Guilherme Alves Gil

Mestre Integrado em Engenharia Electrotécnica e de Computadores

Ambiente de Simulação com PLC para Automação Industrial

Dissertação para obtenção do Grau de Mestre em

Engenharia Electrotécnica e de Computadores

Orientador: Doutor João Almeida das Rosas, Professor Auxiliar, Universidade Nova de Lisboa

Co-orientador: Doutor Luís Brito Palma, Professor Auxiliar, Universidade Nova de Lisboa

Júri

Presidente: Doutor Pedro Alexandre da Costa Sousa, Professor Associado, Universidade Nova de Lisboa

Arguentes: Doutor Paulo José Carrilho de Sousa Gil, Professor Auxiliar, Universidade Nova de Lisboa

Vogais: Doutor João Almeida das Rosas, Professor Auxiliar, Universidade Nova de Lisboa

FACULDADE DE CIÊNCIAS E TECNOLOGIA

UNIVERSIDADE NOVA DE LISBOA

Março, 2021

Ambiente de Simulação com PLC para Automação Industrial

Copyright © Guilherme Alves Gil, Faculdade de Ciências e Tecnologia, Universidade NOVA de Lisboa.
A Faculdade de Ciências e Tecnologia e a Universidade NOVA de Lisboa têm o direito, perpétuo e sem limites geográficos, de arquivar e publicar esta dissertação através de exemplares impressos reproduzidos em papel ou de forma digital, ou por qualquer outro meio conhecido ou que venha a ser inventado, e de a divulgar através de repositórios científicos e de admitir a sua cópia e distribuição com objetivos educacionais ou de investigação, não comerciais, desde que seja dado crédito ao autor e editor.

Este documento foi gerado utilizando o processador (pdf)LaTeX, com base no template NOVAthesis, desenvolvido no Dep. Informática of FCT-NOVA por João M. Lourenço.

Agradecimentos

Esta dissertação marca o final da fase de vida mais importante que passei até agora. Foi uma fase extremamente difícil, mas, ao mesmo tempo não mudaria nenhum aspecto.

Em primeiro lugar gostaria de agradecer ao Professor João Almeida das Rosas por toda a ajuda e paciência que me providenciou e por todos os conhecimentos passados ao longo deste caminho.

Gostaria de agradecer ao Professor Luís Figueira Brito Palma pela oportunidade que me deu neste tema e pelas opiniões dadas sempre com um sentido construtivo.

Aos meus amigos e colegas Nuno Muchaxo, Gonçalo Mestre, Tiago Maurício, João Pedro Ramalho, Tiago Correia, Francisco Rebola, Leandro Filipe e Marco Silva, um muito obrigado.

Aos meus pais Francisco Gil e Maria Filomena Gil um muito obrigado por todo o carinho e ajuda que me deram ao longo da minha vida e fase académica.

À minha namorada, Valentyna Sheyhus, nesta fase final foi a pessoa que mais junta teve de mim, muito obrigado por tudo.

Finalmente, muito obrigado à FCT-UNL e ao DEE por me terem proporcionado um ambiente para estudar aquilo que mais gosto.

Resumo

Os controladores lógicos programáveis são vastamente utilizados em toda a indústria da automação. Testar directamente no sistema físico pode ser um processo demoroso como até pode originar graves problemas nos equipamentos. Estes problemas podem ser encontrados em ambiente académico como também o orçamento pode ser demasiado baixo para a aquisição de sistemas de controlo e software de desenvolvimento dos autómatos. Uma solução para colmatar este problema é a utilização de ambientes de simulação capazes de simular o comportamento de autómatos como também de processos reais. Existem ambientes de simulação desenvolvidos por alguns fabricantes capazes de testar programas escritos segundo a norma IEC 61131-3, mas possuem a contrapartida de as licenças serem dispendiosas.

O objectivo desta dissertação é a descrição do desenvolvimento de um simulador de PLC's com ligação a um simulador de processo industrial. O simulador de PLC's consegue reconhecer instruções da linguagem em Texto Estruturado (ST) e, também, validar a sua sintaxe, através da implementação de um analisador léxico e um analisador sintáctico (utilizando as ferramentas Flex e Bison). Através destes analisadores é possível traduzir código em Texto Estruturado para uma estrutura semanticamente equivalente, que ao ser percorrida, são executadas as operações introduzidas pelo utilizador. Repetindo esta execução de operações e conseguindo ler sinais de entrada e actuar em sinais de saída, é possível controlar o simulador de processo construído. Este simulador possui um modelo de um processo de fluxo de materiais, desenvolvido através do motor de jogos Unity. Foi possível implementar um cenário em 3D, introduzindo, assim, alguns aspectos de gamificação. Também foi desenvolvida uma interface gráfica, utilizando a linguagem de programação Java, onde o utilizador consegue introduzir o seu código em ST, declarar endereços, quer analógicos ou digitais, e verificar a sua evolução. Através desta componente gráfica é possível comandar o simulador de PLC's e fazer a ligação com o simulador de processo. Este projecto foi dado como um sucesso, como demonstram os resultados experimentais. Deste modo, é feita uma contribuição na área do ensino da automação.

Palavras-chave: Linguagem em Texto Estruturado, Simulador, Controlador Lógico Programável, Ambiente de Simulação, C++, Java, Unity

Abstract

Programmable logic controllers are used extensively in the entire automation industry. Testing directly on the physical system can be a time consuming process as it can lead to serious equipment problems. These problems can be found in an academic environment as well as the budget may be too low to purchase control systems and software to develop the PLCs' programs. One solution to overcome this problem is the use of simulation environments capable of simulating the behavior of PLCS as well as real processes. There are simulation environments developed by some manufacturers capable of testing programs written according to the IEC 61131-3 standard, but they have the disadvantage that the licenses are expensive.

The objective of this dissertation is to describe the development of a PLC simulator connected to an industrial process simulator. The PLC simulator can recognize Structured Text (ST) language instructions and also validate its syntax, through the implementation of a lexical analyzer and a syntactic analyzer (using Flex and Bison tools). Using these analyzers it is possible to translate Structured Text code into a semantically equivalent structure, which, when going through it, executes the operations entered by the user. In this way, it is possible to control the process simulator that was built. This simulator has a model of a material flow process, developed using the Unity game engine. It was possible to implement a 3D scenario, thus introducing some gamification aspects. A graphic interface was also developed, using the Java programming language, where the user can enter his code in ST, declare addresses, either analog or digital, and check its evolution. Through this graphic component it is possible to control the PLC simulator and make the connection with the process simulator.This project was considered a success, as demonstrated by the experimental results. In this way, a contribution is made in the area of automation teaching.

Keywords: Structured Text Language, Simulator, Programmable Logic Controller, Simulation Environment, C++, Java, Unity

Índice

Lista de Figuras **xv**

Lista de Tabelas **xvii**

Listagens **xix**

Siglas **xxi**

1 Introdução **1**

1.1 Motivação 1

1.2 Objectivos 2

1.3 Organização do Documento 2

2 Estado da Arte **5**

2.1 Simulação para Automação Industrial 5

2.1.1 Simulação Contínua e Simulação Discreta 6

2.1.2 Estudo de uma simulação 8

2.1.3 Tecnologias de Simulação utilizadas na Indústria 8

2.2 Controlador Lógico Programável 10

2.2.1 Enquadramento Histórico 10

2.2.2 Componentes de um PLC 12

2.2.3 Ciclo de Operação 13

2.2.4 Linguagens de Programação Utilizadas 14

2.2.5 Simulação de PLC's em Ambiente Virtual 14

2.3 Sistemas SCADA 17

2.3.1 Enquadramento histórico 18

2.4 Compiladores e Linguagens de Alto Nível 18

2.4.1 Enquadramento histórico 19

2.4.2 Vantagens e desvantagens do uso de compiladores e interpretadores 20

2.4.3 Estrutura de um compilador 21

2.5 Gamificação no Ensino da Engenharia 22

2.5.1 Práticas para a Gamificação 23

2.5.2 Simulações Lúdicas 24

2.6 Sistemas Existentes 25
2.7 Trabalho Relacionado 26
2.8 Conceitos Fundamentais 27
2.8.1 Especificação da Linguagem Texto Estruturado 27
2.8.2 Temporizadores 32
2.8.3 Caracterização das ferramentas Flex e Bison 33
2.8.4 Árvores de Análise Sintáctica 39
2.8.5 Interoperabilidade entre Java e C++ 42

3 Desenvolvimento **45**
3.1 Arquitectura Proposta e Requisitos 45
3.2 Implementação 47
3.2.1 Implementação de um compilador usando Flex e Bison 47
3.2.2 Interface Gráfica utilizando JavaFX 56
3.2.3 Construção de um Simulador de Processo 60
3.2.4 Comunicação Autómato Virtualizado e Simulador de Processo . . 62

4 Aplicação **65**
4.1 Testes Preliminares 65
4.2 Teste do Simulador 69

5 Conclusão **73**
5.1 Conclusões 73
5.2 Trabalho Futuro 74

Bibliografia **77**

Anexos **81**

I Interpretador de uma Árvore de Análise Sintática **81**

II PID escrito em Matlab **85**

Lista de Figuras

2.1 Parte de um problema de agendamento usando a ferramenta Arena (Retirado de: [41]). 7
2.2 Esquemático para o estudo de uma simulação (Retirado de: [35]). 9
2.3 Arquitectura de um PLC. Retirado de: [13] 12
2.4 Ciclo de operação de um autómato. 13
2.5 Arquitectura de um ambiente de simulação de PLCs. (Retirado e adaptado de: [44]). 15
2.6 Pirâmide da automação segundo o modelo ISA 95 (Retirado e adaptado de: [57]). 17
2.7 Um compilador. Retirado e adaptado de: [10] 19
2.8 Anatomia de um compilador. Retirado de: [22] 21
2.9 Um jogo pode ser uma simulação lúdica. (Retirado e adaptado de: [45]). 25
2.10 Passos para a formação de uma ASB utilizando as ferramentas Flex e Bison. Retirado e adaptado de [42] 34
2.11 Exemplo da geração de um executável para um analisador puramente léxico 36
2.12 Árvore de análise sintática da declaração "2*(3+x)"e os tipos de nós utilizados para a sua implementação 41
2.13 lista de comandos para a geração de um ficheiro Java com funções nativas de C 42
2.14 Configuração da inclusão de bibliotecas de Java num projecto de C++ usando a plataforma Visual Studio 43
2.15 Configuração do projecto escrito em C++ para a geração de uma DLL 43
2.16 Resulta final da configuração de uma DLL. Plataforma NetBeans 44

3.1 Arquitectura proposta. 47
3.2 Árvore de análise sintáctica para a declaração for 54
3.3 Separador IDE. 58
3.4 Componentes de um objecto de jogo tipo cubo. Retirado de: [8] 60
3.5 Modelo implementado. 62
3.6 Comunicação entre Simulador de PLC e simulador de processo. 62
3.7 Separador Unity da interface gráfica. 63

4.1 Programa para teste de operações lógicas. 66

4.2 Resultados do teste de operações lógicas apresentados na linha de comandos do ambiente de desenvolvimento Netbeans. 67
4.3 Resultados do teste da declaração concional IF. 67
4.4 GUI com o teste de um temporizador 68
4.5 Resultado da precisão de um temporizador de 10 segundos. 68
4.6 Código para um controlador PID. 69
4.7 Resultado do teste para um controlador PID. 69
4.8 Código responsável pelo controlo do cenário construído. 70
4.9 Execução do simulador. 71

II.1 Resultado do teste para um controlador PID escrito em Matlab. 86

Lista de Tabelas

2.1 Tipo de dados básicos da linguagem Texto Estruturado. Retirado de: [20]. . . 28
2.2 Nomenclatura dos vários tipos de variáveis. 28
2.3 Características e variáveis dos vários modos. Retirado de [21] 32
2.4 Exemplos de expressões regulares. 35
2.5 Variáveis e funções da ferramenta Flex 35
2.6 Tipos de nós para uma árvore de análise sintáctica 41

3.1 Variáveis para a passagem de valores entre analisadores. 48
3.2 Nós utilizados para a implementação da árvore de análise sintáctica 50
3.3 Funções para a utilização do interpretador 57
3.4 Elementos implementados no simulador 61

Listagens

2.1 Declaração if . . . 29
2.2 Declaração case . . . 30
2.3 Declaração for . . . 30
2.4 Declaração while . . . 31
2.5 Declaração repeat . . . 31
2.6 Exemplo de um temporizador TON . . . 33
2.7 Exemplo de um ficheiro para geração de um analisador puramente léxico 35
2.8 Exemplo da notação Backus Naur Form (BNF) . . . 37
2.9 Exemplo do processo de Análise de Deslocação e Redução . . . 37
2.10 Exemplo de um analisador sintáctico para uma calculadora . . . 38
2.11 Exemplo de um analisador léxico para uma calculadora . . . 39
2.12 Exemplo de uma gramática que constrói uma árvore de análise sintática . 40
2.13 Interpretador para uma calculadora . . . 41
2.14 Classe de java utilizando uma função nativa . . . 42
2.15 Função nativa chamada em C++ . . . 43
2.16 Ficheiro HelloC.java . . . 44
2.17 Ficheiro JNIexample.java . . . 44
3.1 Algumas expressões regulares utilizadas no analisador léxico. . . . 47
3.2 Protótipos de funções e declaração de variáveis do analisador sintáctico . 49
3.3 Variáveis Bison e declaração de terminais . . . 49
3.4 Declaração de símbolos terminais referentes a declarações e operações aritméticas e lógicas . . . 51
3.5 Declaração de símbolos terminais referentes a declarações e operações aritméticas e lógicas . . . 52
3.6 Gramática para a redução de uma expressão . . . 52
3.7 Gramática para a declaração iterativa for . . . 53
3.8 Gramática para um conjunto seguido de declarações . . . 54
3.9 Gramática para um conjunto seguido de declarações . . . 55
3.10 Interpretação da declaração for . . . 56
3.11 Mapas criados para o armazenamento de variáveis e estados das mesmas 56
I.1 Código em C++ para um interpretador . . . 81
II.1 Código para um PID escrito em Matlab . . . 85

Siglas

BNF Backus Naur Form

DLL Dynaminc Linking Library

ERP *Enterprise Resourcer Planning*

FBD Function Block Diagram

GLC Gramática Livre de Contexto

HMI *Human Machine Interface*
HTTP Hypertext Transfer Protocol

IL Instruction List

LAN *Local Area Network*
LD Ladder Diagram

MES *Manufacturing Execution System*

PID Proporcional Integral Derivativo
PLC *Programmable Logic Controller*

RTU *Remote Terminal Unit*

SCADA *Supervisory Control and Dara Acquisition*
SFC Sequential Functional Charts
ST Structured Text

WAN *Wide Area Network*

Capítulo 1

Introdução

1.1 Motivação

Existe a necessidade de testar em ambientes de simulação os programas desenvolvidos quer na indústria quer seja em ambiente académico. Testar directamente em ambiente real pode ser um processo demoroso como também pode originar graves problemas nos sistemas físicos a controlar devido a comandos de actuação incorrectos o que torna esta prática pouco rentável no mundo da indústria. O mesmo se aplica em contexto universitário, onde os formandos não possuem experiência com o equipamento. Os equipamentos a usar (sensores, autómatos, motores, etc) podem estar fora do alcance dos orçamentos de instituições de ensino, algo também com custo elevado são as licenças para os softwares de desenvolvimento de autómatos. Outro problema é o caso de situações disruptivas (pandemia causada pelo COVID-19) que dificultam ou tiram por completo o acesso de alunos e profissionais aos locais de trabalho e assim aos equipamentos.

As mesmas linguagens para Controladores Lógico Programáveis apresentam diferenças de construtor para construtor, mas todas seguem a norma IEC 61131-3, e portanto, possuem características idênticas o que ajuda a mudar de tipo de autómato. Uma destas linguagens tem o nome de Texto Estruturado que se assemelha muito a Pascal ou C, o que é uma mais valia num âmbito escolar, pois os alunos estão familiarizados com este tipo de linguagens de programação.

Um ambiente de simulação, que apenas simule a execução do programa a introduzir no autómato pode ser insuficiente. Pode haver a necessidade de adicionar também um modelo de simulação de um processo real ao ambiente já existente. Deste modo existe uma maior facilidade na validação do programa construído. Também é possível introduzir o tema da gamificação, desenvolvendo um modelo com uma componente gráfica acentuada e utilizando mecanismos de jogo, assim aumentando a imersão dos seus utilizadores na

utilização do simulador.

Um ambiente de simulação ajuda bastante a detecção de erros e a percepção do problema a resolver, daí muitas empresas recorrem a este tipo de método para testarem os programas em desenvolvimento.

1.2 Objectivos

Este projecto tem como objectivo a criação de um simulador para um autómato, que consiga receber código em Texto Estruturado. Será necessário desenvolver um ambiente gráfico que consiga validar o código introduzido pelo utilizador. Este código terá de ser traduzido para uma estrutura que posteriormente consiga ser lida de forma a executar as mesmas operações daquilo que é dado como entrada. O simulador apenas poderá servir para testar o programa sozinho, isto é, apenas verificando as alterações das variáveis declaradas. Também terá de possuir a opção de se conectar a um cenário de simulação onde será possível controlar um modelo de um processo industrial. Em suma, as principais objectivos são:

- Desenvolvimento de uma ferramenta capaz de validar e traduzir código em Texto Estruturado para uma estrutura com a mesma semântica;
- Implementação de um simulador de PLC's;
- Criação de uma interface gráfica;
- Construção de um simulador de processo com estratégias de gamificação;

1.3 Organização do Documento

O presente documento é estruturado em 5 capítulos:

1. Introdução
2. Estado da Arte
3. Implementação
4. Aplicação
5. Conclusão

Este capítulo tem como objectivo apresentar o problema de uma forma geral e a motivação para o desenvolvimento do projecto.

O capítulo 2 aborda os temas relevantes para a concepção da ferramenta construída. Também são referidos alguns sistemas existentes e trabalhos académicos que possuem

semelhanças com este projecto e que contribuíram para o seu desenvolvimento. Por fim são apresentados os conceitos fundamentais das tecnologias utilizadas para este projecto.

No seguinte capítulo, 3, em primeiro lugar é descrita a arquitectura utilizada neste projecto e de seguida são explicados todos os passos do desenvolvimento do sistema construído.

No capítulo, 4, são referidos os testes realizados que tentam validar a concepção do compilador e do comportamento do autómato virtual como também do ambiente de simulação.

Finalmente, o capítulo 5 apresenta as conclusões finais sobre a construção do projecto como também são feitas sugestões para trabalho futuro.

Capítulo 2

Estado da Arte

Intitulado de Estado da Arte, o presente capítulo foca-se em temas relevantes para esta dissertação como também trabalhos académicos desenvolvidos e tecnologias existentes no mercado mundial. Inicialmente é apresentado o tema da simulação, é explicada a distinção entre simulação contínua e discreta, é descrito o procedimento para o estudo de uma simulação e por fim são apresentadas algumas tecnologias utilizadas na indústria. Em segundo, o tema é o controlador lógico programável, é apresentada uma breve história do seu surgimento, a sua arquitetura base, funcionamento e por fim é apresentada uma secção para a simulação de autómatos. De seguida é abordada a importância de linguagens de alto nível como os respetivos compiladores e interpretadores. Na seguinte secção o tema é a gamificação no ensino da engenharia, são apresentados alguns aspetos de jogos que podem ser introduzidos em ambientes educativos e é abordado o tema de simulação lúdica. Seguidamente são apresentados os conceitos fundamentais para a conceção da presente dissertação. Por último são apresentados alguns produtos, de código aberto ou com custos associados, e, alguns trabalhos académicos considerados relevantes para o projecto.

2.1 Simulação para Automação Industrial

As tendências para a globalização e de descentralização na manufactura e automação industrial fazem com que os produtos e o seu desenvolvimento se tornem mais complexos [38]. Simulação é a reprodução de um processo real ou de um sistema durante o decorrer do tempo [11]. Também pode ser entendido como o funcionamento de um modelo de um sistema.

Um modelo é a representação da construção de um sistema e das dinâmicas a estudar. O modelo tem de ser o mais similar possível ao sistema mas ao mesmo tempo simplista,

pois é necessário percebê-lo e testá-lo. Modelos matemáticos podem ser classificados como deterministas (valores de entrada e saída são fixos) ou estocásticos (pelo menos uma da entrada ou saída é probabilística); estático (o tempo não é contabilizado) ou dinâmico (as variáveis podem sofrer mudanças no decorrer tempo). Tipicamente modelos de simulação são estocásticos e dinâmicos [35]. Modelos podem possuir uma componente gráfica de modo a facilitar a visualização da simulação.

Através de simulações e das suas análises é possível ganhar conhecimentos de sistemas complexos ou testar os programas para o seu controlo sem causar distúrbios nos sistemas reais ou mesmo sem estarem implementados [47].

2.1.1 Simulação Contínua e Simulação Discreta

O tópico da simulação pode ser dividido em dois tipos distintos: simulação contínua e simulação de eventos discretos.

Em **simulação contínua** os estados dos modelos representados mudam continuando, pois são modelados por equações diferenciais discretizadas. O conceito deste tipo de simulações não faz com que o modelo implementado seja em tempo continuo nem que sejam utilizadas variáveis de estado contínuas [1].

Um dos exemplos de simulação contínua é a simulação de Monte Carlo. É um tipo de simulação à base de análise estatística para conseguir resultados, sendo muito relacionada com testes aleatórios. Os modelos, tipicamente possuem um certo número de variáveis de entrada e depois destas processadas, outro certo número de variáveis de saída. Os parâmetros de entrada dependem também de factores externos, e, ser o mais realistas possíveis estão sujeitos a discrepâncias derivadas de mudanças sistemáticas. Um modelo que não considere estas variação tem o nome de caso base. Podem existir várias versões dos modelos que considerem o melhor caso, pior, base e intermédios. Esta abordagem traz desvantagens, como o facto de ser difícil de avaliar o pior caso e o melhor para cada parâmetro de entrada.

Na simulação de Monte Carlo, uma distribuição estatística é identificada, e, pode ser usada em cada parâmetro de entrada. Amostras aleatórias são identificadas por cada distribuição, cada amostra representa valores para cada variável de entrada. Por cada conjunto de variáveis de entrada é processado um conjunto de variáveis de saída fazendo assim um cenário. Estas cenários são analisados e decisões terão de ser tomadas para atribuir qual acção tomar e etiquetar os cenários [49].

A **simulação discreta** é baseada na modelação de um sistema como um sistema dinâmico e discreto através da representação do seu estado através de variáveis de estado, cada mudança de estado tem o nome de evento.

Este tipo de simulações pode ser divida em dois métodos. O primeiro tem o nome de *next-event time progression*, não possui estados entre eventos consecutivos, portanto a simulação pode logo saltar para o próximo evento. O segundo, é o método *fixed-increment time progression*, o tempo utilizado para uma simulação é dividido em pequenas partes

sendo que o estado do sistema é actualizado a cada incremento de tempo que passa dependendo dos eventos que ocorreram.

Existem vários formalismos para este tipo de simulações, tais como: Redes de Petri, diagramas de estado e DEVS (*Discrete Event Systems Specifications* ou Especificações de Sistemas de Eventos Discretos) [2].

As simulações discretas são normalmente executadas através de ferramentas de *software* por linguagens com C++ ou por linguagens especializadas em simulações (ex: SIMSCRIPT). Todas estas ferramentas possuem atributos comuns tais como:

Entidades - São representações de elementos reais;

Relações - Servem para ligar entidades entre elas;

Executivo - É responsável por controlar a evolução temporal da simulação e executar os eventos discretos;

Gerador de Números Aleatórios - Criar informação aleatória para variáveis de entrada caso necessário;

Resultados e Estatísticas - Para dar ao utilizador formas de medir o êxito da simulação [3].

O *software* Arena é um exemplo de uma ferramenta para modelação de sistemas em eventos discretos. Este usa o paradigma de simulação em rede de processamento (*processing network* ou apenas PN). Este paradigma consiste, sucintamente, em objectos entraram no sistema modelado (semelhante a máquinas de estado) através de eventos de chegada num nó de entrada e estes correm uma cadeia de vários nós de processamento (que representam por exemplo máquinas de manufactura) onde são sujeitos a actividades de processamentos e de seguida passam para o nó seguinte. O nó de saída é o último nó do sistema e faz com que os objectos saiam do sistema através de um evento de partida [4]. A figura 2.1 ilustra parte de um problema de agendamento (*Job-shop problem*) usando a ferramenta Arena. Pode ser visto o nó de chegada (mais à esquerda) onde os objectos irão entrar, dois nós de processamento (a meio da cadeia de nós), e o nó de partida (nó mais à direita).

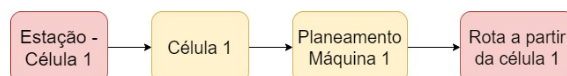

Figura 2.1: Parte de um problema de agendamento usando a ferramenta Arena (Retirado de: [41]).

A simulação de eventos discretos é menos detalhada que a simulação contínua, mas é muito mais simples de implementar e por isso mais usada [35].

2.1.2 Estudo de uma simulação

O estudo da execução de simulações é todo ele um processo iterativo. O modelo do sistema a estudar pode sofrer constantes alterações. Quase sempre é preciso a interacção humana nas fases que formam o estudo de simulação, no estudo do sistema, decisões, desenvolvimento do modelo, análise dos testes e conclusões, excepto na fase da simulação, o que traz um vantagem, pois são quase sempre utilizadas ferramentas digitais para executar simulações. Os passos para desenvolver um modelo, desenvolver uma simulação, e analisá-la são as seguintes (segundo [35]):

1. Identificar o problema;
2. Formular o problema;
3. Recolher e processar informação do sistema real;
4. Desenvolver o modelo;
5. Validar o modelo;
6. Documentar para uso futuro;
7. Escolher métricas e possíveis parâmetros de entrada capaz de influenciá-las;
8. Escolher condições para as execuções da simulação;
9. Executar simulações;
10. Interpretar e apresentar resultados;
11. Recomendar mudança de rumo (podem ser necessários mais testes para aumentar a precisão e assim melhorar a sensibilidade das análises).

Com base nos passos referidos em cima é apresentado uma forma sequencial para o estudo das execuções de simulações, mas não é mandatório seguir na sua integra. Podem ser precisos passos intermédios para atingir o objectivo pretendido. A figura 2.2 ilustra aquilo que foi apresentado.

2.1.3 Tecnologias de Simulação utilizadas na Indústria

Desde o desenvolvimento de um produto, e passando pelo seu funcionamento e manutenção pode sempre ser necessário o uso de simulações ou de ferramentas e ambientes de simulação. Algumas são tecnologias para a simulação são apresentadas de seguida.

A **Realidade Aumentada** é definida como a visualização em tempo real do ambiente físico onde foi adicionada informação virtual. Na indústria automóvel, protótipos de veículos podem ser completados através da adição de componentes virtuais[25]. O seu uso, pode ser estendido para a formação e manutenção dentro de ambientes indústrias

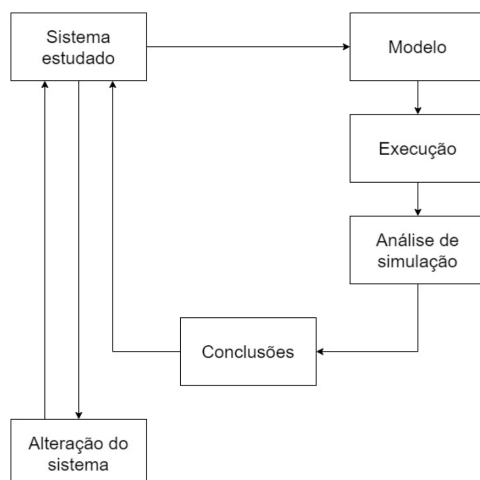

Figura 2.2: Esquemático para o estudo de uma simulação (Retirado de: [35]).

onde, por exemplo, em cada máquina, informação relevante, tal como o seu estado e regras para a sua operação, é disponibilizada. De forma a melhorar a experiência do utilizador em relação ao desenvolvimento de um produto, um capacete ou óculos de realidade aumentada pode ser enviado [37]. A realidade aumentada também é utilizada em calibração visual, anotações digitais e comunicação na Daimler AG, um dos maiores fabricantes de automóveis a nível mundiais [12].

Concepção assistida por computador é o uso da tecnologia que utiliza sistemas informáticos na criação, modificação, análise e optimização do desenvolvimento de um produto [16].

Maquetes Digitais consistem em modelos em 3D que representam a estrutura mecânica de um sistema. Um protótipo virtual é criado para identificar problemas em fases iniciais do projecto levando assim a alterações. Diminuindo assim custos caso fosse recorrido a um modelo físico. Muito ligado a concepção assistida por computador [52].

A **Realidade Virtual** é definida como o uso de sistemas informáticos para gerar a simulação de um ambiente real ou até fictício [33]. É uma tecnologia imersiva para o utilizador, o que traz uma ligação entre humano e máquina maior que as outras tecnologias, abstraindo-se assim do mundo real que o rodeia. Na área da gestão, o ambiente gerado pela realidade virtual disponibiliza a interacção e visualização de utilizadores com o protótipo virtual, colaborando em tempo real [15].

Manufactura assistida por computador é o uso de ferramentas informáticas para planear, gerir e controlar as operações de uma unidade industrial de manufactura. Este tópico não se foca no desenvolvimento do produto, mas sim no fabrico do mesmo [48].

O fluxo de materiais na manufactura é o movimento de materiais seguindo um certo percurso numa unidade industrial. A melhoria em mecanismos de controlo para o fluxo de materiais pode melhorar o desempenho das linhas de transporte com várias rotas possíveis [23]. Uma plataforma para planeamento do movimento automatizado de linhas, integrado com um modelo do sistema de fluxo de materiais que gera automaticamente

os caminhos para objectos sem causar a colisões é de extrema importância [24]. Portanto pode ser crucial **simulações de fluxo de materiais** para empresas de logísticas, aeroportos entre outras empresas e sectores.

2.2 Controlador Lógico Programável

Um PLC (*Programmable logic controller* ou Controlador Lógico Programável) é um computador de uso industrial com memória programável e com funções de lógica de controlo, sequenciação, tempo, manipulação aritmética de dados e capacidades de contagem. Pode ser visto como um computador industrial que possui um *CPU*, memória e uma interface *I\O*. Recebe informação de, por exemplo, vários sensores tais como sensores de fim de curso e de proximidade, executa o programa guardado na sua memória e envia sinais de actuação.

A interface *I\O* é a ponte entre dispositivos e os controladores. Através desta interface o processador consegue ler e aceder quantidades físicas de certa máquina ou processo tais como: proximidade, posição, movimento, nível de água, temperatura, pressão, etc. Baseando-se no estado lido, o *CPU* decide quais os sinais a enviar para dispositivos tais como válvulas, motores, alarmes, etc [56].

2.2.1 Enquadramento Histórico

Antes dos PLC surgirem, os sistemas de controlo eram feitos através de relés. Desta forma as salas de controlo estavam lotadas de cabos, blocos terminais e destes relés. Com este tipo de sistemas havia bastantes problemas associados, tais como:

- A falta de flexibilidade na expansão do sistema, como também a quantidade de tempo desperdiçado a ajustar o processo quando são efectuadas mudanças;
- Resolução de problemas devido à sujidade entre contactos, cabos soltos, blocos terminais mal referenciados, falta de diagramas e respectiva documentação para os sistemas.

A expressão *"Five hours to find it and five minutes do fix it"* nasceu também nesta altura devido a este tipo de problemas que os técnicos confrontavam simultaneamente [51].

Em 1968, Bill Stone, que trabalhava na *Hydramatic Division of General Motors Corporation* apresentou um artigo sobre os problemas de fiabilidade e documentação das máquinas do seu departamento. Também apresentou critérios desenvolvidos por si e por engenheiros da GM para um "controlador padrão de máquinas"(*"standard machine controller"*).

De acordo com estes critérios, o modelo inicial destas máquinas não só teria de eliminar relés, com custos elevados, das linhas de montagens durante alterações e trocar relés electromecânicos não fiáveis, mas também:

- Estender as vantagens dos circuitos estáticos até 90% das máquinas na unidade industrial;
- Redução dos tempos de resolução de problemas, manutenção e programação através de lógica ladder;
- O sistema teria de ser modular, isto é, permitir uma fácil expansão e troca de componentes;
- Teria de funcionar num ambiente industrial sujeito a sujidade, humidade, interferências eletromagnéticas e vibrações;
- Teria de inclui funções lógicas.

Estas especificações e também um pedido para construir este protótipo, foram dados a quatro empresas:

- *Allen-Bradley*;
- *Digital Equipment Corporation* (*DEC*);
- *Century Destroit*;
- *Bedford Associates*.

A *Bedford Associates* cuja equipa era constituída por Richard Morley, Mike Greenberg, Jonas Landau, George Schwenk and Tom Boissevain, construiu a sua unidade com o nome 084, devido a ser o octogésimo quarto projecto da empresa [31, 51]. Esta unidade era modular e robusta, não usava *interrupts* e usava mapeamento directo na memória. Depois de conseguirem financiamento, a equipa decidiu formar uma nova empresa chamada Modicon(*MOdular DIgital CONtroller*) que trabalhava de perto com a *Bedford Associates* para desenvolver o controlador.

Em 1969, foi o ano em que o Modicon 084 ganhou o concurso da *GM*. Este controlador era divido principalmente em três partes: o processador, memória e um *logic solver board* [51].

O Modicon 084 foi desenhado para ser fiável, ser robusto (nem possuía um botão de ON/OFF) e era totalmente fechado (não eram usadas ventoinhas nem havia circulação de ar exterior no interior do dispositivo). Como Richard Morley explica:

"No fans were used, and outside air was not allowed to enter the system for fear of contamination and corrosion. Mentally, we had imagined the programmable controller being underneath a truck, in the open, and being driven around in Texas, in Alaska. Under those circumstances, we wanted it to survive. The other requirement was that it stood on a pole, helping run a utility or a microwave station which was not climate controlled, and not serviced at all" [32].

2.2.2 Componentes de um PLC

Um PLC é constituído principalmente por um *CPU*, uma memória, e os circuitos apropriados para as portas $I \backslash O$. Pode ser visto como uma caixa fechada com contadores, temporizadores e locais para armazenamento de memória. Não existem fisicamente mas são simulados e podem ser considerados como por exemplo contadores e temporizadores virtuais. Cada componente tem uma função específica:

- Inputs - Recebem sinais provenientes de dispositivos de entrada como interruptores, sensores, etc.

- Outputs - São responsáveis por actuar em máquinas, lâmpadas, etc.

- Contadores - Podem fazer contagens crescentes, decrescentes ou as duas. Já que são simuladas a sua velocidade é limitada. Alguns construtores incluem contadores de alta velocidade que são físicos.

- Temporizadores - Estes são muito variados. O mais comum é o *on-delay*. O incremento dos temporizadores pode variar da casa dos milissegundos aos minutos ou até horas.

- Armazenamento - Normalmente são registos. São usados temporariamente como armazenamento de informação que vai ser manipulada. [56]

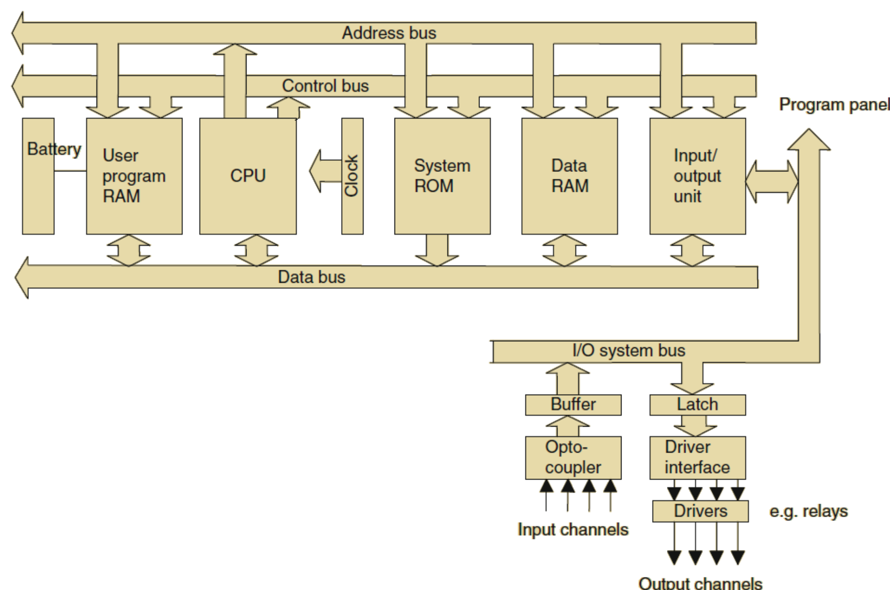

Figura 2.3: Arquitectura de um PLC. Retirado de: [13]

2.2.3 Ciclo de Operação

Um *PLC*, após inicializado, está continuamente a correr o programa que tem guardado em memória e a ser actualizado como resultado dos seus sinais de input. É uma operação cíclica. *PLCs* podem ser operados de forma a que cada input seja visto quando é chamado no programa, assim o seu efeito no programa é determinado e o output correspondente alterado. Este é o modo de operação de actualização contínua.

Esta maneira pode ser pouco eficiente em termos de tempo, especialmente se houver várias centenas de entradas e saídas. De forma a permitir uma execução mais rápida do programa, uma área específica da *RAM* é usada como *buffer* para guardar os valores das portas de saída e entrada. Cada entrada/saída tem um endereço nesta memória. No início de cada ciclo o *CPU* verifica todas as entradas e copia os seus estados para as posições respectivas na *RAM*. De seguida a lógica que equivale ao programa no PLC é executada. Os valores de saída resultantes vão sendo guardados na mesma área da *RAM* que as entradas. No final de cada ciclo todos os valores de saída são transferidos para os respectivos portos de saída. As saídas mantêm os seus estados até à próxima actualização. Este método é chamado de *mass I\O copying*. A sequência pode ser resumida no seguinte:

1. Verificações internas;
2. Verificar as entradas e copiar os seus valores para a *RAM*;
3. Busca, descodifica e executa (*fetch*, *decode and execute*) todas as instruções do programa por ordem, copiando os valores da saída para a *RAM*;
4. Actualiza todos os portos de saída;
5. Repete a sequência.

Na figura 2.4 encontra-se ilustrado o ciclo de operação.

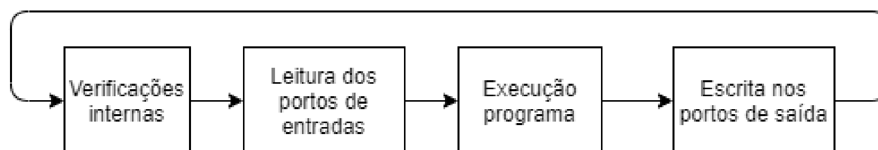

Figura 2.4: Ciclo de operação de um autómato.

O tempo que leva a efectuar um ciclo não é instantâneo, o que quer dizer que as entradas não são vistas sempre, são vistas periodicamente. O tempo de um ciclo está na ordem de 1 aos 50ms. Significa que os valores de saída e entradas na *RAM* são actualizados com estas frequências. Valores de entrada podem mudar a meio do ciclo, podendo ser perdidos. Em geral as entradas são alteradas num tempo maior que um ciclo. Caso não aconteça existem módulos para este tipo de ocasiões.

O tempo que um PLC demora a executar um ciclo depende dos seguintes factores:

- A velocidade do processador;
- O tamanho do programa a ser armazenado;
- O número de entradas e saídas a ser lidas e escritas;
- O número de funções do sistema a serem utilizadas. [13]

2.2.4 Linguagens de Programação Utilizadas

Existem 5 línguas de programação que fazem parte do *IEC* (*International Eletrotechnical Comission*) *Section 61131-3 Standard*. Esta norma permite algumas regras para os *PLCs* e as suas linguagens. Estas são:

- Ladder Diagram (LD) - é baseado na lógica de relés, que usa interruptores e relés mecânicos para controlar processos. É uma programação de blocos em degrau que são ligados através de conexões assemelhando-se aos esquemáticos da lógica de relés;
- Sequential Functional Charts (SFC)- Similar aos fluxogramas, esta linguagem usa degraus e transições até chegar ao resultado final;
- Function Block Diagram (FBD) - também é uma linguagem do tipo gráfica. Esta linguagem descreve uma função entre entradas e saídas por blocos ligados. Foi desenvolvida de modo a facilitar o uso de tarefas repetitivas, contactores, temporizadores, ciclos de PID, etc. Pode-se programar blocos dentro de blocos criando níveis de abstracção.
- Structured Text (ST) - é uma programação em texto e em alto nível tal como Basic, Pascal e "C". O código utiliza várias declarações escritas separadas com ponto e vírgula. Com estas declarações valores de saída e variáveis são alteradas. Podemos utilizar funções tais como FOR, WHILE, IF, ELSE e CASE.
- Instruction List (IL) - Como a linguagem ST, a IL também é uma programação em texto. Assemelha-se a Assembly pois usa também códigos mnemónicos (*mnemonic codes*) tais como LD (*Load*), AND, OR, etc.[19]

2.2.5 Simulação de PLC's em Ambiente Virtual

A simulação de PLC's é utilizada na etapa de desenvolvimento antes da fase de comissionamento dos programas e das linhas de produção que irão integrar as unidades industriais. São uma forma de redução de custos, pois, já não é necessário testar directamente no processo físico. É necessário não só desenvolver os produtos mais também as linhas de produção. Partindo da premissa de que linhas de produção são sistemas dinâmicos cujos estados mudam de acordo com o acontecimento de eventos, pode ser concluindo que possuem características de sistemas de eventos discretos [44].

Os PLC's guardam os valores de endereços de entrada e saída em memória, na chamada *I/O image table*. Enquanto o programa guardado no autómato é executado, o ciclo de operação também é continuamente executado, como já referido anteriormente.

Em [44] é apresentada uma arquitectura para um ambiente de programação de PLC's. Pode ser divida em duas camadas principais: a camada de modelo e a de aplicação. A camada de modelo é divida em três modelos: o modelo da unidade industrial que corresponde a um modelo virtual; modelo de controlo, onde se encontra o programa para o controlo do modelo anterior; o modelo para o mapeamento de endereços de entradas e saídas. O modelo da unidade industrial terá também modelos para cada elemento que o constitui, máquinas, tapetes etc.. . O modelo de controlo controla o modelo da unidade industrial através do modelo de mapeamento de entradas e saídas. A camada de aplicação providência duas interfaces com o utilizador: o simulador do PLC e visualização gráfica do ambiente industrial, onde podem ser verificadas as mudanças de estado. Na figura 2.5 é possível verificar tal arquitectura.

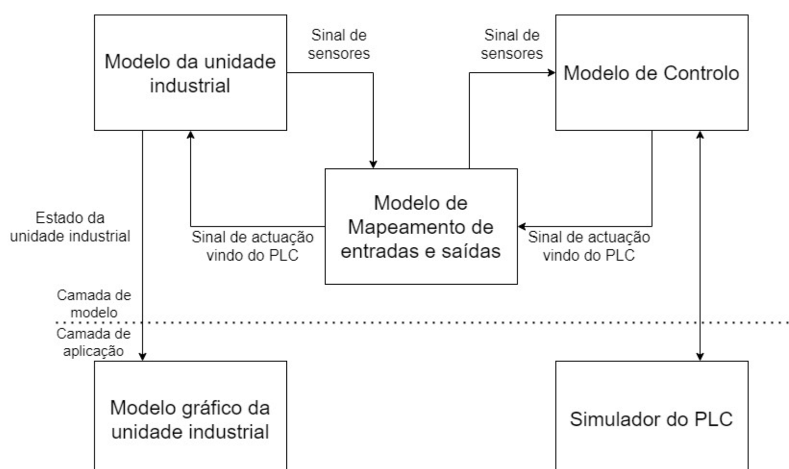

Figura 2.5: Arquitectura de um ambiente de simulação de PLCs. (Retirado e adaptado de: [44]).

A parte da modelação da lógica do ambiente industrial é crucial para o bom desempenho de simuladores. Em [54] e também em [44] é apresentado o formalismo DEVS (já antes referenciado). Este formalismo faz a especificação do modelo de eventos discretos de uma maneira hierárquica e modular. Neste formalismo são especificados dois tipos de modelos: modelo atómico e modelo acoplado. Os modelos acoplados mostram como os modelos atómicos são ligados de uma forma hierárquica.

Um modelo atómico é especificado por um conjunto de sete parâmetros:

$$M =< X; S; Y; \delta_{int}; \delta_{ext}; \gamma; t_a >$$

Que possuem os seguintes significados: conjunto de eventos de entrada; conjunto sequencial de eventos; conjunto de eventos de saída; funções para as transições internas;

funções para as transições externas; funções de saída; funções para a duração de cada evento.

Já um modelo acoplado corresponde a um conjunto de modelos atómicos, e no seu formalismo podem ser visto as relações entre estes modelos. Um modelo acoplado é especificado por também um conjunto de sete parâmetros;

$$DN =< X;Y;M;EIC;EOC;IC;SELECT >$$

Que possuem os seguintes significados: conjunto de eventos de entrada; conjunto de eventos de saída; conjunto de todos os modelos atómicos do modelo acoplado referente; relação externa de entrada entre um modelo atómico do modelo acoplado referente com o de outro acoplado; relação externa de saída entre um modelo atómico do modelo acoplado referente com o de outro acoplado; relações internas; selector de desempate.

Um exemplo pode ser um veiculo autónomo que possui duas tarefas: T1 (mover da posição p1 para a p2) e T2 (mover da posição p2 para p1). Cada tarefa é executada através de eventos exteriores, cujos sinais que identificam o acontecimento destes eventos são recebidos através dos sinais de entrada Sinal1 e Sinal2. Existem quatro estados possíveis: P1, executarT1, P2 e executarT2. Os estados P1 e P2 requerem eventos externos enquanto os outros necessitam de eventos internos que são os eventos disparados pelas duas tarefas [44]. O modelo atómico pode ser entendido como:

$$X = \{Sinal1, Sinal2\}$$

$$S = \{P1, executarT1, P2, executarT2\}$$

$$Y = \{T1Efectuado, T2Efectuado\}$$

$$\delta_{int}(executarT1) = P2$$

$$\delta_{int}(executarT2) = P1$$

$$\delta_{int}(P1, Sinal1) = executarT1$$

$$\delta_{int}(P2, Sinal2) = executarT2$$

$$(executarT1) = T1Efectuado$$

$$(executarT2) = T2Efectuado$$

$$t_a(executarT1) = Tempo1$$

$$t_a(executarT2) = Tempo2$$

2.3 Sistemas SCADA

Uma arquitectura genérica da indústria pode ser dada pela "pirâmide da automação"que pode ser vista na figura 2.6. É um modelo hierárquico que segmenta um sistema de manufactura completo em diversas camadas, em que cada camada tem em comum o tipo de informação que lida, sistemas e calendarização. Este modelo foi definido pela sociedade Internacional de Automação e tem o nome de ISA 95.

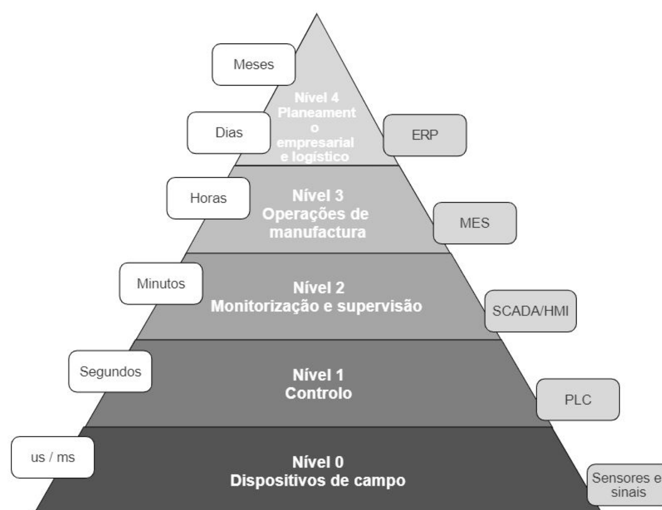

Figura 2.6: Pirâmide da automação segundo o modelo ISA 95 (Retirado e adaptado de: [57]).

A **camada de topo** preocupa-se com planeamento empresarial.

O **nível 3** controla as operações de manufactura. É neste nível decidido que processos devem ser executados e a ordem dos mesmos. Sistemas nesta camada têm o nome de Sistemas de Execução de Manufactura.

No **nível 2** estão os sistemas capazes de monitorizar, adquirir informação e supervisionar a evolução dos processos das instalações. Estes sistemas são os HMI (Interface Humano Máquina) e os sistemas SCADA (irão ser falados de seguida).

Já no **nível 1** estão os autómatos que controlam o processo situado na **base da pirâmide** [57].

Como já dito, a supervisão dos processos pode ser feitas por sistemas SCADA. Esta sigla vem de *Supervisory Control and Data Acquisition* ou Sistemas de Supervisão e Aquisição de Dados. Um sistema SCADA permite que o operador num local à parte de um processo amplamente distribuído consiga fazer alterações aos controladores, monitorizar informação em tempo real de processos, verificar a ocorrência de sinais de alarmes e emergências. Um dos benefícios é a possibilidade de supervisão remota, assim existem reduções de custo pois não é necessário efectuar visitas constantes ao terreno. Este benefício ainda acresce se as instalações se situarem numa zona remota.

Por exemplo, um grupo de várias pequenas barragens hidroeléctricas são ligadas e

desligadas consoante a necessidade do utilizador e estão situadas numa zona remota. É necessário um sistema central que monitorize em tempo real estas infraestruturas.

2.3.1 Enquadramento histórico

Os sistemas SCADA evoluíram através de quatro gerações:

A **primeira geração** é chamada de SCADA monolítica. O sistema era baseado em *mainframes* (computadores de grande porte com o objectivo de processar grandes quantidades de dados) e basicamente não existiam redes, pelo que não havia comunicação com outros sistemas. Poderiam existir WANs mas o único propósito seria ligar RTUs ao computador principal. Um dos problemas encontrados era a falta de protocolos padrão. Cada fabricante usava o seu.

A minimização dos sistemas e o desenvolvimento das tecnologias LAN foram os factores essenciais para a **segunda geração** que tem o nome de SCADA distribuído. Várias estações, mais baratas e de tamanhos reduzidos em relação à de primeira geração e também actuavam como mini computadores, conseguem comunicar entre si em tempo real. Estas estações serviam para comunicações entre outras estações e dispositivos de campo, para poder de processamento de informação recebida das instalações, como RTUs, servidores e HMIS.

Estes elementos podiam ser ligados através de uma LAN e os servidores comunicarvam através de WANs. Os dispositivos, incluindo os sistemas SCADA, ligados em LAN não conseguiam comunicar com dispositivos externos, porque, ainda diferentes fabricantes usavam diferentes protocolos. [55] A **terceira geração** dos sistemas SCADA, definido como SCADA em rede, foi idêntico à segunda geração excepto ter sido orientada para arquitecturas e protocolos padrão entre fabricantes. Com a utilização de protocolos de comunicação padrão os sistemas SCADA já conseguem funcionar em WANs distribuídas e não só em LANs.

Também na terceira geração, os fabricantes começaram a desenvolver plataformas e sistemas operativos dedicados ao SCADA. Um dos benefícios da terceira geração foi a comunicação da WAN através de IP (*Internet Protocol*). Também todos os componentes de um sistemas SCADA poderiam comunicar através de uma ligação de Ethernet [55].

Na actualidade estamos perante uma **quarta geração** de sistemas SCADA chamado de SCADA em serviços WEB. Desta forma a instalação dos sistema é simplificada e as HMIs podem estar presentes num nagedor de internet, assim é largamente acessível[29].

2.4 Compiladores e Linguagens de Alto Nível

O mundo de hoje depende das linguagens de programação, pois todo o *software* existente foi escrito em alguma linguagem de programação para poder ser executado. Mas antes de o programa conseguir ser corrido, tem de ser traduzido para uma linguagem que o computador consiga interpretar.

O *software* que consegue fazer esta tradução chama-se de compilador. Um compilador é um programa que consegue ler um programa de uma certa linguagem, linguagem fonte (*source language*), e traduzi-lo para um programa equivalente mas noutra linguagem, linguagem alvo (*target language*). Um papel importante do compilador é reportar qualquer erro detectado durante o processo de tradução do programa fonte.

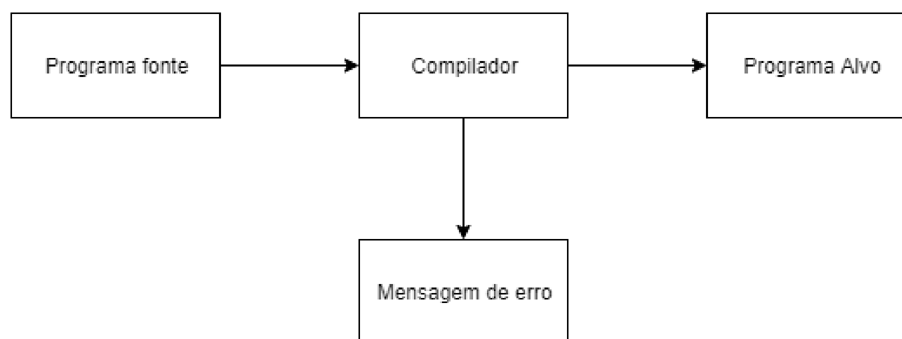

Figura 2.7: Um compilador. Retirado e adaptado de: [10]

2.4.1 Enquadramento histórico

Antes da década de 50 não existiam linguagens de programação de alto nível, o que levava a resolução de certos problemas de computação mais demorados. Grace Murray Hopper (1906-1992) foi talvez a primeira pessoa a acreditar que os computadores deveriam falar uma língua idêntica à que os humanos falam em vez de ser os humanos a compreender as instruções que computadores utilizam. Esta forma de pensar foi posta em prática em 1952 com a criação do compilador A-0 (*Arithmetic Language*, *version 0*) [14].

Em 1954, Hopper passou a ser a directora do departamento de programação automática para o UNIVAC (*UNIVersal Automatic Computer*). Este departamento lançou algumas das primeiras linguagens compiladas tais como Math-Matic (A-3) e Flow Matic (B-0, Business Language, version 0) [27].

Pouco tempo depois, foi entendido que a programação em computadores era demorada e cara, e a ideia de linguagens de programação idênticas às de humanos começou a surgir. Essas linguagens poderiam baixar os custos e aumentar a rapidez de desenvolvimento de *software*. Progresso tomou várias direcções. Para áreas mais cientificas, a IBM desenvolveu o Fortran (*formula translation*). Entregou o primeiro compilador nesta linguagem em 1957 [53].

Para informáticos, uma equipa desenvolveu a linguagem Algol e o seu primeiro compilador surgiu em 1958. Na área da inteligência artificial, durante a década de 60, a linguagem Lisp foi criada e um interpretador foi desenvolvido (o compilador só surgiu na década de 60). Nessa época, como na actualidade, aplicações para a área de negócios são relevantes. Hopper focou-se nesta àrea [53].

Em particular, a linguagem Flow-Matic reflectiu a necessidade que Hooper sentia que na área de negócios as linguagens deveriam parecer o mais possível ao inglês. Lançado

no final da década de 50 foi um sucesso imediato [53].

Reprogramar uma aplicação de computador poderia ser tão dispendioso como programá-la pela primeira vez. Por exemplo, comprar um novo computador poderia significar recomeçar a aplicação. Portanto em 1959, o comité de indústria e governo, com Hopper como conselheira técnica, foi formado com o objectivo de criar uma linguagem de programação orientada para o mundo dos negócios, chamada de Cobol. A primeira versão foi chamada de Cobol60, e compiladores começaram a surgir para vários computadores de empresas.

Como já referido, pode-se verificar que é menos dispendioso desenvolver e executar programas escritos em linguagens de alto nível, como também mais fácil de entender estas linguagens, mas para os computadores conseguirem resolver estes problemas é necessário traduzir os programas para linguagens de baixo nível e para tal são utilizados compiladores e interpretadores.

2.4.2 Vantagens e desvantagens do uso de compiladores e interpretadores

Um interpretador é outro tipo comum de processamento de linguagem. Em vez de produzir um programa alvo, um interpretador executa directamente as instruções especificadas.

Utilizar um compilador é muito mais rápido que interpretador a mapear os *inputs* para *outputs*. Porém, um interpretador é mais eficiente a diagnosticar erros que um compilador, porque executa o programa fonte declaração em declaração.

Para uma linguagem de programação ser **eficiente na execução** os programas escritos nessa linguagem precisam de ser executados num tempo razoável [30].

- Compiladores - Já que o código fonte é convertido em código máquina ou outra linguagem-fonte, a execução não necessita de mais traduções. Portanto, desta forma linguagens compiladas correm mais rapidamente e mais eficientemente;

- Interpretadores - O tempo usado por um interpretador para ler e executar o código fonte atrasa a execução do programa [30]. Também o código fonte é traduzido de novo sempre que é usado, [50], o que faz com que o processo de execução seja ainda mais demorado.

A **manutenção** pode ser mais fácil ou difícil dependendo da facilidade com que o um programa é percebido, reparado e melhorado.

- Compiladores - Como as linguagens que usam compiladores só precisam de ser traduzidas uma vez, o diagnóstico de erros só é detectado nesta etapa;

- Interpretadores - Já que o programa é sempre verificado quando corre o mesmo se aplica aos erros [30].

A **segurança** refere-se à habilidade de prevenção da modificação não autorizada do código fonte [30].

- Compiladores - O código fonte pode ser mantido em privado [39]. O processo de compilação parte o código fonte em código de muito baixo nível e as instruções são reordenadas [26]. Há uma distinção clara entre o ficheiro executável e o de fonte, o que o torna mais seguro [30].

- Interpretadores - Para linguagens interpretadas, não há distinção entre o ficheiro executado e o de fonte [26]. Assim estas são menos seguras que linguagens compiladas.

2.4.3 Estrutura de um compilador

Um compilador pode ser dividido em duas partes: análise e síntese.

A parte de análise é responsável por partir o programa fonte em bocados e formar uma estrutura gramatical. Esta estrutura é usada para criar uma representação intermédia do programa fonte. No caso de haverem erros é a parte de análise que tem de informar o utilizador para que este possa posteriormente corrigir o programa. Também reúne informação sobre o programa fonte e guarda-a numa estrutura de dados denominada tabela de símbolos, que é passada em conjunto com a representação intermediária para a parte de síntese.

A parte de síntese constrói o programa alvo a partir da representação intermédia e a informação da tabela. A parte de análise toma também o nome de *Front End* e a de Síntese *Back End*. Também pode ser verificado que um compilador opera em várias fases. Cada fase transforma uma representação do código fonte noutra, como pode ser visto na figura 2.8.

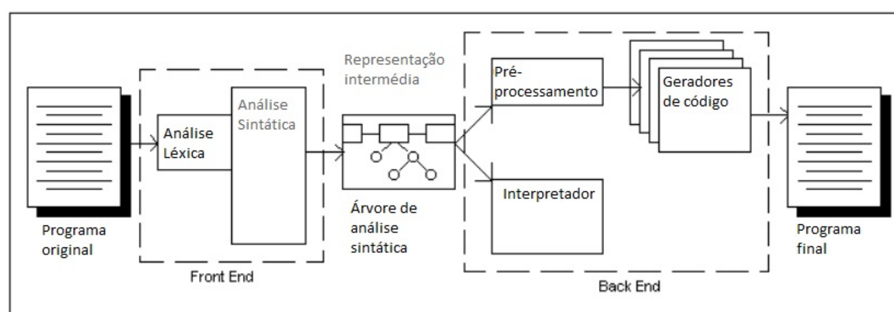

Figura 2.8: Anatomia de um compilador. Retirado de: [22]

A primeira fase de um compilador passa pelo *Tokenizer*, ou *Lexical Analyzer* (Analisador Léxico). Este lê os caracteres que constituem o código fonte e agrupam os caracteres em sequências denominadas por *lexemes*. Para cada *lexeme*, o analisador léxico produz um *token* que pode ser divido em duas partes. A primeira parte é o seu nome, uma simbologia abstracta que é usada durante a fase seguinte. A segunda é o valor atribuído para a entrada respectiva do *token* na tabela de valores (*symbol-table*) [10].

Os *tokens* são passados para o *parser*, ou *Syntax Analyzer* (Analisador Sintáctico). Este usa a primeira parte dos *tokens* para criar uma representação do programa fonte. Esta

representação é uma estrutura tipo árvore em que os nós interiores representam operações e os filhos dos nós representam os argumentos das operações. As seguintes fases utilizam esta estrutura para ajudar a analisar o programa fonte e gerar o programa alvo. [10]

A análise é feita pelo Analisador semântico (*semantic analyzer*), não apresentado na figura. Este analisador verifica se existe coerência semântica entre o programa fonte e a sua linguagem. Por exemplo, o compilador verifica se os operadores possuem operandos que coincidam [10].

Desta forma, é possível passar um programa de uma linguagem de programação qualquer para um programa equivalente mas numa linguagem de programação X.

2.5 Gamificação no Ensino da Engenharia

Nos últimos anos a inserção de elementos de jogos, segundo o conceito de gamificação, tem sido interessante em vários sectores como por exemplo a educação. Acredita-se que a educação a um nível de pré-graduação e profissional trará recompensas e é da responsabilidade das universidades de produzir novas metodologias e ferramentas com esta tecnologia para ajudar a crescer os estudantes, que num futuro irão integrar a indústria [36]. Uma definição de gamificação é a incorporação de elementos de jogos num contexto onde não existe jogos, de forma a aumentar o rendimento nessa área [17]. Já na área da educação este tema é correlacionado com a aprendizagem digital baseada em jogos. Pode ser definido como o uso de "mecanismos de jogos, estéticas e pensamento de jogo para envolver pessoas, motivar acções, promover o ensino e resolver problemas" [28].

Não é necessário desenvolver um jogo ou jogar os que são comercializados para aproveitar o tema da gamificação, pode ser apenas necessário retirar alguns aspectos tais como a obtenção de pontos, prémios e ter uma constante informação de como eficazmente as acções estão a acontecer como também a taxa de sucesso.

Especialmente para um público alvo mais novo o tema de jogos é lhes familiar trazendo assim uma maior facilidade em aprender com elementos de jogos ou jogando um. Segundo Angelos P Markopoulos na referência [36], escolher ambientes relacionados com jogos nas salas de aula não só desenvolve autonomia e competência como o sucesso de gamificar as salas de aula pode trazer benefícios tais como:

- O estudante sente que controla a sua aprendizagem;
- O conceito de tentativa e erro sem repercussões negativas é implementada criando a liberdade para falhar;
- É introduzido um ambiente mais dinâmico;
- É providenciado ao estudante opções de aprender o tema a ser leccionado;
- Os objetivos concluídos podem a passar a ser mais visíveis.

Como já dito este tema está imensamente ligado ao mundo digital. Portanto, outro benefício da gamificação é a importação dos temas aprendidos para um mundo virtual, assim é possível uma maior disponibilidade dos recursos chegarem aos estudantes especialmente para uma situação disruptiva, como por exemplo uma epidemia, onde o acesso a conteúdo de aprendizagem é dificultado à generalidade dos alunos. Quem queira aceder ao conteúdo pode o fazê-lo na segurança e conforto da sua secretária dentro de casa e sem sair dela, como também repetir temas leccionados, assim proporcionando uma aprendizagem mais eficaz. Uma metodologia digital também pode trazer uma redução de custos. Claramente também existem algumas críticas:

- A motivação para a aprendizagem pode estar mais ligada aos elementos do jogo retirando assim motivação daquilo que realmente interessa;
- Alguns docentes acham que a gamificação é uma maneira de aprendizagem menos séria;
- Pode ser considerado demasiado fácil sendo irrelevante;
- A gamificação pode não responder a todos os problemas do estudante e do tema a aprender;
- O tempo perdido em modificar e suportar um tema com gamificação ou simplesmente pode ser perdido mais tempo lecionando com um tema gamificado;

2.5.1 Práticas para a Gamificação

A introdução de mecanismos de jogo são uma forma básica de gamificar uma aplicação, um processo ou até um produto. Mecanismos de jogo entendem-se como as várias acções, comportamentos e mecanismos de controlo efectuados por um jogador num contexto lúdico [34]. Tendo em conta que o público alvo nesta situação são estudantes de engenharia é necessário ter tal noção em consideração na implementação da gamificação. Os elementos de um jogo que podem ajudar são os seguintes [36]:

Pontos são uma característica fundamental. Um sistema de pontos faz com que o jogador consiga saber o seu estado actual, se o que está a fazer está a dar frutos ou não, também é implementado para o utilizador saber o progresso que já fez;

Tabelas de classificação são um dos aspectos competitivos de um jogo. Uma tabela de classificação guarda e ordena os pontos de cada jogador, assim é possível saber o quanto um jogador está melhor ou pior face aos outros;

Distinções marcam a conclusão de metas. Se forem visíveis para o utilizador enquanto ainda não a ganhou é uma forma de saber quais são as finalidades daquilo que foi desenhado. São formas de encorajar a aproximação social;

Níveis são divisões de um jogo, o mesmo pode ser aplicado a um projecto, dividir um projecto torna algo que à primeira vista pareça interminável para várias etapas exequíveis. É aplicado o conceito de dividir e conquistar;

Integração é um aspecto crucial para novos utilizadores. Normalmente é desenvolvido um nível inicial, um tutorial, para que sejam entendidos os vários comandos e o ambiente;

A implementação de **desafios** faz direccionar o jogador para o caminho certo. É preciso dizer ao utilizador o que é necessário fazer para chegar ao final do que foi construído como também é possível criar desafios intermédios;

Colaboração colectiva é a necessidade de dois ou mais jogadores entre ajudarem-se para conseguir superar os desafios.

Em vez de serem aplicados elementos de jogos a uma situação de aprendizagem outra abordagem pode ser o contrário, pegar num jogo e modificá-lo de forma a ter uma componente de aprendizagem ou, simplesmente, fazer um jogo. Alguns géneros para jogos podem ser: **puzzle**, **aventura**, **simulação**, **estratégia**, **estratégia em tempo real**, etc.

2.5.2 Simulações Lúdicas

Um dos temas abordados em engenharia e mais precisamente em Engenharia Electrotécnica e de Computadores é o de automação industrial, e para este caso pode fazer sentido a segunda abordagem de gamificação onde pode ser feito um jogo de simulação para sistemas de automação por exemplo um simulador de linhas de montagem. Um jogo tipo simulação faz sentido nos tempos que correm, pois, os jogos são apresentados com gráficos de alta qualidade e possuem incorporados bons motores de física.

James R. Parker em [45] diz que existem características nos jogos que os fazem tornar uma subclasse das simulações. É dito que historicamente as simulações produziam apenas saídas em forma numéricas enquanto os jogos tinham um ambiente graficamente rico. Nos dias que correm, tal não acontece, já que muitas simulações produzem visualizações dos sistemas modelados recorrendo por vezes a motores de jogos. As partes mais externas entre jogos e simulações convergiram.

É possível converter uma simulação num jogo. Uma simulação básica calcula o estado do sistema modelado durante intervalos de tempo. Normalmente o resultado seria num formato numérico e estaria representado num gráfico, e este gráfico seria toda a parte de visualização disponível. Esta simulação poderia ser convertida para uma simulação lúdica. Este tipo é uma simulação interactiva visual incluindo assim alguns aspectos, mas que tem em falta elementos de jogos, como os mencionados na subsecção anterior, sendo um dos mais importantes, um objectivo bem definido. Como a figura 2.9 ilustra, um jogo pode fazer parte de um grupo mais vasto que são as simulações lúdicas.

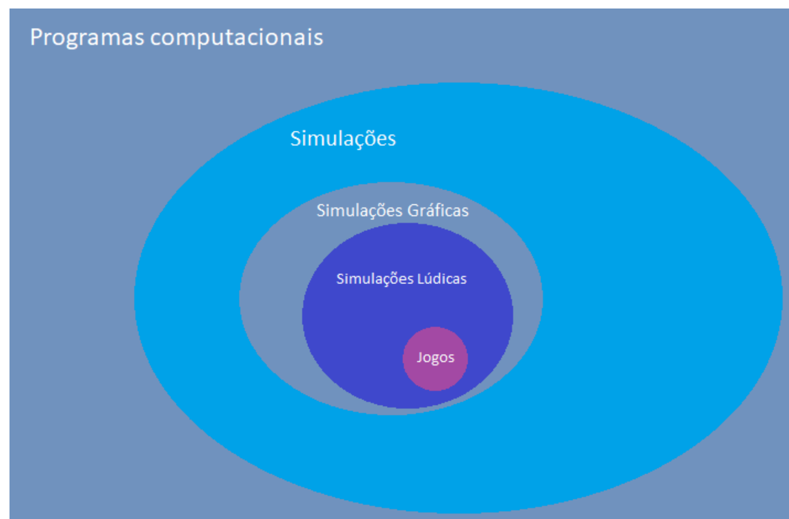

Figura 2.9: Um jogo pode ser uma simulação lúdica. (Retirado e adaptado de: [45]).

Uma simulação lúdica dá ênfase à construção de modelos e a execução de simulações, a um nível de educação é semelhante à gamificação mas não possui o mesmo nível de coerência na avaliação através de respostas correctas, aborda aspectos de descoberta e emulação, dá ênfase à experiência pessoal, ao teste de hipóteses e ao raciocínio científico.

2.6 Sistemas Existentes

Nesta secção, identificam-se alguns sistemas considerados relevantes para o desenvolvimento deste projecto. O objectivo é criar um referencial que permita comparar o projecto desenvolvido com sistemas existentes contribuindo para a identificação de lacunas e questões em aberto.

A plataforma TIA Portal STEP 7 foi desenvolvido pela Siemens AG com o intuito de poder configurar e programar os controladores SIMATIC utilizando as linguagens da norma IEC 61131-3 como também é possível simular e testar o comportamento dos projetos implementados. Uma alternativa independente a plataformas de programação de PLCs é o XSOFT da Codesys, onde, por exemplo, é muito fácil de testar simples códigos com a ajuda de sinais luminosos, botões ou alarmes já embutidos na plataforma.

O motor de jogos Unity criado pela Unity Technologies não é utilizado apenas para jogos, já é possível fazer simulações e mimificar sistemas de automação industrial. Estas funcionalidades são pacotes adicionais e desenvolvidos por outras empresas, não fazem parte deste motor de jogos que para desenvolvimento académico ou pessoal é grátis.

Um destes pacotes tem como nome OBI [5] e é desenvolvido pela Virtual Method. Este pacote é um motor de física em tempo real desenvolvido para o Unity. Pode ser dividido em três categorias: Obi Cloth, Obi Rope e Obi Fluid. O primeiro serve para simulações de objectos tipo tecidos, nestas simulações para corpos macios são usados modelos aerodinâmicos. O segundo serve para criar cordas realistas que interagem entre

si e com o ambiente à volta. Por fim, o Obi Fluid, que é utilizado para criar simulações em 2D e 3D de fluídos. As propriedades físicas de fluídos como tensão superficial, vorticidade e viscosidade podem ser alteradas de acordo com as necessidades do utilizador. Os fluidos podem aderir às superfícies, formar gotas, dividir-se e fundir-se. O pacote OBI é destinado para pequenas simulações.

O pacote para Unity, game4automation [6] é uma solução para simulação de processos Industrias ou para implementação de uma HMI em 3D. Estão implementados vários modelos tais como robôs com garras e tapetes rolantes. É possível ligar a controladores reais.

A plataforma Factory I/0 [7], parecido com o anterior, mas muito completo, não é um pacote para Unity mas é já um produto final utilizando este motor de jogos. Factory I/O é um ambiente de simulação em 3D para a indústria de automação com o intuíto de facilitar a aprendizagem para este tipo de tecnologias de automação. É possível construir rapidamente uma fábrica virtual utilizando uma selecção de elementos utilizados na indústria como: robôs, tapetes rolantes, sensores, botões, sirenes, tanques etc. Também é possível a comunicação com controladores e micro controladores. Apresenta uma interface fácil de utilizar, e, cenários já implementados.

O *software* FluidSim [18] é uma ferramenta de ensino para simulação de circuitos pneumáticos. O utilizador consegue desenhar circuitos eletropneumáticos, sendo esses circuitos simulados com base nos modelos físicos dos componentes. É uma ferramenta de fácil de uso e intuitiva.

2.7 Trabalho Relacionado

Nesta secção, são mencionados alguns trabalhos académicos com o objectivo de recolher elementos que sirvam como ideias para a arquitectura deste projecto.

Foi necessário estudar e analisar vários trabalhos existentes, e, em certas partes similares a este para tentar perceber aspectos importantes e conseguir construir uma arquitectura. São apresentados os aspectos mais relevantes de vários trabalhos realizados e publicados.

Em 2010, F. Narciso e colaboradores [40] tiveram um artigo publicado com o título "A Syntactic Specification for the Programming Languages of the IEC 61131-3 Standard". É definida uma CFG (*Context Free Grammar*), portanto são apresentadas as regras de produção, para o desenvolvimento de um tradutor que consiga gerar código numa linguagem de alto nível (linguagem de programação C) a partir de um programa escrito numa das cinco linguagens de programação do stantard IEC 611131-3 (IL, ST, LD, FBD). O tradutor é possível de implementar, porque pode ser criado um analisador léxico, pois as expressões regulares que descrevem uma linguagem de programação são um caso particular da CFG. Um analisador sintáctico também pode ser criado através desta gramática. Este consegue determinar se as frases escritas pertencem ou não às linguagens de programação propostas pela CFG.

Em 2014, João Pecoreli realizou a sua dissertação de tese para a obtenção do grau de mestre [46] que teve como objectivo um simulador em tempo real capaz de receber código em texto estruturado para de seguida implementar no autómato TSX Micro 3721 da Schneider. A construção deste simulador foi dividido em duas partes: a construção de um compilador e interpretador de linguagens de programação e a segunda parte uma interface gráfica. De forma a gerar as regras formais gramaticais e de produção são usados os sistemas Lex e Yacc. O Lex gera um analisador léxico e o Yacc, posteriormente, forma as regras de produção e a respectiva estrutura hierárquica do programa. Depois desta estrutura ser interpretada é possível traduzir ou compilar um programa em linguagem em texto estruturado para outra linguagem.

Em 2017, L. Brito Palma, V. Brito, J. Rosas e P.Gil, apresentam num artigo [43], as principais funcionalidades de um Simulador de PLC's alojado na web com um ambiente de programação para a linguagem em Texto Estruturado. O simulador possui canais de entrada e saída quer analógicos, quer digitais. Através destes canais é possibilidade a interacção com um modelo autor-regressivo e assim controlá-lo. O simulador de PLC encontra-se no lado do servidor, e, possui um analisador sintáctico (*parser*) implementado capaz de receber várias instruções de texto estruturado. Do lado do cliente existe uma interface gráfica para que o utilizador consiga introduzir o seu programa, declarar endereços e comandar o autómato. De modo a visualizar a evolução do processo a controlar é disponibilizada uma janela com um gráfico, actualizado em tempo real, onde é possível verificar a entrada e saída do processo, como também a referência para que o processo tende. O projecto apresentado mostra grande potencial na área de e-learning.

2.8 Conceitos Fundamentais

2.8.1 Especificação da Linguagem Texto Estruturado

A linguagem Texto Estruturado assemelha-se à linguagem Pascal ou C/C++. Programas nesta linguagem são constituídos por um conjunto de declarações separadas por ponto e vírgulas. As declarações são compostas por declarações pré-definidas e sub-rotinas e são utilizadas com a finalidade de mudar valores a variáveis. As variáveis podem ser variáveis internas, de entrada ou saída.

Algumas das principais instruções são as seguintes:

- Instruções binárias;
- Instruções aritméticas e lógicas;
- Comparações numéricas;
- Instruções de manuseamento de tempo (temporizadores).

2.8.1.1 Tipos de Dados Básicos

Dependendo da necessidade do programador, a linguagem Texto Estruturado apresenta vários tipos de dados básicos desde tipos de dados com o tamanho de um bit até tipos que conseguem lidar com valores decimais e negativos. A tabela em baixo apresenta esses tipos de dados básicos.

Tabela 2.1: Tipo de dados básicos da linguagem Texto Estruturado. Retirado de: [20].

Tipo de dado básico			Game de valores
Binário	Boolean	BOOL	0 ou 1
Inteiro	Word [Sem sinal]	WORD	0 a 65535
	Double Word [Sem sinal]	DWORD	0 a 4294967295
	Word [Com sinal]	INT	-32768 a 32767
	Double Word [Com sinal]	DINT	-2147483646 a 2147483647
Real	Float [Precisão simple]	REAL	-2^128 a -2^-126,0, 2^-126 a 2^128
	Float [Precisão dupla]	LREAL	-2^128 a -2^-126,0, 2^-126 a 2^1284

2.8.1.2 Variáveis

É possível utilizar variáveis para aceder a posições de memória, e assim, permitindo guardar informação dentro do PLC como também ler e escrever nos módulos de saída e entrada.

As variáveis começam com o carácter '%' seguido de um caracter para identificar o tipo de variável e ainda outro carácter que identifica o tipo de dado básico, como pode ser verificado na tabela 2.2.

Tabela 2.2: Nomenclatura dos vários tipos de variáveis.

Sinal inicial	Identificação de memória	Tipo da Var.	Tipo de dados
%	M (Acesso à memória) I (Entrada física) Q (Saída Física)	X	BOOL
		W	INT e WORD
		D	DINT, DWORD
		T	TIME, DATE, TOD e DATE_AND_TIME
		R	REAL

Se for uma variável de entrada ou saída terá de conter o seu endereço físico. Alguns exemplos de variáveis:

- %IX0.3 – Variável booleana de entrada no módulo 0 e entrada 3;
- %QW1.2 – Variável word de saída no módulo 1 e entrada 2;
- %MR7 – Variável real interna do controlador.

Enquanto variáveis de entrada só podem ser lidas, variáveis de saída como também de memória interna podem ser lidas e escritas.

Os exemplos seguintes seguem o padrão IEC 61131-3, o que traz portabilidade ao código. Contudo, certas tecnologias alteram aspectos da linguagem de programação, como por exemplo as variáveis internas, podem ser acedidas apenas chamando um conjunto de caracteres alfanuméricos e não como é convencionado.

2.8.1.3 Declarações Condicionais

De forma a um programa em Texto Estruturado poder executar certos blocos de código, ou até saltar blocos, a linguagem Texto Estruturado possui declarações condicionais.

A mais popular é a *if-then-else*. Se a condição apresentada possuir o valor de verdade, então a primeira secção irá ser executada (bloco de código a seguir ao *then*). Caso a condição for falsa, o segundo bloco de código é executado (bloco de código a seguir ao *else*). Esta segunda secção da declaração pode não ter sido implementada, resultando em que por vezes nenhum bloco de código desta declaração seja executado.

O programador pode querer ter mais do que duas opções de blocos de códigos, para uma melhor implementação pode ser implementado o *elsif* na declaração *if-then-else*. Se a primeira condição não for verdade uma segunda condição é testada (a condição correspondente ao *elsif*). É de frisar que é possível declarar um número de *elsif's* que o programador ache necessário. O *else* também é opcional.

Na listagem 2.1 pode ser visto código genérico para a declaração condicional como também um exemplo prático.

Listagem 2.1: Declaração if

```
1  /* Código genérico */
2  IF <condiçoes> THEN
3    <bloco de código>
4  ELSIF <condiçoes> THEN
5    <bloco de código>
6  ELSE
7    <bloco de código>
8  END_IF
9
10 /* Exemplo */
11 IF temp < 17
12 THEN heating_on := TRUE;
13 ELSIF temp > 25
14 THEN open_window := TRUE;
15 ELSE heating_on := FALSE;
16 END_IF;
```

A outra declaração condicional é a declaração *CASE*. Esta, testa o valor de uma variável, inteira, e dependendo do valor desta variável é executado um bloco de código específico. Se o valor da variável não corresponder com nenhum dos que foram programados, é possível que ainda seja executado um bloco de código (bloco de código referente ao *ELSE*,

caso seja programado). Na listagem 2.2 pode ser visto código genérico para a declaração *CASE* e um exemplo prático.

Listagem 2.2: Declaração case

```
/* Código genérico */
CASE <variável> OF
<valor> : <bloco de código>
<valor> : <bloco de código>
<valor> : <bloco de código>
...
ELSE <bloco de código>
END_CASE;

/* Exemplo */
CASE PumpState OF
0: StateDescription := "0";
1: StateDescription := "1";
2: StateDescription := "2";
ELSE StateDescription := "3";
END_CASE;
```

2.8.1.4 Declarações Iterativas

As declarações iterativas ou apenas ciclos, são usadas quando é necessário repetir uma ou mais declarações, sendo que o número de vezes que este conjunto de declarações são executados pode ser controlado por um estado de uma variável ou uma condição. Na linguagem em Texto Estruturado, existem três declarações iterativas:

- *FOR...DO*;
- *WHILE..DO*;
- *REPEAT...UNTIL*.

A declaração iterativa *FOR...DO* permite um conjunto de declarações serem executadas repetidamente, dependendo do valor de uma variável inteira. A declaração *FOR...DO* genérica é descrita em 2.3.

Listagem 2.3: Declaração for

```
/* Código genérico */
FOR <variável> := <inteiro> TO <inteiro> BY <inteiro> DO
<declaraçoes>
END_FOR

/* Exemplo */
FOR %MW0 := 1 TO 10 BY 1 DO
%QW1.2 := %IW0.1;
END_FOR
```

No exemplo em cima, a atribuição de valores ao endereço de saída é realizada dez vezes. A variável de controlo é a %MW0 e começa com o valor inicial 1, depois de cada execução é testado se o valor da variável chegou ao valor limite, neste caso 10. Se chegar ao valor limite, o ciclo termina e o programa continua executando as declarações seguintes, se não chegar ao valor limite a variável é incrementada em 1 e executa mais uma vez o conjunto de declarações, repetindo-se o processo.

A declaração *WHILE...DO* permite também que uma ou mais declarações sejam executadas enquanto uma expressão booleana tenha o valor de verdade. Na listagem 2.4 é apresentada a declaração genérica como também um exemplo:

Listagem 2.4: Declaração while

```
1  /* Código genérico */
2  WHILE <condiçao> DO
3  <declaraçoes>
4  END_WHILE
5
6  /* Exemplo */
7  WHILE %IX0.0 AND%IX0.1
8  %MW0 := %MW0 + 1;
9  END_WHILE
```

Como pode ser verificado no exemplo em cima, primeiro é testada a condição booleana. Se tiver o valor de verdade, a declaração é executada e volta a testar a condição, repetindo-se o procedimento. Caso a declaração tenha o valor de falso o programa sai do ciclo.

Por último, a declaração *REPEAT...UNTIL* permite a execução de um conjunto de declarações, esta repetição apenas acontece se uma condição booleana declarada pelo utilizador possuir o valor de verdade. Na listagem 2.5 pode ser vista a declaração genérica e um exemplo:

Listagem 2.5: Declaração repeat

```
1  /* Código genérico */
2  REPEAT
3  <declaraçoes>
4  UNTIL <condiçao>
5  END_REPEAT
6
7  /* Exemplo */
8  REPEAT
9  %MW0 := %MW0 + 1;
10 UNTIL %IX0.0 AND%IX0.1
11 END_REPEAT
```

As declarações *WHILE...DO* e *REPEAT...UNTIL* são idênticas exceto que a primeira verifica a condição e dependendo do valor do teste da condição executa o conjunto de declarações enquanto a declaração *REPEAT...UNTIL* executa primeiro o conjunto de declarações e só depois é que testa a condição para saber se continua ou não no ciclo. A

declaração *REPEAT... UNTIL* executa sempre pelo menos uma vez o conjunto de declarações especificadas pelo utilizador.

2.8.2 Temporizadores

Pode existir a necessidade de contar o tempo. Por exemplo, um motor ou uma bomba de água poderá necessitar de trabalhar apenas durante um período específico. Para providenciar tal capacidade de controlo, os PLC's possuem temporizadores capazes de contar segundos ou até milissegundos. Uma abordagem comum é a de considerar o comportamento dos temporizadores como o de relés com bobines. A sua alimentação resulta no fecho ou da abertura dos contactos durante um período de tempo definido. O temporizador é assim utilizado para controlar um outro circuito ou então outra parte do código de um programa em Texto Estruturado. Os temporizadores podem operar em três modos de utilização:

- TON – para gerir atrasos no arranque;
- TOF – para gerir atrasos no desligar;
- TP – gera um pulso com uma duração definida.

Na tabela 2.3 é possível observar as características e variáveis dos vários modos dos temporizadores.

Tabela 2.3: Características e variáveis dos vários modos. Retirado de [21]

Número do temporizador	%TMi	
Mode	TON TOF TP	
Tempo Base	TB	1 min (por padrão), 1s, 100ms, 10ms. Quanto mais pequeno o tempo base mais preciso será o temporizador.
Valor atual	%TMi.V	Word que é incrementado de 0 a %TMi.P quando o temporizador está a correr. Pode ser lido e testado, mas não escrito pelo programa.
Valor predefinido	%TMi.P	0 <= %TMi.P <= 9999. Word que pode ser lido, testado e escrito pelo programa. Por padrão é definido com o valor de 9999. O período ou intervalo gerado é igual a %TMi.P x TB.
Valor de entrada	IN	O temporizador começa com uma mudança de TMi.IN ascendente (TON ou TP) ou descendente (TOF).
Valor de saída	%TMi.Q	Tem o valor de 1 dependendo do temporizador referente.

O temporizador em modo TON inicia em uma mudança de estado ascendente de IN. O valor de %TMi.V aumenta de 0 até %TMi.P em 1 por cada pulso com tempo TB. A saída %TMi.Q muda para 1 quando %TMi.V atinge o valor de %TMi.P e continua com o valor 1 enquanto IN tiver também o valor de 1.

O temporizador em modo TOF funciona de forma diferente. O valor de %TMi.V é igualado a 0, numa mudança de estado ascendente de IN (mesmo se o temporizador estiver a correr). O temporizador começa numa mudança de estado descendente IN. O valor de %TMI.V é incrementado em 1 até %TMi.P em cada pulso de duração TB. %TMi.Q passa para 1 quando é detetada uma mudança de estado ascendente em IN, e, muda para 0 quando %TMi.V alcança o valor de %TMi.P.

Por último, é apresentado o modo TP. O temporizador inicia numa mudança de estado ascendente de IN (caso ainda não tenha sido incializado), o valor de %TMi.V vai incrementando de 0 até %TMi.P a cada pulso de tempo base. A variável %TMi.Q muda para 1 quando o temporizador começa e retorna 0 quando o valor atual fica igualado a %TMi.P. Quando IN e %TMi.Q têm o valor de 0, %TMi.V toma também o valor 0. Este modo não pode ser reiniciado.

Na listagem 2.6 pode ser verificado um exemplo do modo TON:

Listagem 2.6: Exemplo de um temporizador TON

```
1 IF RE %IX1.1 THEN
2   START %TM1;
3 ELSIF FE %I1.1 THEN
4   DOWN %TM1;
5 END_IF
6 %Q2.3 := %TM1.Q;
```

A instrução START %TMi gera uma mudança de estado ascendente em IN. A instrução DOWN %TMi gera uma mudança de estado descendente em IN. As instruções "RE <variável>"e "FE <variável>"servem para testar mudanças de estado ascendente ou descendente da variável em questão.

2.8.3 Caracterização das ferramentas Flex e Bison

É necessário criar uma aplicação que seja capaz de receber código em Texto Estruturado e consiga interpretá-lo para que seja corrido numa outra linguagem, neste caso C/C++. Para a parte da tradução são necessárias as ferramentas Flex e Bison.

A ferramenta Flex é responsável por receber padrões, escritos pelo desenvolvedor, e produzir um analisador léxico ou *scanner* escrito em C/C++. Por sua vez, o scanner converte as cadeias de caracteres, que na sua totalidade formam um programa escrito em Texto Estruturado, em símbolos (em inglês, *tokens*), que são representações numéricas de cada padrão que forma uma ou mais cadeia de caracteres.

O Bison irá gerar um analisador sintáctico ou parser através de um ficheiro onde estão implementadas as regras gramaticais de Texto Estruturado. O parser está incumbido de

usar as regras gramaticais para analisar os símbolos, passados pelo analisador léxico, e construir uma árvore de análise sintáctica. Esta árvore de análise sintáctica impõe uma estrutura hierárquica nos símbolos. Depois de a árvore ser construída, isto é, houver uma estrutura com a mesma semântica do código introduzido pelo utilizador, é necessário caminhar pelos seus nós para interpretar o que originalmente era um programa em Texto Estruturado.

Para produzir uma árvore, são necessários no mínimo dois ficheiros, um ficheiro com a extensão .y e outro .l. O .y é passado ao Bison que por sua vez irá produzir um ficheiro .tab.c que possui o analisador sintáctico e a função yyparse() e outro ficheiro com a extensão .tab.h onde estão as definições dos símbolos. Já o ficheiro. l ao passar com o .tab.h pelo Flex irá produzir um ficheiro com a extensão .yy.c que possui analisador léxico e a função yylex(). Estes ficheiros são compilados e ligados entre si para formar um programa executável. De forma a correr o tradutor, na função main() pode ser chamado a função yyparse() que por sua vez chamará a função yylex().

Na figura 2.10, é possível ver uma imagem ilustrativa do processo de tradução.

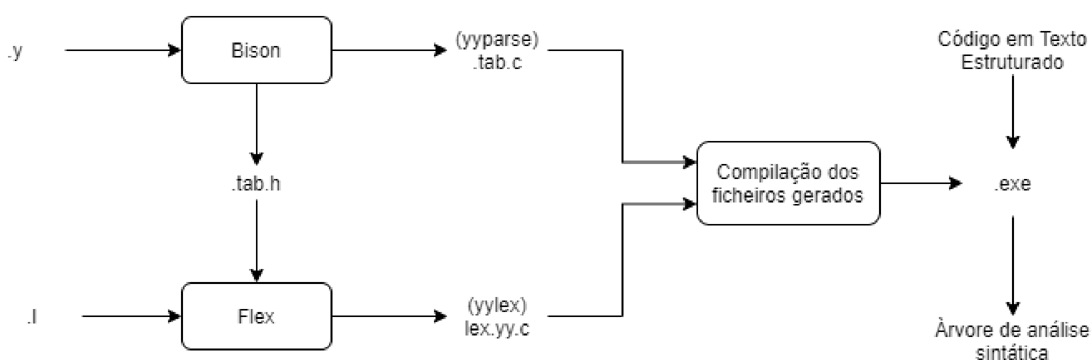

Figura 2.10: Passos para a formação de uma ASB utilizando as ferramentas Flex e Bison. Retirado e adaptado de [42]

2.8.3.1 Flex

Como referido anteriormente a primeira parte do processo de tradução diz respeito à análise léxica. É nesta fase que o código introduzido pelo utilizador é lido e cada conjunto de caracteres que o representa é convertido para uma representação numérica. Estes conjuntos de caracteres conseguem ser identificados através de expressões regulares. Uma expressão regular é um padrão capaz de identificar conjuntos de caracteres específicos. Cada expressão regular tem associado uma representação numérica.

Com as expressões regulares, é possível identificar os padrões que um texto possui, tais como variáveis, números inteiros e reais, operadores, condições ou outras declarações. A tabela 2.4 possui operadores usados e exemplos essenciais das expressões regulares.

O ficheiro associado ao Flex é dividido em três secções: definições, regras e sub-rotinas, divididas entre si por %%. A parte das definições tem como propósito definir bibliotecas e variáveis usadas no resto do ficheiro. A parte das regras é onde são definidas as expressões

Tabela 2.4: Exemplos de expressões regulares.

Padrão	Correspondência
a|b	**a** ou **b**
(ab)+	Uma ou mais cópias de **ab**
abc	**abc**
abc*	**ab abc abcc abccc abcccc ...**
"abc*"	**abc***
abc+	**abc abcc abccc abcccc ...**
a(bc)+	**abc abcbc abcbcbc abcbcbcbc ...**
a(bc)?	**a abc**
[abc]	**a** ou **b** ou **c**
[a-z]	Qualquer letra de **a** até **z**
[a\-z]	**a** ou **–** ou **z**
[-az]	**-** ou **a** ou **z**
[A-Za-z0-9]+	Um ou mais caracteres alfanuméricos
[\t\n]+	Espaço em branco
[^ab]	Qualquer caracter excepto **a** ou **b**
[a^b]	**a** ou ^ou **z**

regulares e as respectivas acções quando são identificados os padrões. A parte das sub-rotinas contem funções de C/C++, como por exemplo, pode ser definida a função main() que é obrigatória nestas linguagens de programação. O Flex possui diversas variáveis definidas que ajudam no processo. Na tabela 2.5 é possível visualizar tais variáveis.

Tabela 2.5: Variáveis e funções da ferramenta Flex

Nome	Funcionalidade
int yylex(void)	Chama o Flex, retorna um símbolo
char *yytext	Apontador para o conjunto de caracteres identificado
yyleng	Tamanho para o conjunto de caracteres identificado
yylval	Valor associado ao símbolo
int yywrap(void)	Avisa se toda a entrada foi lida ou não
FILE *yyout	Ficheiro de saída
FILE *yyin	Ficheiro de entrada
INITIAL	Condição inicial para começar
BEGIN condition	Mudar condição inicial
ECHO	Para escrever o conjunto de caracteres identificado

Na listagem 2.7 é apresentado um exemplo de um programa que apenas usa a ferramenta Flex. Sempre que é lido um operador ou um número inteiro uma mensagem de texto aparece a dizer que caracter ou conjunto caracteres foram escritos. Ao sair do programa, é escrito no ecrã quantas linhas foram escritas pelo utilizador. Na figura 2.11 podem ser verificados os comandos utilizados para a partir do ficheiro lexer.l gerar um ficheiro .c e de seguida compilá-lo e executá-lo.

Listagem 2.7: Exemplo de um ficheiro para geração de um analisador puramente léxico

```
/*** Ficheiro lexer.l ***/

   /* Definiçoes */

%option noyywrap
%{
   int nline = 0;
%}


%% /* Regras */

[-+*/=] { printf("Operador␣lido:␣%s\n", yytext); }
[1-9][0-9]* { printf("Inteiro␣lido:␣%d\n", atoi(yytext)); }
\n { nline++; }
[ \t] { /* ignora espacos em branco */ }
. { printf("Caracter␣nao␣conhecido:␣%c\n", *yytext); }

%% /* Sub-rotinas */
int main(void) {
   yylex();
   printf("Numero␣de␣linhas␣escritas:␣%d\n", nline);
   return 0;
}
```

```
C:\Users\Guilherme Gil>cd C:\TESE\Escrita\Conceitos Fundamentais\Flex e Bison\exemplo_FLEX

C:\TESE\Escrita\Conceitos Fundamentais\Flex e Bison\exemplo_FLEX>flex lexer.l

C:\TESE\Escrita\Conceitos Fundamentais\Flex e Bison\exemplo_FLEX>gcc lex.yy.c

C:\TESE\Escrita\Conceitos Fundamentais\Flex e Bison\exemplo_FLEX>.\a
2+2
Inteiro lido: 2
Operador lido: +
Inteiro lido: 2
3*4
Inteiro lido: 3
Operador lido: *
Inteiro lido: 4
f
Caracter nao conhecido: f
^Z
Numero de linhas escritas: 3

C:\TESE\Escrita\Conceitos Fundamentais\Flex e Bison\exemplo_FLEX>
```

Figura 2.11: Exemplo da geração de um executável para um analisador puramente léxico

2.8.3.2 Bison

O analisador sintáctico, implementado através da ferramenta Bison, tem como objectivo verificar se a disposição dos símbolos enviados pelo analisador léxico formam uma sintaxe coesa. Portanto, esta ferramenta é responsável por lidar com a gramática do texto

a traduzir. Para tal, é necessário que o Flex, ao identificar padrões, envie para o Bison símbolos representativos do que foi lido.

A gramática no Bison é descrita usando a notação BNF. Foi criada por John Backus e Peter Naur para descrever a linguagem ALGOL. Esta notação pode ser usada para expressar Gramática Livre de Contexto (GLC), que é um modo convencionado de descrever a gramática de diversas linguagens de programação.

Os meta-símbolos que são usados na notação BNF são:

- ::= – representa "é um/a";
- | - pode ser lido como "ou";
- <> - indicam uma regra.

Na listagem 2.8 é possível verificar um exemplo que soma e multiplica números.

Listagem 2.8: Exemplo da notação BNF

```
<E> ::= <E> + <E> Regra 1
    | <E> * <E>  Regra 2
    | id  Regra 3
```

Três regras de produção foram especificadas. Termos posicionados à esquerda da regra, tal como o símbolo **E**, são chamados de não terminais. Já o termo **id** (identificador), é um símbolo enviado pelo Flex e é um símbolo terminal.Os símbolos terminais só aparecem no lado direito das regras de produção.

O objetivo do Bison é reduzir uma expressão até apenas sobrar um símbolo não terminal utilizando o processo de Análise de Deslocação e Redução (Shift-Reduce Parser). Na listagem 2.9 está um exemplo que utiliza tal processo começando com a expressão “x+y*z” e reduz até sobrar apenas um não terminal com as três regras vistas anteriormente.

Listagem 2.9: Exemplo do processo de Análise de Deslocação e Redução

```
. x + y * z Deslocaçao
x . + y * z Reduçao Regra 3
E . + y * z Deslocaçao
E + . y * z Deslocaçao
E + y . * z Reduçao Regra 3
E + E . * z Deslocaçao
E + E * . z Deslocaçao
E + E * z . Reduçao Regra 3
E + E * E . Reduçao Regra 2 Multiplicaçao
E + E Reduçao Regra 2 Soma
E.  Finalizado
```

A listagem 2.9 representa o processo de Análise de Deslocação e Redução na prática. Os termos ao lado esquerdo do ponto estão guardados em memória numa pilha, enquanto os termos à direita ainda não foram lidos. Quando um ou um conjunto de termos do topo

da pilha são correspondidos ao lado direito de uma regra de produção esse ou esses saem do topo da pilha e são inseridos o termo correspondente do lado esquerdo da regra, este processo é chamado de redução. Portanto o Bison vai efectuando a leitura da entrada e reduzindo-a até não haver mais deslocações possíveis, apenas sobrando um não terminal.

Olhando para o passo 6 seria possível efectuar a operação de soma, mas a operação de multiplicação tem prioridade. Felizmente, para não haver uma alteração das regras de produção, no Bison é possível indicar a precedência dos operadores. Caso contrário, a gramática apresentada estaria perante um conflito de deslocação e redução e seria uma gramática ambígua.

Um ficheiro com a extensão .y utilizado pelo Bison tem uma estrutura semelhante ao ficheiro com a extensão .l utilizado pelo Flex. Portanto é separado em três partes, definições, regras e sub-rotinas. Em baixo é possível verifacar o exemplo de uma calculadora, que possui um aspecto relevante, pois admite variáveis (normalmente as variáveis estão guardadas numa estrutura de dados, designada por tabela de símbolos), mesmo sendo de uma forma primitiva.

Na listagem 2.10 é visível a parte do ficheiro que o Bison irá utilizar para analisar sintacticamente a entrada. Na zona das declarações, é possível ver as declarações dos símbolos (em %token) e a prioridade entre operadores. Tem mais prioridade quem está declarado mais abaixo no código ('+' e '-' têm menos prioridade que '*' e '/') e que tipo de associatividade tem, se à esquerda, direita ou não associativa. Outro caso interessante é que sempre que é feita uma redução é executada uma acção, por exemplo, quando ocorre a redução de uma operação essa operação é efectuada. Quando é reduzida uma variável para expressão o valor é retirado da tabela de símbolos e quando já não há reduções possíveis é escrito no ecrã o valor da operação efectuada.

Listagem 2.10: Exemplo de um analisador sintáctico para uma calculadora

```
1  /* Declaraçao de simbolos e gramática para um analisador sintático de uma calculadora*/
2  /* declaracoes dos simbolos */
3
4  %token INTEGER VARIABLE
5  %left '+' '-'
6  %left '*' '/'
7
8  /* regras */
9  %%
10
11 program:
12         program statement '\n'
13         |
14         ;
15
16 statement:
17         expr { printf("%d\n", $1); }
18         | VARIABLE '=' expr { sym[$1] = $3; }
19         ;
```

```
20
21 expr:
22        INTEGER
23        | VARIABLE { $$ = sym[$1]; }
24        | expr '+' expr { $$ = $1 + $3; }
25        | expr '-' expr { $$ = $1 - $3; }
26        | expr '*' expr { $$ = $1 * $3; }
27        | expr '/' expr { $$ = $1 / $3; }
28        | '(' expr ')' { $$ = $2; }
29        ;
30 %%
```

Já na listagem 2.11 são apresentadas as expressões regulares que irão alimentar o analisador léxico. Na listagem mencionada, quando um conjunto de caracteres é analisado e é correspondido a uma expressão regular, as acções executadas diferem da listagem 2.7. Agora são passados os símbolos e valores correspondentes para o analisador sintáctico. A variável yyval faz a passagem dos valores correspondestes e são retornados os símbolos nas sub-rotinas que executam estas acções.

Listagem 2.11: Exemplo de um analisador léxico para uma calculadora

```
1 /*Expressoes regulares para um analisador léxico de uma calculadora*/
2
3 [a-z] { yylval = *yytext - 'a'; return VARIABLE; } /* variaveis */
4 [0-9]+ { yylval = atoi(yytext); return INTEGER; }/* inteiros */
5 [-+()=/*\n] { return *yytext; } /* operadores */
6 [ \t] ; /* para espaços em braco
7 . yyerror("invalid character"); /* caracter nao identificado */
```

2.8.4 Árvores de Análise Sintáctica

Uma árvore de análise sintáctica, é um diagrama em forma de árvore do conteúdo significativo de um programa. Através das regras gramaticais (implementadas no Bison), é possível dividir um conjunto de caracteres em uma estrutura ramificada que categoriza as partes desse conjunto de acordo com o seu tipo gramatical. Os aspectos essenciais são passados para a estrutura enquanto certos detalhes são omitidos.

A implementação de uma árvore de análise sintáctica está aliada ao processo da interpretação pois assim é exequível correr e interpretar o código introduzido pelo utilizador as vezes que sejam pretendidas, sem que as etapas de análise léxica e sintáctica sejam efectuadas de novo.

Estes tipos de diagramas são apresentados de maneira contrária há convencional, isto é, o nó raiz está em cima os nós folha estão em baixo. É assim feito este tipo de apresentação pois quando árvore de análise sintáctica é necessário ser corrida começa no nó raiz, que, também representa o símbolo inicial da gramática.

Em cada redução feita pela análise sintáctica, um nó é adicionado à árvore. Quando já não houver mais nenhum símbolo para reduzir (a análise chega ao símbolo inicial), também já mais nenhum nó será adicionado à árvore. O nó raiz é o último a ser introduzido. Os nós folha correspondem sempre a símbolos terminais.

Observando as regras gramaticais escritas utilizando a ferramenta Bison e demonstradas na listagem 2.12, é possível construir uma simples árvore de análise sintáctica dando como entrada a declaração "2*(3+x)".

Listagem 2.12: Exemplo de uma gramática que constrói uma árvore de análise sintática

```
1  program:
2         function { exit(0);}
3         ;
4
5  function:
6           function stmt { ex($2); freeNode($2); }
7         | /* NULL */
8         ;
9
10 stmt:
11          ';' { $$ = opr(';', 2, NULL, NULL); }
12        | expr ';' { $$ = $1; }
13        | assign_var { $$ = $1; }
14        ;
15
16 expr:
17          INTEGER { $$ = con($1); }
18        | VARIABLE { $$ = id($1); }
19        | '-' expr %prec UMINUS { $$ = opr(UMINUS, 1, $2); }
20        | expr '+' expr { $$ = opr('+', 2, $1, $3); }
21        | expr '-' expr { $$ = opr('-', 2, $1, $3); }
22        | expr '*' expr { $$ = opr('*', 2, $1, $3); }
23        | expr '/' expr { $$ = opr('/', 2, $1, $3); }
24        | '(' expr ')' { $$ = $2; }
25        ;
```

À frente de cada regra de produção pode ser vista cada acção que vai ser feita. Sempre que uma redução é efectuada, a acção executada corresponde à operação dessa regra gramatical, mas executada imediatamente em C/C++, na listagem 2.12 tal já não acontece. Cada redução faz adicionar um nó à árvore. Neste exemplo, existem três tipos de nós e um que os encapsula (este último serve para que diferentes tipos de nós se liguem). Cada tipo de nó é uma estrutura escrita em C/C++. Na tabela 2.6 são apresentados os nós utilizados.

Portanto, cada acção das regras de produção chama uma função que cria e adiciona o respetivo nó.

Olhando outra vez para o exemplo da declaração "2*(3+x)", é possível representar a expressão em esquemas de tipo árvore, como são apresentados na figura 2.12. O primeiro

Tabela 2.6: Tipos de nós para uma árvore de análise sintáctica

Nome da estrutura	Definição
conNodeType	Nó de constantes. Guarda o valor de constantes.
idNodeType	Nó de variáveis. Guarda o endereço das mesmas.
oprNodeType	Nó para operações entre outros nós.
nodeTypeTag	Encapsula os restantes nós.

esquema é a árvore de análise sintáctica enquanto o segundo é o mesmo esquema, mas mostra os tipos de nós usados para a construção da árvore.

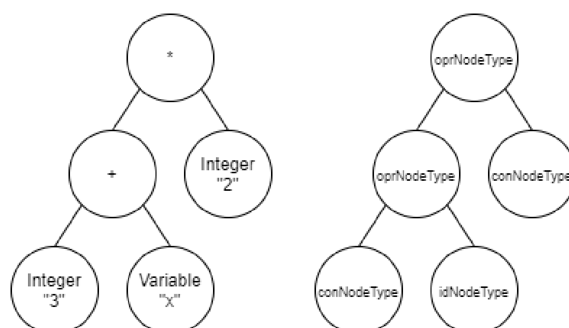

Figura 2.12: Árvore de análise sintática da declaração "2*(3+x)"e os tipos de nós utilizados para a sua implementação

Por fim, é a necessário correr a árvore e executar as operações que a integram. Para tal, é necessária uma função recursiva que em cada iteração executa cada nó ordenadamente. A listagem 2.13 mostra a implementação. Assim, foi possível interpretar uma expressão dada pelo utilizador.

Listagem 2.13: Interpretador para uma calculadora

```
1  int ex(nodeType *p) {
2     if (!p) return 0;
3     switch(p->type) {
4     case typeCon: return p->con.value;
5     case typeId: return sym[p->id.i];
6     case typeOpr:
7        switch(p->opr.oper) {
8        case ';': ex(p->opr.op[0]); return ex(p->opr.op[1]);
9        case '=': return sym[p->opr.op[0]->id.i] = ex(p->opr.op[1]);
10       case UMINUS: return -ex(p->opr.op[0]);
11       case '+': return ex(p->opr.op[0]) + ex(p->opr.op[1]);
12       case '-': return ex(p->opr.op[0]) - ex(p->opr.op[1]);
13       case '*': return ex(p->opr.op[0]) * ex(p->opr.op[1]);
14       case '/': return ex(p->opr.op[0]) / ex(p->opr.op[1]);
15       }
16    }
17    return 0;
18 }
```

2.8.5 Interoperabilidade entre Java e C++

Em diversas situações, programar numa só linguagem não é suficiente, pois diferentes linguagens de programação apresentam vantagens em relação às outras e características diferentes. Nesta dissertação apenas foi mostrado código em C/C++, só que não é suficiente, C++ é uma linguagem rápida, de baixo tempo de execução, mas não possui uma interface gráfica o que resulta numa má experiência para o utilizador. Nesta secção, são descritas as configurações necessárias de um projecto em C++ que gera um executável para uma Dynaminc Linking Library (DLL). Deste modo é possível utilizar componentes de outros sistemas.

A linguagem Java dá a possibilidade da construção de uma interface gráfica. Java possui a biblioteca JNI, Java Native Interface, onde é possível chamar métodos nativos (um método chamado em Java, mas que chama outra função correspondente de uma diferente linguagem), tipicamente disponível numa bibilioteca dinâmica.

Listagem 2.14: Classe de java utilizando uma função nativa

```
1 public class HelloC {
2     native public int sendString(String str);
3 }
```

De seguida o ficheiro cabeçalho, com a extensão .h, necessita de ser criado. Este ficheiro irá conter os nomes correctos das funções que irão ser chamados em C++. Para tal apenas é necessário compilar o ficheiro .java com a classe referida e de seguida executar o comando para a geração do ficheiro de cabeçalho, por fim apenas o ficheiro .class que é gerado no ato da compilação é apagado. Na ilustração 2.13 é possível observar os comandos executados.

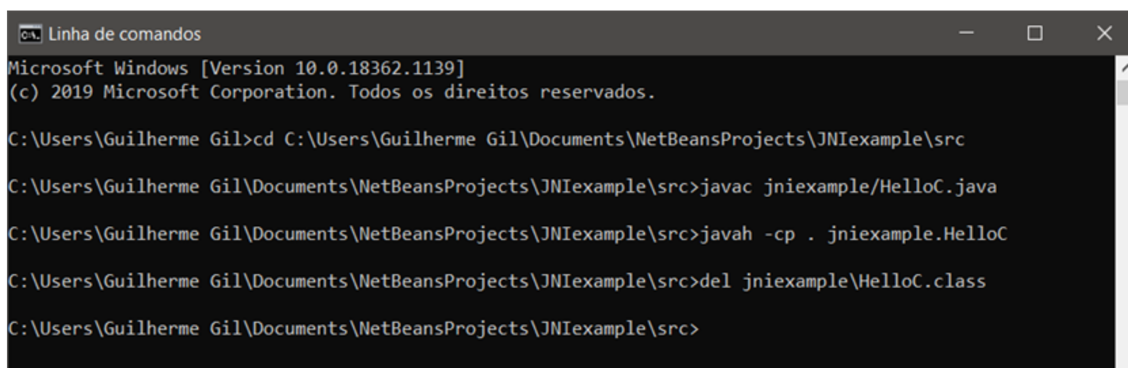

Figura 2.13: lista de comandos para a geração de um ficheiro Java com funções nativas de C

Depois de os últimos passos serem executados, é necessário focar os esforços na parte da linguagem C++. Em primeiro lugar, é necessário incluir o cabeçalho jni.h no projecto, usando a plataforma Visual Studio. Para tal, apenas é necessário incluir os caminhos para a biblioteca para o seu acesso, a figura 2.14 ilustra o processo.

De seguida é necessário criar um ficheiro com a extensão .cpp onde irá ser introduzida a função gerada anteriormente pelo ficheiro cabeçalho, como demonstra a listagem 2.15.

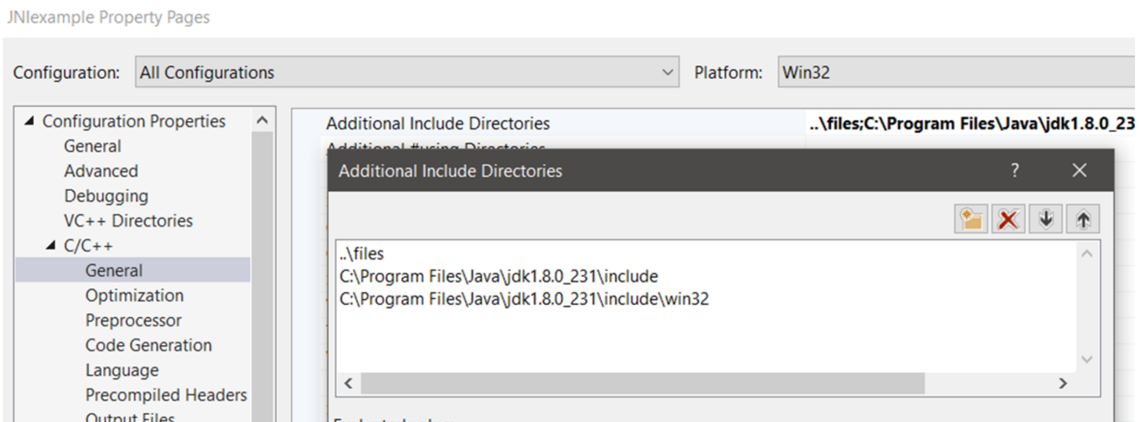

Figura 2.14: Configuração da inclusão de bibliotecas de Java num projecto de C++ usando a plataforma Visual Studio

Listagem 2.15: Função nativa chamada em C++

```
1 #include <jni.h>
2 #include <iostream>
3
4 extern "C" JNIEXPORT jint JNICALL Java_jniexample_HelloC_sendString(JNIEnv* env, jobject
    ↪ obj, jstring jstr) {
5   std::string str = std::string(env->GetStringUTFChars(jstr, 0));
6   std::cout << str << "\n";
7
8   return (jint)1;
9 }
```

Neste momento, depois de compilado o código em C++, seria criado um ficheiro executável, mas não é esse o objectivo. Para serem chamadas funções a partir de java o ficheiro ao ser compilado não pode gerar um executável, mas uma DLL. Para tal acontecer, apenas é necessário mudar nas propriedades do projecto o tipo de configuração para DLL. A figura 2.15 apresenta a configuração necessária.

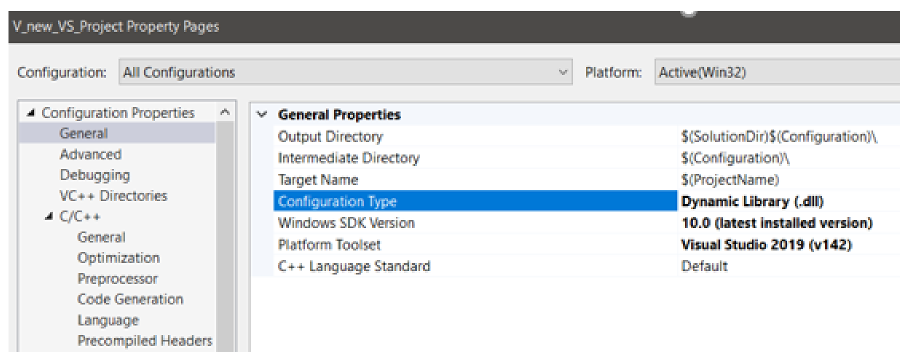

Figura 2.15: Configuração do projecto escrito em C++ para a geração de uma DLL

Por fim, apenas é necessário mudar o ambiente para x64 e copiar as configurações feitas em Win32 para x64. Agora já está tudo implementado da parte de C++.

Voltando para o projecto de Java e à classe HelloC, a classe precisa de chamar a dll, portanto é preciso carregá-la, dando a sua directoria. A classe é visível na figura 2.16.

Listagem 2.16: Ficheiro HelloC.java

```
1  package jniexample;
2
3  public class HelloC {
4      static{
5          System.load("C:\\Users\\Guilherme␣Gil\\source\\repos\\JNIexample\\x64\\Debug\\
               ↪ JNIexample.dll");
6      }
7      native public int sendString(String str);
8  }
```

Finalizando, na função main (listagem 2.17) apenas é necessário criar o objecto da class HelloC e invocar o método sendString, compilar o projecto e corrê-lo.

Listagem 2.17: Ficheiro JNIexample.java

```
1   package jniexample;
2
3  public class JNIexample {
4      public static void main(String[] args) {
5          HelloC hello = new HelloC();
6          int i = hello.sendString("Hello␣World");
7      }
```

Portanto um conjunto de caracteres criado em Java é impresso no ecrã utilizando funções de C++, a figura 2.16 mostra o resultado.

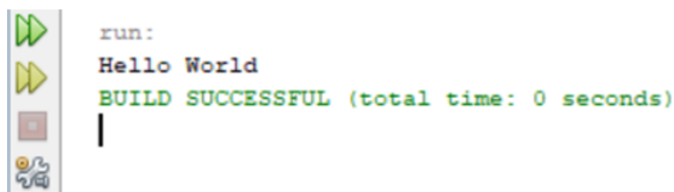

Figura 2.16: Resulta final da configuração de uma DLL. Plataforma NetBeans

Capítulo 3

Desenvolvimento

O presente capitulo é divido duas secções:

- 3.1 Arquitectura Proposta e Requisitos
- 3.2 Implementação

Na secção 3.1, são apresentados os requisitos deste projecto, também é apresentada a divisão do mesmo por tecnologias, são enunciadas as linguagens necessárias para a produção do sistema final, como também a forma como é implementada a interoperabilidade. No final é ilustrado a arquitectura proposta com as interacções entre as diferentes partes que fazem o sistema por inteiro.

Por último, na secção 3.2 é explicado todo o trabalho efectuado referente ao projecto. Desde a implementação dos analisadores léxicos e sintácticos até à construção final do cenário de simulação.

3.1 Arquitectura Proposta e Requisitos

É proposto o desenvolvimento de um sistema que consiga validar e interpretar a linguagem Texto Estruturado (ST) como também consiga suportar simulações. Aliado à parte de Texto Estruturado, é necessário a construção de uma interface gráfica para ajudar o utilizador final a testar e implementar o seu código. É possível dividir o projecto em três partes principais:

1. Construção de um compilador capaz de validar a linguagem Texto Estruturado.

   É necessário validar, como também traduzir o código recebido em Texto Estruturado, para uma árvore de análise sintáctica. Só de seguida é possível interpretar a lógica que o código recebido apresenta.

De forma a proceder à tradução, são necessárias as ferramentas Flex e Bison. A primeira irá gerar um analisador léxico e a segunda um gerador sintático. Para serem gerados são necessários dois ficheiros. Para o analisador léxico é necessário construir um ficheiro com a extensão .l onde irão ser inseridas as expressões regulares necessárias. O segundo ficheiro terá a extensão .y e será inserido a gramática referente à linguagem em Texto Estruturado.

As ferramentas a utilizar são implementadas em C/C++, portanto, o compilador será implementado em C++. Os analisadores léxico e sintáctico e o interpretador ao serem compilados irão gerar um ficheiro executável.

2. Criação de uma interface gráfica e implementação do comportamento do autómato virtual.

   A criação da interface gráfica irá ser desenvolvida utilizando a linguagem Java, pois possui capacidade de criação de elementos gráficos e fácil interoperabilidade entre diferentes sistemas.

   De forma a haver interoperabilidade entre Java e C++, é necessário criar uma DLL, em vez de um ficheiro executável como dito anteriormente. Deste modo, é possível chamar métodos de C++ mas estando a programar em Java. Deste modo será possível implementar o ciclo de operação de um autómato em Java.

   A interface gráfica terá de permitir ao utilizador inserir o código em Texto Estruturado como também um número qualquer de variáveis e endereços. É necessário avisar o utilizador caso haja erros de sintaxe no código introduzido.Terá de possuir dois modos de operação, um passo a passo e outro de execução continua.

3. Construção de um cenário de simulação.

   Mesmo sendo possível fazer testes apenas com a interface gráfica, é necessário um ambiente de simulação de modo a utilizar o potencial do autómato virtualizado. Portanto é criado um cenário de automação industrial utilizando o motor de jogos Unity. Conforme mencionado anteriormente, a incorporação de elementos de jogo para a simulação de ambientes de automação industrial, contribui com inúmeras vantagens, incluindo uma integração mais acentuada por parte do utilizador. Para a comunicação entre a interface gráfica e o cenário de simulação é necessário a criação de um servidor web como também um cliente web. O servidor será do lado do Java e o cliente do Unity e irão comunicar através do protocolo Hypertext Transfer Protocol (HTTP)

Na figura 3.1 é possível observar um esquema da arquitectura proposta e as interacções que os diferentes módulos possuem entre si.

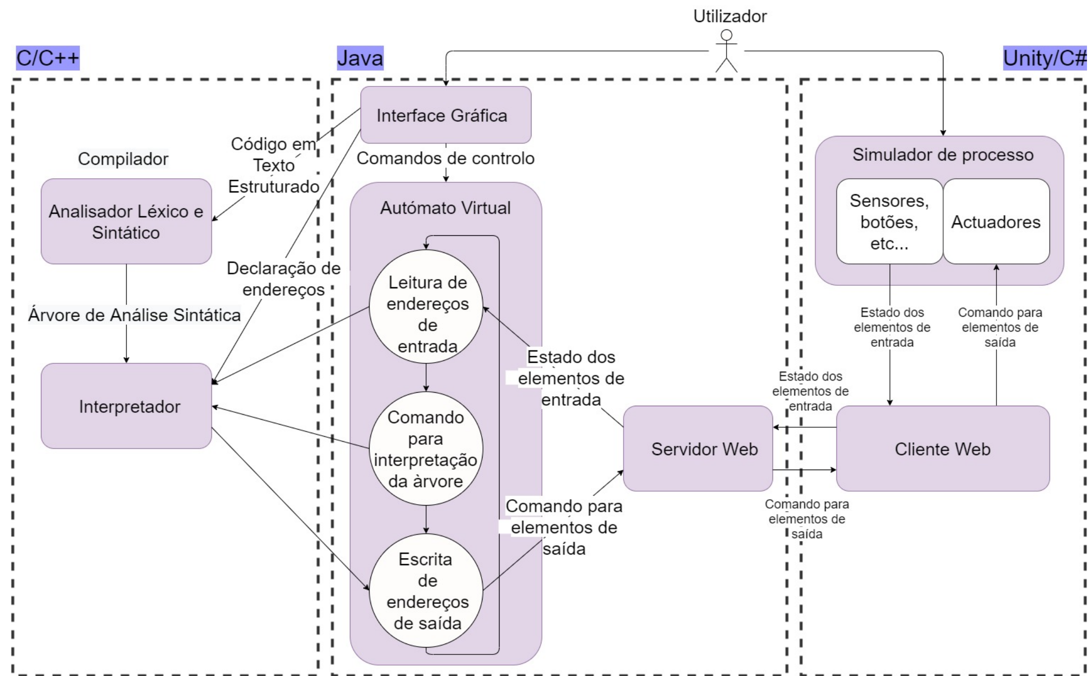

Figura 3.1: Arquitectura proposta.

3.2 Implementação

3.2.1 Implementação de um compilador usando Flex e Bison

O compilador implementado pode ser dividido em três fases essenciais: na análise léxica, análise gramatical e produção de uma árvore de análise sintáctica e por fim a interpretação da árvore. Deste modo, é possível assim receber um programa escrito em Texto Estruturado traduzir a lógica para C++ e conseguir executá-la.

Como já visto anteriormente, a análise léxica é implementada através da ferramenta Flex. Esta recebe uma cadeia de caracteres, que formam um programa em Texto Estruturado, e converte os padrões de caracteres em símbolos. Na listagem 3.1 é possível verificar parte dos padrões implementados:

Listagem 3.1: Algumas expressões regulares utilizadas no analisador léxico.

```
1   [1-9][0-9]* { yylval.iValue = atoi(yytext);
2                                return INTEGER;
3                              }
4
5   [-()<>+*/;{}.:#,] { /* Operadores */
6                return *yytext;
7              }
8   ":=" {
9                return *yytext;
10             }
11
12  ">=" return GE; /* Sinais de Comparaçao */
```

```
13 "<=" return LE;
14 "=" return EQ;
15 "<>" return NE;
16
17 "while" return WHILE; /* Declaraçoes Iterativas */
18 "do" return DO;
19 "end_while" return END_WHILE;
20 "repeat" return REPEAT;
21 "until" return UNTIL;
22 "end_repeat" return END_REPEAT;
23 "for" return FOR;
24 "to" return TO;
25 "by" return BY;
26 "end_for" return END_FOR;
```

É possível observar a implementação da análise léxica de números inteiros, operadores e palavras chaves das declarações iterativas. Se apenas o significado destes padrões interessar para a próxima fase será apenas necessário enviar o símbolo correspondente para o analisador sintáctico. No caso de o padrão ser por exemplo um número inteiro, para tal, no analisador sintáctico é possível declarar dentro da estrutura yyval variáveis que podem ser chamadas no analisador léxico e assim passar valores de um analisador para outro. As variáveis para este efeito podem ser visualizadas na tabela 3.1.

Tabela 3.1: Variáveis para a passagem de valores entre analisadores.

Tipo	Nome	Descrição
int	iValue	Guardar valores inteiros
double	rValue	Guardar valores reais
char*	sIndex	Guardar nomes de variáveis ou outros conjuntos
char*	boolValue	"true"ou "false"

Já na parte da análise sintáctica, os símbolos recebidos são verificados de forma a se perceber se foi construída uma relação lógica entre eles, ou não. Se a combinação de símbolos não fizer sentido, isto é, se não foi construída uma gramática válida, uma mensagem com o conteúdo "Erro sintáctico" é enviado ao utilizador. Com referido na secção 2.8.3 os símbolos (em primeiro lugar são todos terminais) vão sendo analisados sequencialmente, se houver uma redução possível o conjunto respectivo de símbolos serão reduzidos para um símbolo não terminal. Terminais e não terminais vão sendo reduzidos até apenas haver um não terminal. Em cada redução é adicionado um nó à árvore de análise sintáctica.

Na parte das declarações do ficheiro com a extensão .y, podem ser encontradas os protótipos referentes às funções que são chamadas nas regras de produção e que são responsáveis por criar os diferentes tipos de nós da árvore de análise sintáctica. Também são encontrados outros protótipos como por exemplo:

- freeNode - Libertar a memória alocada pela criação de cada nó

- ex – Caso seja necessário correr a árvore;
- yylex – Para chamar o analisador léxico;
- yyerror – Função chamada caso exista um erro de sintaxe.

Também são inicializadas as variáveis STprog e varDeclaration que são apontadores para árvores. A primeira irá apontar para a árvore referente ao corpo do programa e a segunda apontará para uma árvore referente à declaração de variáveis, caso seja necessário à sua utilização. Na listagem 3.2 pode ser visto o bloco de código referente ao analisador sintáctico com os protótipos e declarações referidas.

Listagem 3.2: Protótipos de funções e declaração de variáveis do analisador sintáctico

```
/* prototipos */
nodeType *id_timer(const char* ident, const char* mode);
nodeType *timer_time_base(int num, ...);
nodeType *id_timer_var(const char* ident);
nodeType *opr(int oper, int nops, ...);
nodeType *id(const char* ident);
nodeType *con(int value);
nodeType *con_real(double value);
nodeType *varDataType(int dataTypeID);

void freeNode(nodeType *p);
int ex(nodeType *p);
int yylex(void);
void yyerror(const char *s);

/* declaracao dos apontadores paras as arvores de analise sintatica */
nodeType *varDeclaration;
nodeType *STprog;
```

Através da implementação de vários tipos de nós é possível construir uma árvore de análise sintáctica. Cada tipo é um apontador para uma estrutura que pode guardar valores de variáveis, nomes, informação de temporizadores ou até apontadores para outros nós. Na tabela 3.2 podem ser visualizados cada tipo e que variáveis são declaradas nas suas estruturas.

Os protótipos de funções utilizadas e declarações de variáveis podem ser encontrados na parte do ficheiro com a extensão .y. Nesta secção do ficheiro também estão declarados os símbolos terminais e não terminais.

Como pode ser observado na listagem 3.3, está um "%union" que é responsável por declarar as variáveis que fazem a passagem do Flex para o Bison. Todos os símbolos que estão associados à passagem de valores entre o analisador léxico e o sintáctico, na sua declaração, à frente da palavra chave "%token", possuem o nome da variável que faz a passagem.

Listagem 3.3: Variáveis Bison e declaração de terminais

Tabela 3.2: Nós utilizados para a implementação da árvore de análise sintáctica

Tipo da Estrutura	Variável	Descrição
nodeType	(nodeEnum) type (conNodeType) con (conRNodeType) conReal (idNodeType) id (oprNodeType) opr (idNodeVarType) varType (timerVarType) timerVarType (timerBaseType) timeBase (timerIdType) timerId	Intuito de encapsular os nós, o enumerador nodeEnum diz que tipo de nó está a ser referido.
timerIdType	(char*) timer (int) mode	Nome e tipo do temporizador.
timeBaseType	(int) value	Tempo em milissegundos do timer.
timerVarType	(char*) timer (int) var	Nome do temporizador e que variável foi chamada (ex:%TM1.P).
conRNodeType	(double) value	Valor real.
conNodeType	(int) value	Valor de um inteiro ou um booleano.
idNodeType	(char*) ident	Nome de uma variável.
idNodeVarType	(int) id	Tipo de uma variável: entrada, saída ou variável interna.
oprNodeType	(int) oper (int) nops (struct nodeType*) op[]	Guarda o tipo de operação, o número de outros não associados e um vetor com apontares para esses nós.

```
1  %union {
2      int iValue; /* inteiros */
3      double rValue; /* reais */
4      char* sIndex; /* variaveis */
5      nodeType *nPtr; /* node pointer*/
6      char *boolValue; /* booleanos */
7  };
8
9  /* Terminais */
10 %token <iValue> INTEGER
11 %token <rValue> REAL
12 %token <sIndex> VARIABLE ADRESS TIMER TIMER_VAR TIMER_MODE TIMER_TIME TIMER_MINUTS
       ↪ TIMER_SECONDS
13 %token <boolValue> BOOL_VALUE
```

Já na listagem 3.4, a primeira parte corresponde com os símbolos que correspondem às declarações condicionais, declarações iterativas e funções dos temporizadores. Na parte seguinte, estão as declarações dos símbolos não associativos com a palavra chave "%nonassoc". Estes símbolos possuem uma prioridade entre eles, os declarados numa linha

abaixo têm mais prioridade do que os declarados em cima. Este tipo de declaração é implementado para evitar mensagens de erro de conflito de deslocação e redução, que serão explicados quando a gramática da declaração "if" for apresentada.

Por último, são declarados os símbolos responsáveis pelas comparações lógicas, operações e mudanças de estado. Tal como os símbolos não associativos estes também possuem prioridade (por exemplo a multiplicação tem mais prioridade que uma soma), as declarações mais em baixo no código têm mais prioridade do que as declaradas em cima.

Listagem 3.4: Declaração de símbolos terminais referentes a declarações e operações aritméticas e lógicas

```
1  /* terminais */
2  %token WHILE DO END_WHILE REPEAT UNTIL END_REPEAT FOR TO BY END_FOR IF IF_ELSIF THEN
       ↪ END_IF PRINT EXIT REASSIGN ASSOC
3  %token START DOWN START_TIMER DOWN_TIMER TIMER_VAR_VALUE
4
5  %nonassoc IFX DECL_NO_ATRIBUTION
6  %nonassoc ELSIF ELSE DECL_W_ATRIBUTION
7  %nonassoc FOR_PREC
8
9  %left OR
10 %left XOR
11 %left AND
12 %left GE LE EQ NE '>' '<'
13 %left '+' '-'
14 %left '*' '/'
15 %right UMINUS NOT RE FE
```

No final da secção das declarações, são declarados os símbolos não terminais, estes também possuem um tipo nodeType, portanto à frente da palavra chave "%token" é inserido a variável "nptr" como pode ser visto na listagem 3.5.

Listagem 3.5: Declaração de símbolos terminais referentes a declarações e operações aritméticas e lógicas

```
/* Nao Terminais */
%type <nPtr> program function
%type <nPtr> stmt expr stmt_list if_stmt while_stmt repeat_stmt assign_var ELSIF_part
%type <nPtr> for_stmt
```

A primeira parte adicionada à gramática foram as regras de produção associadas a uma expressão, estas regras são reduzidas para o não terminal expr. As regras de produção que são reduzidas a este não terminal são em primeiro lugar valores inteiros (INTEGER), reais (REAL), e booleanos (BOOL_VALUE), endereços (ADRESS), variáveis internas (VARIABLE) e variáveis de temporizadores (TIMER_VAR). De seguida, já é possível reduzir operações, como operações aritméticas, unárias e de lógica, pois estas regras de produção só aceitam nos seus parâmetros o não terminal expr. As funções para começar e parar os temporizadores (START TIMER e DOWN TIMER), de mudanças de estado de variáveis (Rising Edge - RE e Falling Edge - FE) também são reduzidas para expressões. Na listagem 3.6 é possível a gramática implementada para estas regras de produção.

Cada regra de produção faz executar uma acção, como já dito, cada acção irá adicionar um nó à árvore de análise sintáctica. As funções chamadas nas acções retornam todas um nó, que é igualado ao símbolo que irá surgir devido à redução. No caso da listagem em baixo, irá ser igualado a expr, sendo que o termo “$$” corresponde a este símbolo. Olhando para a primeira regra, o símbolo “INTEGER” irá ser reduzido para “expr”, sendo que “$$” refere-se a “expr” que irá ser igualado a um nó que contem esse valor inteiro. A função “con($1)” irá retornar esse nó, do tipo nodeType mas que contem uma estrutura do tipo conNodeType, e o parâmetro “$1” refere-se ao primeiro símbolo da parte direita da regra, neste caso “INTEGER”.

Olhando agora para a regra da soma. “$$” corresponde à mesma a “expr”, também será igualado a um nó do tipo nodeType mas este contem uma estrutura do tipo oprNodeType, que é um nó que representa operações. A função que retorna este nó tem o nome de opr, sendo que possui como parâmetros o tipo da operação, neste caso o símbolo “+” que é representado por um inteiro, o número de nós que aponta (dois nós), e esses nós que são “expr”.

Listagem 3.6: Gramática para a redução de uma expressão

```
expr:
         INTEGER { $$ = con($1); }
       | REAL { $$ = con_real($1); }
       | BOOL_VALUE { if(strcmp($1,"true")==0){
                              $$ = con(1);
       }
                              else $$ = con(0);
                           }
       | VARIABLE { $$ = id($1); }
       | START TIMER { $$ = opr(START_TIMER, 1, id_timer($2, 0)); }
```

```
11         | DOWN TIMER { $$ = opr(DOWN_TIMER, 1,id_timer($2, 0)); }
12         | RE expr { $$ = opr(RE, 1, $2); }
13         | FE expr { $$ = opr(FE, 1, $2); }
14         | TIMER_VAR { $$ = id_timer_var($1); }
15         | ADRESS { $$ = id($1); }
16         | '-' expr %prec UMINUS { $$ = opr(UMINUS, 1, $2); }
17         | expr '+' expr { $$ = opr('+', 2, $1, $3); }
18         | expr '-' expr { $$ = opr('-', 2, $1, $3); }
19         | expr '*' expr { $$ = opr('*', 2, $1, $3); }
20         | expr '/' expr { $$ = opr('/', 2, $1, $3); }
21         | expr '<' expr { $$ = opr('<', 2, $1, $3); }
22         | expr '>' expr { $$ = opr('>', 2, $1, $3); }
23         | expr GE expr { $$ = opr(GE, 2, $1, $3); }
24         | expr LE expr { $$ = opr(LE, 2, $1, $3); }
25         | expr NE expr { $$ = opr(NE, 2, $1, $3); }
26         | expr EQ expr { $$ = opr(EQ, 2, $1, $3); }
27         | '(' expr ')' { $$ = opr(ASSOC,1, $2); }
28         | expr OR expr { $$ = opr(OR, 2, $1, $3); }
29         | expr XOR expr { $$ = opr(XOR, 2, $1, $3); }
30         | expr AND expr { $$ = opr(AND, 2, $1, $3); }
31         | NOT expr { $$ = opr(NOT, 1, $2); }
32         ;
```

Um bom exemplo de gramática e de uma árvore de análise sintáctica (listagem 3.7 e imagem 3.2) é o ciclo for. Possui uma atribuição de valor inicial, a variável que irá ser testada, o incremento ou decremento dessa e por fim o valor final para que o ciclo pare. “$$” corresponde ao não terminal “for_stmt” do tipo nodeType. Este nodeType possui uma estrutura do tipo oprNodeType, que é retornada através da função, mais uma vez, opr. Esta função recebe como parâmetros o terminal “FOR”, o número de nós que irá receber, e os nós, que são quatro e que são:

- *nodeAssign – Nó para a atribuição inicial de uma variável;
- *nodeReassingLoop – Nó responsável pelo incremento ou decremento da variável inicial;
- *nodeConLoop – Nó responsável pela condição de saída do ciclo;
- *nodeForStaments – Nó que possui as declarações que o irão ser executadas em cada iteração do ciclo.

Listagem 3.7: Gramática para a declaração iterativa for

```
1  for_stmt:
2        FOR VARIABLE ':=' expr TO expr BY expr DO stmt_list END_FOR %prec FOR_PREC
3        {
4          nodeType *nodeAssign = opr('=', 2, id($2), $4);
5          nodeType *nodeReassignLoop = opr(REASSIGN, 2, id($2), $8);
6          nodeType *nodeCondLoop = opr(LE, 2, id($2), $6);
```

```
7         nodeType *nodeForStatements = $10;
8         $$ = opr(FOR, 4, nodeAssign , nodeCondLoop , nodeReassignLoop ,
              ↪ nodeForStatements );
9       }
10      ;
```

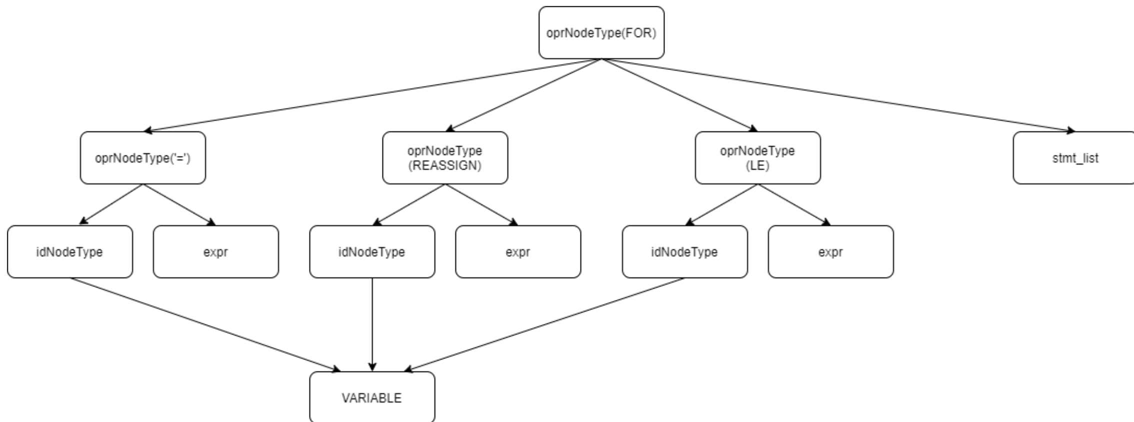

Figura 3.2: Árvore de análise sintáctica para a declaração for

A recursividade é um aspecto interessante na implementação de gramática pois é com ela que é possível ter várias declarações escritas pelo utilizador. A implementação da chamada recursividade à esquerda pode ser vista na listagem 3.8. Uma gramática é recursiva à esquerda se o seu símbolo mais á esquerda do lado direito da gramática for o mesmo que o símbolo à esquerda da gramática ("stmt_list" neste caso).

Listagem 3.8: Gramática para um conjunto seguido de declarações

```
1 stmt_list:
2         stmt { $$ = $1;}
3       | stmt_list stmt { $$ = opr(';', 2, $1, $2); }
4       ;
```

Imaginando um exemplo em que o utilizador escreve 3 declarações, "stmt1 stmt2 stmt3". Utilizando a gramática em cima, em primeiro lugar "stmt1" irá ser puxado para a pilha e será reduzido para "stmt_list", assim as declarações iniciais passarão a ser: "stmt_list stmt2 stmt3". De seguida, será o stmt2 a ser puxado, e a pilha terá dois símbolos "stmt_list stmt2", que corresponde à segunda regra de produção, pelo que uma redução irá acontecer e assim o resultado passará a ser: "stmt_list stmt3". Por fim, o "stmt3" é puxado para a pilha, e, de seguida uma redução de "stmt_list stmt3" para "stmt_list" é executada. Como funciona para três declaração também funciona para qualquer número.

Uma gramática com algumas nuances é a declaração condiciona "if" que pode ser encontrada na listagem 3.9. Foi necessário separar as regras de produção nas várias variantes em que pode ser escrito esta declaração: apenas o "if", "if-else", "if-elsif" e "if-elsif-else". É possível ter um número qualquer de "elsif" com a ajuda de uma gramática idêntica ao da listagem 3.8, que pode ser vista ma gramática do não terminal "ELSIF_part".

Listagem 3.9: Gramática para um conjunto seguido de declarações

```
1  if_stmt:
2          IF expr THEN stmt_list END_IF %prec IFX { $$ = opr(IF, 2, $2, $4); }
3        | IF expr THEN stmt_list ELSE stmt_list END_IF { $$ = opr(IF, 3, $2, $4, $6); }
4        | IF expr THEN stmt_list ELSIF_part END_IF { $$ = opr(IF_ELSIF, 3, $2, $4, $5);}
5        | IF expr THEN stmt_list ELSIF_part ELSE stmt_list END_IF {$$ = opr(IF_ELSIF, 4,
           ↪ $2, $4, $5, $7);}
6        ;
7
8  ELSIF_part:
9          ELSIF expr THEN stmt_list { $$ = opr(ELSIF, 2, $2, $4);}
10       | ELSIF_part ELSIF expr THEN stmt_list { $$ = opr(ELSIF, 3, $3, $5, $1); }
11       ;
```

O símbolo inicial é o símbolo “program”, que resulta da redução do não terminal “function” que por sua vez resulta de uma gramática recursiva de declarações, “stmt”. Quando ocorre a redução para “program” a única ação efetuada é a de guardar a árvore de análise sintática na variável STprog, assim pode ser interpretada as vezes que o utilizador entender o que vai de encontro com o funcionamento continuo de um PLC.

Depois de a gramática ficar completamente analisada, é necessário interpretá-la, isto é, interpretar a árvore de análise sintáctica. Tal processo é efectuado pela classe “interpretor” nos ficheiros interpretor.h e interpretor.cpp. A função principal desta classe e que é chamada quando se quer executar a árvore chama-se “ex” e tem como protótipo “double ex(nodeType* p)”. Devolve double pois é o maior tipo de dado utilizado (serve para representar o tipo “float” em Texto Estruturado). Assim é possível trabalhar com valores booleanos, inteiros e reais. Também pode ser que a função recebe como parâmetro de entrada um apontador para “nodeType”, sendo essa variável o nó da árvore que irá ser executado.

No anexo I é verificada a função “ex” na sua totalidade. A sua funcionalidade está à volta de duas declarações condicionais switch, a primeira é implementada para saber que tipo de nó é o recebido (ver tabela 3.2). Por exemplo, se o nó a ser executado é o de uma constante inteira o seu tipo é “typeCon” e é retornado esse valor. Se o nó for de uma variável, será do tipo “typeId” e também retornará o seu valor. Caso o nó se refira a uma operação será do tipo “typeOpr” e é nesta parte onde a segunda declaração switch é implementada. É através desta segunda declaração switch que o tipo de operação é testado através da variável “oper” que tem como conteúdo o valor da operação a que o nó se refere. Na listagem 3.10, pode ser visto a implementação da interpretação do ciclo for. É perceptível que a função “ex” é recursiva, chama-se a si própria, pois necessita de chamar outros nós. Esses nós são acedidos através do vetor “op”. No caso da implementação do “for”, foram adicionados quatro nós que podem ser vistos na listagem 3.7. Olhando para o código referente à interpretação deste ciclo. pode ser visto que em primeiro lugar é executado o nó que está na posição zero do vector “op” que está encarregue da atribuição de um valor inicial à variável a ser testada. De seguida, a execução entra num ciclo while

onde é testada a condição de saída do for antes de cada iteração. Dentro do while, em primeiro lugar é executado o nó na posição três que se refere às declarações a serem executadas dentro do "for", depois de as declarações serem executadas a variável é actualizada, através do nó de operação "REASSIGN" através da execução do nó na posição dois, de seguida a variável é testada e dependendo do valor, o ciclo continua ou não.

Listagem 3.10: Interpretação da declaração for

```
case FOR: {
        ex(p->opr.op[0]); // nodeAssign
        while (ex(p->opr.op[1])) { // nodeCondLoop
            ex(p->opr.op[3]); // nodeForStatements
            ex(p->opr.op[2]); // nodeReassignLoop
        }
        return 0;
    }
    case REASSIGN: {
        symTable[p->opr.op[0]->id.ident]->word_value = symTable[p->opr.op[0]->id.ident
            ↪ ]->word_value + (int) ex(p->opr.op[1]) ;
        return 0;
    }
```

Com o uso de variáveis existe a necessidade de guardar os seus valores e estados. Para tal no ficheiro interpretor.h são implementados não ordenados como mostra a listagem 3.11. Um mapa possui dois parâmetros uma chave, que nestes mapas são nomes de variáveis, e o conteúdo. Foram implementados três mapas não ordenados:

- symTable – guarda o tipo e valor de endereços de entrada, saída e variáveis de memória interna;
- edgeState – para marcar uma mudança de estado de endereços booleanos no scan cycle do momento;
- timerTable – guarda o estado como também variáveis dos timers.

Listagem 3.11: Mapas criados para o armazenamento de variáveis e estados das mesmas

```
std::unordered_map< std::string, symStruct*> symTable;
std::unordered_map< std::string, int*> edgeState;
std::unordered_map< std::string, timerStruct*> timerTable;
```

Finalmente, no que diz respeito à parte da programação de C++ foram criadas as funções necessárias para haver interoperabilidade com a componentes gráfica escrita em Java. Tais funções estão descritas na tabel3.3

3.2.2 Interface Gráfica utilizando JavaFX

Para implementar uma interface gráfica, foi utilizada a plataforma JavaFX, que é o sucessor da plataforma Java Swing. A plataforma JavaFX é um conjunto de pacotes gráficos e

Tabela 3.3: Funções para a utilização do interpretador

Função	Descrição
parse_a_string	Analisar o programa e criar uma árvore de análise sintática.
turnOn	Ligar o controlador.
singleStep	Corre a árvore uma única vez.
stop	Apaga a árvore
addAddress	Declarar uma variável
assignValueToAddress	Escrever o valor enviado numa variável específica
getAddressValue	Devolve o valor de uma variável
resetEdgeState	Reiniciar todo o conteúdo sobre as mudanças de estado dos endereços
addTimer	Adicionar um temporizador

bibliotecas de Java que possibilitam criar e testar aplicações gráficas do lado do cliente. Foca-se numa estrutura hierárquica tipo árvore com nós, em que a raiz tem o nome de gráfico de cena (em inglês scene graph). Os nós representam os elementos visuais da interface gráfica. São suportados comandos de entrada feitos pelos utilizadores. Com a excepção da raiz, cada nó tem apenas um pai e zero ou mais filhos.

A cena, isto é a janela que o utilizador está a visualizar no momento é controlado por uma tarefa que corre em paralelo ao resto do programa, que tem o nome de tarefa de aplicação ou controlador. É esta tarefa de aplicação que controla todos os eventos disparados através de por exemplo o premir de um botão. Também é nesta classe onde são declarados todos os elementos visuais utilizados.

Para a interface gráfica deste projecto, foram implementados três separadores, todos dentro da mesma cena e, portanto, controlados pela mesma tarefa de aplicação. Os separadores têm os nomes de IDE, PLOTS e Unity e cada um possuí uma funcionalidade própria:

IDE – Contem toda a parte de desenvolvimento de um programa de ST como também é possível observar as mudanças de valores de endereços e variáveis em tempo real;

PLOTS – Contem dois gráficos onde é possível definir variáveis e observar as suas evoluções ao longo da execução;

Unity – Responsável pela ligação do simulador de PLC com um modelo gráfico de um processo físico.

3.2.2.1 Aspeto Visual do Separador IDE

Como o projecto possui já algum nível de complexidade seria impensável implementar a parte gráfica apenas usando código Java. Para tal foi utilizado a aplicação SceneBuilder onde é possível criar todo o aspecto visual através de simples arrasto de elementos.

Também é possível ligar variáveis e eventos aos vários elementos. Esta aplicação cria um ficheiro tipo FXML, idêntico a um XML, que possui todas a informação da componente visual criada no SceneBuilder. De seguida e para que o programa em Java compile apenas é necessário declarar todas as variáveis que representam os elementos como também os metidos que são executados devido ao disparo de eventos.

Como já referido, a interface gráfica foi divida em três separadores sendo que o principal é denominado de IDE, Integrated Integrated Development Environment ou em português Ambiente de Desenvolvimento Integrado, e é onde o utilizador consegue introduzir o seu código em Texto Estruturado, verificar erros de sintaxe e observar a execução do seu código. Na figura 3.3 é possível observar este separador.

Figura 3.3: Separador IDE.

Na parte central do separador existe um campo de texto onde o utilizador pode escrever o seu código em Texto Estruturado. De modo a facilitar o trabalho ao utilizador, também é possível carregar um ficheiro de texto neste campo de texto como também guardar o seu conteúdo e efectuar as operações de copiar e colar texto provenientes de outros ficheiros. À esquerda e à direita estão três tabelas ao que contêm as variáveis declaradas, uma coluna para os endereços de entrada, outra para os de saída e uma terceira para as variáveis internas. Cada tabela possui duas colunas, uma para o nome da variável em questão e outra para o seu valor, esta última coluna é actualizada em tempo real. Para declarar as variáveis, apenas é necessário utilizar os campos de texto em baixo para introduzir o nome e um valor inicial e carregar no botão para adicionar a variável, se a variável estiver correctamente escrita é adicionada á tabela. Os valores introduzidos são aproximados para o tipo de dados da variável. Também é possível remover variáveis das tabelas através do botão “Remove Selected Row”.

No canto inferior esquerdo do separador existe, com o nome de "Output", outro campo

de texto onde são enviadas mensagens de texto relevantes para o utilizador como por exemplo se o programa contém ou não algum problema de sintaxe ou houve uma declaração incorrecta de uma variável. Também é enviado uma mensagem de texto quando uma mudança de estado do controlador é efectuada.

Também na parte esquerda do separador estão, todos os botões e opções de configurações e execução do autómato. Começando por baixo, pode ser observado uma lista para o utilizador escolher uma das suas opções: "Single Run" ou "Continuous Run". A primeira opção apenas permite que o controlador corra um certo número de vezes indicados pelo programador no campo de texto acima. Para correr outra vez apenas é necessário carregar no botão "Run". Esta opção é preferível quando apenas é necessário um certo número de iterações. Já a segunda opção permite que o controlador corra em tempo continuo, preferível para o controlo de sistemas em tempo real.

Já na parte dos botões, foram implementados quatro. O primeiro botão é o de "Load" e permite criar uma instância do autómato e traduzir o código em Texto Estruturado para uma árvore de análise sintáctica. Se o código desenvolvido não apresentar uma gramática correta uma mensagem de erro de sintaxe é apresentada. O segundo botão tem como função para mudar o estado do PLC de desligado para ligado, caso o utilizador tenha escolhido o modo "Single Run" são efectuadas o número de iterações indicadas, caso contrário irá executar a lógica implementada pelo utilizador continuamente. O botão "Run" tem o funcionamento de correr mais uma vez o número de ciclos indicado. Por fim o botão "Stop" faz parar a execução e apagar o estado actual do autómato virtual.

A mesma lógica de um controlador lógico programável foi aplicada, isto é, um ciclo continuo da sequência de:

- Leitura de endereços de entrada;
- Interpretação do programa;
- Actualização dos endereços de saída;
- Repetição do processo.

Para tal foram criadas duas tarefas que correm em paralelo com a parte gráfica: "addressTableThread" e "PLC". Estas duas tarefas são classes de Java com extensão "Thread". A tarefa "addressTable" é responsável pela ligação entre a parte gráfica e o controlador. Possui os métodos necessários para enviar os endereços de entrada para o controlador e pedir os valores dos endereços de entrada de forma a actualizar as tabelas e enviar os mesmos para um simulador, caso esteja haja ligação com um. Já a tarefa "PLC" contem o método para que a árvore seja executada uma vez (todas os métodos dentro das tarefas encontram-se dentro de um ciclo para que se seja possível um funcionamento continuo) como também o método para reiniciar os estados dos endereços.

As tarefas referidas correm em paralelo. Contudo, para executar correctamente o ciclo de um PLC todos os métodos necessitam de ser executados numa ordem exacta. Para

garantir este sincronismo foram utilizados semáforos. Os semáforos são mecanismos que podem ser retirados ao adicionadas as chamadas permissões. Uma tarefa estará à espera para retirar uma certa quantidade de permissões enquanto o semáforo em causa não possui, só quando a outra tarefa acabar de executar a sua parte do ciclo do controlador é que liberta essa quantidade de permissão, é desta forma que é garantido um funcionamento síncrono.

3.2.3 Construção de um Simulador de Processo

De forma a poder-se utilizar o simulador do PLC de uma forma realista, é importante a sua ligação a um sistema ou processo que se queira controlar. Nesta secção, materializa-se o conceito de gamificação, que permite simular de uma forma mais abrangente o funcionamento do PLC. O sistema é modelado utilizando o motor de jogos Unity. Para tal foi construído um modelo lúdico de um ambiente industrial com o motor de jogos Unity.

Um projecto é o equivalente a um jogo e uma cena de um projecto equivale a um nível. Uma cena é um espaço em 3 dimensões onde existem os objectos de jogo, texturas, sons fazendo parte de um único sistema. Um objecto de jogo são os objectos fundamentais no Unity e representam personagens, adereços, ou outros elementos. Por si só nada fazem, mas servem como cápsulas para componentes que implementam as funcionalidades reais. Por exemplo, um objecto luminoso é criado através da ligação de um componente de luz ao objecto de jogo. Já um objecto que representa um cubo sólido possui um componente *Mesh Filter* e um componente *Mesh Renderer* de forma a ser possível desenhar a superfície do cubo. Para implementar o volume do sólido em termos físicos (colisão com outros objectos), é necessário adicionar um component *Box Collider*. Todos os objectos de jogos também possuem uma componente com o nome de transformada, esta componente indica a posição e orientação de cada um [8]. Na figura 3.4 é possível verificar a implementação referida.

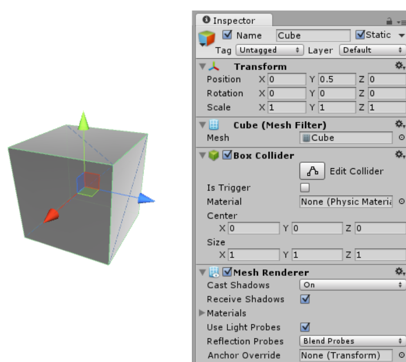

Figura 3.4: Componentes de um objecto de jogo tipo cubo. Retirado de: [8]

O modelo implementado foi o de um exemplo para ordenar caixas consoante o seu peso. As caixas aparecem num tapete rolante, o seu peso é medido, se o peso for um certo valor a caixa irá ser empurrada para uma rampa, caso contrário irá ser empurrado para

uma segunda rampa. Para tal foram implementados vários elementos que interagem, ou directamente através do utilizador ou por leitura e escrita em sensores e actuadores. Na tabela 3.4 é possível verificar elementos implementados.

Tabela 3.4: Elementos implementados no simulador

Nome	Funcionamento
Tapete Rolante	0 - Parar;1 - Mover o tapete.
Tapete Rolante com Balança	Tapete: 0 - Parar ; 1- Mover o tapete. Balança: Valor inteiro com o peso presente em cima da balança.
Braço Automático	0 - Parar; 1 - Mover o braço no sentido positivo ; 2 - Mover no sentido negativo.
Máquina Geradora de Caixas	Gera uma caixa quando há uma mudança do seu estado de 0 para 1.
Botão	0 - Não carregado; 1 - Carregado.
Sensor de Posição	0 - Não acionado; 0- Acionado.

A componente *script* dos objectos de jogo consiste um ficheiro em C# que é escrito pelo programador. Este ficheiro possui dois métodos predefinidos que podem ou não ser utilizados: Start e Update. O primeiro corre apenas uma vez no inicio do programa enquanto o método Update é chamado sempre que há uma actualização do ecrã (a cada *frame*). O método Update é utilizado pelo gerador de caixas e pelos braços automáticos para verificarem o seu estado e o que fazerem consoante alterações dos mesmos, os restantes elementos usam métodos também da biblioteca do Unity mas estes são chamados só no caso do disparo de algum evento.

Os tapetes rolantes utilizam o método OnTriggerStay. Este é sempre chamado em cada actualização do ecrã por cada objecto, neste caso caixas, que estejam em cima do tapete. Cada tapete rolante possui como filho um outro objecto apenas com a componente da posição,e, está está posicionado à frente do tapete. Sempre que uma caixa é posicionada em cima de uma destas transportadoras, esta move-se em direção ao filho da transportadora, simulando assim o compartimento dos tapetes rolantes.

O botão implementado utiliza o método OnMouseDown que é chamado sempre que é carregado com o rato em cima do botão. Neste método em específico a variável do estado, que se refere à posição do botão, é alterada e o material do botão é mudado para dar uma sensação de ligado ou desligado. Já os sensores utilizam dois métodos: OnTriggerStay e OnTriggerExit. São utilizados para actualizar o estado do sensor caso um objecto esteja a tocar no raio que simula o feixe de luz do sensor ou a sair do raio. Este raio possui uma cor vermelha, mas caso algum objecto toque no sensor este passa para verde. Na figura 3.5 pode ser observado o modelo final.

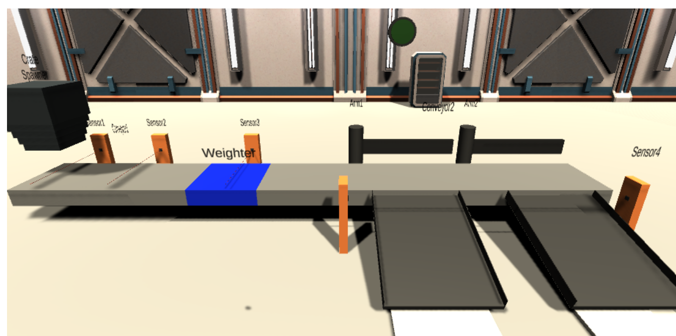

Figura 3.5: Modelo implementado.

3.2.4 Comunicação Autómato Virtualizado e Simulador de Processo

Surgiu a necessidade de comunicar com o simulador, para tal, uma forma de criar esta interoperabilidade é através de um modelo servidor-cliente. Para este caso, foram criados um servidor e um cliente Web que comunicam através do protocolo HTTP. A figura 3.6 exemplifica o processo de comunicação entre simulador de PLC e modelo do processo.

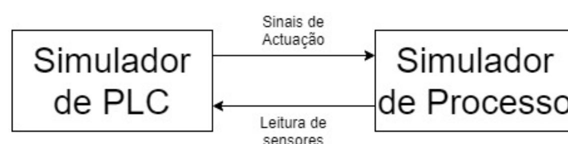

Figura 3.6: Comunicação entre Simulador de PLC e simulador de processo.

Como a aplicação em Java contém toda a lógica para o autómato virtualizado e a interface gráfica faz todo o sentido de ser implementado também o servidor web neste programa. De forma a implementar um servidor web em Java foi utilizado a plataforma Rapidoid [9]. Com esta plataforma é bastante fácil implementar um servidor web. Foi criado uma classe estática com o nome WebServer que possui o método startServer. Para iniciar o servidor é necessário apenas chamar este método que irá especificar a porta de rede e especificar as rotinas para quando um método especifico é chamado utilizando um URL específico. Os métodos utilizados são o POST e o GET e são especificados quatro URLs:

/Unity/OutElements - Usa o método POST para enviar os nomes dos actuadores existentes no simulador para o servidor;

/Unity/InElements - Usa o método POST para enviar os nomes de sensores e botões existentes no simulador para o servidor;

/Unity/Outputs - Usa o método GET para o cliente pedir os valores dos actuadores naquele momento;

/Unity/Inputs - Usa o método POST para enviar os valores dos sensores e botões existentes do simulador para o servidor.

No lado do servidor os nomes e valores dos elementos de entrada são guardados em listas, totalizando quatro listas, duas para guardar os nomes e valores dos elementos de entrada e outras 2 para guardar os nomes e valores de elementos de saída. A ligação entre o simulador e o autómato virtualizado é feita principalmente no separador "Unity" na figura 3.7 é possível observar tal separador.

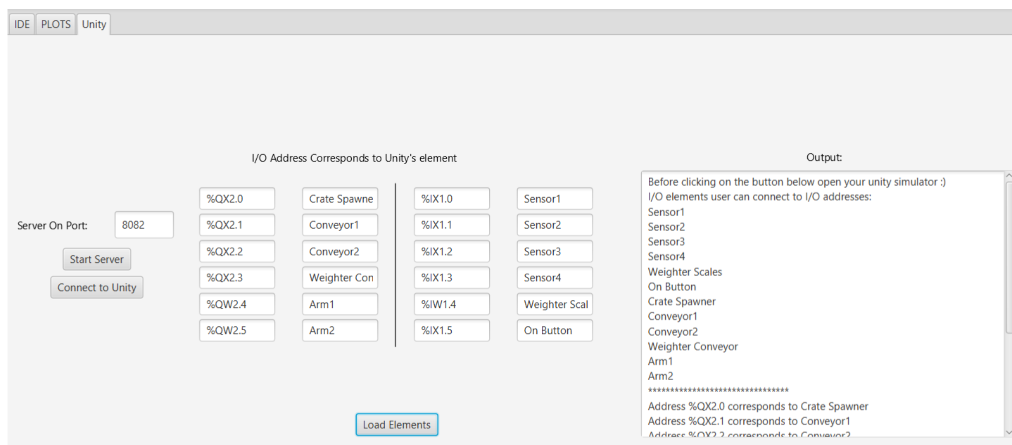

Figura 3.7: Separador Unity da interface gráfica.

O separador em cima referido é responsável pela inicialização da comunicação entre os dois programas. Primeiro, é necessário inicializar o servidor, em que apenas é necessário especificar qual o porto do servidor e carregar no botão "Start Server". De seguida, é enviada uma mensagem ao utilizador para ligar o simulador. Ao ligar o simulador, são enviados os elementos que podem ser endereçáveis através dos URL **/Unity/OutElements** e **/Unity/InElements** e são guardados em listas. Para o utilizador ver quais os elementos disponíveis, só necessita de carregar no botão "Connect to Unity". Nos campos de texto, na zona central do separador, são onde os elementos são associados aos endereços, o endereço no campo à esquerda irá ligar ao elemento imediatamente à direita, por pré-definição os elementos e endereços já estão escritos nos campos de texto. Para finalizar, apenas é necessário carregar no botão "Load Elements", estes endereços irão ser guardados numa lista e estarão na mesma posição que os elementos do simulador guardados noutra lista. Por fim, serão apresentadas mensagens que indicam que endereço liga com que elemento ou se houve algum erro. Depois desta configuração ser realizada, simulador e autómato já conseguem comunicar correctamente.

No inicio do "scan cycle"do autómato irão ser actualizados os endereços de entrada. Portanto é necessário iterar a lista com os endereços e como a mesma posição da lista de elementos corresponde ao endereço interligado apenas é necessário atribuir o valor do elemento ao endereço. No final do ciclo, o mesmo é feito para os endereços de entrada,mas neste caso são atribuídos os valores dos endereços de saída associados aos respectivos elementos. Para terminar o ciclo, as tabelas são actualizadas de modo ao utilizador conseguir verificar as alterações de estados.

Já na parte do simulador, foi criado um objecto de com o nome WebClient apenas com

a componente da posição e um ficheiro em C# com o mesmo nome do objecto. É este ficheiro que irá tratar de toda a comunicação do lado do cliente. A implementação do métodos GET e POST são quase directos através da biblioteca de C#, "System.Net.Http". Com o método GET é recebido um conjunto de caracteres e com o método POST toda a informação enviada para o servidor irá estar à frente do URL, em cada par chave-valor à frente do URL é especificado um elemento do simulador e o seu respectivo valor.

Como já dito um componente script em Unity possui dois métodos principais que não são obrigatórios. No caso do objecto WebClient são os dois utilizados. No método Start são enviados os elementos, que podem ser endereçados, usados no simulador e no Update são chamados os métodos para ler valores de entrada e escrever valores de saída de forma a actualizar o autómato e o estado do modelo lúdico.

Capítulo 4

Aplicação

Este capítulo tem como objectivo ilustrar diferentes testes e os resultados obtidos de modo mostrar a aplicação implementada, como as também suas funcionalidades. É divido em duas secções:

4.1 - Testes Preliminares

4.2 - Teste do Simulador

Na secção 4.1 são ilustrados os testes à interpretação do código em Texto Estruturado. Estes testes são feitos através da interface gráfica criada. São apresentados testes às declarações mais essenciais, aos temporizadores, operações lógicas e aritméticas. Por último, é apresentado a execução do autómato virtual com um Proporcional Integral Derivativo (PID) sem ligação a um processo.

Como o próprio nome indica, na secção 4.2 é feito o teste final, ao simulador criado. É explicado o código que controla o cenário como também é ilustrado uma imagem do sistema a ser executado.

4.1 Testes Preliminares

Antes de efectuar a ligação do autómato virtual a um processo, foi necessário avaliaras funcionalidades do compilador. Para tal, foram realizados testes preliminares com o intuito de validar as principais declarações, operações e tipos de dados básicos da linguagem Texto Estruturado. Mesmo não fazendo parte desta linguagem foi criada a instrução PRINT para escrever na janela da plataforma Netbeans variáveis de interesse, facilitando assim o processo de testes.

Em primeiro lugar foram testadas as operações lógicas: AND, NOT e XOR. A figura 4.1 mostra o código em texto estruturado escrito. A primeira etapa passa por declarar

três variáveis, var0, var1 e var2, dando valores iniciais, verdade, verdade e falso, respectivamente. Na declaração de variáveis de memória interna, é pedido ao utilizador que insira à frente da variável o seu tipo, por exemplo, a variável boolean var0, para ser declarada é necessário escrever "var0:bool"no campo de texto associado à tabela de variáveis de memória interna e temporizadores. Depois da declaração das variáveis e o programa escrito, para o mesmo ficar carregado no autómato virtual apenas é necessário carregar no botão "Load". Como não existe nenhum erro de sintaxe, uma mensagem a confirmar a compilação é apresentada.

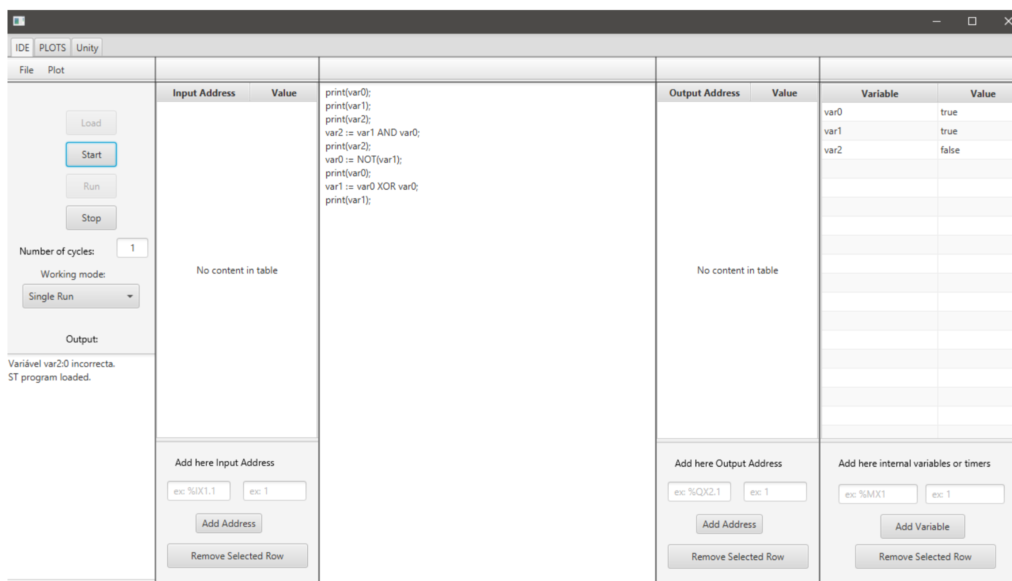

Figura 4.1: Programa para teste de operações lógicas.

É necessário apenas correr o código uma vez, pelo que é escolhida a opção "Single Run". A figura 4.2 apresenta os resultados escritos através da invocação de várias declarações PRINT no código em Texto Estruturado. Em primeiro lugar são escritos os valores iniciais das variáveis anteriormente, var0 e var1 possuem o valor de verdade, portanto é escrito "1"e var2 possui inicialmente o valor de falso,logo é escrito no ecrã "0". A primeira operação é um "TRUE AND TRUE", sendo o resultado claramente "TRUE"e de seguida é atribuído à variável var2 que tem imediatamente o seu valor escrito (quarto valor apresentado). A próxima operação é a atribuição de um "NOT TRUE"à variável var0 e claro é apresentado o valor "0", de falso, no ecrã. Por último é atribuído a var1 o resultado de "FALSE XOR FALSE"que é "FALSE"e portanto é escrito por último o valor de 0. É possível concluir as operações lógicas testadas foram implementadas com sucesso.

O próximo teste terá a finalidade de verificar a implementação da declaração condicional IF-ELSE-ELSIF. A figura 4.3 apresenta a GUI já com o teste efectuado. Pode ser visto que o teste foi efectuado com sucesso, pois foram testadas as condições do IF e do ELSIF e retornaram valores falsos (o que está correcto com a lógica implementada) e assim foi apenas executada a declaração do ELSE. É possível deduzir tal coisa pois cada endereço de saída declarado tinha como valor inicial "FALSE"e o único endereço

```
Printing variable: 1
Printing variable: 1
Printing variable: 0
Printing variable: 1
Printing variable: 0
Printing variable: 0
```

Figura 4.2: Resultados do teste de operações lógicas apresentados na linha de comandos do ambiente de desenvolvimento Netbeans.

mudado para "TRUE"foi o que tinha a nova atribuição de valor na sub-rotina do ELSE. Pode ser concluído que a declaração condicional IF-ELSE-ELSIF e as condições lógicas foram implementadas com sucesso.

IDE | PLOTS | Unity
File Plot

Load
Start
Run
Stop
Number of cycles: 1
Working mode: Single Run
Output:
ST program loaded.
Starting Controller.

Input Address	Value
%IW1.0	2
%IW1.1	1

Add here Input Address
ex: %IX1.1 ex: 1
Add Address
Remove Selected Row

```
if %IW1.0 > 5 then
    %QX2.0 := 1;
elsif %IW1.1 = 2 then
    %QX2.1 := 1;
else
    %QX2.2 := 1;
end_if;
```

Output Address	Value
%QX2.0	false
%QX2.1	false
%QX2.2	true

Add here Output Address
ex: %QX2.1 ex: 1
Add Address
Remove Selected Row

Figura 4.3: Resultados do teste da declaração concional IF.

Também foi necessário testar a precisão do temporizador. Para tal foi escrito o código na figura 4.4. O código é simples e apenas faz ligar o temporizador. Para a variável %TM1 ser declarada é necessário na parte de inserir o valor declarar as propriedades do mesmo, portanto foi escrito "TON,T#1s,10", a primeira parte refere-se ao tipo de temporizador, a segunda parte é referente ao tempo base, 1 segundo, e a terceira parte é o valor 10, sendo que o temporizador terá uma duração máxima de 10 segundos (1 segundo x 10). Na parte do código, quando o temporizador termina a variável %QX2.3 ficará com o valor de verdade.

Foram criadas duas variáveis em C++, no interpretador, para guardar quando o temporizador começou e quando acabaram os 10 segundos, de seguida foi fazer a diferença e imprimir no ecrã. Este teste foi executado no modo "Continuos Run"e quando o temporizador chegou ao seu fim foi escrito no ecrã "1"que era o valor atual da variável %QX2.3, antes imprimia sempre o valor zero pois o temporizador não tinha chegado ao seu tempo

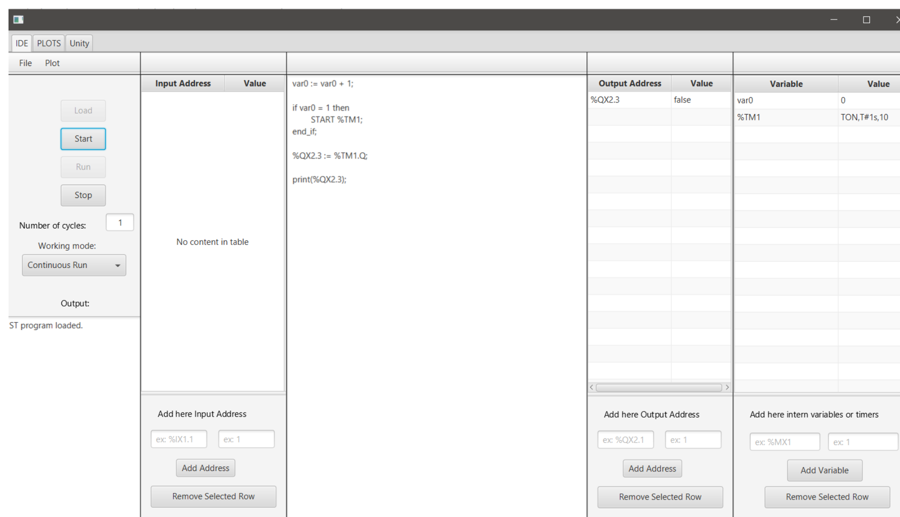

Figura 4.4: GUI com o teste de um temporizador

limite, e foi também impresso o tempo, em segundos, da duração do ecrã, sendo o seu valor de 10.0031s, o que parece ser bastante preciso. Na figura 4.5 é possível observar essas impressões.

```
Tempo que o temporizador demorou: 10.0031
Endereco %QX2.3 com o valor : 1
Printing variable: 1
```

Figura 4.5: Resultado da precisão de um temporizador de 10 segundos.

Para terminar os testes preliminares foi implementado um controlador PID, sem processo, de forma a testar as operações aritméticas. O código utilizado encontra-se na figura 4.6. A variável a controlar é o endereço %QF2.0 que tem de alcançar o valor 0 (referência, indicado pela variável setpoint). Para as constantes kp, ki e kd foram escolhidos os valores 0.1,0.5 e 0.01 respectivamente.

Para este teste, foram escolhidos 100 ciclos no modo de operação "Single Run". A evolução da variável a ser controlada poder ser vista na figura 4.7, este gráfico pode ser encontrado no separador "PLOTS". O algoritmo apresentado também foi escrito e executado utilizando a ferramenta Matlab, como pode ser verificado no anexo II. A implementação do controlador usando as duas ferramentas geraram resultados semelhantes, pois foram comparados os últimos valores da variável controlada e em ambos os casos esta variável estava igualada -0.1128. Desta forma foi possível concluir que as operações aritméticas foram implementadas com êxito. É de notar que as constantes do controlador poderiam ser optimizadas, mas este teste tinha apenas como âmbito comparar duas implementações utilizando ferramentas diferentes.

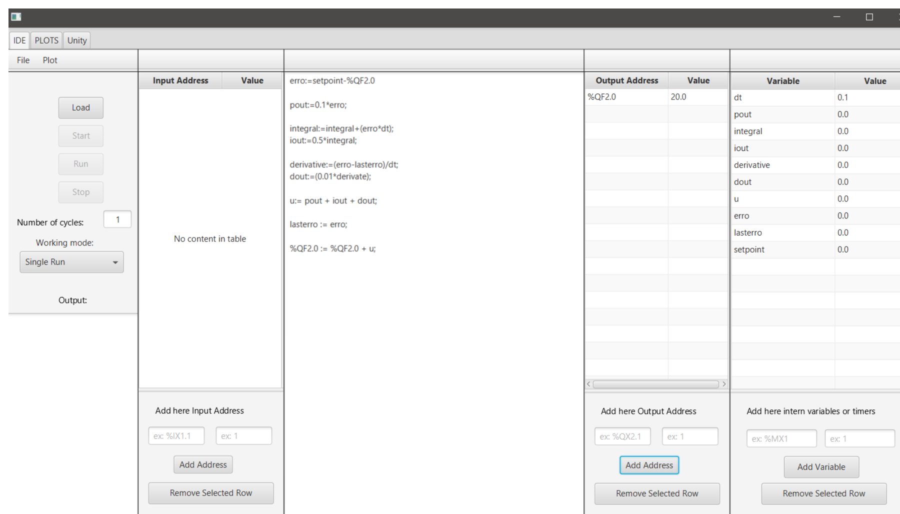

Figura 4.6: Código para um controlador PID.

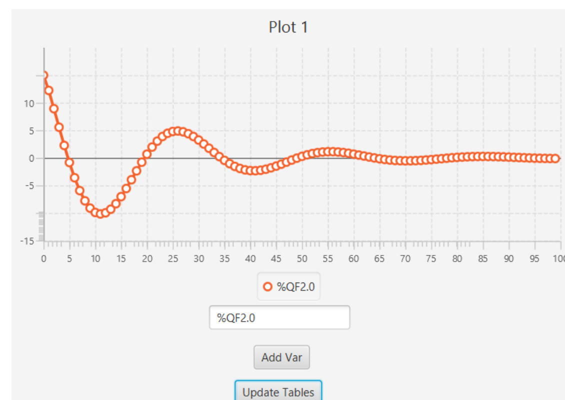

Figura 4.7: Resultado do teste para um controlador PID.

4.2 Teste do Simulador

Como já referido, foi construído um simulador de processo utilizando a plataforma Unity. À entrada da linha, são recebidas caixas com dois pesos diferentes. Cada uma é pesada e dependendo do valor serão empurradas para uma diferente rampa. Para os tapetes rolantes ligarem, é disponibilizado um botão para o utilizador começar o processo. Sempre que uma caixa descer umas das rampas irá tocar num objecto de jogo chamado ExitPad. Ao tocar nessa plataforma a caixa irá desaparecer (para facilitar o processo).

A figura 4.8 possui uma imagem da interface gráfica com o código responsável por controlar o cenário construído. O código apresentado é essencialmente constituído por declarações IF de modo a criar respostas às mudanças de valores de sensores e do botão. Em primeiro lugar, são declaradas as instruções caso o botão seja carregado, caso aconteça, uma caixa irá ser gerada e os três tapetes rolantes irão ser ligados.

A segunda declaração condicional tem o valor de verdade quando uma caixa é gerada e passa pelo Sensor 1. Quando tal acontece (e acontece sempre que uma caixa é criada)

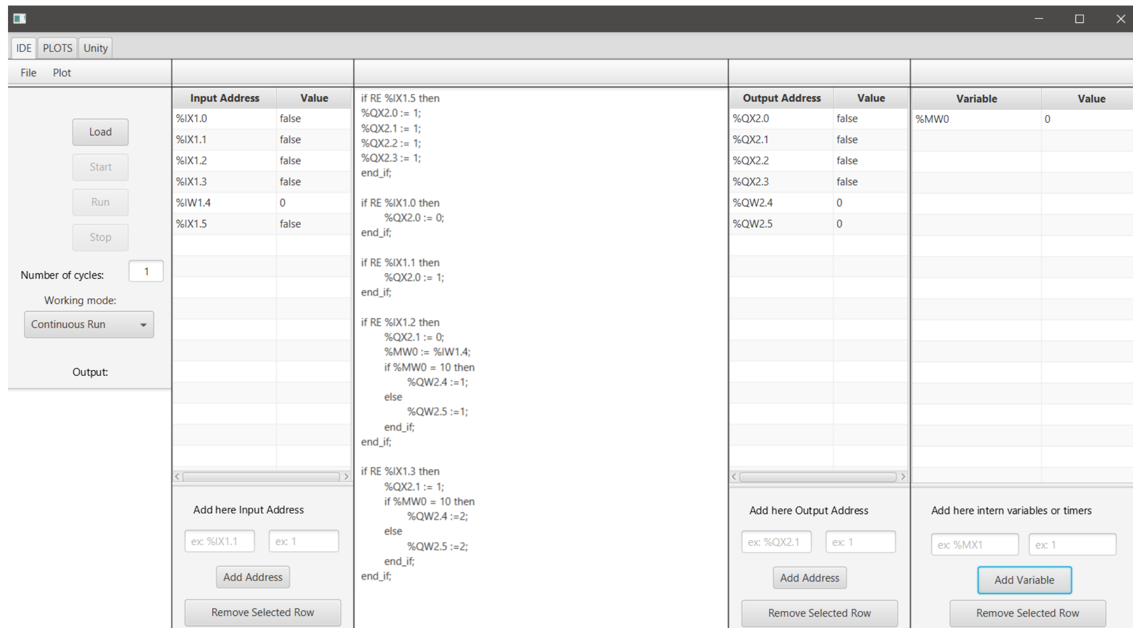

Figura 4.8: Código responsável pelo controlo do cenário construído.

o gerador de caixas é reiniciado. O terceiro IF faz com que o valor estado do gerador de caixas passe a um, caso o antigo valor seja 0 é gerado uma nova caixa, tal acontece quando uma outra caixa passe pelo Sensor 2.

Passando para a próxima declaração condicional, esta executa as declarações dentro da mesma sempre que uma caixa passa pelo Sensor 3. Portanto, quando uma caixa pela balança, o primeiro tapete é parado e o valor do peso da caixa que está em cima da balança é guardado numa variável interna. Dependendo do peso da caixa um dos braços irá ser accionado.

Passando para a última declaração IF. Quando uma caixa já saiu completamente do tapete e entrou numa das rampas o Sensor 4 é accionado, quando tal acção acontece o primeiro tapete é ligado novamente e o braço que foi accionada anteriormente irá voltar à posição de repouso.

Na figura 4.9 é possível observar uma caixa a ser levada para a segunda rampa e a accionar o Sensor 4.

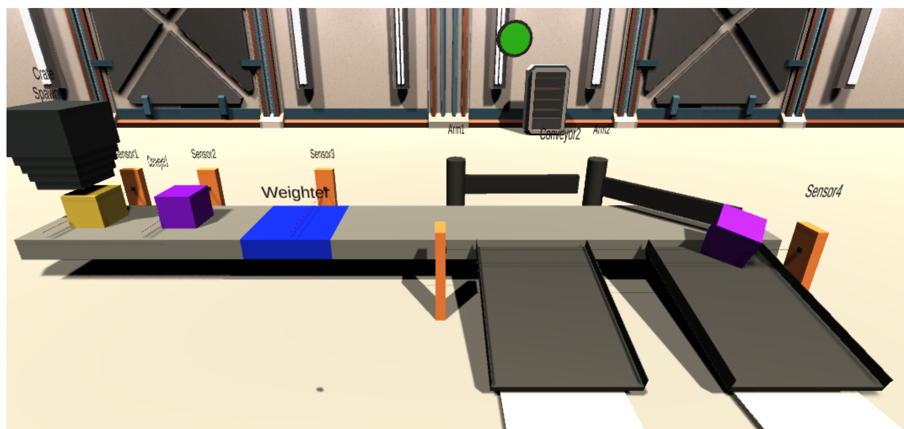

Figura 4.9: Execução do simulador.

Capítulo 5

Conclusão

5.1 Conclusões

O tema da simulação é importante na indústria como também a nível educativo. As linhas de produção devem parar o menos possível para aumentar o rendimento por isso pode ser necessário a criação de um ambiente de simulação numa fase de criação de sistemas ou mudanças dos mesmos, deste modo é possível antever o comportamento dos mesmos. Num aspecto educativo, PLCs podem ser muito dispendiosos ou completamente fora do orçamento de instituições de ensino o que pode levar à virtualização do equipamento para redução de custos. O aspecto da virtualização também é importante num ambiente disruptivo, onde não é possível a utilização pessoal do material disponibilizado.

Foi desenvolvido um sistema de simulação para a linguagem de programação Texto Estruturado. Este sistema é capaz de validar código na linguagem mencionada e executar a sua lógica escrita. Também existe a possibilidade de o sistema comunicar com um cenário de simulação de modo a testar um processo, contudo não é obrigatório. O projecto desenvolvido é divido em quatro partes essenciais: compilador para a linguagem Texto Estruturado, interface gráfica, autómato virtualizado e por fim um cenário de simulação.

A parte do compilador primeiro passa por uma fase de tradução do código em Texto Estruturado para uma árvore de análise sintáctica, que é uma representação hierárquica em forma de árvore da mesma lógica do código de entrada. Para este processo foram produzidos um analisador léxico e outro sintáctico. O analisador léxico lê todos os elementos do código dado como em entrada e irá atribuir a cada um, um símbolo respectivo dependendo do seu tipo (if irá ser atribuído um IF, o valor 1 irá ser atribuído o símbolo INTEGER, etc..). Para que o tradutor consiga associar os elementos a símbolos é necessário no ficheiro associado à ferramenta Flex adicionar as expressões regulares relativas a Texto Estruturado.

O analisador sintáctico começa onde o analisador léxico terminou, irá verificar as relações que os símbolos possuem entre si. Mais precisamente, irá ser verificado se a gramática escrita pelo utilizador é a correta segundo a gramática especificada. Durante o processo de verificação a árvore de análise sintáctica irá sendo construída. A construção do analisador sintáctico foi feita através da ferramenta Bison. A linguagem Texto Estruturado é um caso particular da GLC (*Gramática Livre de Contexto*) e portanto foi importante seguir as suas regras gramaticais na construção deste analisador.

O interpretador pega na árvore e corre os seus nós executando assim as operações correspondentes ao código introduzido em primeiro lugar. É capaz de fazer tal coisa porque cada nó da árvore é especificado que operação deve ser efectuada.

Na parte seguinte foi construída uma interface gráfica, utilizando a plataforma JavaFX, o utilizador consegue introduzir o seu código em Texto Estruturado, declarar as variáveis que entenda e testar o seu código. Foi também nesta fase implementado o comportamento do autómato virtual. Paralelamente à interface gráfica, foi feito o ciclo de operação do autómato: as variáveis de entrada são lidas, o programa corre uma vez (neste caso a árvore de análise sintáctica é percorrida), as variáveis de saída são escritas, repete-se o ciclo.

Por fim é construído um cenário de simulação utilizando o motor de jogos Unity. Para uma boa fase de testes é importante que haja simulações e para tal foi implementado um cenário em 3D para que o código introduzido seja testado mais facilmente e seja mais gratificante para que o estudante que utilize o simulador consiga percepcionar aquilo que está a implementar.

A construção deste simulador é vista como um sucesso como pode ser observado no capítulo 4. Foi testado apenas o autómato virtual sem estar ligado a nada como também ligado ao cenário de simulação. O primeiro conjunto de testes têm como objectivo apresentar as diversas funcionalidades da gramática implementada relativamente à linguagem Texto Estruturado. O último teste apresenta a robustez do sistema final, onde é possível ligar-se um processo e controlá-lo utilizando o código em Texto Estruturado introduzido por um utilizador. Desde modo, são utilizados todos os diferentes módulos construídos durante as várias fazes do projecto.

5.2 Trabalho Futuro

O objectivo principal na construção do simulador apresentado é a sua utilização em aulas de ensino superior para que alunos consigam melhorar a sua experiência na aula de automação industrial. A pesquisa académica é construída uma em cima de outras, de modo a melhorar a qualidade de pesquisas futuras, deste modo alguns pontos podem ser vistos como trabalho futuro, que neste projecto em específico ou em algum idêntico:

- Trabalhar na gestão de erros de sintaxe. Seria bom mostrar qual a linha e até posição do carácter onde surgiu o erro. Arranjar forma de facilitar a procura de erros por parte dos utilizadores;

- Criar cenários cada vez mais complexos. O projecto apresentado já possui robustez quanto à adição de mais elementos no simulador, portanto seria facilmente possível criar um cenário maior e assim elevar a exigência pedida aos alunos;
- A adição de mais linguagens da norma IEC 31161-3. Seria vantajoso adicionar, por exemplo, a linguagem Ladder.

Bibliografia

[1] acedido em 27-11-2020. URL: https://sim4edu.com/explain/what-is-continuous-simulation.html.

[2] acedido em 27-11-2020. URL: https://sim4edu.com/explain/what-is-discrete-event-simulation.

[3] acedido em 27-11-2020. URL: https://www.tutorialspoint.com/modelling_and_simulation/modelling_and_simulation_quick_guide.htm.

[4] acedido em 27-11-2020. URL: https://sim4edu.com/explain/what-is-processing-network-simulation.

[5] acedido em 12-11-2020. URL: http://obi.virtualmethodstudio.com/.

[6] acedido em 12-11-2020. URL: https://assetstore.unity.com/packages/tools/utilities/game4automation-digital-twin-professional-143543#description.

[7] acedido em 12-11-2020. URL: https://factoryio.com/.

[8] acedido em 19-11-2020. URL: https://docs.unity3d.com/560/Documentation/Manual/class-GameObject.html.

[9] acedido em 19-11-2020. URL: https://www.rapidoid.org/.

[10] A. V. Aho, M. S. Lam, R. Sethi e J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. 2006.

[11] J. Banks. "Introduction to simulation". Em: *2000 Winter Simulation Conference Proceedings (Cat. No. 00CH37165)*. Vol. 1. IEEE. 2000, pp. 9–16.

[12] G. Baratoff e H. Regenbrecht. "Developing and applying AR technology in design, production, service and training". Em: *Virtual and augmented reality applications in manufacturing*. Springer, 2004, pp. 207–236.

[13] W. Bolton. *Programmable Logic Controllers*. Elsevier Science, 2015. ISBN: 9780081003534. URL: https://books.google.pt/books?id=sDqnBQAAQBAJ.

[14] P. E. Ceruzzi, E Paul et al. *A history of modern computing*. MIT press, 2003.

[15] G. Chryssolouris, D. Mavrikios e M. Pappas. “A web and virtual reality based paradigm for collaborative management and verification of design knowledge”. Em: *Methods and Tools for Effective Knowledge Life-Cycle-Management*. Springer, 2008, pp. 91–105.

[16] R. W. Conway, B. M. Johnson e W. L. Maxwell. “An experimental investigation of priority dispatching”. Em: *Journal of Industrial Engineering* 11.3 (1960), pp. 221–229.

[17] S. Deterding, D. Dixon, R. Khaled e L. Nacke. “From game design elements to gamefulness: defining"gamification"”. Em: *Proceedings of the 15th international academic MindTrek conference: Envisioning future media environments*. 2011, pp. 9–15.

[18] F. Didactic. *FluidSIM Pneumatics Guide*. Festo Didactic, 2004.

[19] M. Dixon. *WHAT ARE THE MOST POPULAR PLC PROGRAMMING LANGUAGES?* https://realpars.com/plc-programming-languages/.

[20] M. Eletric. *MELSEC iQ-R Structured Text (ST), Programming Guide Book*.

[21] S. Eletric. *PL7 Micro/Junior/Pro, Description of the PL7 software*. URL: https://www.se.com/ww/en/download/document/35015365K01000/.

[22] J. A. Farrell. “Compiler Basics”. Em: (1995). URL: http://www.cs.man.ac.uk/~pjj/farrell/compmain.html.

[23] N. O. Fernandes e S. do Carmo-Silva. “Generic POLCA—A production and materials flow control mechanism for quick response manufacturing”. Em: *International Journal of Production Economics* 104.1 (2006), pp. 74–84.

[24] M. Fischer, H. Renken, C. Laroque, G. Schaumann e W. Dangelmaier. “Automated 3d-motion planning for ramps and stairs in intra-logistics material flow simulations”. Em: *Proceedings of the 2010 Winter Simulation Conference*. IEEE. 2010, pp. 1648–1660.

[25] J. Fründ, J. Gausemeier, C. Matysczok e R. Radkowski. “Using augmented reality technology to support the automobile development”. Em: *International Conference on Computer Supported Cooperative Work in Design*. Springer. 2004, pp. 289–298.

[26] O. Grillmeyer. *Exploring computer science with Scheme*. Springer Science & Business Media, 2013.

[27] *Introducing a New Language for Automatic Programming Univac Flow-Matic*. Sperry Rand Corporation. Univac Data Processing Division, 1957.

[28] K. M. Kapp. *The gamification of learning and instruction: game-based methods and strategies for training and education*. John Wiley & Sons, 2012.

[29] S. M. Kelapure, S. S. Akella e J. G. Rao. “Application of web services in SCADA systems”. Em: *International Journal of Emerging Electric Power Systems* 6.1 (2006).

[30] A. E. Kwame, E. M. Martey e A. G. Chris. "Qualitative Assessment of Compiled, Interpreted and Hybrid Programming Languages". Em: ().

[31] M. A. Laughton e M. G. Say. *Electrical engineer's reference book*. Elsevier, 2013.

[32] B. G. Liptak. *Instrument Engineers' Handbook, Volume Two: Process Control and Optimization*. CRC press, 2018.

[33] S.-Y. Lu, M Shpitalni e R. Gadh. "Virtual and augmented reality technologies for product realization". Em: *CIRP Annals* 48.2 (1999), pp. 471–495.

[34] A. MARCZEWSKI. *Game mechanics in gamification*. acedido em 08-Novembro-2020. 2013. URL: https://www.gamified.uk/2013/01/14/game-mechanics-in-gamification/.

[35] A. Maria. "Introduction to modeling and simulation". Em: *Proceedings of the 29th conference on Winter simulation*. 1997, pp. 7–13.

[36] A. P. Markopoulos, A. Fragkou, P. D. Kasidiaris e J. P. Davim. "Gamification in engineering education and professional training". Em: *International Journal of Mechanical Engineering Education* 43.2 (2015), pp. 118–131.

[37] D Mourtzis e M Doukas. "A web-based platform for customer integration in the decentralised manufacturing of personalised products". Em: *Procedia CIRP* 3 (2012), pp. 209–214.

[38] D. Mourtzis e M. Doukas. "The evolution of manufacturing systems: From craftsmanship to the era of customisation". Em: *Handbook of research on design and management of lean production systems*. IGI Global, 2014, pp. 1–29.

[39] W. A. Najjar, W. Bohm, B. A. Draper, J. Hammes, R. Rinker, J. R. Beveridge, M. Chawathe e C. Ross. "High-level language abstraction for reconfigurable computing". Em: *Computer* 36.8 (2003), pp. 63–69.

[40] F. Narciso, A. Rios-Bolivar, F. Hidrobo e O. Gonzalez. "A syntactic specification for the programming languages of the IEC 61131-3 standard". Em: *Proceedings of the 9th WSEAS international conference on computational intelligence, man-machine systems and cybernetics*. 2010, pp. 171–176.

[41] G. M. Nawara e W. S. Hassanein. "Solving the job-shop scheduling problem by Arena simulation software". Em: *International Journal of Engineering Innovations and Research* 2.2 (2013), p. 161.

[42] T. Niemann. *Lex Yacc*. URL: https://www.epaperpress.com/lexandyacc/index.html.

[43] L. B. Palma, V. Brito, J Rosas e P. Gil. "WEB PLC simulator for ST programming". Em: *2017 4th Experiment@ International Conference (exp. at'17)*. IEEE. 2017, pp. 303–308.

[44] S. C. Park, C. M. Park e G.-N. Wang. "A PLC programming environment based on a virtual plant". Em: *The international journal of advanced manufacturing technology* 39.11-12 (2008), pp. 1262–1270.

[45] J. Parker e K. Becker. "The simulation-game controversy: What is a ludic simulation?" Em: *International Journal of Gaming and Computer-Mediated Simulations (IJGCMS)* 5.1 (2013), pp. 1–12.

[46] J. M. F. Pecorelli. "Simulador de linguagem em texto estruturado para autómato TSX3721". Tese de mestrado. 2014.

[47] C. D. Pegden, R. P. Sadowski e R. E. Shannon. *Introduction to simulation using SIMAN*. McGraw-Hill, Inc., 1995.

[48] Z.-M. Qiu e Y. Wong. "Dynamic workflow change in PDM systems". Em: *Computers in Industry* 58.5 (2007), pp. 453–463.

[49] S. Raychaudhuri. "Introduction to Monte Carlo simulation". Em: *2008 Winter Simulation Conference*. 2008, pp. 91–100. DOI: 10.1109/WSC.2008.4736059.

[50] K. A. Redish e W. F. Smyth. "Program Style Analysis: A Natural by-Product of Program Compilation". Em: *Commun. ACM* 29.2 (fev. de 1986), 126–133. ISSN: 0001-0782. DOI: 10.1145/5657.5661. URL: https://doi.org/10.1145/5657.5661.

[51] V. R. Segovia e A. Theorin. "History of Control History of PLC and DCS". Em: *University of Lund* (2012).

[52] R. Sibois, K. Salminen, M. Siuko, J Mattila e T. Määttä. "Enhancement of the use of digital mock-ups in the verification and validation process for ITER remote handling systems". Em: *Fusion Engineering and Design* 88.9-10 (2013), pp. 2190–2193.

[53] G. Strawn e C. Strawn. "Grace Hopper: Compilers and Cobol". Em: *IT Professional* 17.1 (2015), pp. 62–64.

[54] D. Thapa, C. M. Park, K. H. Han, S. C. Park e G.-N. Wang. "Architecture for modeling, simulation, and execution of PLC based manufacturing system". Em: *2008 Winter Simulation Conference*. IEEE. 2008, pp. 1794–1801.

[55] A. Ujvarosi. "Evolution of SCADA systems". Em: *Bulletin of the Transilvania University of Brasov. Engineering Sciences. Series I* 9.1 (2016), p. 63.

[56] G. P. Zimmerman. "Programmable logic controllers and ladder logic". Em: *Rapid City: Dr. Alfred R. Boysen, Department of Humanities, South Dakota School of Mines and Technology* (2008).

[57] M. Åkerman. "Implementing Shop Floor IT for Industry 4.0". Tese de doutoramento. Jun. de 2018.

Anexo

Interpretador de uma Árvore de Análise Sintática

Listagem I.1: Código em C++ para um interpretador

```
double interpretor::ex(nodeType* p){

    if (!p) return 0;
    switch (p->type) {
    case typeCon: return p->con.value;
    case typeConReal: return p->conReal.value;
    case typeId: return varValue(p);
    case typeDataType: return p->varType.id;
    case typeTimerVar: return timer_var_value(p);
    case typeTimeBase: return p->timeBase.value;
    case typeOpr:
        switch (p->opr.oper) {
        case WHILE: while (ex(p->opr.op[0])) ex(p->opr.op[1]); return 0;
        case REPEAT: do ex(p->opr.op[0]); while (!(ex(p->opr.op[1]))); return 0;
        case FOR: {
            ex(p->opr.op[0]); // nodeAssign
            while (ex(p->opr.op[1])) { // nodeCondLoop
                ex(p->opr.op[3]); // nodeForStatements
                ex(p->opr.op[2]); // nodeReassignLoop
            }
            return 0;
        }
        case REASSIGN: {
            symTable[p->opr.op[0]->id.ident]->word_value = symTable[p->opr.op[0]->id.ident
                ↪ ]->word_value + (int) ex(p->opr.op[1]) ;
            return 0;
        }
```

```
27          case IF: {
28              if (ex(p->opr.op[0]))
29                  ex(p->opr.op[1]);
30              else if (p->opr.nops > 2)
31                  ex(p->opr.op[2]);
32              return 0;
33          }
34          case IF_ELSIF: {
35              if (ex(p->opr.op[0])) {
36                  ex(p->opr.op[1]);
37              }
38              else
39                  if (ex(p->opr.op[2]) == 0 && p->opr.nops == 4)
40                      ex(p->opr.op[3]);
41              return 0;
42          }
43          case ELSIF: {
44              if (p->opr.nops == 3) {
45                  if (ex(p->opr.op[2]) == 1) {
46                      return 1;
47                  }
48                  else {
49                      if (ex(p->opr.op[0])) {
50                          ex(p->opr.op[1]);
51                          return 1;
52                      }
53                      else return 0;
54                  }
55              }
56              else
57                  if (p->opr.nops == 2) {
58                      if (ex(p->opr.op[0])) {
59                          ex(p->opr.op[1]);
60                          return 1;
61                      }
62                      else return 0;
63                  }
64          }
65          case PRINT: std::cout << "Printing variable: " << ex(p->opr.op[0]) << std::endl;
              ↪ return 0;
66          case EXIT: exit(0);
67          case ';': ex(p->opr.op[0]); return ex(p->opr.op[1]);
68          case '=': {
69              double aux = assignValue(p);
70              if (p->opr.op[0]->id.ident[0] == '%')
71              {
72                  std::cout << "Endereco " << std::string(p->opr.op[0]->id.ident) << " com o
                      ↪ valor : " << aux << "\n" ;
73              }
74              return aux;
```

```
75          }
76          case UMINUS: return (double) ( -ex(p->opr.op[0]) );
77          case '+': return (double) ( ex(p->opr.op[0]) + ex(p->opr.op[1]) );
78          case '-': return (double) ( ex(p->opr.op[0]) - ex(p->opr.op[1]) );
79          case '*': return (double) ( ex(p->opr.op[0]) * ex(p->opr.op[1]) );
80          case '/': return (double) ( ex(p->opr.op[0]) / ex(p->opr.op[1]) );
81          case '<': return (double) ( ex(p->opr.op[0]) < ex(p->opr.op[1]) );
82          case '>': return (double) ( ex(p->opr.op[0]) > ex(p->opr.op[1]) );
83          case GE: return (double) ( ex(p->opr.op[0]) >= ex(p->opr.op[1]) );
84          case LE: return (double) ( ex(p->opr.op[0]) <= ex(p->opr.op[1]) );
85          case NE: return (double) ( ex(p->opr.op[0]) != ex(p->opr.op[1]) );
86          case EQ: return (double) ( ex(p->opr.op[0]) == ex(p->opr.op[1]) );
87          case ASSOC: return (double) ex(p->opr.op[0]);
88          case OR: return (double) ex(p->opr.op[0]) || ex(p->opr.op[1]);
89          case XOR: return (double) ( (int) ex(p->opr.op[0]) ^ (int) ex(p->opr.op[1]) );
90          case AND: return (double) ( ex(p->opr.op[0]) && ex(p->opr.op[1]) );
91          case NOT: {
92              if (ex(p->opr.op[0]) > 0)
93                  return 0;
94              else return 1;
95          }
96          case RE: return isRE(std::string(p->opr.op[0]->id.ident));
97          case FE: return isFE(std::string(p->opr.op[0]->id.ident));
98          case DECL: return declareVar(p);
99          case DECL_ADRESS: return declareAdress(p);
100         case DECL_TIMER: return timer_declare(p);
101         case START_TIMER: return timer_start(p);
102         case DOWN_TIMER: return timer_down(p);
103         case PRINT_SEMI_COLON: return ex(p->opr.op[0]);
104         }
105     }
106     return 0;
107 }
```

Anexo

PID escrito em Matlab

Listagem II.1: Código para um PID escrito em Matlab

```matlab
out=zeros(100,1);
out(1) = 20;

setpoint=0;
dt = 0.1;
max = 100;
min = -100;
kp = 0.1;
kd = 0.01;
ki = 0.5;
errorArray = zeros(101,1);
integral = 0;
error = 0;
for k = 2:100
    %out(k,1)=2;
    % calculate error
    error = setpoint - out(k-1);
    %proportional term
    pout = kp * error;
    %integral term
    integral = integral + error*dt;
    iout = ki * integral;
    %derivative term
    derivative = (error - errorArray(k-1)) ./ dt;
    dout = kd * derivative;
    %calculate total output
    u = pout + iout + dout;
    %Restrict total output
    if u>max
```

```
30            u=max;
31        elseif u<min
32            u=min;
33        end
34
35        %Save error to previous error
36        errorArray(k) = error;
37        out(k)=out(k-1)+u;
38    end
39
40
41    timeArray=zeros(100,1);
42    for k = 2:100
43        timeArray(k)=timeArray(k-1)+dt;
44    end
45
46    plot(timeArray,out)
47    }
```

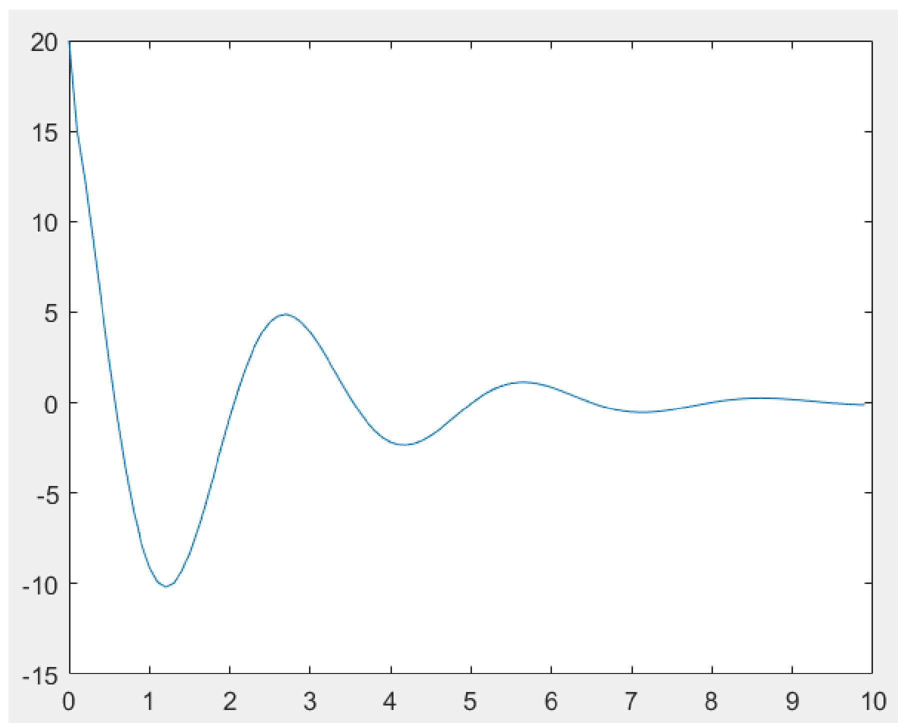

Figura II.1: Resultado do teste para um controlador PID escrito em Matlab.