`). Como elementos do grafo, pode-se ter a declaração de nós (elemento `vertex`) e de arestas (elemento `edge`). Logo, o formato GraX não permite a declaração de grafos hierárquicos.

A declaração de um nó (`vertex`) e de uma aresta (`edge`) possuem um identificador (parâmetro `id`), um tipo (`type`), podendo possuir atributos declarados em elementos simples (elemento `attr`), assim, configurando um grafo atribuído e tipado. Adicionalmente, um nó pode ter a declaração explícita da ordem das suas arestas usando o parâmetro `lambda`, que declara esta ordem usando os identificadores das arestas. Para indicar o nó fonte e destino de uma aresta os parâmetros `alpha` e `omega` são usados, respectivamente. O elemento `attr` possui dois parâmetros que indicam o seu nome (name) e o seu valor (`value`).

Exemplo 5.3 *Para ilustrar a descrição dos elementos que compõe o formato* GraX*, um trecho de descrição de um grafo é fornecido.*

```
<grax schema="meta.scx">

  <vertex id="n1" type="cn1" lambda="e2 e1 e3">
    <attr name="attN" value="node1" />
  </vertex>
  ...
  <edge id="e1" type="ce1" alpha="n1" omega="n5" />
    <attr name="attE" value="edge1" />
  </edge>
  ...
</grax>
```

Neste exemplo, existe a declaração de um grafo GraX*, cujo o esquema da classe de grafos é definido no arquivo* `meta.scx`*. São definidos dois elementos do grafo: (1) um nó com identificador* `n1`*, do tipo* `cn1` *e com ordenação local das suas arestas* `e2`*,* `e1` *e* `e3`*, e possui um atributo denominado* `attN` *cujo valor é* `node1`*. (2) uma aresta denominada* `e1`*, do tipo* `ce1`*, que possui nó fonte com identificador* `n1` *e nó alvo identificado por* `n5`*, e possui um atributo denominado* `attE` *cujo valor é* `edge1`*.*

O formato GraX não permite hiperarestas e grafos hierárquicos. Entretanto, é possível declarar a ordenação das arestas em um nó e a herança múltipla.

Para representar a ordenação das arestas em um nó de um AGraph, usa-se a própria ordenação estrutural da implementação dos nós de um nó AGraph, pois, as arestas dos grafos AGraphs possuem uma ordenação implícita na estrutura de implementação dos nós, logo, a primeira aresta na estrutura de um nó é a primeira aresta na ordenação, podendo explicitar a ordenação com a declaração de um atributo do nó. Por exemplo, `order`, que pode ser um atributo do tipo `string` que armazena a sequência de identificadores das arestas em um nó, sendo que esta sequência representa a ordenação das arestas.

5.5 Graph eXchange Language - GXL

Este formato foi confirmado como um padrão nos formatos de comunicação [28]. Originado da junção das principais características dos formatos GraX, *Tuple Attribute Language* (TA [12]) e do formato em grafo do sistema de reescrita de grafos PROGRES ([4]).

GXL envolve a instância do grafo e o respectivo esquema de grafo. Os esquemas deste formato descrevem classes de grafos tipados, atribuídos, direcionados, ordenados, incluindo hiperarestas e grafos hierárquicos.

O formato de representação das instâncias e dos esquemas é o XML, ambos contidos no mesmo arquivo para cada instância.

Os grafos são descritos com os seguintes elementos: `<graph>` define grafos, `<node>` define nós, `<edge>` define arestas, `<attr>` atributos dos elementos dos grafos com nome e valor, e tipos pré-definidos (`<bool>`, `<int>`, `<float>` e `<string>`). As hiperarestas são definidas com a utilização dos elementos `<rel>` e `<relend>`, que permitem hiperarestas direcionais ou adirecionais e ordenadas.

Além das características próprias dos grafos, GXL possibilita a referência a objetos externamente armazenados (`<locator>`), e valores estruturados (`<enum>`, `<seq>`, `<set>`, `<bag>` e `<tup>`).

Como elemento de um grafo, pode-se ter descrições de outros grafos, ou seja, definição de grafos aninhados, sendo uma maneira de descrever grafos hierárquicos.

Exemplo 5.4 *O seguinte trecho é um exemplo de descrição de uma instância de grafo, que usa o formato* GXL *cujo o esquema é denominado* `schema.gxl`*, como indicado pelo parâmetro* `xlinkhref` *do elemento* `type` *de* `graph`.

```
<gxl>
<graph id="g1">
  <type xlinkhref="schema.gxl" />
  <node id="n1" >
    <type xlinkhref="schema.gxl#Function" />
    <attr name="name">
      <string>main</string>
    </attr>
  </node>
  ...
  <edge id="e1" from="n1" to="n2" toorder="1" >
    <type xlinkhref="schema.gxl#isCaller" />
    <attr name="line">
      <int>8</int>
    </attr>
  </edge>
  ...
</graph>
</gxl>
```

Neste exemplo, são definidos: (1) um grafo identificado por `g1`*, (2) um nó (*`node`*) identificado por* `n1` *com um atributo* `name` *do tipo* `string` *que armazena o valor* `main`*, e, (3) uma aresta (*`edge`*) do nó* `n1` *ao* `n2` *identificada por* `e1`*, possuindo um atributo inteiro denominado* `line` *que armazena o valor* `8`*, e esta é a primeira aresta de chegada no nó* `n2`*, como é informado pelo parâmetro* `toorder`.

Observe que cada elemento possui um atributo `type` *que fornece o nome do esquema que descreve este elemento.*

A diferença significante entre GXL e AGraphs é a disponibilidade de definição de esquemas GXL e a referência externa. Diferente do GXL, os AGraphs possuem esquemas fixos, mas a possibilidade de geração automática fornece uma diversidade de esquemas possíveis de ser gerados. Com relação a referência externa, nos estudos realizados, esta característica não ficou compreensível de ser realizada em AGraphs, sendo sugerida para análise nos trabalhos futuros.

5.6 XML

Um formato bastante difundido, principalmente na transferência de dados na *internet (Web)*, é o XML [7] que é a sigla para linguagem extensível de marcações (*eXtensible Markup Language*) e é derivada de *SGML*[6].

Semelhante a um arquivo *HTML*[19], um arquivo XML é formado por elementos denominados *marcações*. Estas marcações são identificadas por *tags* e podem possuir parâmetros e atributos.

Os elementos em XML podem ser estendidos, ou seja, diferente de *HTML*, podem ser definidas novas marcações. Esta é a principal característica XML que possibilita o seu uso na transferência de dados entre ferramentas.

Segundo a classificação em [13], XML é um formato que utiliza dados estruturados para representar os dados, sem restrições com relação ao nível de representação (abstração), somente com codificação textual, sendo que o mecanismo de transferência de dados usado é o fluxo de dados estruturados.

Como citado, XML permite a definição de outras marcações, sendo o mais geral de todos os formatos apresentados nesta seção. A descrição dessas novas marcações é fornecida no arquivo *XML Schema*, que pode ser localizado separadamente do arquivo que contém as dados da instância, assim, pode-se classificar o esquema do formato XML como mutável, sendo definido explicitamente e com a localização externa com relação à ferramenta que usa este formato.

Esta capacidade de declarar novas marcações possibilita que XML seja o formato mais usado nas ferramentas de compartilhamento de informações, inclusive para representar outros formatos (por exemplo, GraphML, GraX e GXL), como observar-se nos formatos apresentados nesta seção.

Existem várias características distintas entre os formatos XML e AGraphs, por exemplo, o tipo da sintaxe abstrata, o mecanismo transferência e o tipo de esquema. Contudo, a principal diferença desses formatos reside na capacidade de novas marcações pelo formato XML. O conceito de geração automática associado aos AGraphs não possibilita a definição de novos elementos, assim, um elemento de um AGraph, é um nó, uma aresta, um grafo, ou um atributo de nó, restringindo a sua aplicação à representação de dados com grafos. Diferente do que ocorre com AGraphs, XML que possui a capacidade de definição de novas marcações (elementos), permitindo a representação de dados de naturezas diversas.

Capítulo 6

Conclusão

De modo geral, programas manipulam e transferem dados que são modelados usando estruturas de dados. Tradicionalmente, estes dados eram modelados usando um formato *ad-hoc*. Devido aos requisitos de uma nova tendência (padronização, abertura e interoperabilidade) trabalhos definindo padrões de estruturas e modelos de transferência de dados têm sido desenvolvidos.

Este trabalho propôs a definição de um formato usado em ferramentas desenvolvidas na UFRN, inicialmente de maneira *ad-hoc*, denominado AGraphs. AGraphs é um formato de representação e transferência de dados que possui uma biblioteca de manipulação com uma interface simples.

Com a definição do formato AGraphs, o desenvolvimento de ferramentas de suporte e as comparações com outros formatos, com o intuito de obter características significantes, são tarefas necessárias, assim, também pertencendo aos objetivos deste trabalho.

A definição do formato foi construída em três categorias de especificações: textual, gráfica e formal. Neste trabalho, a especificação textual é a base da descrição do formato, auxiliada pelas especificações formais e gráficas, embora as informações fornecidas por estas especificações se completem.

O desenvolvimento da especificação formal foi muito importante para o prosseguimento das atividades. Com a construção desta especificação foi possível observar propriedades citadas anteriormente na descrição textual que não eram compatíveis com as implementações dos AGraphs, obrigando alterações na definição textual ou nas implementações. Como exemplo dessa afirmação, pode-se citar a necessidade de informar a restrição do tipo de nó na declaração de uma `hiperaresta` em um esquema, que era uma informação desnecessária e foi removida da atual implementação. Esta restrição deixou de ser informada na declaração da `hiperaresta` para ser definida pelo tipo de nó lista usado para construir a lista ligada desta `hiperaresta`.

Na definição de um formato, é necessária a determinação da organização dos elementos que o formam e da sua biblioteca de manipulação. Conceitualmente, o formato AGraphs pode-se aplicar a várias linguagens de programação. Todavia, a sua instância é associada a uma única linguagem, assim, para definir uma instância do formato, algumas informações são indispensáveis, sendo estas informações fornecidas nos esquemas. Consequentemente, a especificação textual do formato foi divida em três definições: (a) da estrutura de dados, (b) da biblioteca de manipulação, e, (c) do esquema AGraphs.

A descrição da estrutura de dados AGraphs contém as relações entre os elementos que compõem o grafo, a descrição das bibliotecas fornece os perfis das funções de manipulação e a descrição do esquema provê as informações necessárias para a definição da

semântica do formato AGraph para uma linguagem representada.

Para proporcionar apoio às aplicações que empregam AGraphs como formato de representação e/ou transferência de dados, foram desenvolvidas duas ferramentas: (a) gerador automático, e, (b) a implementação do método de compactação do formato persistente textual.

Devido à associação existente entre uma instância do formato AGraphs com uma linguagem representada, foi desenvolvido o gerador automático, que constrói uma biblioteca AGraphs a partir da descrição do esquema do grafo ou da sintaxe da linguagem representada. A modalidade de geração a partir do esquema AGraph admite arquivos em um formato de entrada próprio (formato *customizado*) que descreve a estrutura dos elementos do grafo. A outra modalidade de geração usa a sintaxe concreta em um formalismo como entrada, proporcionando que a sintaxe também desempenhe o papel de documentação do AGraph gerado.

Embora a modalidade de geração a partir da sintaxe propicie um reuso da sintaxe concreta, devido à implementação do gerador, existem várias restrições nas regras sintáticas que impedem a controlabilidade sobre o AGraph resultante, diferente do que ocorre com a geração *customizada*.

Uma vez que o formato AGraph pode ser usado como formato de transferência, na tentativa de evitar que os arquivos no formato persistente textual possuam tamanhos indesejáveis, foi desenvolvido um método de compactação. Este método reduz o tamanho da representação textual removendo as redundâncias existentes entre as descrições dos nós do mesmo tipo, gerando um arquivo no formato persistente binário.

Como foi observado na seção 4.2, o método de compactação não obteve resultados satisfatórios se comparado com o método de compactação .gzip. A principal causa deste desempenho é atribuída a existência das redundâncias do grafo em memória que são refletidas na representação textual, mas não removidas totalmente. Propõe-se como trabalho futuro, a adaptação do método para abranger as redundâncias do ponto de vista do grafo em memória.

Com a intenção de perceber características significantes e proporcionar análises sobre a atual organização conceitual e estrutural do formato AGraph, comparações com outros formatos foram realizadas. Destas comparações, pode-se concluir que algumas características presentes nos formatos podem ser representadas em AGraph, por exemplo, as arestas atribuídas e ordenadas, e as características que não podem ser observadas podem ser importantes em várias aplicações, como por exemplo, a herança e a referência a elementos externos.

Estas duas últimas não são características óbvias em AGraphs, logo, como trabalho futuro, é proposta uma análise mais profunda sobre a importância delas nas aplicações que podem usar AGraphs.

Adicionalmente, existem tarefas propostas que não foram executadas e tarefas geradas a partir dos estudos (análises) deste trabalhos. Estas tarefas são dispostas como trabalhos futuros:

- manual de referência para as bibliotecas AGraphs, o gerador automático e a compactação. Estes manuais deverão conter informações relativas aos procedimentos de instalação, uso e API.
- redefinição do formato textual persistente usando a representação XML, possibilitanto uma capacidade maior de interoperabilidade com outras ferramentas.

- desenvolvimento de funções adicionais que manipulem AGraphs. Como exemplo dessas funções, pode-se citar as funções : (a) *coleta de lixo*, remove todos os nós sem referência, (b) *cópia* de um AGraph, cria um AGraph com os nós armazenando os mesmos dados, e (c) *liberar* memória de um grafo, liberando a memória de todos os elementos que o compõem.

Referências Bibliográficas

[1] E. Astesiano, M. Bidoit, H. Kirchner, B. K.-Brückner, P. D. Mosses, D. Sannella, and A. Tarlecki. CASL: The Common Algebraic Specification Language. *Theoretical Computer Science*, 286(2):153–196, 2002.

[2] P. Borovanský, H. Cirstea, H. Dubois, C. Kirchner, H. Kirchner, P.E. Moreau, C. Ringeissen, and M. Vittek. *ELAN Users Manual*. INRIA Lorraine & LORIA, Nancy, France, 2000. Available at `http://elan.loria.fr`.

[3] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An Overview of ELAN. In C. Kirchner and H. Kirchner, editors, *Proc. Second Intl. Workshop on Rewriting Logic and its Applications*, Electronic Notes in Theoretical Computer Science, Pont-à-Mousson (France), Sep. 1998. Elsevier.

[4] U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt, and M. Marshall. GraphML progress report: Structural layer proposal. In *Proceedings 9th International Symposium on Graph Drawing (GD '01), Springer Lecture Notes in Computer Science 2265*, 2002.

[5] CoFI. *The CoFI Algebraic Specification Language*. Available at the CoFI home page: `http://www.cofi.info`.

[6] D. Connolly. Overview of SGML Resources, 2004. `http://www.w3.org/MarkUp/SGML/`.

[7] World Wide Web Consortium. Extensible Markup Language (XML) 1.0 (Third Edition). Technical report, World Wide Web Consortium, 2004. Available at: `http://www.w3.org/TR/2004/REC-xml-20040204/`.

[8] H.A. de Jong and P.A. Olivier. Generation of abstract programming interfaces from syntax definitions. *Journal of Logic and Algebraic Programming*, 59, 2004.

[9] D. Déharbe, S. Shankar, and E. Clarke. Model Checking VHDL with CV. In G. Gopalakrishnan and P. Windley, editors, *Formal Methods in Computer-Aided Design*, volume 1522 of *Lecture Notes in Computer Science*, pages 508–514. Springer Verlag, 1998.

[10] J. Ebert, B. Kullbach, and A. Winter. GraX – An Interchange Format for Reengineering Tools. In F. Balmas, M. Blaha, and S. Rugaber, editors, *Proceedings of the Sixth Working Conference on Reverse Engineering*, pages 89–98, 1999.

[11] S. Escobar and A.M. Moreira. Proposta de uma ferramenta de apoio formal à especificação e re-utilização de software. In *Anais do III Workshop de Métodos Formais*, João Pessoa, Brasil, 2000.

[12] R. Holt. An Introduction to TA: The Tuple Attribute Language. Technical report, Departament of Computer Science, University of Waterloo and Toronto, November 1998.

[13] D. Jin. Exchange of software representations among reverse engineering tools. Technical report, Department of Computing and Information Science - Queens University, December 2001.

[14] R. Kazman, S. Woods, and S. Carrière. Requirements for Integrating Software Architecture and Reengineering Models: CORUM II. In *Working Conference on Reverse Engineering(WCRE'98),Hawaii*, October 1998.

[15] G. Lima, A.M. Moreira, D. Déharbe, D. Pereira, D. Sena, and J. Vidal. FERUS: um ambiente de desenvolvimento de especificações CASL. In *Proceedings of SBES'2002 (Simpósio Brasileiro de Engenharia de Software): Sessão de ferramentas*, pages 1–6, October 2002.

[16] J. Martin. RSF File Format. Posted to the Rigi Developer Email Distribution List, 1999.

[17] A.M. Moreira. Parametrização de Componentes de Especificação com Preservação de Semântica. In *XII Simpósio Brasileiro de Engenharia de Software: SBES'98*, Maringá, PR, Brazil, 1998.

[18] A.M. Moreira, C. Ringeissen, and A. Santana. A Tool Support for Reusing ELAN Rule-Based Components. In J.L. Giavitto and P.E. Moreau, editors, *Proceedings of the 4th International Workshop on Rule-Based Programming, RULE'03*, Technical Report DSIC-II/11/03, pages 67–82. Departamento de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, June 2003.

[19] W3C recommendation. HTML 4.0 Specification. `http://www.w3.org/TR/1998/REC-html40-19980424/`, 1998.

[20] G. Sander. VCG – Visualization of Compiler Graphs. Technical report, Universität des Saarlandes, February 1995.

[21] A. Schurr. Developing Graphical (Software Engineering) Tools with PROGRESS, Formal Demonstration. In *Proceedings of the 19th International Conference on Software Engineering (ICSE'97)*, pages 618–619. IEEE Computer Society Press, May 1997.

[22] Frederick Smith and Greg Morrisett. Comparing mostly-copying and mark-sweep conservative collection. *SIGPLAN Not.*, 34(3):68–78, 1999.

[23] G. St-Denis, R. Schuaer, and R.K. Keller. Selecting a Model Interchange Format: The SPOOL Case Study. In *Proceeding of the Thirty-Trird Annual Hawaii International Conference on System Sciences*. IEEE Press, 2000.

[24] S. Tichelaar, S. Ducasse, and S. Demeyer. FAMIX: Exchange Experiences with CDIF and XMI. In *Proceedings of the Workshop on Standard Exchange Formats (WoSEF)*, 2000.

[25] M.G.J. van den Brand, H.A. de Jong, P. Klint, and P.A. Olivier. Efficient annotated terms. *Software-Practice and Experience*, 30:259–291, 2000.

[26] A. van Deursen. An Overview of ASF+SDF. In A. van Deursen, J. Heering, and P. Klint, editors, *Language Prototyping: An Algebraic Specification Approach*, pages 1–30. World Scientific Publishing Co., 1996.

[27] E. Visser. *Syntax Definition for Language Prototyping.* PhD thesis, University of Amsterdam, Sep. 1997.

[28] A. Winter, B. Kullbach, and V. Riediger. An Overview of the GXL Graph Exchange Language. In S. Diehl, editor, *Revised Lectures on Software Visualization International Seminar*, number 2269 in LNCS, pages 324–336, London, UK, 2002. Springer-Verlag.

[29] K. Wong. *RIGI User's Manual - Version 5.4.4.* Departmente of Computer Science, University of Victoria, June 1998.

[30] S. Woods, L. O'Brien, T. Lin, K. Gallagher, and A. Quilici. An Architecture For Interoperable Program Understanding Tools. In *Proceedings of the Sixth International Conference on Program Comprehension*, 1998.

[31] J. Ziv and A. Lempel. A universal algorithm for sequencial data compression. In *IEEE Transactions on Information Theory*, volume 23, pages 337–343, 1977.

[32] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate conding. In *IEEE Transactions on Information Theory*, volume 24, pages 530–536, 1978.

Apêndice A

Especificação formal - *B*

A especificação formal desenvolvida na linguagem *B* é fornecida. Foram desenvolvidos dois tipos especificações em *B*:

- uma especificação descrevendo a estrutura estática de AGraphs, fornecendo a estrutura e a relação entre os elementos que compõe o grafo, denominada *especificação estática*. Esta especificação corresponde à descrição das informações de um esquema AGraph.
- uma especificação que reflete as influências sofridas pela estrutura na aplicação das funções, denominada *especificação dinâmica*. Esta especificação descreve a organização estrutural dos nós e seus comportamentos em memória, se aproximando da implementação da representação e da biblioteca AGraph.

A.1 Especificação estática

Nesta especificação foram criados três arquivos "máquinas" (`MACHINE`): `Datatype`, `NodeDescription` e `UnitMeta`.

`Datatype` define os tipos pré-definidos (`PredefinedTypeNames`), os tipos enumerados (`Types`) e suas propriedades. Relembrando, os tipos pré-definidos dependem da linguagem de implementação. Por este motivo, são definidos como um subconjunto dos identificadores de tipos (`TYPENAMES`) na linha 25. E os tipos enumerados (declarados no esquema AGraph) são representados como uma relação entre um identificador de tipo (`TYPENAMES`) e o seu conjunto de valores possíveis (`ENUMVALUES`). E, obviamente, os tipos pré-definidos não podem ser tipos enumerados e vice-versa (linha 26).

```
01. MACHINE Datatype
02.
03. SETS
04.   TYPENAMES ; /* names of the enumerated types */
05.   ENUMVALUES  /* values of the enumerated types */
06.
07. CONSTANTS
08.   Types, /*
09.           * Associates enumerated type names
10.           * to the set of their values.
11.           */
12.   PredefinedTypeNames /*
```

```
13.                        * contains the names of predefined
14.                        * (basic) types of the language.
15.                        */
16.
17. PROPERTIES
18.   Types : TYPENAMES +-> POW(ENUMVALUES) &
19.
20.   /*
21.    * predefined type names are type names, but their values
22.    * are (the only ones) not defined in Types
23.    */
24.
25.   PredefinedTypeNames <: TYPENAMES &
26.   dom(Types) = TYPENAMES - PredefinedTypeNames
27.
28. END
```

`NodeDescription` é a máquina que descreve os elementos que compõem um AGraph e as relações entre eles. Os elementos de um AGraphs são representados pelas seguintes constantes: `nodeName` (nó), `edge` (aresta), `attribute` (atributo de nó), e `hypedge` (hiperaresta).

Para definir os tipos dos elementos que compõem os tipos de nó, é necessário utilizar os tipos pré-definidos e enumerados descritos na máquina `Datatype`. Por este motivo, a máquina `NodeDescription` inclui a máquina `Datatype` (linha 11).

Nesta descrição, um tipo de nó (`node`) é formado por um identificador de tipo (`nodeNames`) que relaciona a um conjunto de atributos (`attribute`), de arestas (`edge`) e de hiperarestas (`hyperedge`) nas linhas 64-67.

Para melhorar a legibilidade da descrição, um identificador de tipo de nó é relacionando ao conjunto de identificadores de atributo (`attributeNames`), de arestas (`edgeNames`) e hiperarestas (`hyperedgeNames`).

E para obter a descrição desses elementos, usam-se as relações `attribute`, `edge` e `hyperedge`, respectivamente para os atributos, arestas e hiperarestas (linha 38-39, 47-48 e 56-57, respectivamente).

Observe que, como o conjunto de identificadores é compartilhado (`NAME`), é importante restringir que a interseção entre os identificadores dos tipos de nós, atributos, arestas e hiperarestas é vazio (linha 71).

```
01. MACHINE
02.   NodeDescription
03.   /*
04.    * defines the Agraph data structure
05.    * which are its types of nodes, and
06.    * the structure of each node, with
07.    * its attributes, edges and hyperedges.
08.    */
09.
10. SEES
11.   Datatype
12.   /*
13.    * enumerated and predefined types and their
14.    * corresponding values.
15.    */
```

```
16.
17. SETS
18.   NAME  /*
19.          * Set of the attribute, edge, hyperedge
20.          * and node identifiers.
21.          */
22. CONSTANTS
23.   attribute, edge, hyperedge, /* elements that build the nodes */
24.   node,                       /* nodes */
25.
26.   /* subsets of NAME for each kind of element */
27.   attributeNames, edgeNames, hyperedgeNames, nodeNames
28.
29. PROPERTIES
30.
31.   nodeNames <: NAME &
32.
33.   /*
34.    * a typed attribute is defined its name and type, which may
35.    * be a language specific enumerated type or a pre-defined one.
36.    */
37.
38.   attribute: NAME +-> TYPENAMES &
39.   attributeNames = dom(attribute) &
40.
41.   /*
42.    * an edge of a given name may point to a set of
43.    * different node types - this function defines which
44.    * are those types.
45.    */
46.
47.   edge: NAME +-> POW1(nodeNames) &
48.   edgeNames = dom(edge) &
49.
50.   /*
51.    * an hyperedge is defined by its name and a list
52.    * node types, which may point to a set of different
53.    * node types.
54.    */
55.
56.   hyperedge: NAME +-> (nodeNames * POW1(nodeNames)) &
57.   hyperedgeNames = dom(hyperedge) &
58.
59.   /*
60.    * a node declaration is defined by its name (nodeNames)
61.    * a set of attributes, edges and hyperedges.
62.    */
63.
64.   node: NAME +->
65.         (POW(attributeNames) *
66.          POW(edgeNames) *
67.          POW(hyperedgeNames)) &
68.
69.   nodeNames = dom(node) &
70.
71.   attributeNames /\ edgeNames /\ hyperedgeNames /\ nodeNames = {}
72.
73. END
```

`UnitMeta` contém as definições dos tipos de nós especiais: nó raiz (`rootnode`), nós importação (`imports`) e nós lista (`lists`). Para definir a estrutura desses tipos especiais de nós, são usadas as definições de tipos de nós e seus elementos da máquina `NodeDescription` e os tipos pré-definidos e enumerados da máquina `Datatype`. Por causa destes motivos, a máquina `UnitMeta` inclui as máquinas `Datatype` e `NodeDescription` (linha 13-14).

A máquina `UnitMeta` possui dois parâmetros: language_name e `STRINGPARAM`. `language_name` é a etiqueta da linguagem representada pertencente ao conjunto de etiquetas válidas definido no conjunto `STRINGPARAM`.

Em um esquema, devem existir somente um tipo de nó raiz (linha 32), um conjunto de tipos de nós importação (linha 35) e um conjunto de tipos de nós lista (linha 38). Estes tipos especiais possuem as mesmas estruturas dos tipos de nós simples definidos na máquina `NodeDescription`.

Algumas propriedades especiais nestes tipos de nós especiais são declarados:

1. Se nenhum tipo de nó lista foi definido, não é possível simular hiperarestas (linha 40). Esta propriedade é óbvia, devido ao fato que para simular hiperarestas é necessário o uso de nós do tipo lista.

2. Como consequência da propriedade anterior, se não foi definido nenhum tipo de nó lista, não é possível definir tipo de nó importação (linha 59). Como a propriedade anterior, esta também é uma propriedade óbvia, na importação de unidade, são utilizadas hiperarestas, mas como não foi definida nenhuam lista, não é possível definir hiperaresta e consequentemente, lista do tipo importação.

3. E, nenhum dos tipos de nó especiais podem ser de duas categorias simultaneamete (linha 51, 54 e 57).

```
01. MACHINE
02.   UnitMeta(language_name, STRINGPARAM)
03.   /* specifies the language schema for the given language */
04.
05.   /*
06.    * STRINGPARAM is STRING type. STRING type cannot be used in
07.    * the PO generator of the machines.
08.    */
09.
10. CONSTRAINTS
11.   language_name : STRINGPARAM
12.
13. SEES
14.   Datatype, NodeDescription
15.   /*
16.    * Datatype - where all enumerated types for the description
17.    * of the language are defined
18.    */
19.
20.   /*
21.    * NodesDescription - this machine defines the language specific
22.    * Agraph data structure which are its types of nodes, and the
23.    * structure of each node, with its attributes, edges and
```

```
24.    * hyperedges.
25.    */
26.
27. CONSTANTS
28.   rootnode, imports, lists
29.
30. PROPERTIES
31.   /* root node type*/
32.   rootnode: node &
33.
34.   /* imports nodes types */
35.   imports: POW(node) &
36.
37.   /* lists nodes types */
38.   lists: POW(node) &
39.
40.   (lists = {} => hyperedge = {}) & /*
41.                                     * If there are no list nodes,
42.                                     * there can be no hyperedges
43.                                     */
44.   /*
45.    * The second element of the hyperedges
46.    * is a list node type.
47.    */
48.   dom(ran(hyperedge)) <: dom(lists) &
49.
50.   /* The intersection between all nodes is empty */
51.   rootnode /: imports &
52.
53.   /* Hom before */
54.   rootnode /: lists &
55.
56.   /* Hom before */
57.   imports /\ lists = {} &
58.
59.    (lists = {} => imports = {}) /*
60.                                 * If there are no list nodes,
61.                                 * there can be no import nodes
62.                                 */
63.  END
```

A.2 Especificação dinâmica

A especificação dinâmica representa a estrutura de um AGraph em memória. A estrutura do AGraph é representado pelos seus elementos e suas relações, e também o comportamento desses elementos a aplicação das funções. Nesta especificação, as funções de criação de um nó (`make`) e de acesso (`query` e `set`) são especificadas.

Com relação a máquina `Datatype` descrita na especificação estática, nenhuma alteração foi realizada. Aquela especificação também é válida para a especificação dinâmica e possui a mesma funcionalidade.

```
01. MACHINE Datatype
02.
```

```
03. SETS
04.   TYPENAMES ; /* names of the enumerated types */
05.   ENUMVALUES  /* values of the enumerated types */
06.
07. CONSTANTS
08.   Types, /*
09.           * Associates enumerated type names
10.           * to the set of their values.
11.           */
12.   PredefinedTypeNames /*
13.                        * contains the names of predefined
14.                        * (basic) types of the language.
15.                        */
16.
17. PROPERTIES
18.     Types : TYPENAMES +-> POW(ENUMVALUES) &
19.
20.     /*
21.      * predefined type names are type names, but
22.      * their values are (the only ones) not defined in Types.
23.      */
24.     PredefinedTypeNames <: TYPENAMES &
25.     dom(Types) = TYPENAMES - PredefinedTypeNames
26.
27. END
```

Na especificação da máquina `NodeDescription`, algumas alterações, com relação à descrição desta máquina na especificação estática, foram necessárias. Estas alterações são relacionadas ao suporte da definição das operações e dos nós lista.

```
01. MACHINE
02.  NodeDescription
03.
04.   /*
05.    * defines the Agraph data structure
06.    * which are its types of nodes, and the structure of
07.    * each node, with its attributes, edges and hyperedges.
08.    */
09.
10. SEES
11.   Datatype
12.
13.   /*
14.    * enumerated and predefined types and their
15.    * corresponding values.
16.    */
17.
18. SETS
19.   /* Set of the attribute, edge, hyperedge and node identifiers */
20.   NAME
21.
22. CONSTANTS
23.   attribute, edge, hyperedge, /* elements that build the nodes */
24.
25.   nodeattributeNames, /* Function of node +-> attributeNames */
26.   nodeedgeNames, /* Function of node +-> attributeNames */
```

```
27.   nodehyperedgeNames, /* Function of node +-> attributeNames */
28.
29.   /* subsets of NAME for each kind of the elements */
30.   attributeNames, edgeNames, hyperedgeNames,
31.   nodeNames, listnodeNames, anynodeNames
32.
33. PROPERTIES
34.
35.   nodeNames <: NAME &
36.   listnodeNames <: NAME &
37.   anynodeNames <: NAME &
38.
39.   /*
40.    * anynodeNames are the names for
41.    * all node that are not list nodes.
42.    */
43.   listnodeNames \/ anynodeNames = nodeNames &
44.   listnodeNames /\ anynodeNames = {} &
45.
46.   /*
47.    * a typed attribute is defined its name and type, which may
48.    * be a language specific enumerated type or a pre-defined one.
49.    */
50.
51.   attributeNames <: NAME &
52.   attribute: attributeNames --> TYPENAMES &
53.
54.   /*
55.    * an edge of a given name may point to a set of
56.    * different node types - this function defines which
57.    * are those types.
58.    */
59.
60.   edgeNames <: NAME &
61.   edge: edgeNames --> POW1(nodeNames) &
62.
63.   /*
64.    * an hyperedge is defined by its name and a list
65.    * node types, which may point to a set of different
66.    * node types
67.    */
68.
69.   hyperedgeNames <: NAME &
70.   hyperedge: hyperedgeNames --> listnodeNames &
71.
72.   /*
73.    * a node declaration is defined by its name (nodeNames)
74.    * a set of attributes, edges and hyperedges.
75.    */
76.
77.   nodeattributeNames : nodeNames --> POW(attributeNames) &
78.   nodeedgeNames : nodeNames --> POW(edgeNames) &
79.   nodehyperedgeNames : nodeNames --> POW(hyperedgeNames) &
80.
81.   /*
82.    * All list nodes have the next and previous hyperedges,
83.    * and the value edge.
84.    */
```

```
85.
86.   !l . (l : listnodeNames =>
87.        (#next . ( next : hyperedgeNames &
88.                   hyperedge(next) = l &
89.                   next : nodehyperedgeNames(l)))) &
90.
91.   !l . (l : listnodeNames =>
92.        (#previous . ( previous : hyperedgeNames &
93.                       hyperedge(previous) = l &
94.                       previous : nodehyperedgeNames(l)))) &
95.
96.   !l . (l : listnodeNames =>
97.        (#value . ( value : edgeNames &
98.                    l : edge(value) &
99.                    value : nodeedgeNames(l)))) &
100.
101.   attributeNames /\ edgeNames /\ hyperedgeNames /\ nodeNames = {}
102.
103. END
```

Além dos identificadores dos elementos dos nós (`attributeNames`, `edgeNames` e `hyperedgeNames`) e dos nós (`nodeNames`) foram definidos outros identificadores (`listnodeNames` e `anynodeNames`). `listnodeNames` é o conjunto de identificadores dos nós do tipo lista, e `anynodeNames` é o conjunto de identificadores dos nós de qualquer tipo de nó diferente de lista. Logo, um nó, exclusivamente, pertence à categoria de nós lista ou não (linha 43 e 44).

Na máquina `NodeDescription`, os elementos especiais que compõem um nó do tipo lista são definidos. Esses elementos especiais são: a aresta para o nó que possui a informação armazenada (`value`) e as hiperarestas para o nó próximo e anterior na lista (`previous` e `next`). Estes elementos são descritos nas linhas 86-99, determinando características como: (a) unicidade do elemento (linha 87 e 93) e (b) tipos de nós associados corretamente (linhas 89, 94 e 99).

```
01. MACHINE
02.   NodeInstance
03.
04. SEES
05.   Datatype, NodeDescription
06.
07. VARIABLES
08.   nodeId, /* Identificador único para cada nó */
09.   nodeInstanceName,
10.   nodeInstanceAttributes,
11.   nodeInstanceEdges,
12.   nodeInstanceHyperedges
13.
14. INVARIANT
15.
16.   nodeId <: NAT1 &
17.
18.   nodeInstanceName:  nodeId --> nodeNames &
19.   nodeInstanceAttributes:  nodeId -->
20.                            POW(attributeNames * ENUMVALUES) &
21.   nodeInstanceEdges:  nodeId -->
```

```
22.                        POW(edgeNames * nodeId) &
23.   nodeInstanceHyperedges: nodeId -->
24.                            POW(hyperedgeNames * nodeId) &
25.
26.   /*
27.    * Verificação da cardinalidade e compatibilidade entre
28.    * os nós e seus atributos na estrutura do grafo.
29.    */
30.   !pos . (pos : nodeId =>
31.           !a . (a : dom(nodeInstanceAttributes(pos)) =>
32.                 /*
33.                  * Existencia e unicidade de
34.                  * um atributo em um mesmo nó.
35.                  */
36.                 ((card({a} <| (nodeInstanceAttributes(pos))) = 1) &
37.                 /* Valores dos atributos compativeis */
38.                 (nodeInstanceAttributes(pos)(a) :
39.                  Types(attribute(a))) &
40.                 /* Atributos corretos nos nós */
41.                 (a : nodeattributeNames(nodeInstanceName(pos)))))) &
42.
43.   /*
44.    * Verificação da cardinalidade e compatibilidade entre os nós
45.    * e suas arestas na estrutura do grafo.
46.    */
47.   !pos . (pos : nodeId =>
48.           !e . (e : dom(nodeInstanceEdges(pos)) =>
49.                 /* Unicidade de uma aresta em um mesmo nó */
50.                 (card({e} <| (nodeInstanceEdges(pos))) <= 1) &
51.                 /* Arestas permitidas aos nós. */
52.                 (e : nodeedgeNames(nodeInstanceName(pos))))) &
53.
54.   !pos . (pos : nodeId =>
55.           !e . ((e : dom(nodeInstanceEdges(pos)) &
56.                  card({e} <| (nodeInstanceEdges(pos))) = 1) =>
57.                  /* Valores das arestas compativeis */
58.                  nodeInstanceName(nodeInstanceEdges(pos)(e)) :
59.                   edge(e))) &
60.
61.  /*
62.   * Verificação da cardinalidade e compatibilidade entre os nós
63.   * e suas arestas multivaloradas na estrutura do grafo.
64.   */
65.   !pos . (pos : nodeId =>
66.           !h . (h : (dom(nodeInstanceHyperedges(pos))) =>
67.                 /*
68.                  * Existencia e unicidade de
69.                  * uma aresta em um mesmo nó.
70.                  */
71.                 (card({h} <| (nodeInstanceHyperedges(pos))) <= 1) &
72.                 /* Arestas multivaloradas permitidas aos nós. */
73.                 (h : nodehyperedgeNames(nodeInstanceName(pos))))) &
74.
75.   !pos . (pos : nodeId =>
76.            !h . ((h : (dom(nodeInstanceHyperedges(pos))) &
77.                  (card({h} <| (nodeInstanceHyperedges(pos)))) = 1) =>
78.                   /* Valor do nó lista compatível */
79.                   (nodeInstanceName(nodeInstanceHyperedges(pos)(h)) =
```

```
80.                    hyperedge(h))))
81.
82. INITIALISATION
83.   nodeId,
84.   nodeInstanceName,
85.   nodeInstanceAttributes,
86.   nodeInstanceEdges,
87.   nodeInstanceHyperedges := {},{},{},{},{}
88.
89. OPERATIONS
90.
91.   /* make */
92.
93.   make(Ats, Eds, Hyps, pnN) =
94.    PRE
95.       pnN : nodeNames &
96.       /* Verificar se os valores correspondem aos atributos.*/
97.       Ats <: (attributeNames * ENUMVALUES) &
98.
99.       /*Verifica se os atributos podem ser associados ao nó.*/
100.       !a . (a : dom(Ats) =>
101.            (a : nodeattributeNames(pnN) &
102.             Ats(a) : Types(attribute(a)))) &
103.
104.       /* Verifica se os valores correspondem às arestas.*/
105.       Eds <: (edgeNames * nodeId) &
106.
107.       /* Verifica se as arestas podem ser associadas ao nó.*/
108.       !e . (e : dom(Eds) =>
109.            (e : nodeedgeNames(pnN) &
110.             nodeInstanceName(Eds(e)) : edge(e))) &
111.
112.       /*
113.        * Verificar se os valores correspondem
114.        * às arestas multivaloradas.
115.        */
116.       Hyps <: (hyperedgeNames * nodeId) &
117.
118.       /*
119.        * Verifica se as arestas multivaloradas
120.        * podem ser associadas ao nó.
121.        */
122.       !h . (h : dom(Hyps) =>
123.            (h : nodehyperedgeNames(pnN) &
124.             nodeInstanceName(Hyps(h)) = hyperedge(h)))
125.
126.     THEN
127.       ANY pos WHERE
128.         pos : NAT1 &
129.         pos /: nodeId
130.       THEN
131.         nodeInstanceName := nodeInstanceName \/ {pos |-> pnN} ||
132.         nodeInstanceAttributes := nodeInstanceAttributes \/
133.                                   {pos |-> Ats} ||
134.         nodeInstanceEdges := nodeInstanceEdges \/ {pos |-> Eds} ||
135.         nodeInstanceHyperedges := nodeInstanceHyperedges \/
136.                                   {pos |-> Hyps}
137.       END
```

```
138.      END;
139.
140.    /* set functions */
141.
142.    /*
143.     * pId : nodeId,
144.     * pnA : attributeNames,
145.     * pV : ENUMVALUES
146.     */
147.
148.    setA(pId, pnA, pV) =
149.      PRE
150.        pId : nodeId &
151.        pnA : attributeNames &
152.        pV : ENUMVALUES &
153.        pV : Types(attribute(pnA)) &
154.        pnA : dom(nodeInstanceAttributes(pId))
155.      THEN
156.        nodeInstanceAttributes :=
157.           nodeInstanceAttributes <+ {pId |->
158.                                      (nodeInstanceAttributes(pId) <+
159.                                       {pnA |-> pV})}
160.      END;
161.
162.    /*
163.     * pId : nodeId,
164.     * pnE : edgeNames,
165.     * pV : nodeId
166.     */
167.
168.    setE(pId, pnE, pV) =
169.      PRE
170.        pId : nodeId &
171.        pnE : edgeNames &
172.        pV : nodeId &
173.        nodeInstanceName(pV) : edge(pnE) &
174.        pnE : nodeedgeNames(nodeInstanceName(pId))
175.      THEN
176.        nodeInstanceEdges :=
177.           nodeInstanceEdges <+ {pId |->
178.                                 (nodeInstanceEdges(pId) <+
179.                                  {pnE |-> pV})}
180.     END;
181.
182.    /*
183.     * A restrição dos nós apontados por um
184.     * hyperedge depende do nó lista utilizado.
185.     */
186.
187.    /*
188.     * pId : nodeId,
189.     * pnH : hyperedgeNames,
190.     * pV : nodeId -> List node
191.     */
192.
193.    setH(pId, pnH, pV) =
194.      PRE
195.        pId : nodeId &
```

```
196.       pnH : hyperedgeNames &
197.       pV : nodeId &
198.       nodeInstanceName(pV) = hyperedge(pnH) &
199.       pnH : nodehyperedgeNames(nodeInstanceName(pId))
200.     THEN
201.       nodeInstanceHyperedges :=
202.          nodeInstanceHyperedges <+ {pId |->
203.                                    (nodeInstanceHyperedges(pId) <+
204.                                     {pnH |-> pV})}
205.     END;
206.
207.   /* query functions */
208.
209.   /*
210.    * pId -> nodeId,
211.    * pnA -> attributeNames
212.    */
213.
214.   respA <-- queryAttribute(pId, pnA) =
215.     PRE
216.       pId : nodeId &
217.       pnA : attributeNames &
218.       pnA : dom(nodeInstanceAttributes(pId))
219.     THEN
220.       respA := nodeInstanceAttributes(pId)(pnA)
221.     END;
222.
223.   /*
224.    * pN -> nodeId,
225.    * pnE -> edgeNames
226.    */
227.
228.   respE <-- queryEdge(pId, pnE) =
229.     PRE
230.       pId : nodeId &
231.       pnE : edgeNames &
232.       pnE : nodeedgeNames(nodeInstanceName(pId)) &
233.       card({pnE} <| (nodeInstanceEdges(pId))) = 1
234.     THEN
235.       respE := nodeInstanceEdges(pId)(pnE)
236.     END;
237.
238.   /*
239.    * pId -> nodeId,
240.    * pnH -> hyperedgeNames
241.    */
242.
243.   respH <-- queryHyperedge(pId, pnH) =
244.     PRE
245.       pId : nodeId &
246.       pnH : hyperedgeNames &
247.       pnH : nodehyperedgeNames(nodeInstanceName(pId)) &
248.       card({pnH} <| (nodeInstanceHyperedges(pId))) = 1
249.     THEN
250.       respH := nodeInstanceHyperedges(pId)(pnH)
251.     END
252. END
```

Apêndice B

Especificação gráfica - Diagramas UML

Neste apêndice, são fornecidos os diagramas UML construídos para obter a especificação gráfica de AGraph.

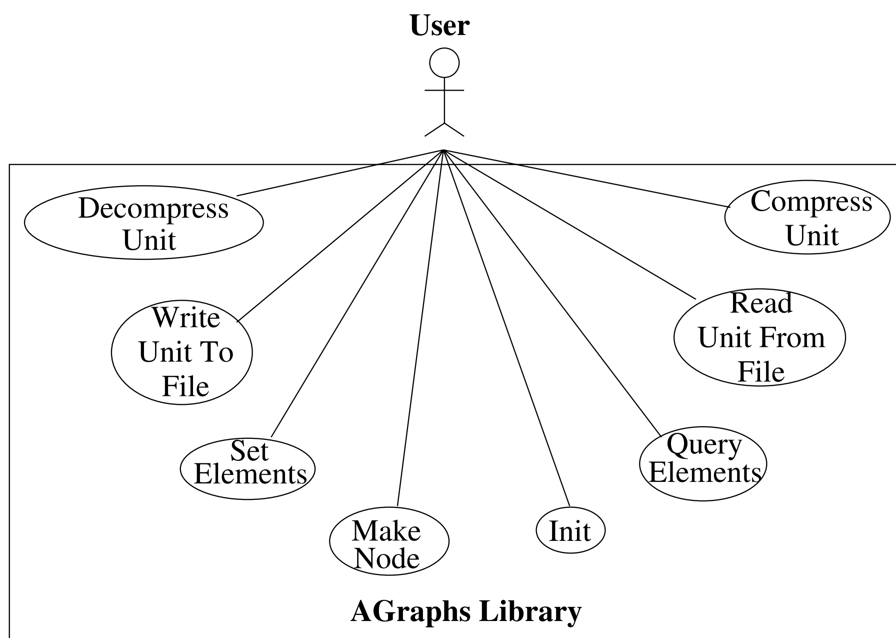


Figura B.1: *Diagrama UML de Casos de Uso* contendo as funções das bibliotecas AGraphs.

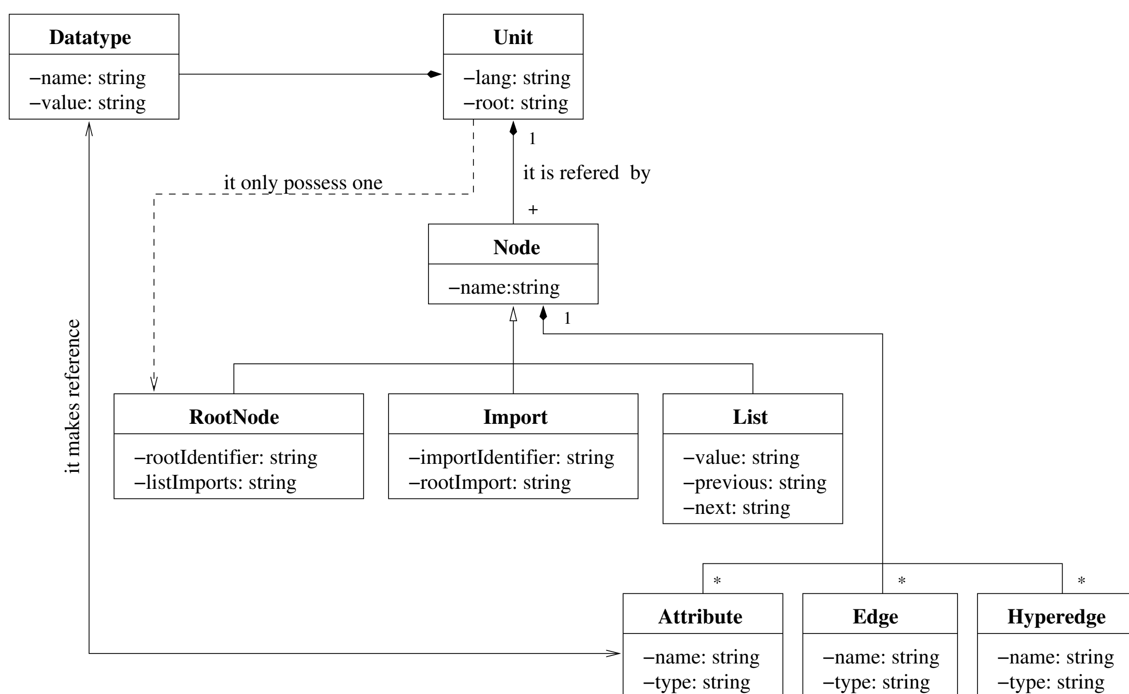


Figura B.2: *Diagrama UML de Classes* descrevendo os elementos que compõem os esquemas AGraphs.

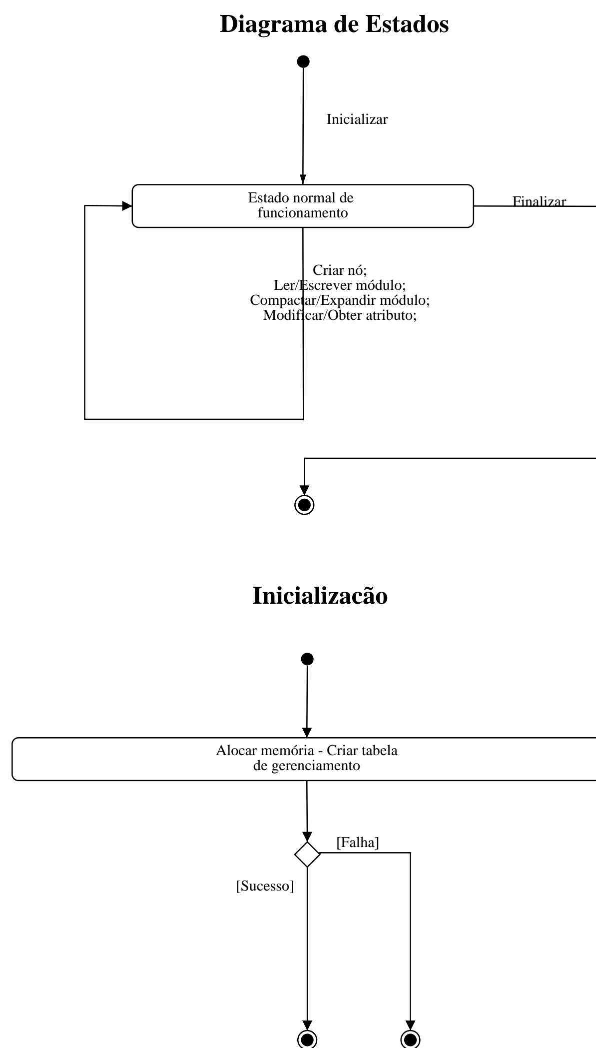


Figura B.3: *Diagrama UML de Estados* do AGraph e de *Atividades* da função de inicialização da biblioteca.

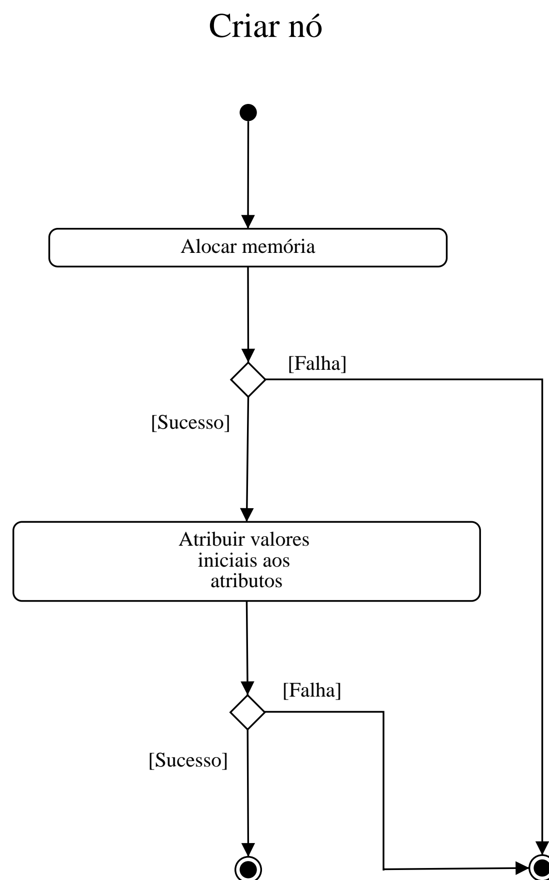


Figura B.4: *Diagrama UML de Atividades* da função de criação de nós (`make`).

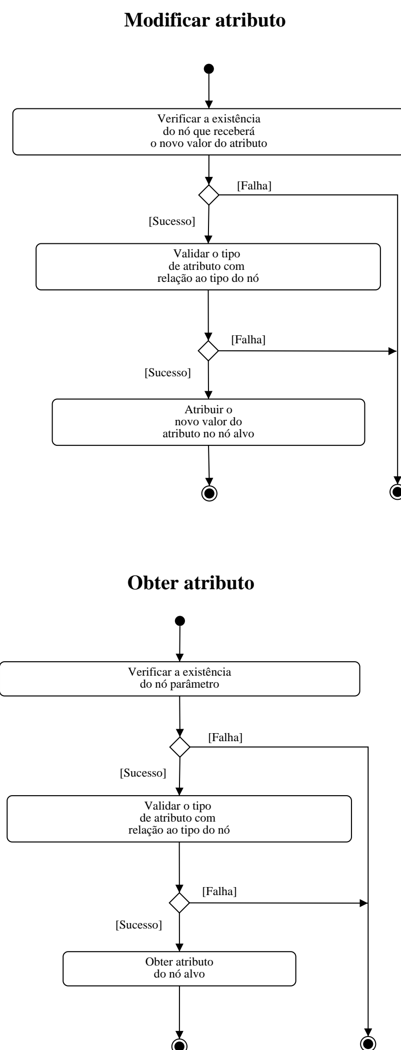


Figura B.5: *Diagramas UML de Atividade* das funções acessores (`set` e `get`).

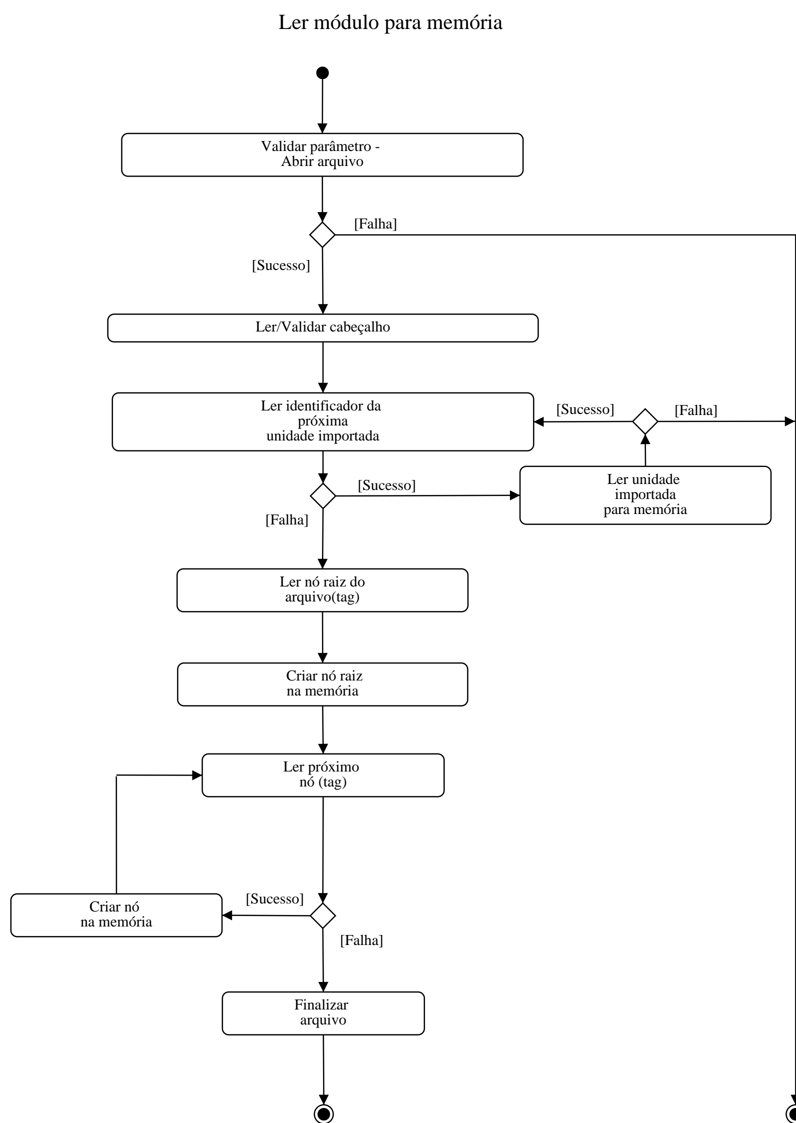


Figura B.6: *Diagrama UML de Atividades* da função de leitura de uma unidade (`read`).

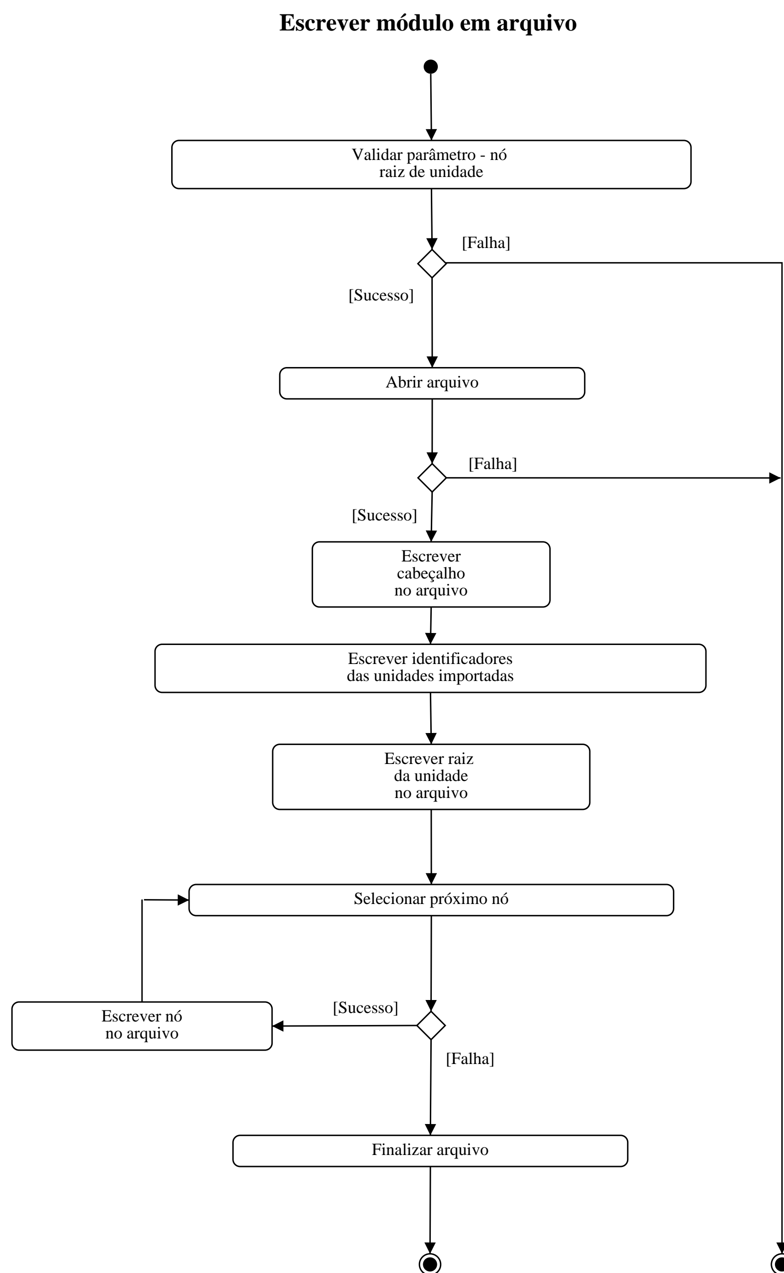


Figura B.7: *Diagrama UML de Atividades* da função de escrita de uma unidade (`write`).

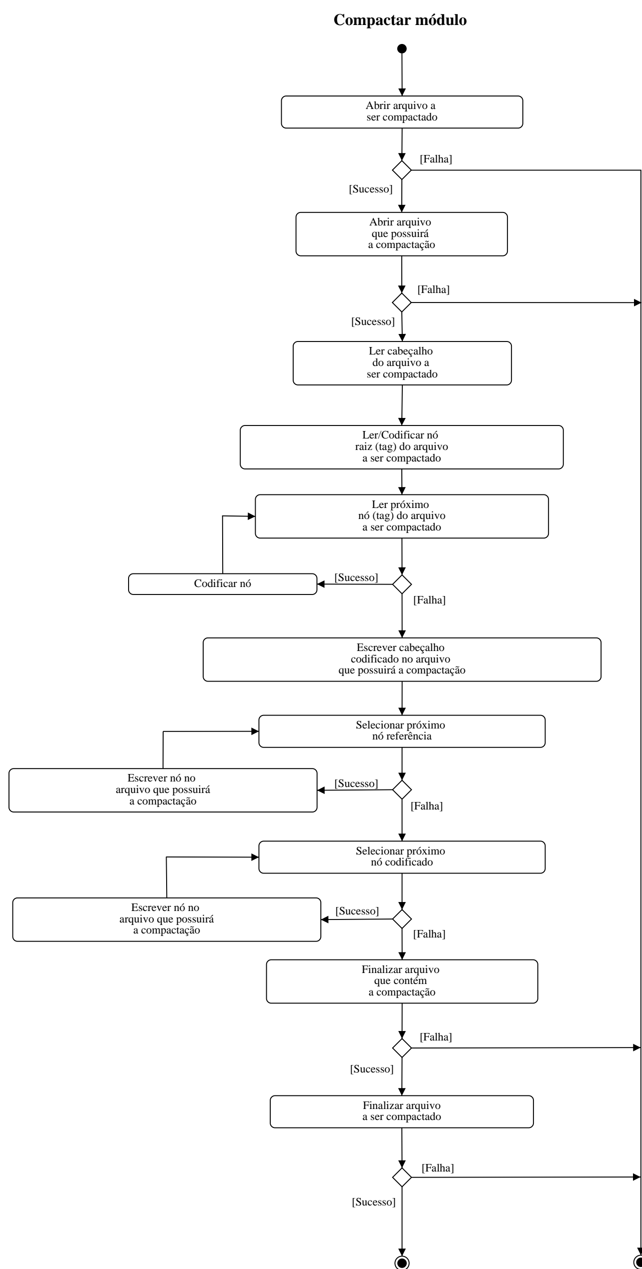


Figura B.8: *Diagrama UML de Atividades* da função de compactação (`compress`).

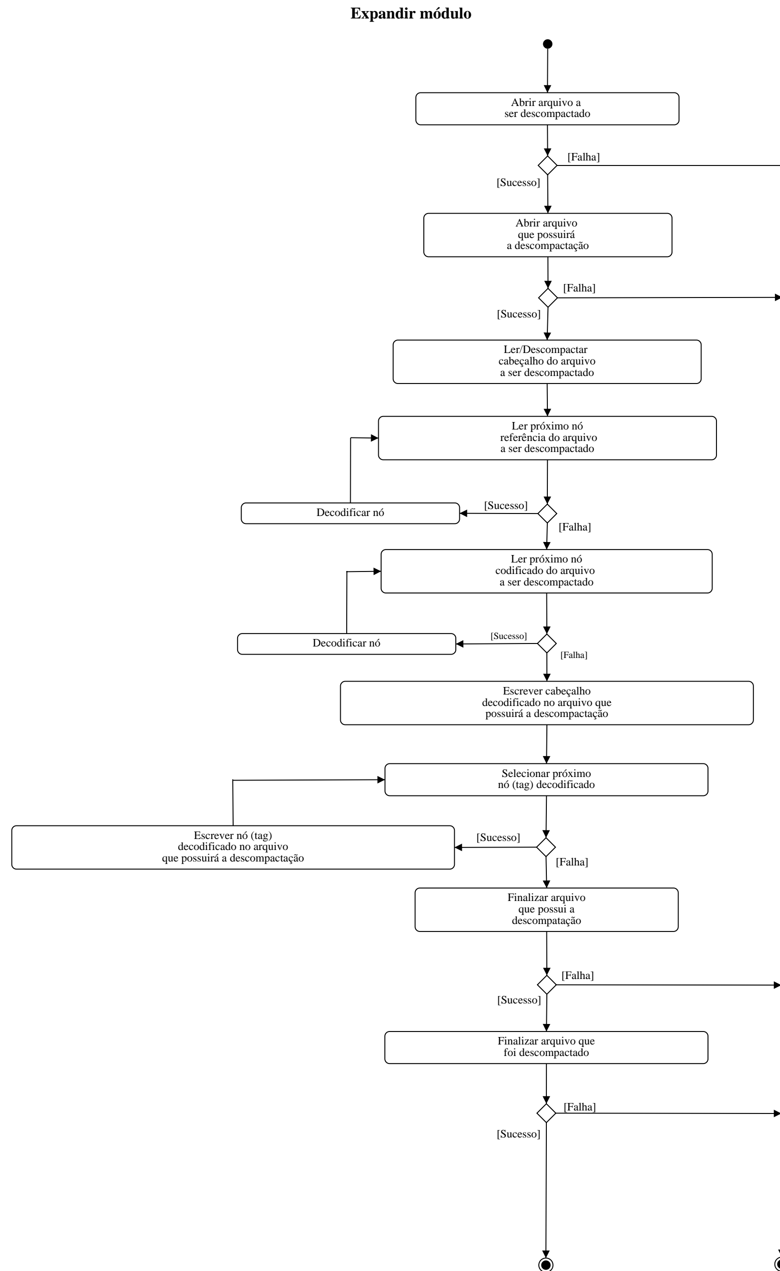
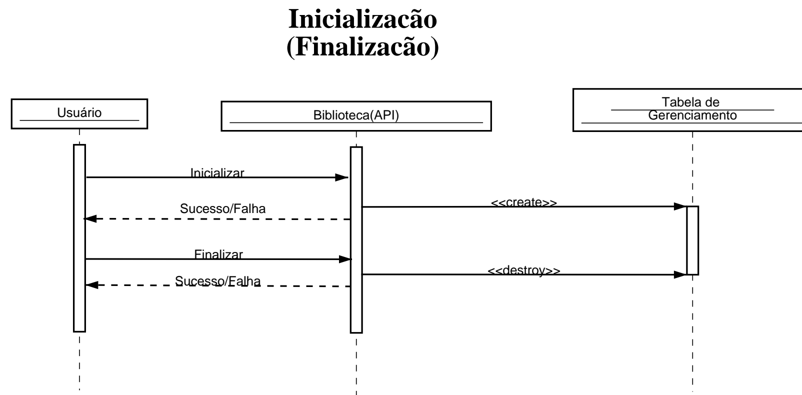


Figura B.9: *Diagrama UML de Atividades* da função de descompactação (`decompress`).



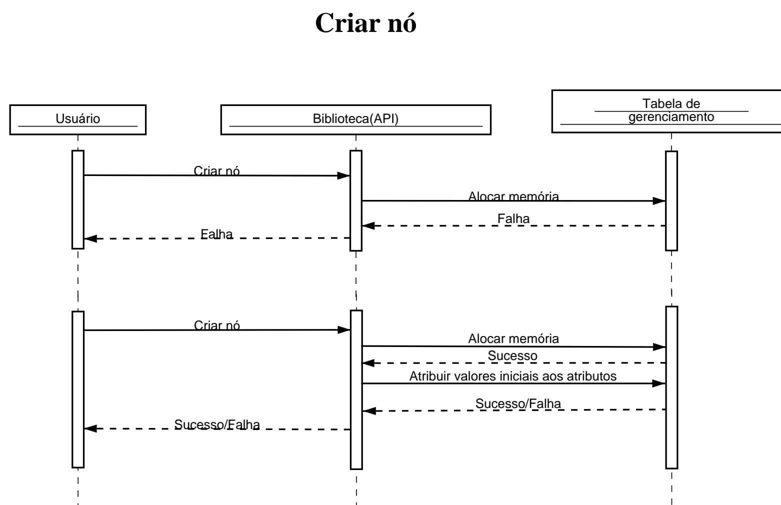
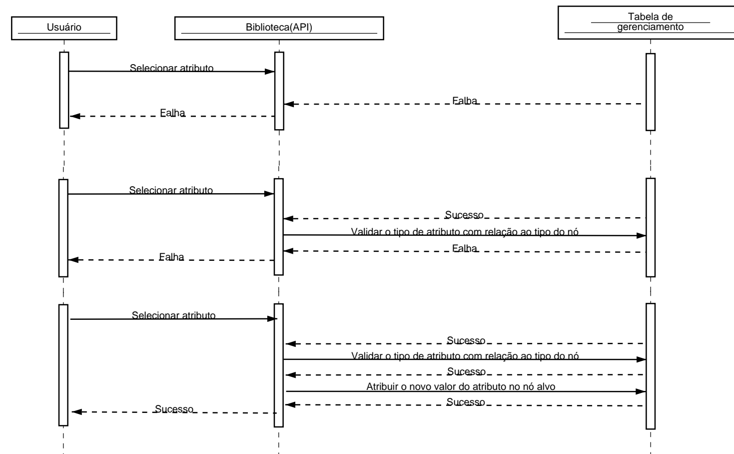


Figura B.10: *Diagramas UML de Sequência* das funções de inicialização (`init`) e criação de um nó (`make`).

Modificar atributo



Obter atributo

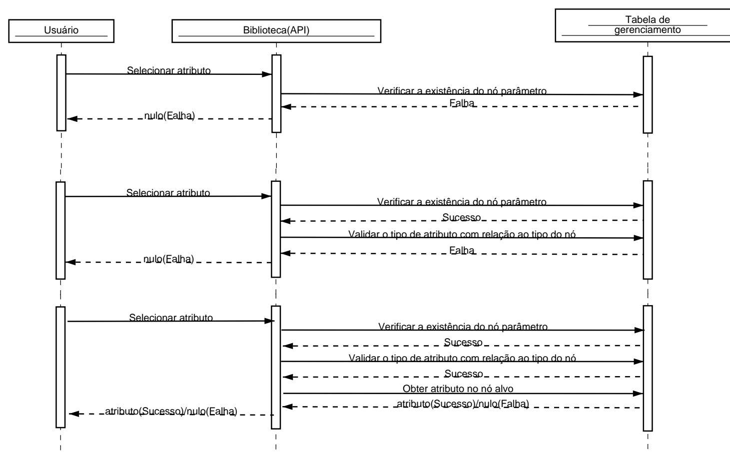


Figura B.11: *Diagramas UML de Sequência* das funções acessores (`set` e `get`).

Figura B.12: *Diagrama UML de Sequência* da função de leitura de uma unidade (`read`).

Escrever módulo em arquivo

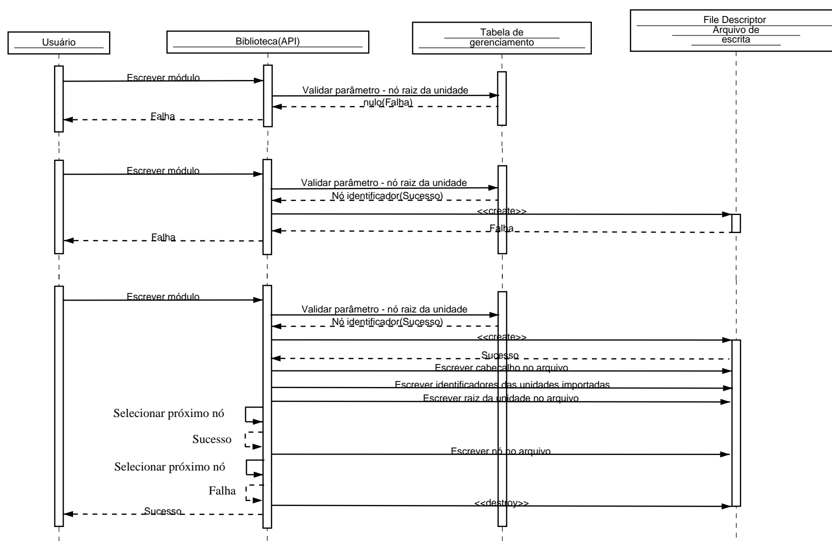


Figura B.13: *Diagrama UML de Sequência* da função de escrita de uma unidade (`write`).

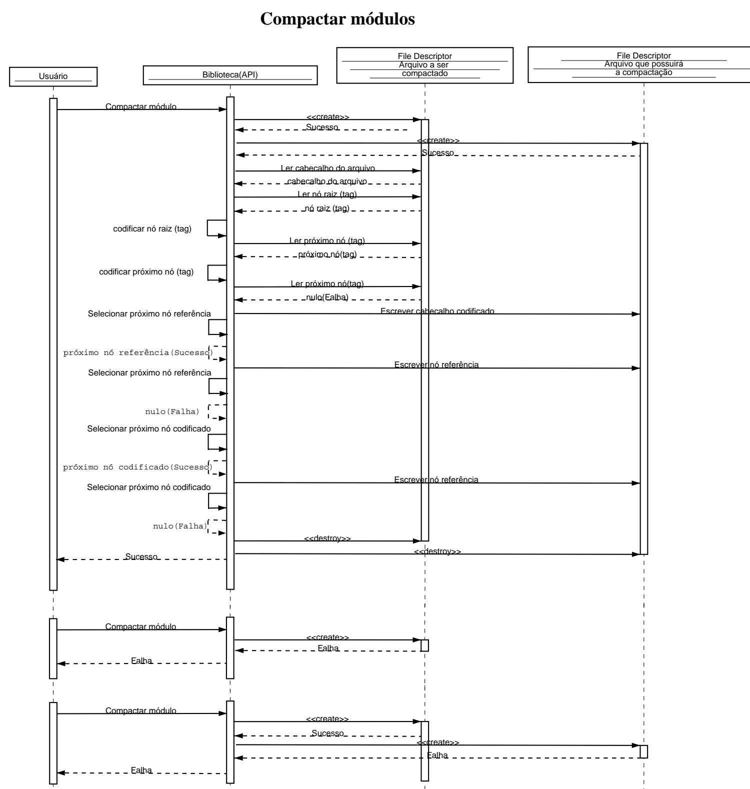


Figura B.14: *Diagrama UML de Sequência* da função de compactação de uma unidade (`compress`).

Figura B.15: *Diagrama UML de Sequência* da função de descompactação de uma unidade (`decompress`).

Apêndice C

Linguagem de entrada para o gerador

A descrição do *XML Schema* para a sublinguagem XML aceita como entrada para o gerador automático da API é fornecida a seguir. Este contém duas seções: a seção de definição da estrutura, e a definição dos tipos de dados. A seção de definição da estrutura declara as tags (elementos) que aparecem no arquivo XML de entrada. Os tipos de dados descrevem os possíveis conteúdos das tags.

```
/*************************************************************/
<?xml version="1.0"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
   targetNamespace="http://www.w3.org/2001/XMLSchema">

  <element name="Unit" type="unit_node" />
  <element name="DataType" type="datatype_node" />
  <element name="Node" type="node_node" />
  <element name="RootNode" type="rootnode_node" />
  <element name="Import" type="import_node" />
  <element name="List" type="list_node" />
  <element name="Attribute" type="attribute_node" />
  <element name="Edge" type="edge_node" />
  <element name="Hyperedge" type="hyperedge_node" />

  <complexType name="unit_node" >
     <sequence minOccurs="1" maxOccurs="unbounded">
        <choice>
          <element ref="DataType" />
          <element ref="Node" />
          <element ref="Import" />
        </choice>
          <element ref="List" minOccurs="1" />
          <element ref="RootNode" minOccurs="1"
                                  maxOccurs="1" />
     </sequence>

     <attribute name="lang" type="string" use="required" />
     <attribute name="root" type="string" use="required" />

  </complexType>

  <complexType name="datatype_node" >
     <attribute name="name" type="string" use="required" />
     <attribute name="value" type="string" use="required" />
  </complexType>
```

```
<complexType name="node_node" >
   <sequence minOccurs="0" maxOccurs="unbounded">
      <choice>
        <element ref="Attribute" />
        <element ref="Edge" />
        <element ref="Hyperedge" />
      </choice>
   </sequence>

   <attribute name="name" type="string" use="required" />

</complexType>

<complexType name="rootnode_node" >
   <sequence minOccurs="0" maxOccurs="unbounded">
      <choice>
        <element ref="Attribute" />
        <element ref="Edge" />
        <element ref="Hyperedge" />
      </choice>
   </sequence>

   <attribute name="name" type="string"
  use="required" />
   <attribute name="rootIdentifier" type="string"
            use="required" />
   <attribute name="listImports" type="string"
                                 use="required" />
</complexType>

<complexType name="import_node" >
   <sequence minOccurs="0" maxOccurs="unbounded">
      <choice>
        <element ref="Attribute" />
        <element ref="Edge" />
        <element ref="Hyperedge" />
      </choice>
   </sequence>

   <attribute name="name" type="string"
                       use="required" />
   <attribute name="importIdentifier" type="string"
                                      use="required" />
   <attribute name="rootImport" type="string"
                                use="required" />
</complexType>

<complexType name="list_node" >
   <sequence minOccurs="0" maxOccurs="unbounded">
      <choice>
         <element ref="Attribute" />
         <element ref="Edge" />
         <element ref="Hyperedge" />
      </choice>
   </sequence>

   <attribute name="next" type="string" use="required" />
```

```
      <attribute name="previous" type="string" use="required" />
      <attribute name="value" type="string" use="required" />

   </complexType>

   <complexType name="attribute_node" >
      <attribute name="name" type="string" use="required" />
      <attribute name="type" type="string" use="required" />
   </complexType>

   <complexType name="edge_node" >
      <attribute name="name" type="string" use="required" />
      <attribute name="type" type="string" use="required" />
   </complexType>

   <complexType name="hyperedge_node" >
      <attribute name="name" type="string" use="required" />
      <attribute name="type" type="string" use="required" />
   </complexType>

</schema>

/************************************************************/
```