Alessandro Mendonça Maciel

Desenvolvimento de uma ferramenta computacional para modelagem e conversão de dados de fluxo de potência em sistemas elétricos para um padrão XML

São Carlos

Junho de 2010

Alessandro Mendonça Maciel

Desenvolvimento de uma ferramenta computacional para modelagem e conversão de dados de fluxo de potência em sistemas elétricos para um padrão XML

Trabalho de conclusão de curso apresentado ao Departamento de Engenharia Elétrica da Escola de Engenharia de São Carlos da Universidade de São Paulo como parte dos requisitos para a conclusão do curso de Engenharia Elétrica com ênfase em Sistemas de Energia e Automação.

Orientador: Luís Fernando Costa Alberto

São Carlos

Junho de 2010

Ficha catalográfica preparada pela Seção de Tratamento
da Informação do Serviço de Biblioteca – EESC/USP

M152d

Maciel, Alessandro Mendonça
Desenvolvimento de uma ferramenta computacional para modelagem e conversão de dados de fluxo de potência em sistemas elétricos para um padrão XML / Alessandro Mendonça Maciel ; orientador Luís Fernando Costa Alberto. -- São Carlos, 2010.

Trabalho de Conclusão de Curso (Graduação em Engenharia Elétrica com ênfase em Sistemas de Energia e Automação) -- Escola de Engenharia de São Carlos da Universidade de São Paulo, 2010.

1. Conversão de dados. 2. Linguagem XML. 3. ANAREDE. 4. Fluxo de potência. I. Título.

À minha família e amigos.

Resumo

A linguagem XML (*Extensible Markup Language*) consiste em uma linguagem de marcação que permite a leitura de dados estruturados, possibilitando declarações precisas do conteúdo desses dados, que podem ser tabelas, figuras ou, no contexto deste trabalho, parâmetros que descrevem um sistema elétrico.

O objetivo principal desse trabalho é criar uma integração entre a linguagem XML e a linguagem do software de simulação de fluxo de potência ANAREDE, que é largamente utilizado no Brasil por concessionárias de energia elétrica, empresas de consultoria em engenharia elétrica e equipes de pesquisa, facilitando a leitura e escrita de dados para este software.

Para atingir tal objetivo, foi desenvolvida uma aplicação na linguagem de programação C++, que suporta uma programação orientada a objetos, característica que foi largamente utilizada durante a implementação do código fonte da aplicação.

Esta aplicação consiste em um conversor de dados, que "traduz" a linguagem padrão do arquivo de entrada de dados do software ANAREDE em um padrão XML que descreve um sistema elétrico, assim como também faz o caminho inverso, ou seja, "traduz" um código que representa um sistema elétrico no padrão XML para o padrão de entrada de dados do ANAREDE.

O uso de uma linguagem de programação orientada a objetos permite que o código fonte desenvolvido possa ser facilmente expandido. Logo, o conversor de dados pode futuramente ser ampliado de forma a permitir que a conversão seja feita entre outras linguagens diferentes do padrão XML ou do padrão de entrada de dados do ANAREDE.

Abstract

The XML language (*Extensible Markup Language*) consists in a markup language that allows the reading of strutured data, allowing precise declarations of its content, which could be tables, figures or, in this project´s context, parameters that describe an electric system.

The main goal of this project is to create an integration between the XML language and the language of the load flow simulation software ANAREDE, that is largely used in Brazil by electric energy companies, electrical engeneering consulting companies and research groups, favoring the reading and writing data for this software.

To reach that goal, an application was developed in the C++ programming language, which supports an object-oriented programming, feature that was largely explored in the implementation of the application´s source code.

This application consists of a data converter, that "translates" the standard language of the ANAREDE's file data entry into a XML pattern which describes an electric system, as well as it does the opposite way, in other words, "translates" a code that represents an electric system in the XML pattern into ANAREDE's data entry pattern.

The use of an object-oriented programming language, allows the developed source can be easily expanded. Hence, the data converter can be expanded in the future to convert data between other languages in addition to the XML pattern or the ANAREDE's data entry pattern.

Sumário

Lista de Figuras **viii**

Lista de Tabelas **x**

1 Introdução **1**

1.1 Organização do documento 2

2 Conceitos de programação **4**

2.1 Encapsulação e classes 4

2.2 Alocação dinâmica 5

2.2.1 *Arrays* alocados dinamicamente 5

2.2.2 Listas encadeadas 6

3 Modelagem da estrutura de dados **8**

3.1 O sistema elétrico em uma estrutura de dados 8

4 O padrão de entrada de dados do ANAREDE **13**

4.1 Barras no padrão ANAREDE 13

4.2 Linhas no padrão ANAREDE 14

5 A linguagem XML **16**

5.1 Leitura de documentos XML 16

5.2 O XML Parser 18

5.3 Dados do sistema elétrico em XML 18

5.3.1 A *tag* buses 19

5.3.2 A *tag* lines 19

5.3.3 A *tag* topology 20

6 Documentação das classes **24**

6.1 A classe barra 24

6.1.1 Funções membro da classe barra 25

6.2 A classe dados_lin 27

6.2.1 Funções membro da classe dados_lin 27

6.3 A classe ramos 29

6.3.1 Funções membro da classe ramos 30

6.4 A classe sistema 31

6.4.1 Funções membro da classe sistema 31

7 Conclusão **34**

Apêndice A – Apresentação de resultados **35**

Referências **38**

Lista de Figuras

1.1 Esquema geral da aplicação desenvolvida. 2

2.1 Exemplo genérico do arranjo de memória de uma lista. 7

3.1 Sistema simples de três barras. 8

3.2 Nó da lista encadeada de ramos. 9

3.3 Estrutura de dados após os dados de uma linha serem adicionados. 10

3.4 Estrutura de dados após os dados de duas linhas serem adicionados. 10

3.5 Estrutura de dados após os dados de três linhas serem adicionados. 11

3.6 Estrutura de dados clássica. 12

4.1 Barras no padrão de entrada de dados do ANAREDE. 14

4.2 Linhas no padrão de entrada de dados do ANAREDE. 15

5.1 Exemplo simples de XML. 16

5.2 Exemplo de XML com *child tags*. 17

5.3 Exemplo de XML com *child tags* mais estruturado. 17

5.4 Exemplo de XML utilizando atributo. 17

5.5 Exemplo de sistema elétrico escrito em XML. 19

5.6 Exemplo da *tag* buses. 20

5.7 Exemplo da *tag* bus. 20

5.8 Exemplo de uma *tag* lines. 22

5.9 Exemplo de uma *tag* line. 22

5.10 Exemplo de uma *tag* topology. 23

5.11 Exemplo de uma *tag* branch. 23

6.1 Fluxograma básico da aplicação. 33

A.1 Sistema no padrão de entrada de dados do ANAREDE. 35

A.2 Conversão para XML - Parte 1. 36

A.3 Conversão para XML - Parte 2. 37

Lista de Tabelas

4.1 Padrão ANAREDE para dados de barra. 14

4.2 Padrão ANAREDE para dados de linha. 15

5.1 Descrição das *child tags* contidas em bus. 21

5.2 Descrição das *child tags* contidas em line. 21

6.1 Descrição das variáveis da classe barra. 25

6.2 Descrição das funções *set* contidas na classe barra. 26

6.3 Descrição das funções *get* contidas na classe barra. 26

6.4 Descrição das variáveis da classe dados_lin. 28

6.5 Descrição das funções *set* contidas na classe dados_lin. 28

6.6 Descrição das funções *get* contidas na classe dados_lin. 28

6.7 Descrição das variáveis da classe ramos. 30

6.8 Descrição das funções *get* contidas na classe ramos. 30

1 Introdução

Programas de simulação tem se tornado cada vez mais importantes em estudos de sistemas elétricos. Um desses simuladores que se destaca pela grande importância é o software de análise de redes ANAREDE, especializado em simulações de fluxo de carga e fluxo de potência.

O ANAREDE foi desenvolvido de forma a permitir simulações de grandes sistemas de transmissão de energia elétrica. No Brasil, grande parte do sistema interligado nacional está documentado na forma de arquivos de entrada do ANAREDE, o que torna este software ainda mais essencial em estudos de sistemas elétricos de potência.

No entanto, um problema enfrentado pelos usuários desse software é seu padrão de entrada de dados. As características dos sistemas elétricos são armazenadas em arquivos de texto pouco amigáveis e muito susceptíveis a erros de sintaxe, uma vez que os dados são armazenados de forma que um simples espaço a mais entre os dados pode acarretar em falha do ANAREDE.

Para facilitar o desenvolvimento e a modificação desse padrão de entrada, desenvolveu-se nesse trabalho uma ferramenta computacional que "traduz" o código do ANAREDE para um padrão XML, assim como também realiza o caminho inverso, ou seja, "traduz" o padrão XML para o código do arquivo de entrada de dados do ANAREDE. Todavia, esse trabalho se preocupa apenas em manipular os dados de linhas e barras do sistema, que correspondem aos dados de maior quantidade na maioria dos sistemas elétricos, ou seja, na maioria dos casos o número de barras e linhas é significativamente maior que o número de geradores, por exemplo.

Para realizar essa conversão, foi implementada uma estrutura de dados chamada EPSD (*Electric Power System Data*), previamente desenvolvida no LACOSEP (Laboratório de Análise Computacional de Sistemas Elétricos de Potência) na Escola de Engenharia de São Carlos, onde os dados do sistema elétrico são gravados e lidos de forma segura e prática. Assim sendo, a figura 1.1 mostra um esquema geral da aplicação desenvolvida.

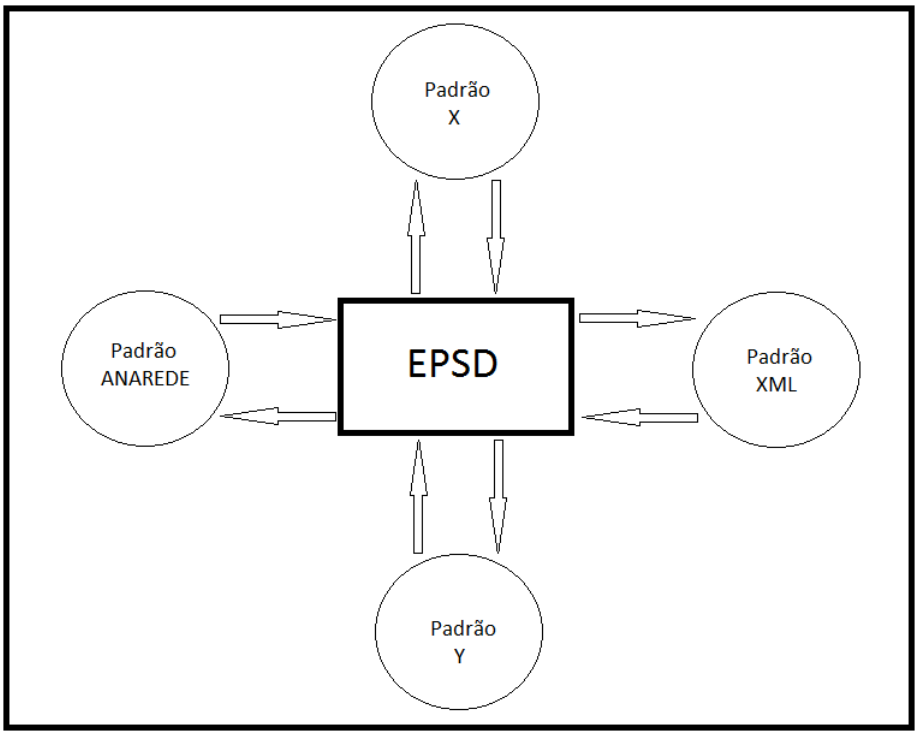

Figura 1.1: Esquema geral da aplicação desenvolvida.

Como sugere a figura 1.1, a estrutura de dados é parte essencial do desenvolvimento do software, assim como também percebe-se que a aplicação pode ser futuramente ampliada. No entanto, este trabalho tratou apenas da conversão de dados entre o padrão XML e o padrão de entrada de dados do ANAREDE.

1.1 Organização do documento

Nos próximos capítulos e seções serão abordados conteúdos que foram considerados essenciais para o entendimento do projeto como um todo.

No capítulo 2 serão discutidos alguns conceitos de programação que foram largamente utilizados e que são essenciais para o entendimento da estrutura de dados usada no desenvolvimento da aplicação.

No capítulo 3 será mostrada a modelagem dos dados do sistema elétrico na estrutura de dados, sendo mostrado passo-a-passo a inserção dos dados na mesma e quais as vantagens dessa modelagem.

No capítulo 4 será apresentado o padrão do arquivo de entrada de dados do ANAREDE, assim como uma breve discussão sobre as desvantagens do código.

O capítulo 5 contém uma breve introdução à linguagem XML, que basicamente mos-

tra como ler documentos XML e informa como os dados de um sistema elétrico foram organizados em um padrão XML.

A documentação das classes contidas no código fonte da aplicação está apresentada no capítulo 6 e, por fim, o capítulo 7 apresenta as conclusões finais sobre o desenvolvimento do trabalho.

2 Conceitos de programação

A linguagem C++ é um aprimoramento de sua linguagem antecessora, a linguagem C ANSI. O objetivo desse aprimoramento foi permitir que programadores possam gerenciar e compreender programas maiores e mais complexos. Dentre muitas das vantagens do uso da linguagem C++ podem ser citadas flexibilidade, legibilidade, facilidade de manutenção e suporte à programação orientada a objetos. A principal vantagem dessa linguagem que levou a mesma a ser escolhida para a realização desse trabalho foi a possibilidade da programação orientada a objetos. Essa abordagem de organização é essencialmente diferente do desenvolvimento tradicional de software, onde estruturas de dados e rotinas são desenvolvidas apenas de forma fracamente acopladas [8].

O objetivo deste capítulo não é descrever os detalhes de programação da linguagem, mas sim apresentar alguns conceitos de programação que são suportados pela linguagem C++ e que foram essenciais para o desenvolvimento do trabalho.

2.1 Encapsulação e classes

Uma classe em C++ é uma estrutura de dados onde ficam alocadas as variáveis de determinada aplicação e também as funções (chamadas de funções membro) que manipulam essas variáveis. Essas funções executam as tarefas inerentes à classe e somente elas tem acesso às variáveis da classe[1]. Isso faz com que os aspectos externos de uma classe, os quais são acessíveis às outras classes, sejam separados dos detalhes internos de implementação, formando um conceito chamado encapsulação ou *esconder informação*.

O uso de encapsulação permite que a implementação de um objeto possa ser modificada sem afetar as aplicações que usam esse objeto [8].

Logo, o uso de classes faz com que o código do programa possa ser mais facilmente

[1]Em C++ existe o conceito de amizade entre classes. Classes amigas permitem o acesso livre das variáveis entre si. Esse conceito ainda é muito discutido entre programadores e é considerado por muitos inadequado por "ferir" os conceitos de abstração e encapsulação de dados.

modificado se comparado à códigos de linguagens que não suportam orientação a objetos. Razões para modificações poderiam ser correção de erros, melhoria de desempenho, mudança da plataforma de execução, entre outras. Além disso, evita erros que poderiam ser facilmente cometidos em outras linguagens não-orientadas a objetos. Por exemplo, no contexto desse projeto, existe uma classe de objetos que manipula as variáveis de barra e uma que manipula as variáveis de linha. Não seria possível o programador modificar um dado de barra através de uma função da classe que manipula as variáveis de linha, evitando um erro que poderia ser de difícil detecção.

Erros como esse podem ser comuns em programas com códigos grandes e com muitas variáveis, mas são praticamente impossíveis de serem cometidos usando classes.

Portanto, o uso de classes nesse projeto foi de fundamental importância, visto que a expansão das funções do mesmo se torna muito mais fácil dessa forma, permitindo que futuramente a conversão de dados seja expandida para outros softwares além do ANAREDE, aproveitando as funções que já foram desenvolvidas.

Exemplos práticos e dicas de implementação são largamente expostas por Deitel[4] e foram bastante utilizadas no decorrer do trabalho.

2.2 Alocação dinâmica

A alocação dinâmica, que é prática muito comum na programação, permite que as variáveis sejam alocadas na memória quando o software está sendo executado e liberar essa memória quando as variáveis não forem mais usadas, ao invés de ter essa memória pré-definida na compilação do código.

Essa prática não é exclusiva das linguagens orientadas a objetos, mas é essencial em todas as aplicações em que o número de dados que serão processados pelo programa não é conhecido.

2.2.1 *Arrays* alocados dinamicamente

Um *array* de dados consiste em uma estrutura de dados simples, que contém vários elementos que podem ser acessados por meio de um índice.

Logo, a seguinte declaração cria estaticamente um *array* de números inteiros de dez elementos:

int vetor[10];

Quando for acessado o elemento vetor[0], será retornado o primeiro elemento do *array*, vetor[1] o segundo elemento e assim por diante até o vetor[9], que constitui o último elemento.

O problema dessa declaração é que o número de dados que serão utilizados foi limitado em dez. Além disso, se apenas dois elementos forem requeridos, ocorre um desperdício de memória, uma vez que foi reservada memória para dez elementos.

No caso da aplicação desenvolvida nesse trabalho, essa declaração é inviável, já que podem existir desde sistemas elétricos com poucas unidades de barras até sistemas com milhares de barras. Além disso, o compilador possui uma memória reservada limitada para variáveis estáticas, tornando necessária a alocação dinâmica [4].

Logo, os dados de barra são alocados em um *array* criado dinamicamente, ou seja, quando o software está sendo executado e já é conhecido o número de barras do sistema, então a mémoria é alocada de acordo com esse número e o *array* é criado. Um exemplo de declaração de um array alocado dinamicamente está apresentado a seguir:

barra *array = new barra [numero_de_barras];

No exemplo citado, ocorre a criação de um array de objetos da classe barra, onde o número de barras é definido em tempo de execução.

Aplicações de *arrays* alocados dinamicamente e vários exemplos de implementação podem ser encontrados em [3].

2.2.2 Listas encadeadas

Assim como um *array* alocado dinamicamente, a lista encadeada é uma estrutura de dados flexível na qual a memória total alocada é sempre proporcional ao número de elementos da mesma. No entanto, no caso de listas encadeadas, elementos (chamados comumente de nós) podem ser removidos ou adicionados à lista em qualquer ponto, o que não é possível com os *arrays* que uma vez criados terão sempre o mesmo número de elementos[5].

Uma lista encadeada, porém, não possui um espaço contíguo na memória e portanto não pode ser acessada através de índices como os *arrays*. Logo, cada elemento de uma lista encadeada possui necessariamente um ponteiro[2] que aponta para o próximo elemento

[2]Um ponteiro é uma variável que contém um endereço de memória.

da lista, formando uma estrutura onde os dados são encadeados[3]. Para percorrer uma lista, é necessário conhecer o endereço de memória do primeiro elemento da lista, ou seja, ter um ponteiro que aponta para o primeiro elemento da lista. A figura 2.1 mostra um exemplo genérico do arranjo de memória de uma lista.

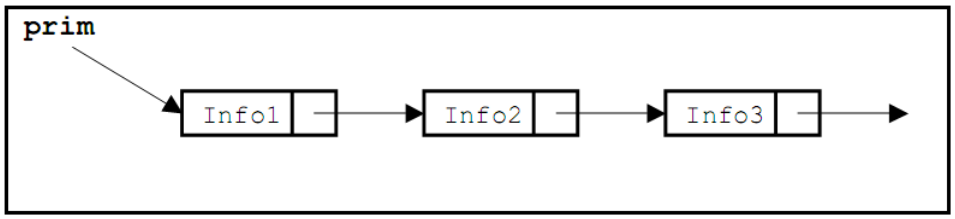

Figura 2.1: Exemplo genérico do arranjo de memória de uma lista.

No contexto do trabalho, a flexibilidade das listas encadeadas foi utilizada para criar uma lista de dados das linhas e listas de conexões entre as linhas e barras à medida em que os dados são lidos do arquivo. A maneira como essas listas foram organizadas será melhor discutida no capítulo 3.

[3]No fim de uma lista encadeada, o elemento aponta para NULL, que representa um endereço nulo em C++.

3 Modelagem da estrutura de dados

Para que os dados do sistema elétrico sejam convertidos, seja para o padrão XML ou seja para o padrão ANAREDE, é necessário que antes eles sejam alocados numa estrutura de dados (EPSD), onde serão organizados de maneira a ficarem de alguma forma conectados, permitindo assim o acesso aos elementos de forma segura.

Como informado no capítulo 2, os dados de barra são alocados em um *array* criado dinamicamente, enquanto os dados de linha e ramos (conexões entre as linhas e barras) do sistema são alocados em listas encadeadas. Este capítulo visa definir como essas estruturas estão acopladas entre si.

3.1 O sistema elétrico em uma estrutura de dados

Para ilustrar como os dados do sistema são alocados na estrutura de dados criada pelo programa, será considerado o sistema da figura 3.1, que é um sistema bastante simples de apenas três barras e três linhas.

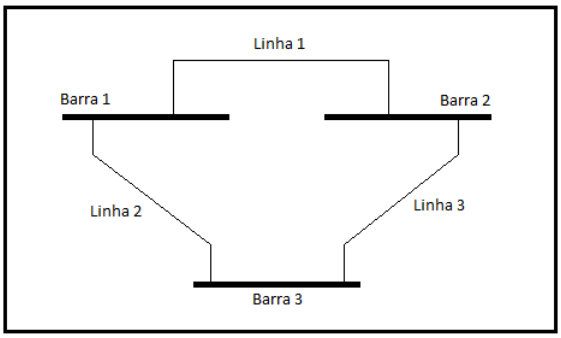

Figura 3.1: Sistema simples de três barras.

Conclui-se da figura do sistema, que o mesmo possuirá um *array* de barras com três elementos e uma lista encadeada de dados de linhas também com três elementos. As estruturas mais complexas, no entanto, são as listas encadeadas que mostram as conexões do sistema. Cada elemento dessas listas possui três ponteiros: um para a barra de destino,

um para os dados da linha que interliga as barras e um para o próximo elemento da lista. Além disso, o número de listas de ramos é proporcional ao número de barras, ou seja, cada elemento do *array* de barras possui um ponteiro para o primeiro elemento de uma lista encadeada de ramos, que permite saber quais são as barras vizinhas e quais são as linhas que fazem as ligações. A figura 3.2 mostra um exemplo genérico de um elemento da lista encadeada de ramos.

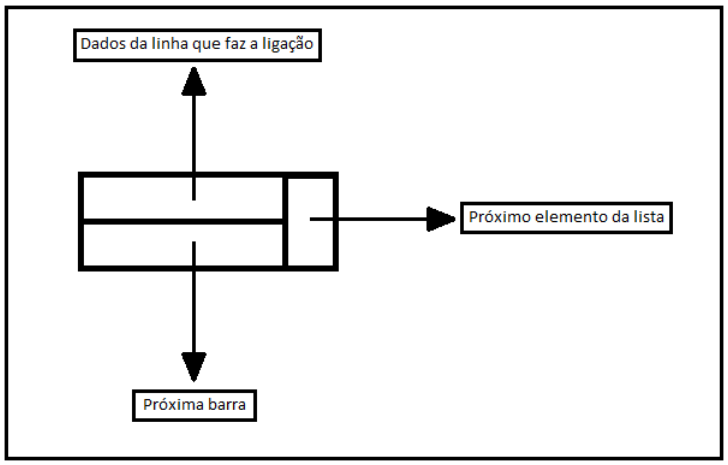

Figura 3.2: Nó da lista encadeada de ramos.

Inicialmente, o *array* de barras é criado e os elementos do mesmo são preenchidos com os dados das barras do sistema. Uma vez que um *array* depois de criado não permite que elementos adicionais sejam inseridos, antes que possam ser criadas as listas de dados de linha e ramos, o *array* de barras deve estar necessariamente criado e preenchido com os dados das barras do sistema. Dessa forma, à medida que são lidos os dados de linha, são inseridos nós da lista de dados de linha e das listas de ramos. Para mostrar melhor essa relação, a figura 3.3 mostra como fica a estrutura de dados após a criação do *array* de barras e um elemento dos dados de linhas adicionado, considerando o sistema apresentado na figura 3.1.

Como sugere a figura 3.3, um nó da lista encadeada de dados de linha foi criado acrescentando os dados da linha “1”. A linha “1”, por sua vez, interliga as barras “1” e “2” (ver figura 3.1). São criadas então outras duas listas encadeadas, uma para a barra “1” e uma para a barra “2”. O nó criado que está sendo apontado pela barra “1” indica que uma das barras vizinhas é a barra “2” e que elas são ligadas por meio da linha “1”. Analogamente, o nó que está sendo apontado pela barra “2” indica que uma das barras vizinhas é a barra “1” e que elas são ligadas por meio da linha “1”.

Quando os dados da linha “2” são adicionados, um novo nó da lista de dados de linha é criado, assim como as listas que fazem o mapeamento do sistema. A figura 3.4 mostra

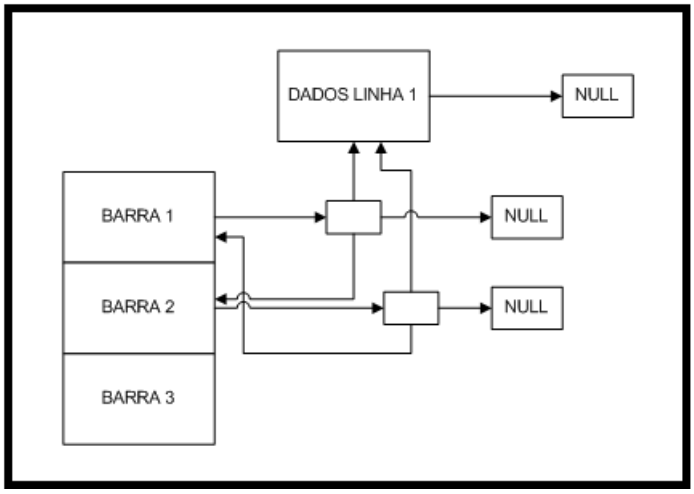

Figura 3.3: Estrutura de dados após os dados de uma linha serem adicionados.

como fica a estrutura de dados quando os dados da linha "2" são adicionados à mesma.

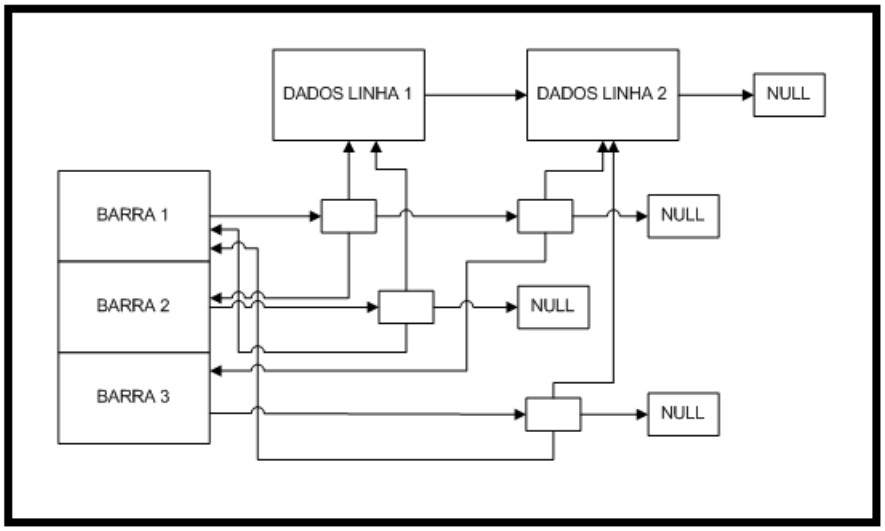

Figura 3.4: Estrutura de dados após os dados de duas linhas serem adicionados.

Verifica-se de acordo com a figura 3.4 que além do novo nó da lista de dados de linhas, outros dois nós são criados. Um deles é acrescentado à lista encadeada da barra "1" e indica que a mesma está ligada à barra "3" por meio da linha "2". Reciprocamente, um nó é criado na barra "3" e indica que a mesma está ligada à barra "1" através da linha "2".

Por fim, a última linha é adicionada à estrutura de dados e o quadro geral da estrutura é apresentado na figura 3.5.

Com a inserção do nó com os dados da linha "3", novamente outros dois nós são criados. Um deles é acrescentado à lista encadeada da barra "2", indicando que a mesma está conectada à barra "3" por meio da linha "3". Da mesma forma, outro nó é acrescentado à lista encadeada da barra "3" e mostra que a mesma está conectada à barra "2" por meio

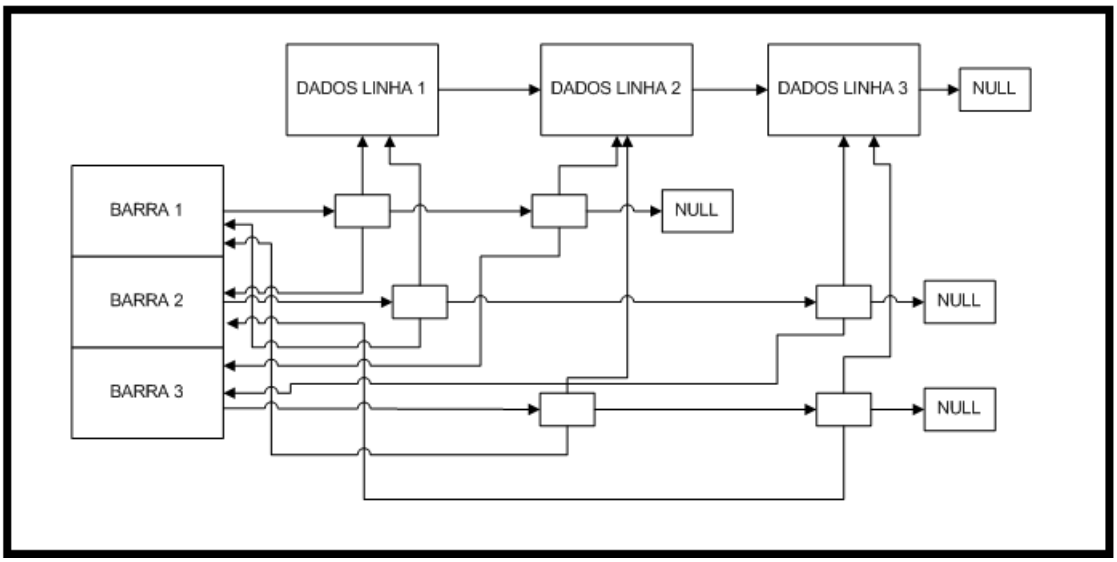

Figura 3.5: Estrutura de dados após os dados de três linhas serem adicionados.

da linha "3".

Além das vantagens em relação à memória, onde não ocorre redundância de dados, esta estrutura apresenta algumas outras vantagens.

Uma delas é a velocidade de acesso aos dados. Se o usuário deseja fazer uma busca para verificar quais linhas e quais barras estão ligadas à uma determinada barra, basta fazer uma busca no vetor de barras e então percorrer a lista encadeada associada a esta barra para ter essas informações. Caso essa estrutura não fosse criada mas fosse desejado obter essas informações, sucessivas buscas na lista encadeada de dados de linha e no vetor de barras seriam necessárias, ocasionando queda de desempenho do software.

Caso fosse desejado que esta otimização da busca fosse obtida sem esta estrutura, seria necessário criar outra estrutura com dados redundantes. A figura 3.6 mostra como seria uma estrutura clássica de programação para o exemplo da figura 3.1 com os dados da linha "1" adicionados. Verifica-se que os dados necessitariam de uma duplicação, provocando um uso de memória muito maior do que na estrutura utilizada.

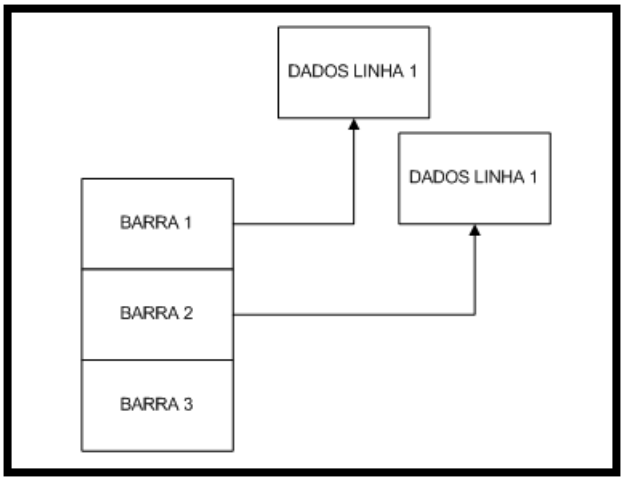

Figura 3.6: Estrutura de dados clássica.

Logo, a estrutura de dados EPSD utilizada neste trabalho para modelar os dados otimiza a alocação de memória, assim como as funções de busca e acesso aos dados.

4 O padrão de entrada de dados do ANAREDE

O software de análise de redes ANAREDE [1], desenvolvido pelo CEPEL [2] é um simulador de sistemas elétricos largamente utilizado no Brasil, mas seu arquivo de entrada de dados possui algumas grandes desvantagens.

A principal desvantagem dos arquivos de entrada de dados do ANAREDE está no posicionamento dos dados em colunas, sistema que é "herança" da linguagem de programação FORTRAN. Esse posicionamento permite com que um simples espaço entre os dados faça com que o ANAREDE cometa erros na leitura, além de dificultar a leitura dos dados do documento pelo usuário do software.

Outra desvantagem que também é "herança" do FORTRAN é o ponto decimal implícito. Em alguns dados, existe um ponto decimal na coluna que não é representado pelo usuário. Dessa forma, um dado como, por exemplo, "95.63" é escrito apenas na forma "9563", o que facilita ainda mais os erros de leitura pelo usuário.

Com o objetivo de apresentar o padrão de entrada de dados do ANAREDE, este capítulo traz uma amostra do que pode ser escrito utilizando este padrão. No entanto, assim como a ferramenta computacional desenvolvida nesse projeto, a amostra será limitada aos padrões de escrita de barras e linhas de sistemas elétricos.

4.1 Barras no padrão ANAREDE

Em um arquivo de entrada de dados do ANAREDE, a diretiva "DBAR" marca o início dos dados de barra e os números "99999" marcam o fim. Logo, cada linha que estiver entre essas marcações é definida com dados de uma barra. O que identifica qual dado está escrito é o posicionamento em relação às colunas. A tabela 4.1 mostra uma relação com os dados e entre quais colunas da linha os mesmos devem estar posicionados para serem corretamente lidos.

Colunas	**Dado**
1 à 5	Número de identificação da barra
8	Tipo de barra
11 à 22	Nome da barra
25 à 28	Tensao na barra
29 à 32	Ângulo na barra em radianos
33 à 37	Geração de energia ativa na barra
38 à 42	Geração de energia reativa na barra
59 à 63	Carga ativa na barra em MW
64 à 68	Carga reativa na barra em MVAr
69 à 73	Capacitância shunt em MVAr

Tabela 4.1: Padrão ANAREDE para dados de barra.

Para ilustrar como esses dados são apresentados no arquivo de entrada do ANAREDE, a figura 4.1 mostra um exemplo de como seria a entrada de dados para as barras do sistema da figura 3.1.

```
DBAR
(   0    |    1    |    2    |    3    |    4    |    5    |    6    |    7    |
(234567890123456789012345678901234567890123456789012345678901234567890123456789
(Num)OETGb(   nome   )Gl( V)( A)( Pg)( Qg)( Qn)( Qm)(Bc  )( Pl)( Ql)( Sh)Are(vf)
    1  1       Barra 1  1000  0.2500.    0.-99999999.    202   0.    0.
    2  1       Barra 2  1000  0.1500.    0.-99999999.    203   0.    0.
    3  1       Barra 3  1000  0.2950.    0.-99999999.    204   0.    0.
99999
```

Figura 4.1: Barras no padrão de entrada de dados do ANAREDE.

4.2 Linhas no padrão ANAREDE

Em se tratando de linhas no arquivo de entrada do ANAREDE, o que delimita o início e o fim da escrita dos dados são a diretiva “DLIN” e os números “99999”. Assim como no padrão de escrita de barras, o que determina a informação apresentada são os posicionamentos entre as colunas da linha no arquivo. A tabela 4.2 mostra uma relação com os dados e entre quais colunas da linha os mesmos devem estar posicionados para serem corretamente lidos.

Considerando novamente o sistema da figura 3.1, a figura 4.2 mostra um exemplo de como as linhas são inseridas em arquivo de entrada do ANAREDE.

Colunas	**Dado**
1 à 5	Barra de origem da linha
11 à 15	Barra de destino da linha
21 à 26	Resistência da linha
27 à 32	Reatância da linha
33 à 38	Susceptância shunt da linha
39 à 43	Tap
54 à 58	Defasagem
59 à 63	Carga ativa na barra em MW
64 à 68	Carga reativa na barra em MVAr
69 à 73	Capacitância shunt em MVAr

Tabela 4.2: Padrão ANAREDE para dados de linha.

```
DLIN
(    0    |    1    |    2    |    3    |    4    |    5    |    6    |    7    |
(234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890
(De )d O d(Pa )NcEP ( R% )( X% )(Mvar)(Tap)(Tmn)(Tmx)(Phs)(Bc  )(Cn)(Ce)Ns
   1           2             3.60       0.939             000
   2           3             3.60       0.939             000
   3           1             3.60       0.939             000
99999
```

Figura 4.2: Linhas no padrão de entrada de dados do ANAREDE.

5 A linguagem XML

A linguagem de marcação XML define uma sintaxe genérica usada para marcar dados com simples, legíveis *tags* (etiquetas). Ela provê um formato padrão para documentos de computadores. Este formato é flexível o bastante para ser personalizado em domínios tão diversos quanto páginas de internet, troca de dados eletrônicos, genealogia, serialização de objetos, sistemas de correio de voz, e mais [6].

O objetivo deste capítulo é oferecer uma introdução à linguagem XML, explicando de forma sucinta a leitura das *tags*, estruturas presentes em arquivos XML que permitem que os dados sejam escritos de forma estruturada. Também será mostrado como os dados do sistema elétrico serão disponibilizados para leitura, além de uma breve discussão sobre os *XML parsers*, que são analisadores de grande importância para a implementação do XML em ferramentas computacionais.

5.1 Leitura de documentos XML

Um documento XML contém texto, nunca dados binários. Ele pode ser aberto por qualquer programa que tenha a habilidade de ler um arquivo de texto [6]. O exemplo mostrado na figura 5.1 ilustra um documento escrito na linguagem XML na forma mais simples possível.

<pessoa> Joao da Silva </pessoa>

Figura 5.1: Exemplo simples de XML.

O documento mostrado no exemplo da figura 5.1 faz uma breve descrição de uma pessoa, através da *tag* pessoa. Dessa forma os itens <pessoa> e </pessoa> são chamados de *start-tag* e *end-tag*, marcam respectivamente o início e o fim da *tag* no documento e tudo o que estiver entre elas representam o conteúdo da *tag*.

É claro que este exemplo é muito simples; em uma aplicação mais real seria necessário um nivel de detalhamento maior para descrever uma pessoa. Por isso, uma *tag* pode conter subníveis, ou *tags* filhas (*child tags*). O exemplo mostrado na figura 5.2 estende o exemplo da figura 5.1 com um nível maior de detalhamento.

```
<pessoa>
        <nome> Joao </nome>
        <sobrenome> da Silva </sobrenome>
        <profissao> Engenheiro </profissao>
</pessoa>
```

Figura 5.2: Exemplo de XML com *child tags*.

O exemplo da figura 5.2 mostra com clareza que uma *tag* pode conter outras *tags*. Isso possibilita um nível muito grande de detalhamento, uma vez que as *child tags* podem também conter outras *tags*, que por sua vez podem conter outras. Para ilustrar essa possibilidade a figura 5.3 mostra um documento um pouco mais elaborado.

```
<pessoa>
        <nome>
                <primeiro_nome> Joao </primeiro_nome>
                <sobrenome> da Silva </sobrenome>
        </nome>
        <profissao>
                <formacao>Engenharia Mecanica</formacao>
                <cargo> Engenheiro Junior </cargo>
        </profissao>
</pessoa>
```

Figura 5.3: Exemplo de XML com *child tags* mais estruturado.

Por fim, uma *tag* pode conter atributos. Um atributo é uma informação do elemento que aparece sempre na *start-tag* e cada elemento pode conter vários atributos. Para ilustrar o uso de atributos, a figura 5.4 mostra o exemplo da figura 5.3 com o atributo "idade"adicionado à pessoa.

```
<pessoa   idade = "26" >
        <nome>
                <primeiro_nome> Joao </primeiro_nome>
                <sobrenome> da Silva </sobrenome>
        </nome>
        <profissao>
                <formacao>Engenharia Mecanica</formacao>
                <cargo> Engenheiro Junior </cargo>
        </profissao>
</pessoa>
```

Figura 5.4: Exemplo de XML utilizando atributo.

Além do conteúdo das *tags*, um documento XML pode conter comentários, que podem ser utilizados para explicar os dados contidos no documento ou fornecer outras informações

importantes sobre o mesmo. Um comentário sempre aparece entre "<!–" e " –>", como será ilustrado na seção 5.3.

5.2 O XML Parser

Parte do sucesso da linguagem XML se deve ao grande número de XML *parsers*. Os *parsers* são analisadores disponíveis para uso em várias linguagens de programação e tornam muito mais fácil a implementação do XML em ferramentas computacionais, pois são capazes de ler documentos XML, validar a sintaxe, ler o conteúdo das *tags*, escrever documentos XML sem erros de sintaxe, entre outras funções.

Em outras palavras, os *parsers* garantem a confiabilidade na troca de informações com documentos XML.

Para a elaboração do projeto, algumas opções de *parsers* foram analisadas a fim de se encontrar a melhor opção para a implementação na linguagem de programação C++. Foram testadas as opções:

- Xerces C++ [10];
- RapidXML [7];
- TinyXML++;

Após alguns testes com esses *parsers*, a opção escolhida foi o TinyXML++. A principal vantagem dele em relação aos outros foi a facilidade na implementação e configuração do mesmo, que também se mostrou bastante rápido e estável. O endereço eletrônico do *parser* TinyXML++ [9] contém muito exemplos de implementação e uma documentação completa que foi de grande valia para o sucesso do trabalho.

5.3 Dados do sistema elétrico em XML

Para alocar os dados do sistema elétrico em um documento XML, foram consideradas as variáveis de barra e linha, assim como os ramos, que descrevem, para cada barra, quais são suas barras vizinhas e quais linhas estão conectadas.

O padrão XML para o sistema elétrico está apresentado na figura 5.5.

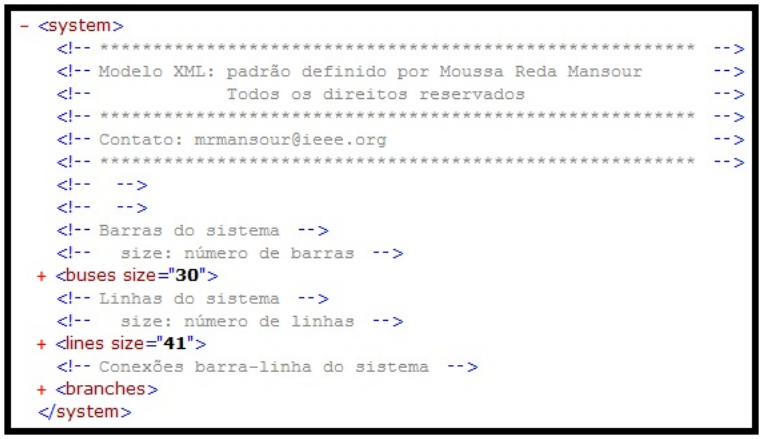

Figura 5.5: Exemplo de sistema elétrico escrito em XML.

Como sugere a figura 5.5, o sistema elétrico é descrito por uma *tag* principal chamada system. Essa *tag* por sua vez possui três *child tags*, buses, lines e topology, que descrevem as barras, linhas e a topologia do sistema, respectivamente. O sinal de "+" à esquerda das *child tags* indicam que as mesmas possuem outras *child tags*, que serão descritas nas seções seguintes.

5.3.1 A *tag* buses

Para representar o conjunto de barras do sistema elétrico, a *tag* buses possui várias *child tags* chamadas bus. Em cada *tag* bus, encontra-se o conjunto de dados de uma barra. O atributo size encontrado em buses indica o número de barras do sistema. A figura 5.6 mostra um exemplo da *tag* buses.

O sinal de "+" à esquerda das *tags* bus indicam que as mesmas possuem outras *child tags*, que descrevem as variáveis elétricas que são associadas às barras do sistema. A figura 5.7 mostra o detalhe de uma das *tags* bus.

Como mostrado na figura 5.7, várias variáveis são associadas para a descrição de uma barra por meio de diversas *tags* e um atributo "id", que representa um identificador da linha.

A tabela 5.1 mostra uma breve descrição do conteúdo dessas *tags*.

5.3.2 A *tag* lines

A *tag* lines é semelhante à *tag* buses, porém contém os dados das linhas que interligam as barras do sistema. A figura 5.8 mostra um exemplo da *tag* lines. Verifica-se que ela

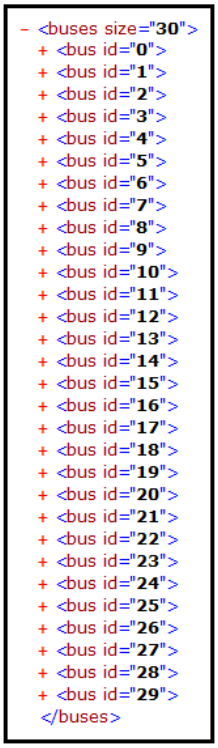

Figura 5.6: Exemplo da *tag* buses.

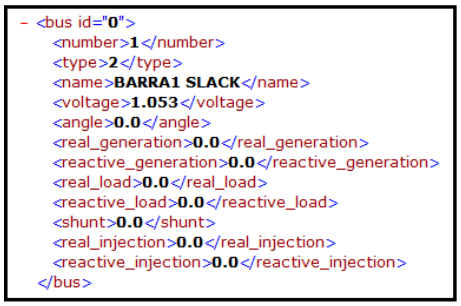

Figura 5.7: Exemplo da *tag* bus.

possui um atributo "size", que representa o número de linhas do sistema, e várias *child tags* chamadas line, cada uma com um conjunto de dados de uma linha.

Cada *tag* line possui um conjunto de dados que descrevem uma linha atráves de outras *child tags* e um atributo chamado "id", que constitui um identificador fictício da linha. Um exemplo de uma *tag* line está exposto na figura 5.9.

A tabela 5.2 mostra uma breve descrição das *child tags* contidas em line.

5.3.3 A *tag* topology

O elemento topology do documento descreve os ramos do sistema, ou seja, identifica para cada barra quantas linhas saem dela, quais são essas linhas e para onde elas vão

Child tag	**Descrição**
number	Número real da barra
type	Tipo de barra
name	Nome da barra
voltage	Tensao na barra
angle	Ângulo na barra em radianos
real_generation	Geração de energia ativa na barra
reactive_generation	Geração de energia reativa na barra
real_load	Carga ativa na barra em MW
reactive_load	Carga reativa na barra em MVAr
shunt	Capacitância shunt MVAr
real_injection	Injeção de energia ativa na barra
reactive_injection	Injeção de energia reativa na barra

Tabela 5.1: Descrição das *child tags* contidas em bus.

Child tag	**Descrição**
from	Barra de origem da linha
to	Barra de destino da linha
resistance	Resistência da linha em p.u.
reactance	Reatância da linha em p.u.
shunt	Susceptância da linha em MVAr
tap	Tap da linha

Tabela 5.2: Descrição das *child tags* contidas em line.

(barras vizinhas). De certa forma, esta informação é redundante, pois as informações contidas nessa *tag* são retiradas da *tag* line e são utilizadas apenas para facilitar o acesso aos dados em algumas aplicações. A figura 5.10 mostra o aspecto geral da *tag* topology.

Assim como nas *tags* buses e lines, topology possui várias *child tags*, que neste caso são chamadas branch. Cada uma delas expõe as informações de ramos para uma barra e possuem os atributos "from" e "size", que representam a identificação da barra e o número de linhas que saem dela, respectivamente. A figura 5.11 apresenta um exemplo de uma *tag* branch com suas *child tags*.

Através da figura 5.11, nota-se a presença das *child tags* data. Dentro delas, estão as informações "to" e "id_data", que representam a identificação da barra de destino e o índice do vetor de barras correspondente. No exemplo da figura 5.11, portanto, a barra identificada por "0" possui duas linhas ligadas à ela e ambas vão para a barra identificada por "1".

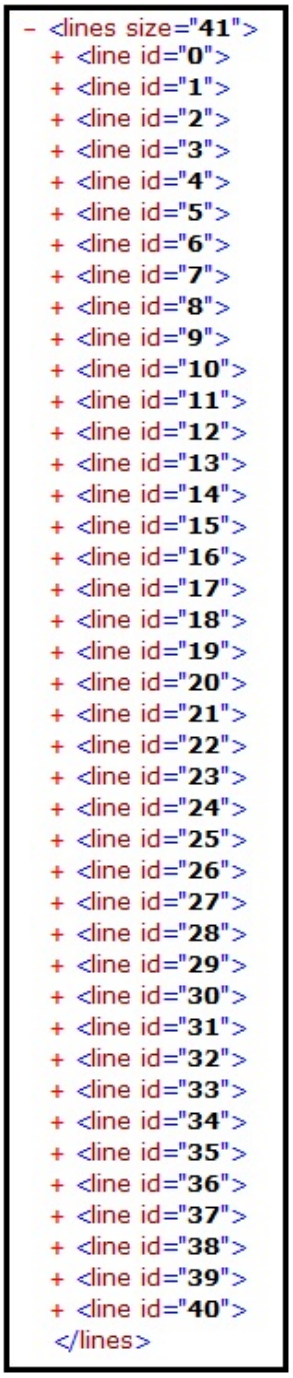

Figura 5.8: Exemplo de uma *tag* lines.

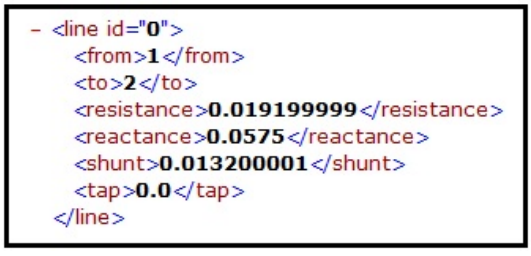

Figura 5.9: Exemplo de uma *tag* line.

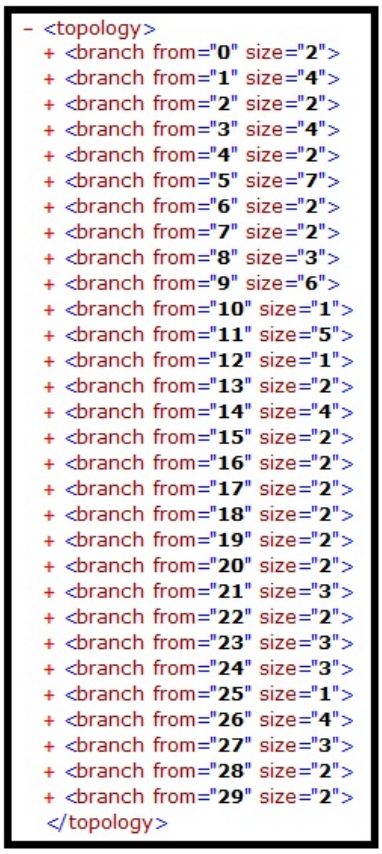

Figura 5.10: Exemplo de uma *tag* topology.

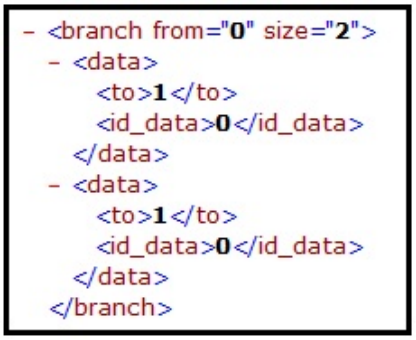

Figura 5.11: Exemplo de uma *tag* branch.

6 Documentação das classes

Enquanto o uso de classes pode facilitar o entendimento e a modificação do código em um projeto envolvendo programação, não documentá-las corretamente pode causar problemas para os futuros usuários das mesmas.

Logo, esse capítulo visa descrever as variáveis e as funções-membro que estão alocadas em cada classe.

Como as variáveis da classe só podem ser acessadas pelas funções-membro, foram definidas como padrão de programação para todas as classes as funções "*set*" e "*get*", que permitem modificar ou obter, respectivamente, uma determinada variável da classe.

6.1 A classe barra

A classe barra é a mais simples de todas as classes desenvolvidas, pois não possui funções-membro que executam tarefas, mas apenas funções que modificam ou retornam o valor das variáveis da classe.

As variáveis presentes na classe barra estão apresentadas na tabela 6.1.

Variável	**Tipo**	**Descrição**
id	inteiro	Identificador da barra.
tipoBarra	inteiro	Tipo de barra
nome	string	Nome da barra
tensao	*double*	Tensao na barra em p.u.
teta	*double*	Ângulo na barra em radianos
gerAtiva	*double*	Geração de energia ativa na barra
gerReativa	*double*	Geração de energia reativa na barra
carAtiva	*double*	Carga ativa na barra em MW
carReativa	*double*	Carga reativa na barra em MVAr
shunt	*double*	Capacitância shunt MVAr
injecaoAtiva	*double*	Injeção de energia ativa na barra
injecaoReativa	*double*	Injeção de energia reativa na barra
geracao	*double*	Valor da geração com tensão controlada
*inicio_ptr	ramos[a]	Ponteiro que aponta para o primeiro elemento da lista de ramos
qtdeLinhas	inteiro	Quantidade de linhas ligadas à barra
ref	inteiro	Indica se a barra é a referência

[a] O tipo ramos se refere à classe ramos, que será descrita em outra seção.

Tabela 6.1: Descrição das variáveis da classe barra.

Essas variáveis, contudo, só são acessíveis pelas funções da classe, que são apresentadas apresentadas nas seções seguintes.

6.1.1 Funções membro da classe barra

Por se tratar de uma classe que não executa tarefas, as funções da classe barra são essencialmente funções que atribuem ou retornam valor das variáveis.

Como padrão definido, essa classe apresenta suas funções *"set"* e *"get"*. As tabelas 6.2 e 6.3 listam essas funções e suas respectivas descrições.

Protótipo	**Descrição**
void [b] set_nome(string)	Atribui um nome à barra
void set_id(int)	Atribui um valor de identificação à barra
void set_tensao(double)	Atribui um valor para a tensão na barra
void set_tipobarra(int)	Atribui um tipo à barra
void set_teta(double)	Atribui um valor ao ângulo da barra
void set_gerAtiva(double)	Atribui um valor à geração ativa na barra
void set_gerReativa(double)	Atribui um valor à geração reativa na barra
void set_geracao(double)	Atribui um valor à geração com tensão controlada
void set_carAtiva(double)	Atribui um valor à carga ativa na barra
void set_carReativa(double)	Atribui um valor à carga reativa na barra
void set_shunt(double)	Atribui um valor à capacitância shunt na barra
void set_injecaoAtiva(double)	Atribui um valor à injeção de energia ativa na barra
void set_injecaoReativa(double)	Atribui um valor à injeção de energia reativa na barra
void set_ramo(ramos*)	Atribui um endereço ao ponteiro inicio_ptr

[b] Uma função do tipo void não retorna nenhuma variável.

Tabela 6.2: Descrição das funções *set* contidas na classe barra.

Protótipo	**Descrição**
string get_nome()	Função que retorna o nome da barra
int get_id()	Função que retona o id de cada barra
double get_tensao()	Função que retorna a tensão da barra
int get_tipobarra()	Função que retorna o tipo da barra
double get_teta()	Função que retorna o ângulo da barra
double get_gerAtiva()	Função que retorna geração ativa da barra
double get_gerReativa()	Função que retorna a geração reativa da barra
double get_geracao()	Função que retorna a geração com tensão controlada
double get_carAtiva()	Função que retorna a carga ativa da barra
double get_carReativa()	Função que retorna a carga reativa da barra
double get_shunt()	Função que retorna a carga shunt da barra
double get_injecaoAtiva()	Função que retorna a injeção de energia ativa da barra
double get_injecaoReativa()	Função que retorna a injeção de energia reativa da barra
int get_qtdeLinhas()	Função que retorna a quantidade de linhas que estão ligadas à barra
ramos *get_ramo()	Função que retorna o endereço apontado por inicio_ptr

Tabela 6.3: Descrição das funções *get* contidas na classe barra.

A funções *set* são muito úteis quando há necessidade de atribuir valor à alguma variável, mas quando se deseja atribuir valor à todas as variáveis de uma vez, chamar todas as funções *set* de uma vez deixaria o código muito poluído. Por isso, foi desenvolvida

a função *set_barra*, que pode atribuir valores à todas as variáveis (exceto à *qtdeLinhas* e à *inicio_ptr*) de uma vez. A função *set_barra* tem o seguinte protótipo:

void set_barra(string n_nome, int n_id, double n_tensao, double n_teta, double n_gerAtiva, double n_gerReativa, double n_carAtiva, double n_carReativa, double n_injecaoAtiva, double n_injecaoReativa, double n_shunt, double n_geracao, int n_tipoBarra, int n_ref);

Além de *set_barra*, foi desenvolvida uma função para incrementar a variável *qtdeLinhas*. Isso porque, esta variável é incrementada enquanto se cria a lista encadeada de ramos. Para cada nó adicionado à lista de ramos da barra, a variável *qtdeLinhas* é incrementada, sinalizando que mais uma linha está conectada à barra. A função que faz esse incremento tem o seguinte protótipo:

void incrementa_linha();

Por fim, há uma função para imprimir na tela os valores armazenados nas variáveis. Esta função é útil quando o programador quer conferir de forma rápida se os dados estão sendo passados para as variáveis de forma correta. O protótipo dessa função é definido por:

void mostra_barra();

6.2 A classe dados_lin

A classe dados_lin possui todas as variáveis necessárias para armazenar os dados de linha e criar uma lista encadeada com esses dados. Essa classe possui além das funções *set* e *get*, funções que podem criar ou destruir nós da lista encadeada de dados.

As variáveis presentes na classe estão apresentadas na tabela 6.4.

Da mesma forma da classe barra, funções-membro foram definidas para manipular essas variáveis.

6.2.1 Funções membro da classe dados_lin

Da mesma forma que a classe barra, a classe dados_lin tem suas funções *"set"* e *"get"*, as quais estão listadas nas tabelas 6.5 e 6.6.

Variável	**Tipo**	**Descrição**
id	inteiro	Identificador fictício da linha
resistencia	double	Resistência da linha
reatancia	double	Reatância da linha
tap	*double*	Tap da linha
defasagem	*double*	Defasagem da linha em radianos
shunt	*double*	Susceptância shunt da linha em MVAr
idBarraOrigem	inteiro	Identificação da barra de origem da linha
idBarraDestino	inteiro	Identificação da barra de destino da linha
*prox	dados	Variável que armazena o endereço do próximo nó da lista encadeada

Tabela 6.4: Descrição das variáveis da classe dados_lin.

Protótipo	**Descrição**
void set_idlinha(int)	Atribui um valor de identificação à linha
void set_resistencia(double)	Atribui um valor para a resistência da linha
void set_reatancia(double)	Atribui um valor para a reatância da linha
void set_tap(double)	Atribui um valor para o tap da linha
void set_defasagem(double)	Atribui um valor à defasagem da linha
void set_shunt(double)	Atribui um valor à susceptância shunt da linha
void set_origem(int)	Atribui um valor à identificação da barra de origem
void set_destino(int)	Atribui um valor à identificação da barra de destino
void set_prox(dados*)	Atribui um endereço ao ponteiro prox

Tabela 6.5: Descrição das funções *set* contidas na classe dados_lin.

Protótipo	**Descrição**
int get_idlinha()	Função que retorna a identificação da linha
double get_resistencia()	Função que retorna a resistência da linha
double get_reatancia()	Função que retorna a reatância da linha
double get_tap()	Função que retorna o tap da linha
double get_defasagem()	Função que retorna o valor da defasagem da linha
double get_shunt()	Função que retorna a susceptância shunt da linha
int get_origem()	Função que retorna a identificação da barra de origem
int get_destino()	Função que retorna a identificação da barra de destino
dados* get_prox()	Função que retorna o endereço do próximo elemento da lista encadeada

Tabela 6.6: Descrição das funções *get* contidas na classe dados_lin.

Analogamente à classe barra, a classe dados_lin também possui uma função que permite com que todos os dados sejam incluídos de uma só vez (com exceção do ponteiro *prox*). Essa função chama-se *set_dados* e possui o seguinte protótipo:

void set_dados(Id in_id, double in_resistencia, double in_reatancia,

double in_tap, double in_defasagem, double in_shunt,
Id in_idorigem, Id in_iddestino);

No entanto, cada vez que um novo nó precisar ser inserido na lista, memória deve ser alocada para este nó e ele deve ser corretamente posicionado na lista encadeada. Para desempenhar essa função, foi desenvolvida uma função *insere*, que posiciona um novo elemento no fim da lista encadeada de dados toda vez que ela for chamada. Essa função recebe como parâmetros, além dos dados de linha, um ponteiro para o primeiro elemento da lista encadeada, como sugere seu protótipo:

void insere(dados *raiz, Id in_id, double in_resistencia, double in_reatancia,
double in_tap, double in_defasagem, double in_shunt, Id in_idorigem,
Id in_iddestino);

Como uma lista encadeada possui vários elementos, é interessante que se tenha um função que busque um elemento específico nesta lista. Por isso, foi criada uma função de busca chamada *busca*, que recebe como parâmetros o ponteiro para o primeiro elemento da lista e a identificação da linha que se deseja buscar, retornando o endereço de memória do elemento com tal identificação. O protótipo da mesma é como segue:

dados *busca(dados *raiz, int id_target);

Outra função que pode ser interessante para o programador é a função que imprime na tela os dados dos nós da lista. Para realizar essa tarefa, a função *mostra_lista* recebe como parâmetro o ponteiro para o primeiro elemento da lista e apresenta o seguinte protótipo:

void mostra_lista(dados *raiz);

Por fim, assim como alocar os dados dinamicamente faz parte da boa prática de programação, liberar a memória quando os dados não precisam mais ser utilizados também faz. Foi criada então a função *clear_list*, que recebe como parâmetro um ponteiro para o primeiro nó e percorre a lista deletando cada um dos nós. A função apresenta o seguinte protótipo:

void clear_list(dados *raiz)

6.3 A classe ramos

Como descrito no capítulo 3, as variáveis da classe ramos são essencialmente ponteiros, os quais estão na tabela 6.7.

Variável	**Tipo**	**Descrição**
*dados_linha	dados	Ponteiro para os dados de linha
*prox_barra	barra	Ponteiro para a próxima barra da linha
*prox	ramos	Ponteiro para o próximo elemento da lista encadeada

Tabela 6.7: Descrição das variáveis da classe ramos.

6.3.1 Funções membro da classe ramos

A classe ramos é um pouco diferente das classes já descritas, pois as funções *set* não fazem muito sentido, uma vez que os ponteiros recebem os endereços enquanto os nós são criados e esses endereços não mudam, por mais que mudem os conteúdos dos mesmos. As funções *get*, no entanto, são interessantes quando se deseja saber as informações sobre os ramos de uma barra e estão apresentadas na tabela 6.8.

Protótipo	**Descrição**
barra* get_prox_barra()	Função que retorna um ponteiro para a próxima barra da linha
dados* get_dados_linha()	Função que retorna um ponteiro para o nó da lista de dados
ramos* get_prox()	Função que retorna um ponteiro para o próximo elemento da lista de ramo

Tabela 6.8: Descrição das funções *get* contidas na classe ramos.

Para criar os nós da lista, foi desenvolvida a função *cria_no*. Essa função recebe como parâmetros um ponteiro com o endereço do nó da lista de dados correspondente à linha que faz a ligação entre as barras, um ponteiro com o endereço do primeiro elemento do vetor de barras e o número de barras do sistema. Assim, a função aquisita quais são as barras de origem e destino da linha, e cria os nós tanto na barra de origem como na barra de destino, como descrito no capítulo 3. A função *cria_no* possui o seguinte protótipo:

void cria_no(dados* linha, barra* vetor, int qtdeBarras);

Assim como a classe dados, a classe ramos também possui uma função para imprimir a lista na tela, porém a mesma só disponibiliza para leitura as identificações das barras e linhas, o que é suficiente para saber se os ramos estão sendo alocados de maneira correta. Recebe como parâmetro um ponteiro com o endereço para o primeiro nó da lista e o protótipo da mesma é definido por:

void mostra_lista(ramos* raiz);

Para liberar a memória alocada pela lista, a função *clear_list* da classe ramos recebe

como parâmetro um ponteiro com o endereço do primeiro elemento da lista e tem protótipo definido por:

void clear_list(ramos *raiz);

6.4 A classe sistema

A classe sistema é diferente das outras classes já apresentadas. Ela possui a implementação da montagem da estrutura de dados e do conversor de dados, ou seja, ela utiliza as outras classes já descritas para montar funções que permitam o funcionamento da ferramenta computacional.

Suas variáveis são basicamente ponteiros dos tipos, barra, dados e ramos, que são declarados para que as classes possam ser corretamente usadas na implementação.

Portanto, a classe sistema não possui funções *set* e *get*, apenas funções que realizam tarefas utilizando as classes já descritas.

6.4.1 Funções membro da classe sistema

As funções dessa classe podem ser divididas em dois tipos: escrita e leitura. Basicamente, as funções de leitura leem os dados de um arquivo e armazenam na estrutura de dados, enquanto as de escrita escrevem os dados da estrutura em um arquivo.

Portanto, é impossível usar uma função de escrita sem que se tenha uma estrutura de dados montada. A tentativa de uso dessas funções sem que a estrutura esteja corretamente montada provavelmente resultará em erro de execução ou produzirá uma saída que não é desejada.

A funções de leitura, podem realizar dois tipos de leitura: ler de arquivos de texto (com extensão *.txt*) ou ler de documentos XML (com extensão *.xml*).

Para ler os dados de arquivos de texto, duas funções devem ser utilizadas. A primeira delas é a *vetorbarrastxt*, que abre o arquivo padrão ANAREDE e cria o vetor de objetos da classe barra com todos os dados de barra presentes no arquivo. A função recebe como parâmetro uma variável com o nome do arquivo e tem o seguinte protótipo:

void vetorbarrastxt(char *filename);

A segunda delas é a *listadadostxt*, que cria a lista de dados de linha e também as listas

de ramos a partir do arquivo padrão ANAREDE. É importante salientar que, quando lendo arquivos do tipo texto, esta função jamais pode ser chamada antes da função *vetorbarrastxt*, pois para a criação da lista de ramos o vetor de barras deve estar necessariamente criado. Essa função também recebe como parâmetro uma variável com o nome do arquivo e tem o protótipo definido por:

void listadadostxt(char *filename);

Para ler documentos XML, apenas uma função precisa ser chamada. Essa função não recebe nenhum parâmetro, pois ela mesma requisita o nome do arquivo durante a execução. Ela lê o arquivo XML com ajuda do *parser* e monta a estrutura de dados. Seu protótipo é definido por :

void read_xml();

As funções de escrita podem escrever tanto em documentos XML como em arquivos de texto uma vez que a estrutura de dados já esteja montada. As funções de escrita, *write_xml* e *write_txt*, não recebem nenhum parâmetro e possuem respectivamente os seguinte protótipos:

void write_xml();

void write_xml();

Por fim, a função *clear_system* libera a memória alocada pela estrutura. Ela primeiramente faz uma varredura no vetor de barras e elimina as listas de ramos apontadas pelos elementos do vetor, depois limpa a lista de dados de linha e, finalmente, libera a memória alocada para o vetor de barras. O seguinte protótipo define a função:

void clear_system();

Para proporcionar uma visão geral da aplicação desenvolvida e da maneira como as funções das classes interagem entre si, a figura 6.1 apresenta o fluxograma básico da aplicação. É importante notar que praticamente todas as ações apresentadas no fluxograma podem ser feitas através das funções já descritas neste capítulo.

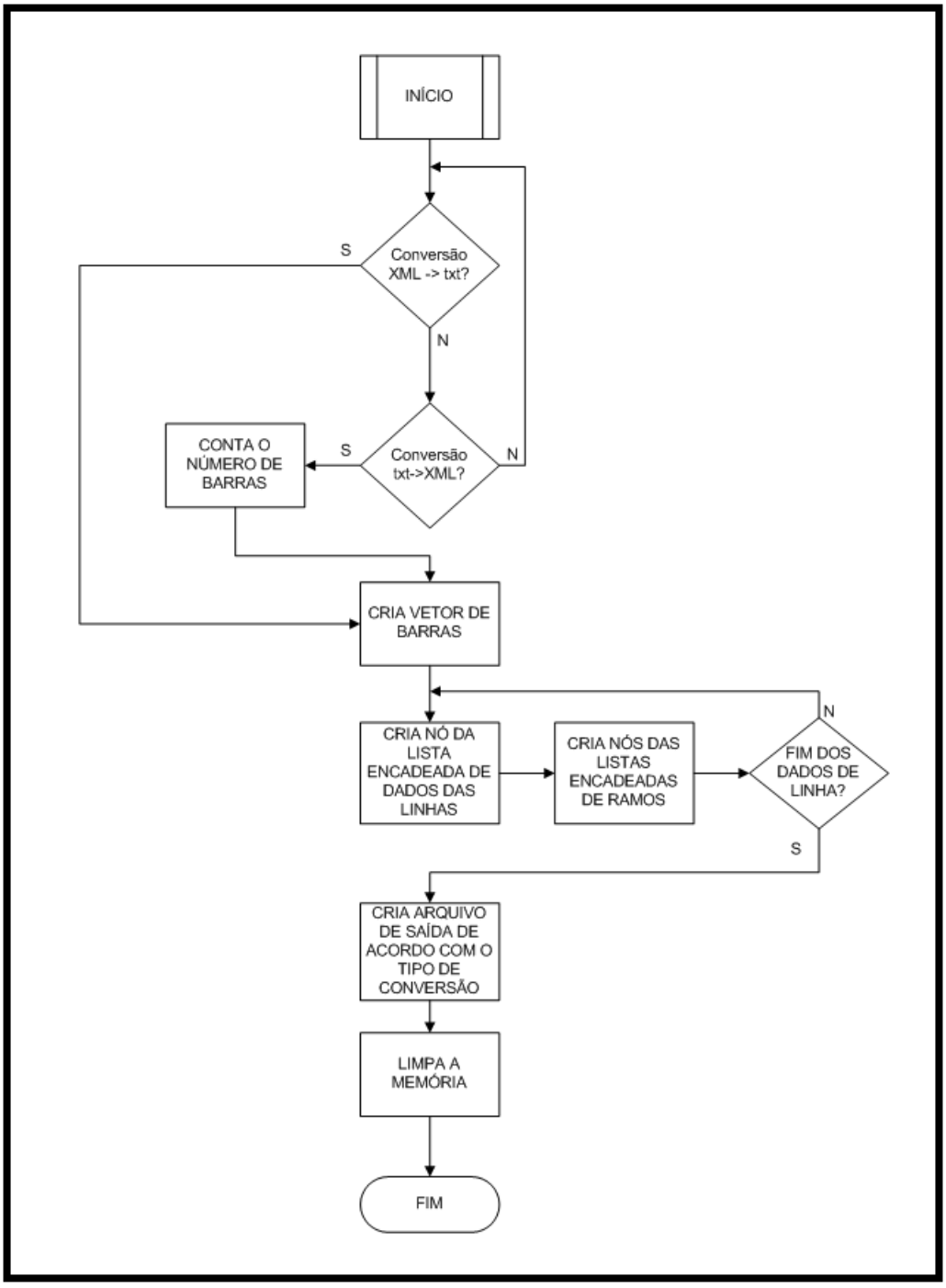

Figura 6.1: Fluxograma básico da aplicação.

7 Conclusão

O conversor de dados foi implementado como planejado, testado e verificado que é capaz de gerar os documentos nos padrões esperados, nas duas vias de conversão. A ferramenta foi capaz de formar uma estrutura de dados de maneira a permitir um acesso confiável aos dados, viabilizando o projeto.

Com o conversor funcionando, os usuários do programa ANAREDE podem ter uma opção a mais na montagem de sistemas para simulação ou na modificação de dados que já se apresentem no padrão ANAREDE. O padrão XML facilita a leitura e escrita dos dados, diminuido a possibilidade de erros e tornando mais rápido e confiável o processo de montagem de sistemas para simulação.

Contudo, talvez mais importantes que a própria aplicação implementada, as classes foram disponibilizadas de forma a permitir uma mais fácil expansão do código até aqui desenvolvido. O uso de uma linguagem de programação orientada a objetos fez toda a diferença no sentido de permitir uma futura ampliação das capacidades do conversor. Ampliar não somente o suporte à outros padrões, mas também o suporte à outros dados do ANAREDE, como geradores, cargas, transformadores e outros.

Uma boa documentação das classes foi disponibilizada aos futuros usuários das mesmas, facilitando o uso e modificação e, consequentemente, essa possível ampliação.

APÊNDICE A – Apresentação de resultados

Para demonstrar o funcionamento da ferramenta computacional desenvolvida, será apresentada a conversão de um arquivo de texto no formato padrão de entrada do software ANAREDE que descreve o sistema de três barras apresentado na figura 3.1 para o padrão XML apresentado. A figura A.1 mostra o arquivo de texto, enquanto as figuras A.2 e A.3 mostram a conversão efetuada.

```
DBAR
(   0    |    1    |    2    |    3    |    4    |    5    |    6    |    7    |
(234567890123456789012345678901234567890123456789012345678901234567890123456789
(Num)OETGb(   nome   )Gl( V)( A)( Pg)( Qg)( Qn)( Qm)(Bc  )( Pl)( Ql)( Sh)Are(vf)
    1  1       Barra 1  1000  0.317.2383.2-999.9999.                        11000
    2  1       Barra 2  1000  0.   0.   0.                   450   45       11000
    3  1       Barra 3  1000  0.3600.   0.-99999999.         0.   0.       21000
99999



DLIN
(   0    |    1    |    2    |    3    |    4    |    5    |    6    |    7    |
(234567890123456789012345678901234567890123456789012345678901234567890123456789
(De )d O d(Pa )NcEP ( R% )( X% )(Mvar)(Tap)(Tmn)(Tmx)(Phs)(Bc  )(Cn)(Ce)Ns
   1          2 1           3.60      0.939            000
   2          3 1           3.60      0.939            000
   3          1 1           3.60      0.939            000
99999
```

Figura A.1: Sistema no padrão de entrada de dados do ANAREDE.

```
<?xml version="1.0" encoding="UTF-8" standalone="no
<system>
  <!--Descricao do sistema eletrico-->
  <buses size="3">
    <bus id="1">
      <number>1</number>
      <type>1</type>
      <name>Barra 1</name>
      <voltage>1</voltage>
      <angle>0</angle>
      <real_generation>317.2</real_generation>
      <reactive_generation>383.2</reactive_generati
      <real_load>0</real_load>
      <reactive_load>0</reactive_load>
      <shunt>0</shunt>
      <real_injection>0</real_injection>
      <reactive_injection>0</reactive_injection>
    </bus>
    <bus id="2">
      <number>2</number>
      <type>1</type>
      <name>Barra 2</name>
      <voltage>1</voltage>
      <angle>0</angle>
      <real_generation>0</real_generation>
      <reactive_generation>0</reactive_generation>
      <real_load>450</real_load>
      <reactive_load>45</reactive_load>
      <shunt>0</shunt>
      <real_injection>0</real_injection>
      <reactive_injection>0</reactive_injection>
    </bus>
    <bus id="3">
      <number>3</number>
      <type>1</type>
      <name>Barra 3</name>
      <voltage>1</voltage>
      <angle>0</angle>
      <real_generation>3600</real_generation>
      <reactive_generation>0</reactive_generation>
      <real_load>0</real_load>
      <reactive_load>0</reactive_load>
      <shunt>0</shunt>
      <real_injection>0</real_injection>
      <reactive_injection>0</reactive_injection>
    </bus>
  </buses>
```

Figura A.2: Conversão para XML - Parte 1.

```
<lines size="3">
  <line id="0">
    <from>1</from>
    <to>2</to>
    <resistance>0</resistance
    <reactance>3.6</reactanc
    <shunt>0</shunt>
    <tap>0.939</tap>
  </line>
  <line id="1">
    <from>2</from>
    <to>3</to>
    <resistance>0</resistance
    <reactance>3.6</reactanc
    <shunt>0</shunt>
    <tap>0.939</tap>
  </line>
  <line id="2">
    <from>3</from>
    <to>1</to>
    <resistance>0</resistance
    <reactance>3.6</reactanc
    <shunt>0</shunt>
    <tap>0.939</tap>
  </line>
</lines>
<branches>
  <branch from="1" size="2">
    <data>
      <to>3</to>
      <id_data>3</id_data>
    </data>
    <data>
      <to>2</to>
      <id_data>2</id_data>
    </data>
  </branch>
  <branch from="2" size="2">
    <data>
      <to>3</to>
      <id_data>3</id_data>
    </data>
    <data>
      <to>1</to>
      <id_data>1</id_data>
    </data>
  </branch>
  <branch from="3" size="2">
    <data>
      <to>1</to>
      <id_data>1</id_data>
    </data>
    <data>
      <to>2</to>
      <id_data>2</id_data>
    </data>
  </branch>
</branches>
</system>
```

Figura A.3: Conversão para XML - Parte 2.

Referências

1 ANAREDE (Página principal). Disponível em: <http://www.anarede.cepel.br/>. Acesso em: 24/04/2010.

2 CEPEL (Página principal). Disponível em: <http://www.cepel.br/>. Acesso em: 24/04/2010.

3 CESTA, A. A. - *C++ como uma linguagem de programação orientada a objetos*. Campinas: UNICAMP. (1996).

4 DEITEL, H. M.; DEITEL, P. J. - *C++ How to program*. Prentice Hall. (2005).

5 DROZDEK, A. - Estrutura de dados e algoritmos em C++. Thomson Pioneira. (2002).

6 HAROLD, E. R.; MEANS, W. S. - *XML in a Nutshell*. O'Reilly Media. (2002).

7 RapidXML (Página principal). Disponível em: <http://rapidxml.sourceforge.net/>. Acesso em: 24/04/2010.

8 RICARTE,I. L. M. *Programação orientada a Objetos com C++*. Campinas: UNICAMP. (1995).

9 TINYXML (Página principal). Disponível em: <http://www.grinninglizard.com/tinyxml/>. Acesso em: 24/04/2010.

10 Xerces C++ (Página principal). Disponível em: <http://xerces.apache.org/xerces-c/>. Acesso em: 24/04/2010.