Faculdade de Engenharia da Universidade do Porto

Ambiente Integrado de Desenvolvimento IEC 61131-3 para Eclipse

Mário David Correia Mendonça

Dissertação realizada no âmbito do
Mestrado Integrado em Engenharia Electrotécnica e de Computadores
Major Automação

Orientador:
Mário Jorge Rodrigues De Sousa (Prof. Dr.)

Julho de 2017

Resumo

Para a automatização de processos industriais, é comum o uso de controladores programáveis (PLC). A programação destes controladores é geralmente realizada recorrendo a linguagens de programação próprias (IL, ST, SFC, LD e FBD), definidas pela norma IEC 61131-3. Apesar de existirem já vários compiladores funcionais para estas linguagens, a quantidade de IDE's disponíveis para programação nas línguas da Norma é reduzida, sendo a grande maioria dos IDE's software comercial. Estes IDEs não só têm a desvantagem de acarretarem muitas vezes custos elevados, também tendem a não implementar totalmente a Norma IEC 61131-3, levando a alguns problemas.

Esta dissertação foi escrita com o objetivo de documentar o desenvolvimento de um novo IDE, integrado com a plataforma Open-Source Eclipse. O trabalho desenvolvido que vai ser apresentado nesta dissertação focou-se especificamente na criação de dois novos editores gráficos, nomeadamente para as linguagens FBD e LD. Estes editores foram criados de forma a poderem ser integrados com o trabalho já realizado por outros estudantes em dissertações anteriores, completando assim a componente de editores gráficos do IDE para IEC 61131-3.

Abstract

In the automation of industrial processes, the use of programmable logic controllers (PLC) is widespread. The programming of these controllers is achieved using a few programming languages specifically designed for this purpose, such languages are defined in the IEC 61131-3 Norm. Despite the existence of many functional compilers for these languages, the amount of available IDE's to available to use them is small and mostly dominated by proprietary and commercial software. These IDE's have some disadvantages, aside from sometimes having a high cost, they also tend to not fully implement the IEC 61131-3 Norm, leading to problems in portability between IDE's.

This dissertation was written with the goal of documenting the development of a new IDE, integrated with the Eclipse Open-Source platform. The work put into this IDE which this dissertation features was mostly focused on the creation of two new graphical editors. These editors implement the FBD and LD programming languages. The editors were integrated with the work already developed by other students in previous dissertations, thus completing the graphical editor component for the IEC 61131-3 IDE.

Agradecimentos

Quero agradecer os meus pais, sem o seu apoio, carinho e motivação não teria sido possível estar onde estou hoje. Estarei para sempre grato por todo o apoio que me deram e continuam a dar ao longo da minha vida, e espero conseguir fazer-vos orgulhosos sempre.

Agradeço também ao meu orientador, Professor Mário Jorge Rodrigues de Sousa, pela disponibilidade e orientação ao longo do desenvolvimento deste projeto.

Por fim, quero agradecer a todos os professores que me ajudaram e ensinaram ao longo da minha vida académica.

Índice

Lista de Figuras

Lista de Tabelas

Abreviaturas e Símbolos

PLC	Programmable Logic Controller
IL	Instruction List
ST	Structured Text
SFC	Sequential Function Chart
LD	Ladder Diagram
FBD	Function Block Diagram
IDE	Integrated Development Environment
POU	Program Organization Units
GEF	Graphical Editing Framework
MVC	Model-View-Controller
EMF	Eclipse Modelling Framework

Capítulo 1

1. Introdução

Esta dissertação, realizada no âmbito do Mestrado Integrado de Engenharia Eletrotécnica e de Computadores, tem o objetivo de documentar o trabalho realizado no estudo, conceção e implementação de um ambiente de desenvolvimento integrado para linguagens da norma IEC 61131-3. O foco desta dissertação é os dois novos editores gráficos que foram desenvolvidos para as linguagens FBD e LD.

Ao longo desta dissertação, serão descritos os vários conceitos considerados no decorrer do desenvolvimento. Ambos os editores criados vão ser apresentados, com a sua estrutura interna e funcionalidades devidamente exploradas, finalmente serão tiradas conclusões acerca do trabalho desenvolvido e será discutido possível trabalho futuro a ser realizado.

Neste capítulo inicial será dada uma breve introdução e enquadramento do projeto, serão apresentados os objetivos do projeto e as motivações.

1.1 Contexto

Para facilitar a compreensão do propósito desta dissertação, esta secção contém alguma informação que permite contextualizar a existência do IDE desenvolvido. Muitos dos conceitos aqui discutidos serão explorados com maior detalhe no capítulo seguinte, sendo o propósito desta secção apenas uma breve introdução.

1.1.1 PLC

Na indústria, cada vez mais são usados os PLC (*Programmable Logic Controller*) para automatização e controlo de vários processos de fabrico em linha, os PLC são computadores programáveis que interagem com atuadores e sensores através de ligações elétricas e remotas.

Ao contrário de um computador tradicional, os PLC estão especificamente preparados para funcionar num ambiente industrial, onde se encontram condições como temperaturas elevadas, vibração, interferências eletromagnéticas e poeiras [5].

Os PLC são também concebidos para terem muito boa resposta em tempo real, possibilitando uma melhor resposta a situações de emergência ou falha, reduzindo assim o risco de acidentes. Os PLC são capazes de funcionar continuamente durante anos sem precisarem de elevada manutenção, sendo perfeitos para utilização em ambientes industriais, onde linhas de montagem raramente podem parar.

1.1.2 IEC 61131

Nos primórdios da sua implementação e utilização, não existiam regras específicas sobre como operar e programar, esta ausência de regras levava a que os PLC tivessem diferentes funcionamentos e formas de programar conforme o seu modelo, fornecedor ou versão. Esta falta de coerência entre diferentes PLC levava a uma dificuldade por parte dos programadores ao desenvolver código e dificultava a portabilidade de software entre diferentes PLC. Devido a esta inflexibilidade aconteciam situações em que, por exemplo, haviam elevados custos de atualização de software caso uma empresa decidisse atualizar o seu hardware, ou caso linguagens utilizadas se tornassem obsoletas em relação a linguagens mais modernas. Para tentar resolver este problema, a norma IEC 61131 foi desenvolvida. A norma IEC 61131, publicada em 1992 pela I*nternational Electrotechnical Convention*, foi criada para definir um conjunto de regras de programação e funcionamento dos PLC's, esta norma é largamente utilizada na programação de PLC's por todo o mundo [1].

A norma IEC 61131 está atualmente organizada em nove secções diferentes, cada uma tratando uma parte específica do funcionamento dos PLC's, desde requisitos de hardware a regras de programação e a exemplos de implementação. A secção três da norma IEC 61131, a IEC 61131-3, diz respeito às cinco diferentes linguagens de programação a utilizar nos PLC. Estas linguagens são a IL (*Instruction List*), ST (*Structured Text*), SFC (*Sequential Function Chart*), LD (*Ladder Diagram*) e FBD (*Function Block Diagram*). As linguagens IL e ST são linguagens textuais, uma de baixo nível e uma de mais alto nível, semelhantes ao Assembly e ao C, respetivamente. As linguagens SFC, LD e FBD são todas linguagens gráficas com regras e níveis de complexidade variados. Através destas linguagens é que são programados a maioria dos PLCs atualmente utilizados no mundo da automação industrial.

1.1.3 Ferramentas de programação IEC 61131

Para a programação de PLCs nas linguagens da IEC 61131 existem alguns ambientes integrados de desenvolvimento (IDE's) dos quais se pode escolher. Estes IDE's são programas que através de interfaces gráficas e textuais permitem a fácil criação, interpretação e modificação de programas nas linguagens da Norma. Para além de suportarem as linguagens de programação comuns, estes IDE's geralmente têm componentes que permitem a configuração de PLCs e que fazem a gestão de outros aspetos do seu funcionamento, tais como a consulta e modificação das suas variáveis, a gestão de comunicações entre equipamentos e a configuração de tarefas. Tudo isto é feito conforme o que é ditado pela norma IEC 61131.

Existem alguns IDE's comerciais populares, estes programas têm a desvantagem de não serem gratuitos, sendo necessário pagar para utilizar todas as suas funcionalidades, também têm a desvantagem de não adotarem a norma IEC 61131 na sua totalidade, dificultando a fácil reutilização e portabilidade de código entre IDE's diferentes [4]. Exemplos de IDE's comerciais são o ISaGRAF, o CoDeSys, e o Unity Pro [1], estes IDE's são geralmente parecidos em estrutura, diferindo apenas em alguns aspetos. Estes IDE's são bons a proporcionar uma interface funcional na qual podem ser desenvolvidos programas para PLC's.

Para além destas opções comerciais, existem ainda alguns IDE's populares OpenSource, o Beremiz é um exemplo destes IDE's. O Beremiz, desenvolvido pela empresa francesa LOLITECH em conjunto com a Universidade do Porto, é um IDE gratuito para IEC 61131 com o propósito de fornecer aos engenheiros de automação uma ferramenta robusta e boa para a programação de PLC. Um objetivo chave deste IDE é seguir totalmente a norma IEC 61131, algo que não é totalmente feito em outros IDE's comerciais que leva a dificuldades de portabilidade de programas entre diferentes IDE's. [4]

1.2 Motivação

Apesar de existirem várias opções de IDE's para o desenvolvimento de programas nas linguagens da norma IEC 61131-3, a maioria destes são comerciais, podendo vir a ter custos elevados. Uma popular alternativa gratuita, o Beremiz, é competente nas suas funcionalidades, mas quando comparado com outros IDE's tais como o CoDeSys ou o ISaGRAF deixa algo a desejar, especialmente na sua componente de editores gráficos.

A motivação desta dissertação, tal como a das que foram realizadas no mesmo âmbito por alunos anteriores, foi a criação de um IDE livre, modular e fácil de implementar que consiga ser uma alternativa boa aos IDE's existentes, de forma a oferecer mais opções a programadores e a estudantes. Um outro ponto interessante do desenvolvimento deste IDE é a capacidade de ser estendido para a nova edição da norma IEC 61131-3, que apesar de não vir a ser abordada nesta dissertação, poderá ser alvo de futuros desenvolvimentos.

1.3 Objetivos

No início desta dissertação, encontravam-se já desenvolvidos os editores para três das cinco linguagens definidas pela Norma IEC 61131-3. Os editores existentes permitiam programar nas linguagens ST, IL e SFC. Em termos de objetivos, esta dissertação tinha dois objetivos principais claros:

- Criação de um editor gráfico capaz de suportar a linguagem FBD, utilizando a plataforma Eclipse e a *framework* Graphiti;
- Criação de um editor gráfico capaz de suportar a linguagem LD, utilizando a plataforma Eclipse e a *framework* Graphiti;

Para além destes objetivos principais, alguns objetivos opcionais foram estabelecidos, estes objetivos eram para ser completos caso o tempo restante permitisse. Os objetivos opcionais eram os seguintes:

- Integração dos novos editores com o navegador de projetos IEC 61131-3 previamente estabelecido;
- Integração dos componentes do editor FBD com o editor LD;
- Integração dos componentes do editor FBD e LD com o editor SFC;
- Criação de um módulo de exportação/importação XML para programas FBD;
- Criação de um módulo de exportação/importação XML para programas SFC;

Destes objetivos opcionais, só foi possível concluir os dois primeiros. A razão pela qual não se pode concluir os outros objetivos opcionais foi maioritariamente por limitações de tempo. O terceiro objetivo opcional, a integração dos novos editores no editor SFC, poderá ser possível no futuro, mas será necessária a reestruturação e reescrita de grandes partes do editor SFC para a suportar corretamente.

Os módulos de importação/exportação XML iriam necessitar de muito tempo de desenvolvimento para serem construídos corretamente. Como este tempo já não era disponível no final da criação dos editores gráficos, escolheu-se utilizar o tempo restante para aperfeiçoar os editores criados, de forma a que estes tivessem um muito bom desempenho.

1.4 Estrutura da dissertação

Esta dissertação encontra-se dividida em quatro capítulos, estes capítulos encontram-se organizados de tal forma que seja simples de compreender o trabalho desenvolvido e acompanhar o seu desenvolvimento.

O primeiro capítulo, a Introdução, serve para enquadrar o projeto desenvolvido, expor os objetivos propostos e dar uma melhor informação do propósito da dissertação e da sua estrutura.

O segundo capítulo, Revisão Bibliográfica, irá explorar os vários conceitos envolvidos na criação do projeto. Será neste capítulo que se irá falar sobre tópicos relativos à Norma IEC 61131-3 em si, o ambiente integrado de desenvolvimento Open-*Source* Eclipse, onde o projeto está a ser desenvolvido. Neste capítulo serão também exploradas as ferramentas utilizadas ao longo do desenvolvimento do projeto, nomeadamente o *Graphiti*, uma ferramenta que permite a criação de editores gráficos, e o EMF, uma ferramenta fácil de utilizar que permite a criação de metamodelos.

O terceiro capítulo, Desenvolvimento, contém a descrição dos vários componentes e funcionalidades implementadas. Neste capítulo pode-se encontrar documentação relativa aos metamodelos desenvolvidos, aos plug-ins criados e às várias funcionalidades e suas respetivas classes e métodos.

No quarto capítulo, Testes e Conclusões, são descritos alguns testes realizados aos editores implementados, o propósito destes testes é demonstrar a robustez das funcionalidades dos editores novos e expor algumas particularidades sobre os mesmos. Neste capítulo são dadas algumas apreciações gerais do trabalho desenvolvido, são discutidos e justificados alguns problemas com os editores desenvolvidos e com o projeto em si. Finalmente, é discutido o trabalho que terá de ser realizado numa dissertação futura.

Capítulo 2

2. Revisão Bibliográfica

Neste capítulo serão explorados vários tópicos relevantes para o desenvolvimento do projeto. O projeto trabalha com a Norma IEC 61131-3, será este o primeiro tópico que será abordado neste capítulo. Para além da Norma, o projeto depende de conhecimentos relacionados com o Eclipse e a programação de plug-ins para o mesmo, sendo o Eclipse o segundo ponto a ser tratado neste capítulo. Ainda na área de programação Eclipse, será explorada a ferramenta *Graphiti*, que foi utilizada no desenvolvimento do projeto.

Para dar uma perspetiva do tipo de IDE's competidores, serão brevemente mencionadas algumas alternativas para a programação na Norma. Este tópico não será alvo de grande discussão visto que o foco principal está no desenvolvimento de *plug-ins* para Eclipse.

Finalmente será demonstrado o trabalho que já se encontrava realizado no início da dissertação. Esta dissertação é a quarta numa série de trabalhos relacionados com a criação de um ambiente de desenvolvimento IEC 61131-3 para Eclipse, sendo necessário mostrar o trabalho já existente no início do desenvolvimento.

2.1 IEC 61131-3

A Norma IEC 61131 é uma norma introduzida em 1993 com o objetivo de normalizar vários aspetos da automação industrial. Esta norma atualmente é utilizada em todo o mundo para a programação de PLC's. A Norma está dividida em nove partes, cada uma das nove partes trata de um tópico diferente relacionado com automação industrial. As nove partes da Norma são[15]:

- IEC 61131-1: Informações gerais;
- IEC 61131-2: Requisitos de equipamento e testes;
- IEC 61131-3: Linguagens de programação;
- IEC 61131-4: Diretrizes de utilizador;
- IEC 61131-5: Especificação de serviços de comunicação;
- IEC 61131-6: Segurança funcional;
- IEC 61131-7: Programação de controladores com lógica difusa;
- IEC 61131-8: Diretrizes para implementação de linguagens de programação;
- IEC 61131-9: (SDCI) Interfaces de comunicação digitais "single-drop" para sensores e atuadores pequenos;

Destas nove secções da Norma, a terceira secção é de principal interesse para esta dissertação. A terceira secção da norma IEC 61131, a IEC 61131-3, trata das diferentes linguagens de programação a utilizar quando se programa PLC's. Nesta parte da norma, pode-se dividir a informação em duas partes [6], os elementos comuns a todas as linguagens de programação e as linguagens em si.

2.1.1 Tipos de dados e variáveis

Na norma IEC 61131 todos os parâmetros utilizados dentro dos programas têm um tipo de dados específico, estes tipos de dados definem a forma como podem ser utilizados os parâmetros e como estes são tratados internamente. Alguns tipos de dados padrão são *boolean*, *integer*, *real*, *byte*, *word*, *date*, *time_of*_day e *string* [6], o utilizador pode definir outros tipos de dados baseando-se em tipos já existentes através da criação de estruturas e vetores.

As variáveis são semelhantes a variáveis de outras linguagens de programação, obedecendo às regras definidas pelos seus tipos de dados. Podem ser locais (sendo possível reutilizar os seus nomes noutros locais no código sem problemas) ou globais. Uma diferença que estas variáveis têm em relação às variáveis normalmente utilizadas em programação está na forma como elas são definidas. Estas variáveis podem ser utilizadas em programas como variáveis convencionais, mas também podem ser associadas a endereços de input/output específicos no hardware. É através destas variáveis externas que o PLC comunica com o material (sensores, atuadores, etc…) a que está ligado, e é nestas variáveis que são guardados parâmetros durante a execução dos programas.

2.1.2 Configurações, Resources e Tasks

Os projetos da norma IEC 61131-3 estão organizados de uma maneira específica. A um projeto/PLC geralmente é associada uma única configuração que define aspetos importantes como o hardware utilizado, cartas de entradas/saídas a onde são ligados os vários sensores e atuadores, recursos de memória, recursos de processamento e a forma como é feita a comunicação.

Dentro da configuração pode-se definir ***Resources*** (recursos), os *Resources* funcionam como "blocos" independentes, correspondendo geralmente a unidades de processamento diferentes, são os *Resources* que executam os programas.

Dentro dos *Resources* tem-se conjuntos de ***Tasks*** (tarefas), estas tarefas representam os programas a realizar. Cada *Task* pode ter um tipo de funcionamento diferente, podendo ser executados periodicamente ou através de *triggers*, como a alteração de uma variável. A configuração destas *Tasks* permite a criação de diferentes tipos de comportamentos, sendo possível criar programas com maior prioridade e limites de tempo-real exigentes, que permitem por exemplo interromper o funcionamento normal e desempenhar uma tarefa específica numa situação de falha, como interromper o funcionamento e acionar um alarme visual.

2.1.3 POU's

Os POU's (*Program Organization Units*), são os blocos básicos de construção de programas definidos pela IEC 61131-3, estão divididos em 3 subclasses:

- Funções (FUNC): Este POU é semelhante às funções convencionais de outras linguagens de programação, recebe um conjunto de entradas e retorna um valor. A norma IEC define algumas funções predefinidas como adição, valor absoluto, raiz quadrada, seno e cosseno. A Função não tem memória interna, tendo sempre o mesmo *output* para o mesmo conjunto de *inputs*. O utilizador pode definir as suas próprias funções, que pode depois reutilizar noutros POU's;
- *Function Block*(FB): Este POU funciona de forma semelhante a um circuito integrado de eletrónica. Ao contrário das funções, um FB pode conter variáveis internas que mantêm o seu valor entre execuções. Este funcionamento permite uma maior versatilidade relativamente às Funções. O código que está no interior do FB está escondido do utilizador, sendo só visíveis os *inputs* e *outputs* e o nome do FB. Este encapsulamento do funcionamento interno serve para separar diferentes níveis de programação. Esta diferenciação dos vários níveis permite a criação de programas

complexos através da utilização e reutilização de cadeias de FBs e Functions. É possível ao utilizador definir os seus próprios FB's utilizando qualquer uma das linguagens da IEC 61131-3. Graças à sua modularidade os FB's são uma ferramenta muito útil para a construção de programas complexos;

- Programa (PROG): Este tipo de POU representa um programa de mais alto nível (análogo ao "main" de outras linguagens de programação), pode conter FB's e funções e ser escrito em qualquer uma das linguagens da Norma. Normalmente é através deste tipo de POU que todos os outros POU's são acedidos;

2.1.4 Linguagens de programação

Na norma IEC 61131-3 são definidas cinco linguagens de programação diferentes. Estas são as linguagens na qual são escritos os programas a executar nos PLC's. Existem três linguagens gráficas e duas textuais. Sendo o objetivo desta dissertação o desenvolvimento da interface para programação de duas destas linguagens, LD e FBD, estas duas linguagens serão descritas mais exaustivamente na secção seguinte, sendo apenas referidas na lista seguinte. As linguagens definidas pela norma IEC 61131-3 são:

- ***Instruction List* (IL):** A linguagem IL é uma linguagem textual, semelhante ao código *assembly*, não é muito apropriada para a criação de programas complexos e é geralmente pouco utilizada, mas é mantida por uma questão histórica e porque algum software mais antigo pode estar escrito ainda em IL;

- ***Ladder Diagram*(LD):** A linguagem LD é uma linguagem gráfica baseada na lógica de relés que antecedeu os PLC's. Nesta linguagem é comum trabalhar-se com entradas e saídas do tipo *boolean*, correspondentes a sinais *HIGH* e *LOW* na eletrónica. É possível definir operações lógicas como AND e OR através da disposição de componentes chamados *Contacts*, que servem para a deteção de variáveis de *input* e podem ter vários tipos de funcionamento. As variáveis de *output* são por sua vez manipuladas através de componentes chamados *Coils* que podem também ter diferentes comportamentos conforme o seu tipo. Nesta linguagem é também possível a utilização de Function Blocks e Functions para a criação de programas mais complexos;

- ***Structured Text* (ST):** O ST é uma linguagem textual muito próxima de linguagens como C, Pascal e outras. Contém todos os elementos essenciais de uma linguagem de programação moderna, permitindo programar funções complexas facilmente;

- ***Function Block Diagram*(FBD):** Linguagem gráfica que lida com a utilização de diferentes Function Blocks e Functions interligados, muito popular na indústria de processos. Cada Function Block desempenha uma só função e cabe ao programador definir as interligações entre Function Blocks. Devido à capacidade de Function Blocks serem definidos através de qualquer outra linguagem, é possível a reutilização de blocos básicos várias vezes para criar programas bastante complexos. Devido ao encapsulamento dos diferentes Function Blocks esta linguagem permite uma fácil interpretação e separação entre diferentes níveis de controlo;

- ***Structured Flow Chart* (SFC):** Ao contrário das outras linguagens de programação, o SFC não pode ser utilizado exclusivamente na criação de um programa. Esta linguagem é uma linguagem gráfica baseada em máquinas de estados. Em termos de aspeto é semelhante a um diagrama de fluxo (*flowchart*) ou um Grafcet. Uma importante distinção entre SFC e uma máquina de estados convencional é a capacidade de existir mais que um estado ativo a cada momento, sendo possível a divergência e convergência de fluxos ao longo do funcionamento de um programa. Devido às semelhanças com um *flowchart* tradicional, o SFC é muito fácil de

compreender e é a melhor linguagem a utilizar para facilmente demonstrar o funcionamento de um programa;

2.1.4.1 Ladder Diagrams

Como foi referido na secção anterior, o LD é uma das três linguagens gráficas de programação definida pela Norma IEC 61131-3. Esta linguagem é baseada na lógica cablada com relés que antecedeu os PLC's. Devido a esta inspiração em circuitos físicos, programas LD funcionam de uma forma parecida com circuitos elétricos reais. As variáveis tratadas por diagramas LD são geralmente booleanas, com um funcionamento equivalente a níveis HIGH e LOW da lógica cablada.

Os diagramas LD têm poucos elementos base, uns elementos que são comuns a todos os diagramas são os *PowerRails* à esquerda e à direita. Estes elementos são essenciais para qualquer programa nesta linguagem. Os *PowerRails* servem como barramentos contendo o nível HIGH e LOW (PowerRail esquerdo e direito, respetivamente). Aos PowerRails são ligados todos os outros componentes funcionais de um diagrama LD, sendo o fluxo de energia o que dita a sequência de funcionamento de um programa. Os PowerRails tomam a forma de duas linhas horizontais de onde podem ser feitas ligações. Estes PowerRails ditam o sentido de leitura do programa, que é iniciada no PowerRail da esquerda e terminada no PowerRail da direita.

Aos PowerRails são ligados os outros componentes base da linguagem LD, as ***Coils*** e os ***Contacts***. Os *Contacts* funcionam como um impedimento ao fluxo de energia entre um *PowerRail* e o outro, estes *Contacts* representam relés físicos e normalmente têm uma variável que lhes é designada. Dependendo do tipo de *Contact* em questão, a resposta à mudança da variável que lhes é atribuída será diferente, os tipos de *Contacts* existentes, com as suas respetivas representações gráficas, são:

- *Contacts* Simples (representação -| |-): impedem a passagem de sinal do lado esquerdo para o lado direito até a variável que lhes é associada ser ***true***;
- *Contacts* Invertidos (representação -|/|-): permitem a passagem de sinal até que a variável que lhes é associada passa a ***true***;
- *Contacts Rising/Falling Edge* (representação -|RE|- e -|FE|-, respetivamente):: bloqueiam a passagem de sinal até detetarem uma mudança da variável que lhes é associada, ***true*** para ***false*** no *rising edge* e ***false*** para ***true*** no *falling edge*.

Semelhantes aos *Contacts*, as ***Coils*** representam atuadores que estão ligados ao circuito de relés. Tal como para os *Contacts,* as *Coils* têm uma variável que lhes é associada na qual elas atuam. Dependendo do tipo de *Coil*, a resposta a uma mudança de sinal no seu lado esquerdo é diferente. Os tipos de *Coils* existentes, e as suas representações gráficas, são as seguintes:

- *Coils* Simples (representação -( )-): colocam o valor do sinal à sua esquerda na variável que lhes é associada;
- *Coils* Invertidas (representação -(/)-): colocam o inverso do sinal à sua esquerda na variável que lhes é associada;
- *Coils Set* (representação -(S)-): quando recebem um sinal ***true*** no seu pino esquerdo, colocam a variável que lhes é associada a ***true***, esta variável mantém-se neste estado até ser ativada uma *Coil "reset"* correspondente;
- *Coils Reset* (representação -(R)-): quando são ativadas, revertem quaisquer outros comandos de *"set"* que lhes tenham sido atribuídos;
- *Coils Rising Edge/Falling Edge* (representação -(RE)- e -(FE)-, respetivamente): estas *Coils* colocam a sua variável associada a ***true*** sempre que detetam um *rising edge*/*falling edge* à sua esquerda (mudança de sinal de ***true*** para ***false*** e vice-versa, respetivamente);

Num diagrama LD, os vários *Coils* e *Contacts* são colocados e interligados. Através de diferentes combinações de componentes e ligações é possível realizar quaisquer operações lógicas complexas de uma forma simples de interpretar.

De forma a exemplificar o funcionamento de um circuito LD, encontra-se na figura seguinte (Fig. 1.) uma representação de uma função “AND” entre duas variáveis “var1” e “var2”, neste caso a variável “var3” deve-se ligar se e só se ambas as variáveis “var1” e “var2” estiverem ***true***.

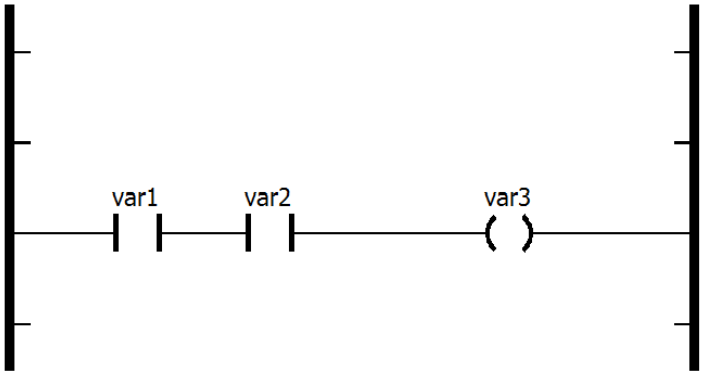


Fig. 1. – Exemplo de diagrama LD

Como pode ser observado na figura, quando a “var1” é ***true***, o fluxo de energia passa para o lado esquerdo do Contact seguinte, quando a “var2” é ***true***, o fluxo de energia é permitido até ao lado esquerdo da *Coil*. Para que a *Coil* seja ativa, e a “var3” seja colocada a ***true***, é necessário que ambos os *Contacts* sejam ativos ao mesmo tempo. Efetivamente sendo criada a condição “var3=var1 AND var2”.

Para expandir a funcionalidade dos diagramas LD, é possível inserir Functions e Function Blocks. Estes componentes permitem funcionamentos mais complexos e a divisão de diferentes níveis de código. Um exemplo deste funcionamento é a criação de uma função “XOR”. Esta função leva dois *inputs* e dá um *output*. O *output* desta função está a ***true*** quando só uma das suas entradas é ***true***, ou seja, se ambos os seus *inputs* tiverem o mesmo sinal, o output é ***false***. Programar esta função é possível em LD, sendo uma possível implementação desta função visível na figura seguinte (Fig. 2.).

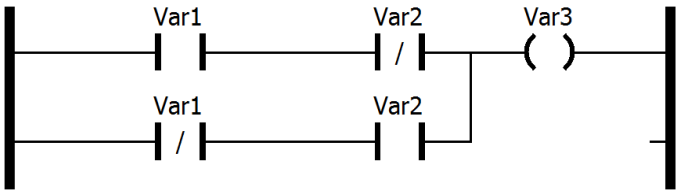


Fig. 2. – Exemplo de função XOR criada em LD

Como se pode observar na figura, a função é composta por duas linhas diferentes, sendo o valor de “var3” possível exprimir da seguinte forma: “var3 = (Var1 AND NOT Var2) OR (NOT Var1 AND Var2)”.

Apesar deste tipo de construção permitir a implementação da função XOR, é possível que seja necessário utilizar esta operação base várias vezes ao longo da criação de um programa complexo. Neste caso não se torna prático construir sempre o diagrama acima quando se quer realizar a função XOR. Para facilitar a programação torna-se mais viável definir uma *Function* que possa ser reutilizada para esta operação. A biblioteca de funções *standard* da Norma IEC 61131-3 já inclui uma função para XOR. Mas para fim de demonstração será exemplificada uma possível implementação da mesma função na figura seguinte (Fig. 3.).

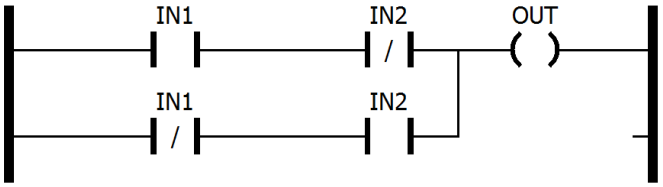


Fig. 3. – Função XOR reconfigurada para ser utilizada num Function Block.

Criando um novo POU do tipo Function com a linguagem LD e copiando o diagrama previamente criado para XOR, declarando as variáveis “Var1”, “Var2” e “Var3” como “IN1”, “IN2” e “OUT”, do tipo BOOL e de classe INPUT (no caso das primeiras duas variáveis) e OUTPUT, define-se o cabeçalho que declara a nova *Function* como tal. Depois deste passo, torna-se possível a utilização da função XOR criada, como pode ser visto na figura seguinte.

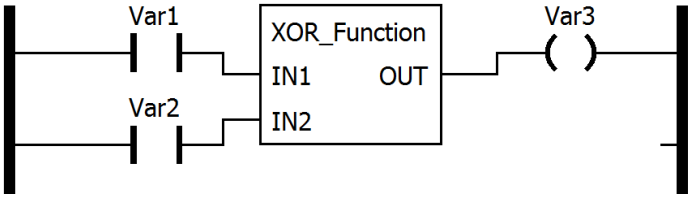


Fig. 4. – Exemplo de utilização da função XOR, como um Function Block.

Como pode ser observado, este tipo de implementação permite reduzir o número de componentes a colocar para cada XOR que se pretenda realizar. Os componentes de diagramas FB têm muita versatilidade, sendo possível criar vários tipos de componentes de elevada complexidade que podem ser reutilizados conforme necessário.

2.1.4.2 *Function Block Diagram*

Os **Function Blocks** são objetos muito utilizados na programação na norma IEC 61131-3, estes objetos têm um aspeto semelhante a circuitos integrados, são uma caixa com várias entradas e saídas das quais não se sabe o aspeto interno. Os ***Function Blocks*** são caracterizados por poderem utilizar variáveis internas que mantêm o seu valor entre utilizações, desta forma é possível programar ***Function Blocks*** cuja resposta varia em função não só dos seus *inputs*, mas também em função das utilizações anteriores. Esta capacidade de manter variáveis internas possibilita a fácil implementação de controladores tais como os PID.

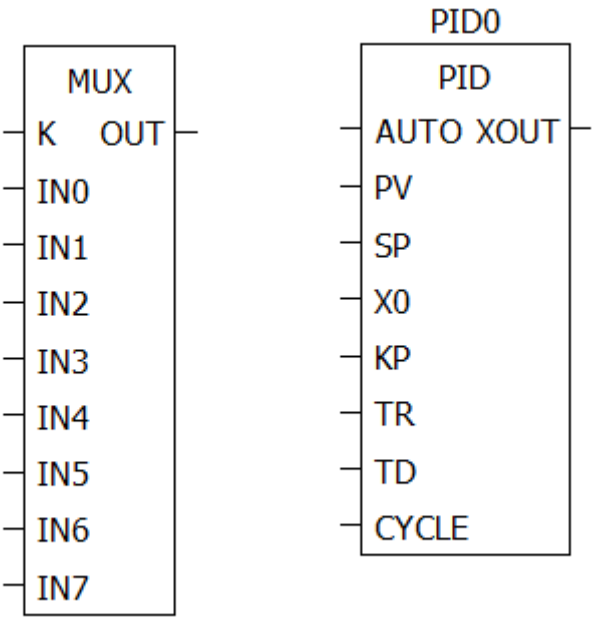


Fig. 5. – Exemplo de Function e Function Block. Um multiplex e um controlador PID.

As ***Functions*** têm um aspeto semelhante aos ***Function Blocks***, mas não permitem a utilização de variáveis internas que mantêm o seu valor entre utilizações. Ao contrário dos *Function Blocks* as *Functions* têm sempre a mesma resposta para o mesmo conjunto de inputs. Os FBs e *Functions* podem ser escritos e utilizados em todas as linguagens da norma. No entanto **a** linguagem **FBD, *Function Block Diagram***, é especialmente concebida para sua utilização. Os FBD, ou diagramas de *Function Blocks*. São um tipo de programas definidos pela norma IEC 61131-3 em que todo o programa é escrito através da interligação de vários *Function Blocks e Functions*.

É comum num programa escrito em FBD ver-se uma elevada separação em níveis de controlo, por exemplo num PLC que deva controlar uma série de tapetes rolantes que têm vários sensores e atuadores cada. É possível a um utilizador conceber um controlador simples para controlar um destes tapetes rolantes lendo os seus sensores e atuando nos seus motores para transportar cargas. O utilizador pode depois utilizar este controlador simples e defini-lo como um *Function Block* genérico. Através da interligação de diferentes *Function Blocks*, cada um com um tapete rolante associado, torna-se possível construir um programa de alto nível que se encarregue de gerir a movimentação de cargas ao longo de uma linha de montagem, sem que seja necessário ao utilizador programar o controlador de cada tapete rolante individualmente. Esta modularidade é um aspeto essencial da linguagem FBD, sendo esta linguagem extremamente poderosa para qualquer tipo de controlo e automação industrial.

2.2 Eclipse

A plataforma **Eclipse** foi desenvolvida em 1998 pelo IBM, com o objetivo de criar um conjunto de ferramentas comuns que programadores pudessem utilizar em vários tipos diferentes de projetos de desenvolvimento de *software*. Um objetivo da IBM para o Eclipse foi a criação de uma comunidade de pessoas que pudessem desenvolver e utilizar a sua plataforma. Em Novembro de 2001 a IBM adotou o modelo *Open-Source* de forma a aumentar o número de pessoas a utilizar o Eclipse e permitir um desenvolvimento maior por parte da comunidade.

O Eclipse adotou uma versão diferente do modelo Open-Source, neste modelo a plataforma em si era ainda aberta e gratuita, mas empresas comerciais eram ainda incentivadas a desenvolver e vender ferramentas dentro da plataforma. O IBM continuou a desenvolver conteúdo para o Eclipse, em conjunto com a comunidade que tinha criado.

O Eclipse é um software modular, capaz de ser expandido por qualquer um. A principal maneira de expandir as funcionalidades do Eclipse funciona através de plug-ins. Estes plug-ins são módulos independentes que podem comunicar entre si através de interfaces denominadas de **extension points** (Fig. 6.).

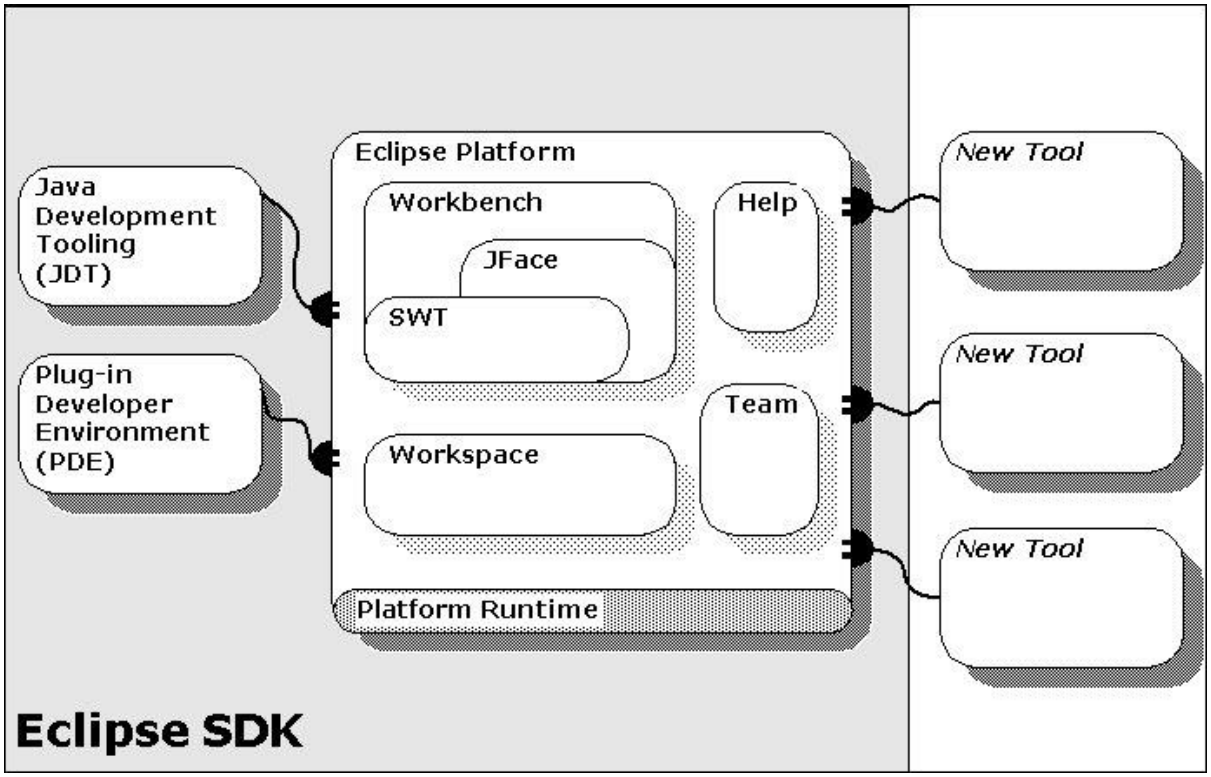


Fig. 6. – Arquitetura interna do Eclipse [11]

O desenvolvimento de plug-ins para o Eclipse envolve sempre a escolha de um *extension point* a utilizar, sendo depois desenvolvido o novo plug-in e integrado na plataforma. O novo plug-in pode também definir novos tipos de *extension point*, que poderão ser utilizados por outros para futura expansão. Esta natureza à base de *extension points* e plug-ins resulta numa elevada liberdade no que conta a funcionalidades novas que podem ser implementadas, sendo mantida sempre uma estrutura comum que é fácil aos outros de interpretar e expandir.

2.2.1 Workbench

O **Workbench** (Fig. 7.) é a janela principal do Eclipse, é a partir dele que todas as outras funcionalidades da plataforma podem ser acedidas. O Workbench não passa de uma janela onde se podem colocar várias componentes visuais, estas partes têm vários tipos diferentes, sendo que dois tipos de objeto são particularmente importantes, os **Views** e **Editors**.

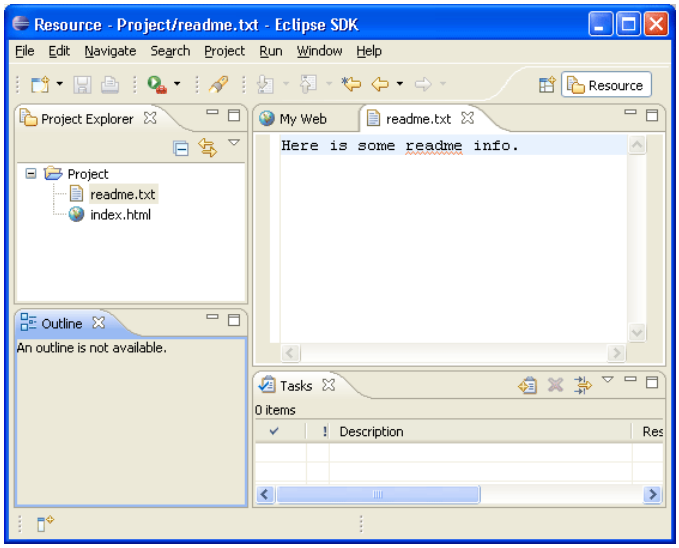


Fig. 7. – Janela do Workbench [11]

2.2.2 Editors

Os **Editors** (Fig. 8.) são ferramentas que permitem ao utilizador criar e editar ficheiros, é nos *editors* que o código é desenvolvido. Para permitir o desenvolvimento de ficheiros, os *editors* oferecem funções que permitem criar, abrir, guardar e fechar documentos.

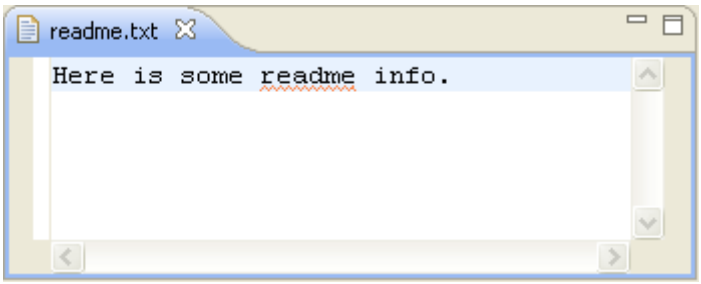


Fig. 8. – Exemplo de um Editor de texto [11]

Novos *editors*, tais como editores de Java, C/C++, HTML, são obtidos como plug-ins, estes *editors* são as ferramentas essenciais que dão funcionalidade ao Eclipse como um IDE para criação de software.

2.2.3 Views

As **Views** (Fig. 9.) são janelas que permitem observar informação relativa a um objeto no qual se está a trabalhar no Workbench, a informação visível numa *view* pode mudar à medida que o utilizador realiza alterações. As *views* tendem a apoiar o trabalho que está a ser feito no editor com as informações que mostram, podendo tomar várias formas desde tabelas, gráficos, navegadores e outras.

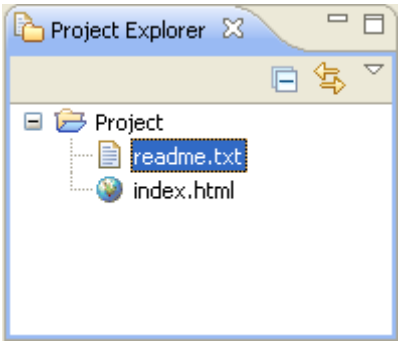


Fig. 9. – Exemplo de uma View [11]

2.2.4 Extension Points

O Eclipse é uma plataforma que funciona à base de plug-ins, estes plug-ins são componentes distintos que adicionam funcionalidades à plataforma. Para o desenvolvimento de plug-ins para o Eclipse é preciso a escolha de como estes são integrados na plataforma principal. A forma pela qual o Eclipse define como são integrados os módulos e como estes comunicam com a plataforma e entre eles é o uso de algo chamado ***extension points***.

Os *extension points* são interfaces pré-definidas pelo Eclipse, ou definidas por outros módulos, que descrevem a forma como os módulos podem comunicar e ser interpretados. Alguns exemplos de *extension points* básicos para o Workbench são:

- org.eclipse.ui.views
- org.eclipse.ui.editors
- org.eclipse.ui.commands
- org.eclipse.ui.menus
- org.eclipse.ui.handlers
- org.eclipse.ui.bindings

Da lista anterior, os primeiros dois *extension points*, org.eclipse.ui.views e org.eclipse.ui.editors são os *extension points* utilizados para a criação de Views e Editors, os outros *extension points* servem para a definição dos vários outros componentes do *Workbench*. Para a criação de um plug-in, é necessária a escolha de um *extension point* a utilizar, sendo necessário declarar num documento plugin.xml o *extension point* (Fig.10.).

```
<extension
   point="org.eclipse.ui.views">
      <category
            id="org.eclipse.ui.examples.contributions.viewCategory"
            name="%contributions.viewCategory.name">
      </category>
      <view
            category="org.eclipse.ui.examples.contributions.viewCategory"
            class="org.eclipse.ui.examples.contributions.view.InfoView"
            id="org.eclipse.ui.examples.contributions.view"
            name="%contributions.view.name">
      </view>
</extension>
```

Fig. 10. – Exemplo da declaração de um extension point num ficheiro plugin.xml [11]

2.2.5 Ferramentas Gráficas

Como o trabalho desenvolvido para esta dissertação envolveu a criação de editores gráficos, foram usadas algumas ferramentas que permitiram a criação destes componentes. A ferramenta que foi escolhida nas dissertações prévias para criar o editor gráfico de SFC foi o ***Graphiti***, esta ferramenta funciona utilizando componentes da *framework* **GEF**. Tal como para o editor SFC, o Graphiti foi escolhido na criação dos editores FBD e LD. De seguida serão abordadas estas ferramentas, sendo o foco principal o *Graphiti.*

2.2.5.1 GEF

A ferramenta **Graphical Editing Framework (GEF)** é utilizada para a criação de interfaces gráficas de utilizador. Esta *framework* está dividida em três plug-ins, sendo eles:

- Draw2d – *toolkit* básico para criação eficiente de layouts
- GEF – *framework* MVC interativa que utiliza Draw2d
- Zest – *toolkit* de visualização que utiliza Draw2d

O componente GEF é o plug-in que é utilizado para a criação de gráficos e interfaces 2D, este plug-in segue um modelo *Model-View-Controller* (**MVC**) (Fig. 11.)

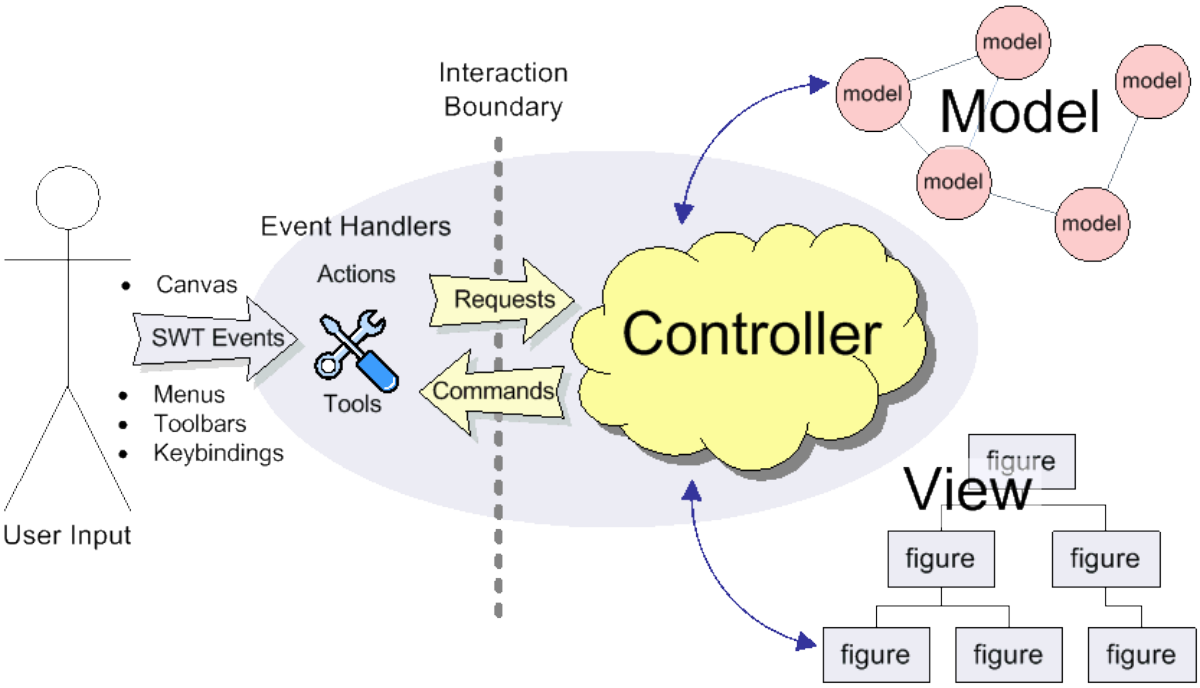


Fig. 11. – Ilustração de arquitetura MVC [11]

No modelo MVC acima ilustrado, o GEF está representado como a região no centro, é ele que faz a ligação entre o modelo e a vista de uma aplicação. O utilizador pode interagir com o controlador através de inputs específicos que são processados para *Requests* e *Commands*. Os *Requests* e *Commands* são utilizados de forma a encapsular as interações e os seus efeitos no modelo.

O modelo do sistema é a forma que os dados tomam internamente, é aqui que todos os dados visíveis na representação gráfica se encontram. Quando o utilizador altera alguma coisa no programa, geralmente só o modelo é que é alterado diretamente, sendo a vista alterada como consequência.

A vista de uma aplicação é tudo o que o GEF permite ao utilizador ver, ou seja, uma representação gráfica da informação contida no modelo. Cabe ao GEF manter esta vista atualizada enquanto o utilizador insere comandos e interpretar os dados do modelo para a criação das figuras.

Para cada elemento visual, geralmente existe um controlador. Este controlador está responsável por editar o elemento, e é ele que faz a interligação entre o modelo e a vista.

2.2.5.2 Graphiti

O ***Graphiti*** é uma ferramenta gráfica que permite a criação simples de editores gráficos e *views* que permitem analizar os modelos internos desses editores. O *Graphiti* utiliza dois componentes do Eclipse, o Eclipse *Modelling Framework* (EMF) e o GEF. O EMF é um componente que permite a criação de modelos e o GEF permite a criação dos editores gráficos. Um problema com o GEF é que, apesar de ser poderoso, é também exageradamente complexo e requer muito esforço e um grande nível de estudo e habituação para a criação de editores. O *Graphiti* foi criado com o objetivo de fornecer uma alternativa mais simples à criação de componentes gráficos. O *Graphiti* essencialmente esconde as partes mais complexas do GEF e permite ao utilizador criar algo sem que seja necessário grande tempo de habituação e aprendizagem. Apesar do *Graphiti* ser uma boa ferramenta para a criação de editores gráficos, o facto de ser uma versão mais simplificada do GEF, uma ferramenta de maior complexidade, resulta num menor nível de liberdade em relação ao que pode ser feito.

A organização interna do *Graphiti* (Fig. 12.) é semelhante à do GEF, sendo fornecido ao utilizador uma visualização do modelo a ser desenvolvido, todas as alterações ao modelo em si são tratadas pelo *Graphiti*, sendo necessário ao utilizador apenas interação com a interface fornecida.

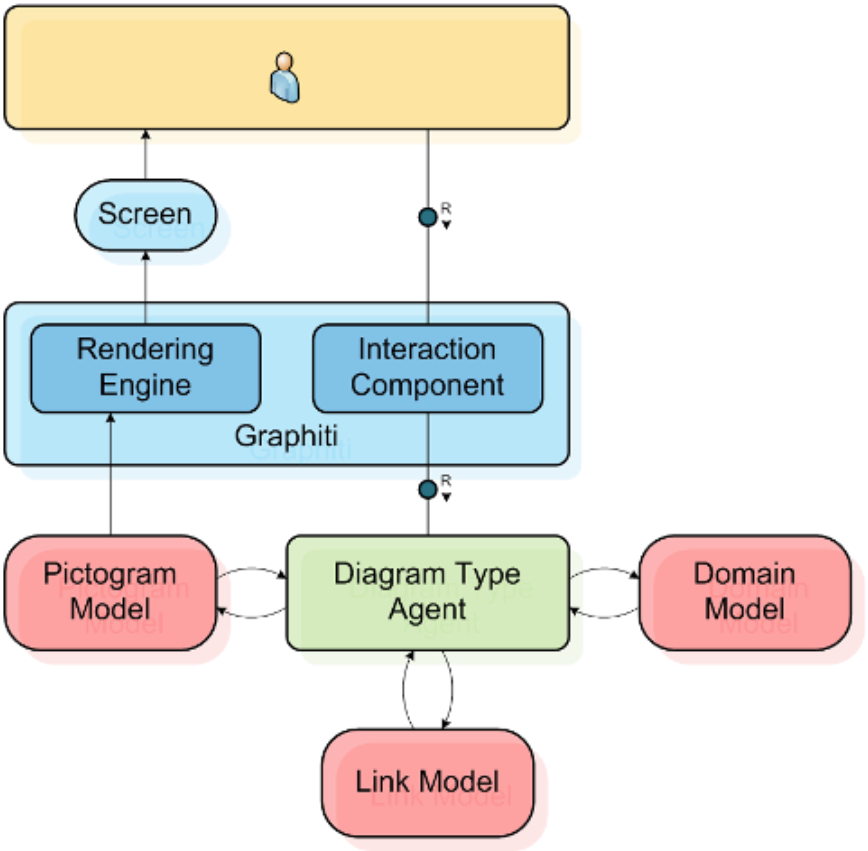


Fig. 12. – Arquitetura interna do Graphiti [11]

Como pode ser visto na figura anterior, o *Graphiti* em si apresenta uma visualização interativa do diagrama que está a ser feito ao utilizador através da sua interface gráfica. O utilizador pode interagir de algumas formas com esta interface (colocar objetos novos, mover objetos, criar conexões entre objetos). O *Graphiti* trata de processar os comandos e pedidos realizados pelo utilizador e por sua vez interage com um *Diagram Type Agent*, que contém todas as regras relativas a como o diagrama deve funcionar. O Diagram Type Agent trata de gerir os modelos do diagrama a ser editado, sendo um destes modelos, o *Pictogram Model*, utilizado pelo Graphiti para apresentar a interface gráfica.

Este *Diagram Type Agent* é o que deve ser implementado caso se queira criar um novo editor gráfico. Cada Diagram Type define os tipos de componentes gráficos que podem ser utilizados, o tipo de objetos de modelo que são criados, as regras sobre como funcionam estes componentes e como é

permitido ao utilizador interagir com eles. O Diagram Type Agent interage com três componentes. O *Domain Model*, o *Link Model* e o *Pictogram Model*.

O *Domain Model* é o modelo que guarda a informação relativa aos diagramas que estão a ser desenvolvidos. Neste modelo são guardados estruturas e objetos relativos aos vários componentes gráficos visíveis pelo utilizador. Qualquer componente gráfico que seja visível ao utilizador deve ter um objeto no modelo correspondente, sendo possível a cada objeto no modelo ter mais que um componente gráfico a o representar.

O *Pictogram Model* é o modelo da parte gráfica de um diagrama *Graphiti*, é aqui que são guardadas as várias informações relativas a como representar as várias visualizações dos objetos do modelo. O *Pictogram Model* não é a representação gráfica em si, mas sim um modelo que guarda todas as informações utilizadas pelo *Graphiti* para construir a representação gráfica. A forma pela qual o *Graphiti* permite ao utilizador criar componentes gráficos é através da utilização de objetos básicos de pictograma (retângulos, linhas, elipses, caixas de texto, etc…). O Pictogram Model mantém uma lista ordenada de todos estes componentes à medida que são colocados no diagrama.

O *Link Model* é o componente de um diagrama *Graphiti* encarregue de gerir informações relacionadas com as várias correspondências entre componentes gráficos e os seus objetos no *Domain Model*. Através do Link Model torna-se possível identificar facilmente qual o objeto de modelo através da sua representação gráfica, e vice-versa.

2.3 Outros IDE's para a Norma IEC 61131-3

Existem atualmente no mercado algumas opções para IDE's. A maioria dos IDE's mais populares são comerciais, estes IDE's comerciais não só têm a desvantagem de muitas vezes terem custos bem elevados, têm ainda a desvantagem de não aplicarem totalmente a norma IEC 61131-3, isto quer dizer que muitas vezes programas que funcionam num IDE não funcionam num outro sem serem parcialmente reescritos [4]. Geralmente o tipo de IDE a utilizar depende do fornecedor do PLC a ser programado, alguns PLC's têm IDE's com os quais se dão melhor, mas isso não impede a escolha de uma outra alternativa. Nesta secção serão descritos alguns dos IDE's mais populares, primeiro serão descritos alguns IDE's comerciais, finalmente será descrito o IDE Beremiz, uma alternativa gratuita popular.

2.3.1 ISaGRAF

O **ISaGRAF** é um IDE para a norma IEC 61131. Este IDE é comercial, desenvolvido pela *Rockwell Automation*, Inc. O ISaGRAF permite a utilização de todas as linguagens definidas pela norma, a exportação de código fonte em formato "C" e permite a criação de controladores I/O personalizados. [12]

2.3.2 Unity Pro

Desenvolvido pela *Schneider Electric*, o **Unity Pro** é um outro IDE comercial para a norma IEC 61131. Este IDE suporta todas as funcionalidades standard para um IDE IEC 61131, tais como a criação de *Function Blocks* personalizados, a simulação de um PLC para testes, *Animation tables* para visualização de variáveis. [13]

2.3.3 CoDeSys

O **CoDeSys** é um outro IDE comercial utilizado para o desenvolvimento de programas na norma, tal como os outros suporta todas as linguagens da norma. Para além das funcionalidades base,

o CoDeSys tem uma loja onde se podem encontrar plug-ins para adicionar funcionalidades extra, tais como gestão de versões, UML e automação de testes.[14]

2.3.4 Beremiz

O **Beremiz** é um IDE *OpenSource* desenvolvido pela LOLITECH em conjunto com a universidade do Porto. Este IDE é uma opção OpenSource muito utilizada e é muito robusta. O Beremiz permite programar em todas as linguagens definidas pela norma IEC 61131-3, este IDE difere de muitos IDE's comerciais por seguir totalmente a norma IEC 61131-3, esta característica resulta na fácil portabilidade de programas construídos neste IDE para outros IDE's. [4]

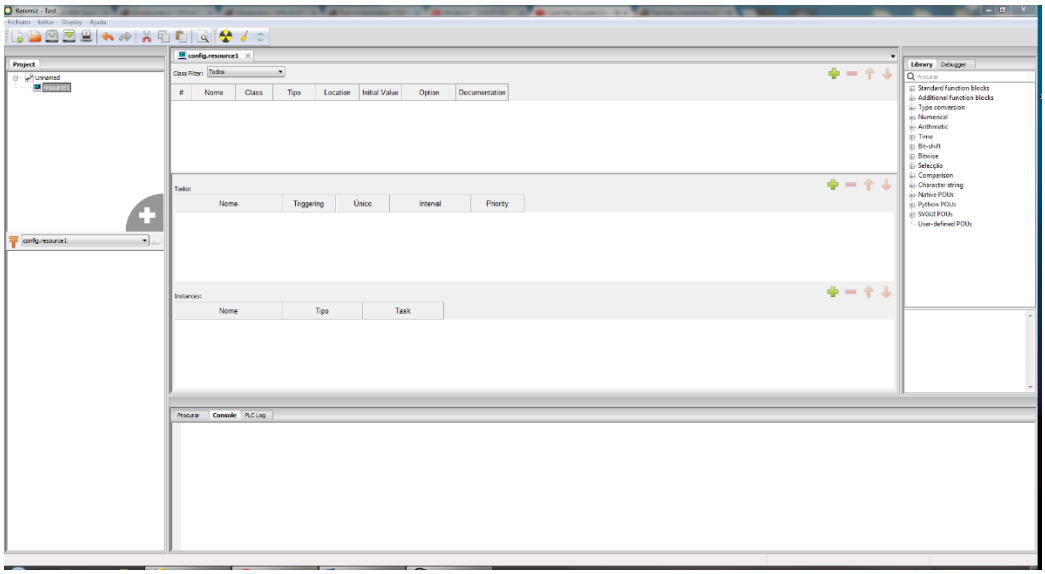


Fig. 13. – Janela Principal do Beremiz

2.4 Trabalho já realizado

Sendo a quarta numa sequência de dissertações sobre o mesmo tópico, existe já uma grande quantidade de trabalho previamente realizado em relação ao desenvolvimento de um IDE em Eclipse para a norma IEC 61131.

Em 2014, a primeira dissertação realizada com o objetivo de criar o IDE foi feita por Filipe Miguel de Jesus Ramos. Nesta primeira dissertação foram desenvolvidos editores para ambas as linguagens textuais (IL e ST), foi também desenvolvida uma parte de um IDE para SFC.

Em 2015, a segunda dissertação relativa a este tema foi realizada por José António Fernandes Ferreira. O trabalho realizado nesta segunda iteração foi o estudo do trabalho anterior e a finalização do módulo de SFC que se encontrava incompleto no final da dissertação anterior. Para além deste editor foram adicionados um módulo que permitia a importação de XML e a possibilidade de criar novos projetos da norma e um navegador que permitia organizar os POU's e editores.

Em 2016 a terceira dissertação a contribuir para este projeto foi realizada por Filipe Rafael dos Santos Ribeiro. Foram desenvolvidos dois novos plug-ins, estes plug-ins permitem a criação de tipos de dados e de variáveis. Foi também melhorado o módulo de navegação e reestruturado o código de todo o projeto para um formato mais eficiente (Fig. 14.).

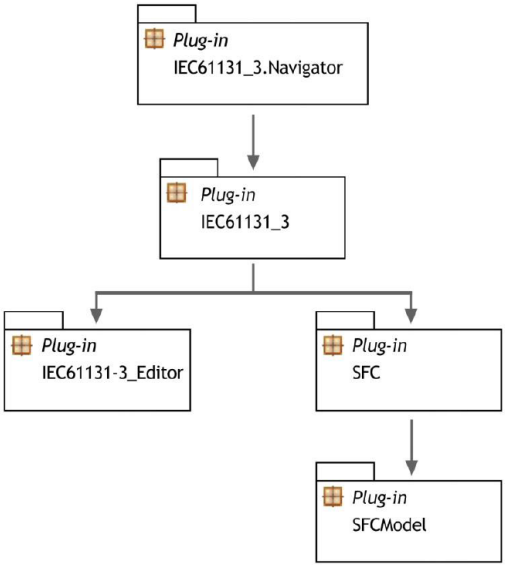


Fig. 14. – Package Diagram UML da estrutura do IDE como ele se encontra agora [3]

Resumindo, no início desta dissertação começou-se com um IDE para a norma IEC 61131-3 com um editor IL, um editor ST e um editor SFC funcionais. Para além dos editores já implementados existia um navegador, um módulo de configuração de variáveis para POUs, e um módulo que permitia a criação de tipos de dados.

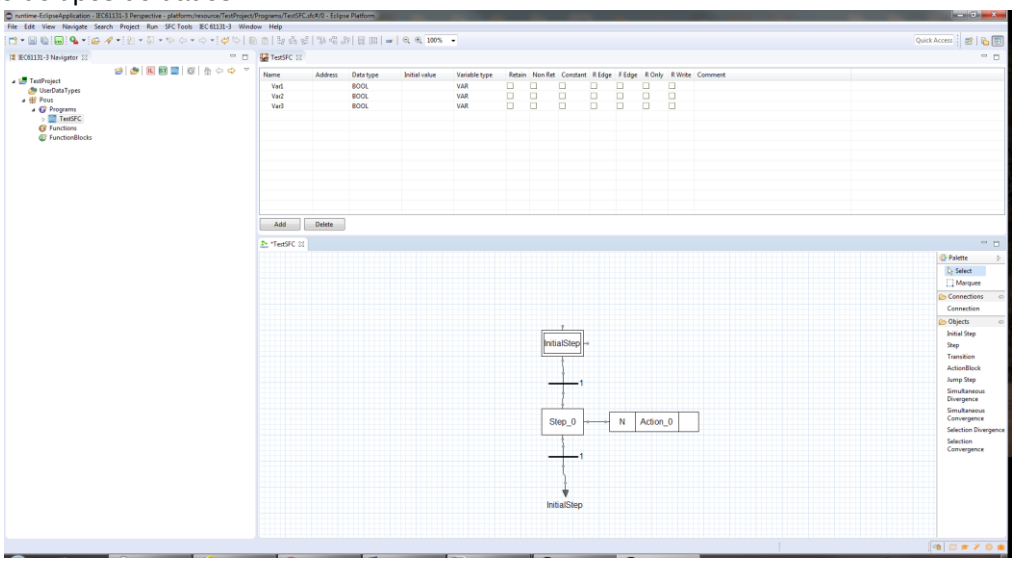


Fig. 15. – Janela do IDE Eclipse, com navegador, editor gráfico de SFC e lista de variáveis

Capítulo 3

3. Desenvolvimento

Neste capítulo será apresentado o software desenvolvido. Inicialmente será descrito o processo de inserção dos novos editores no navegador já previamente criado por outros alunos, depois será tratado o desenvolvimento do editor de *Function Block Diagrams*, seguido do editor de *Ladder Diagrams*.

3.1 Integração com navegador

Para permitir a criação dos programas através da *perspective* IEC 61131-3 previamente desenvolvida, foi necessária a alteração do navegador IEC 61131-3. Os dois novos editores utilizam o extension point "**org.eclipse.graphiti.ui.diagramTypes**". Sendo interpretados pelo *Graphiti* como novos tipos de diagramas. Normalmente estes novos tipos de diagramas são criados através de um *wizard* pertencente ao *Graphiti*. Este *wizard* permite a escolha entre os vários ***diagramType*** disponíveis, criando um novo ficheiro do tipo "*.diagram"* que utiliza o ***diagramType*** escolhido pelo utilizador*.*

De forma a que a criação de diagramas que utilizem os novos **diagramTypes** fosse possível, foi necessária a alteração da classe **IEC61131ProjectView** que trata da criação dos vários POUs implementados. Foram adicionadas duas novas ações a esta classe, correspondentes à criação de um novo POU em LD e em FBD. Para além destas novas ações foram também criados ícones novos para representar estes novos tipos de diagrama. Os novos editores tiveram *filetypes* novos associados através do uso do extension point "**org.eclipse.core.contenttype.contentTypes**", sendo estes o "**.fbd**" e o "**.ld**"**.**

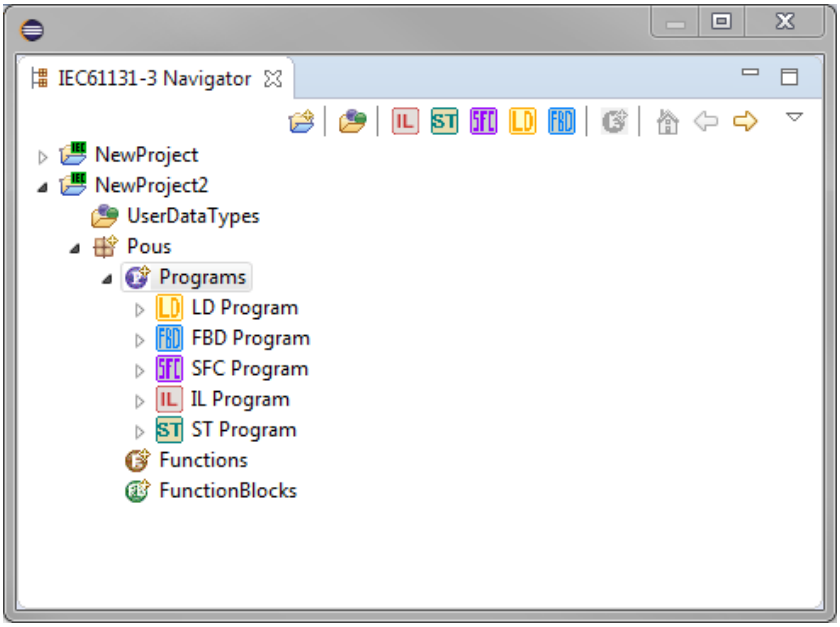


Fig. 16. – Navegador IEC 61131-3, com os novos botões para criação de diagramas LD e FBD.

3.2 Editor FBD

Nesta secção será descrito o editor FBD implementado. Começar-se-á por uma descrição do processo de criação de um metamodelo ecore, que foi usado como base para a criação do editor em si. Após a referência ao metamodelo, será explorada a estrutura interna do editor, sendo referidos os tipos de classes e métodos que foram utilizados. Para finalizar, serão exploradas uma a uma as funcionalidades principais implementadas, estas funcionalidades são em geral ligadas à criação dos componentes base de programas FBD, e as conexões entre eles.

3.2.1 Metamodelo ecore

Como foi referido no capítulo anterior, um diagrama *Graphiti* está separado em três partes, a representação gráfica, o modelo interno e as ligações (*links*) entre ambos. Para poder criar um editor utilizando o Graphiti é comum utilizar um metamodelo ecore que define o tipo de objetos que vão ser tratados no editor.

Para o editor FBD, foi criado um metamodelo ecore (Fig. 17.) com a ajuda da ***Eclipse Modelling Framework***. Para criar o metamodelo foi necessário definir várias interfaces representativas das classes existentes no modelo com métodos "get….()" representativos dos vários atributos das classes do metamodelo. Depois de definidas as interfaces foram utilizadas ferramentas do EMF para gerar o código que implementou o metamodelo. Houve várias atualizações ao metamodelo ecore ao longo do desenvolvimento do editor, no entanto, o metamodelo atual pode ser visto no **Anexo A**.

Para os *Function Block Diagrams*, existem dois tipos principais de componentes que devem ser implementados por um editor. Estes dois tipos são as várias etiquetas que representam variáveis de entrada, saída e entrada/saída, e as Funções e Function Blocks em si. Esta última classe de objetos foi implementada no metamodelo sob o nome "BlockObject", com o Function Block sendo uma extensão dos BlockObjects contendo o nome da instância, algo que não se encontra nas Funções.

Para além destes componentes, criou-se o componente **FBDConnection**, representando as conexões entre objetos. Implementou-se também o objeto **FBDProjectData**, este último não foi utilizado mas será útil na implementação futura de importação/exportação XML.

O EMF gerou automaticamente uma outra classe, a classe **FunctionBlockDiagramFactory**, esta classe permite a instanciação de cada um dos objetos do metamodelo, sendo essencial para que o modelo ecore possa ser utilizado. Concluído o metamodelo ecore, a implementação do editor em Graphiti era possível.

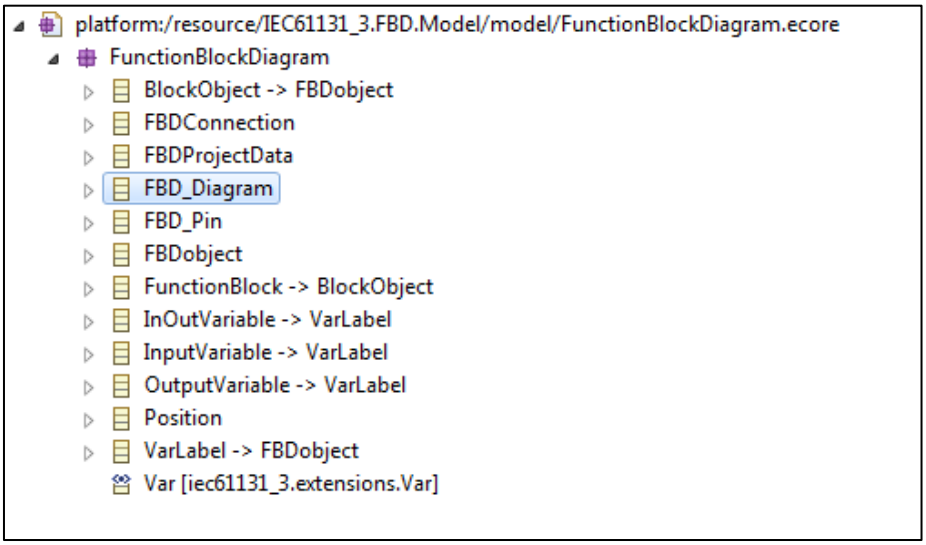


Fig. 17. – Metamodelo ecore implementado.

3.2.2 Estrutura do editor

Como já foi referido anteriormente, o editor FBD funciona utilizando o framework ***Graphiti***. Esta ferramenta define uma estrutura base que todos os editores gráficos desenvolvidos neste projeto utilizam. Cada editor funciona como um "*diagramType*", um tipo de diagrama de *Graphiti*. Vários componentes de um editor gráfico, tais como a inserção de novos objetos, a remoção de objetos, a manipulação dos objetos (movimentação e mudança de tamanho), e outras funcionalidades estão já implementadas por classes que podem ser modificadas e expandidas.

Parte do trabalho desenvolvido foi a modificação das várias funcionalidades já implementadas por omissão. A maior parte do trabalho foi, no entanto, a criação das funcionalidades que permitiram a introdução, modificação e controlo dos vários objetos (classes e representações gráficas).

Para a definição do novo editor como um "diagramType" foram implementadas as extensões "**org.eclipse.graphiti.ui.diagramTypes"** e **"org.eclipse.graphiti.ui.diagramTypeProviders"**, o primeiro *extension point* define o projeto como um plug-in do tipo diagramType e o segundo define uma classe como o diagramTypeProvider. Como já foi referido na secção anterior, um outro *extension point* foi utilizado para definir o tipo de ficheiro ".fbd" para diagramas criados neste editor, "**org.eclipse.core.contenttype.contentTypes**".

O ***diagramTypeProvider***, no caso do editor FBD a classe "**FBDDiagramTypeProvider**", é uma classe essencial que define o tipo de diagrama que se trata. Esta classe é uma extensão da classe "*AbstractDiagramTypeProvider*" e define quais são as classes "*FeatureProvider*" e "*ToolBehaviorProvider*", tendo estas sido implementadas com os nomes "***FBDFeatureProvider***" e "***ToolBehaviorProvider***". Estas duas classes são onde se define as várias funcionalidades (*Features*) do editor e algumas definições de como o *Graphiti* se deve comportar.

Na classe ***ToolBehaviorProvider*** foi alterado o método *getDoubleClickFeature*(), este método, tal como diz o nome, retorna o método que deve ser executado sempre que se clica duas vezes num objeto, este método foi implementado com o nome ***DoubleClickUpdateFeature*** e será descrito mais tarde.

De forma a que não fossem apresentados avisos sempre que se fosse eliminar um objeto, a classe ***deleteWithoutPrompt*** foi implementada. Esta classe limita-se a realizar o método base de eliminação de objetos, saltando o passo de apresentar um pedido de confirmação ao utilizador. Esta

classe foi também implementada de forma a limpar o modelo de objetos que não devam lá estar, de forma a eliminar, por exemplo, os objetos “FBD_Pin” sem *parent*.

Ainda no ***FBDToolBehaviorProvider***, foi necessário alterar o método base *isDefaultBendPointRenderingActive*() para retornar “false”. Esta modificação permite a utilização de conexões entre objetos com ângulos de 90º, algo que até agora não tinha sido feito no editor SFC. Nas figuras seguintes (Fig. 18. e Fig. 19.) pode-se observar o aspeto dos diagramas antes e depois desta modificação.

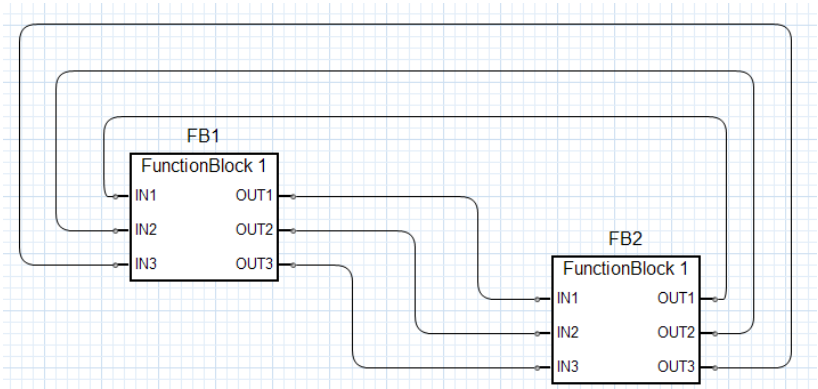


Fig. 18. – Exemplo de conexões com a isDefaultBendpointRenderingActive() a true.

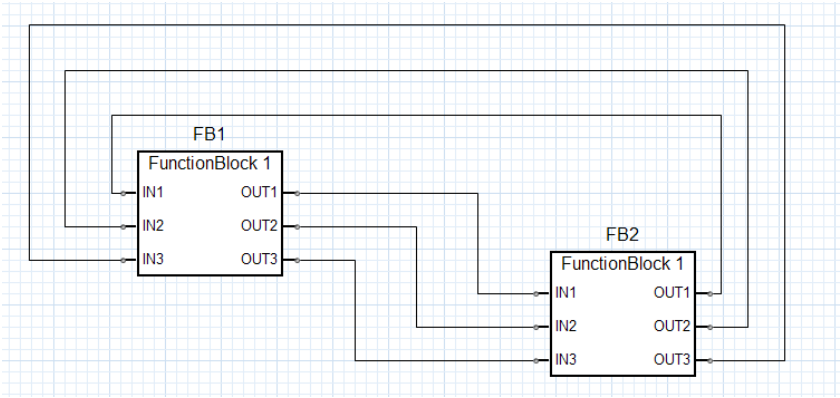


Fig. 19. – Exemplo de conexões com a isDefaultBendpointRenderingActive() a false.

Para além de ter sido feita esta alteração nos editores FBD e LD. Foi necessário modificar o *ToolBehaviorProvider* do editor SFC, que até agora tinha o aspeto incorreto.

As várias classes que tratam da criação das representações gráficas dos objetos utilizam constantes em comum. Estas constantes são usadas para definir, entre outros aspetos, coisas como a espessura das linhas a utilizar, as cores a utilizar, os tipos de letra e os tamanhos relativos de certos componentes. Para facilitar a modificação destes atributos criou-se a classe ***FBDobjectData***, que se limita a guardar uma lista de variáveis a ser usadas pelas outras classes do editor.

A classe **FBDFeatureProvider** é uma classe essencial para o funcionamento do editor, é nesta classe que foram listadas todas as funcionalidades (*Features*) implementadas, sendo necessário apenas listar corretamente as funcionalidades que se quer disponibilizar ao utilizador para elas aparecerem na interface de utilizador gerada automaticamente pelo *Graphiti.*

Para facilitar a interpretação da estrutura do editor implementado, o **Anexo C** contém um diagrama de classes do FeatureProvider com todas as funcionalidades relevantes.

3.2.3 Funcionalidades Implementadas

No *Graphiti*, cada componente gráfico deve ter, no mínimo, duas *Features* associadas, sendo estas uma "*createFeature*" e uma "*addFeature*". A "*createFeature*" trata de instanciar o objeto do modelo relativo ao componente gráfico e a "addFeature" trata de criar a representação gráfica do objeto.

Como já foi referido no ponto 3.2.1, o *Graphiti* funciona através de "*links*", ou seja, "ligações" entre os componentes gráficos e os objetos no modelo. Esta separação do modelo e da sua representação gráfica permite, por exemplo, que um só objeto tenha mais que uma representação gráfica, ou que representações gráficas sejam eliminadas sem que os objetos sejam eliminados.

As várias "*createFeatures*" e "*addFeatures*" criadas, que serão descritas de seguida, foram declaradas no ***FBDFeatureProvider***, aparecendo automaticamente como funcionalidades na interface gráfica do editor. Na figura seguinte (Fig. 20.) é possível observar a interface automaticamente gerada do editor, com os vários componentes gráficos implementados na lista à direita, prontos a ser usados.

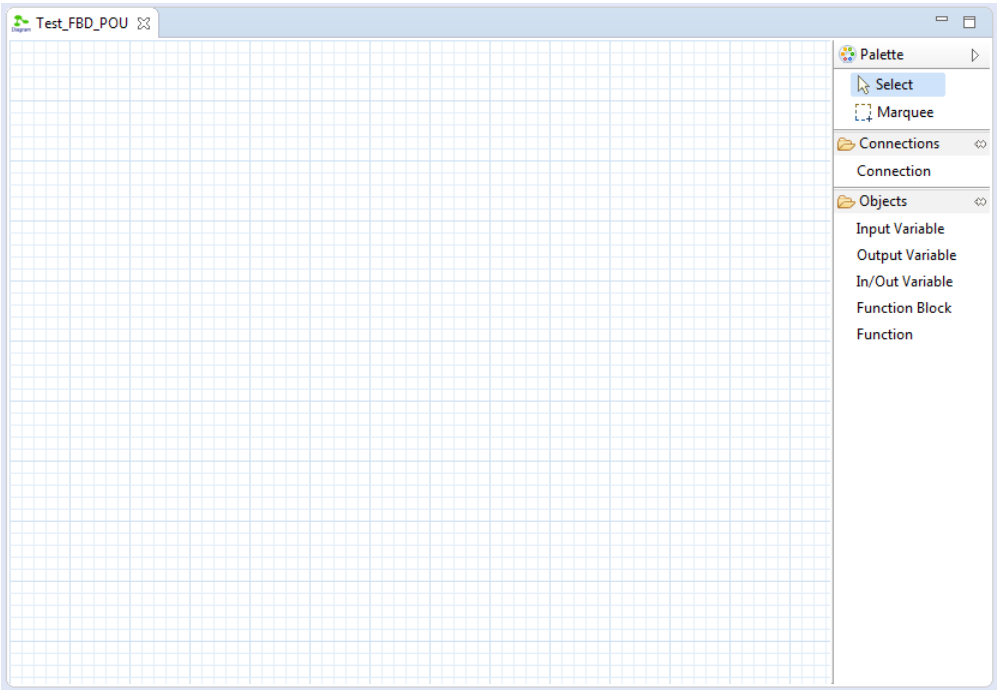


Fig. 20. – Interface do editor FBD, criada automaticamente pelo Graphiti

3.2.3.1 Function Blocks / Functions

Foram implementadas as *createFeatures* e *addFeature*s para os Function Blocks e Functions, com os nomes:

- *AddFunctionBlockFeature*;
- *AddFunctionFeature*;
- *CreateFunctionBlockFeature;*
- *CreateFunctionFeature*;

Ambas as *createFeatures* realizam tarefas semelhantes, confirmam que o objeto onde se quer colocar o novo objeto é de facto um diagrama através do método "*canCreate*()" e criam uma nova instância do objeto que se quer colocar através do método respetivo da ***FunctionBlockDiagramFactory***, este funcionamento é igual em todas as outras *createFeatures*, à exceção da criação de conexões. Depois de realizar a criação do objeto, a *createFeature* requisita uma representação gráfica, sendo executada a *addFeature* correspondente.

As *addFeatures* implementam, no mínimo, dois métodos essenciais. Estes métodos são o "*canAdd*()" e o "*Add*()", em ambos os métodos o argumento utilizado é o ***context*** , este argumento é essencial e muito comum em classes utilizadas pelo *Graphiti*. Este *contexto* pode ser de vários tipos, mas normalmente contém toda a informação relevante à ação que se está a realizar, nomeadamente informação como o tipo de ação pretendida, a posição X e Y de um novo objeto, o diagrama onde este objeto se encontra ou deverá ser colocado, e outras informações.

3.2.3.1.1 Diálogos de configuração

Para a implementação das *addFeatures* dos *Function Blocks* e *Functions*, o método "*canAdd*()", que normalmente serviria só para identificar se era possível adicionar um novo objeto, foi utilizado para realizar a configuração do novo objeto. Para tal, foram implementados duas caixas de diálogo, uma para *Function Blocks* e uma para *Functions*. Estas caixas de diálogo são apresentadas quando o utilizador tenta colocar um destes objetos no diagrama.

Os diálogos foram implementados com as classes ***PromptFunctionBlock*** e ***PromptFunction***, na *package* iec61131_3.fbd.dialogs. Nas imagens seguintes (Fig. 21. e Fig. 22.) estão visíveis as duas janelas de diálogo, ambas as janelas são semelhantes, sendo a única diferença a introdução de um campo de texto para a escrita de um nome no caso da criação de um *Function Block*.

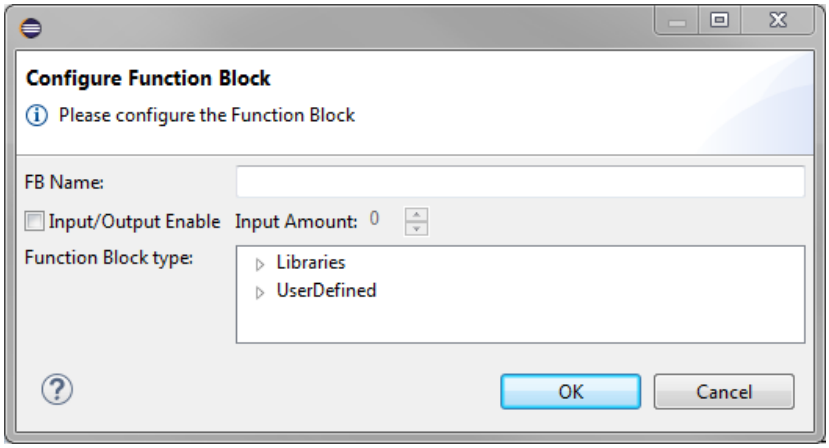


Fig. 21. – Diálogo de configuração de Function Blocks.

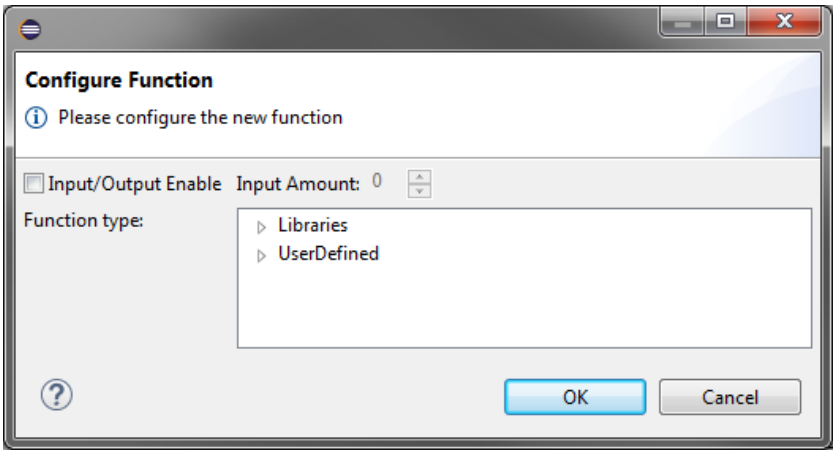


Fig. 22. – Diálogo de configuração de Functions.

De seguida serão descritas as várias funcionalidades destas janelas de diálogo:

- **Input/Output Enable**: a Norma prevê a criação de blocos com entradas e saídas do tipo BOOL especiais que devem ser suportadas, este "*toggle*" permite a sua ativação para o novo objeto a ser configurado;
- **Input Amount**: a norma define a existência de alguns objetos com um número variável de inputs, como por exemplo uma função que realize a soma de todos os seus inputs. Caso um utilizador tenha selecionado uma função que suporte variáveis extensíveis, este campo é ativado e permite ao utilizador a escolha do número de inputs pretendido. Esta funcionalidade será explorada melhor na secção seguinte;
- **Function/Function Block Type**: esta caixa é a componente mais importante da caixa de diálogo, permite ao utilizador selecionar entre os vários *Function Blocks* e *Functions* disponíveis, sejam estas geradas pelo próprio utilizador ou pré-feitas (como no caso das funções *standard*). A forma pela qual este navegador funciona é simples, quando é iniciado o diálogo a *addFeature* envia como um argumento a localização do diagrama atual, através desta informação todas as *Functions* e *Function Blocks* definidos no projeto IEC 61131-3 atual são recolhidas e apresentadas. Para além das funções definidas pelo utilizador, a caixa de diálogo procura através de um algoritmo recursivo dentro de uma pasta com um nome específico, sendo este nome "FuncLibraries" e "FBLibraries". Devido ao algoritmo de busca recursivo é possível adicionar bibliotecas ao ficheiro "FuncLibraries" ou "FBLibraries" manualmente, sendo estas automaticamente representadas na caixa de diálogo. Atualmente é necessário que estas pastas sejam colocadas no mesmo ficheiro onde os vários projetos IEC 61131-3 são criados, mas será possível alterar isto no futuro para que as funções pré-definidas sejam instaladas no mesmo local do próprio Eclipse, e sejam lidas através daí;

Para a definição de *Functions* e *Function Blocks* e a leitura das várias variáveis de entrada/saída, foi utilizada a ferramenta já previamente implementada numa tese anterior. Esta ferramenta é o editor de ficheiros ".tvar", ficheiros estes que são criados sempre que um novo POU é construído, nesta ferramenta os ficheiros ".tvar" podem ser editados como tabelas de variáveis, (Fig. 23.) sendo possível adicionar, remover e modificar as várias variáveis.

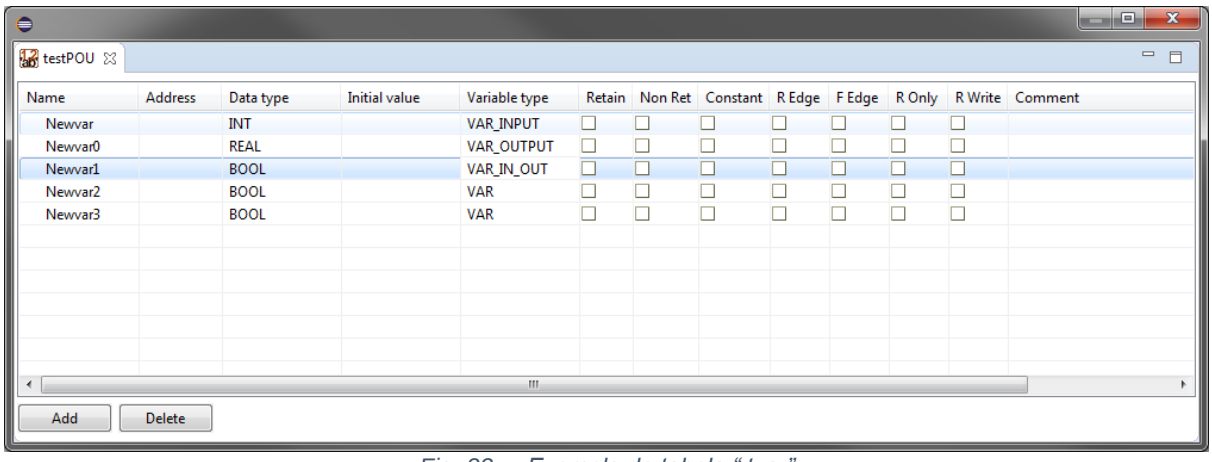


Fig. 23. – Exemplo de tabela ".tvar"

São estes ficheiros ".tvar" que funcionam como "cabeçalhos" que o editor FBD utiliza para definir o aspeto dos vários objetos que são criados. Para a leitura destes objetos foi criada uma versão modificada da classe previamente implementada **ieec61131_3.VarTableEditor.VarList** chamada "***VarLoader***", esta classe limita-se a abrir e ler um ficheiro ".tvar" que lhe é indicado.

3.2.3.1.2 Variáveis extensíveis

Para a definição de POUs com variáveis extensíveis, optou-se pela deteção de uma sequência específica de argumentos a ser colocada no campo "*Comment*" de uma variável no seu respetivo ficheiro ".tvar". Um exemplo de uma definição de uma Function que implementa um método de ADD de duas ou mais variáveis de entrada pode ser vista na figura seguinte (Fig. 24).

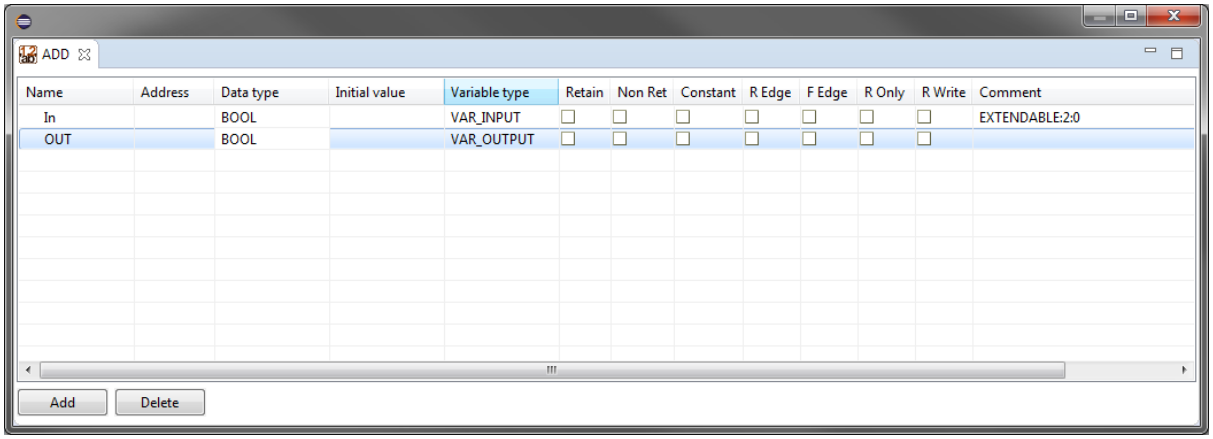


Fig. 24. – Definição de variáveis para função ADD.

Como pode ser observado na figura, a função tem dois tipos de variáveis, a primeira variável é do tipo VAR_INPUT, ou seja, é um input e a segunda entrada é do tipo VAR_OUTPUT, sendo o único output da função. Para permitir que seja realizado o ADD de duas ou mais entradas foi colocado no campo "*Comment*" da primeira variável o argumento "EXTENDABLE:2:0", este argumento tem três componentes. O primeiro componente serve para sinalizar ao código que se trata de uma variável extensível, o segundo componente é um número que representa o número mínimo de inputs, neste caso dois, o terceiro componente é um número que define a partir de que número começam os nomes dos inputs, visto que em alguns casos estes inputs começam com o nome "IN0" e noutros começam com o nome "IN1". Na imagem seguinte (Fig. 25.) podem ser observadas duas *Functions* criadas utilizando este ficheiro ".tvar" como guia, uma com dois inputs e outra com seis. Note-se que é impossível ao código suportar mais que uma variável extensível num só ficheiro ".tvar", sendo sempre escolhida a primeira variável encontrada com o comentário correto. Este funcionamento é intencional visto que a Norma não define objetos com mais que uma variável extensível.

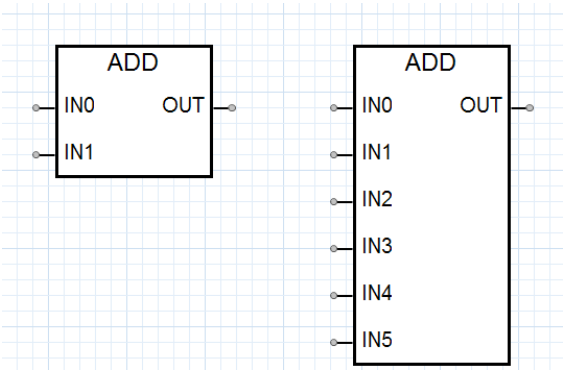


Fig. 25. – Exemplos de duas instâncias diferentes de uma Function com variável extensível.

Apesar deste tipo de funcionamento ser só esperado para um número reduzido de funções da Norma, optou-se por permitir a qualquer utilizador aproveitar desta funcionalidade, para tal sendo só necessário utilizar o método previamente referido ao definir uma variável que se queira tornar extensível.

3.2.3.1.3 Representações gráficas

Uma vez concluída com sucesso a configuração do novo objeto, seja ele *Function* ou *Function Block*, o método *"canAdd*()" preenche os vários campos relevantes no objeto do modelo com as informações obtidas do diálogo e do ficheiro ".tvar" relevante e retorna ***true***. Caso o utilizador decida cancelar a configuração utilizando o botão "Cancel" na caixa de diálogo, o método "*canAdd*()" retorna ***false*** e nenhum objeto é criado.

Caso o método "*canAdd*()" retorne ***true***, o código passa ao método "*Add*()" Neste método é que a representação gráfica dos objetos é construída, todos os objetos são criados recorrendo a métodos já existentes do *Graphiti* que permitem a criação de vários elementos gráficos do tipo "*pictogramElement*", estes objetos são componentes simples, tais como retângulos, linhas retas, ou elipses. Através da reorganização e manipulação destes componentes base é que se construiu os objetos mais complexos dos vários editores.

Como as representações das *Functions* e *Function Blocks* são tão semelhantes (Fig. 26.), serão descritas simultaneamente, sendo a única diferença entre as duas a introdução de um nome no topo dos *Function Blocks*.

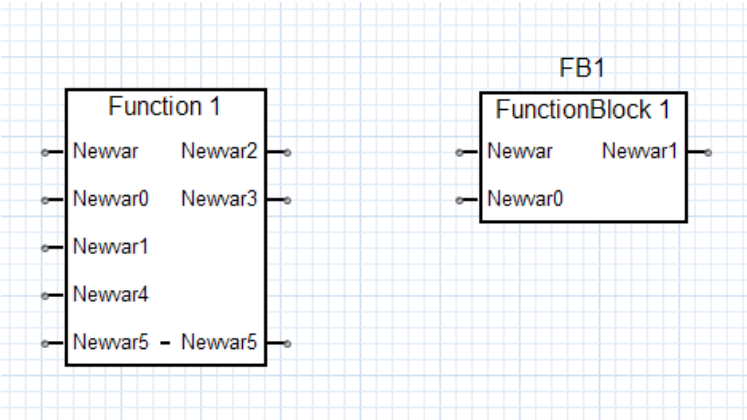


Fig. 26. – Exemplos de representações gráficas implementadas para Function e Function Block.

As representações gráficas destes objetos têm o seguinte funcionamento:

- A partir do objeto do modelo, as várias variáveis ***Input***, ***Output*** e ***Input/Output*** são recolhidas, para todas estas variáveis são criadas entradas/saídas na representação gráfica;
- No caso de variáveis ***Input-Output***, a representação é feita segundo a norma, com um pino input e um pino output ao mesmo nível, com uma linha a unir os dois;
- Caso a variável em questão é do tipo ***Rising Edge*** ou ***Falling Edge***, é adicionado um ">" ou um "<", respetivamente. Este tipo de representação é definida pela Norma;
- Caso o "*Input/Output Enable*" tenha sido ativado previamente, são criados em primeiro lugar os pinos correspondentes;

Tanto a altura como a largura das representações gráficas dos objetos são adaptadas automaticamente aos vários componentes internos (nomes de variáveis, nome da função) do Function Block/ Function, assegurando-se sempre que o tamanho das representações gráficas é correto.

Um ponto importante na forma de como funciona o modelo do editor FBD é a utilização dos objetos "FBD_Pin". Estes objetos são simples, contendo informação relativa a um nome de variável, o tipo de variável associada e um método chamado "getParent()". Através deste método é possível obter o objeto que contém este pino. A principal razão pela qual esta estrutura de objetos do modelo foi utilizada está no funcionamento através de "links" do *Graphiti*. Os pontos aos quais se pode ligar uma conexão chamam-se de *Anchors*, estas *Anchors* devem estar ligadas a um objeto através do método "*link*()", ao ligar estas *Anchors* a pinos específicos, em vez de ligar ao *Function Block* / *Function* a que

pertencem, é possível facilmente descobrir o FBD_Pin correspondente a cada *Anchor*, e vice-versa. Por sua vez os vários pinos de uma *Function* ou *Function Block* estão contidos em listas obtidas através dos métodos "*getInputPins*()" e "*getOutputPins*()", sendo trivial desta forma saber se uma Anchor corresponde a um Input ou Output. Para além disso, devido à informação acerca do tipo de variável contida dentro do próprio pino, torna-se possível negar ao utilizador a criação de conexões entre tipos de dados não correspondentes, esta funcionalidade será explorada melhor numa secção seguinte.

O facto de que todos os objetos contêm os seus pins de uma forma ordenada permite ao código percorrer os diagramas seguindo as várias ligações, os vários objetos, conseguindo chegar a qualquer outro objeto para o qual haja uma conexão. Esta funcionalidade será muito útil numa futura implementação de exportação XML de programas criados neste editor.

3.2.3.2 Variáveis Input/Output/Input-Output

O segundo tipo de componente principal do editor FBD é a etiqueta que representa uma variável. Este tipo de componente é normalmente utilizado para definir onde são introduzidas as variáveis de entrada e saída do POU. A sua estrutura é simples, sendo apenas umas etiquetas com o nome da variável em si e um ou dois pinos, conforme o tipo de variável.

As três variedades de variáveis externas que foram implementadas são as variáveis de ***Input***, ***Output*** e ***Input-Output*** (Fig. 27.), tendo as primeiras um só pino para efetuar ligações e a última tendo dois pinos, um do seu lado esquerdo e outro do seu lado direito.

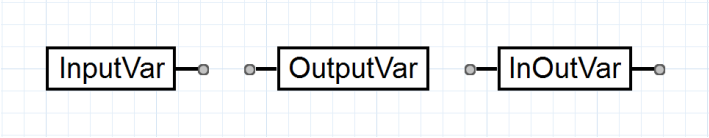


Fig. 27. – Representações gráficas das etiquetas de variáveis.

As classes desenvolvidas relativas a estas variáveis são as seguintes:

- *CreateInOutVarLabelFeature*;
- *CreateInputVarLabelFeature*;
- *CreateOutputVarLabelFeature*;
- *AddInOutVarLabelFeature*;
- *AddInputVarLabelFeature*;
- *AddOutputVarLabelFeature*;

Tal como no caso dos *Function Blocks* e *Functions*, foi implementada uma caixa de diálogo (Fig. 28.) com o nome ***PromptVariable*** que é apresentada quando o método "*canAdd*()" é executado, esta caixa de diálogo é mais simples, pedindo ao utilizador um nome para a variável a ser associada à etiqueta, este nome é por sua vez apresentado na representação gráfica.

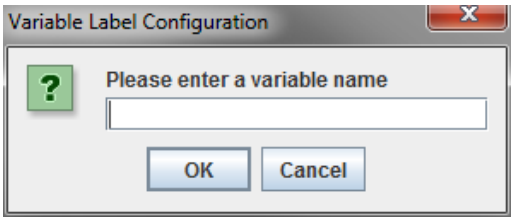


Fig. 28. – Diálogo de configuração de etiqueta de variável.

De forma semelhante ao que foi feito nos Function Blocks e Functions, o tamanho da representação gráfica varia conforme o tamanho do nome da variável escolhido (Fig. 29.).

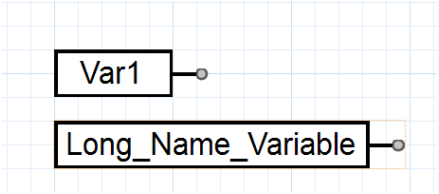


Fig. 29. – Exemplo de tamanho variável para etiquetas.

3.2.3.3 Conexões

Para permitir a ligação entre os objetos do editor, foram implementadas conexões como extensões das conexões base já oferecidas pelo *Graphiti.* Para descrever as conexões implementadas será inicialmente descrito o funcionamento de conexões.

As conexões no *Graphiti* funcionam de forma semelhante aos outros componente gráfico, têm também classes ***createFeature*** e ***addFeature*** e estas têm as mesmas tarefas de criar o objeto de modelo e criar o componente gráfico, respetivamente. A grande diferença entre as conexões e os componentes gráficos está nos métodos implementados pela ***createFeature*** e a forma pela qual a componente gráfica é construída.

A classe ***createFeature*** responsável pela criação de uma conexão implementa os mesmos métodos "*canCreate()*" (se uma ligação pode ser criada, é executado quando se seleciona o ponto final da ligação) e "*create*()" (cria a ligação), mas para além destes tem um outro método chamado "*canStartConnection()*" que define se é possível ou não iniciar uma conexão quando se seleciona um ponto inicial para a mesma, geralmente uma Anchor.

A representação gráfica das conexões é muito diferente dos outros componentes gráficos. Todas as conexões utilizam o componente gráfico chamado ***FreeFormConnection***. Estas conexões contêm informação relativa ao ponto inicial da ligação, ao ponto final, e a quaisquer pontos intermédios onde haja uma mudança de trajetória, estes pontos são chamados de ***Bendpoints***. O *Graphiti* desenha uma linha que começa no ponto inicial, passa pelos *bendpoints* por ordem e termina no ponto final.

De forma a permitir ao utilizador que este iniciasse uma conexão simplesmente clicando e arrastando de um pino ao outro, o campo ***getDragAndDropFeature*** no *FBDFeatureProvider* foi alterado para retornar as ***getCreateConnectionFeatures()***.

O Graphiti suporta reconnectionFeatures, esta funcionalidade permite ao utilizador "agarrar" nas pontas de uma conexão e refazê-la noutro lugar. Como esta funcionalidade interferia com a funcionalidade drag-and-drop e levava o utilizador a refazer outras ligações quando pretendia criar novas ligações, foi necessário implementar uma ReconnectionFeature cujo método canCreateConnection() retorna ***false***. Esta funcionalidade foi criada para o editor FBD e LD. Foi necessário também criar esta funcionalidade para o editor SFC previamente implementado, porque esta não se encontrava corretamente implementada.

3.2.3.3.1 CreateFBDConnectionFeature

As ligações em FBD são um caso especial das ligações base do *Graphiti*, porque só podem suportar linhas verticais e horizontais, foi necessário então uma implementação especializada destas ligações base já fornecidas.

Começando por descrever a classe ***createFBDConnectionFeature***, o método "*canStartConnection()*" permite ligações quando começam em *Anchors*. O método "*canCreate()*" implementa várias verificações que impedem ao utilizador a criação de ligações não permitidas. Uma lista do tipo de verificações feitas em novas ligações é:

- Confirmação da correspondência entre os tipos de variável dos dois pinos a serem ligados (por exemplo, uma variável do tipo INT não pode ser ligada a uma variável do tipo BOOL) – Esta verificação é feita sempre que se faz uma ligação entre *Function*

Blocks e *Functions*, optou-se por não implementar este funcionamento para etiquetas de variáveis de forma a evitar o caso específico de o utilizador modificar o tipo de variável depois da etiqueta já ter sido colocada. Fica então à responsabilidade do utilizador definir corretamente as variáveis que usa para entrada/saída e à responsabilidade do compilador a ser implementado de detetar erros de correspondência entre variáveis;

- Deteção se uma variável vem de um output ou de um input, sendo que só ligações output->input e input->output são permitidas.
- Confirmar que o input ao qual se está a ligar não tem já alguma ligação feita, visto que em FBD não é permitida a ligação de mais que um output a um input.

Depois das confirmações serem feitas, o método "*canCreate()"* retorna ***true*** caso a ligação seja permitida, sendo assim criada a ligação e passando-se à classe "**addFBDConnectionFeature**", que cria a representação gráfica da conexão.

3.2.3.3.2 AddFBDConnectionFeature

Houve um grande cuidado para que as conexões mantivessem o aspeto e o funcionamento correto independentemente das ações do utilizador. Observou-se que era muito difícil impedir que o utilizador criasse ligações que fugissem à norma de ligações com linhas verticais ou horizontais. Para que estas ligações tivessem sempre o aspeto pretendido foi necessário remover algumas liberdades ao utilizador, nomeadamente a capacidade de adicionar/remover *bendpoints*, também foram implementadas algumas restrições relativas à capacidade do utilizador de mover *bendpoints*.

Para que não fosse possível ao utilizador remover *bendpoints*, a classe "***RemoveBendpointFeature***" foi modificada, nomeadamente o método "*canRemoveBendpoint*()" foi alterado para retornar sempre ***false***.

Para a criação das conexões em si, foram analisados todos os tipos de ligações comuns em diagramas FBD. Três tipos principais de ligações foram encontradas e implementadas:

- Ligações entre dois objetos com um output na esquerda e um input na direita (tipo 1)(Fig. 30.), as mais comuns e simples em termos de número de *bendpoints*;

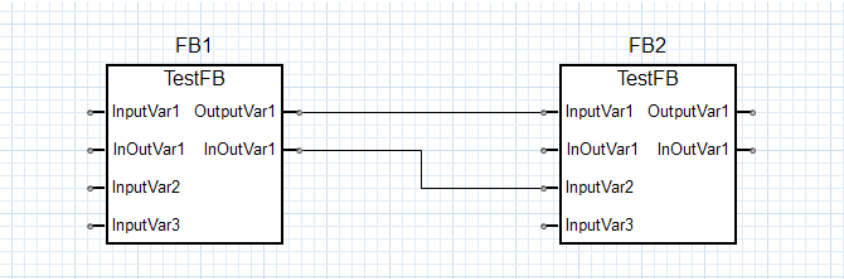


Fig. 30. – Exemplos de ligações tipo 1.

- Ligações entre dois objetos com um input na esquerda e um output na direita (tipo 2) (Fig. 31.): Este tipo de ligação é também comum em diagramas FBD, sendo especialmente comum em casos de *feedback* num sistema:

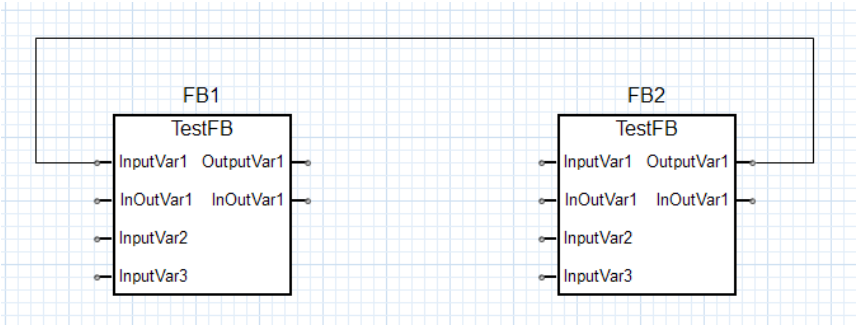


Fig. 31. – Exemplo de ligações tipo 2.

- Ligações entre dois pinos num só FunctionBlock/Function (tipo 3)(Fig. 32.);

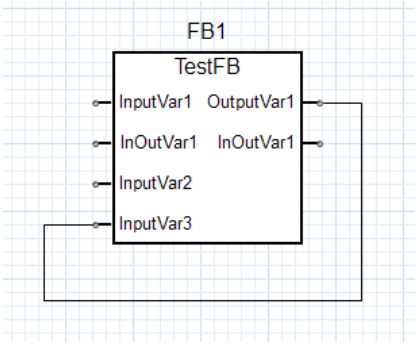


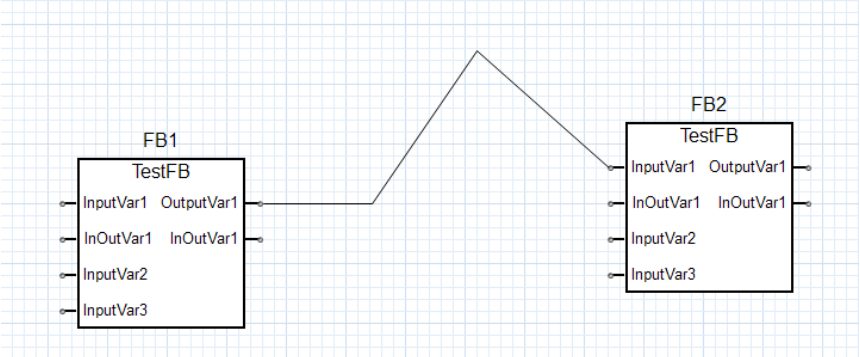
Fig. 32. – Exemplo de ligações tipo 3.

Independentemente do tipo de ligação pretendida, todas as representações gráficas são construídas pela classe ***addFBDConnectionFeature***, no método "*Add()"*. Neste método foi implementado um conjunto de verificações que permite ao código decidir qual dos três tipos de ligações deve ser criada. Dependendo do tipo de ligação pretendida, o código calcula as posições corretas dos *bendpoints* e adiciona-os à ligação, garantindo assim que todas as novas ligações têm sempre o aspeto correto quando são criadas.

3.2.3.3.3 MoveBendpointFeature

Para permitir ao utilizador maior controlo sobre o aspeto dos seus diagramas, foi implementada a classe ***MoveBendpoint***. Esta classe extende a "*DefaultMoveBendpointFeature*", tendo sido o seu método "*moveBendpoint()*" alterado.

Sem esta classe, o utilizador podia clicar em qualquer *bendpoint* já existente e alterar a sua posição sem qualquer restrição. Este funcionamento levava à criação de ligações com linhas diagonais, algo que não é permitido num diagrama FBD (Fig.33.).



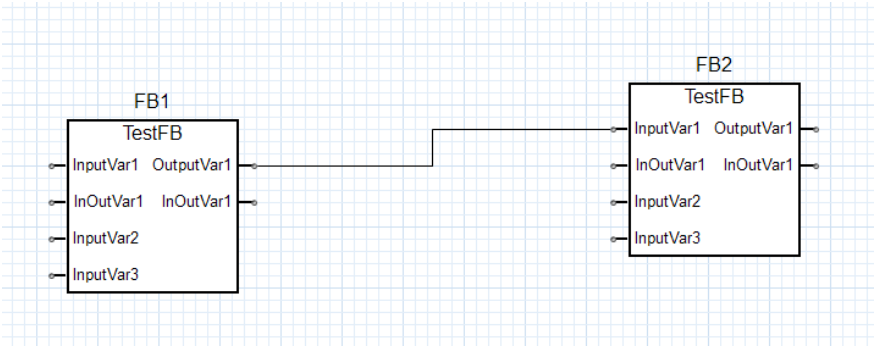


Fig. 33. – Exemplo de movimento de bendpoints, antes e depois de implementar o moveBendpoint.

Para permitir ao utilizador alterar as ligações sem que estas tenham aspetos indevidos, a classe ***MoveBendpoint*** realiza um conjunto de verificações, muito semelhantes às verificações feitas quando uma ligação é criada. Depois das verificações serem feitas, o código analisa o movimento que se quer realizar e, se permitido, este movimento de bendpoint é realizado, sendo a ligação reorganizada para manter o aspeto correto. Caso o movimento não seja permitido (existindo um conjunto de restrições para cada tipo de ligação, que serão referidas a seguir), o código reorganiza a ligação de forma a chegar o mais próximo do que o utilizador tentou, dentro dos limites estabelecidos.

Um exemplo deste funcionamento encontra-se nas figura seguinte (Fig. 34.)

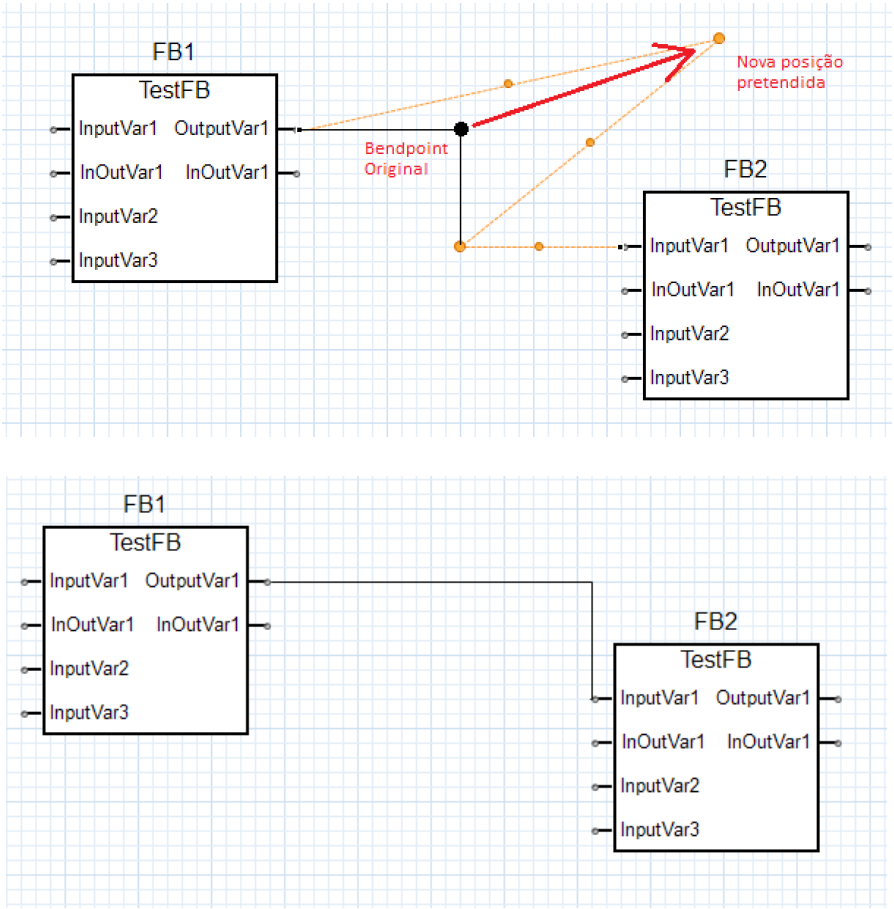


Fig. 34. – Exemplo de funcionamento do moveBendpoint.

Na figura, o utilizador clicou no *bendpoint* superior, tentando-o mover acima do pino original e à direita do pino final. Esta nova posição encontra-se fora dos limites estabelecidos, logo a ligação foi reorganizada de forma a respeitar os limites.

Relativamente às restrições escolhidas para que tipo de movimentos são permitidos, as imagens seguintes demonstram as zonas nas quais não é permitido mover as os *bendpoints*. O tipo

de limites estabelecidos varia conforme dos três tipos de ligações que se está a modificar. As várias restrições aos tipos de ligação são visíveis nas figuras seguintes (Fig. 35, Fig. 36, Fig. 37.)

- Ligações tipo 1 (output à esquerda e input à direita): Neste tipo de ligação os *bendpoints* são forçados a estar horizontalmente entre os pinos de input e output. Os Bendpoints são também automaticamente alinhados verticalmente com os respetivos pinos, dentro destes limites é possível modificar a posição horizontal dos bendpoints;

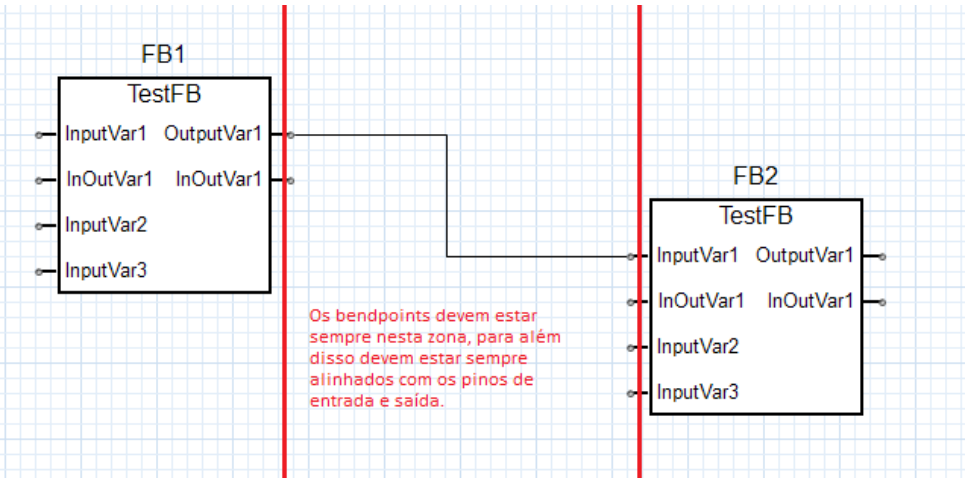


Fig. 35. – Exemplo de ligações tipo 1, com restrições ao movimento de bendpoints.

- Ligações tipo 2 (output à direita e input à esquerda): Nestas Ligações os *bendpoints* não podem ser colocados entre as linhas vermelhas. Para além desta restrição os *bendpoints* mais próximos dos pinos (assinalados a verde), são sempre alinhados com os seus respetivos pinos. Os pinos assinalados a vermelho podem ser movidos horizontalmente e verticalmente, no entanto não é possível movê-los de tal forma que a ligação intersete um dos objetos entre os quais se estabelece a ligação, qualquer ligação que resulte numa interseção com um dos objetos simplesmente é reorganizada acima ou abaixo do objeto que seria intersetado.

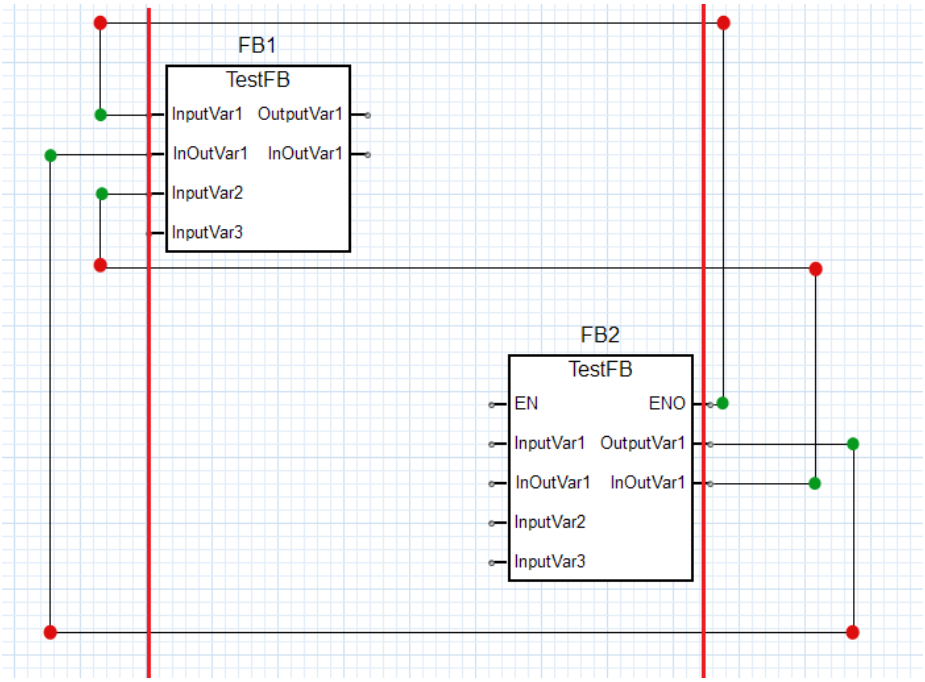


Fig. 36. – Exemplo de ligações tipo 2, com bendpoints assinalados e restrições ao seu movimento.

- Ligações tipo 3 (auto-ligação entre input e output pertencentes ao mesmo objeto): Este caso é semelhante ao anterior, sendo os *bendpoints* próximos dos pinos sempre alinhados com os respetivos pinos e não sendo permitido movimentar os bendpoints de tal forma que as linhas de conexão intersetem o próprio objeto.

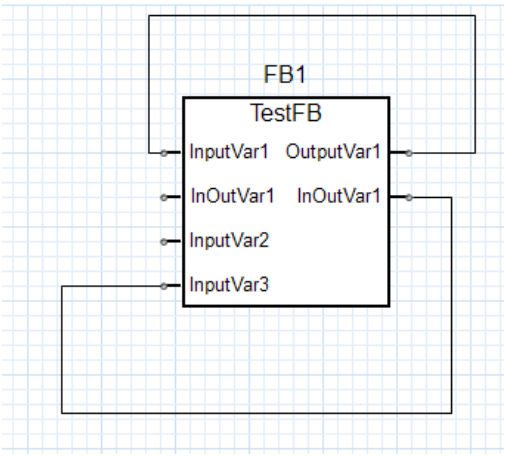


Fig. 37. – Exemplo de ligações tipo 3.

3.2.3.3.4 MoveElement

Apesar de tanto a criação como a modificação de conexões estarem corretamente implementadas para assegurar sempre o aspeto correto, foi necessário assegurar que as conexões existentes entre os objetos eram corretamente atualizadas sempre que um dos objetos fosse movido, para além disso, era necessário assegurar que os atributos dos objetos (posição x/y) eram corretamente atualizados sempre que um deles era movido, porque os métodos de criação de

conexões dependiam destes atributos. Para tratar das movimentações de objetos, a classe ***MoveElement*** foi implementada. Esta classe realiza a translação do objeto, atualizando os seus atributos corretamente. Para além desta tarefa, esta classe também percorre todas as ligações que saem ou entram nos pinos do objeto movido, eliminando-as e recriando-as através da classe ***AddFBDConnectionFeature*** previamente apresentada. Como as ligações são sempre refeitas ao mover um objeto, é possível passar as ligações do tipo 1 para o tipo 2 e vice-versa quando apropriado, mais uma vez assegurando sempre um aspeto correto. Na figura seguinte (Fig. 38) pode-se observar o resultado de uma movimentação de um objeto com e sem a classe ***MoveElement*** implementada:

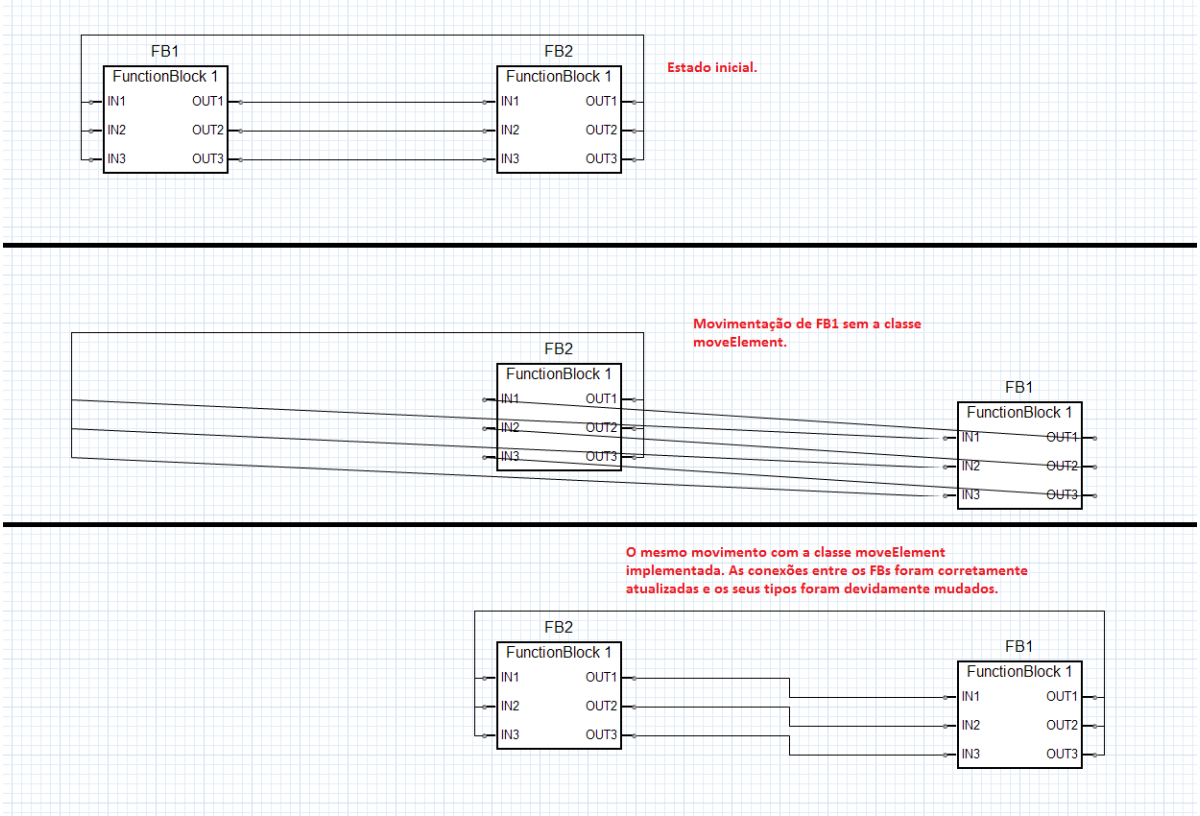


Fig. 38. – Exemplo de movimentação sem moveElement e com moveElement.

3.2.3.4 UpdateFeatures, DoubleClickUpdateFeature

Concluídas as conexões, serão agora tratadas algumas funcionalidades extra que foram implementadas no editor FBD, nomeadamente as ***UpdateFeatures***. As *UpdateFeatures* são classes dedicadas à atualização dos componentes gráficos de um diagrama *Graphiti*, normalmente estas servem para detetar alterações nos objetos de modelo e permitir a reorganização dos componentes gráficos corretamente. No caso do editor FBD estas *UpdateFeatures* foram implementadas para todos os objetos gráficos. A maneira pela qual o utilizador pode aceder a estas funcionalidades está definida através do duplo clique no objeto que pretende modificar. Para permitir este funcionamento foi implementada a classe ***DoubleClickUpdateFeature***, esta *Feature*, que estende a classe *AbstractCustomFeature*, realiza só o pedido da *UpdateFeature* para o objeto onde se clique duas vezes, cabendo depois à *UpdateFeature* respetiva modificar o objeto.

Quanto às *UpdateFeatures* implementadas, foram criadas as seguintes classes:

- *UpdateBlockObjectFeature* – Para FBs e *Functions;*
- *UpdateVarLabelFeature* – Para as etiquetas de variáveis;

O funcionamento das *UpdateFeatures* é simples, sempre que são executadas, preparam-se para eliminar o objeto a ser modificado. Executam o método "*canAdd*()" do objeto (que, no caso dos objetos do editor FBD, fazem com que um diálogo de criação seja apresentado ao utilizador), se o utilizador configurar corretamente o seu objeto e clicar em "OK", o objeto original é destruído, sendo construído um objeto novo com as especificações definidas pelo utilizador no local do objeto original.

Este funcionamento tem algumas limitações, o aspeto de um Function Block ou Function pode não acompanhar o objeto original se for alterado, por exemplo, o ficheiro ".tvar" respetivo, sendo necessário o utilizador recriar o objeto através da *updateFeature*, esta limitação deve-se ao funcionamento do *Graphiti.* Seria possível implementar uma função que verificasse sempre o ficheiro ".tvar" para discrepâncias com a representação gráfica, mas isso iria resultar em várias verificações a serem executadas regularmente para cada POU colocado.

3.2.4 Conclusão

Para resumir, o editor FBD implementado suporta as seguintes funcionalidades:

- Criação de todos os componentes base de diagramas FBD, nomeadamente as variáveis e os *Function Blocks* e *Functions*, completos com diálogos funcionais que permitem a configuração dos mesmos*;*
- Criação de conexões entre os objetos, com conexões robustas que se comportam de forma intuitiva e mantêm o aspeto composto por linhas horizontais e verticais em todas as situações;
- Implementação de funcionalidades extra, tais como a capacidade de reconfigurar os objetos através do duplo clique e a implementação do suporte para bibliotecas de POUs pré-definidas criadas pelos utilizadores. Apesar destas funcionalidades não serem essenciais para a criação de um editor, permitem um funcionamento mais versátil;

Em termos de funcionalidades, o editor FBD dá-se por concluído, estando a par com outros editores semelhantes para o mesmo efeito. Uma possível melhoria futura seria uma possível expansão da forma como as conexões entre elementos são lidadas. Atualmente usa-se uma abordagem à base de casos-tipo que poderia ser substituída por uma abordagem genérica que permitisse ao utilizador criar e remover *bendpoints* à vontade e manter o aspeto correto com ligações verticais e horizontais. A principal razão pela qual esta abordagem não foi tomada deveu-se a limitações inerentes ao funcionamento do *Graphiti*, mas possivelmente versões futuras do *Graphiti* poderão suportar este tipo de funcionamento corretamente.

3.3 Editor LD

O desenvolvimento do editor LD tomou um caminho muito semelhante ao do editor FBD. A estrutura de ambos os editores é semelhante, sendo ambos tipos de diagrama *Graphiti.* A apresentação do desenvolvimento do editor LD foi muito parecida, sendo que nesta secção os mesmos tópicos serão abordados.

3.3.1 Metamodelo ecore

Tal como no editor anterior, foi necessário criar um metamodelo ecore para estabelecer o tipo de objetos de modelo a ser utilizados pelo *Graphiti*. Os componentes gráficos a ser implementados foram, em geral, mais simples do que os *Function Blocks* e *Functions*, sendo apenas de três tipos: ***PowerRails***, ***Contacts*** e ***Coils***, com diferentes subtipos existentes para cada.

No **Anexo B** pode-se encontrar um diagrama de classes que representa o metamodelo implementado (Fig. 39.), este anexo deve ser consultado para melhor compreensão da estrutura do metamodelo e das interações entre os vários objetos.

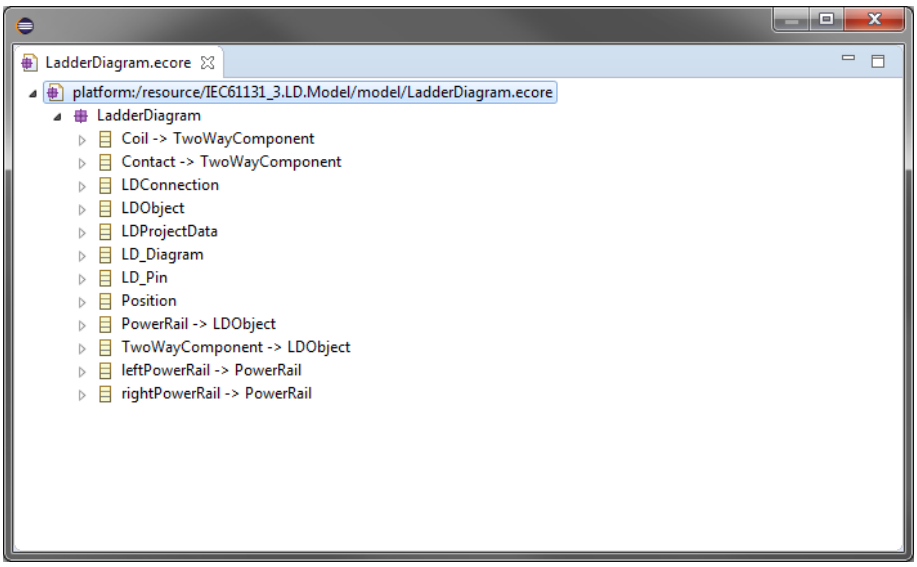


Fig. 39. – Metamodelo ecore implementado para o editor LD.

De forma a poderem ser facilmente geridas as ligações, foi adotado um sistema de pinos semelhante ao do editor anterior. A classe que implementa o pino no editor LD é o **LD_Pin**, este pino não guarda qualquer informação relativa a tipo de dados, porque as ligações entre componentes LD são sempre booleanas. Esta abordagem à volta de pinos foi escolhida mais uma vez de forma a facilitar a criação de um exportador XML no futuro, sendo que desta forma se torna mais simples ler as ligações entre objetos através do código, especialmente no caso de *PowerRails*.

3.3.2 Estrutura do editor

O editor tem uma estrutura muito semelhante ao editor previamente criado, utilizando os *extension points* que o definem como um tipo de diagrama, "**org.eclipse.graphiti.ui.diagramTypes"** e **"org .eclipse.graphiti.ui.diagramTypeProviders"**, e também foi utilizado o *extension point* "**org.eclipse.core.contenttype.contentTypes**", que define todos os objetos do tipo ".ld" como diagramas criados neste editor.

Algumas funcionalidades foram diretamente copiadas do editor FBD, estas funcionalidades são o uso da ***DoubleClickUpdateFeature*** que permite a reconfiguração de elementos, a classe ***deleteWithoutPrompt***, essencial para desativar os avisos sempre que se elimina algum componente, e para limpar os pinos sem *parent* do modelo e evitar aumento indesejado do tamanho dos ficheiros gravados. Outras classes foram copiadas, tais como a FBDobjetcData, agora com um nome LDobjectData, que desempenha a mesma função para os elementos LD. Outras classes foram copiadas do editor FBD, para assegurar um funcionamento semelhante.

A classe *diagramTypeProvider* foi também implementada para este editor neste caso intitulada de "**LDDiagramTypeProvider**", que estabelece o tipo de diagrama para que este possa ser interpretado pelo *Graphiti*. Para além desta classe foram também criadas a *FeatureProvider* e *ToolBehaviorProvider*, com os nomes "**LDFeatureProvider**" e "**ToolBehaviorProvider**".

Para facilitar a interpretação da estrutura do editor implementado, o anexo D contém um diagrama de classes do FeatureProvider que mostra as *Features* implementadas.

3.3.3 Funcionalidades Implementadas

Para os diagramas LD, foram definidos quatro tipos de objetos a ser implementados, sendo estes os Contacts, as Coils e os PowerRails esquerdo e direito. Estes objetos, para além das conexões entre eles, formam os componentes base de todos os diagramas LD, sendo ainda possível e esperado que um utilizador possa introduzir objetos de diagramas FBD, esta segunda parte da integração com o editor FBD, um objeto opcional desta dissertação, será tratada numa secção seguinte, sendo agora descritas apenas as funcionalidades exclusivas ao editor LD.

3.3.3.1 Coils e Contacts

Como foi já referido no capítulo 2, os *Coils* e *Contacts* têm subtipos que modificam o tipo de funcionamento esperado. Estes diferentes subtipos são contidos no metamodelo do diagrama LD no atributo "Type" da classe "TwoWayComponent", que é a classe base dos *Coils* e *Contacts* , desta forma só foi necessário implementar duas *Features* para os *Coils* e *Contacts*. As classes implementadas são as seguintes:

- CreateCoilFeature;
- CreateContactFeature;
- AddCoilFeature;
- AddContactFeature;

As classes **CreateCoilFeature** e **CreateContactFeature**, estão encarregues de instanciar os objetos do modelo através da classe **LadderDiagramFactory** gerada automaticamente na criação do metamodelo ecore. As classes **AddCoilFeature** e **AddContactFeature** são as classes encarregues da construção das representações gráficas dos objetos.

3.3.3.1.1 Diálogos de configuração

Dentro das *AddFeatures*, encontra-se o método "*canAdd*()" que é utilizado para decidir se é permitido ou não adicionar objetos, este método é utilizado para apresentar ao utilizador um diálogo de configuração que permite escolher o tipo de *Coil*/*Contact* a ser adicionado. Para além do tipo de objeto a ser criado, é também possível definir a variável a ser associada ao novo objeto, escrevendo o seu nome. Os diálogos foram implementados pelas classes "***PromptCoil***" e "***PromptContact***", na *package* **iec61131_3.ld.dialogs**.

Nas figuras seguintes (Fig. 40. e Fig. 41.) podem ser observados ambos os diálogos criados para as *Coils* e *Contacts*, respetivamente.

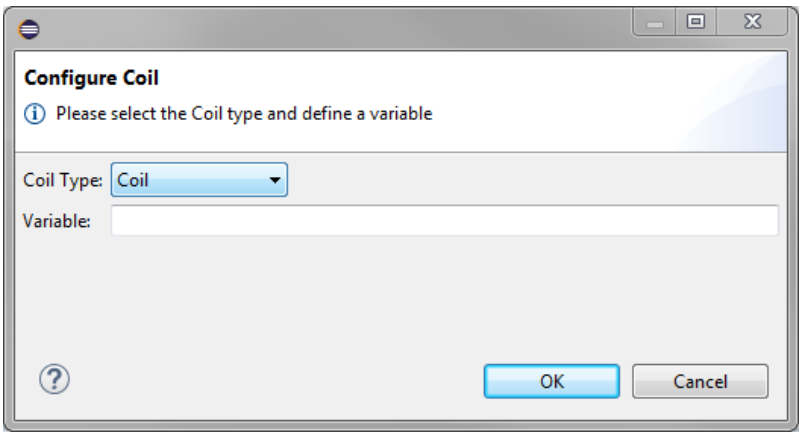


Fig. 40. – Diálogo de configuração de Coils.

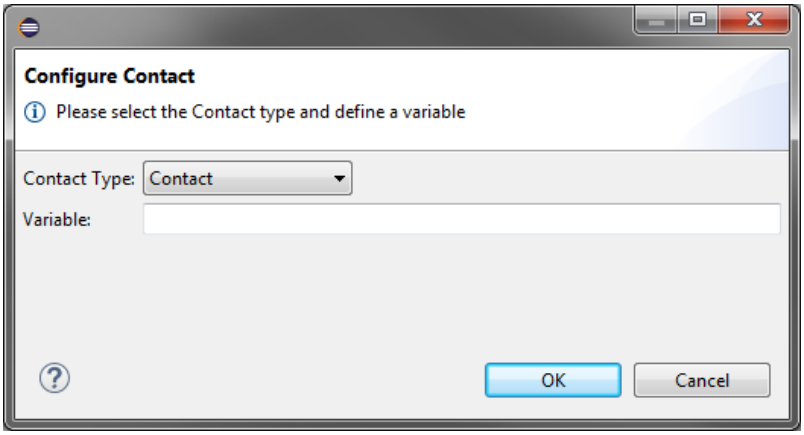


Fig. 41. – Diálogo de configuração de Contacts.

3.3.3.1.2 Representações Gráficas

Como foi anteriormente referido, as *AddFeatures* das *Coils* e dos *Contacts* tratam da representação gráfica dos objetos. O método "*Add*()" foi modificado de forma a construir o aspeto básico de uma *Coil* e do *Contact*, dependendo depois das informações contidas no objeto do modelo o nome da variável é adicionado acima e um caracter é colocado no centro do objeto, para representar o seu tipo.

Os tipos de Coils implementadas (Fig. 42.) são:

- *Coil* normal;
- *Coil* invertida;
- *Coil* "*set*";
- *Coil* "*reset*";
- *Coil* "*rising edge*";
- *Coil* "*falling edge*";

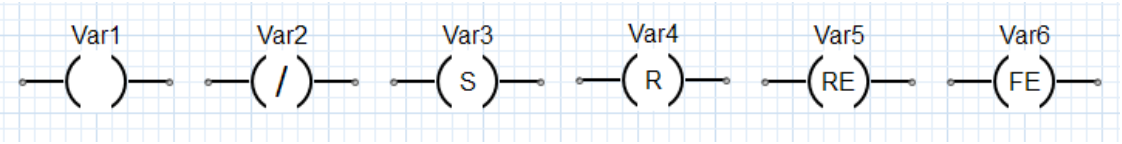


Fig. 42. – Representações gráficas de todos os tipos de Coil implementadas

No caso dos *Contacts*, foram implementados os seguintes tipos (Fig. 43.):

- *Contact* normal;
- *Contact* invertido;
- *Contact* "*falling edge*";
- *Contact* "*rising edge*";

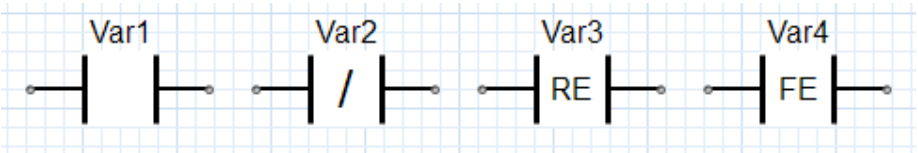


Fig. 43. – Representações gráficas de todos os Contacts implementados

3.3.3.2 PowerRails

Os *PowerRails* foram implementados como objetos independentes, em vez de dois subtipos de um só objeto "*PowerRail*". Como os dois tipos diferentes de *PowerRail* são classes diferentes, foi necessário criar uma *AddFeature* e uma *CreateFeature* para cada um. As *Features* implementadas relativas aos *PowerRails* são:

- *CreateRightPowerRailFeature*;
- *CreateLeftPowerRailFeature*;
- *AddRightPowerRailFeature*;
- *AddLeftPowerRailFeature*;

Mais uma vez, as CreateFeatures limitam-se a instanciar os *PowerRails* e as *AddFeatures* constroem a representação gráfica.

3.3.3.2.1 Diálogo de Configuração

Os *PowerRails* têm apenas um atributo que requer configuração, o número de pinos. Para realizar a escolha do número de pinos foi implementado um diálogo simples com o nome "***PromptPinAmmount***". Este diálogo é apresentado ao utilizador no método "*canAdd()*" de qualquer um dos dois *PowerRails*. A figura seguinte mostra o aspeto da janela de diálogo:

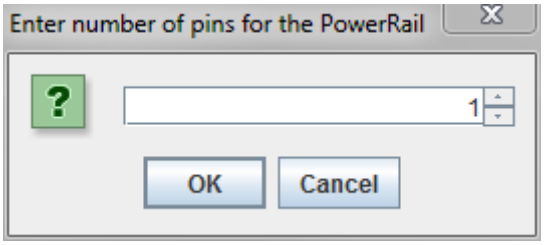


Fig. 44. – Diálogo de configuração de PowerRails

3.3.3.3 Conexões

De forma semelhante ao que foi feito para o editor FBD, as conexões foram implementadas através de um par de classes, nomeadamente a ***CreateLDConnectionFeature*** e ***AddLDConnectionFeature***. Estas ligações foram implementadas de forma quase idêntica às ligações do editor FBD, mas existem algumas diferenças:

- O método "*canCreate*()" da classe *CreateLDConnectionFeature* realiza menos verificações, sendo só garantido que uma ligação é permitida apenas entre uma entrada e uma saída, sendo ainda só permitida a criação de ligações entre dois componentes diferentes;
- A classe *AddLDConnectionFeature* implementa só um dos três tipos de ligação implementados no editor FBD, isto deve-se ao facto de normalmente as ligações LD seguirem sempre a mesma direção, da esquerda para a direita. A figura seguinte (Fig. 45.) mostra o aspeto das ligações implementadas:

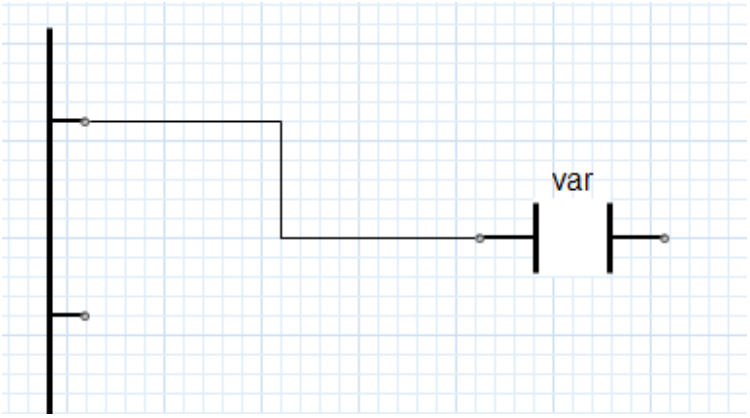


Fig. 45. – Exemplo das conexões implementadas para o editor LD

3.3.3.3.1 MoveBendpoint e MoveElement

A classe **MoveBendpoint** foi de novo utilizada. Como só existia um tipo de ligação para o diagrama LD esta classe foi muito mais simples. As restrições são semelhantes às do diagrama LD, forçando sempre ligações compostas por linhas horizontais e verticais (Fig.46.).

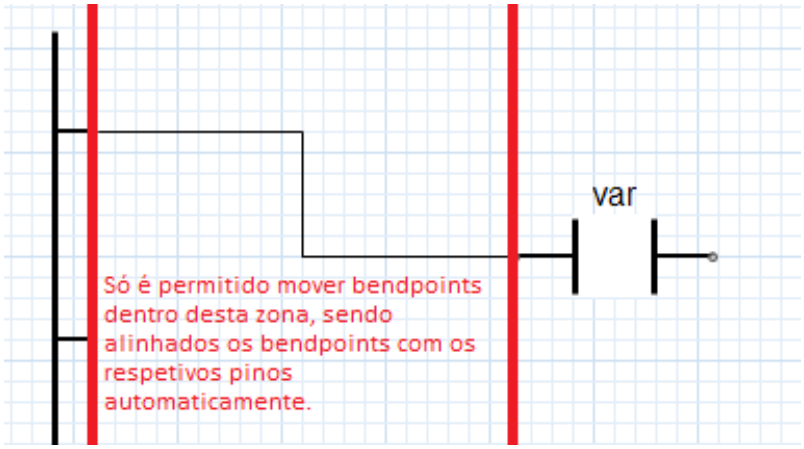


Fig. 46. – Exemplo das restrições aplicadas ao reposicionamento de bendpoints para ligações LD

A classe MoveElement foi de novo utilizada, sendo idêntica à classe com o mesmo nome utilizada no editor FBD. A razão pela qual não foi necessário alterar esta classe deve-se ao seu funcionamento de eliminar e reconstruir as conexões sempre que se move um objeto. Desde que a *Feature* encarregue de criar as ligações esteja correta as ligações são sempre reconstruídas corretamente ao mover um objeto.

3.3.3.4 UpdateFeatures

De forma semelhante ao editor FBD, a classe ***DoubleClickUpdateFeature*** foi implementada, esta classe limita-se a chamar a *updateFeature* relevante quando o utilizador faz duplo clique num objeto.

As *updateFeatures* foram criadas também para os novos objetos, sendo elas:

- *UpdateCoilFeature*;
- *UpdateContactFeature*;
- *UpdatePowerRailFeature*;

No caso das *Coils* e *Contacts*, as *updateFeatures* permitem a alteração da variável associada aos objetos, e permitem a alteração do tipo de objeto. Desta forma torna-se possível ao utilizador colocar vários *Coils* e *Contacts* nas posições que pretende que estes estejam, e só depois preocupar-se com a sua configuração.

Para os PowerRails, a updateFeature limita-se a mostrar de novo o diálogo de configuração de PowerRails, através do qual o utilizador pode escolher o número de pinos que pretende. Desta forma é possível adicionar ou remover pinos ao powerRail a qualquer altura.

3.3.3.5 Integração com editor FBD

A Norma IEC 61131-3 define que num diagrama na linguagem LD é permitido inserir objetos tais como Function Blocks, Functions, e variáveis de entrada/saída/entrada-saída. Para permitir esta funcionalidade no editor LD, foi necessário interligar este editor com o editor FBD. O processo de integração começou pela modificação da classe ***LDFeatureProvider***. Devido ao funcionamento do *Graphiti* em que qualquer *AddFeature* e *CreateFeature* devidamente declarada no *FeatureProvider* aparece automaticamente na interface de utilizador, foi só necessário importar as várias classes *AddFeature* e *CreateFeature* relativas a cada um dos objetos FBD para estes poderem ser utilizados nos diagramas LD.

Foi necessário realizar várias modificações às classes relacionadas com as conexões entre objetos para suportar os objetos FBD em simultâneo com os objetos LD. No caso da ***AddLDConnectionFeature***, foi inserido o código relevante da ***AddFBDConnectionFeature***. O código deteta o tipo de objetos de origem/fim para cada ligação nova e, caso a ligação seja entre dois objetos FBD, o código realiza ligações com o código do ***AddFBDConnectionFeature***. Se a nova ligação é entre dois objetos LD, ou entre um objeto FBD e um objeto LD, o código constrói uma ligação utilizando o mesmo tipo de ligação mostrado na secção 3.3.3.3.

A classe ***MoveBendpoint*** foi modificada de forma a adicionar ao funcionamento já descrito na secção 3.3.3.3.1 o tipo de restrições e correções automáticas referidas na secção 3.2.3.3, caso a ligação a ser manipulada seja entre dois objetos FBD.

3.3.4 Conclusão

O editor de diagramas LD conseguiu implementar todas as funcionalidades esperadas de um editor do seu tipo. Em suma, as funcionalidades introduzidas foram:

- Criação de Coils, Contacts e PowerRails com todos os seus subtipos corretamente representados, com caixas de diálogo simples que permitem a sua configuração;
- Conexões com um funcionamento robusto que asseguram um aspeto correto tal como definido pela Norma em qualquer situação;
- Criação de funcionalidades extra como permitir ao utilizador a reconfiguração de qualquer elemento do diagrama com um duplo clique;
- Integração com o editor FBD previamente implementado, com todas as funções corretamente importadas;

No fim do seu desenvolvimento, o editor LD está praticamente completo. Devido à forma como o *Graphiti* só permite ligações entre dois objetos, o tipo de ligações características de diagramas LD não foi possível ser implementado totalmente. Para que este tipo de ligações fosse possível, em que mais do que dois objetos podem partilhar a mesma ligação, seria necessário modificar o *Graphiti*. Uma alternativa possível para resolver esta falha no *Graphiti* será um tratamento mais cuidado de como ligações entre objetos são interpretadas quando o exportador XML for implementado.

Capitulo 4

4. Testes e Conclusões

Os objetivos principais do trabalho desta dissertação, a criação de editores LD e FBD, foram concluídos corretamente. Os objetivos opcionais da criação de exportadores/importadores para os novos diagramas não puderam ser implementados devido a limitações de tempo. Foi, no entanto, possível realizar a integração do editor FBD com o editor LD corretamente, restando apenas a integração do editor LD com o editor SFC.

Neste capítulo serão mostrados testes às várias funcionalidades dos editores implementados, de forma a demonstrar todas as suas capacidades. Para além dos testes apresentados neste capítulo, foram realizados muitos mais testes ao longo do desenvolvimento dos editores. Estes testes tiveram variados níveis de complexidade e especificidade e foram feitos de forma a antecipar o maior número possível de situações que podem acontecer ao longo da utilização normal dos editores. Ao fim de cada secção serão visíveis tabelas que referem a maioria dos testes realizados e os seus resultados. Quaisquer erros que foram encontrados ao longo destes testes foram devidamente corrigidos.

4.1 Testes ao editor FBD

Nesta secção serão mostrados vários testes às funcionalidades implementadas no editor FBD. Estes testes começam pela criação de um projeto, seguido da criação de todos os elementos gráficos implementados através da construção de um diagrama complexo, e concluindo com testes às conexões, ao movimento de *bendpoints*, e ao movimento de objetos com conexões já presentes.

Para além dos testes demonstrados nesta secção, será apresentada no final uma tabela que documenta o tipo de testes que foram realizados, com os resultados obtidos.

4.1.1 Criação de novo diagrama FBD

Para criar um novo diagrama FBD através do navegador IEC 61131-3 pode-se utilizar os novos botões como visto nas figuras (Fig. 47, Fig. 48, Fig. 49.). Após clicar no botão é apresentada uma janela de diálogo para inserção do nome do novo POU.

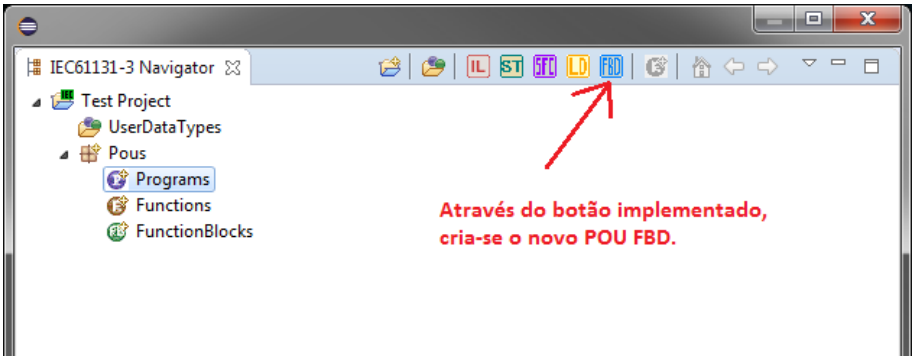


Fig. 47. – Navegador de projetos IEC 61131-3, com os novos botões

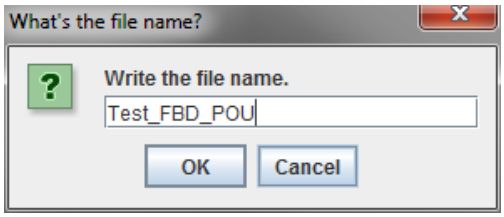


Fig. 48. – Diálogo de criação de novo POU em FBD

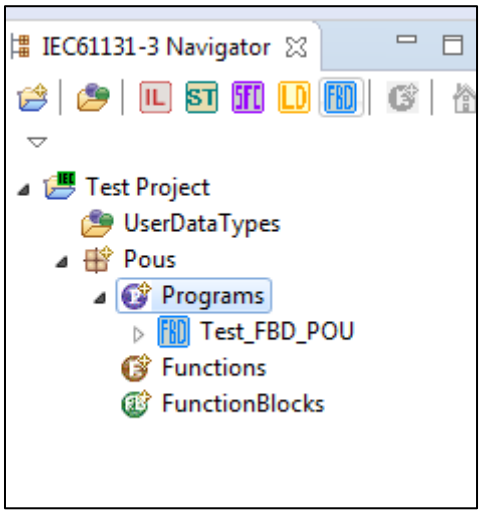


Fig. 49. – Navegador IEC 61131-3, com o novo POU criado

O novo POU é criado, encontrando-se na pasta que estava selecionada, neste caso "Programs".

4.1.2 Criação de diagramas complexos

Ao clicar duas vezes no novo programa FBD criado, é apresentada a interface do editor. Através desta interface é possível introduzir qualquer um dos componentes gráficos criados. Na figura seguinte (Fig. 50.) encontra-se um exemplo de um diagrama de alguma complexidade que demonstra todas as funcionalidades implementadas, sendo algumas destas funcionalidades realçadas a vermelho.

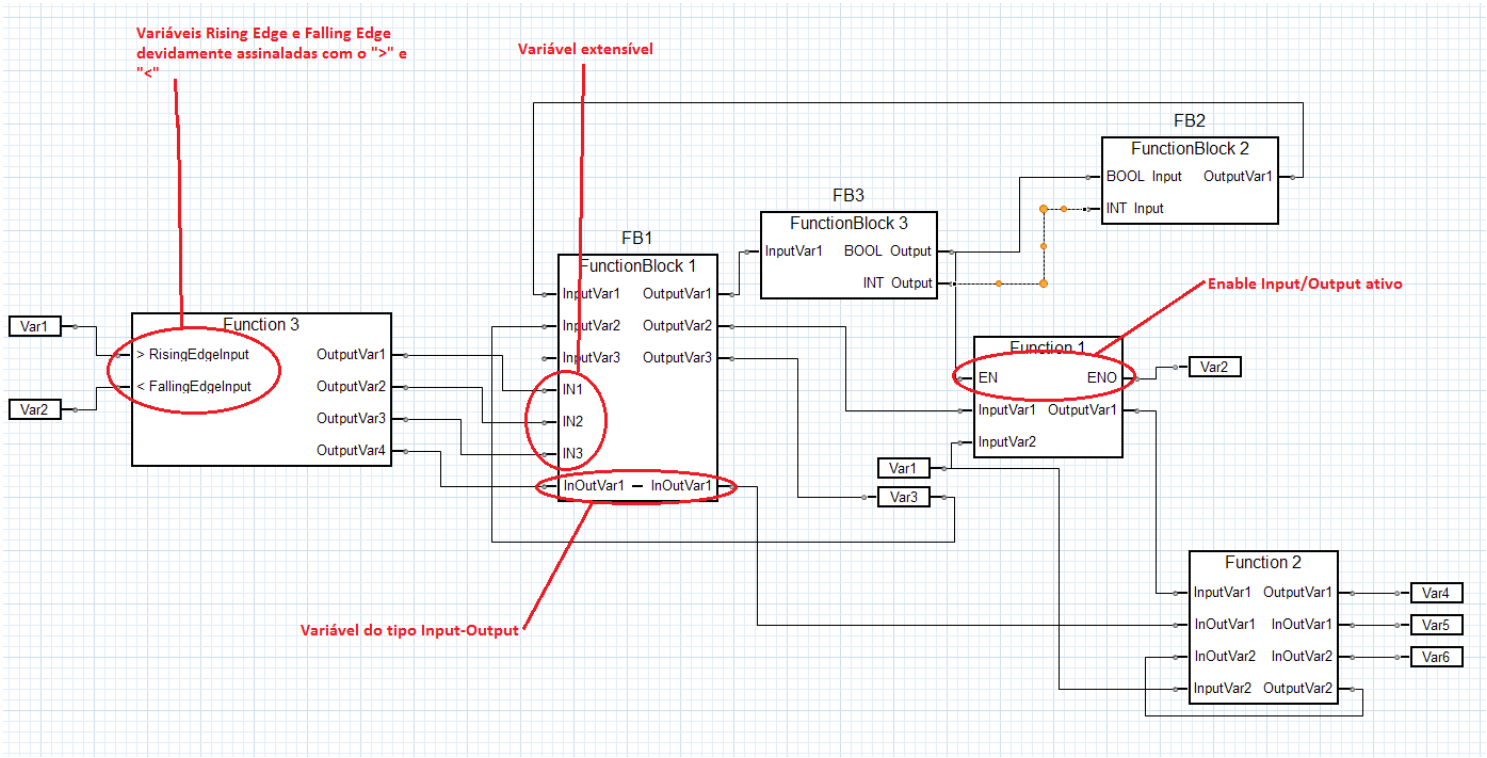


Fig. 50. – Exemplo de diagrama complexo criado com o editor FBD implementado.

Como pode ser observado pela figura, o editor FBD pode suportar diagramas complexos mantendo um aspeto legível e de acordo com a Norma.

4.1.4 Testes de conexões

Nesta secção será testada a criação e modificação de conexões. Visto que as conexões no editor FBD são tão complexas e contêm tantas regras, é necessário provar que tudo funciona como é devido.

4.1.4.1 Teste às restrições de conexão

Como explorado na secção 3.2.3.3, as conexões só são permitidas em alguns casos concretos:

- O tipo de variável deve corresponder (Fig. 51.), caso a ligação seja entre um FB e uma *Function*, ou entre dois FB's ou duas *Functions*.
  Neste caso as variáveis "BOOL Output" e "BOOL Input" São do tipo "BOOL" e as variáveis "INT Output" e "INT Input" são do tipo "INT".

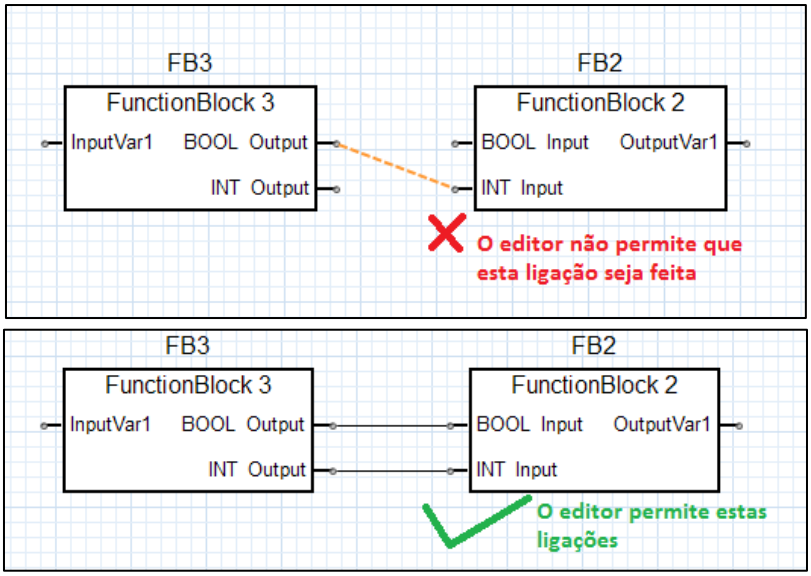


Fig. 51. – Exemplo de restrições aplicadas às ligações, conforme os tipos de variável.

- Só é possível ligar um *output* a um *input*, ou vice-versa (Fig.52.);

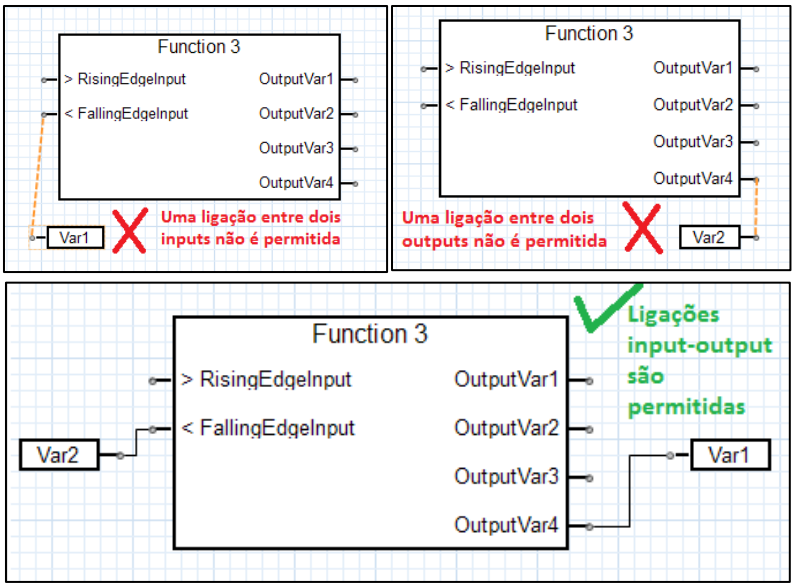


Fig. 52. – Exemplo de restrições às conexões, quanto à correspondência input-output.

- Não pode ser ligado mais que um *output* a um *input* (Fig.53.);

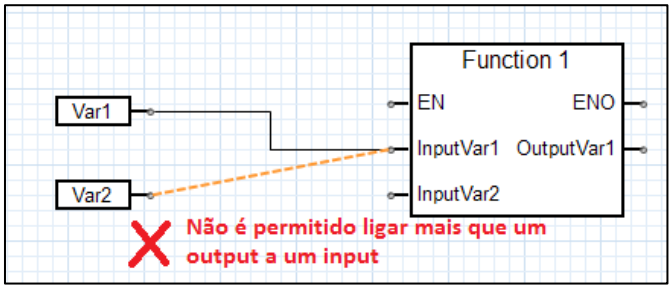


Fig. 53. – Exemplo de restrições de conexão, quanto ao limite de um só output por cada input.

- Um output pode ligar-se a vários *inputs* (Fig. 54.);

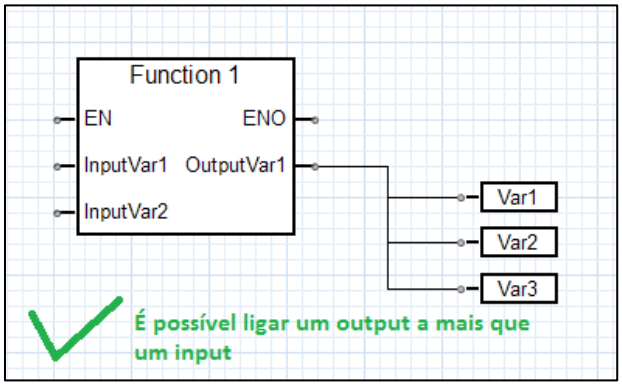
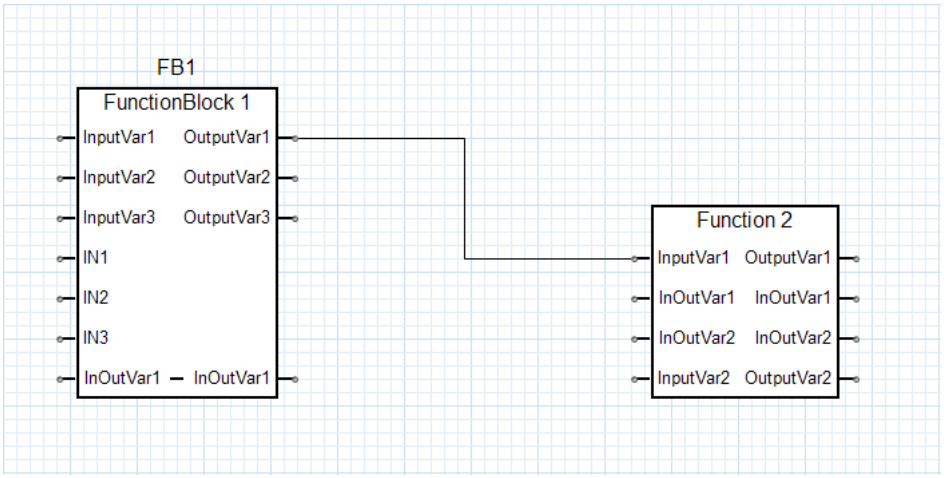


Fig. 54. – Exemplo de capacidade de um output ligar-se a vários inputs diferentes.

4.1.4.2 Teste ao *MoveBendpoint*

Nesta secção serão testadas as funcionalidades de movimentação de *bendpoints*. Nomeadamente serão testadas algumas tentativas de criar ligações incorretas, para os três tipos de ligações, de forma a testar a robustez da classe *moveBendpoint* contra comportamento indevido por parte do utilizador.

Inicialmente será testada a ligação do tipo 1 (secção 3.2.3.3), entre dois objetos com um output na esquerda e um input na direita. Neste teste tentou-se mover um dos bendpoints para além das áreas restritas onde pode estar, dentro de um dos objetos. Nas figuras seguintes (Fig. 55.) pode-se ver a sequência de movimentos, começando pelo estado inicial, passando-se a uma imagem intermédia que mostra o movimento testado e no fim sendo visível o estado após a movimentação.



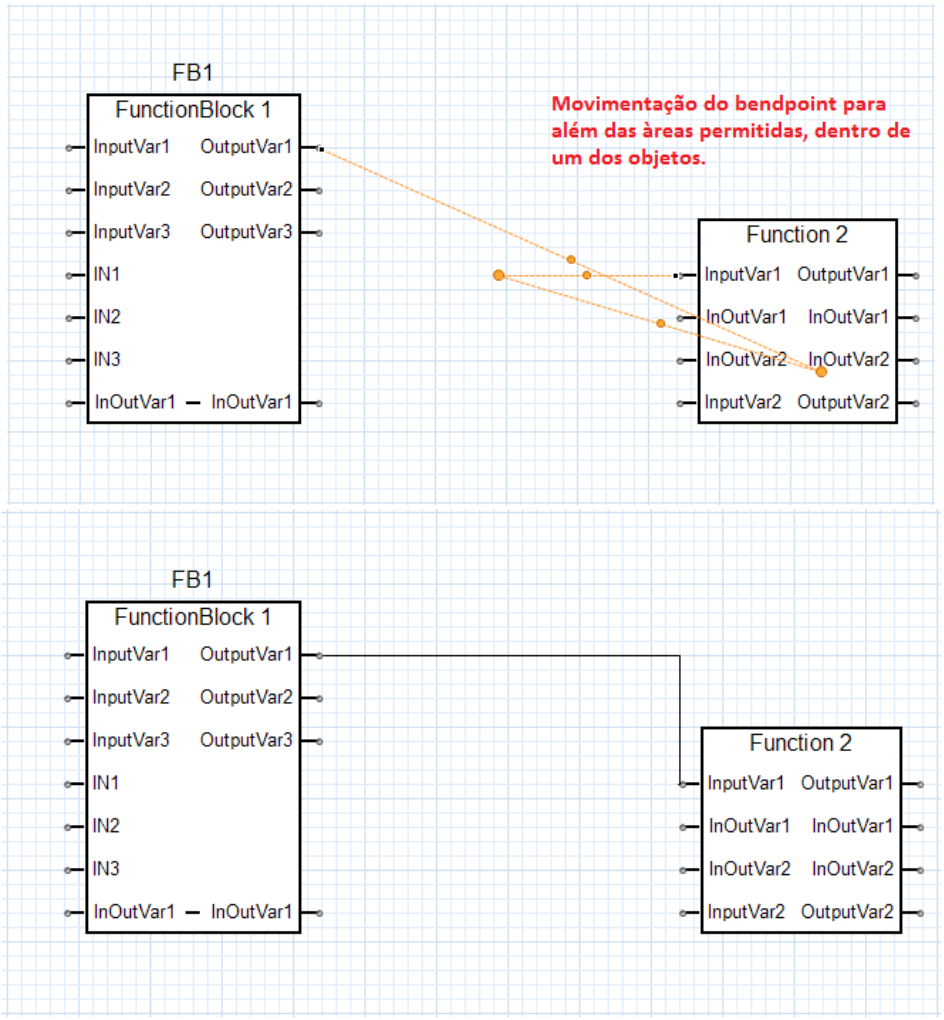


Fig. 55. – Testes ao moveBendpoint para ligações FBD, para ligações tipo 1.

Como pode ser observado pelas imagens, a classe *moveBendpoint* realizou o movimento mais próximo do que o utilizador tentou realizar, sem quebrar as restrições impostas neste tipo de ligação. Como o *bendpoint* foi movido todo para a direita, para além do seu limite direito, acabou colocado em cima do seu limite direito. Também se pode observar que os *bendpoints* foram automaticamente alinhados com os seus pinos.

De seguida serão testadas as ligações tipo 2, em que um output está à direita e um input está à esquerda. Um movimento semelhante ao que foi realizado para o teste anterior foi feito, sendo visíveis nas figuras seguintes (Fig. 56.) as três etapas.

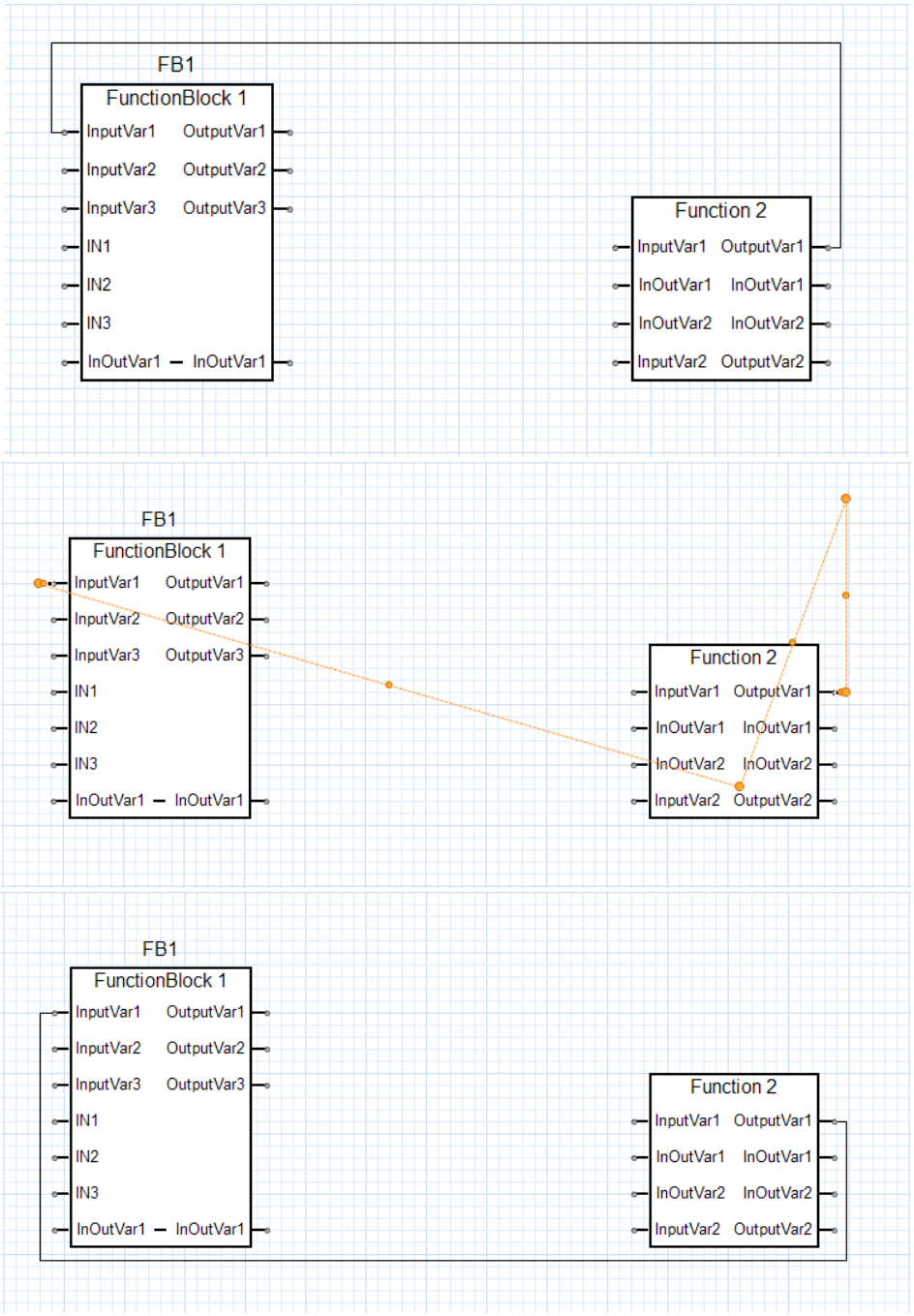


Fig. 56. – Testes ao moveBendpoint para ligações FBD, para ligações tipo 2.

Neste caso houve uma tentativa de mover o bendpoint superior esquerdo para dentro do objeto da direita, muito além do seu limite direito. O código mais uma vez corrigiu este movimento criando uma ligação que passa pela parte inferior de ambos os objetos. O código tenta sempre antecipar a intenção do utilizador ao fazer estas modificações, resultando num comportamento intuitivo.

Não serão mostrados testes para o terceiro tipo de ligações, devido ao seu comportamento muito semelhante às ligações do tipo 2.

Em todos os casos, foram realizados vários testes a vários tipos de movimentos para além destes, tendo sido corrigidos quaisquer erros que foram encontrados. Resultou desta sequência de

testes um código muito robusto que consegue responder sempre de maneira correta ao comportamento do utilizador.

4.1.4.3 Teste ao *MoveElement*

Para testar a capacidade do *moveElement* de atualizar corretamente as ligações, foi realizado um teste com várias ligações dos três tipos. Nas figuras seguintes (Fig. 57 e Fig. 58.) .é possível ver o antes e o depois de mover um objeto. É possível analisar as ligações reformadas corretamente desta forma.

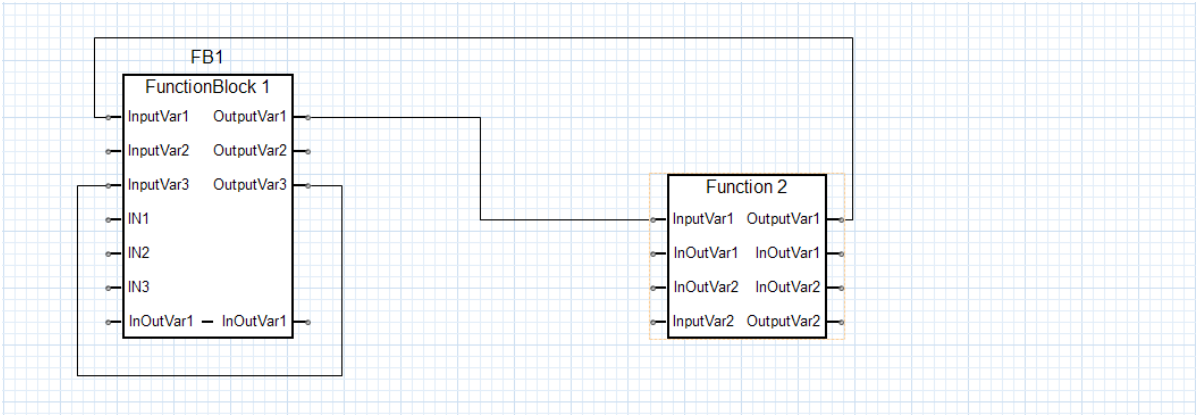


Fig. 57. – Testes ao moveElement, antes de mover.

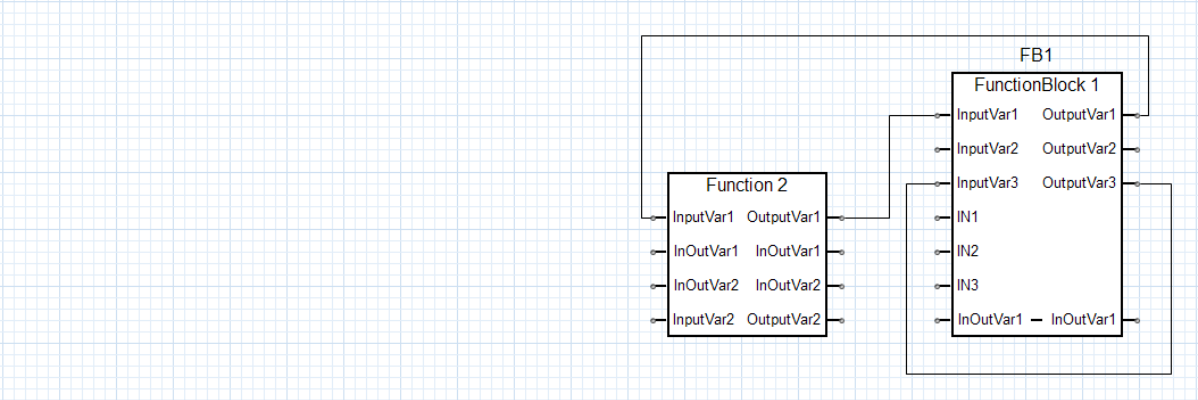


Fig. 58. – Testes ao moveElement, depois de mover.

Como pode ser observado na figura, as ligações mantiveram-se com um aspeto correto, tendo sido devidamente transformadas de ligações entre o tipo 1 e 2 quando as posições horizontais dos objetos foram invertidas. Entretanto a auto-conexão do function Block manteve-se com o aspeto igual como seria de esperar.

4.1.5 Tabela de testes

Como referido no início desta secção, a tabela seguinte contém uma lista de vários outros testes que foram realizados. O propósito desta tabela é expor o tipo de testes que foram realizados e confirmar quais as funcionalidades que foram devidamente testadas e que têm um funcionamento correto.

Tabela 1 – Tabela de testes realizados ao editor FBD

Funcionalidade testada	Descrição	Resultados obtidos	Observações
Criação de novo POU FBD	Criação de novo POU com nome válido	Novo POU criado como esperado	Comportamento correto
Criação de novo POU FBD	Criação de novo POU com nome inválido	Apresentada mensagem de erro, POU não foi criado	Comportamento correto
Criação de Function Blocks / Functions: Testes aos diálogos de configuração	Configuração e criação de novo elemento com uma configuração válida	Novo elemento criado corretamente	Comportamento correto
Criação de Function Blocks / Functions: Testes aos diálogos de configuração	Configuração incorreta de nome para function block. Primeiro com não sendo escolhido um nome e depois com um nome composto por mais que uma palavra	Novo elemento não é criado, é apresentada uma mensagem de erro e diálogo de configuração novamente aberto	Comportamento correto
Criação de Function Blocks/Functions: Testes aos diálogos de configuração	Seleção de função com variáveis extensíveis	Campo “Input Amount” é ativado e desativado quando um outro objeto que não usa variáveis extensíveis é selecionado	Comportamento correto
Criação de Function Blocks/ Functions: Testes aos diálogos de configuração	Ativação de Input/output enable e escolha de um valor para número de variável extensível	Objeto criado corretamente, com os novos dois pinos Input/Output Enable e com o número correto de inputs relativos à variável extensível	Comportamento correto
Criação de Function Blocks/ Functions: Testes aos diálogos de configuração	Clique no botão de “cancel”	Objeto não é criado	Comportamento correto
Criação de Function Blocks/ Functions: Testes aos diálogos de configuração	Criação de novo objeto após ter sido cancelada a criação no teste anterior	Objeto é corretamente criado	Comportamento correto
Criação de Function Blocks/Functions	Criação de objeto à base de ficheiro “.tvar” , com variáveis de input, output, input-output e do tipo Rising Edge/Falling Edge	Objeto corretamente criado, variáveis representadas de forma correta	Comportamento correto
Criação de Function Blocks/Functions	Criação de objeto à base de ficheiro “.tvar”, com mais outputs que inputs	Objeto criado corretamente	Comportamento correto
Criação de Function Blocks/Functions	Criação de objeto com mais inputs que outputs	Objeto criado corretamente	Comportamento correto
Criação de Function Blocks/Functions	Criação de objeto com nome de grandes dimensões	Objeto criado corretamente	Comportamento correto
Criação de etiquetas de variáveis	Criação de etiquetas de Input, Output e Input-Output com variáveis válidas	Objetos criados corretamente	Comportamento correto
Criação de etiquetas de variáveis	Criação de etiquetas de Input, Output e Input-Output com variáveis inválidas	Objeto não é criado, apresentada mensagem de erro	Comportamento correto

	(mais que uma palavra)		
Criação de etiquetas de variáveis	Criação de etiquetas de Input, Output e Input-Output sem variáveis definidas	Objeto é criado, apresentando uma etiqueta em branco	Este tipo de comportamento pode não parecer correto, mas é intencional. Serve para permitir ao utilizador colocar várias etiquetas de uma só vez e só depois dar-lhes nome através da doubleClickUpdateFeature()
Criação de etiquetas de variáveis	Criação de etiquetas com nome de variável de grandes dimensões	Objetos criados corretamente, com tamanho ajustado para mostrar os nomes completamente	Comportamento correto
Criação de etiquetas de variáveis	Cancelamento de criação de etiquetas através do diálogo de configuração	Objetos não são criados	Comportamento correto
Criação de etiquetas de variáveis	Criação de novas etiquetas após cancelamento de criação no teste anterior	Objetos são corretamente criados	Comportamento correto
Criação de conexões	Criação de conexões dos três tipos, em condições válidas	Conexões corretamente criadas	Comportamento correto
Criação de conexões	Criação de conexões em três situações inválidas: conexão entre dois inputs, dois outputs, e em pinos cujas variáveis não têm o mesmo tipo de dados	Conexões não são criadas	Comportamento correto
Testes ao MoveElement	Movimentação de vários objetos, com todos os tipos de conexões presentes. De forma que seja necessário mudar os tipos de conexão	Movimento corretamente feito, objeto de modelo tem as suas posições X e Y corretamente atualizadas e as conexões são devidamente atualizadas para os novos tipos	Comportamento correto
Testes ao MoveBendpoint	Movimentação de bendpoints, pertencentes aos três diferentes tipos de ligação, dentro dos limites corretos	Conexões atualizadas corretamente, sendo os bendpoints devidamente alinhados e um aspeto de ligações mantendo o aspeto correto	Comportamento correto
Testes ao MoveBendpoint	Movimentação de bendpoints, percentences aos três diferentes tipos de ligação, para vários locais indevidos que causariam aspetos incorretos se fossem permitidos	Conexões adaptadas o melhor possível dentro dos limites definidos, aspeto correto mantido	Comportamento correto
Testes às updateFeatures	Teste ao duplo clique e atualização de todos os tipos de objeto	Objetos tomam o aspeto correto após atualização, caso cancelada a atualização os objetos não são modificados	Comportamento correto

Gravação e carregamento de ficheiros	Teste à funcionalidade de gravação e carregamento, após todos os outros testes serem realizados	Diagramas são guardados corretamente. Objetos guardados correspondem corretamente aos objetos vistos na interface gráfica.	Comportamento correto

4.2 Testes ao editor LD

Nesta secção serão demonstrados alguns dos testes mais importantes para demonstrar a funcionalidade do editor LD. Os testes a ser realizados para o editor LD são maioritariamente relacionados com a criação dos vários componentes, a atualização dos componentes e a criação de conexões. Mais uma vez, no final desta secção estará disponibilizada uma tabela com todos os testes efetuados, e os seus resultados.

4.2.1 Criação de novo diagrama LD

Tal como na secção anterior, foi testada a criação de um novo diagrama LD. O procedimento para a criação de um diagrama deste tipo é idêntico ao da criação de um diagrama FBD, sendo necessário apenas clicar no botão correspondente ao diagrama LD. Nas figuras seguintes (Fig.59, Fig. 60, Fig. 61.) pode-se observar a criação de um novo diagrama LD.

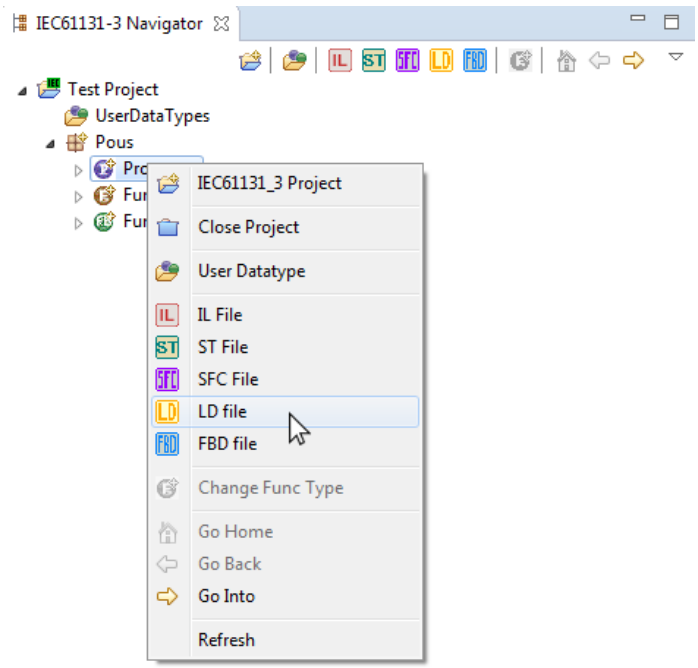


Fig. 59. – Teste à criação de POU em LD.

Nesta figura pode-se observar o método alternativo de criar um novo diagrama, através do botão direito do rato.

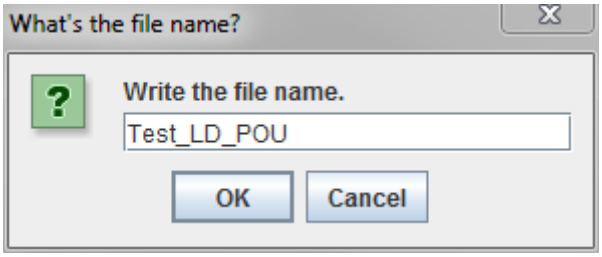


Fig. 60. – Diálogo de criação de novo POU em LD.

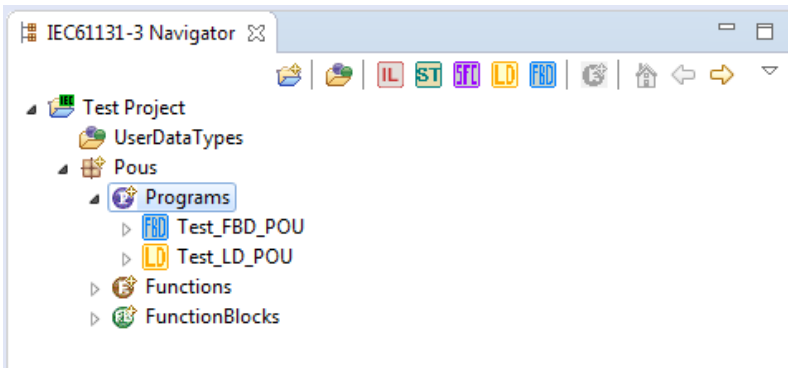


Fig. 61. – Navegador IEC 61131-3, com o novo POU criado.

Criado o novo diagrama LD, serão agora testadas as outras funcionalidades do editor LD. Começando pela criação de diagramas complexos.

4.2.2 Criação de diagramas complexos

Na figura seguinte pode-se observar um diagrama de alguma complexidade criado com o editor desenvolvido (Fig. 62).

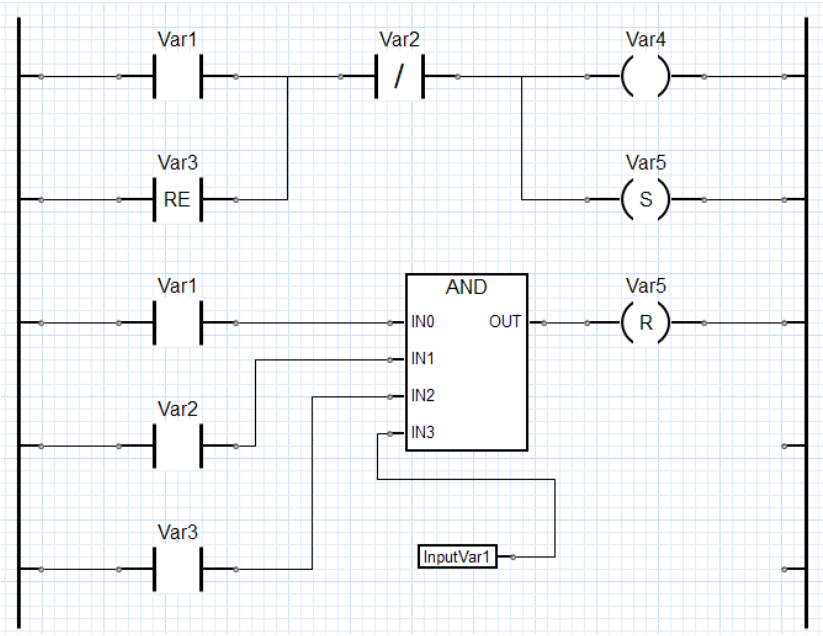


Fig. 62. – Exemplo de diagrama complexo criado no editor LD implementado.

Como pode ser observado na figura, é possível criar diagramas complexos, contendo vários tipos de Coils e Contacts. Também se pode observar a utilização de uma *Function* do editor FBD, com o nome “AND”, e uma etiqueta para uma variável de entrada, com o nome *InputVar1*. Na realidade são possíveis todos os objetos que são utilizados nos diagramas FBD, e estes comportam-se como seria de esperar. Para testar as updateFeatures, imagine-se que se queria adicionar mais uma linha ao diagrama seguinte, com uma variável extra a ligar à função “ADD”. Sem as *updateFeatures* a única maneira de fazer isto seria eliminando o *PowerRail* esquerdo, eliminando a *Function* e substituindo por uma nova *Function* e *PowerRail*. Como as updateFeatures e a *DoubleClickUpdateFeature* foram implementadas, só é necessário clicar no PowerRail e na Function duas vezes, reconfigurá-lo para ter mais um pino cada. Na imagem seguinte (Fig. 63.) pode-se observar o resultado da reconfiguração.

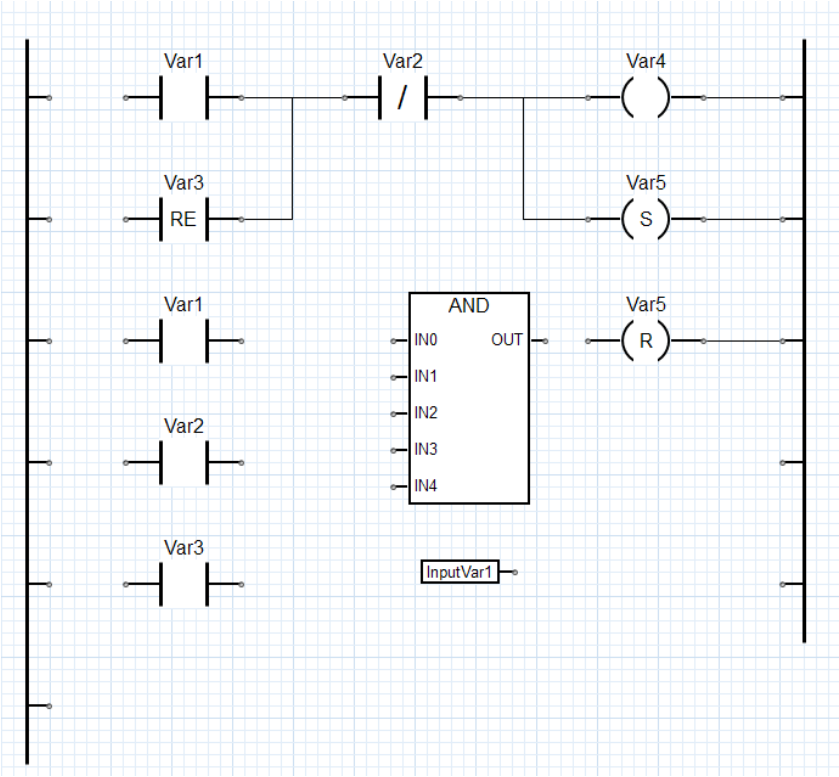


Fig. 63. – Resultado da utilização da updateFeature no PowerRail esquerdo e na Function.

Como pode ser observado na figura, algumas ligações entre os componentes foram eliminadas, esta é uma limitação da forma como a *updateFeature* está implementada, visto que o objeto anterior é eliminado e um objeto novo é criado no mesmo sítio. Apesar desta limitação esta funcionalidade ainda é mais eficiente que a alternativa. No futuro poderá ser possível expandir as *updateFeatures* para guardarem as ligações.

4.2.3 Testes às conexões

De maneira semelhante ao que foi feito para o editor FBD, serão apresentados alguns testes às conexões. As conexões entre componentes FBD já foram apresentadas, portanto só vão ser mostradas as conexões entre componentes LD e entre componentes LD e FBD. As conexões entre dois componentes LD e entre componentes LD e FBD são funcionalmente iguais e muito parecidas com as conexões do tipo 1 apresentadas na secção 3.2.4.

4.2.3.1 Testes às restrições de conexão

No que conta às restrições sobre quais conexões são permitidas, foram implementadas apenas duas restrições:

- Quanto a ligações entre componentes LD, um Output só pode ser ligado a um Input. Este tipo de restrição não faz muito sentido quando se lida com um diagrama LD tradicional, porque normalmente quaisquer ligações que partilhem componentes como uma só ligação. Como o *Graphiti* não permite ligações entre mais que dois componentes, é necessário ao utilizador ligar manualmente todos os outputs aos inputs respetivos. As figuras seguintes (Fig. 64.) exemplificam esta limitação:

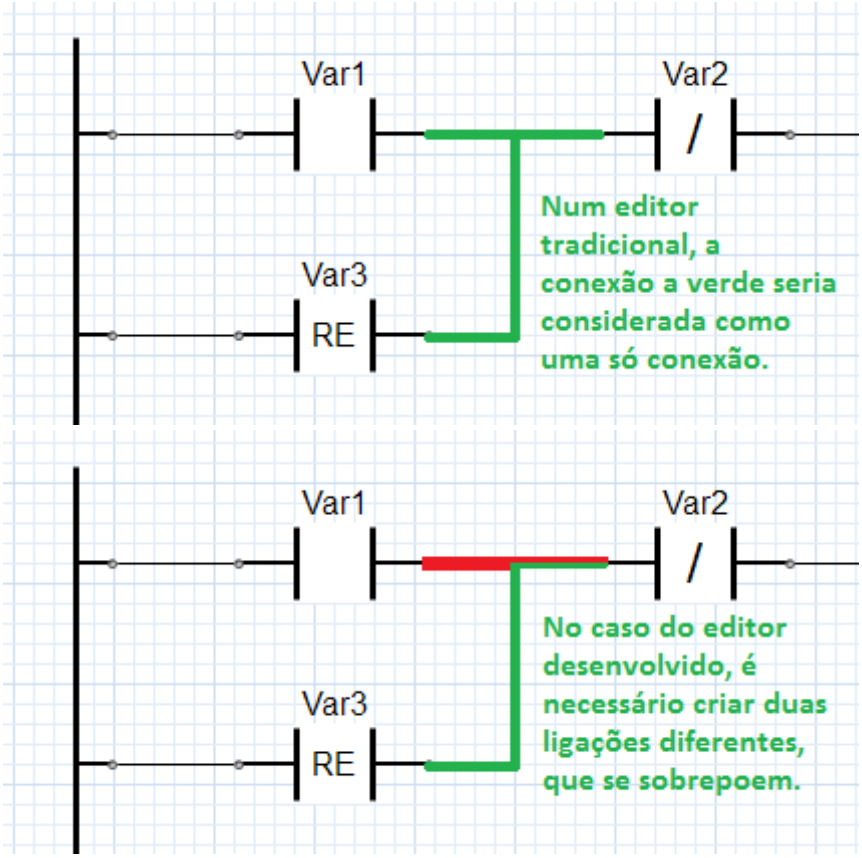
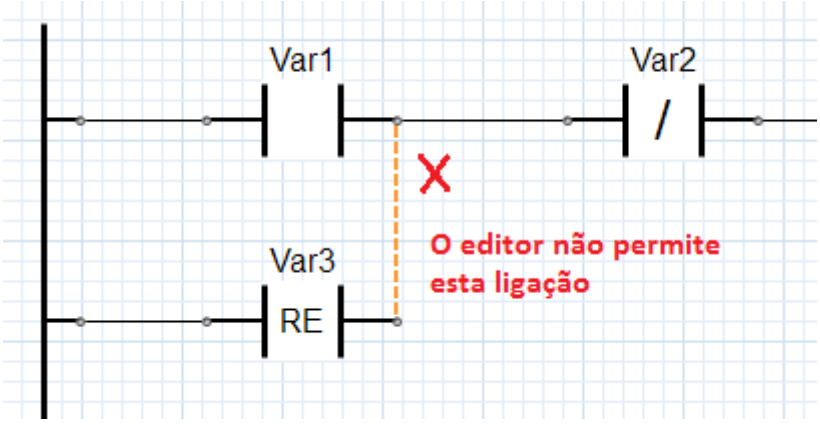


Fig. 64. – Comparação entre ligações implementadas e ligações tradicionais LD.

Como se pode observar nas figuras, o editor interpreta cada ligação individualmente, sendo então essencial restringir o tipo de ligações a output-input. As figuras seguintes (Fig. 65.) exemplificam quais as ligações permitidas.



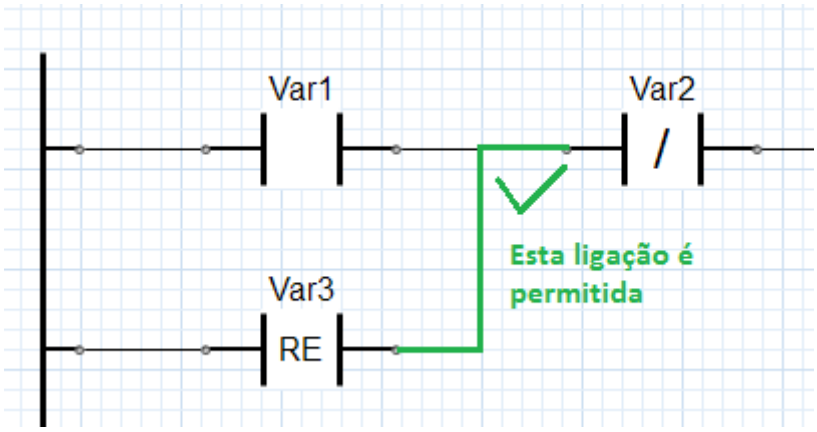


Fig. 65. – Testes às restrições de ligação, neste caso a correspondência input-output.

- Um outro tipo de restrição aplicada pelo editor ao tipo de conexões que se pode realizar é a verificação dos tipos de dados das variáveis. Todos os pinos dos componentes LD contam como variáveis do tipo **BOOL** em termos de ligação, sendo impedido ao utilizador de realizar ligações a componentes FBD cujos tipos de dados das variáveis não sejam BOOL. Esta restrição, apesar de não ser essencial, permite evitar futuros erros por parte do utilizador.

4.2.3.2 Teste ao *MoveElement*

A classe *MoveBendpoint* comporta-se de forma igual à do editor FBD, não sendo necessário demonstrar novos testes para o editor LD. A classe *MoveElement* tem um funcionamento semelhante à do editor FBD, mas com uma limitação que deve ser referida. Esta limitação vem da forma como a classe *AddLDConnectionFeature* foi implementada. As sequência de figuras seguinte (Fig. 66.) mostra a limitação desta classe.

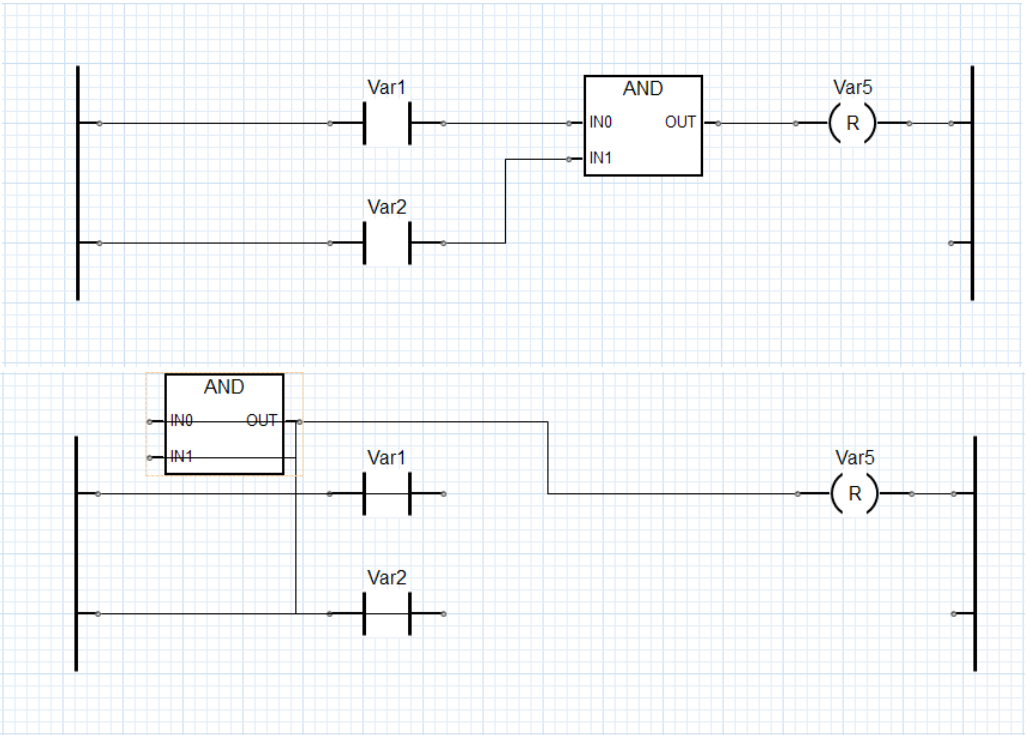


Fig. 66. – Exemplo da representação incorreta resultante da implementação do AddLDConnectionFeature.

Como pode ser observado nas figuras, mover a Function atrás dos *Contacts* resulta num aspeto incorreto das conexões. Este tipo de funcionamento é errado, mas necessário devido à forma como se implementou a addFeature das conexões. Da forma como está feita, a addFeature assume em todos os casos que as ligações entre os componentes LD e FBD são do tipo 1, ou seja, com um *output* à esquerda e um *input* à direita. Foi feita a escolha de não implementar o tipo 2 de ligações para estes casos por uma razão importante. Como já foi referido anteriormente, os diagramas LD contam ligações como se fossem ligações com cabos elétricos. Este tipo de funcionamento é lógico quando se lida só com componentes LD, mas os componentes FBD têm algumas restrições, nomeadamente que um input só pode ser ligado a um output, e que um output nunca pode ser ligado a um outro output. A figura seguinte (Fig. 67.) demonstra um caso raro em que o utilizador poderia criar algo que iria causar problemas na interpretação do diagrama.

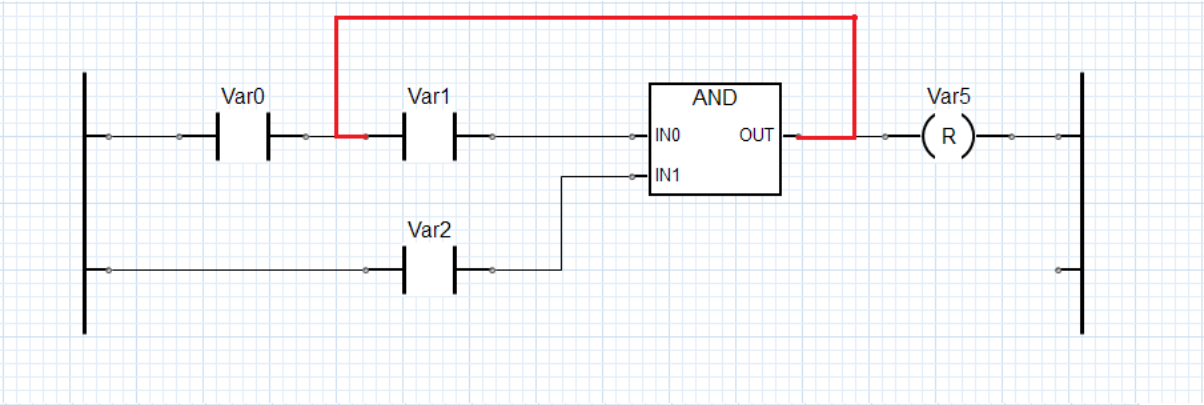


Fig. 67. – Tipo de ligação problemática que se quer evitar.

Neste caso, o funcionamento do programa é óbvio para um observador. A entrada do Contact com a variável “Var1” será um OR lógico entre “Var0” e a saída da Function. No entanto, devido à forma como as ligações LD funcionam, a ativação do Contact “Var0” iria levar um sinal elétrico para a Coil “Var5” e para o *Output* da *Function*. Este funcionamento iria confundir o compilador e iria possivelmente causar um comportamento indevido. O aspeto incorreto das ligações é uma maneira indireta de levar o utilizador a fazer a construção correta do seu diagrama, como mostra a figura seguinte (Fig. 68.):

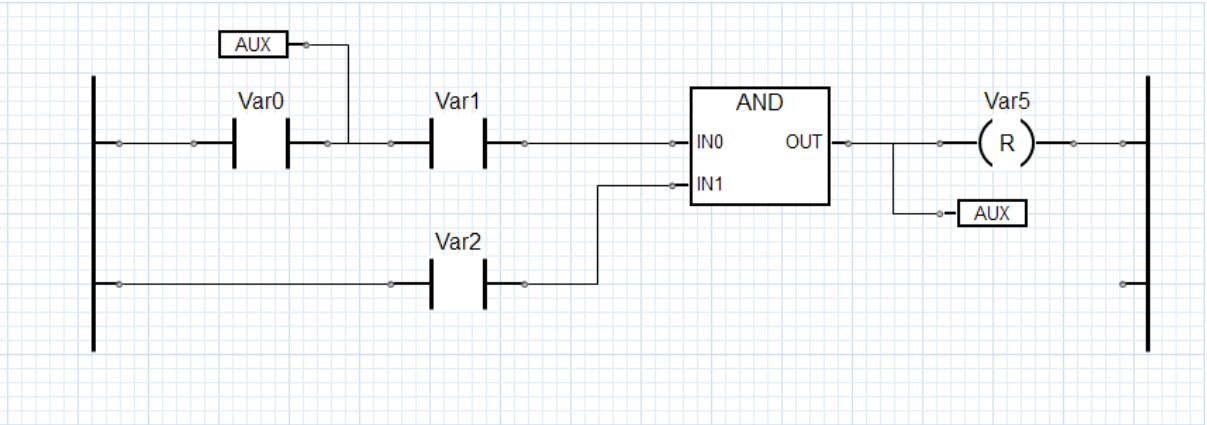


Fig. 68. – Exemplo de como devem ser feitas ligações transversais.

Desta forma o utilizador evita os erros referidos. Conseguindo também um aspeto mais correto do seu diagrama.

4.2.4 Tabela de testes

Como referido no início desta secção, a tabela seguinte contém uma lista de vários testes que foram realizados. O propósito desta tabela é expor o tipo de testes que foram feitos e confirmar quais as funcionalidades que foram devidamente testadas e que têm um funcionamento correto.

Tabela 2 – Tabela de testes realizados ao editor FBD

Funcionalidade testada	**Descrição**	**Resultados obtidos**	**Observações**
Criação de novo POU LD	Criação de novo POU com nome válido	Novo POU criado como esperado	Comportamento correto
Criação de novo POU LD	Criação de novo POU com nome inválido	Apresentada mensagem de erro, POU não foi criado	Comportamento correto
Criação de Coils/Contacts	Configuração e criação de novos elementos com configuração válida. Todos os tipos de Coils/Contacts foram testados	Novos elemento criado corretamente	Comportamento correto
Criação de Coils/Contacts	Configuração e criação de novo elemento com uma configuração inválida (variável com mais que uma palavra)	Mensagem de erro apresentada, diálogo de configuração novamente apresentado	Comportamento correto
Criação de Coils/Contacts	Configuração e criação de novo elemento sem variável definida	Novo elemento criado corretamente	Apesar de não parecer correto, este funcionamento é intencional para permitir ao utilizador criar vários objetos e só configurar depois.
Criação de Coils/Contacts	Cancelamento de criação através do diálogo de configuração	Novo objeto não é criado	Comportamento correto
Criação de Coils/Contacts	Criação de novo elemento após cancelar no teste anterior	Novo elemento criado corretamente	Comportamento correto
Criação de PowerRails	Criação de novos PowerRails com números de pinos variáveis	Novos elementos criados corretamente	Comportamento correto
Criação de PowerRails	Cancelamento de criação através de diálogos de configuração	Novos elementos não são criados	Comportamento correto
Criação de PowerRails	Criação de novos elementos após cancelar no teste anterior	Novos elementos corretamente criados	Comportamento correto
Conexões	Criação de conexões válidas	Novas conexões criadas corretamente	Comportamento correto
Conexões	Criação de conexões inválidas, entre input-input e output-output	Novas conexões não criadas	Comportamento correto
Testes ao MoveElement	Movimentação de vários objetos, com todos os tipos de conexões presentes. De forma que seja necessário mudar os tipos de conexão	Movimento corretamente feito, objeto de modelo tem as suas posições X e Y corretamente atualizadas e as conexões são devidamente atualizadas para os novos tipos	Comportamento correto
Testes ao MoveBendpoint	Movimentação de bendpoints, pertencentes	Conexões atualizadas corretamente, sendo os	Comportamento correto

	aos três diferentes tipos de ligação, dentro dos limites corretos	bendpoints devidamente alinhados e um aspeto de ligações mantendo o aspeto correto	
Testes ao MoveBendpoint	Movimentação de bendpoints, percentences aos três diferentes tipos de ligação, para vários locais indevidos que causariam aspetos incorretos se fossem permitidos	Conexões adaptadas o melhor possível dentro dos limites definidos, aspeto correto mantido	Comportamento correto
Testes às updateFeatures	Teste ao duplo clique e atualização de todos os tipos de objeto	Objetos tomam o aspeto correto após atualização, caso cancelada a atualização os objetos não são modificados	Comportamento correto
Testes à integração com editor FBD	Repetição dos testes realizados na tabela de testes para o editor FBD	Todos os testes correm como esperado	Comportamento correto
Criação de conexões entre objetos LD e FBD	Conexões de vários elementos LD com elementos FBD	Conexões obedecem a todas as restrições de input-output e tipos de variável	Comportamento correto

4.3 Conclusões

Ambos os editores implementados apresentam um funcionamento correto, com uma apresentação limpa e fácil de compreender. Houve muito cuidado para que o funcionamento dos editores fosse intuitivo e agradável de usar.

O editor FBD está funcionalmente completo e correto, sendo possível expandi-lo no futuro, com componentes novos tais como caixas de texto para comentários, capacidade de redimensionamento de componentes ou uma melhor implementação das conexões, que permita a adição e remoção de *bendpoints* e tenha menos restrições.

A certa altura será necessário implementar os vários *Function Blocks* e *Functions* definidos pela norma, através da modificação de ficheiros ".tvar". A forma pela qual as bibliotecas pré-feitas de *Function Blocks* e *Functions* deve ser alterada, possivelmente com um ficheiro .zip contendo todos os ficheiros ".tvar" que é exportado para o desktop ou para o local de instalação do Eclipse. Caso isto seja feito será necessário alterar os diálogos de criação de *Functions* e *Function Blocks* adequadamente.

Quanto ao editor LD, este encontra-se funcionalmente completo também. Existem alguns problemas relativos às ligações, mas estes problemas são maioritariamente inerentes ao *Graphiti* e só poderão ser resolvidos alterando o código do *Graphiti* em si. Até lá, será necessário que o compilador/exportador XML possa interpretar corretamente as conexões entre objetos.

4.4 Trabalho futuro

Para uma dissertação futura, o maior objetivo será a criação de exportadores/importadores XML, tanto para diagramas FBD como para diagramas LD. Para além dos exportadores e importadores, será necessário refazer o navegador IEC 61131-3, que tem ainda um funcionamento muito limitado. O maior problema com o navegador atualmente é a incapacidade de eliminar componentes (projetos, programs, functions e function blocks).

Ainda mais no futuro encontra-se a integração com um compilador e a configuração de tarefas, partes essenciais de qualquer IDE para programas na Norma IEC 61131-3.

Referências

[1] Filipe Ramos. Eclipse IDE for the programming languages of the standard IEC 61131. Dissertação de Mestrado, FEUP, 2014.

[2] José Ferreira. Ambiente integrado em Eclipse para desenvolvimento de programas IEC 61131-3. Dissertação de Mestrado, FEUP, 2015.

[3] Filipe Ribeiro. Ambiente de desenvolvimento integrado para programação IEC 61131-3. Dissertação de Mestrado, FEUP, 2016.

[4] Beremiz User Manual by LOLITECH disponível em http://www.beremiz.org/doc visitado a 31/01/2017

[5] Phillip Lipson, Geert vander Zalm, Bosch Rexroth Corp, “Inside Machines: PC versus PLC: Comparing control options”, 05 de Novembro de 2011 http://www.controleng.com/single-article/inside-machines-pc-versus-plc-comparing-control-options/9bf8690c6f23b11370bec90b52cb15c9.html Visitado a 25/06/2017

[6] PLCOpen, “Introduction into IEC 61131-3 Programming Languages”, disponível em http://www.plcopen.org/pages/tc1_standards/iec61131-3/. Visitado a 21/06/2017

[7] Karl-Heinz John, Michael Tiegelkamp, “IEC 61131 – 3: Programming Industrial Automation Systems”, Springer, 1995

[8] Gary Cernosek, “A brief History on Eclipse” 15 de Novembro de 2005, disponível em http://www.ibm.com/developerworks/rational/library/nov05/cernosek/ Visitado a 19/06/2017

[9] Várias informações e dados obtidos em https://eclipse.org/ Visitado a 19/06/2017

[10] Documentação e guias de utilizador do Eclipse disponíveis em http://help.eclipse.org/neon/ Visitado a 21/06/2017

[12] Documentação e FAQs relativos ao Isagraf, disponíveis em http://www.isagraf.com/index.htm?http://www.isagraf.com/pages/documentation/documentation_main.html Visitado a 02/02/2017

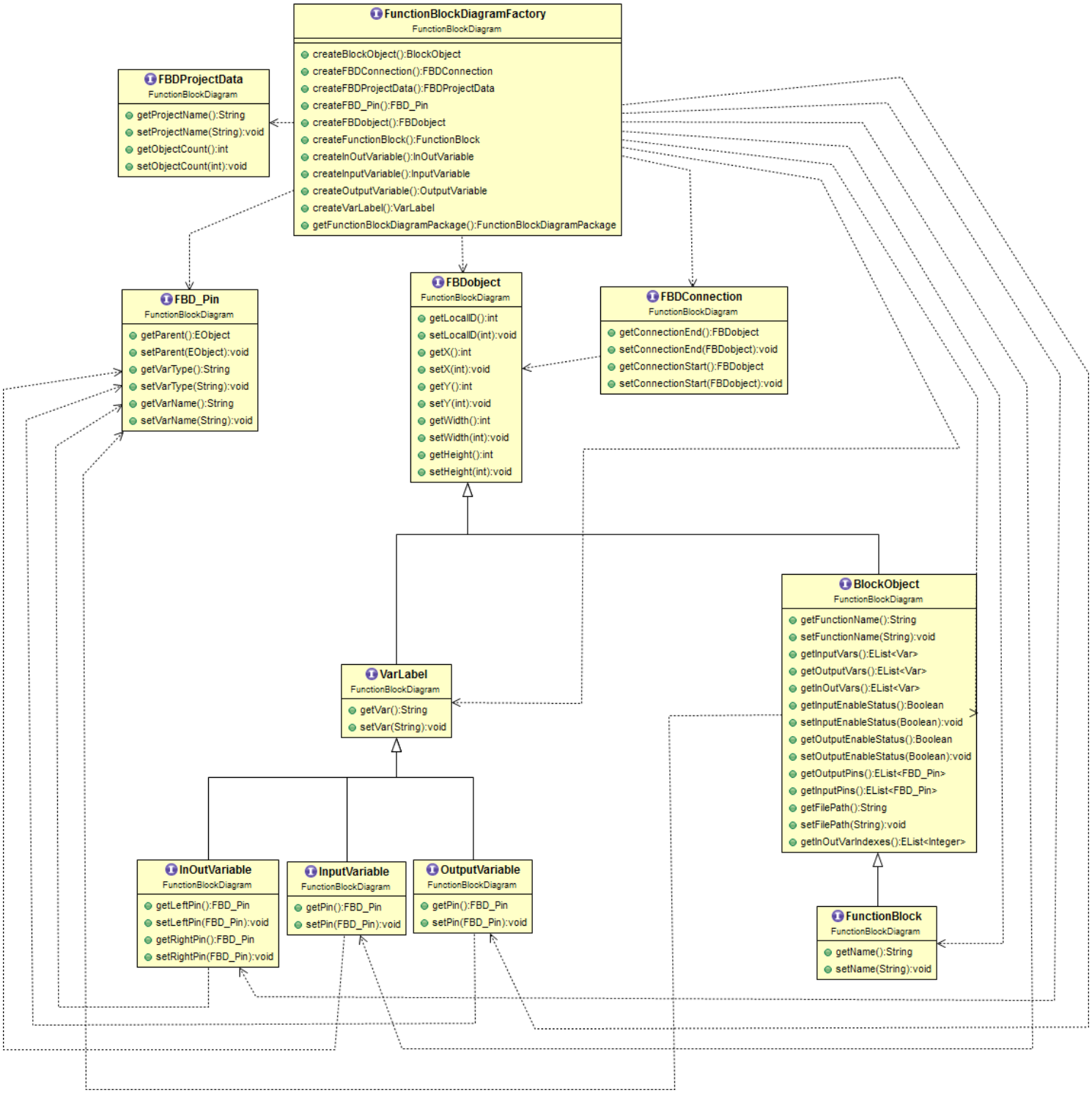
[13] Descrições do software Unity pro disponíveis em http://www.schneider-electric.com/en/product-range/548-unity-pro/166344562-software-description Visitado a 02/02/2017

[14] Descrição do software codesys, disponível em https://www.codesys.com/the-system/why-codesys.html Visitado a 02/02/2017

[15] “TC1 Standards” disponível em http://www.plcopen.org/pages/tc1_standards/ Visitado a 19/06/2017

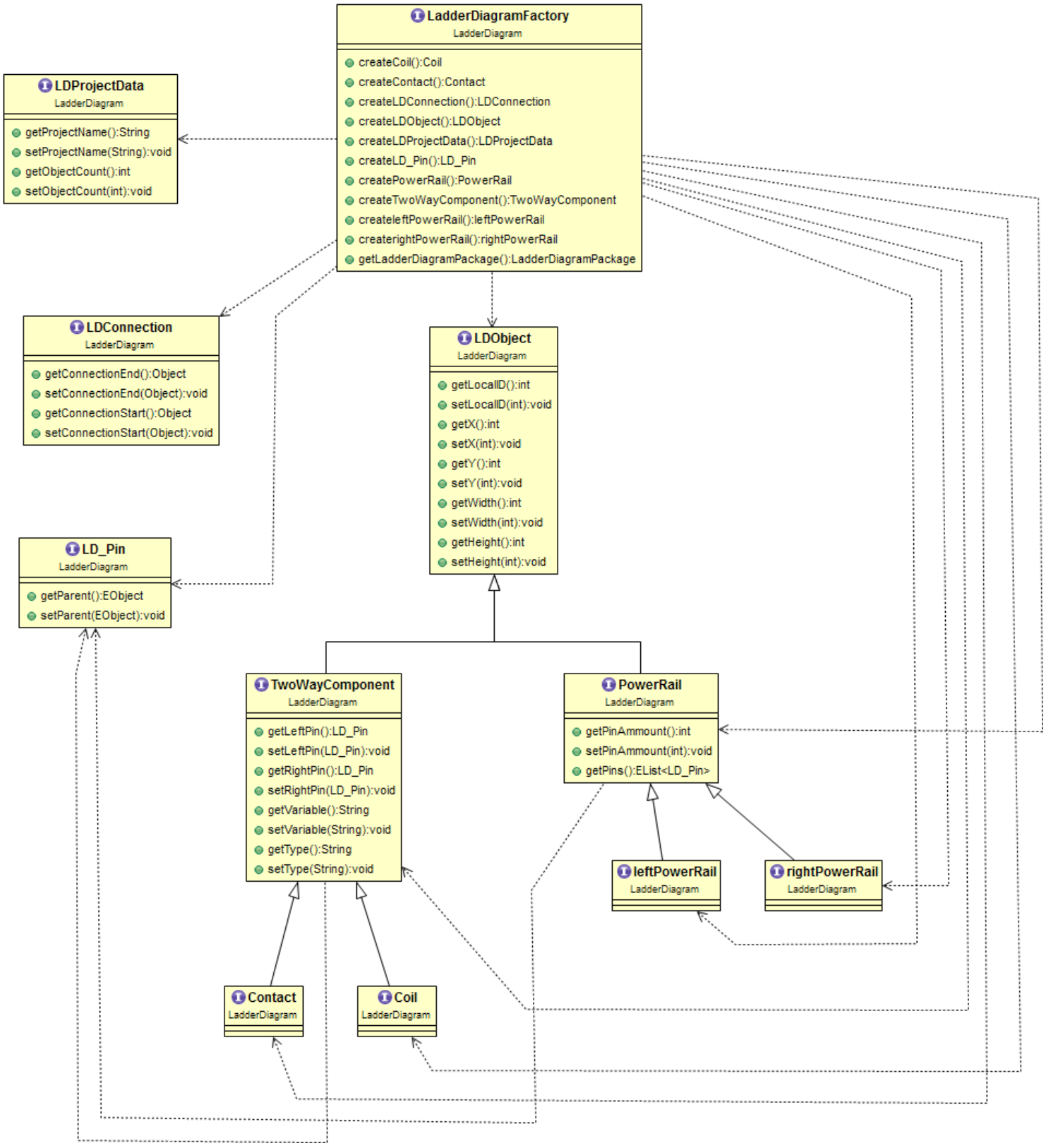
ANEXO A

Metamodelo ecore utilizado no editor FBD



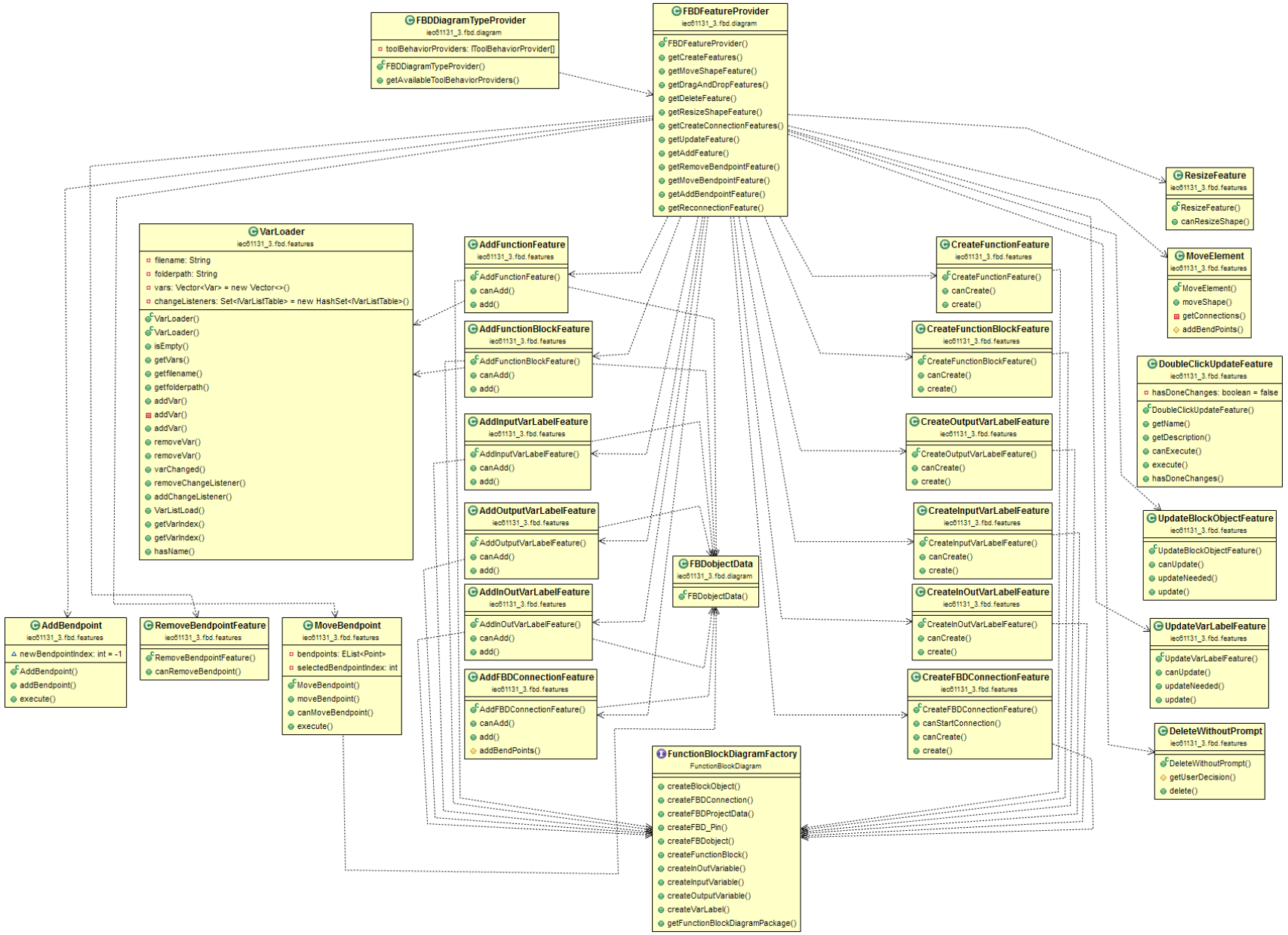
ANEXO B

Metamodelo ecore utilizado no editor LD



ANEXO C

Diagrama de classes relativo ao FeatureProvider do editor FBD



ANEXO D

Diagrama de classes relativo ao *FeatureProvider* do editor LD