DAS Departamento de Automação e Sistemas
CTC Centro Tecnológico
UFSC Universidade Federal de Santa Catarina

Gerador de Código para Definição de Classes de Equipamentos

Relatório submetido à Universidade Federal de Santa Catarina

como requisito para a aprovação da disciplina:

DAS 5511: Projeto de Fim de Curso

Gustavo Kerezi

Florianópolis, Julho de 2017

Gerador de Código para Definição de Classes de Equipamentos

Gustavo Kerezi

Esta monografia foi julgada no contexto da disciplina

DAS 5511: Projeto de Fim de Curso

e aprovada na sua forma final pelo

Curso de Engenharia de Controle e Automação

Prof. Leandro Buss Becker

_____

Banca Examinadora:

Prof. Leandro Buss Becker
Orientador no Curso

Prof. Hector Bessa Silveira
Responsável pela disciplina

Prof. Cristian Koliver, Avaliador

Matheus Krüger Winter, Debatedor

Leonardo Rubin Quaini, Debatedor

Agradecimentos

A meus pais, que deram sangue e suor a cada conquista, a meus familiares que incentivaram a busca pelos meus objetivos, aos amigos da turma 2011.2 da Engenharia de Controle e Automação, aliados diários em uma trajetória nada curta, aos colegas de trabalho com quem adquiri grandes aprendizados, aos amigos de longe que permaneceram na vida que eu abdiquei, e especialmente, a todos os professores, do primário, fundamental, ensino médio e superior, herois silenciosos e partes igualmente importantes da conclusão deste ciclo.

Resumo

A grande complexidade dos processos da indústria petroquímica tem levado, ao longo dos anos, ao desenvolvimento de sistemas computacionais de controle e supervisão cada vez mais avançados e modulares. Aplicações flexíveis são capazes de se adaptar a uma ampla gama de problemas, replicando estratégias de controle em diferentes plantas e campos, mas podem requerer diferentes etapas de ajustes e definições para funcionar corretamente. Este trabalho, inserido dentro de um projeto de pesquisa do Departamento de Automação e Sistemas da UFSC em parceria com a Petrobras, tem por objetivo simplificar a etapa de pré-configuração de um software de operação e controle utilizado pela empresa, realizada através da criação de arquivos específicos. Aplicando conceitos de Model Driven Development é proposta uma aplicação para geração automática de arquivos de pré-configuração compatíveis, que modifica o processo de descrição dos tipos de equipamentos de campo presentes nas plantas controladas pelo software da Petrobras. A definição manual dos arquivos de pré-configuração, com edições diretas ao código fonte, é substituída por uma definição gráfica através da interface de uma nova aplicação, capaz de importar arquivos já existentes utilizando transformações Text to Model geradas a partir da ferramenta ANTLR, e capaz de gerar novos arquivos utilizando transformações Model to Text geradas com a ferramenta JET. O novo fluxo de criação de arquivos proposto, através da nova aplicação criada, visa possibilitar a modificação do conteúdo dos arquivos sem preocupações a respeito das estruturas de dados, liberando o projetista para focar nas sintonias e estruturas de controle.

Palavras-chave: Geração de código, Integração de sistemas,.

Abstract

The high complexity of processes within the petrochemical industry has led, through recent years, to the development of more advanced and modular control and monitoring systems. Applications created in a more flexible way are capable to adapt to a broad spectrum of problems, replicating control strategies in different fields and plants, but may require multiple configuration steps in order to function as intended. This work, inserted within a research project from the Department of Automation and Systems at UFSC, in partnership with Petrobras, aims to simplify the pre-configuration stage of a control and operation software used by the company, that happens via the definition of specific files. Applying concepts of Model Driven Development, an application is proposed to generate automatically compatible pre-configuration files, modifying the process of describing the kinds of field equipment in the plants controlled by Petrobras' software. A manual definition of pre-configuration files, with straight source code manipulation, is replaced by a graphical definition in a new application interface, capable of importing already existing files using Text to Model transformations generated with the ANTLR tool, as well as capable of generating new files using Model to Text transformations generated with the JET tool. The workflow for pre-configuration files definition proposed, using the new application, intends to enable changes in the files contents without worrying about data structure, gaining time to focus in the control structures and tuning.

Keywords: Code generation, System integration.

Lista de ilustrações

Lista de abreviaturas e siglas

MPA - Módulo de Procedimentos Automatizados

EMF - Eclipse Modeling Framework

JET - Java Emitter Templates

ANTLR - ANother Tool for Language Recognition

MDD - Model Driven Design

OPC - Object linking and embedding (OLE) for Process Control

DLL - Dynamic-Link Library

Cenpes - Centro de Pesquisas e Desenvolvimento Leopoldo Américo Miguez de Mello

MPC - Model Predictive Control

UML - Unified Modeling Language

DSLs - Domain-Specific Languages

M2T - Model to Text

T2M - Text to Model

GUI - Graphical User Interface

AST - Abstract Syntax Tree

IDE - Integrated Development Environment

Sumário

1 Introdução

O desenvolvimento de software para auxiliar o controle de processos na indústria é um dos principais pilares da automação nos dias de hoje. As estratégias de operação e monitoramento de plantas, antes relegadas a controladores locais implementados em hardware, evoluíram para sistemas integrados em diferentes níveis através de redes industriais, dispondo tanto de informações de processo (tipicamente advindas de CLPs ou sensores e atuadores inteligentes) quanto informações gerenciais, de logística ou estratégicas.

O aumento da complexidade dos processos, por sua vez, gera uma demanda por softwares cada vez mais sofisticados e de utilização mais complexa, o que pode exigir do operador diferentes etapas de configuração, utilização de múltiplas aplicações simultaneamente e um profundo conhecimento do processo e da estrutura lógica das aplicações.

O estudo desenvolvido no presente Projeto de Fim de Curso (PFC) está focado em aplicações de operação, monitoramento e controle de plantas na indústria petroquímica. O trabalho está inserido no escopo do projeto de pesquisa "Desenvolvimento de Algoritmos de Controle Preditivo Não Linear e de Avaliação de Desempenho de Controladores Preditivos para Plataformas de Produção de Petróleo", e foi realizado no Departamento de Automação e Sistemas (DAS) da Universidade Federal de Santa Catarina (UFSC). O projeto tem como parceiro a Petrobás, através do seu Centro de Pesquisas e Desenvolvimento Leopoldo Américo Miguez de Mello (Cenpes).

Tendo por finalidade a implementação de algoritmos de controle avançado em plantas da indústria petroquímica, a equipe do projeto se divide em duas frentes: uma equipe de controle, que busca obter as sintonias e estruturas de controle mais adequadas para cada plataforma, e uma equipe de software, na qual foram desenvolvidas as tecnologias deste trabalho, que tem como principal função implementar as soluções da equipe de controle e integrá-las com as aplicações que realizam a operação e controle em campo.

Nas próximas seções, serão apresentados os problemas específicos que se deseja solucionar com os desenvolvimentos desse projeto, bem como um panorama geral das tecnologias utilizadas.

1.1 Infraestrutura e operação

Uma das características importantes no desenvolvimento de novas soluções de controle é a interoperabilidade com sistemas já implementados e que se encontram em operação. A exemplo de sistemas de alta complexidade bem consolidados, como *Enterprise Resource Planning Suites*, foi utilizado o conceito de aplicações modulares intercomunicáveis no contexto do projeto de

pesquisa.

Nas refinarias e plataformas de produção da Petrobrás praticamente toda operação, supervisão e controle dos equipamentos em campo é feita através de um *software* especificamente projetado para esta finalidade. O *software* é chamado Módulo de Procedimentos Automatizados (MPA), tendo sido construído pelo Instituto Tecgraf, da Pontifícia Universidade Católica do Rio de Janeiro (PUC-RIO).

O MPA possui um módulo de configuração (MPA Client), onde é necessário definir o projeto a ser implementado, e um módulo de execução (MPA Server). Os dados de projeto do MPA ficam armazenados em três arquivos diferentes, cada um com uma etapa de configuração, que posteriormente são carregados para o servidor de execução MPA.

Os algoritmos de controle desenvolvidos pelo projeto são implementados em uma *Dynamic Link Library* (DLL) utilizando a linguagem C++. Na DLL estão presentes funções de inicialização, que carregam as sintonias e definições de controle iniciais de um arquivo **.xml**, e funções que executam o algoritmo de controle desejado, recebendo medições das variáveis e outras configurações do MPA, e retornando o sinal a ser aplicado na planta. O MPA, por sua vez, se comunica com os atuadores em campo através de um servidor *OLE for Proccess Control* (OPC).

No ambiente de simulação a dinâmica da planta deve ser modelada em separado, por exemplo, através do software Matlab, e a comunicação entre a planta e o MPA fica por conta do servidor OPC. A estrutura simplificada de simulação atualmente em uso é exibida na Figura 1.

ã

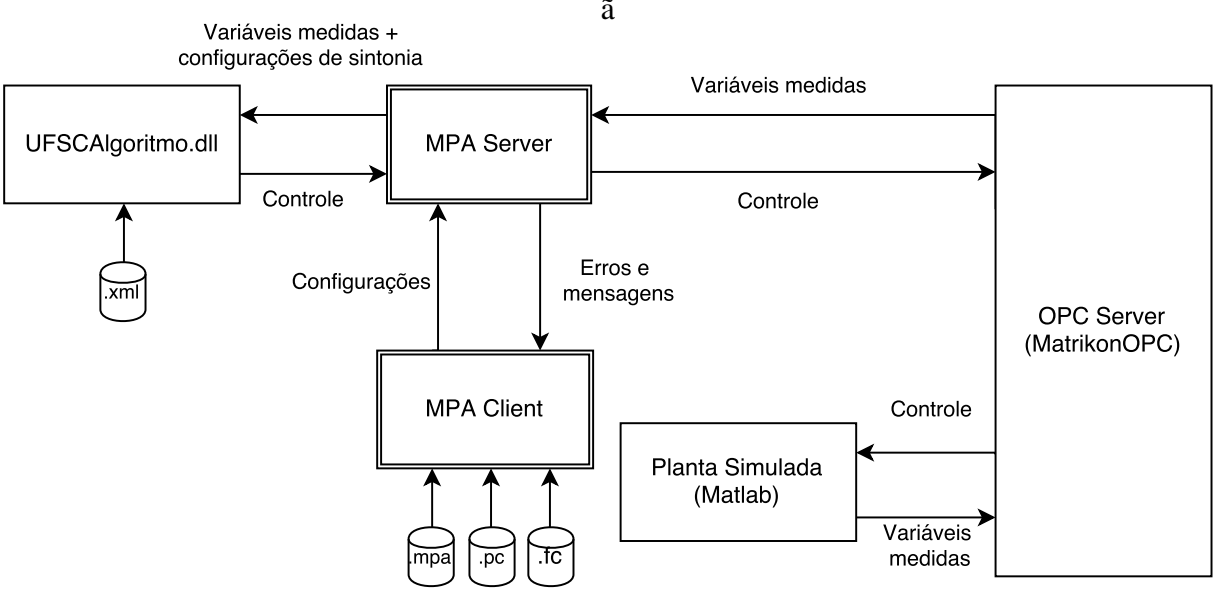

Figura 1 – Visão Geral

1.2 Problemática e motivação

Além de uma alta confiabilidade e robustez da estrutura de controle, um dos grandes objetivos do projeto é a capacidade de utilização da mesma estrutura de controle implementada pela DLL em diferentes plataformas de extração de gás natural. Apesar de o ajuste de sintonias, pesos e modelos da planta ser feito no **.xml** de configuração, cada projeto de controle depende também de uma correta configuração do MPA.

Para que o controle tenha acesso aos dados da planta, seja ela simulada ou em operação, é necessário executar três etapas de configuração no MPA: Pré-Configuração; Definição de Planta e Definição de Diagrama de Execução.

O Diagrama de Execução (**.fc**) define a ordem de execução das funções. Ele é responsável, por exemplo, por implementar o fluxo básico de controle (Ler Variáveis Medidas do OPC -> Chamar DLL para Cálculo do Controle -> Escrever Sinal de Controle no OPC -> Esperar e Repetir).

O arquivo de definição da planta (**.pc**) armazena Pontos de Controle e os Equipamentos instanciados na planta. Pontos de Controle contêm as *tags* de comunicação com o servidor OPC e indicam o endereço onde o MPA deve buscar informações sobre as variáveis da planta. 'Equipamentos' são tipos de dados específicos do MPA, podendo simbolizar tanto um equipamento físico da planta, como um 'Compressor' ou entidades abstratas como uma 'Variável Manipulada' ou mesmo um 'Arquivo de Diagnóstico'. No arquivo **.pc**, a planta está definida de forma final, i.e., todos os componentes são devidamente instanciados nas quantidades presentes em campo, não sendo permitida a adição de novos equipamentos ou pontos após o envio das configurações para o Servidor MPA.

Já no arquivo de Pré-Configuração (**.mpa**) são definidas as classes de equipamentos disponíveis para instanciação, sendo tais classes criadas numa etapa anterior à definição da planta. Nele é possível definir quais são os atributos presentes em determinado tipo de equipamento, escrever funções que ficam disponíveis para os diagramas de execução, ou mesmo configurar políticas de acesso para cada variável. Desta forma, na pré-configuração, dizemos que podem existir 'Equipamentos' do tipo 'Compressor', e que um 'Compressor' pode conter um atributo 'Vazão' do tipo 'Real'. Na definição da planta, por sua vez, elencamos que a aplicação atual possui um número *n* de compressores, e para cada um deles, associamos uma *tag* OPC que indica o valor da vazão daquele compressor. Essa configuração multietapas permite uma grande flexibilidade das aplicações e permite um desacoplamento entre o projeto da pré-configuração, da planta e dos diagramas de execução.

A interface do MPA possui abas para gerenciamento das plantas (**.pc**) e dos diagramas de execução (**.fc**), o que torna a configuração dos projetos simples e elegante. No entanto, a pré-configuração ocorre através da importação de um arquivo **.mpa** já finalizado, sem a possibilidade de edição dentro do software. Para modificar funções ou adicionar equipamentos customizados,

o usuário precisa abrir o arquivo com um editor de textos e codificar os elementos diretamente na linguagem de programação Lua [3], atentando devidamente à estrutura da pré-configuração. A inserção de parâmetros impróprios ou a ausência de elementos obrigatórios da estrutura do arquivo de pré-configuração fazem com que o arquivo não consiga ser importado para o MPA, o que inviabiliza a instanciação da planta e a continuidade do projeto. Desta forma, a pré-configuração se mostra bastante sensível a alterações, além de ser de baixa legibilidade, pois mesmo arquivos simples podem conter milhares de linhas de código.

Mesmo o manual de usuário do MPA [1] não se aprofunda na etapa de pré-configuração, apenas citando que "Usualmente, a fase de pré-configuração é feita poucas vezes no processo de utilização do MPA". Embora a afirmação se prove verdadeira para a execução em campo, onde a alteração de equipamentos da planta não costuma ser frequente, no contexto do desenvolvimento de novos algoritmos de controle (e validação em diferentes plataformas), há a necessidade de adequar as classes de equipamentos para cada plataforma. Isso se deve ao fato de que os equipamentos podem ter diferentes atributos e limitações, variando de fabricantes, faixas de operação, modelos, entre outros.

No contexto do projeto de pesquisa no qual o presente trabalho se encontra inserido, 'Equipamentos' no MPA podem ser classes abstratas que armazenam informações pertinentes para o controle. Desta forma, para explorar novas estruturas de controle, necessita-se alterar chamadas da DLL ou mesmo parametrizar configurações antigas. Para tanto, faz-se necessário alterar o arquivo de pré-configuração, tendo como única fonte de informação arquivos de pré-configurações existentes. Para se ter uma ideia, ao longo do projeto em questão - antes de se iniciar este trabalho - foram modeladas três plataformas completas, sendo que inúmeras alterações foram realizadas nos arquivos de pré-configuração. Isso dá um indicativo sobre a relevância de se trabalhar com questões relacionadas à geração de arquivos de pré configuração.

1.3 Objetivos do trabalho

Tomando como base a própria interface de configuração de plantas do MPA, e com a intenção de simplificar etapas de criação da pré-configuração, objetiva-se o desenvolvimento de uma aplicação *stand-alone* que permita a definição de classes de equipamentos de maneira visual, de forma a abstrair a estrutura do código para o usuário, e que a partir disso seja capaz de gerar arquivos de pré-configuração semanticamente válidos para importação no MPA. A solução também deve ser capaz de ler arquivos **.mpa** já existentes para facilitar a edição de equipamentos e atributos, bem como permitir que sejam criadas bibliotecas com os componentes utilizados mais frequentemente.

Também é possível listar os objetivos específicos:

- Análise da estrutura dos arquivos de pré-configuração, identificação dos componentes e

classificação em obrigatórios ou opcionais.

- Criação de um processo de transformação para gerar arquivos em conformidade com a estrutura identificada.
- Criação de um processo de transformação para leitura de arquivos **.mpa** para objetos lógicos.
- Integração das transformações em uma aplicação *stand-alone*, utilizando uma interface gráfica intuitiva.

Dessa maneira, pretende-se fazer um estudo mais minucioso de tecnologias e ferramentas para solucionar cada um dos objetivos propostos. O desenvolvimento de transformações de código e de interfaces gráficas são assuntos bastante presentes na literatura, com tecnologias consolidadas que serão apresentadas posteriormente. Já para a estrutura dos arquivos de pré-configuração, em paralelo a este trabalho, uma tese de mestrado no escopo do mesmo projeto de pesquisa se encontra em desenvolvimento com a proposição de um metamodelo para a criação de software para a indústria petroquímica. O Gerador de Código para Definição de Classes de Equipamentos adquire, portanto, como objetivo secundário, a validação da estrutura do metamodelo mencionado e fornece sua contribuição científica como um estudo de caso da utilização dessa estrutura.

Do ponto de vista dos *stakeholders* do projeto de pesquisa, espera-se que a definição de classes de equipamentos para novas plataformas seja mais rápida e confiável com a utilização do Gerador, e que outras contribuições futuras possam ser incorporadas à aplicação.

1.4 Da disposição dos capítulos

Este documento se divide em 6 capítulos, que permitem um melhor aprofundamento em cada aspecto do desenvolvimento. No Capítulo 2, é apresentado o software utilizado para integração dos controladores, o Módulo de Procedimentos Automatizados (MPA) e a estrutura dos arquivos de pré-configuração. No Capítulo 3, são mostradas as tecnologias nas quais a solução foi desenvolvida, nominalmente as transformações através de *Java Emitter Templates* (JET) e *ANother Tool for Language Recognition* (ANTLR), bem como o metamodelo utilizado. No Capítulo 4, a solução proposta é detalhada de maneira mais aprofundada, assim como o fluxo de desenvolvimento a ser seguido e o modelo de dados da aplicação. No Capítulo 5 estão presentes os detalhes da implementação dos processos de transformação de código e no Capítulo 6 são mostrados os resultados obtidos no contexto do projeto de pesquisa. Por fim, no Capítulo 7, são elencadas algumas conclusões e resultados futuros.

2 Análise da aplicação legada

Para compreender a natureza do gerador de código proposto neste trabalho e sua relevância para o projeto de pesquisa, apresenta-se em maior profundidade o Módulo de Procedimentos Automatizados (MPA). O software é utilizado para executar os algoritmos de controle avançado desenvolvidos no projeto relacionado e possui algumas particularidades de configuração que serão discutidas. O capítulo inicia com uma análise geral do software MPA e finaliza discutindo a estrutura dos arquivos de pré-configuração necessários para a instanciação de plantas.

2.1 MPA

O Módulo de Procedimentos Automatizados (MPA) é uma ferramenta voltada ao desenvolvimento de aplicações de controle e automação de processos industriais [4], criado pelo Instituto Tecgraf da PUC-RIO sob encomenda da Petrobras.

O MPA foi concebido de maneira modular para permitir uma grande flexibilidade de uso e ser adaptável à diferentes sistemas. Ele é dividido em um servidor de execução (MPA Server) e um módulo de configuração e gerenciamento (MPA Client).

O fluxo de desenvolvimento de uma aplicação de controle através do MPA se inicia pela descrição dos equipamentos presentes na planta a ser controlada e numa etapa de pré-configuração. A primeira etapa é realizada externamente ao MPA, direto na linguagem de programação Lua, num arquivo com extensão **.mpa**. O conjunto de classes de equipamentos a ser definido pode conter representações de atributos físicos dos equipamentos (como vazões, pressões, temperaturas, concentrações) e também uma série de funções. As funções são individuais de cada equipamento e servem para representar ações específicas, como por exemplo *abrir válvula*, *soar alarme*, *estimar custo*, entre outras. Uma vez definidas as classes de equipamentos com seus atributos e funções é necessário importar o arquivo **.mpa** gerado na aplicação MPA Client.

Uma fez feita a importação do arquivo de pré-configuração é necessário fazer a configuração da planta. Para tanto, o MPA Client fornece uma interface gráfica para a configuração de Plantas (gerando arquivos de configuração com extensão **.pc**) e Diagramas de Execução (associados a arquivos com extensão **.fc**). A Figura 2 mostra um exemplo de instanciação de um equipamento do tipo ‘Forno’ em uma planta específica.

Na aba “Planta”, as classes de equipamento da pré-configuração ficam disponíveis em uma árvore, à esquerda da janela, e devem ser instanciadas de acordo com o projeto de cada planta. Ao selecionar uma classe de equipamento na árvore, uma lista com todos os equipamentos daquele tipo presentes na planta é exibida à direita, enquanto um resumo das informações sobre o

equipamento é mostrado na parte inferior. Os atributos do equipamento aparecem como colunas na lista de ‘Instanciação de Equipamentos’, e possuem um esquema de cores específico.

No exemplo, o forno ‘oven_1’ possui como atributos duas válvulas de controle (FT209 e FT219), que foram descritas no MPA como equipamentos da classe ‘Válvula Simples’. Isso significa que elas devem constar na lista de ‘Instanciação de Equipamentos’ quando a classe ‘Válvula Simples’ for selecionada na árvore, e que cada uma delas pode conter atributos e funções próprias. Atributos que fazem referência a outros equipamentos recebem uma cor de fundo verde na lista. De maneira semelhante, são atributos do forno ‘oven_1’ valores de temperatura de passe (TI302 e TI306), que foram descritos no MPA com a classe ‘Ponto Real’. Classes do tipo ‘Ponto’ são objetos especiais do MPA que armazenam *tags* de comunicação com o servidor OPC, o que permite obter medições de sensores em campo sempre que necessário. Atributos do tipo Ponto recebem uma cor de fundo azul na listagem de equipamentos. Por fim, é possível definir atributos com valores simples (como Real, Inteiro ou String), que recebem uma cor de fundo branca.

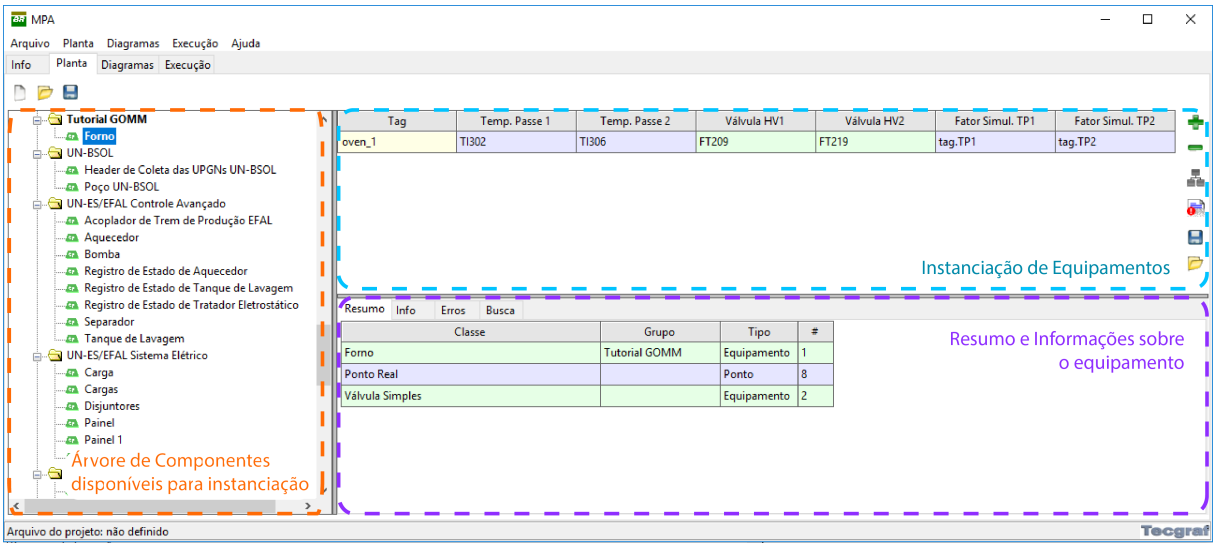

Figura 2 – Interface MPA Client - Aba ‘Planta’

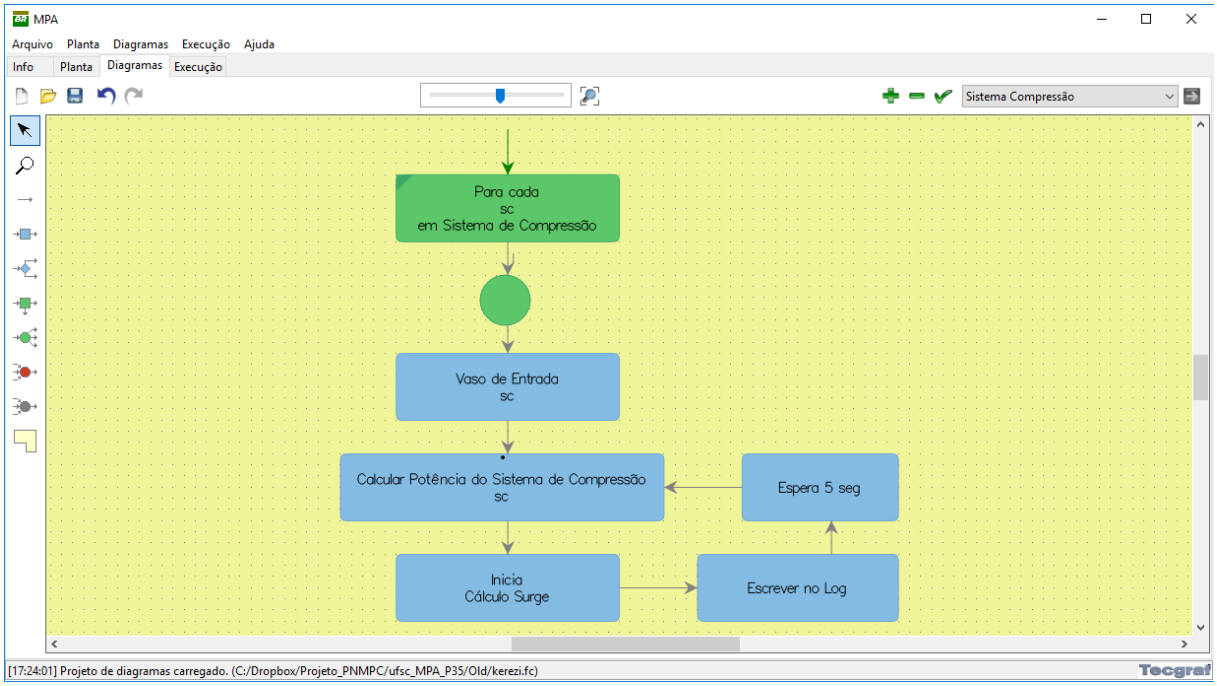

Figura 3 – Interface MPA Client - Aba ‘Diagrama’

Por sua vez, a aba “Diagramas” permite definir o fluxo das operações a serem executadas. Isso é feito através de blocos, usando-se a linguagem *Sequencial Function Charts* ou uma linguagem de Fluxos, criada especialmente para o MPA. Os blocos no diagrama executam as funções definidas na pré-configuração enquanto as setas e outros operadores lógicos (Iteradores, Escolhas, Executores) definem a sequência de operação das funções. Dessa maneira, em tempo de execução, os blocos de função têm acesso aos atributos dos equipamentos instanciados na aba ‘Planta’ e, por consequência, têm acesso aos atributos físicos do processo - quando a definição de atributos for feita por pontos OPC.

Na Figura 3 é mostrado um exemplo de diagrama de fluxos que tem como finalidade fazer o monitoramento de um sistema de compressão. A sequência de execução calcula parâmetros de interesse para cada sistema de compressão a partir de variáveis medidas e escreve um registro das respostas. A mesma estrutura de *loop* contínuo pode ser utilizada para realizar o controle do processo, implementar estimadores, sistemas de segurança, alarmes, entre outros, de acordo com a necessidade de cada projeto. Uma vez finalizadas as configurações, o projeto está preparado para o envio ao servidor de execução *MPA Server*.

O MPA Server é responsável pela operação e controle na planta. Tipicamente, o servidor roda continuamente em um PC e se comunica com os dispositivos de campo através de uma rede OPC (*OLE for Process Control*). A identificação dos equipamentos físicos, as *tags* OPC, bem como a sequência das operações executadas devem ser importadas do cliente.

As configurações de planta e dos diagramas podem ser enviadas ao servidor através

da rede ou através de um arquivo local com extensão **.app**. No entanto, quando configurado a partir de arquivo, o servidor também requer como argumento de execução a mesma pré-configuração (**.mpa**) utilizada no cliente, contendo as classes de equipamentos e o código das funções disponíveis na planta.

Por conta da conectividade com o servidor OPC, os projetos em execução podem ser integrados com sistemas já existentes, em uma nova camada de supervisão. O diagrama completo do fluxo de configuração de uma aplicação no MPA pode ser visto na Figura 4.

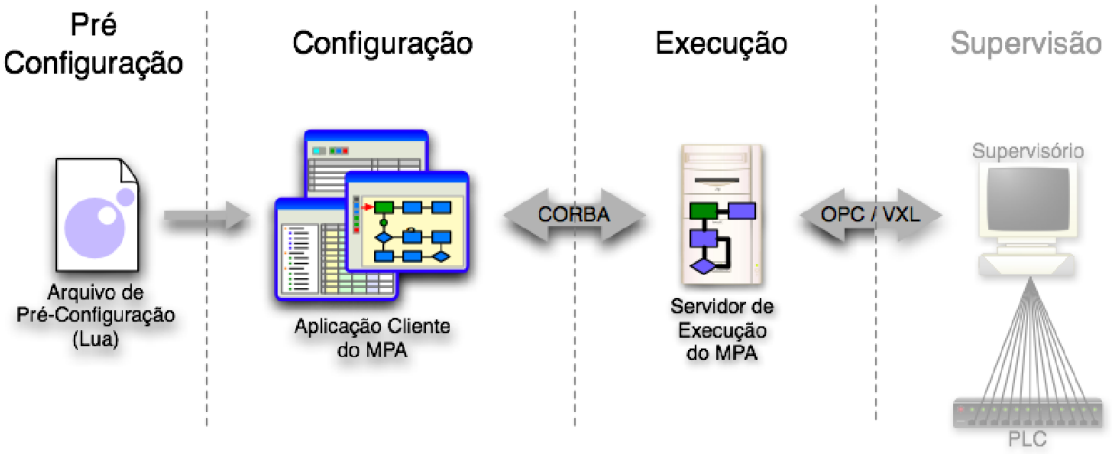

Figura 4 – Fluxo do MPA [1]

No escopo do projeto de pesquisa da UFSC, as plantas, diagramas e arquivos de pré-configuração possuem equipamentos e funções específicas de controle, além de uma interface de comunicação com funções da biblioteca **UFSCAlgoritmo.dll**, uma biblioteca dinâmica escrita em C++ que implementa o *Model Predictive Control*, estrutura estudada pela equipe de controle, atualmente em validação em diferentes plataformas. Tendo em vista que o presente trabalho se concentra em criar soluções para a etapa de Pré-Configuração do MPA, não serão discutidos detalhes acerca da implementação da biblioteca de controle.

2.2 Arquivos de Pré-Configuração

Conforme antecipado previamente, arquivos de pré-configuração podem conter a descrição de classes de equipamentos e funções. É necessário carregá-los ao criar/abrir um projeto no *MPA Client* e também no momento da inicialização do *MPA Server*. Destaca-se que mudanças de plataforma ou de estratégias de controle costumam exigir alterações no arquivo de pré-configuração.

Para criar novos arquivos de pré-configuração o projetista deve ter um profundo conhecimento sobre o funcionamento do MPA, saber os identificadores dos atributos que deseja

modelar, elementos obrigatórios e ter uma constante preocupação com a estrutura do arquivo. A automação do processo de criação de arquivos de pré-configuração, portanto, deve ser capaz de preservar esse conhecimento. É necessário analisar e mapear a estrutura dos arquivos para, posteriormente, incorporar essas regras durante a geração automática de código a partir de uma interface gráfica.

Apesar da extensão específica, um arquivo **.mpa** é interpretável por editores de texto simples, e é composto por uma mistura entre códigos na linguagem Lua e estruturas aninhadas, que lembram vagamente a definição de objetos em JSON.

Em alto nível, a pré-configuração pode conter três elementos: Classes de Equipamentos, Funções e Blocos de Código. A seguir se apresenta a estrutura de cada um desses elementos em detalhes, relacionando as definições na pré-configuração e no *MPA Client*.

2.2.1 Classes de Equipamentos

A definição de uma classe de equipamento ocorre dentro do identificador `class{ }`. Dentro dele, a definição de atributos é feita seguindo uma estrutura de hash ‘`parametro = "valor",`’ ou ‘`parametro = { lista_de_parametros },`’. Na Figura 5, é possível ver um exemplo de uma classe de equipamento chamada ‘Compressor’ em uma pré-configuração. O atributo `id` é obrigatório para todos os equipamentos, e é usado para referenciar a classe criada. O parâmetro `name` define o nome a ser exibido na árvore de classes, à esquerda (ver Figura 2). O parâmetro `description` fica disponível na aba ‘Info’ do equipamento, na parte inferior da aba de definição de planta (ver Figura 2).

Um `group` aninha a classe de equipamentos a uma pasta específica na árvore, para melhorar a organização, enquanto um `bases` define relacionamentos de herança entre classes de equipamentos. Pode estar presente ainda o parâmetro booleano `isPoint`, que chaveia o aninhamento do equipamento para a pasta ‘Equipamentos’ ou ‘Pontos’ na árvore de componentes da planta.

```
class{ id = "compressor", name = "Compressor", group = "Sistema de Compressão",
  bases = {},
  description = [[Compressor.]],
  attributes = {
    {  id = "pressao_succao",   name = "Pressão de Sucção", type = "REAL_POINT",
       access = "grw",
       description = [[Ponto real que informa a pressão de sucção do compressor.]],
    },
    {  id = "temperatura_succao",   name = "Temperatura de Sucção", type =
    "REAL_POINT",
       access = "grw",
       description = [[Ponto real que informa a temperatura de sucção do
       compressor.]],
    },
    {  id = "pressao_descarga", name = "Pressão de Descarga", type = "REAL_POINT",
       access = "grw",
       description = [[Ponto real que informa a pressão de descarga do compressor.]],
    },
    {  id = "temperatura_descarga", name = "Temperatura de Descarga", type =
    "REAL_POINT",
       access = "grw",
       description = [[Ponto real que informa a temperatura de descarga do
       compressor.]],
    },
    {  id = "vazao_massica",    name = "Vazão Mássica", type = "REAL_POINT",
       access = "grw",
       description = [[Ponto real que informa a vazão mássica do compressor.]],
    },
  },
  methods = {
    { id = "estimar_vazao_primeiroCompressor", name = "Estimar vazão do 1°
    Compressor da Linha",
      description = [[Calcular vazão mássica do 1° compressor de uma linha de
      compressão.]],
      parameters = {
        { name = "Vaso de entrada", type = "vaso_entrada" },
      },
      results = {
          { name = "Vazão", type = "REAL" },
      },
      code = [============================================================[
                function(self, vaso)
                  if complex.type(mtc) == 'complex' then
                    print('mtc COMPLEXO e real_mtc =', real_mtc)
                    return real_mtc
                  else
                    print('mtc nao COMPLEXO e mtc =', mtc)
                    self.vazao_massica:write(mtc)
                    return mtc
                  end
                end
              ]============================================================],
    },
  },
}
```

Figura 5 – Exemplo de um Equipamento na Pré-Configuração

Os últimos componentes que podem ser definidos em uma classe de equipamentos são listas. Uma lista de `attributes` é apresentada como colunas na janela de instanciação de equipamentos (Figura 6). Cada `attribute` da lista pode conter `id`, `name` e `description` com funções similares às descritas anteriormente. O parâmetro `type` define o tipo do atributo, que pode ser básico (String, Inteiro, Booleano, Real), Ponto (com conectividade OPC) ou mesmo um equipamento.

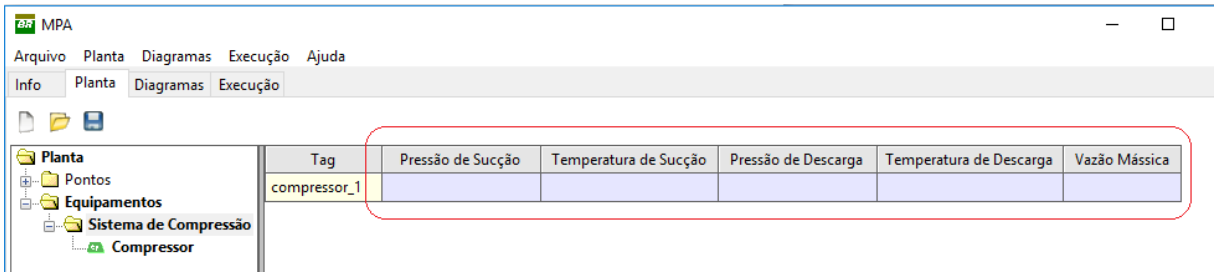

Figura 6 – Attributes da pré-configuração como colunas do equipamento na definição de planta no MPA

O parâmetro `access` define permissões de leitura e escrita do `attribute`. O MPA pode criar automaticamente métodos 'Definir `attribute`' e 'Obter `attribute`' que ficam disponíveis para utilização no diagrama (Figura 7), baseado nas permissões de `access` de cada `attribute`.

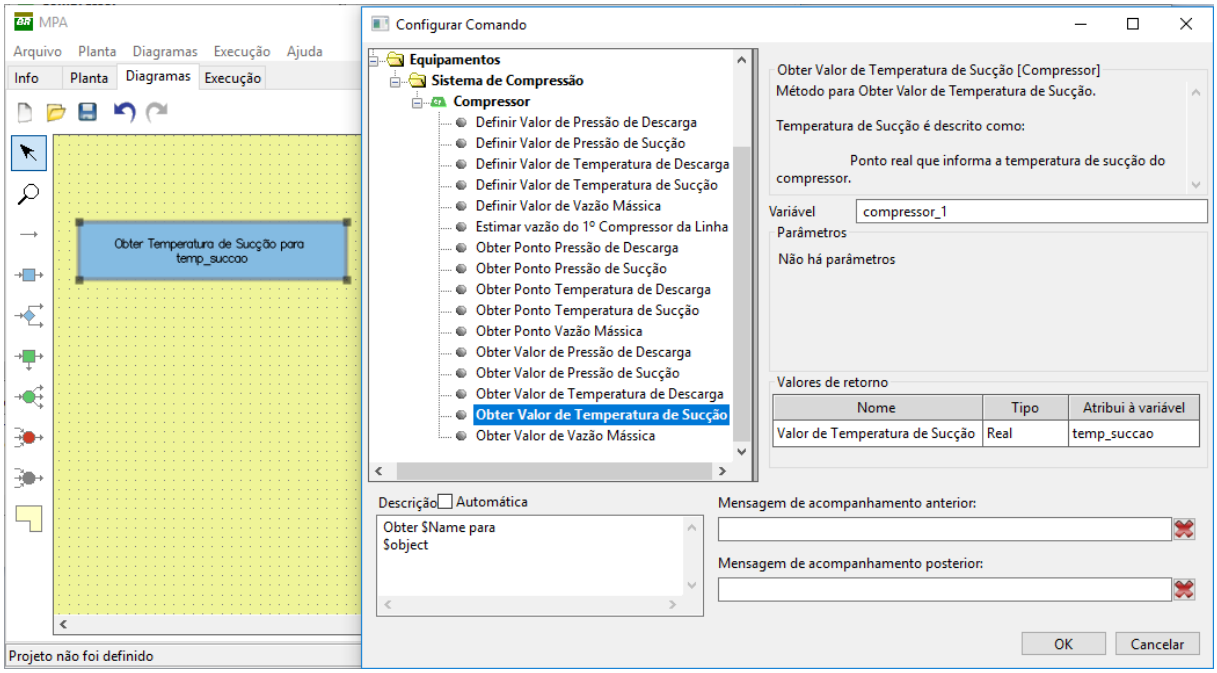

Figura 7 – Atribuição de uma função da pré-configuração a um bloco de função do diagrama MPA

Analogamente, uma lista de `methods` pode ser definida. Cada `method` da lista fica disponível para utilização no diagrama, na janela 'Configurar Comando' (Figura 7). Além de `id`, `name` e `description`, cada `method` possui um parâmetro `code`, que será a função interpretada pelo servidor ao entrar no bloco associado a esse `method` em tempo de execução, seguindo o diagrama. Cada `method` possui listas internas de `parameters` e `results`, que correspondem a variáveis de 'argumento' e 'retorno' do `code`. O `code` deve conter uma função escrita na

linguagem Lua com argumentos e retornos compatíveis com as listas definidas no `method` (funções em Lua permitem o retorno de múltiplos parâmetros).

2.2.2 Funções

Por serem construídos com linguagens de programação gráficas, os diagramas de execução do MPA são capazes de implementar funções naturalmente. No entanto, algoritmos de maior complexidade acabam requerendo diagramas muito grandes, o que dificulta a visibilidade e manutenção do código. Como alternativa, o MPA permite a definição de funções dissociadas de Equipamentos na Pré-Configuração.

As funções possuem o identificador `func{ }` e têm a mesma estrutura de um `method`, explicado anteriormente. A Figura 8 mostra um exemplo de função que calcula o valor médio em um vetor. As funções avulsas também aparecem na janela 'Configurar Comando' na criação de diagramas de execução(ver Figura 7).

```
1  func{ id = "average", name = "Média",
2      description = [[Função que recebe lista de valores reais e retorna a média
       desses valores.]],
3      parameters = { {name = "Lista", type = "REAL[]" }, },
4      results = { {name = "Média", type = "REAL" }, },
5      code = [=[ function (l)
6          local n = #l
7          if n == 0 then
8              return 0
9          end
10         local m = sum(l)
11         m = m / n
12         return m
13     end]=],
14 }
```

Figura 8 – Exemplo de uma Função na Pré-Configuração

As variáveis utilizadas dentro do `code` da função podem ser definidas localmente ou estar presentes dentro de equipamentos específicos. No segundo caso, mesmo com as classes de equipamentos definidas na pré-configuração, é necessário definir na planta (**.pc**) uma instância do equipamento desejado, e passar a instância como parâmetro da função no fluxo do diagrama (**.fc**) para o correto funcionamento.

2.2.3 Blocos de Código

A terceira macro estrutura presente nos arquivos de Pré-Configuração do MPA são blocos de código, que possuem um identificador '`code = [=[ ]=]`'. O bloco `code` pode conter código acessível globalmente para o MPA. Todos os blocos avulsos presentes no arquivo **.mpa** são concatenados e interpretados de maneira única. Para o projeto de pesquisa, é no `code` que são definidas as interfaces com as funções da **UFSCAlgoritmo.dll**.

Na Figura 9 é possível observar um bloco de código contendo três interfaces com funções da DLL (init_control, executa_iteracao e envia_ptb). É importante fazer uma distinção entre inicializações (como a importação da *Foreign Function Interface* 'alien' na linha 5, necessária para a chamada de funções de bibliotecas externas ou a declaração de arquivos externos, como a UFSCAlgoritmo na linha 7) e interfaces entre MPA e funções da DLL (linhas 9, 14 e 20).

A definição de interfaces entre as funções da DLL de controle UFSCAlgoritmo seguem a seguinte estrutura:

id_funcao_no_mpa = assert(**Nome_da_DLL.id_funcao_na_dll**)

id_funcao_no_mpa:types(**lista_de_argumentos**)

onde `lista_de_argumentos` contêm os tipos de dados dos argumentos da função definida na DLL. O primeiro argumento da lista é do tipo 'int', e define o status de retorno da função. Os parâmetros seguintes costumam ser ponteiros para buffers, onde são compartilhadas as variáveis de interesse.

```
1   include('base.mpa')
2
3   code = [=[
4
5   alien = require "alien"
6
7   UFSCAlgoritmo = assert(alien.UFSCAlgoritmo)
8
9   init_control = assert(UFSCAlgoritmo.IniciaMPA)
10  init_control:types("int",
11  "pointer"
12  )
13
14  executa_iteracao = assert(UFSCAlgoritmo.ExecutaMPA)
15  executa_iteracao:types("int",
16  "pointer"
17  )
18
19
20  envia_ptb = assert(UFSCAlgoritmo.MPA_perturbacoes)
21  envia_ptb:types("int",
22  "pointer",
23  "pointer",
24  "pointer",
25  "pointer"
26  )
27
28  ]=]
```

Figura 9 – Exemplo de um Bloco de Código na Pré-Configuração

A partir dessa declaração, `id_funcao_no_mpa` passam a ser funções no MPA, ainda invisíveis para o *MPA Client*, mas acessíveis dentro das funções avulsas ou dos métodos de equipamentos da Pré-Configuração para realizar chamadas externas para a DLL. Assim, o diagrama de execução do MPA consegue executar os comandos de controle presentes na **UFSCAlgoritmo.dll**, inicializar sintonias, passar medições de variáveis do OPC para cálculo do

controle, interpretar códigos de erro ou *timeouts* dos algoritmos de otimização, entre outras funcionalidades, através das funções do MPA.

Uma vez identificada a estrutura e os elementos possíveis na Pré-Configuração, é preciso definir uma modelagem lógica capaz de abarcar todos os componentes presentes no arquivo **.mpa**. Com a definição de classes orientadas a objeto que armazenem as informações de interesse, é possível focar em transformações de leitura/escrita e na criação de uma ferramenta gráfica que auxilie na edição desses objetos.

Em paralelo ao estudo das tecnologias para a geração das transformações entre objetos lógicos e código, um metamodelo das classes de equipamentos, baseado no MPA, foi desenvolvido no escopo de um trabalho de mestrado no mesmo projeto.

No próximo capítulo se apresenta em maiores detalhes o metamodelo em questão, bem como uma revisão de duas ferramentas da literatura estudadas para implementação de transformações de código. A primeira, *Java Emitter Templates* (JET), busca auxiliar na criação de transformações para geração de arquivos textuais com estruturas pré-definidas. A segunda é a *ANother Tool for Language Recognition* (ANTLR), que tem como objetivo simplificar a criação de transformações para análise sintática e leitura de arquivos. Em conjunto com o metamodelo,

3 Ferramentas Utilizadas

O escopo completo da implementação descrita neste trabalho pode ser dividida em quatro problemas principais: a modelagem lógica da estrutura de arquivos de pré-configuração do MPA; a criação de transformações que gerem automaticamente arquivos **.mpa** a partir de objetos lógicos; a criação de transformações capazes de interpretar arquivos **.mpa** e extrair objetos lógicos a partir do texto; e, por fim, a criação de uma interface gráfica que incorpore os três elementos, de forma a simplificar o desenvolvimento de novas aplicações.

Neste capítulo são apresentadas discussões gerais a respeito das tecnologias utilizadas para resolver cada um dos problemas elencados. É traçada uma visão geral sobre o metamodelo relacionado, o qual serviu como base para a estrutura de dados das informações presentes no arquivo de pré-configuração, além de explorar a fundamentação por trás das ferramentas *open source* utilizadas para cada uma das transformações: os templates JET e as gramáticas da ANTLR.

3.1 O Metamodelo

O processo de busca por novas estratégias de controle, especialmente em Pesquisa e Desenvolvimento, requer uma alta carga de experimentação. Ferramentas de software costumam ser grandes aliadas nesse processo, agilizando a obtenção de parâmetros e sintonias, na simulação e análise de resultados, na otimização dos algoritmos, entre outras funcionalidades. Não raramente, novas ferramentas são criadas para automatizar processos repetitivos.

A criação de sistemas *ad-hoc* costuma ser lenta e sua confiabilidade depende do conhecimento do projetista sobre o domínio do problema e as aplicações relacionadas. No projeto de pesquisa, está sendo proposta uma abordagem baseada em *Model Driven Design* para auxiliar a criação de ferramentas que dialogam com o MPA.

De maneira geral, o trabalho é objeto de uma tese de mestrado que busca simplificar a criação e manutenção de aplicações voltadas para a indústria petroquímica, através do uso de modelos. Entre os resultados obtidos está a proposição de um metamodelo de classes que organiza os relacionamentos entre objetos, baseado nos componentes presentes no MPA. A estrutura inicial proposta é apresentada na Figura 10.

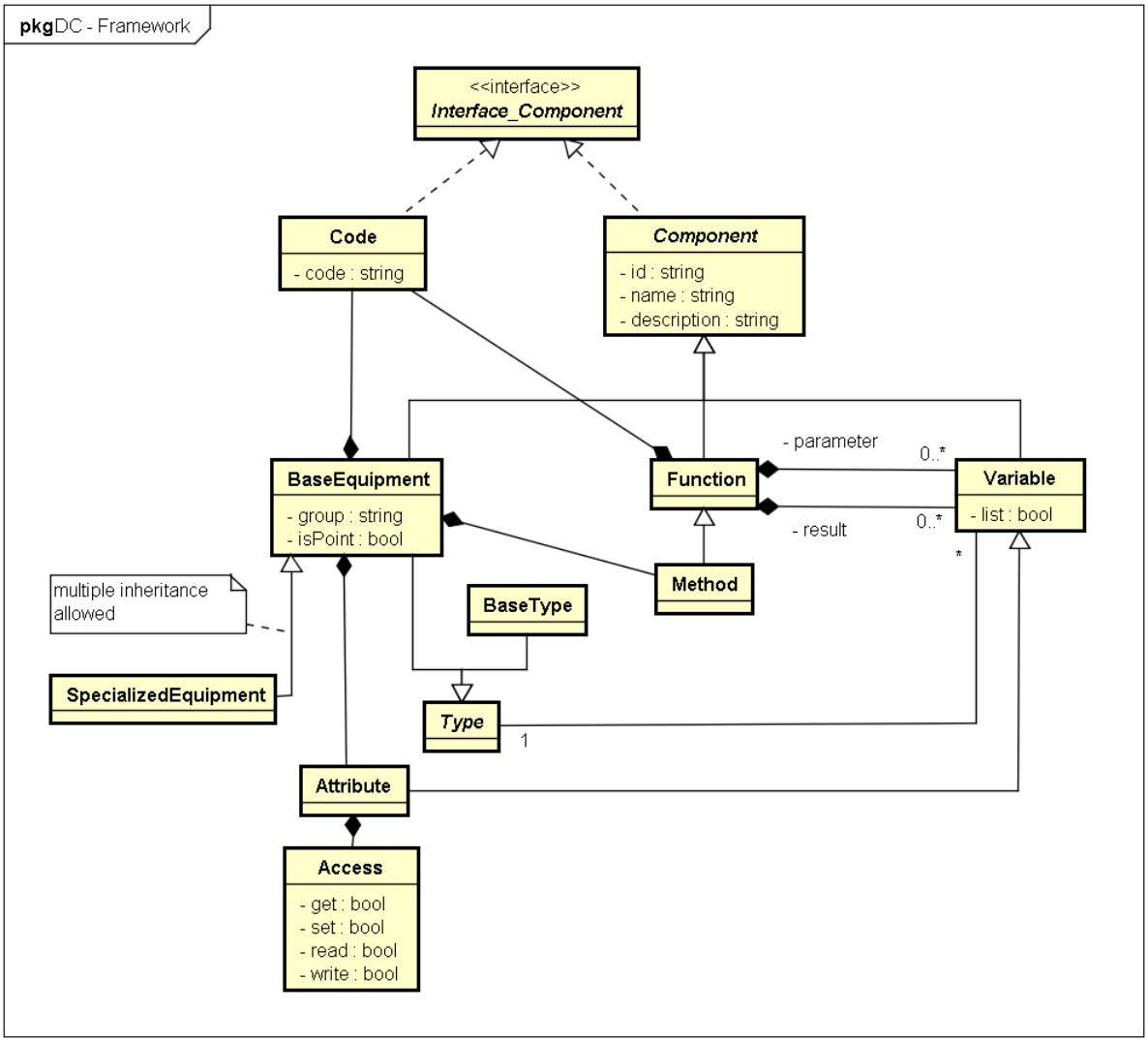

Figura 10 – Estrutura do metamodelo

A modelagem das classes foi desenvolvida utilizando o *Eclipse Modeling Framework* (EMF), um *plugin* para construção de aplicações usando MDD no software **Eclipse**. O metamodelo é construído com base no meta-metamodelo **Ecore**, que faz parte do EMF. Com o metamodelo é possível, dentro do próprio **Eclipse**, instanciar classes e gerar código automaticamente na linguagem Java (com *getters* e *setters* para cada atributo), de maneira similar a ferramentas baseadas em *Unified Modeling Language* (UML), por exemplo.

É possível observar uma grande similaridade dos relacionamentos das classes do metamodelo com a estrutura da pré-configuração apresentada no Capítulo 2.

Todas as classes do modelo, a exceção de `Code` e `Type/BaseType` recebem uma herança de `Component`, portanto possuem os parâmetros básicos `id`, `name` e `description`. A classe `BaseEquipment` remete ao Equipamento no MPA, pode estar associada a várias instâncias de `Attribute` e de `Method`, e pode ainda conter um `Code`.

Cada `Attribute` pode estar associado a políticas de acesso, modeladas na classe `Access`, ao passo que cada `Method`, por herdar a classe `Function`, pode estar associado a listas da classe `Variable` (que representariam parâmetros e resultados de função no mpa), além de possuir um `Code`, contendo o código a ser interpretado na execução.

Uma instância da classe `Variable` deve estar associada a um `Type`, que pode ser `BaseType` ou um próprio `BaseEquipment`, para permitir a modelagem de recursividades naturais nas relações entre equipamentos. Um exemplo é um Equipamento 'Máquina CNC' ter como atributo um Equipamento 'Motor'.

As classes do metamodelo proposto conseguem reproduzir com fidelidade as estruturas de 'Equipamento' e 'Função', presentes na Pré-Configuração do MPA. A estrutura 'Bloco de Código' também pode ser descrita em alto nível como um `Code`, mas uma maior granularidade é necessária para identificar as interfaces entre funções da DLL e o MPA, internas ao Bloco de Código.

A estrutura apresentada serviu como base para definição de classes utilizadas na implementação do Gerador. Eventuais simplificações ou expansões na estrutura para auxiliar o desenvolvimento são detalhadas no Capítulo 5.

Neste ponto, é salutar evidenciar algumas confusões que costumam surgir ao trabalhar com múltiplas camadas de desenvolvimento. A Figura 11 apresenta uma representação de um projeto utilizando MDD a partir de modelos UML, cuja estrutura ilustra os níveis de informação do projeto.

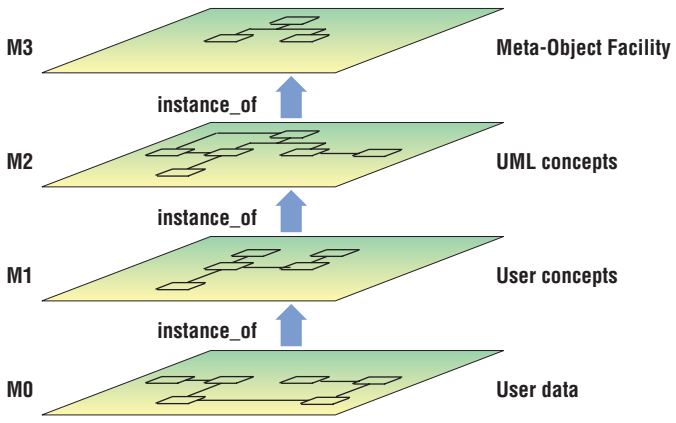

Figura 11 – Infraestrutura de modelagem padrão para o *Object Management Group* [2]

No fluxo de trabalho da criação de uma aplicação de controle através do MPA, os equipamentos definidos na planta (**.pc**) representam a camada mais baixa (M0), contendo os dados da planta. No exemplo apresentado na Figura 12, uma '**bomba_reservatorio**' (M0) é uma instância (objeto) da classe '**Bomba Simples**' (M1). A classe '**Bomba Simples**', definida na

pré-configuração, por sua vez, passa a ser uma instância (objeto) das classes do metamodelo (M2), e o próprio metamodelo é uma instância das classes do meta-metamodelo Ecore (M3) disponíveis no *plugin* EMF.

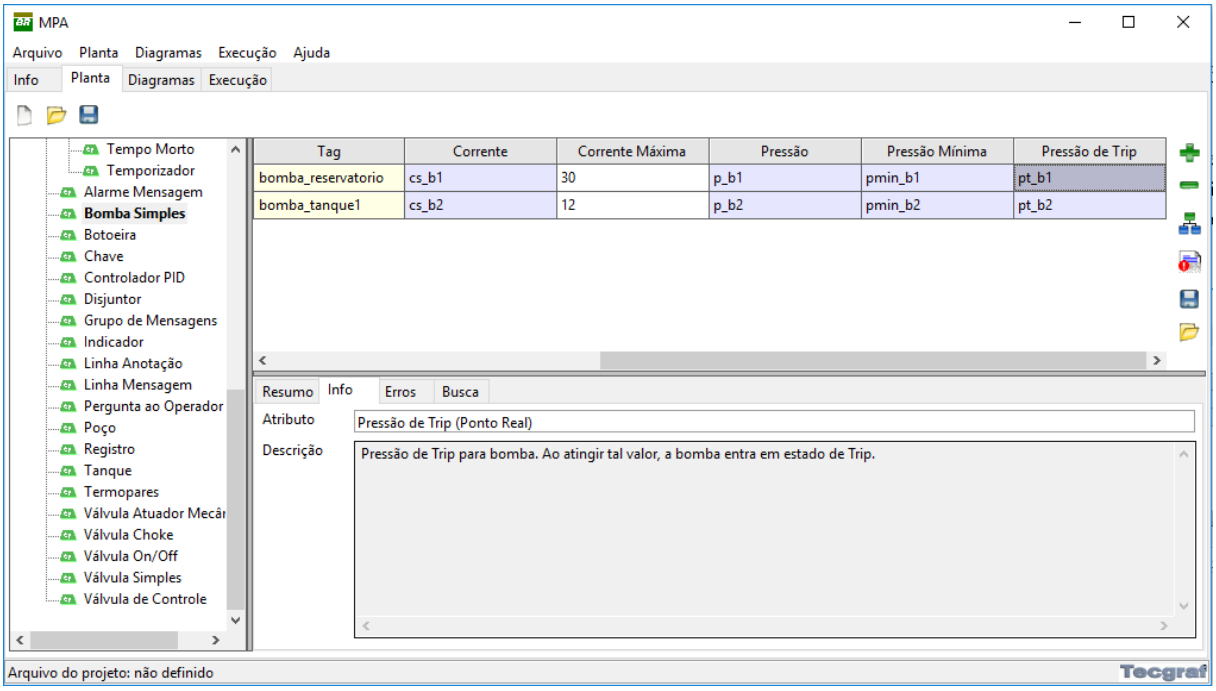

Figura 12 – Exemplo de configuração de planta no MPA

Dessa maneira, para o escopo do Gerador, trabalharemos com uma nomenclatura sempre no nível M1, onde os 'objetos lógicos' (instâncias em *runtime*) da aplicação são, na verdade, as classes que modelam os objetos na aplicação final (MPA).

3.2 Java Emitter Templates

Uma vez definida uma estratégia para armazenar a estrutura de dados das classes de equipamentos e funções presentes no arquivo pré-configuração, é preciso endereçar a escrita dos arquivos **.mpa** a partir dos objetos lógicos. Dentro do pacote de ferramentas disponíveis no EMF existem os Java Emitter Templates [5], que simplificam a construção de arquivos de texto a partir de objetos.

Projetos JET produzem transformações *Model to Text* (M2T) automaticamente a partir de *templates* contendo a estrutura de saída desejada. No contexto do MDD, é possível obter classes em diferentes linguagens utilizando transformações JET a partir de um único modelo conceitual de uma aplicação (em UML, por exemplo). Para isso, é necessário descrever no *template* a estrutura da linguagem de destino, instruções de verificação e quais campos dinâmicos devem ser adicionados para completar o arquivo gerado.

Uma vez definido o *template* em um projeto JET, uma ferramenta chamada **JET Builder** é chamada automaticamente. O JET Builder compreende as instruções e a estrutura modeladas no *template* e gera uma classe em Java com a transformação M2T.

Todas as classes geradas pelo JET Builder possuem uma função **generate()**, que aceita um único argumento como entrada (tipicamente, um modelo de dados). O conteúdo da função é uma mistura de séries de *prints* com a estrutura fixa definida no *template*, verificações condicionais e *prints* dos elementos dinâmicos presentes no modelo. A saída da função é a String com o texto completo.

A classe gerada pode, então, ser incorporada as mais diversas aplicações, e o conteúdo da função pode ser escrito em um arquivo, inserido em bancos de dados, ou mesmo apenas apresentado para o usuário. É possível ainda construir aplicações que utilizam o próprio JET Builder, permitindo a modificação do próprio *template* em tempo de execução do *software*. No entanto, a dependência do Eclipse ainda existe, sendo necessário rodá-lo, no mínimo, de maneira '*headless*' (sem a interface gráfica).

Na Figura 13 é possível ver o fluxo de desenvolvimento de um projeto utilizando transformações JET, além de um exemplo de *template* e sua classe de implementação correspondente, criada pelo JET Builder.

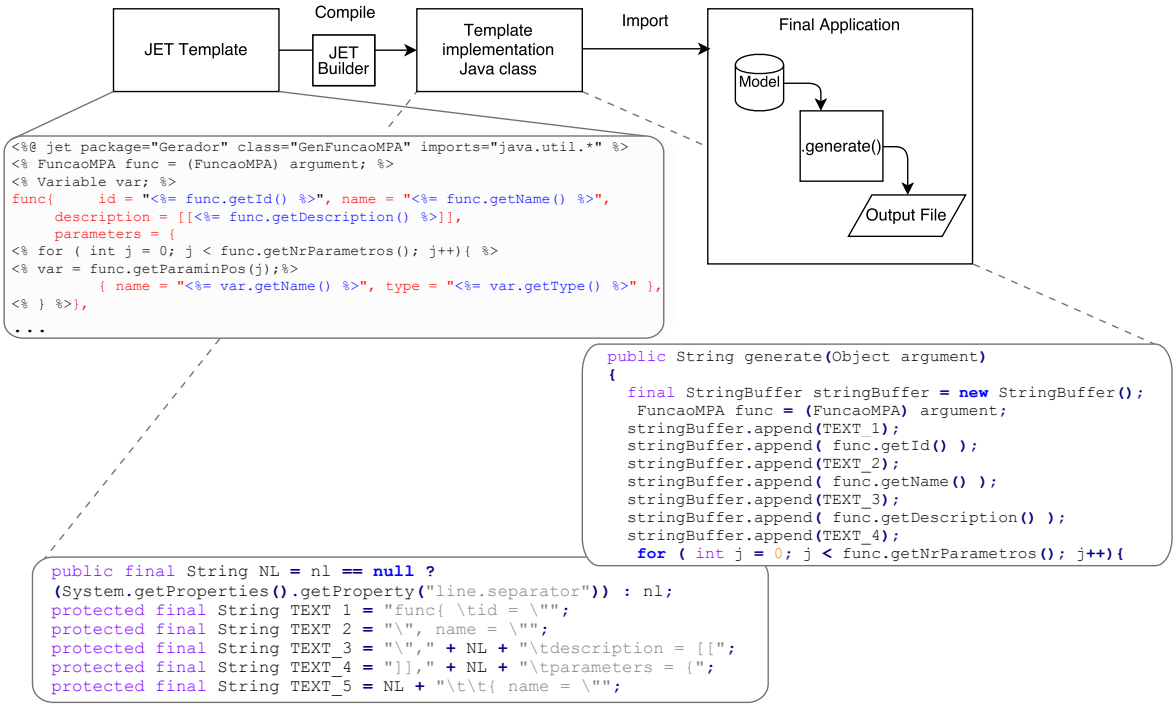

Figura 13 – Exemplo de fluxo de projeto JET

Um projeto JET no Eclipse possui uma pasta específica para a criação do *Templates*. A

extensão dos arquivos de *template* é definida, por convenção, a partir da extensão de destino da transformação incorporando o sufixo 'jet'. Uma transformação para XML, por exemplo, teria um *template* com a extensão .xmljet, e uma transformação para Java usaria .javajet.

Dentro do arquivo de *template*, um cabeçalho define informações sobre a classe de implementação, bibliotecas necessárias, *package* do projeto em que deve ser inserida, entre outras definições específicas. Internamente, na tradução do *template* para a classe de implementação, todo texto simples é seccionado por linhas e incorporado a um StringBuffer, que será o retorno da função generate().

A estrutura do *template* utiliza *tags* '**<% %>**' para definir trechos com código a ser interpretado pela classe, naquele ponto da geração do arquivo. Esses trechos não são adicionados ao StringBuffer, têm o texto transcrito literalmente para dentro da classe. Dessa maneira, validações condicionais com if-else e iteradores podem ser construídos dentro do próprio *template*, usando trechos de código Java entre as *tags*, fazendo com que o conteúdo seja impresso apenas quando desejado.

Uma segunda estrutura de *tags* '**<%= %>**' permite a execução de funções, adicionando automaticamente seu resultado ao StringBuffer. Tal construção é útil para incorporar informações do *argument* no arquivo final, como pode ser visto no exemplo da Figura 13. O *template* espera como argumento um modelo de **FuncaoMPA**, e utiliza os *getters* do modelo para adicionar as informações diretamente no texto final.

De maneira simples, a utilização dos JET Templates facilita o desenvolvimento de tradutores M2T por permitir ao projetista a definição da estrutura do arquivo sem se preocupar com as funções de escrita específicas por linguagem, gerenciamento de buffers ou mesmo com a legibilidade do código da transformação. Por outro lado, a implementação de estratégias mais sofisticadas (utilizando estruturas de ***for***s e ***if***s) continua possível utilizando as *tags*, dando a flexibilidade necessária para solucionar grande parte dos problemas de geração de código através de modelos.

Além disso, as transformações geradas podem ser incorporadas a *plugins* e utilizadas na criação de artefatos de software reutilizáveis, como *Domain Specific Languages* (DSLs). No contexto do MDD, as DSLs disponibilizam a desenvolvedores terceiros ferramentas de criação de modelos (em conformidade com um metamodelo para domínios específicos) que integram transformações M2T como as geradas pela ferramenta JET. Dessa forma, é possível definir aplicações completas através de modelos, e ainda obter código funcional em diferentes linguagens gerado automaticamente, ao executar transformações JET incorporadas.

3.3 ANother Tool for Language Recognition

Uma das ferramentas mais conhecidas de análise léxica, leitura e interpretação textual *open source* é a ANTLR. Utilizada em gigantes como Twitter, Hadoop, Oracle e NetBeans IDE para análise das mais diversas estruturas sintáticas (sejam interpretáveis por humanos ou por máquinas), a ferramenta alcança a 4ª versão com *reviews* extremamente positivos. Essa alternativa foi estudada para simplificar a criação de transformações capazes de interpretar os arquivos de pré-configuração .mpa e extrair as informações relevantes para objetos lógicos, em concordância com o metamodelo já apresentado.

Para explicar como a ANTLR funciona, primeiramente deve-se apresentar de maneira breve uma série de conceitos relacionados a interpretação de arquivos textuais. O elemento principal de tais programas é a 'Linguagem', que pode ser definida como um conjunto de sentenças válidas em uma estrutura, criadas a partir de frases, subfrases, e símbolos de vocabulário [6]. Aqui, buscamos dissociar o conceito de 'Linguagem' com as linguagens naturais (definidas pelas regras de cada Idioma) e verificar sua existência nas regras que regem linguagens de programação (como C, C++, Java) ou estruturas de dados (SQL, XML, CSV).

Programas capazes de reconhecer textos em uma determinada linguagem são chamados de *parser* na literatura. Evidentemente, esses programas são específicos para cada linguagem, pois dependem do conhecimento das regras que regem as construções sintáxicas (i.e. devem ser capazes de verificar a validade de cada sentença). Em linguagem natural, isto significa compreender que 'Eu tenho escrito o _____' é uma frase em primeira pessoa, escrita no tempo verbal pretérito perfeito, que precisa ser completada com um substantivo (documento) para ser válida frente as regras da Língua Portuguesa, e que analogamente, o trecho 'int i = 10 _' escrito em um arquivo C é uma a atribuição de variável e precisa ser completada com um elemento ';' para ser válida pela definição da linguagem.

Em linguagem natural, a identificação dessas construções feitas pelo *parser* imita um processo similar ao que utilizamos ao ler frases em um livro. Por estarmos tão acostumados a essa atividade, pode ser difícil perceber que ela depende de uma etapa anterior, a da identificação dos caracteres que compõem cada palavra. Essa etapa é denominada de *lexer* ou *tokenizer*, e computacionalmente corresponde a identiifcar, a partir de uma *stream* de caracteres, quais são as 'palavras' (ou Tokens) válidos para aquela linguagem.

Em linguagem natural, seria possível identificar agrupamentos de caracteres como Palavras, Espaços, Vírgulas e outros sinais de pontuação na análise de um *lexer*, enquanto o trabalho do *parser* corresponderia à identificação das funções de cada um deles nas frases, em Sujeito, Verbo, Adjetivo, por exemplo, além da categorização da própria construção verbal (em 'Oração Subordinada Substantiva Subjetiva', por exemplo).

Na Figura 14 é possível verificar o fluxo típico de uma aplicação de análise sintática. O arquivo é interpretado pelo *lexer* caractere a caractere criando uma *stream* de Tokens, que

identificam conjuntos léxicos válidos para a linguagem. No exemplo, ‘sp’ corresponde a um `ID`, ’=’ a um `Igual`, ‘100’ está em conformidade com um `Inteiro` e ‘;’ é identificado como `EndOfLine`.

Uma vez identificadas pelo *lexer*, as sequências de *tokens* são então categorizadas pelo *parser* através de regras específicas da linguagem. Na Figura 15, podemos verificar que existe uma regra ‘assign’ na definição da linguagem que corresponde à sequência de *tokens*: `Id -> Igual -> Expr -> EndOfLine`. Note que enquanto `Id`, `Igual` e `EndOfLine` são *tokens*, `Expr` corresponde a uma nova regra da linguagem, nesse caso, pode ser composta por um *token* `Inteiro`, ou por outras sequências de *tokens* não presentes no exemplo.

Dessa maneira, o resultado da etapa de análise do *parser* é uma *Abstract Syntax Tree* (AST), uma estrutura em árvore que identifica as expressões encontradas no arquivo de entrada, em diferentes níveis, onde cada nó pode ser uma nova regra da linguagem e cada folha é obrigatoriamente um único *token*.

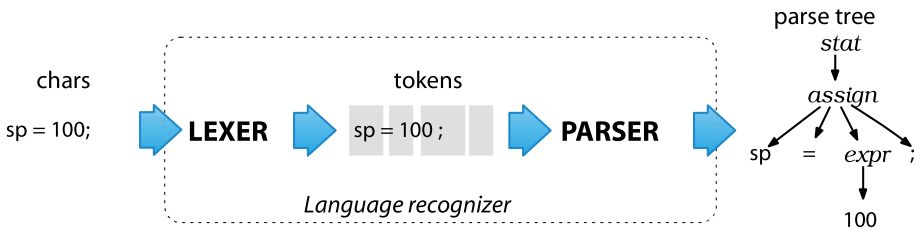

Figura 14 – Exemplo de fluxo de uma ferramenta de análise sintática

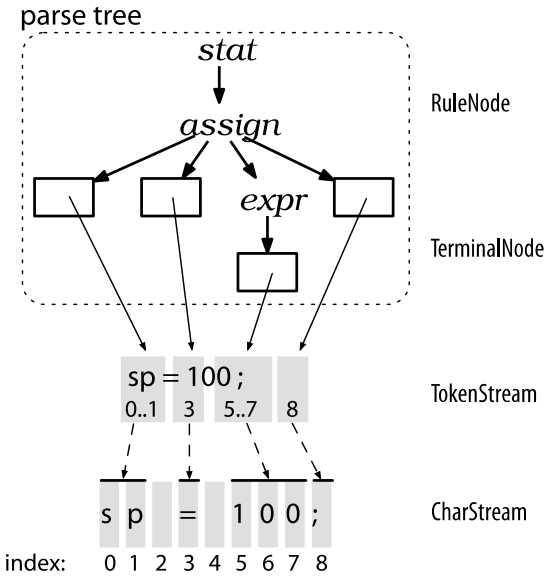

Figura 15 – Exemplo de fluxo de uma ferramenta de análise sintática

O ANTLR é, portanto, uma ferramenta para a construção dessas ferramentas de re-

conhecimento de linguagens, que proporciona diversas facilidades e geração automática de transformações. Cabe ao usuário construir gramáticas de *lexer* e *parser*, definindo regras de tokenização e de construção de nós da árvore de arquivos.

As gramáticas são definidas em arquivos com extensão **.g4** e podem ser compiladas para gerar automaticamente classes que executam a operação de leitura e identificação de arquivos compatíveis com a gramática. A própria estrutura do ANTLR também gera funções de entrada e saída de cada nó da árvore, na forma de *Listener*s. O usuário pode sobrescrever essas funções e identificar o contexto, realizar operações ou verificar a concordância com padrões pré-estabelecidos. Tais funções podem ser adaptadas para a criação de transformações Text to Model (T2M) específicas para a linguagem definida.

No escopo da leitura de arquivos de pré-configuração do MPA, as funções de entrada e saída de nós podem ser utilizadas para instanciar objetos lógicos do metamodelo a partir das informações contidas em cada nó, tudo de forma automática a partir da definição das gramáticas e das funções do *listener*. Mais detalhes da implementação serão fornecidos no Capítulo 5.

Neste capítulo foi traçado um panorama geral das ferramentas utilizadas na criação de transformações (M2T - T2M) e sobre a estrutura de dados a ser utilizada na implementação do gerador de código proposto. No capítulo a seguir, procura-se aprofundar nos detalhes da proposta de *software*, nos casos de uso da aplicação e no modelo de dados.

4 Gerador de Código Proposto

O presente capítulo é voltado para se definir em maiores detalhes a aplicação a ser disponibilizada ao usuário final, bem como o elemento principal do trabalho, o gerador de código para definição de classes de equipamentos. Para tanto, apresenta-se o fluxo de trabalho proposto para a implementação do gerador de código, observando-se as especificidades de cada uma das etapas intermediárias e das tecnologias utilizadas (apresentadas no capítulo anterior). Também estão presentes uma modelagem dos casos de uso da aplicação final e a estrutura de dados simplificada com base no metamodelo.

4.1 Fluxo de desenvolvimento

Como elencado no capítulo anterior, a utilização de ferramentas de *software* para a criação das transformações M2T e T2M acaba criando uma certa dependência da utilização da linguagem Java no projeto. Com a necessidade de implementação de outros elementos do programa (as classes do metamodelo e a *Graphical User Interface*), dois cenários de integração foram estudados, um mantendo todo o desenvolvimento em uma *stack* única em Java, e o outro utilizando técnicas como Invocação Remota de Procedimentos para comunicação entre vários micro-serviços. Para simplificar o desenvolvimento, bem como auxiliar na manutenibilidade futura do sistema, optou-se pela implementação dos artefatos restantes utilizando Java, em projetos da IDE (*Integrated Development Environment*) Eclipse.

O fluxo de trabalho para o desenvolvimento da aplicação final foi pensado de maneira modular, de forma a solucionar cada um dos problemas de maneira independente, permitir testes unitários para cada uma das transformações e posteriormente, utilizar os resultados de cada etapa em uma aplicação única e modular.

Cada um dos quatro pilares da aplicação (classes do metamodelo, transformação M2T, transformação T2M e GUI) deve ser desenvolvido em um projeto Java diferente, seguindo a ordem apresentada. O resultado de cada etapa busca apresentar classes funcionais, que possam ser testadas localmente e integradas de maneira uniforme na aplicação final.

De maneira geral, tanto a estrutura de classes quanto a estrutura das transformações é definida pela utilização das ferramentas descritas no Capítulo 3. Não caberia no desenvolvimento deste projeto, portanto, explorar novos métodos de geração de código a partir de modelos ou de analisadores sintáticos. No entanto, é preciso definir com maior precisão as funcionalidades da GUI apresentada ao usuário.

Na Figura 16 apresenta-se um diagrama de casos de uso que contém as funcionalidades que o gerador de código aqui proposto deve satisfazer.

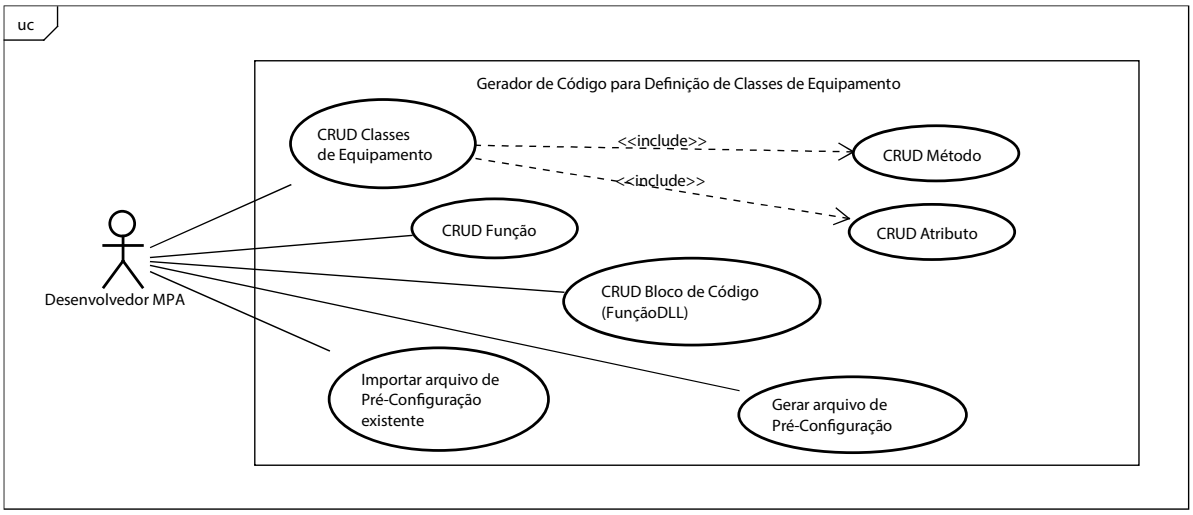

Figura 16 – Diagrama de Casos de Uso do Gerador

O desenvolvedor de uma aplicação no MPA deve ser capaz de adicionar, editar, visualizar e remover (*Create, Read, Update, Delete* - CRUD) Classes de Equipamentos na aplicação. O gerenciamento das Classes de Equipamentos também envolve, como casos de uso derivados, o gerenciamento de Atributos e Métodos associados ao equipamento. O mesmo processo deve ser possível para a definição de Funções avulsas e Blocos de Código (focando nas estruturas que fornecem interfaces entre as funções do MPA e da DLL de controle, denominadas FunçãoDLL). Também em termos de funcionalidades desejadas, o usuário deve ter a disposição a capacidade de importar arquivos de Pré-Configuração para visualizar em *runtime* seu conteúdo, bem como gerar arquivos de Pré-Configuração a partir do contexto corrente dos elementos presentes no Gerador.

Nota-se que a estrutura dos dados do arquivo de pré-configuração permeia todas as etapas da utilização do gerador. Dessa forma, a primeira etapa do presente consiste em implementar as classes do metamodelo que armazenarão os dados de execução do Gerador. Posteriormente, deve ser criado um projeto JET importando essas classes para a geração de transformações M2T para cada elemento com Caso de Uso CRUD (Classe de Equipamento, Função e Bloco de Código). Em seguida é necessário implementar gramáticas utilizando o ANTLR para a geração de transformações T2M, responsáveis por fazer a leitura de arquivos **.mpa** para objetos lógicos. Cada uma das transformações deve ser validada a partir de testes individuais de leitura/escrita de diferentes arquivos de pré-configuração já existentes. Por fim, um projeto final em Java deve ser criado para implementar a interface gráfica do Gerador de Código, integrando a estrutura de dados e as duas transformações implementadas.

4.2 Modelo de dados

Tomando como base a estrutura do metamodelo utilizado, procedeu-se com a implementação das classes que armazenam os dados dos arquivos de Pré-Configuração, utilizando a linguagem Java em um projeto do Eclipse para manter a consistência dentre a *stack* do Gerador de código.

Para cada Caso de Uso CRUD do Gerador foi criada uma classe específica, mantendo os relacionamentos presentes no metamodelo (Equipamento ↔ Método; Equipamento ↔ Atributo, Método ↔ Variável, Função ↔ Método).

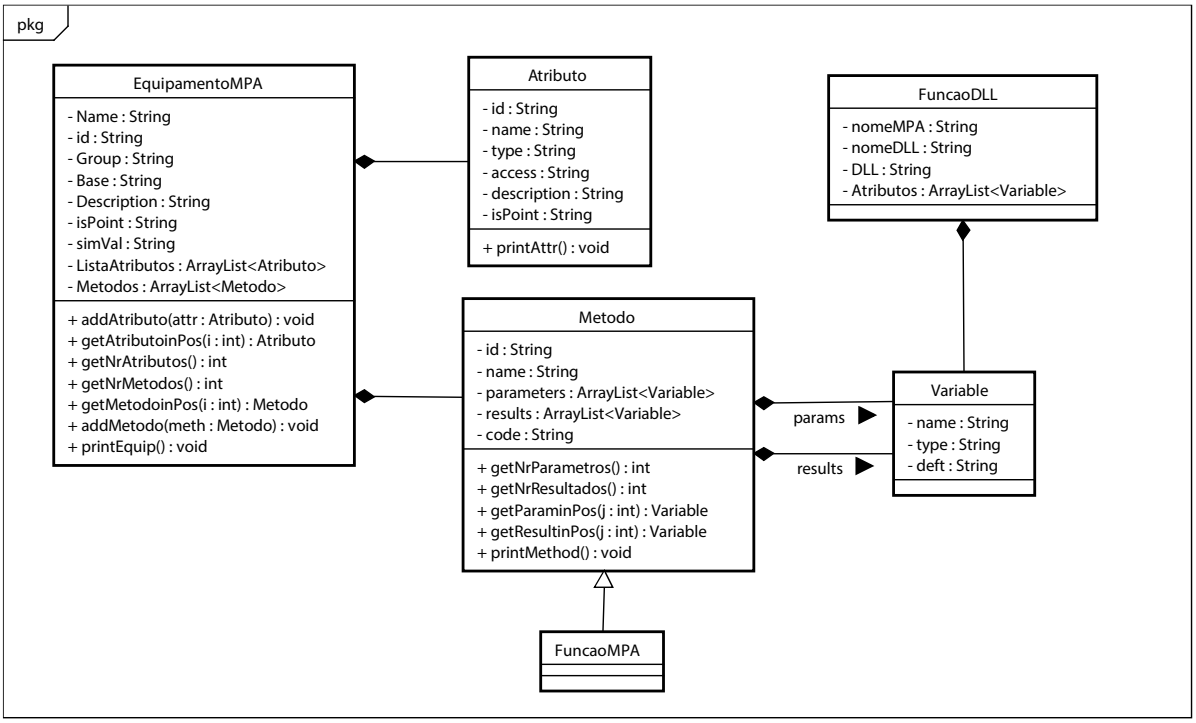

Figura 17 – Diagrama de Classes do Gerador

A Figura 17 mostra o modelo de dados proposto. Como o objeto de trabalho principal é texto, grande parte dos atributos é modelada como **String**. As classes modeladas e suas funções principais são descritas abaixo:

- `EquipamentoMPA.java` - Armazena informações gerais sobre um Equipamento.
- `Atributo.java` - Define uma característica (física ou lógica) de um Equipamento, que fica disponível na aba de instanciação da planta no MPA. Atributos podem ser mensuráveis e estar conectados com a planta via OPC.
- `Metodo.java` - Define uma função do Equipamento, que fica disponível na aba de construção de diagramas de execução no MPA.

- `FuncaoMPA.java` - Define uma função, dissociada de equipamentos, que fica disponível na aba de construção de diagramas de execução no MPA.
- `Variable.java` - Armazena informações sobre parâmetros/retornos de funções (`Metodo` ou `FuncaoDLL`).
- `FuncaoDLL.java` - Modela uma interface entre uma função da DLL e uma função do MPA, presentes dentro de um bloco de código.

Os atributos de todas as classes possuem *getters* e *setters* para manipulação de objetos já instanciados, que foram omitidos para facilitar a visualização do diagrama. Além disso, funções específicas foram criadas para manipulação de listas: adicionar objeto na lista; obter objeto em posição n da lista; obter número de atributos da lista.

As funções de manipulação de lista estão presentes na classe `EquipamentoMPA`, para lidar com instâncias de `Atributo` e `Metodo` associadas ao equipamento. De maneira análoga, na classe `Metodo`, há funções de manipulação de lista para as instâncias de `Variable` (que representam parâmetros e resultados do método). Para a `FuncaoDLL`, o arquivo de Pré-Configuração exige que sejam definidos os tipos das variáveis na definição de uma interface entre função do MPA e da DLL, sem se importar com seu nome. Assim, a estrutura `Variable` foi reaproveitada para armazenar esses dados em uma lista de atributos associada à `FuncaoDLL`

Funções de impressão também foram implementadas nas classes `Metodo`, `EquipamentoMPA` e `Atributo`, para facilitar a visualização dos objetos em *runtime* durante a execução de testes.

A modelagem conceitual aqui apresentada serve como base para a criação das transformações, que serão apresentadas em maiores detalhes no próximo capítulo. Serão explorados em maior profundidade os *templates* necessários para a geração de transformações JET e as gramáticas que dão origem a *parsers* utilizando ANTLR. Posteriormente, será abordada a integração dos modelos e das transformações em uma aplicação gráfica disponível para o usuário final.

5 Implementação das Transformações

Ao longo do presente trabalho já foram descritos o funcionamento do MPA, os problemas na etapa de pré-configuração que se deseja solucionar com a criação do Gerador de Códigos, a necessidade de criação de transformações para leitura e escrita do arquivo **.mpa** e as ferramentas capazes de simplificar a sua criação. Para finalizar, no presente capítulo apresenta-se o processo de desenvolvimento das transformações de código, que funcionam como motor de inteligência para a aplicação final.

Neste capítulo também são elencados os artefatos de *software* efetivamente criados no escopo deste trabalho, os detalhes de sua implementação e integração, além dos resultados obtidos para cada uma das etapas propostas, considerando as especificidades do arquivo de Pré-Configuração do MPA.

Como o modelo de classes da aplicação já foi apresentado na seção 4.2, é possível considerar que o leitor esteja familiarizado com sua estrutura. A implementação das classes `.java` do modelo proposto foi feita em um projeto no Eclipse. Algumas instâncias dos objetos foram criadas com base em um arquivo de pré-configuração já existente e utilizado no projeto, em uma função *main* criada para testes. O conteúdo dos objetos foi, então, inspecionado através das funções *print* criadas e comparado com o **.mpa**.

De maneira semelhante, de acordo com o fluxo de desenvolvimento apresentado, projetos específicos seriam criados para cada uma das transformações e da interface gráfica do gerador final, para manter a intercambialidade entre os elementos. Uma vez satisfatoriamente validado o modelo, o conjunto de classes criadas pode ser importado manualmente a cada novo projeto criado e usado como modelo lógico da aplicação final. A seguir será relatado o desenvolvimento de cada uma das transformações.

5.1 Transformações M2T

As transformações M2T (*Model to Text*), como já mencionadas em capítulos anteriores, têm como objetivo gerar conteúdo textual a partir de instâncias de objetos do modelo conceitual da aplicação. Na prática, no contexto do gerador de pré-configurações para o MPA, isto significa compreender um `EquipamentoMPA` e escrevê-lo dentro da estrutura `class{ }` (descrita na seção 2.2.1), populando textualmente todos os elementos associados ao objeto lógico, como `Atributos` e `Metodos`. O resultado dessas transformações deverá ser armazenado em arquivos **.mpa**, dando origem a uma pré-configuração garantidamente compatível com o *software* MPA. Note que os objetos `EquipamentoMPA` são modelos de classes de equipamentos para o MPA.

Para auxiliar na definição das transformações M2T utilizou-se a ferramenta *Java Emitter*

Templates (JET). Como apresentado no Capítulo 3, a ferramenta JET depende da definição de *templates* que incorporam a estrutura do texto a ser criado, de maneira a simplificar a formatação e facilitar o preenchimento das lacunas com as informações do modelo de dados.

Para habilitar a criação de projetos JET, é necessário instalar o pacote EMF no Eclipse. Foi utilizada a distribuição *Eclipse Modeling Tools - Version Neon 4.6.3* para a criação do projeto. Após a criação de um projeto simples em Java, é possível convertê-lo para um projeto JET. Essa configuração trata de criar uma pasta específica para a criação dos *templates* e da compilação do projeto utilizando o *JETBuilder* sempre que algum arquivo é modificado.

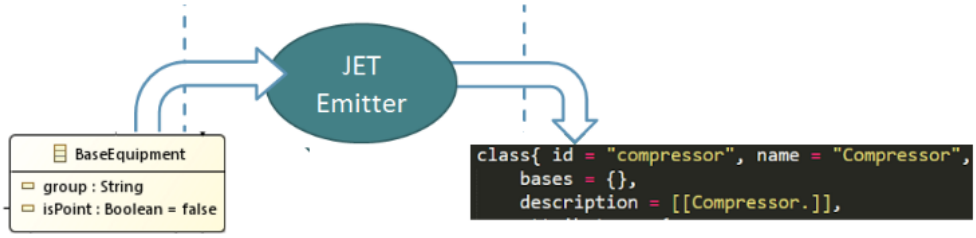

Figura 18 – Transformação Model To Text (M2T)

Cada um dos elementos principais identificados no arquivo de Pré-Configuração (Equipamento, Função e Bloco de Código) foi modelado em um *template* correspondente. Como o arquivo destino da Pré-Configuração possui extensão **.mpa**, todos os *templates* utilizam o sufixo **.mpajet**. Cada *template* é compilado automaticamente dando origem a uma classe de transformação correspondente, listadas a seguir:

- `equipamento.mpajet`
  - `GenEquipamentoMPA.java`
- `funcao.mpajet`
  - `GenFuncaoMPA.java`
- `funcaodll.mpajet`
  - `GenFuncaoDLL.java`

Cada classe gerada possui uma função **.generate()**, que recebe como argumento um objeto `EquipamentoMPA`, `FuncaoMPA` ou `FuncaoDLL`, respectivamente. O projeto JET depende, portanto, das classes do modelo criadas anteriormente.

Como a estrutura de cada *template* modifica a geração da **.generate()** na classe de implementação, definiu-se uma série de verificações a serem feitas no argumento recebido pela função. No caso dos equipamentos, por exemplo, atributos como ‘group’, ‘isPoint’, ‘base’ e ‘simVal’ são opcionais, e não devem ser impressos no texto gerado caso constem com valores nulos no objeto `EquipamentoMPA` recebido como parâmetro. Dessa maneira, garantimos a

consistência da estrutura do texto gerado e, posteriormente, a compatibilidade do arquivo de Pré-Configuração completo com o MPA.

A estrutura dos *templates* é similar para cada uma das transformações. Na Figura 19, podemos ver uma parte do *template* para a tradução de um `EquipamentoMPA`. Todo texto simples no arquivo de *template* é impresso ao executar a transformação, enquanto expressões definidas entre '**<% %>**' ou '**<%= %>**' são interpretadas/interpretadas e impressas, respectivamente, na classe que implementa as transformações (como visto na seção 3.2).

Depois de identificada a estrutura fixa a partir de um arquivo de pré-configuração existente, é necessário completar as lacunas a partir dos objetos lógicos do modelo recebidos como argumento e também criar verificações para adequar o texto quando não houver dados ou definir iteradores para varredura de listas.

```
class{ id = "termopares", name = "Termopares",
    bases = {},
    description = [[Equipamento para agrupar um conjunto de termopares.]],
    attributes = {
        {   id = "temperaturas", name = "Temperaturas", type = "REAL_POINT[]",
            access = "",
            description = [[]],
        },
    },
...
```

Exemplo preconfig.mpa

```
class{ id =             , name =             ,
    bases = {},
    description = [[                                                    ]],
    attributes = {
        {   id =               , name =               , type =               ,
            access = "",
            description = [[]],
        },
    },
...
```

Estrutura fixa do .mpa

```
class{ id = "<%= equip.getId() %>", name = "<%= equip.getName() %>", <%
if(equip.getGroup()==null){}else if(!equip.getGroup().equals("")){%> group = "<%=
equip.getGroup() %>", <% } %>
<% if(equip.getIsPoint()!= null){ if(equip.getIsPoint().equals("true")){ %> isPoint
= true, <% }} %>
    bases = {<% if(equip.getBase()==null){}else{
    if(!equip.getBase().equals("")){%>"<%= equip.getBase() %>"<%}} %>},
    <% if(equip.getSimVal()==null){} else if(!equip.getSimVal().equals("")){ %>
    simulationValue = "<%= equip.getSimVal() %>", <% } %>
    description = [[<%= equip.getDescription() %>]],
    attributes = {
<% for ( int i = 0; i<equip.getNrAtributos(); i++){ %>
<% attr=equip.getAtributoinPos(i); %>
        {   id = "<%= attr.getId() %>", name = "<%= attr.getName() %>", type = "<%=
        attr.getType() %>",
            access = "<%= attr.getAccess() %>",
            description = [[<%= attr.getDescription() %>]],
        },
<% } %>
    },
...
```

Template JET

Figura 19 – Estrutura do *template* JET para um EquipamentoMPA

É importante notar que cada transformação gera o texto equivalente a um objeto único (EquipamentoMPA, FuncaoMPA ou FuncaoDLL). O arquivo **.mpa** a ser criado pelo Gerador deve ser obtido a partir da composição de diversas chamadas da função **.generate()** de cada tipo de transformação, de acordo com os objetos instanciados em *runtime* no Gerador. Dessa maneira, é possível manter no contexto da aplicação listas separadas de Equipamentos e Funções, e não uma única lista com todos os tipos de objetos, o que reduz significativamente a complexidade tanto dos *templates* JET quanto da implementação da interface gráfica.

As três classes de transformação geradas foram testadas individualmente, com base em arquivos **.mpa** já existentes. Objetos lógicos `EquipamentoMPA`, `FuncaoMPA` e `FuncaoDLL` (e outros objetos `Atributo` e `Metodo` associados) foram instanciados em uma nova classe `main` no projeto JET, contendo as informações disponíveis na pré-configuração. Para cada estrutura, foi executada a função **.generate()** da classe correspondente e a saída, obtida no console, foi comparada com o arquivo original. Os *templates* foram, então, ajustados para manter indentações similares às utilizadas pela equipe de projeto, para manter a legibilidade do código em casos que o gerador eventualmente não seja utilizado.

Uma vez finalizada a criação de todos os *templates* e obtidas as classes de implementação `GenEquipamentoMPA.java`, `GenFuncaoMPA.java` e `GenFuncaoDLL.java` através da compilação do *JETBuilder*, é possível executar as transformações em projetos em projetos simples do Java (independentes do JET). Assim como o modelo de dados, as classes de implementação se tornam artefatos de *software* a serem integrados com a aplicação final.

5.2 Transformações T2M

De forma espelhada, as transformações *Text to Model* (T2M) devem ser capazes de analisar um arquivo de texto e extrair, de dentro de sua estrutura, informações relevantes para a instanciação de objetos lógicos de um modelo de dados conceitual. Dentro do escopo deste trabalho, isso significa identificar os elementos principais que compõem um arquivo de pré-configuração **.mpa**, descritos na seção 2.2, obter seus dados e popular adequadamente um objeto do modelo com essas informações.

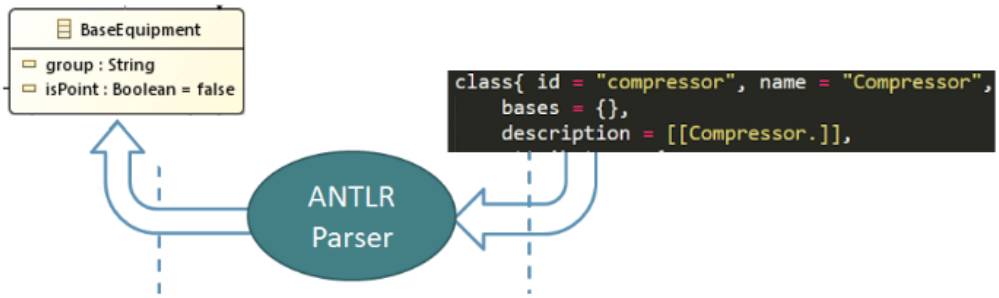

Figura 20 – Transformação Text To Model (T2M)

Para simplificar a criação dessas transformações utilizou-se a ferramenta ANTLR, apresentada no Capítulo 3. As transformações e arquivos foram compilados utilizando os comandos no console, armazenados em um diretório separado, onde também foram importados os arquivos do modelo de dados da aplicação.

Em cenários de operação real, é comum que a pré-configuração utilizada nos projetos do MPA seja disponibilizada em um único arquivo, que pode conter tanto Classes de Equipamentos, Funções ou Blocos de Código de maneira combinada e não ordenada. Uma transformação de leitura e análise do arquivo, portanto, deve ser capaz de identificar e separar cada um desses elementos sintaticamente. Já foi feita uma análise das estruturas de cada um desses componentes no Capítulo 2, sendo assim, as gramáticas a serem definidas utilizando o ANTLR precisam incorporar esse conhecimento.

Seguindo o fluxo de desenvolvimento de transformações com o ANTLR, devem ser definidas gramáticas de *lexer* e *parser*, que descrevam adequadamente a linguagem a ser compreendida pela transformação. A gramática de ***Lexer*** é responsável por agrupar os caracteres em ***Tokens*** identificáveis, enquanto a gramática de ***Parser*** identifica as regras da linguagem com base na entrada tokenizada e monta uma estrutura em árvore AST com as informações presentes no arquivo.

Cada uma das gramáticas foi implementada em um arquivo **.g4** separado. A Figura 21 mostra um trecho da gramática criada no arquivo `MPAFileLexer.g4`. São definidos como *tokens* algumas palavras chaves presentes na estrutura do arquivo **.mpa** (como '**id**', '**name**', '**description**'), caracteres de interesse como '**=**', '**{**' e '**}**', além de elementos compostos, como o QTEXT na linha 9, que compreende qualquer texto entre aspas ou entre elementos '**[[ ]]**'.

A partir da gramática (e tendo uma referência ao **.jar** da ferramenta ANTLR como variável de ambiente) uma classe de implementação **.java** é gerada ao utilizar o comando 'antlr4'. A classe de implementação possui funções internas para corresponder sequências de caracteres do arquivo a regras de textittokens definidos na gramática do *lexer*. A compilação da gramática do *lexer* também gera um arquivo **.tokens**, que lista e atribui ids a todos os *tokens* definidos.

- `MPAFileLexer.g4`
  - `MPAFileLexer.java` - Gerado
  - `MPAFileLexer.tokens` - Gerado

A ordem de aparição das regras no arquivo **.g4** define a prioridade de atribuição de uma sequência de caracteres ao *token*, para o caso de uma eventual sobreposição de regras. O *lexer* também consegue identificar contextos, de maneira rudimentar, através da utilização de '**mode**'s. O **mode** define um conjunto de regras de tokenização que se aplica apenas dentro daquele contexto. A atribuição a um *token* específico pode disparar uma mudança de modo (linha 2, 3 e 23). Isso tem especial relevância pois o tratamento de certas palavras chaves (como '**code**') é

diferenciado quando dentro de uma Função ou de um Bloco de Código. Enquanto o código da Função é mantido inteiro, com comentários e tabulações, o código de um Bloco de Código deve ser inspecionado, e dentro dele, identificadas as interfaces com a DLL (`FuncaoDLL`).

```
1   lexer grammar MPAFileLexer;
2   ALTFNC       : 'func{'  -> pushMode(INSIDE);
3   ALTCLASS     : 'class{' -> pushMode(INSIDE);
4
5   •••
6   mode INSIDE;
7   CLASS        : 'class{';
8   FNC          : 'func{';
9   QTEXT        : (('"'  .*? '"') | (OPKY OPKY .*? CLKY CLKY));
10
11  STRMETH      : 'methods';
12  STRDEFAULT   : 'default';
13  STRBASE      : 'bases';
14  STRID        : 'id';
15  STRNAME      : 'name';
16  STRDESC      : 'description';
17  STRPRM       : 'parameters';                          Palavras Chave
18
19  OPKY         : '[';
20  CLKY         : ']';
21  OPCOL        : '{';
22  CLOSEANDJUMP: '}' ~(',' | '\n')+ [\n];
23  CLCOL        : '}'  ->popMode;
24  OPPAR        : '(';
25  CLPAR        : ')';
26  CMA          : ',';                                   Caracteres Especiais
27
28  FLOAT        : DIGIT+ '.' DIGIT* | '.' DIGIT+;
29      fragment
30      DIGIT    : [0-9];
31  INT          : [0-9]+;
32  STRBOOL      : ('true' | 'false');                    Estruturas Derivadas
33
34  •••
```

Figura 21 – Exemplo da estrutura da gramática do *Lexer*

De forma semelhante, a gramática do *Parser* interpreta a estrutura do arquivo em alto nível. É possível ver na Figura 22 um trecho da definição da gramática no arquivo `MPAFileParser.g4`. Já é possível verificar uma similaridade com a estrutura de árvore: `doc` (linha 4) é uma regra-raiz da gramática e pode ser composta por 4 elementos: Função MPA; Bloco de Código; Classe de Equipamento ou Includes de outros arquivos. Cada um dos elementos é modelado através de uma regra (`mpafunction`, `dllblock`, `equipment` e `include`), que por sua vez é composta por outras regras, recursivamente, até chegar em *tokens* definidos no vocábulo gerado pelo *Lexer*.

```
1   parser grammar MPAFileParser;
2
3   options { tokenVocab=MPAFileLexer; }
4   doc : ( mpafunction | dllblock | equipment  | include )+ ;   Macro-Elementos
5
6   equipment    : (CLASS | ALTCLASS) expr+  (ALTCLCOL| CLCOL) ;
7   expr         : id
8                | name
9                | description
10               | parameters
11               | results
12               | code
13               | base
14               | group                                  Componentes de Equipamento
15  •••
16
17  name         : STRNAME EQ QTEXT (CMA)*;
18  id           : STRID EQ QTEXT CMA;
19  group        : STRGROUP EQ QTEXT CMA;
20  base         : STRBASE EQ OPCOL (QTEXT)* CLCOL (CMA | ALTCMA);
21  description  : STRDESC EQ QTEXT  CMA;
22  isPoint      : STRISPOINT EQ (QTEXT | STRBOOL) CMA;
23  code         : (ALTSTRCODE | STRCODE) (EQ | ALTEQ) (CODESNP | QTEXT CMA);
    •••                                  Regras terminais - Mapeiam para sequências de Tokens
```

Figura 22 – Exemplo da estrutura da gramática do *Parser*

A gramática do *parser* foi compilada com o comando 'antlr4', da mesma maneira que a gramática do *lexer*, e deu origem a uma outra classe de implementação `MPAFileParser.java` e uma classe auxiliar `MPAFileParserBaseListener.java`. A classe de implementação do *parser* recebe como *input* um arquivo devidamente tokenizado pela etapa anterior, e gera uma árvore AST de acordo com as regras definidas na gramática. Cada uma das regras (`doc`, `expr`, `name`, `id`, ainda no exemplo da Figura 22) se transforma em um possível nó da árvore AST ao final da transformação de leitura do arquivo.

É possível, então, navegar através da árvore completa, executando funções a cada nó ou folha de interesse. Na classe `MPAFileParserBaseListener.java`, são declarados automaticamente cabeçalhos dessas funções de entrada e saída, para cada regra presente na linguagem. Assim, definimos uma nova classe *identifiers.java* que sobrescreve as funções geradas no `BaseListener`, com o objetivo de extrair as informações sobre equipamentos e funções da pré-configuração.

Já no arquivo `identifiers.java` são instanciados os objetos do modelo de dados da aplicação a partir das informações contidas nas folhas da árvore, i.e., o conteúdo de cada macro-elemento do arquivo .mpa. Dessa forma, ao chegar no final da leitura, é possível obter listas contendo todos os objetos `EquipamentoMPA`, `FuncaoMPA` e `FuncaoDLL` presentes no arquivo.

A validação das transformações de leitura deve ser realizada em duas etapas. Inicialmente, é preciso verificar se as gramáticas criadas estão adequadas e conseguem identificar os elementos do arquivo **.mpa**. Para isso, é possível utilizar o comando 'grun' do ANTLR para visualizar a

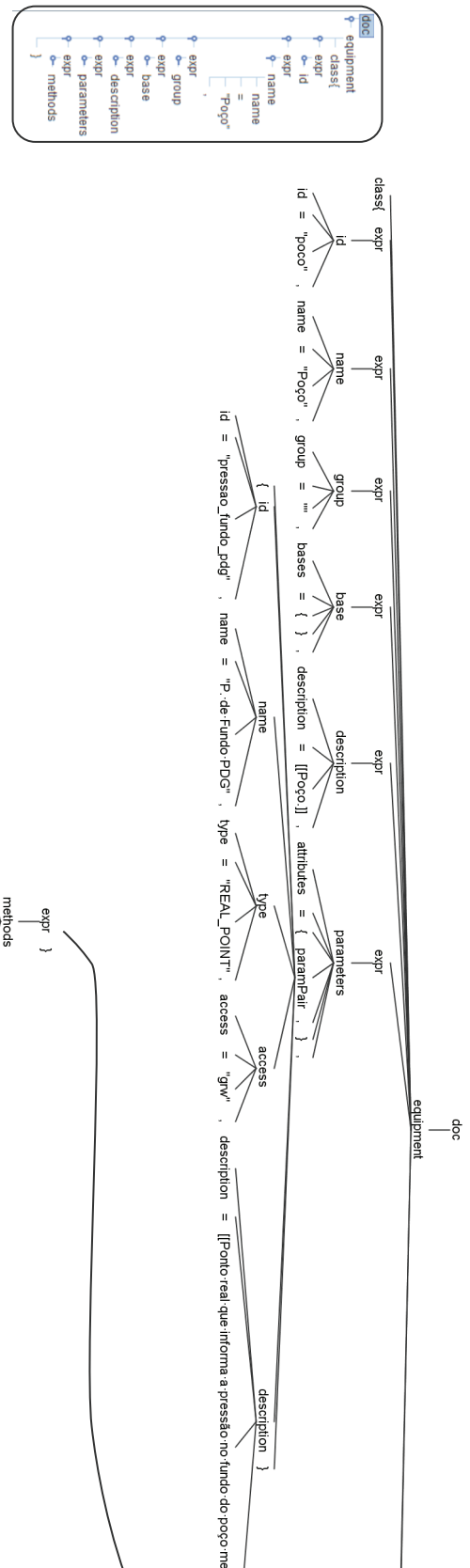

Figura 23 – Árvore de uma classe de equipamento Poço

árvore AST completa gerada pela transformação. A Figura 23 mostra a árvore obtida para uma classe de equipamentos de exemplo, que modela um “Poço”.

Assim como Equipamentos, foram validados individualmente os outros macro-elementos, e posteriormente, foram validados arquivos com múltiplos elementos diferentes, testados em diferentes ordenações. Os ajustes necessários foram realizados nas gramáticas quando a leitura apresentou resultados não satisfatórios. Uma segunda etapa de validações compreende a correta instanciação dos objetos lógicos do modelo a partir das folhas da árvore.

Uma classe `usage.java` foi criada para executar as transformações e ‘caminhar’ pela árvore, chamando as funções de entrada e saída de nós. Ao final, para todas as listas obtidas pela transformação de leitura são chamadas as funções de impressão objeto a objeto, e os resultados apresentados no console são validados com as informações do arquivo **.mpa** original. Eventuais informações faltantes ou inconsistências foram ajustadas nas funções classe `identifiers.java`.

- `MPAFileParser.g4`
  - `MPAFileParser.java` - Gerado
  - `MPAFileParserBaseListener.java` - Gerado
- `identifiers.java`
- `usage.java`

Assim como as transformações geradas através do JET, as classes de implementação geradas a partir do ANTLR funcionam de maneira independente do projeto. Os arquivos com as gramáticas **.g4** não precisam estar presentes na aplicação final para a execução das transformações, mas podem ser mantidos para expansão ou *updates* da estrutura do arquivo, de maneira simples.

Durante esse capítulo foi possível explorar o desenvolvimento das transformações M2T-T2M utilizando como auxílio as ferramentas *Java Emitter Templates* e *ANother Tool for Language Recognition*. A partir de definições de alto nível (*templates* e gramáticas) foram obtidas transformações funcionais e independentes, o que auxilia na manutenibilidade e na facilidade de atualização de cada uma das soluções para incorporar requisitos de versões mais atuais do MPA.

No capítulo seguinte será apresentado o resultado final deste trabalho: o Gerador de Código que integra as transformações, o modelo de dados e fornece uma interface ao usuário, simplificando a etapa de pré-configuração dos projetos no *software*.

6 Resultados e Discussões

Apesar de extremamente importantes para a solução proposta por este trabalho, os processos de transformação apresentados nos capítulos anteriores, sozinhos, têm um baixo impacto no processo de configuração de um projeto no MPA. É integrando esses artefatos de *software* como motores de inteligência em uma aplicação de configuração e gerência de arquivos .mpa que seu valor começa a despontar.

No presente capítulo é feita uma análise da aplicação final do Gerador de Códigos, a interface que disponibiliza para os usuários as funcionalidades já desenvolvidas através de uma interface simples e intuitiva.

6.1 Estrutura geral

Na Seção 4.1, após uma análise da utilização das ferramentas que auxiliariam a criação das transformações, ficou claro que manter a *stack* do projeto com uma única linguagem tornaria a integração dos artefatos de *software* bem mais simples. Dessa maneira, para a implementação da interface, foi criado um novo projeto em Java. Todas as classes do modelo de dados e das transformações de leitura e escrita geradas foram importadas, e o binário (.jar) com a ferramenta ANTLR foi adicionado como dependência do projeto, para habilitar as funções do *parser*.

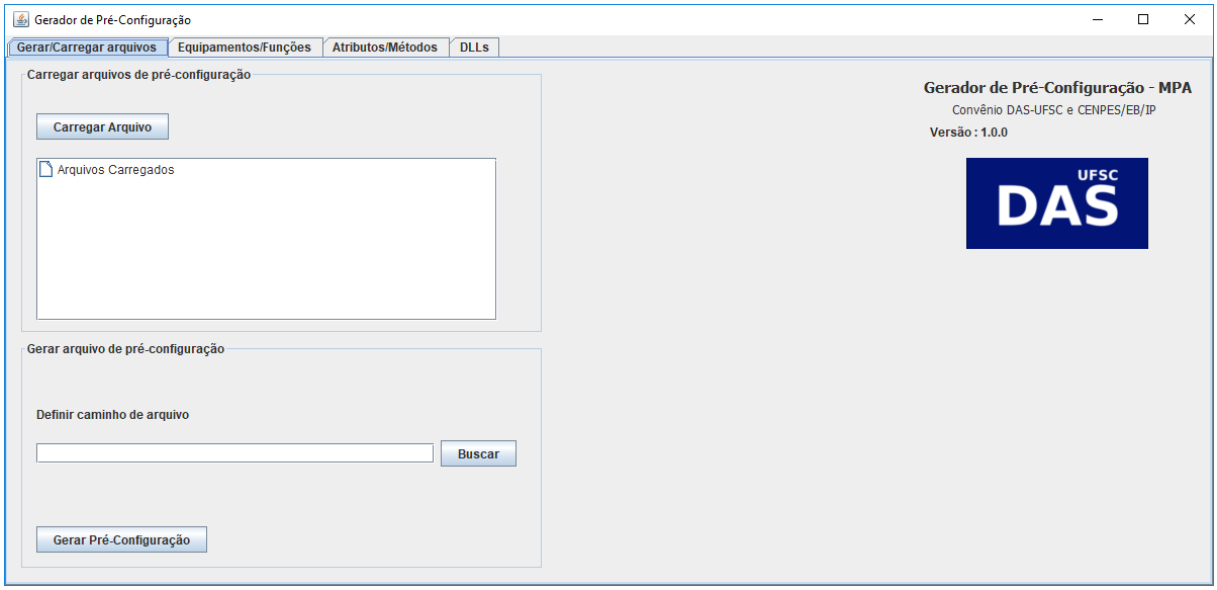

Figura 24 – Tela inicial do Gerador

Para a criação de elementos gráficos foram usados os pacotes `javax.swing` e `java.awt`. A aplicação não necessita de instalação, requer apenas a instalação do *Java Runtime Environment*

(JRE) por conta da aplicação ser baseada na *Java Virtual Machine*. O programa é disponibilizado aos usuários finais como um único arquivo *stand-alone* executável (**.jar**) com menos de 2 MB.

Na Figura 24 é exibida a tela inicial do *software*. A disposição das funcionalidades é feita em abas, separando os elementos principais para facilitar a visualização. Na aba inicial 'Gerar/Carregar arquivos', é possível gerar um arquivo .mpa com os dados da aplicação em um caminho definido pelo usuário, carregar arquivos .mpa já existentes para a aplicação e verificar informações sobre a versão da aplicação. Uma lista com o histórico de arquivos já lidos fica visível ao usuário no painel correspondente.

A leitura dos arquivos dispara o processo de transformação através do *parser* criado, elemento a elemento. Elementos no arquivo que possuam o mesmo nome de objetos já instanciados na aplicação não interrompem a leitura mas são ignorados, gerando um aviso de duplicidade ao usuário (Figura 25). Os objetos lidos ficam disponíveis em cada uma das respectivas abas. Arquivos **.mpa** que corrompidos ou com a estrutura fora dos padrões, i.e., arquivos inválidos para importação no MPA, são identificados pelo gerador e retornam ao usuário uma mensagem de erro (Figura 26).

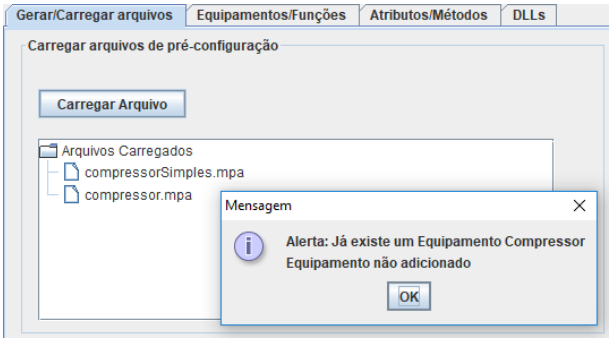

Figura 25 – Alerta de Equipamento duplicado durante leitura de arquivo

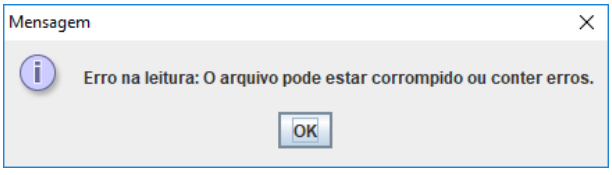

Figura 26 – Mensagem de erro na leitura de arquivo

Informações gerais sobre Funções e Classes de Equipamentos são exibidas na segunda aba, apresentada na Figura 27. Uma árvore à esquerda de cada painel mostra os objetos instanciados na aplicação. Selecionar um objeto na árvore apresenta as informações gerais sobre o respectivo objeto no painel à direita.

Botões embaixo de cada árvore permitem a adição de Equipamentos e Funções, abrindo uma nova janela com as informações gerais permitidas pela estrutura da pré-configuração,

como pode ser visto na Figura 28. Da mesma maneira, é possível editar ou excluir os objetos selecionados na árvore utilizando os botões correspondentes.

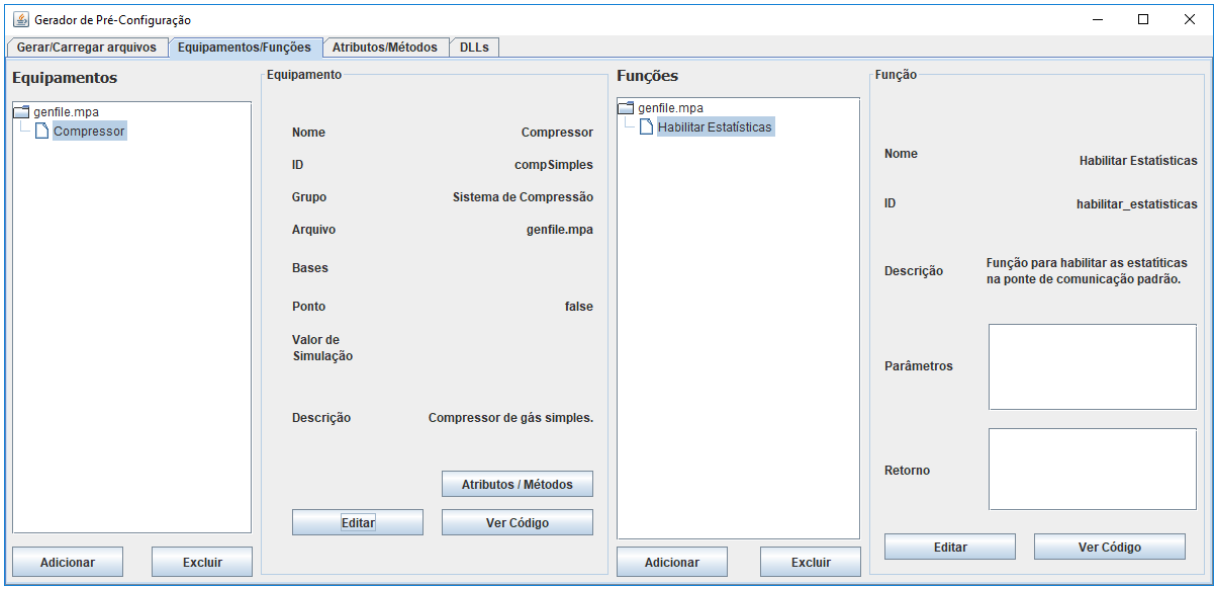

Figura 27 – Aba 'Equipamentos/Funções' do Gerador

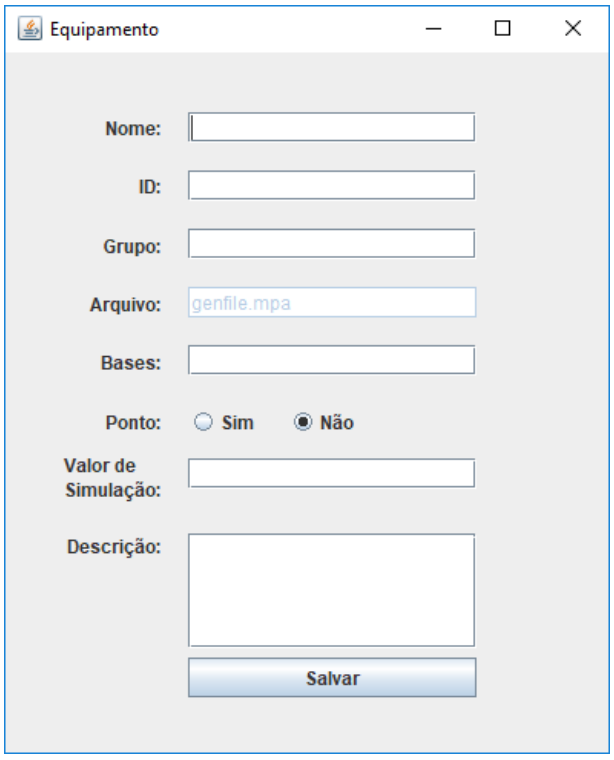

Figura 28 – *Pop-up* de adição de um Equipamento

Como Equipamentos possuem estruturas mais complexas, na Aba Atributos/Métodos são exibidas as configurações complementares para cada Equipamento (ver Figura 29). A seleção de um Equipamento na árvore à esquerda atualiza as listas de atributos e métodos associados a

aquele equipamento, cujas informações podem ser visualizadas no painel à direita de cada lista ao selecionar um objeto.

De maneira similar à definição de Equipamentos e Funções, a criação ou edição de Atributos e Métodos abre novas janelas com todas as opções disponíveis (Figura 30).

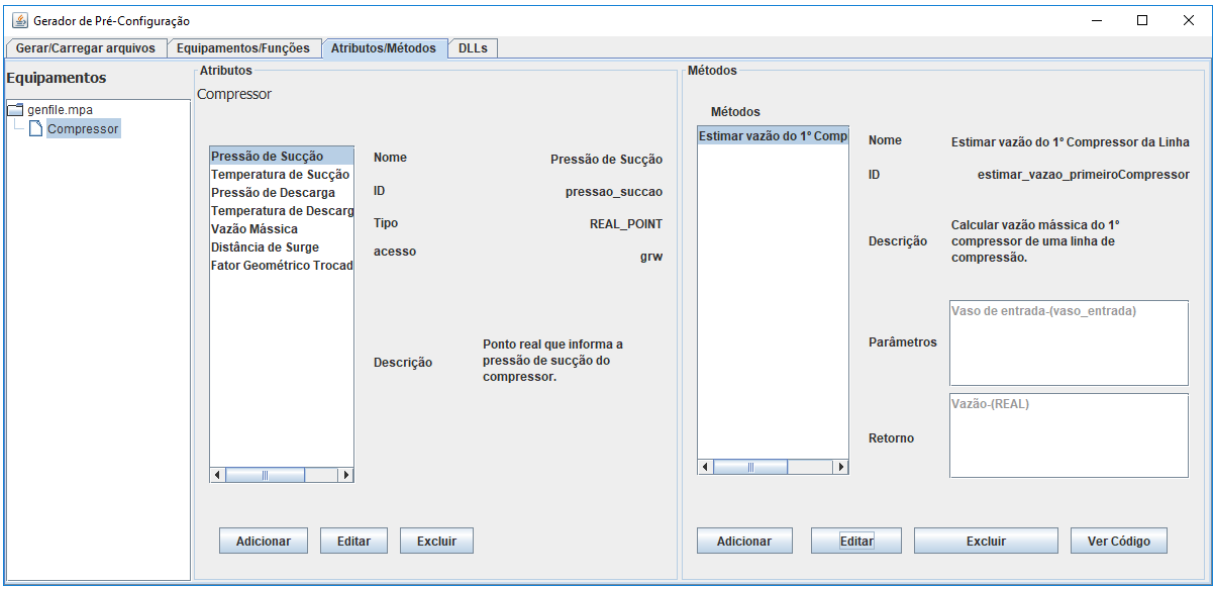

Figura 29 – Aba 'Atributos/Métodos' do Gerador

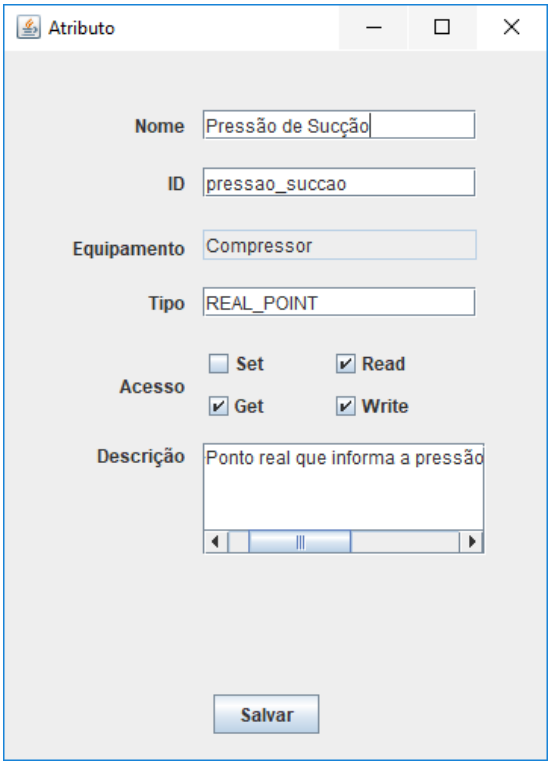

Figura 30 – *Pop-up* de edição de um Atributo

A exclusão de um Equipamento é feita exclusivamente na aba ‘Equipamentos/Funções’, e internamente executa a eliminação em cascata de objetos `Atributo` e `Metodo` associados ao Equipamento excluído. Isso evita problemas no gerenciamento de memória da aplicação, principalmente para arquivos de pré-configuração maiores, ou o aparecimento de objetos ‘fantasmas’ nas listas de Atributos e Métodos do Gerador.

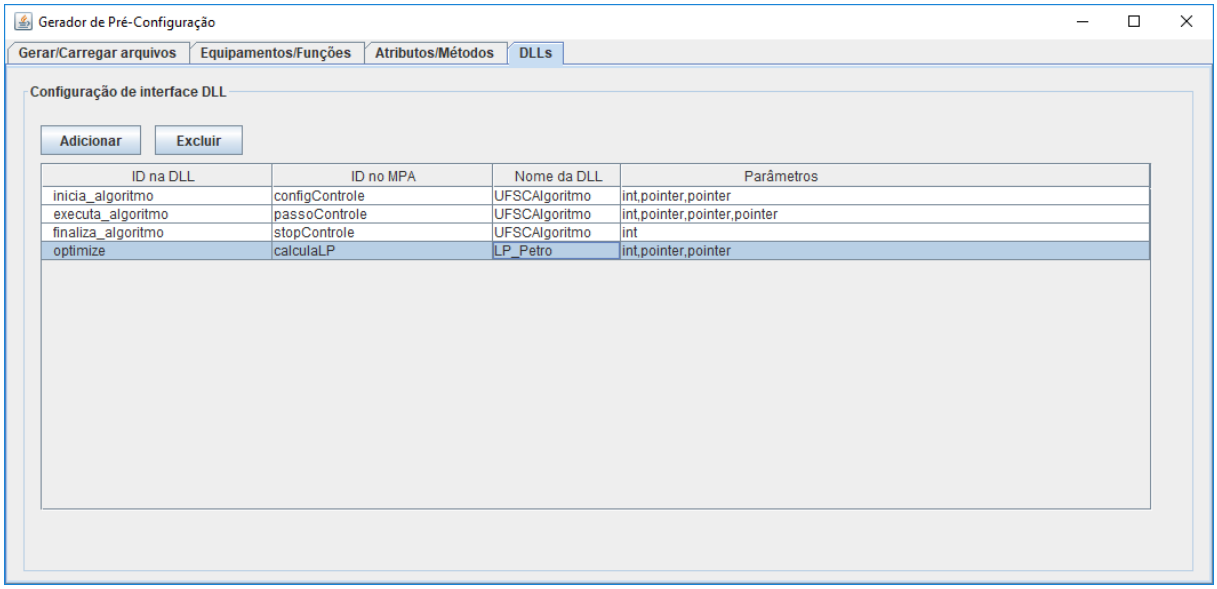

Figura 31 – Aba ‘DLLs’ do Gerador

Por fim, interfaces entre funções do MPA e funções advindas de DLLs também podem ser declaradas na Aba ‘DLLs’ do Gerador, como pode ser visto na Figura 31. Uma estrutura em tabela exibe uma lista com o nome da função desejada na DLL, o nome equivalente (*alias*) disponível para a pré-configuração, além do nome da DLL onde a função está inserida e os tipos dos argumentos declarados no cabeçalho da função.

Todos os objetos declarados, sejam Equipamentos, Atributos e Métodos associados, Funções ou Interfaces DLL, são elencados para exportação no arquivo `genfile.mpa`. A pré-configuração gerada automaticamente, por utilizar a estrutura definida pelos *templates* das transformações descritas anteriormente, tem garantida a compatibilidade com o MPA, e é o primeiro passo para as definições necessárias para os projetos de controle.

6.2 Análise de resultados

Apesar de não modelado como um requisito durante a construção do gerador, as transformações criadas buscam reproduzir com um grau mínimo de alterações arquivos de pré-configuração já existentes. As tabulações, convenções de espaços e vírgulas adotadas pelos arquivos de exemplo foram incorporadas na estrutura das transformações e, portanto, são objetos de análise durante os testes.

Para verificar a completude e a similaridade entre arquivos originais e gerados, foram realizados testes de leitura e escrita em diversos arquivos .mpa criados para o projeto. Um desses testes, apresentado abaixo, utiliza como entrada o arquivo `preconfig.mpa`, que conta com 6 Equipamentos, 29 Atributos e 5 Métodos, totalizando 423 linhas de código. O arquivo foi importado pelo Gerador e nenhuma modificação foi feita através da interface. Um arquivo `genfile.mpa` foi gerado automaticamente através da transformação.

Para realizar uma comparação entre arquivos gerados manualmente e automaticamente, utilizou-se a ferramenta online *DiffNow*, da PrestoSoft LLC. O resultado da comparação feita pode ser visto na Figura 32. É possível observar que o arquivo gerado automaticamente é muito semelhante ao original, com menos de 3% de dissemelhança entre os arquivos. As mudanças mais presentes são em quebras de linha: trechos com múltiplos espaços no arquivo original são colapsados em apenas um no arquivo gerado e elementos da estrutura fixa como `'method={}'` quando não possuem nenhum conteúdo ganharam uma quebra de linha entre as chaves no arquivo gerado.

Outra característica importante é que os comentários e os includes lidos do arquivo original não são impressos e/ou tratados pelo gerador, na versão atual, quando presentes fora da estrutura de `'code={}'` de uma função ou método. No caso apresentado, o arquivo original continha linhas comentadas identificando componentes de partes específicas da planta, que facilitam a identificação dos componentes. Como se propõe a utilizar o Gerador para gerenciamento de toda a etapa de pré-configuração, no novo fluxo de desenvolvimento o projetista só teria contato com o conteúdo do .mpa através do *software* e, portanto, não veria os comentários dentro do arquivo. Ainda assim, manter os comentários de interesse é uma importante funcionalidade a ser explorada em iterações futuras, para casos em que eventualmente os desenvolvedores não utilizem a ferramenta aqui proposta.

Outros arquivos de pré-configuração utilizados no projeto também foram lidos com sucesso. Um arquivo de pré-configuração 'base', disponibilizado pelo próprio MPA, com funções de cálculo e alguns equipamentos simples apresentou problemas na leitura, revelando um detalhe não incorporado na modelagem. O Gerador entende que o conteúdo de um bloco de código avulso na pré-configuração deve conter uma interface ou definição da DLL, portanto, espera estruturas fixas para a leitura. No entanto, o MPA aceita a definição de outros elementos de código como funções nativas em Lua, cuja transformação ANTLR não foi modelada para compreender, e possivelmente outras estruturas não presentes em nenhum arquivo utilizado pelo projeto até agora, o que limita o escopo de sua utilização.

Ainda assim, a remoção desse bloco (que poderia ser modelado na estrutura das Funções do MPA de forma compatível com o modelo de dados da aplicação) permitiu a leitura do arquivo 'base', que conta com 33 funções e 37 equipamentos, totalizando 5597 linhas, o que demonstra a capacidade da aplicação de lidar com uma quantidade consideravelmente maior de dados.

As melhorias empíricas advindas da utilização do *software* serão verificadas a partir de

sua utilização continuada, inicialmente na equipe do projeto de pesquisa do DAS, e posteriormente, pelos engenheiros da própria Petrobras.

Figura 32 – Comparação de um arquivo de pré-configuração original e um arquivo gerado

7 Conclusões

De maneira geral, o trabalho aqui apresentado se insere em um conjunto de diferentes aplicações que visam auxiliar o desenvolvimento de projetos de controle para plantas petroquímicas, criadas por trabalhos anteriores. Com o novo *software*, o fluxo de desenvolvimento de um projeto de controle se torna mais regular, o que reduz a curva de aprendizado para novos estudantes e torna mais próxima a experiência do usuário em cada etapa. Levar a pré-configuração dos projetos a um alto nível, sem precisar trabalhar diretamente com código, reduz a necessidade de voltar a atenção aos detalhes de integração (tipagem de dados, consistência entre aplicações). Para os projetistas de controle da equipe da UFSC, isso significa dedicar-se mais às sintonias e estruturas de controle, otimizando seu tempo e os resultados de campo.

A abordagem utilizada para o desenvolvimento do gerador de código, apoiada em conceitos da *Model Driven Development* (MDD), se prova bastante vantajosa para o projeto, apesar de ser apenas um elo na cadeia entre a definição de classes e a criação de uma aplicação de controle baseada nos objetos intermediários definidos pelo usuário. A modularidade utilizada durante o desenvolvimento da aplicação permite que interface, transformação de leitura e transformação de escrita possam ser modificadas de maneira independente, de maneira que eventuais mudanças nas ferramentas, tecnologias ou mesmo na estrutura dos arquivos de pré-configuração possam ser testadas de maneira individual e substituídas sem a necessidade de grandes modificações.

Além de se tornar uma ferramenta útil para o projeto, o *software* apresentado neste trabalho traz uma forte contribuição por solidificar um caminho teórico e um *background* de procedimentos para a criação processos de transformação de leitura ou escrita de código, possibilitando que novos problemas e soluções com a temática de geração de código sejam exploradas de maneira mais ágil. A utilização das tecnologias ANTLR e JET possibilitaram a obtenção de resultados de maneira rápida, mas, principalmente, da visualização do problema em questão sob uma perspectiva de alto nível, trabalhando com a definição de *templates* e gramáticas, elementos mais facilmente modificáveis do que o código das transformações resultantes.

A aplicação dialoga, ainda, com o trabalho de mestrado que propõe o metamodelo apresentado na seção 3.1, e deve funcionar como objeto de estudo de validação e refinamento de sua estrutura.

Os resultados obtidos com a ferramenta mostram as dificuldades em modelar estruturas utilizando engenharia reversa, i.e., sem ter o conhecimento de todas as possibilidades que a pré-configuração no MPA permite. Apesar de satisfatória no âmbito dos projetos de controle propostos pela equipe da UFSC, a aplicação não garante a leitura de todo e qualquer arquivo de pré-configuração compatível com o MPA, o que reforça a importância de áreas como a

modelagem de sistemas na engenharia de software e na automação industrial.

7.1 Perspectivas e Trabalhos Futuros

Em relação a melhoramentos e linhas de pesquisa a serem seguidas, é possível elencar os seguintes pontos para incorporação de funcionalidades no gerador:

- Adicionar ao modelo de dados da aplicação uma nova classe para manutenção de comentários feitos no arquivo de pré-configuração, para facilitar a visualização em casos de o gerador não ser utilizado.
- Incorporar mecanismos de versionamento para cada elemento definido no arquivo, de maneira a verificar automaticamente compatibilidade das funções e/ou atributos com arquivos de pré-configuração anteriores.
- Estudar o desenvolvimento de uma segunda transformação *Model to Text* para permitir a análise do conteúdo dos blocos `'code={}'` presentes nos métodos do equipamentos e nas funções, para efetuar uma checagem de erros de forma similar a IDEs e compiladores, verificando se atributos de outros equipamentos acessados nas funções estão presentes na pré-configuração.
- Melhorar o gerenciamento do arquivo a ser exportado, dando ao usuário a possibilidade de quais equipamentos/funções escolher para a exportação, e permitir a importação/exportação de múltiplos arquivos simultaneamente.

Referências

1 MPA - Módulo de Procedimentos Automatizados. (Manual do Sistema). [S.l.]: Tecgraf. Acesso: 20/06/2017. Citado 3 vezes nas páginas 9, 4 e 10.

2 ATKINSON, C.; KUHNE, T. Model-driven development: a metamodeling foundation. *IEEE software*, IEEE, v. 20, n. 5, p. 36–41, 2003. Citado 2 vezes nas páginas 9 e 19.

3 IERUSALIMSCHY, R. *Programming in lua*. [S.l.]: Roberto Ierusalimschy, 2006. Citado na página 4.

4 DESENVOLVIMENTO E EXECUÇÃO DE APLICAÇÕES DE CONTROLE E AUTOMAÇÃO DE PROCESSOS INDUSTRIAIS. Tecgraf. Disponível em: <http://www.tecgraf.puc-rio.br/pt/software/sw-mpa.html>. Citado na página 7.

5 JET Tutorial Part 1 (Introduction to JET). Eclipse Foundation. Acesso: 20/06/2017. Disponível em: <https://eclipse.org/articles/Article-JET/jet_tutorial1.html>. Citado na página 20.

6 PARR, T. *The definitive ANTLR 4 reference*. [S.l.]: Pragmatic Bookshelf, 2013. Citado na página 23.