



UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA



Escola Tècnica
Superior d'Enginyeria
Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Desarrollo de una aplicación gráfica para la edición de diagramas Grafcet y ladder

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Ludovic Jeckeln

Tutor: Francisco Rodríguez Ballester

2014 - 2015

Resumen

Un diagrama Grafcet es un modelo de representación gráfica, de los sucesivos comportamientos de un sistema lógico definido por sus entradas y salidas. Un esquema Ladder o diagrama de contactos es otra representación muy popular dentro de los autómatas programables debido a que está basado en los esquemas eléctricos de control clásicos.

Estos diagramas son utilizados para definir una lógica interpretada por los autómatas industriales. El usuario encargado de diseñar el diagrama requiere una aplicación que permita crear, editar y exportar esta estructura lógica.

El propósito de este proyecto es crear una aplicación gráfica con el lenguaje de programación deseado para la edición de diagramas Grafcet y Ladder con la principal característica de poder exportar los diagramas en fichero XML.

En este documento se introduce un nuevo software desarrollado con el lenguaje de programación C#, en el *framework* .NET y Visual Studio. Se explicará el análisis, diseño, interfaz y estructura del sistema, así como las herramientas utilizadas y detalles de implementación.

Palabras clave: Grafcet, Ladder, automatización, editor, C#, interfaz gráfica de usuario, WPF.

Abstract

A Grafcet diagram is a graphical representation of the successive behaviors of a defined logical system with inputs and outputs. A Ladder diagram or contacts diagram is another representation very popular for programmable automats thanks to a design based on the circuit diagrams of relay logic hardware.

These diagrams are used to define a logical behavior interpreted by industrial programmable logic controllers. The user in charge of designing this kind of diagram needs an application that allows to create, edit and export this logic structure.

The purpose of this project is to create a graphical application with the desired programming language to edit Grafcet and Ladder diagrams with the main characteristic of being able to export these diagrams to XML files.

This document will introduce a new software developed with the programming language C#, within the .NET and Visual Studio framework. We will explain the analysis, design, interface and structure of the system, as well as the tools used for development and some implementation details.

Keywords: Grafcet, Ladder, automation, editor C#, Graphical User Interface, WPF.



Tabla de contenidos

1.	Introducción	9
1.1	Automatización	9
1.1.1	Autómata industrial	9
1.1.2	Grafcet	10
1.1.3	Ladder.....	11
1.2	Motivación	13
1.3	Estructura de la memoria.....	14
2.	Análisis.....	15
2.1	Definiciones.....	15
2.2	Modelo del dominio	16
2.3	Especificación de requisitos.....	18
3.	Diseño	21
3.1	Antecedentes.....	21
3.1.1	SFCEDIT.....	21
3.1.2	FreeSFC (Diseño de gráficos SFC o GRAFCET)	22
3.1.3	Grafcet Designer - TecAtlant	22
3.1.4	Classic Ladder.....	23
3.1.5	SoapBox Snap	23
3.2	Herramientas utilizadas.....	24
3.3	Herramientas complementarias	27
3.3.1	GhostDoc	27
3.3.2	Sandcastle.....	28
3.3.3	Doxygen.....	29
3.4	Patrón de diseño	32
4.	Interfaz de la aplicación.....	35
4.1	Barra de herramientas	36
4.2	Área de diseño de diagramas	37
4.3	Editores de propiedades y opciones	38
5.	Estructura de la aplicación.....	41
5.1	Patrón Modelo – Vista - VistaModelo	41
5.2	Diagrama de Clases implementado	44
5.3	Serialización XML y reconstrucción del <i>DiagramData</i>	46



5.4 Flujo de la aplicación	48
6. Detalles de Implementación.....	51
6.1 WPF 2D <i>Graphics</i>	51
6.2 <i>ObservableCollection</i>	51
6.3 <i>DataTemplate</i>	52
6.4 <i>Grid Snapping</i>	53
6.5 Referencias cíclicas y Serialización XML.....	53
7. Conclusiones	55
Glosario.....	57
Bibliografía	59
Libros	59
Referencias de Fuentes Electrónicas	60
Referencias generales	61
Repositorio del proyecto	61
Anexo – Manual de Usuario.....	63

Tabla de figuras

Figura 1 Estructura de un autómata.....	9
Figura 2 Ejemplo básico de un diagrama Grafcet	10
Figura 3 Contactos Ladder	11
Figura 4 Bobinas Ladder	12
Figura 5 LADDER para la función $M = A(B'+C)D'$	12
Figura 6 Elementos de Memoria en Ladder	12
Figura 7 SFCEDIT.....	21
Figura 8 FreeSFC.....	22
Figura 9 Grafcet Designer - TecAtlant.....	22
Figura 10 Classic Ladder	23
Figura 11 SoapBox Snap	24
Figura 12 Microsoft .NET	24
Figura 13 Visual Studio 2013.....	25
Figura 14 Interfaz del IDE Visual Studio 2013.....	26
Figura 15 GhostDoc	27
Figura 16 Ejemplo GhostDoc.....	27
Figura 17 Sandcastle	28
Figura 18 Librería generada con Sandcastle	28
Figura 19 Doxygen	29
Figura 20 DoxyWizard - Project.....	30
Figura 21 DoxyWizard - Mode.....	30
Figura 22 DoxyWizard - Output.....	31
Figura 23 Doxygen - Librería HTML.....	31
Figura 24 Flujo del patrón MVVM	32
Figura 25 Ventana principal de la aplicación	35
Figura 26 Barra de herramientas	36
Figura 27 Ventana Principal - Ejemplo de diseño.....	37
Figura 28 Editor de opciones y editor de propiedades	38
Figura 29 Ventana Acerca de (About)	39
Figura 30 XAML de DataTemplate y ListBox para la clase Transition.....	41
Figura 31 Propiedad SelectedObject - Lógica del set	42
Figura 32 Ejemplo de binding del nombre y coordenadas	42
Figura 33 Clase Command	43
Figura 34 About Command	43
Figura 35 Asignación del DataContext.....	44
Figura 36 Serialización XML.....	46
Figura 37 Etiqueta XmlIgnore.....	46
Figura 38 Reconstrucción de las Connections	47
Figura 39 Reconstrucción de la lógica.....	48
Figura 40 ToggleButton Binding.....	48
Figura 41 ObservableCollection extendida.....	51
Figura 42 DataTemplate Triggers	52
Figura 43 Grid Snapping	53



1. Introducción

En este capítulo introductorio vamos a describir los conceptos básicos de la Automatización, comentar la motivación del proyecto y por ultimo describir la estructura de la memoria.

1.1 Automatización

1.1.1 Autómata industrial

Para poder controlar una cadena de producción, se necesita aparatos que monitorizan el conjunto de acciones que se tiene seguir paso a paso a lo largo del proceso industrial. Estos dispositivos se llaman autómatas. [1] Están conectados a una serie de sensores (entradas / *inputs*) y un conjunto de máquinas (salidas / *outputs*).

El autómata programable consta de una Unidad Central de Proceso (CPU), memoria, fuente de alimentación y su sistema a de Entrada/Salida. [2]

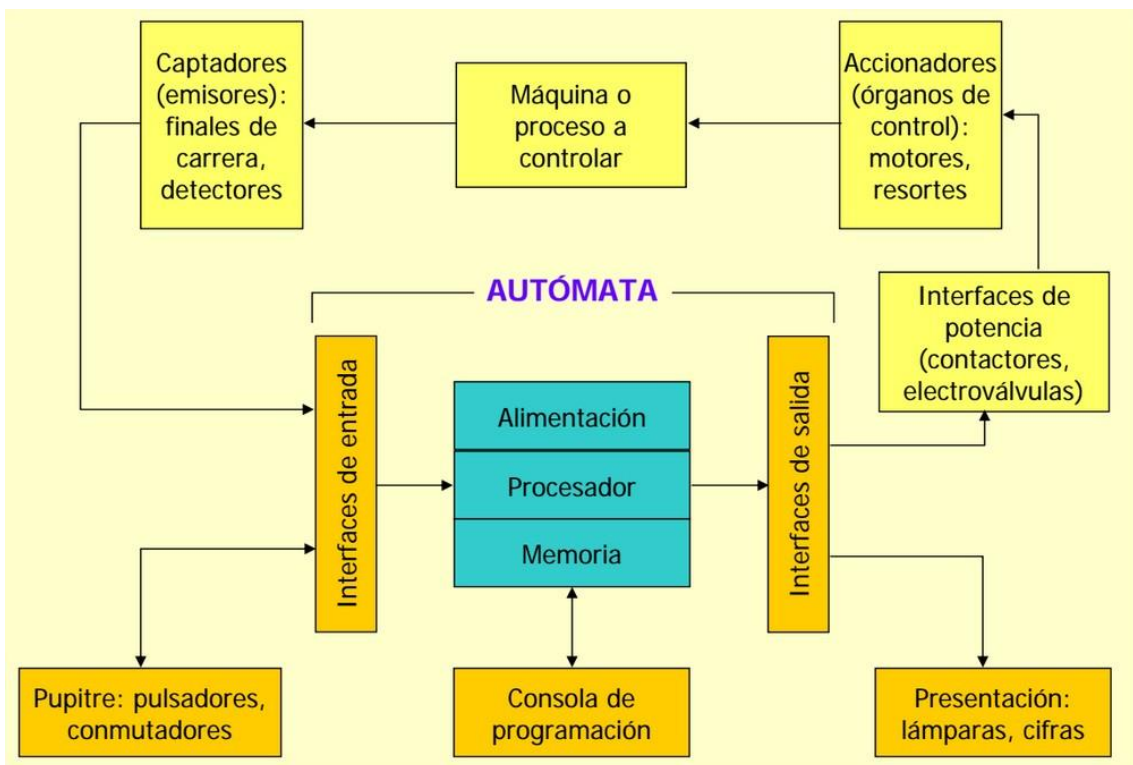


Figura 1 Estructura de un autómata

Vemos en la Figura 1 que los tres tipos de interacciones con el autómata son las Interfaces de entrada, Interfaces de salida y la Consola de programación. [3] Para poder programar el autómata, se le asigna una serie de instrucciones a través de la consola con un lenguaje de nivel bajo que tiene en cuenta la cantidad de entradas y salidas disponibles, así como los componentes internos como por ejemplo los contadores. [4]

Sin embargo el proceso de programación a este nivel es engorroso y se utiliza otras formas de representación para manipular y diseñar la lógica deseada. Para ello se utilizan los diagramas de automatización y control.

1.1.2 Grafcet

Para desarrollar la programación lógica de un autómeta se disponen de varios lenguajes de programación. Por una parte tenemos los lenguajes informáticos: bajo nivel (lista de instrucciones) y alto nivel (Basic, Pascal, C). Y por otra parte tenemos los lenguajes gráficos: diagramas de contactos o ladder, flujogramas, diagramas de funciones lógicas, Grafcet.

En la asignatura Control por Computador (CCO) impartida en la rama de especialización Ingeniería de Computadores se estudia el lenguaje Grafcet con el contenido docente y la bibliografía disponible. [4]

GRAF CET: Gráfico de Control de Etapas y Transiciones. Es un método gráfico de modelado y diseño de automatismos basado en las Redes de Petri, desarrollado por AFCET y ADEPA. Sus principales características son: no busca minimización de puertas lógicas ni memoria, metodología rigurosa, muy estructurado (claridad, legibilidad, sintético). [5]

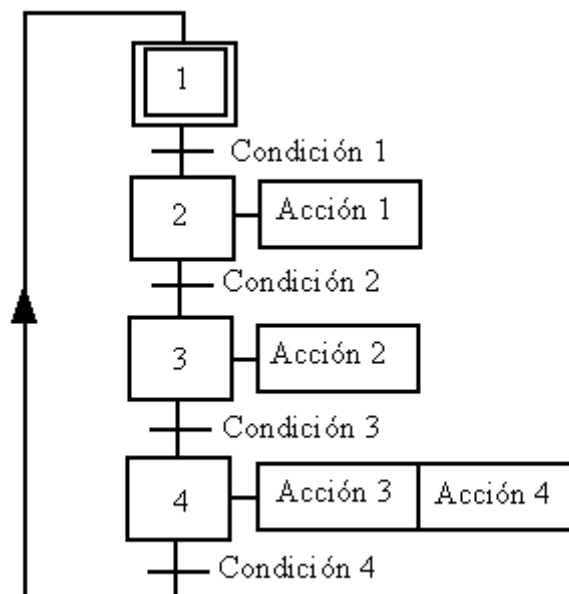


Figura 2 Ejemplo básico de un diagrama Grafcet

Vemos en la Figura 2 un Grafcet que dispone de 4 etapas, 4 transiciones, acciones y el cableado entre estos elementos, con además un enlace ascendente que vuelve a la etapa 1.

Etapa: Representa una situación del sistema donde todo o parte del órgano de control no varía respecto a las entradas/salidas del sistema automatizado. Una etapa con doble borde es una etapa de inicialización. En el proyecto el objeto asociado se denominará *Node*.

Acción: Se realiza mientras esté activa la etapa del sistema a la que se asocia. Pueden ser de dos tipos: nivel, impulso.

Transición: Barrera que asocia dos etapas consecutivas y cuyo franqueamiento hace posible la evolución del sistema. En la estructura de la aplicación el objeto se denominará *Transition*.

Receptividad: Condición asociada a cada transición

Arco: Porción de recta que une una transición con una etapa o una etapa con una transición. El objeto que representará un arco en el proyecto se denominará *Connection*.

[6] Las reglas de evolución del diagrama son:

- Activación incondicional de la(s) etapa(s) inicial(es).
- Transiciones se han de disparar en el instante en que son disparables y si hay múltiples han de hacerlo simultáneamente
- Si una etapa se activa y desactiva simultáneamente ha de permanecer activada
- Una transición validada con su receptividad verdadera provoca la activación de las etapas posteriores y desactivación de las precedentes simultáneamente

1.1.3 Ladder

Para poder programar un autómata usando un diagrama de contactos, además de estar familiarizado con cada una de las reglas de los circuitos de conmutación, es necesario identificar cada uno de los elementos que tiene este lenguaje. A continuación en la Figura 3 y Figura 4 se describen de modo general los más comunes.

Contacto de cierre		Contacto establecido cuando el objeto bit que lo controla pasa al estado 1.
Contacto de apertura		Contacto establecido cuando el objeto bit que lo controla pasa al estado 0.

Figura 3 Contactos Ladder



Bobina directa		El objeto bit asociado toma el valor del resultado del área de prueba.
Bobina inversa		El objeto bit asociado toma el valor inverso del resultado del área de prueba.

Figura 4 Bobinas Ladder

En el modelo de diagrama Ladder cada bobina representa el resultado de una función y a que se conecta (área de prueba).

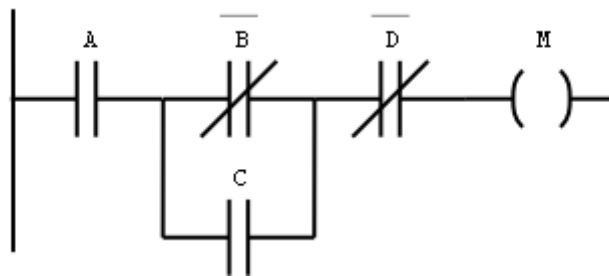


Figura 5 LADDER para la función $M = A(B'+C)D'$

En la Figura 5 vemos el ejemplo de una función booleana. Los contactos en paralelo se suman, y los demás se multiplican. Es una forma de representar expresiones booleanas de manera intuitiva con aspecto parecido a los circuitos eléctricos clásicos.

Otra particularidad de Ladder es la capacidad de disponer de una variedad de componentes como: contadores, temporizadores, monoestables y elementos de memoria.

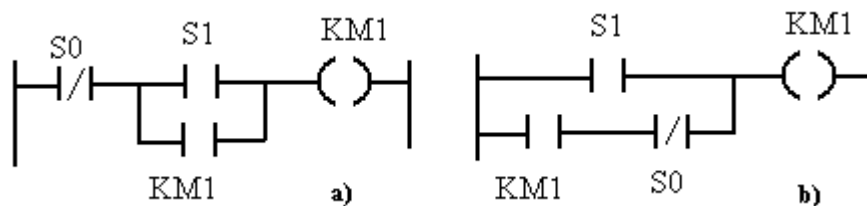


Figura 6 Elementos de Memoria en Ladder

En la Figura 6 podemos observar dos ejemplos de implementación de un elemento de memoria, donde tenemos a la vez un contacto KM1 y una bobina KM1. Esto implica que la función lógica depende de su estado anterior. Otra forma de implementar esta misma funcionalidad sería utilizando las bobinas Set y Reset que son respectivamente bobinas de desactivación y activación.

Combinando el uso de los lenguajes Grafset y ladder se puede diseñar y programar el comportamiento que debe tener un autómatas siguiendo el patrón parecido a una máquina de estados. De hecho para trazar ciertas funciones lógicas se utiliza funciones de Karnaugh. Toda esta lógica de automatización está muy relacionada con diversas asignaturas de la carrera y es un paso intermedio entre la Ingeniería Industrial y la Ingeniería Informática.

1.2 Motivación

A lo largo de la carrera hemos tenido la ocasión de abordar muchos aspectos de la lógica booleana, máquinas de estados y diagrama de flujos. Una asignatura que abarca la temática en particular es: Teoría de Autómatas y Lenguajes. Con esta asignatura me surgió un gran interés en la forma de representar comportamientos lógicos y automatizados con estados y transiciones. Me pareció fundamental poder representar de forma gráfica y esquemática la resolución de problemas lógicos de una manera intuitiva y fácil de manipular. Pero sin embargo el estudio de esta temática no daba ejemplos de aplicación en la industria.

Al elegir la rama de Ingeniería de Computadores, disponemos de la asignatura Control por Computador que abarca automatización industrial y control. La materia esta impartida por profesores del departamento de Ingeniería de Sistemas y Automática. Para mí fue un gran cambio con respecto a toda la docencia estudiada en el resto de la carrera y por primera vez entramos en el sector Industrial de las cadenas de producción, utilizando conocimientos previos de lógica, autómatas y matemáticas.

En la presentación de la asignatura se comenta hasta qué punto la lógica de control es importante y omnipresente alrededor nuestro, en hogares como en empresa. Hasta el robot de cocina más básico requiere una lógica de control que monitorice la velocidad de rotación del motor. Se desglosa también la cadena de producción, separando diversas etapas, acciones y procesos de verificación con sensores que llevan a cabo el proceso industrial. El poder analizar este proceso con una visión lógica y automática me pareció fascinante y por lo tanto considero que esta asignatura fue mi preferida de la carrera.

La necesidad de este tipo de Diagramas es esencial para un Ingeniero de Control y Automatización. Para poder utilizar estas representaciones es ideal disponer de una Interfaz Gráfica para el Usuario que permita: editar y crear diagramas fácilmente e intuitivamente, restringir el error humano, simular el comportamiento lógico deseado y exportar los esquemas a formatos interpretables por los autómatas industriales. Se puede evidentemente programar los autómatas de a bajo nivel con solo instrucciones, pero disponer de estos diagramas facilita y acelera el proceso de diseño e implementación de la lógica necesaria.



El interés por la temática ha sido la principal motivación para escoger este proyecto de oferta pública. Otra característica que me ha impulsado hacía este trabajo de fin de Grado ha sido poder disponer de libertad de elección del lenguaje de programación usado para su desarrollo. Es la oportunidad perfecta para poder descubrir un nuevo lenguaje y entorno de desarrollo que no ha sido estudiado a lo largo de la carrera. Por ello se ha escogido C# en el entorno .NET con el desafío de descubrir sus capacidades y su funcionamiento. Ha sido una dificultad añadida por no escoger el lenguaje principal enseñado a lo largo de la titulación (Java) pero es una experiencia nueva y muy interesante.

Además otra asignatura que me marcó y aprecié ha sido la Ingeniería Software, con desarrollo multicapas. Con lo cual el proyecto abarcaba tres elementos importantes de interés: Automatización, Ingeniería Software y un nuevo lenguaje de programación.

1.3 Estructura de la memoria

Este documento está compuesto por los siguientes capítulos:

- **Capítulo 2: Análisis.** En esta fase del desarrollo del software se estudia los conceptos necesarios, el modelo del dominio y la especificación de los requisitos.
- **Capítulo 3: Diseño.** En la fase de diseño se estudia diseños precedentes al proyecto, herramientas disponibles y alternativas.
- **Capítulo 4: Interfaz.** Este apartado consiste en describir los elementos de la interfaz: posicionamiento, habilitación y respuesta.
- **Capítulo 5: Estructura.** Se define la arquitectura del software, la organización e interacción de las diversas clases entre sí, la lógica y el flujo del programa.
- **Capítulo 6: Implementación.** Este capítulo comenta varios detalles y anécdotas relevantes de la implementación del proyecto.
- **Capítulo 7: Conclusiones.** Por último se describe las conclusiones que se han obtenido a través del trabajo de investigación, desarrollo y la toma de decisiones.

2. Análisis

En este capítulo veremos: las definiciones establecidas, el modelo del dominio y la especificación de requisitos del proyecto.

2.1 Definiciones

Para analizar el posible desarrollo del proyecto se han elegido ciertos términos para estructurar el sistema y la interacción de los elementos necesarios para representar los diagramas. A continuación se comentan las definiciones y conceptos del proyecto:

Connector: Línea recta que conecta dos puntos del diagrama. No tiene ninguna lógica determinada, simplemente representa una parte de un arco del diagrama. Sin embargo, existe el caso particular en el que un arco está compuesto de solo un *connector*.

Connection: Representa un arco entre dos elementos lógicos del diagrama. Está compuesta de varias líneas (*connectors*), generalmente horizontales o verticales. Una *connection* tiene un sentido lógico con inicio y destino donde un elemento se considera previo y el otro se interpreta como siguiente en el flujo del diagrama. La *connection* será el elemento principal para interpretar concatenación de elementos y la activación y desactivación de aquellos. No se distinguen gráficamente si son ascendentes o descendientes, sino que el usuario debe aplicar los arcos siempre desde una etapa anterior a una siguiente siempre siguiendo los pasos del manual de usuario para seguir un buen patrón de diseño y mantener la legibilidad del diagrama.

Node: Etapa lógica del diagrama Grafcet que representa un estado que puede estar o no activo e indica en qué posición y situación se encuentra el autómata. El *node* puede tener varias *connections*, aquellas que venga por la parte superior de la etapa representan los arcos que conecta la etapa a transiciones previas. Por otro lado, las

Transition: Transición del diagrama Grafcet que indica el requisito condicional para que una etapa activa pueda desactivarse y activar la o las etapas siguientes a la transición. Como para el *node* las *connections* de las etapas previas a la transición deberán llegar por su parte superior, mientras que las *connections* de las etapas siguientes saldrán de las transiciones por la parte inferior.

Input: Entrada del diagrama Ladder, simboliza un contacto que se establece cuando el objeto bit que lo controla está activado, es decir que el valor esta puesto a 1. La combinación de estos contactos se representa mediante arcos y se encuentra en la parte izquierda del diagrama de contactos. Cada contacto de cierre tiene arcos previos y siguientes. Los elementos previos se conectarán por la parte izquierda del *Input* y los siguientes por la parte derecha del mismo. La entrada puede estar negada, por lo que toma el valor binario opuesto a la señal asociada.

Output: Salida del diagrama Ladder que representa una bobina donde el objeto bit asociado a esta toma el valor resultado del área de prueba. Las bobinas se encuentran en la parte derecha del esquema Ladder y cada una redirige el resultado de una función lógica definida por la combinación de contactos esquematizada. Cada contacto de cierre tiene arcos previos y siguientes. Los elementos previos se conectarán por la parte izquierda del *Output*. La bobina puede estar negada y por lo tanto asignar un valor opuesto al resultado del circuito “combinacional” de contactos.

2.2 Modelo del dominio

En este proyecto en concreto, empezamos el desarrollo del proyecto desde su estado inicial para ofrecer un prototipo base para que otros alumnos puedan seguir expandiendo nuevas funcionalidades y componentes. Por esta razón tenemos la elección del lenguaje de programación y herramientas a utilizar a lo largo del desarrollo.

El modelo base de la aplicación consiste en disponer de un *grid* gráfico que permita: visualizar todo tipo de elementos lógicos que hemos definido previamente, poder cambiar sus coordenadas y desplazar cada elemento en cualquier momento, así como borrar los elementos.

En esta versión del prototipo, el buen uso de los elementos lógicos es a discreción del usuario, ya que no se dispone de intérprete o compilador interno de los diagramas Grafcet y Ladder. Establecer un intérprete lógico que verifique la coherencia y la validación de los diagramas sería una expansión propuesta a la hora de retomar el proyecto.

El diagrama Grafcet se compondrá de:

- *Nodes*
- *Transitions*
- *Connections*

El diagrama Ladder tendrá los siguientes componentes:

- *Inputs*
- *Outputs*
- *Connections*

En el caso de los diagrama Grafcet, a la hora de hacer el diseño gráfico, para los trazos paralelos, simplemente hemos decidido unir varios arcos a la misma transición.

Para poder guardar el estado de los diagramas de la aplicación se almacenarán los objetos en un fichero XML utilizando “serialización”. Por lo tanto se podrá importar o exportar usando el formato XML.

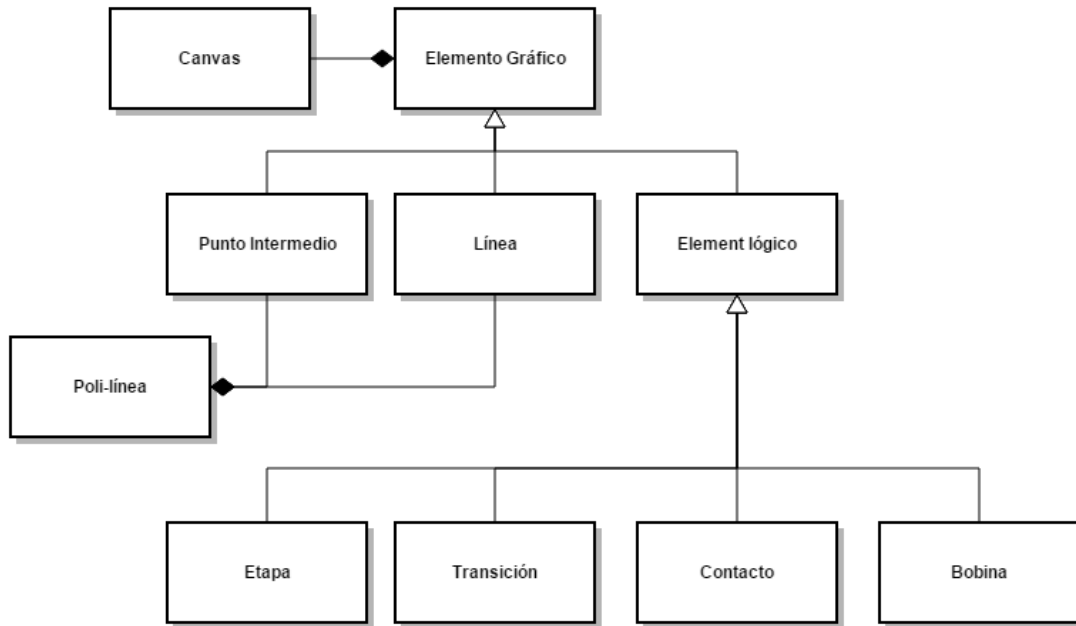


Diagrama 1 Diagrama de Clases

En el diagrama 1 podemos observar las clases y conceptos que hemos definido en la fase de análisis. Cada elemento gráfico tiene está compuesto por varios Canvas para ser dibujado. Las clases que heredan del Elemento Gráfico son: Punto Intermedio, Línea y Elemento lógico. La Poli-línea está compuesta de puntos intermedios y líneas. El elemento lógico se especializa como: Etapa, Transición, Contacto y Bobina.

2.3 Especificación de requisitos

A continuación se listan los requisitos del proyecto:

- Requisitos generales de la aplicación:
 1. Gestionar varios diagramas simultáneamente de tipo Grafcet y Ladder.
 2. Utilizar un formato compatible con ficheros XML.
 3. Poder conectar elementos a través de un arco, eligiendo su trayectoria
 4. Apuntar etiquetas lógicas básicas: elementos previos y siguientes.

- Requisitos básicos de comandos posibles:
 1. Añadir elementos al diagrama: Node, Transition, Connection, Input, Ouput, NotInput, NotOutput.
 2. Borrar elementos del diagrama.
 3. Conectar elementos entre sí.
 4. Desplazar / Reposicionar elementos cambiando sus coordenadas en el *grid* que re-dibuja cada elemento gráfico.
 5. Eliminar elementos y los arcos asociados al elemento.
 6. Empezar un nuevo proyecto, borrar el estado actual de la aplicación.
 7. Abrir un proyecto, des-serializando el archivo XML elegido por el usuario.
 8. Guardar proyecto, serializando todos los elementos lógicos y conexiones en formato XML.

- Requisitos del área de edición:
 1. Posicionar los elementos en coordenadas predefinidas
 2. Elegir la altura y anchura del área de edición
 3. Utilizar barras de *scroll* para acceder a más área de trabajo.
 4. Pre-visualizar el elemento nuevo que está en proceso de creación.
 5. Borrar los elementos que no se han validado después de su Pre-visualización.

3. Diseño

Este capítulo abordará los distintos *softwares* antecedentes al proyecto, las herramientas utilizadas y las complementarias y el patrón de diseño utilizado.

3.1 Antecedentes

Existen varias alternativas para adquirir un software propietario o libre que ofrezca los requisitos y funcionalidades de este proyecto: edición de los diagramas Grafcet y/o ladder que debe ejecutar el autómata. A continuación veremos varios ejemplos de estas aplicaciones utilizadas y presente en la industria así como en el ámbito académico o de código abierto.

3.1.1 SFCEDIT

SFCEDIT [7] es una herramienta que permite diseñar fácilmente diagramas Grafcet complejos para documentar automatización. Permite tener abiertas múltiples ventanas de diseño y un completo sistema de reconocimiento lógico y configuración de parámetros y estilos de visualización (Cambios de fuente, negrita, itálico, colores, color de impresión). Permite imprimir los esquemas y también ofrece integración con Microsoft Word, AutoCAD y Microsoft Visio. La aplicación permite producir Grafcets que siguen rigurosamente el estándar IEC 60848. En la Figura 7 podemos observar el interfaz gráfico de usuario.

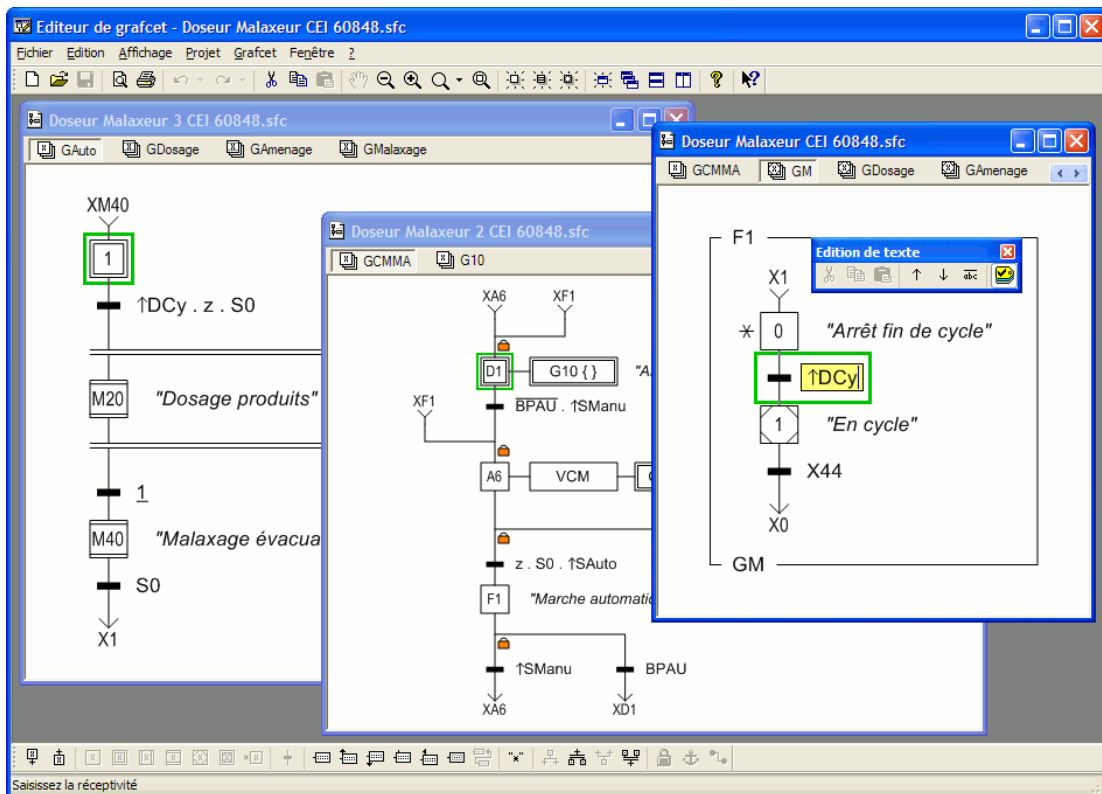


Figura 7 SFCEDIT



3.1.2 FreeSFC (Diseño de gráficos SFC o GRAFCET)

FreeSFC [8] es una extensión gratuita para instalar sobre el paquete OpenOffice que permite dibujar de una forma sencilla y rápida gráficos secuenciales SFC o GRAFCET. Este *add-on* instala una galería de símbolos que permiten representar esquemas SFC o GRAFCET desde OpenOffice Draw u OpenOffice Impress. En la Figura 8 tenemos una imagen de la interfaz de esta extensión.

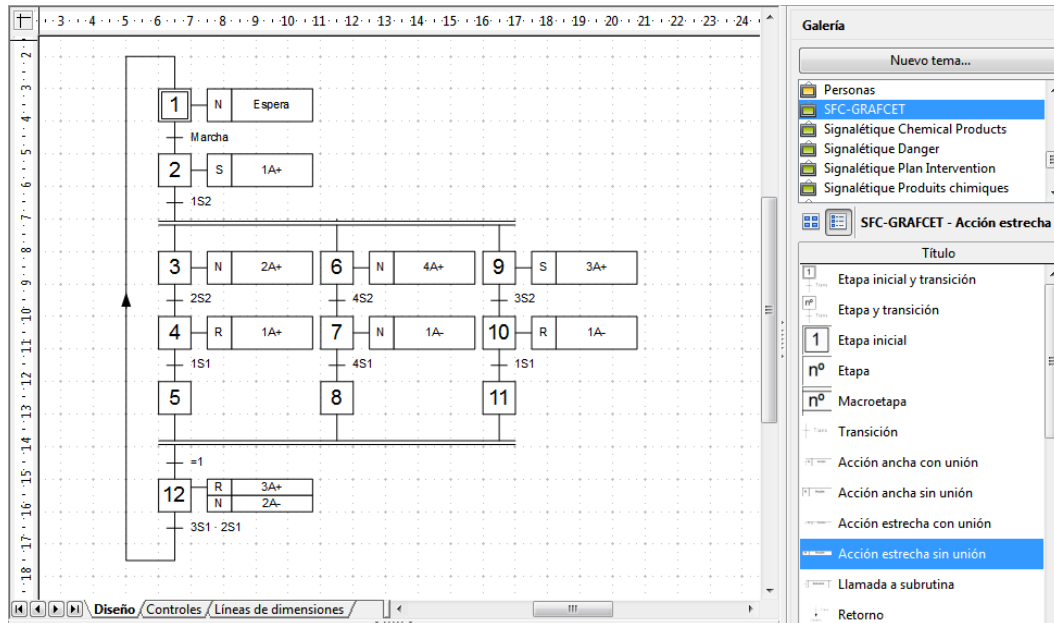


Figura 8 FreeSFC

3.1.3 Grafcet Designer - TecAtlant

La empresa National Instruments propone un software propietario llamado Grafcet Designer – TecAtlant. [9] Proponen una licencia con precio elevado de 850€ a fecha de este documento. [10] Es una *Add-on* del software LabVIEW y sus características son:

- Posibilidad de ver pasos activos para depuración más fácil
- Reducción del tiempo de desarrollo y validación
- Ejecución y simulación de aplicaciones de sistema de control
- Herramienta WYDWYG(*What you design is what you get*)



Figura 9 Grafcet Designer - TecAtlant

3.1.4 Classic Ladder

Classic Ladder [11] es un proyecto de código abierto que permite el diseño de diagramas de contactos, ejecutable en entorno Linux. El software ha sido desarrollado con el lenguaje de programación C. Está destinado a uso educacional. Tenemos un ejemplo de ejecución de esta aplicación en la Figura 10.

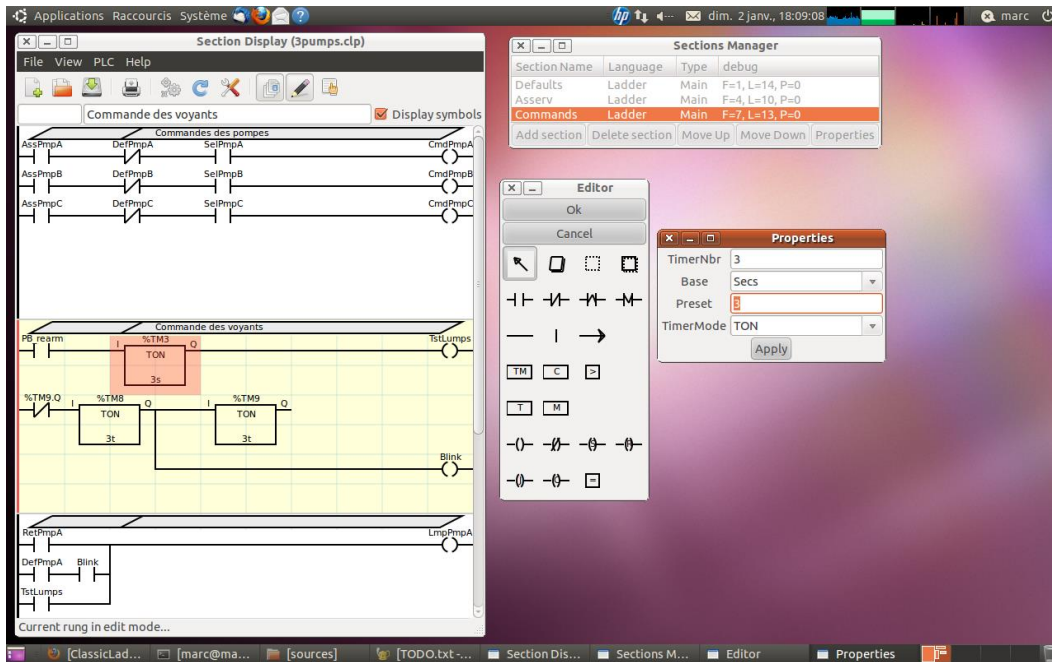


Figura 10 Classic Ladder

3.1.5 SoapBox Snap

SoapBox Snap [12] es una aplicación de código libre desarrollada en el entorno .NET. No está destinado a uso industrial sino a simulación y diseño de diagramas Ladder. Dispone de *drivers* dispositivos Phidgets de Entrada/Salida así como puertos USB. También dispone de un sistema de integración para Arduino. Se puede apreciar el diseño de la aplicación en la Figura 11.

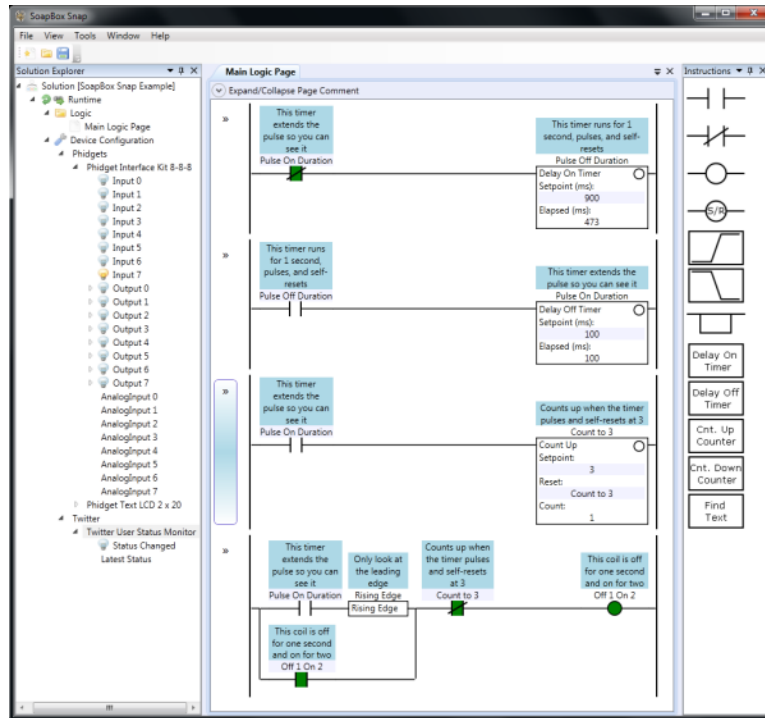


Figura 11 SoapBox Snap

3.2 Herramientas utilizadas

Como se ha comentado previamente en el apartado de motivación, se ha escogido el lenguaje de programación C# [13] para desarrollar el proyecto. En la carrera hemos aprendido distintos lenguajes: C, C++, Java, Haskell, Python, etc... Se precisó en la oferta del trabajo de fin de grado la elección libre del lenguaje y se ha decidido utilizar el lenguaje C# y el entorno de desarrollo .NET / Visual Studio.



Figura 12 Microsoft .NET

Viniendo con conocimientos de los lenguajes Java y C gracias a la docencia de la titulación, se puede familiarizarse bastante rápido al lenguaje C#: es un lenguaje de programación orientado a objetos desarrollado y estandarizado por Microsoft como parte de su plataforma .NET. [14]

Para desarrollar aplicaciones gráficas e interfaces de usuario, la plataforma .NET dispone de muchas herramientas de diseño a través de Visual Studio. El entorno de desarrollo para este proyecto ha sido la versión Express 2013 de Visual Studio. Aunque sea una versión básica y sin licencia no se ha notado a penas lagunas en su uso y su potencial. Las dos características que han faltado y cuya ausencia han impactado ligeramente el desarrollo del proyecto han sido: no disponer de manejador de excepciones visual para examinar excepciones internas, no poder incluir extensión de documentación al programa.



Figura 13 Visual Studio 2013

Al ser una tecnología desarrollada por Microsoft, la principal ventaja y desventaja de esta plataforma es su gran eficiencia y optimización en un sistema operativo Windows y una baja flexibilidad y compatibilidad a la hora de trasladarse en otro sistema. Entre los puntos fuertes que se han apreciado al descubrir y aprender esta plataforma de desarrollo, tenemos los siguientes:

- Coherencia de ejecución: No se puede modificar código durante ejecución de la instancia o durante una fase de *debugging*.
- Uso de *snippets*: Es muy intuitivo añadir atajos que permiten generar secciones de código que se necesitan a menudo: constructor, propiedades, manejador de eventos, etc.
- Extensa librería: Muchos recursos, clases y métodos disponibles por defecto y se pueden añadir muchos más cuando se añaden *assembly references*.
- Documentación intuitiva a través de código XML
- Posibilidad de visualizar arborescencia de objetos o excepciones durante la fase de *debugging*. Uso de regiones de código para (des)plegar secciones.
- Las palabras reservadas 'is' y 'as' son muy poderosas e intuitivas, más agradables y cortas comparado con los métodos de java.
- El uso de variable vacía y temporal con la palabra reservada 'var'.
- Un sistema IntelliSense muy eficaz (auto-completado/pre-visualización de la documentación de las clases, métodos y propiedades).
- Desarrollo más rápido en general y *debugging* completo con respecto a C#.

Desarrollo de una aplicación gráfica para la edición de diagramas Grafcet y ladder

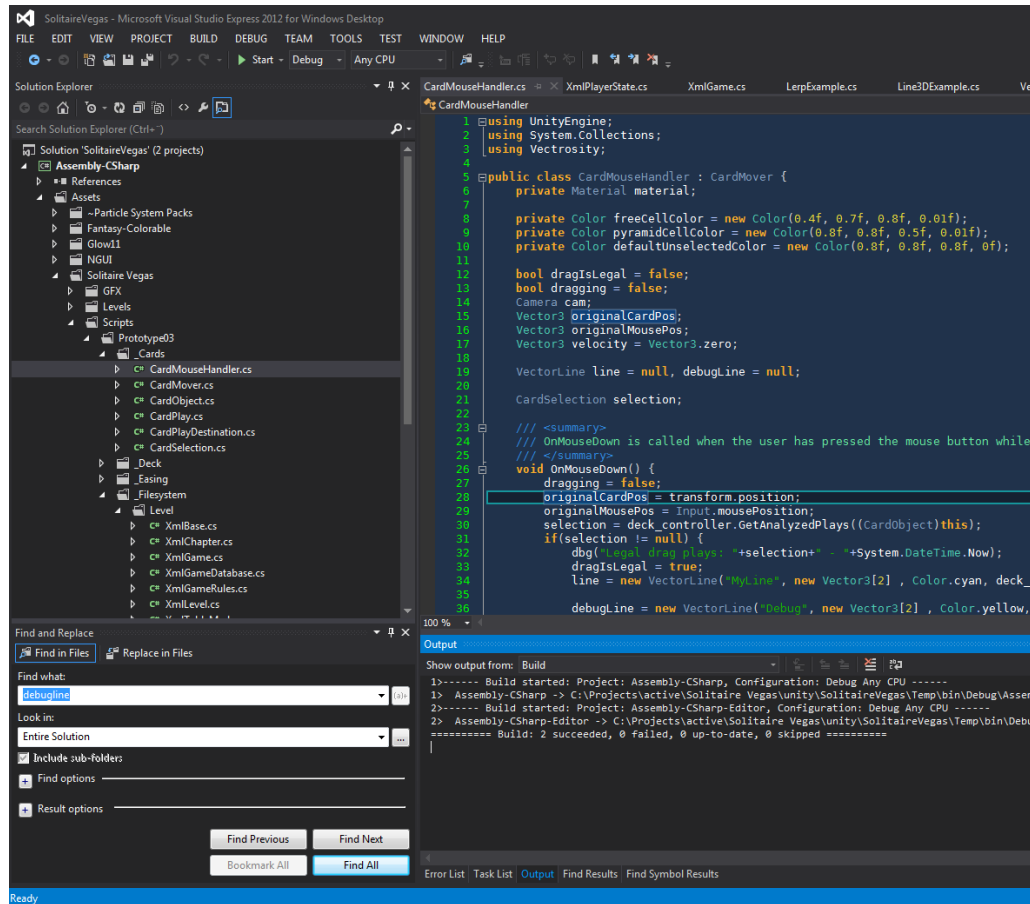


Figura 14 Interfaz del IDE Visual Studio 2013

Una vez usamos la plataforma .NET tenemos que tomar una nueva decisión ya que disponemos de dos tipos de GUI *frameworks*: WinForms y WPF (*Windows Presentation Foundation*). WinForms no tiene suficiente flexibilidad y está bastante limitado ya que sus elementos gráficos y controles no son completamente configurables. Con WPF disponemos de varias ventajas y herramientas muy útiles:

- Es más nuevo con respecto a WinForms y dispone de más controles
- Tiene mucha flexibilidad, una vez establecida la estructura se puede editar la interfaz con facilidad.
- Dispone del lenguaje XAML basado en XML que permite diseñar y separar el trabajo entre diseño y programación
- *Databinding* que permite separar datos de la interfaz y conectar dinámicamente las capas, encapsulando el acceso o escritura a datos desde la interfaz.
- Utiliza aceleración del hardware para re-dibujar la interfaz, con mayor rendimiento.
- Desarrollo de aplicaciones Windows así como aplicaciones web a través con Silverlight/XBAP

3.3 Herramientas complementarias

En este apartado vamos a comentar las herramientas que se han investigado y probado a lo largo del desarrollo de la documentación del código de cada una de las clases y sus respectivos métodos y propiedades.

3.3.1 GhostDoc

GhostDoc [15] [16] es una extensión disponible para Visual Studio que permite generar automáticamente comentarios de documentación XML para métodos y propiedades basándose en su tipo, parámetros, nombre, valor de retorno y otras informaciones contextuales.



Figura 15 GhostDoc

Sin embargo no se puede utilizar esta herramienta con una versión Express de Visual Studio. En la Figura 16 tenemos un ejemplo de la documentación que genera esta extensión, utilizando los parámetros y el valor de retorno.

```
/// <summary>
/// Calculates the tax.
/// </summary>
/// <param name="NetValue">The net value.</param>
/// <returns>System.Double.</returns>
public double CalculateTax(double NetValue)
{
    return NetValue * 1.18;
}
```

Figura 16 Ejemplo GhostDoc

3.3.2 Sandcastle

Sandcastle [17] produce documentación precisa y comprensiva utilizando el estilo de la librería MSDN. Este programa refleja las fuentes utilizadas por las *assemblies* y opcionalmente puede integrar comentarios de documentación en formato XML.



Figura 17 Sandcastle

Sin embargo el proyecto Sandcastle ya no está soportado por Microsoft desde junio 2010. Pero sigue actualizándose por una comunidad de desarrolladores en GitHub. [18]

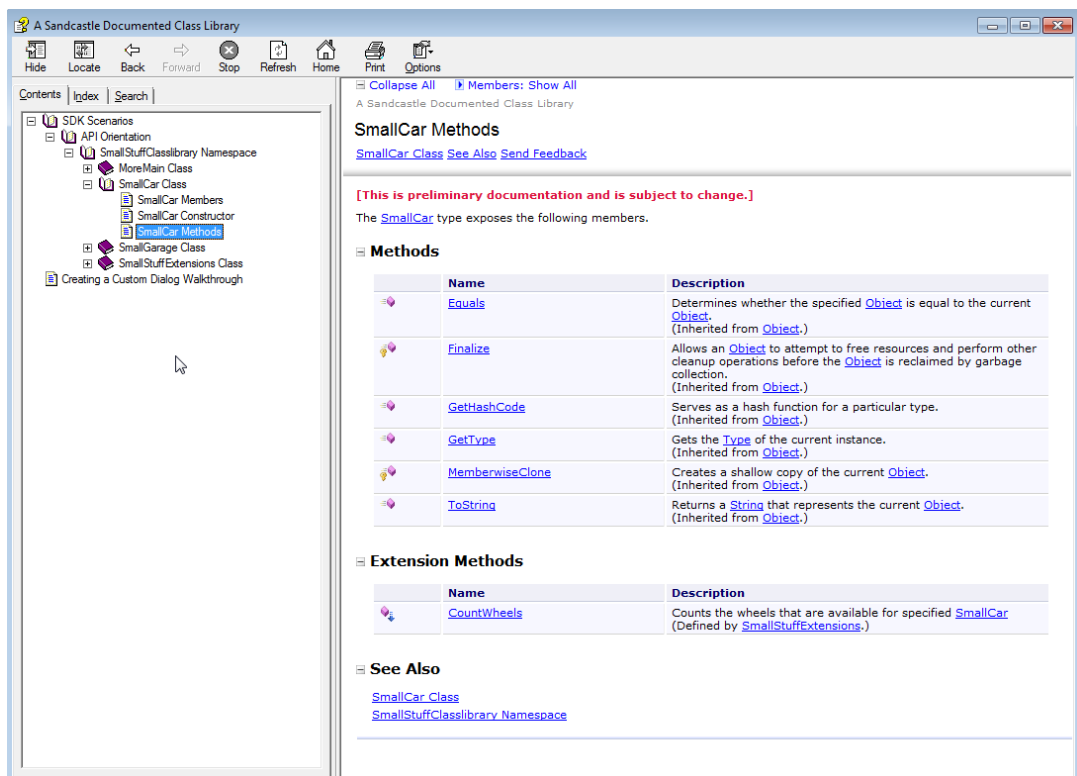


Figura 18 Librería generada con Sandcastle

En la Figura 18 tenemos un ejemplo de librería auto-generada por Sandcastle.

3.3.3 Doxygen

Doxygen es un generador de documentación para C++, C, Java, Objective-C, Python, IDL (versiones Corba y Microsoft), VHDL y en cierta medida para PHP, C# y D. Dado que es fácilmente adaptable, funciona en la mayoría de sistemas Unix así como en Windows y Mac OS X. La mayor parte del código de Doxygen está escrita por Dimitri van Heesch.



Figura 19 Doxygen

Doxygen es un acrónimo de dox (document) gen (generator), generador de documentación para código fuente.

1. Puede generar un navegador de documentación online (en HTML) y/o una manual de referencia offline (en LaTeX) a partir de un conjunto de códigos fuente. También tiene soporte para generar datos en formato RTF (MS-Word), PostScript, PDF con hiper-enlaces, HTML comprimido y páginas de manual Unix. La documentación se extrae directamente desde los archivos fuentes por lo que facilita mantener la consistencia con el código.
2. Se puede configurar Doxygen para extraer la estructura del código de ficheros fuentes no documentados. Esto es muy práctico para guiarse rápidamente en un proyecto de gran tamaño con muchas clases y fuentes. Doxygen también puede visualizar las relaciones que tienen varios elementos entre si y generar automáticamente grafos de dependencias, diagramas en función de la herencia o colaboración.
3. También se puede utilizar Doxygen para crear documentación normal.

Se ha decidido utilizar la herramienta Doxygen para generar la documentación de este proyecto debido a su gran flexibilidad y facilidad de uso. En las Figuras a continuación vamos mostrar cómo se ha empleado Doxygen para generar una documentación HTML y los ficheros LaTeX. En la Figura 20 vemos que tenemos elegir un nombre para el proyecto, sinopsis, número de versión, directorio de los códigos fuentes y el directorio de destino para la documentación. Paso seguido, como figura en la siguiente Figura 21, elegimos para que tipo de lenguaje queremos optimizar el proceso de documentación y elegimos C#. Y por último en la Figura 22 elegimos el tipo de formato que deseamos para la documentación, en nuestro caso HTML y LaTeX.

Desarrollo de una aplicación gráfica para la edición de diagramas Grafcet y ladder

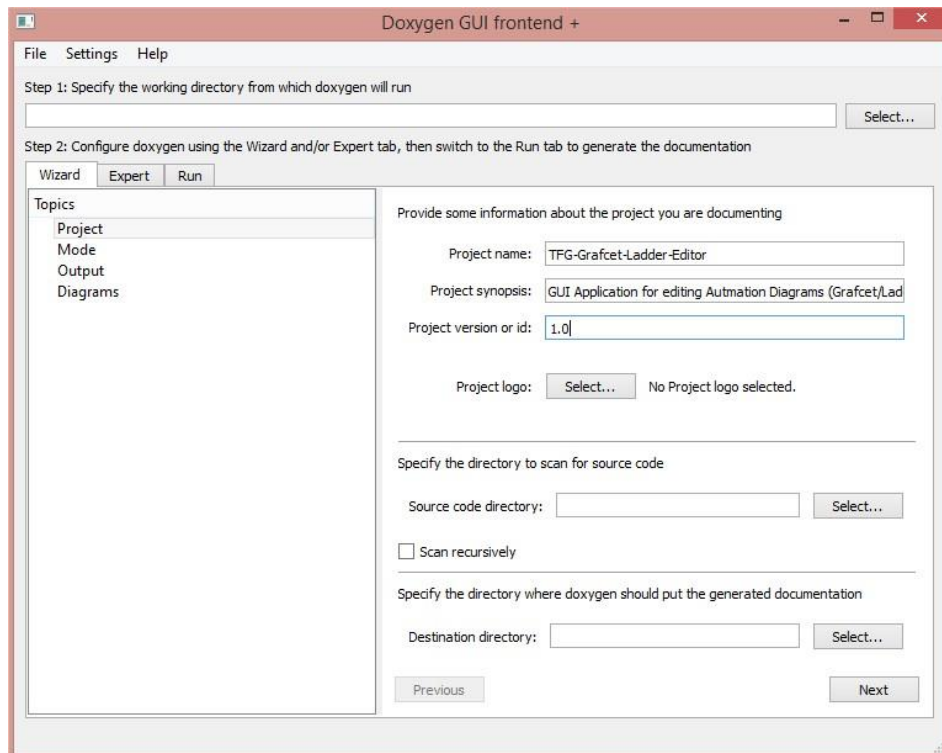


Figura 20 DoxyWizard - Project

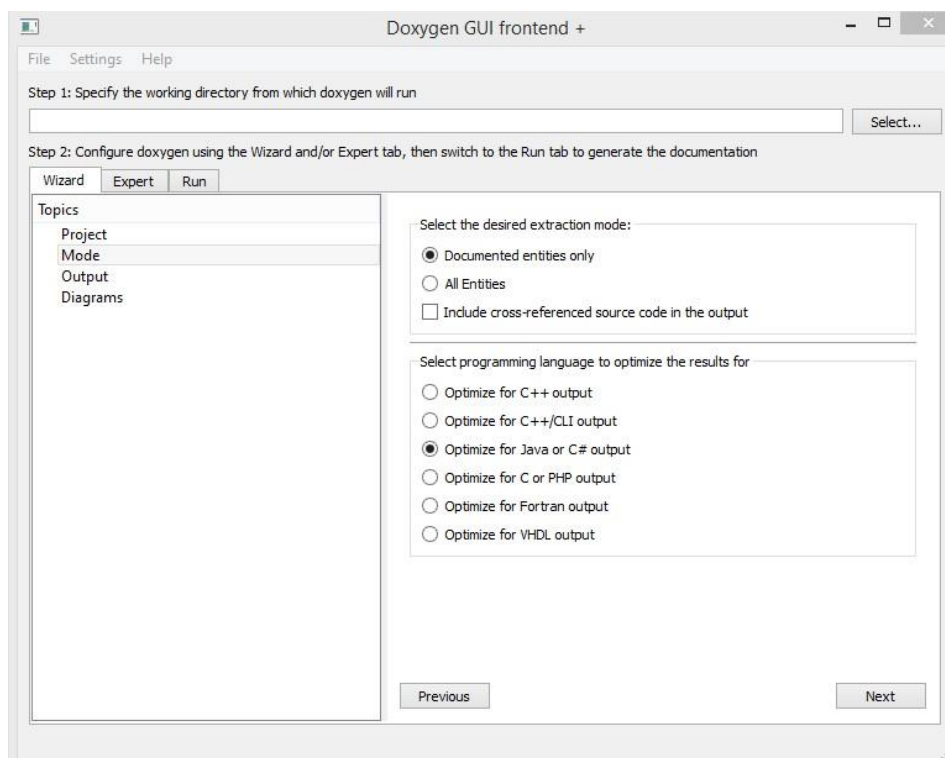


Figura 21 DoxyWizard - Mode

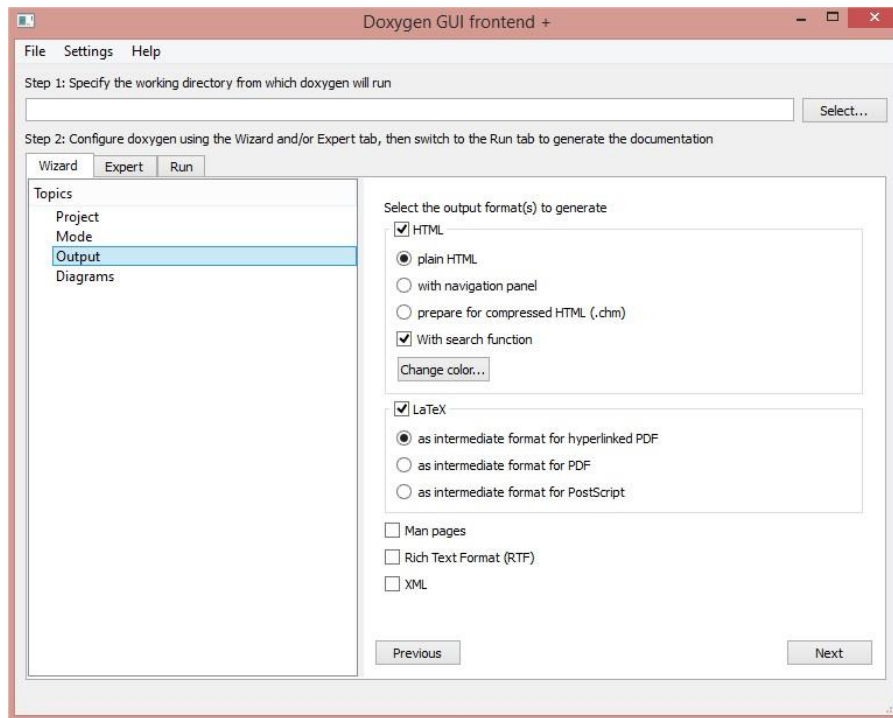


Figura 22 DoxyWizard - Output

Una vez generado los documentos, podemos ver en la Figura 23 el resultado del fichero HTML que agrupa toda la documentación escrita con etiquetas XML en Visual Studio, presentada bajo la forma de una API clásica. La documentación está disponible en GitHub: <https://github.com/cloporte/TFG-Grafcet-Ladder-Editor>

TFG-Grafcet-Ladder-Editor 1.0

Graphic editor for Automation Diagrams (GRAF CET/LADDER)

Main Page Packages **Classes** Search

Class List Class Index Class Hierarchy Class Members

Class List

Here are the classes, structs, unions and interfaces with brief descriptions: [detail level 1 2]

PrototipoTFG	
C About	Interaction logic for About.xaml
C App	Interaction logic for App.xaml
C Command	Simple implementation of ICommand Serves as an abstraction of Actions performed by the user via interaction with the UI (for instance, Button Click)
C Connection	A Connection is a group of Connectors and InterNodes. These represent a connection between two Logic Elements by using multiple lines and intermediate points. The Start and End of the Connection are references to LogicElement. The logic implies that the Starting element is followed by the Ending Element, which means the Start considers the End as a Next element, whereas the End considers the Start as a Previous element.
C Connector	Each connector represents each line drawn between Diagram Objects. The Connector's X and Y properties are always 0 because the line coordinates are actually determined, by the Start.X, Start.Y and End.X, End.Y Nodes' properties.
C DiagramData	DiagramData is a group of ObservableCollections of all the Elements a Diagram has : Nodes, Transitions, InterNodes, Inputs, Outputs, NotInputs, NotOutputs, Connectors and Connections. It is all the data needed to save or load a Diagram.
C DiagramObject	DiagramObject is the basic class that represents a graphical element that might be displayed on the Diagram Area A DiagramObject has a name, can be a preview element, can be highlighted and also has coordinates.
C InputLadder	An Input is part of the Ladder Diagram, represents an entry for a signal.
C InterNode	An InterNode is an intermediate point that are between the Connectors of a Connection Diagram.
C LogicElement	A LogicElement is a DiagramObject with logic references as Previous and Next.
C MainViewModel	MainViewModel is the main Business Logic Layer
C MainWindow	Interaction logic for MainWindow.xaml
C Node	A Node is a Logical Step or State into the GRAFCET Diagram. To include a compiler / simulator version this Object would need a 'IsActive' property to know which Nodes are active during an interpretation and launch of the Diagram.
C NotInputLadder	An NotInput is a negated version of the Ladder Diagram's Input, represents an entry for a signal with a 'not' applied to it.
C NotOutputLadder	An NotInput is a negated version of the Ladder Diagram's Output, represents an entry for a signal with a 'not' applied to it.
C OutputLadder	An Output is part of the Ladder Diagram, represents an exit for a signal, connecting the result of a logical expression to a component
C Transition	A Transition is an element part of the GRAFCET Diagram. It represents the logical condition between different nodes that need to fulfilled in order to continue Step by Step the execution of the Diagram.

Generated by 1.8.10

Figura 23 Doxygen - Librería HTML



3.4 Patrón de diseño

Model-View-ViewModel (MVVM) es el patrón de diseño de aplicaciones más popular y usado para aplicaciones WPF. Se ha escogido este patrón para desacoplar código de interfaz de usuario y código que no sea de interfaz de usuario. Utilizando MVVM, se puede definir una interfaz de usuario de forma declarativa mediante XAML. También se usa *databinding* para vincular la UI a otras capas que contengan datos y/o comandos de usuario. [19] La infraestructura del *databinding* permite un acoplamiento débil que sincroniza la interfaz de usuario y los datos vinculados. Todas las entradas de usuario se redirigen entonces a los comandos asociados.

El patrón MVVM organiza el código de tal manera que se puede cambiar partes individuales sin que los cambios alteren a las demás partes. Las ventajas del patrón que son más relevantes son:

- Estilo de codificación exploratorio e iterativo.
- Simplifica el *Unit Testing*.
- Herramientas de diseño como Expression Blend.
- Facilita la colaboración en equipo, desglosando actividades de diseño e implementación.

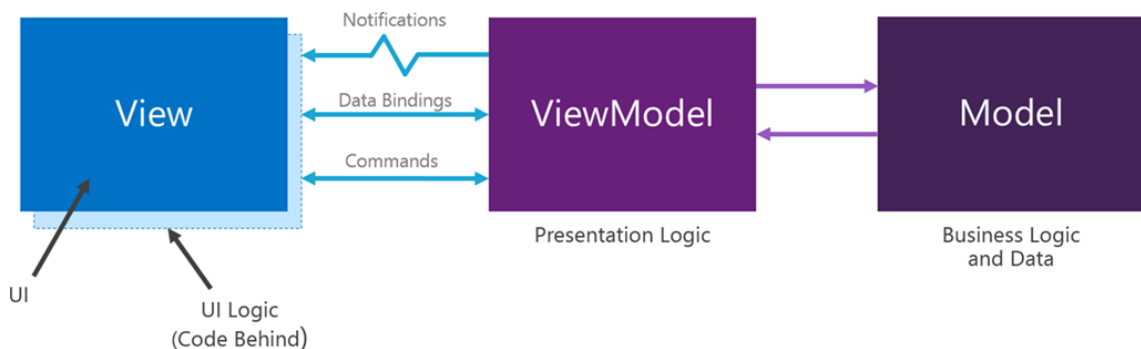


Figura 24 Flujo del patrón MVVM

Como se ve en la Figura 24, el patrón se compone de tres capas:

- **Capa de modelo (*Model*):** contiene todo el código que implementa la lógica principal del sistema, definiendo los tipos necesarios para modelar el dominio del proyecto. La capa de modelo es totalmente independiente de las otras capas.
- **Capa de vista (*View*):** engloba la interfaz de usuario que se define con marcado declarativo (XAML). El *databinding* define la conexión entre los distintos componentes específicos de la interfaz y varios miembros del *ViewModel* (o incluso miembros de la capa modelo).
- **Capa de modelo de vista (*ViewModel*):** proporciona destinos de Databinding para la capa de vista. El *ViewModel* expone el modelo de forma directa o da acceso a miembros que a su vez encapsulan miembros de modelo. El *ViewModel* también puede definir miembros para realizar un seguimiento de los datos que son relevantes para la interfaz de usuario pero no para el modelo, como el orden de visualización de una lista de elementos.

Una de las ventajas que tenemos al usar la separación por capas es que nos va permitir facilitar la comprensión del código. Esto es gracias a que el código específico de ciertas características es independiente del resto del código y eso hace el aprendizaje más intuitivo (podemos reutilizar el código en otras aplicaciones).
[20]

4. Interfaz de la aplicación

En este capítulo vamos a comentar la estructura de la interfaz gráfica de usuario. Trataremos de desglosar la GUI en tres apartados: barra de herramientas, área de diseño y editores de propiedades. En la Figura 25 podemos apreciar estos distintos *layouts*. En la parte superior tenemos la barra de herramientas, en la parte izquierda se encuentran los editores de propiedades e opciones y por último el área de diseño.

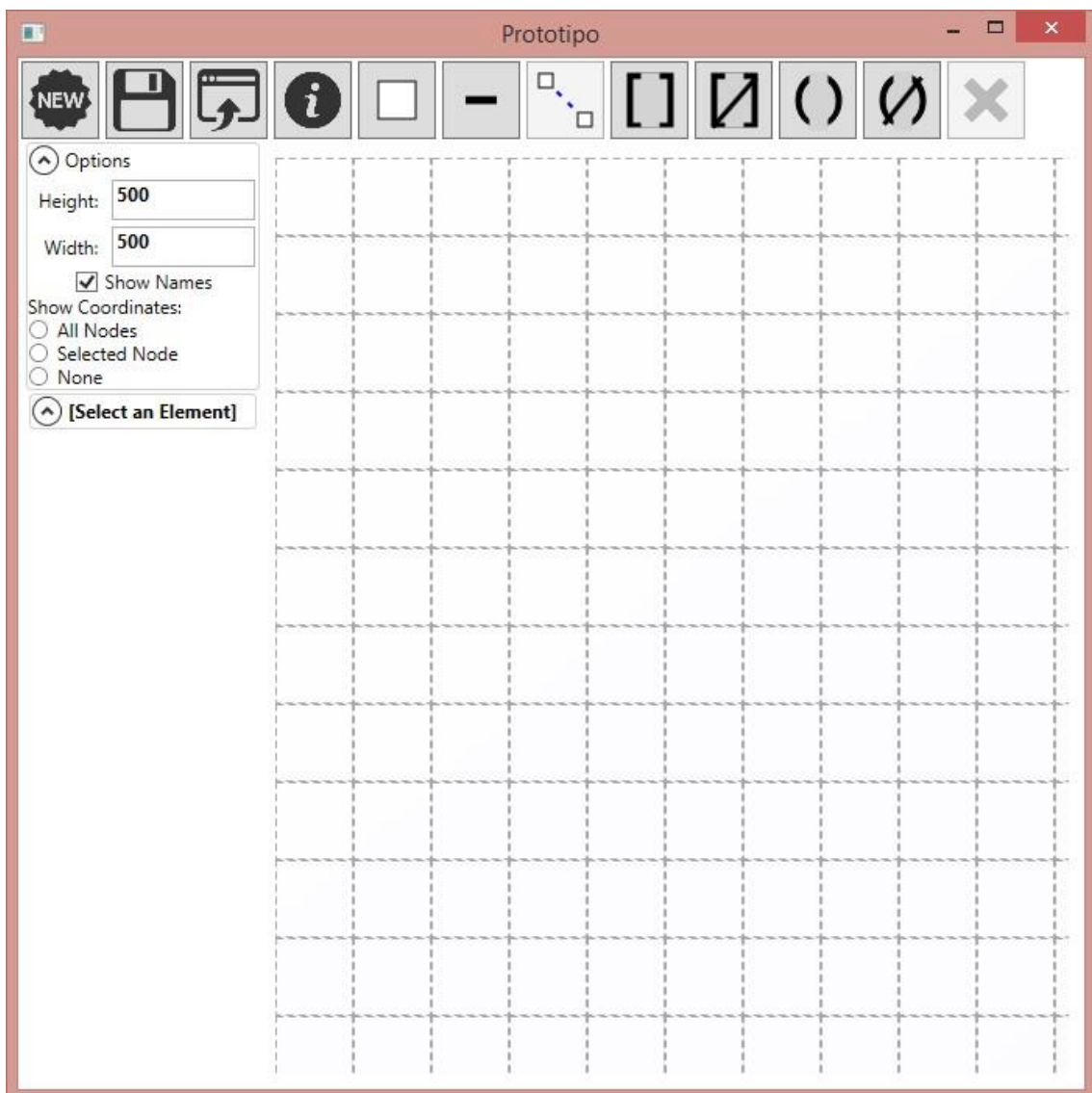


Figura 25 Ventana principal de la aplicación

4.1 Barra de herramientas

Vamos a comentar cada uno de los botones de la barra de herramientas de izquierda a derecha siguiendo la Figura 26. Cada botón dispone de un *Tooltip* cuando el cursor del ratón permanece encima de este.



Figura 26 Barra de herramientas

- **Nuevo (New):** Con este botón se borra y reinicia el estado del área de diseño, pero se comprueba si el usuario tiene elementos gráficos inicializados. Si el usuario tiene un diagrama empezado se abre un dialogo simple donde se pregunta al usuario si desea guardar el/los diagrama(s) actuales.
- **Guardar (Save):** Lanza la función de guardado en formato XML. Un dialogo aparece entonces para que el usuario decida de la ruta y el nombre del archivo que quiere guardar.
- **Abrir (Open):** De la misma manera que la función Nuevo, se comprueba si el usuario tiene trabajo empezado y si lo desea guardar. Una vez se ha resuelto ese aspecto, este botón abre un dialogo para abrir un fichero de formato XML para importarlo en el área de diseño.
- **Acerca de (About):** Abre un dialogo con información relativa al proyecto: autor, fecha, organización, enlace hacia la organización y enlace del manual de usuario.
- **Insertar etapa (Node):** Activa el modo de inserción de una nueva etapa en la zona de diagrama.
- **Insertar transición (Transition):** Permite insertar una transición en el área de diseño
- **Conectar (Connection):** Este botón esta deshabilitado por defecto y se habilita cuando el usuario selecciona un elemento gráfico. Una vez activado el modo de inserción de puntos intermedios del arco (*InterNode*).
- **Insertar contacto (Input):** Crea un nuevo contacto para que el usuario lo pueda posicionar en el lienzo.
- **Insertar contacto negado (Not Input):** Inserta una entrada negada en el diagrama.
- **Insertar bobina (Output):** Activa el modo de inserción de bobina en el área de trabajo.
- **Insertar bobina negada (Not Output):** Añade una bobina negada, que invierte el valor lógico de la función de la combinación de contactos.
- **Eliminar Elemento (Delete):** Se habilita este botón cuando tenemos seleccionado un elemento gráfico cualquiera y nos permite eliminarlo. Si el elemento tiene conexiones, estas también son eliminadas. También permite eliminar arcos.

4.2 Área de diseño de diagramas

El área de diseño es la parte más importante de la interfaz donde el usuario va a realizar el diseño de diagramas, donde se representan cada uno de los elementos gráficos de tipo Grafcet y/o Ladder.

Una rejilla demarca las coordenadas propuestas al usuario para ayudar a posicionar los elementos de forma coherente. Se reduce la libertad de posición para encajar los elementos y evitar error humano.

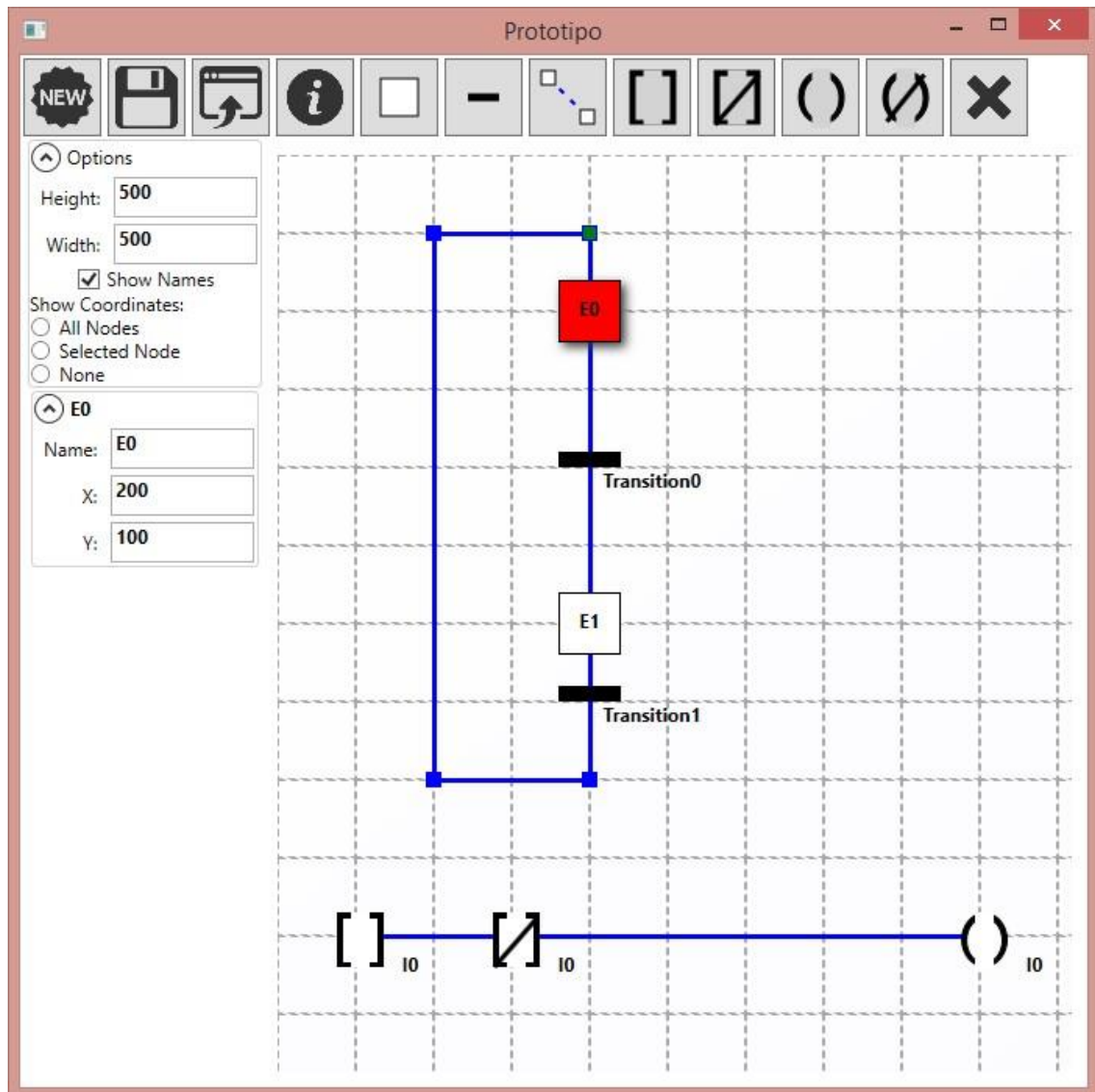


Figura 27 Ventana Principal - Ejemplo de diseño

En la Figura 27 podemos apreciar un ejemplo de diseño con varios elementos gráficos: etapas, transiciones, contactos y una bobina. Los arcos pueden tener puntos intermedios si es necesario, como por ejemplo el enlace ascendente representado en la figura que tiene 4 puntos intermedios seleccionables. Cada línea de un arco también es seleccionable.

4.3 Editores de propiedades y opciones

La última sección a comentar de la ventana principal son los editores. Tenemos en la parte superior de este apartado el editor de opciones como se ve en la Figura 28. Este editor permite seleccionar la altura y la anchura en pixeles del área deseada para obtener las barras de *scroll*.

Dispone de un *checkbox* que activa o desactiva la impresión de los nombres de los elementos gráficos en el diagrama. También se dispone de unos botones radio si se desea visualizar las coordenadas de los elementos.

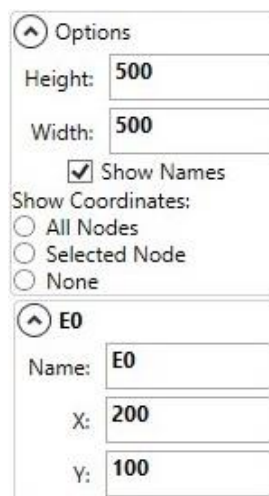


Figura 28 Editor de opciones y editor de propiedades

El segundo editor se encarga de mostrar las propiedades. En este caso se muestran las coordenadas del elemento gráfico seleccionado. En el caso de seleccionar una línea aparecen los campos de los dos elementos unidos por la línea. El usuario puede modificar directamente estos campos introduciendo nuevos valores.

En la Figura 29 tenemos la ventana de información “Acerca de”. En este dialogo se muestra una imagen de la Escola Tècnica Superior d’Enginyeria Informàtica, seguido unas etiquetas que describen el autor, la fecha de publicación, la universidad, enlaces hacía su web y manual de usuario. También tenemos en la parte inferior un botón para cerrar el dialogo de información.



Figura 29 Ventana Acerca de (About)

5. Estructura de la aplicación

En este capítulo abarcaremos la estructura lógica de la aplicación y las interacciones de las clases entre sí. Los apartados son los siguientes: MVVM, diagrama de clases implementado, serialización, flujo de la aplicación.

5.1 Patrón Modelo - Vista - VistaModelo

A continuación veremos las propiedades clave del patrón MVVM que hemos utilizado para este proyecto:

DataTemplate: es un concepto similar al *Control Templates*. Proporcionan una gran flexibilidad y una solución ponderosa para reemplazar la apariencia visual de un elemento de datos dentro de un control como: *ListBox*, *ComboBox* o *ListView*. En este proyecto se utiliza *DataTemplate* para dibujar los elementos gráficos usando el control *ListBox*, todo a través de XAML. La idea principal del *DataTemplate* es enlazarlo con el *DataContext* para poder representar los datos que contiene el contexto. Se ve un ejemplo de implementación de *DataTemplate* en la Figura 30.

PropertyChanged: En C# las propiedades siguen un cierto patrón de implementación. Por una parte un campo privado empezando por una letra minúscula y luego una campo público que empieza con la primera letra en mayúscula. El campo público se declara con llaves y se definen el get y el set de la propiedad. Estos patrones de acceso y modificación pueden contener lógica compleja (Figura 31) o señales que indican a la capa de vista el evento *PropertyChanged* (Figura 32). Para que un elemento de la capa de vista observe la señal, se utiliza en XAML la palabra reservada *Binding* y se asigna el nombre de la propiedad que se quiere observar/enlazar.

```
<!-- This is the DataTemplate that will be used to render the Transition class -->
<DataTemplate DataType="{x:Type local:Transition}">
  <Thumb DragDelta="Thumb_DragTransitions"
    IsEnabled="{Binding IsSelected,RelativeSource={RelativeSource FindAncestor, AncestorType={x:Type ListBoxItem}}}"
  <Thumb.Template>
    <ControlTemplate TargetType="Thumb">
```

Figura 30 XAML de *DataTemplate* y *ListBox* para la clase *Transition*

```

public DiagramObject SelectedObject
{
    get { return _selectedObject; }
    set
    {
        Nodes.ToList().ForEach(x => x.IsHighlighted = false);
        Transitions.ToList().ForEach(x => x.IsHighlighted = false);
        InterNodes.ToList().ForEach(x => x.IsHighlighted = false);
        Inputs.ToList().ForEach(x => x.IsHighlighted = false);
        Outputs.ToList().ForEach(x => x.IsHighlighted = false);
        NotInputs.ToList().ForEach(x => x.IsHighlighted = false);
        NotOutputs.ToList().ForEach(x => x.IsHighlighted = false);

        if (_selectedObject != null && !_selectedObject.IsNew && (_selectedObject is Node || _selectedObject is Transition
            || _selectedObject is InterNode || _selectedObject is OutputLadder || _selectedObject is InputLadder
            || _selectedObject is NotOutputLadder || _selectedObject is NotInputLadder))
        {
            PreviousObject = _selectedObject; // Save the previous SelectedObject if it is not new
        }
        _selectedObject = value;
        OnPropertyChanged("SelectedObject");
        // Enable Delete and InterNode Command only when we are selecting an object
        DeleteCommand.IsEnabled = value != null;
        InterNodeEnabled = value != null;

        // If we are selecting a Connector, we highlight its start and end
        var connector = value as Connector;
        if (connector != null)
        {
            if (connector.Start != null) connector.Start.IsHighlighted = true;
            if (connector.End != null) connector.End.IsHighlighted = true;
        }
    }
}

```

Figura 31 Propiedad SelectedObject - Lógica del set

```

<TextBox Grid.Column="1" Grid.Row="0" Text="{Binding Name, UpdateSourceTrigger=PropertyChanged}" />
<TextBox Grid.Column="1" Grid.Row="1" Text="{Binding X}" />
<TextBox Grid.Column="1" Grid.Row="2" Text="{Binding Y}" />

```

Figura 32 Ejemplo de binding del nombre y coordenadas

Command: Permite hacer un *binding* de un evento asociado elemento gráfico de la interfaz de usuario (por ejemplo: clic, pulsado, etc...) con un comando que ejecuta un método. De esta forma se puede asociar de forma abstracta cualquier elemento de la UI con el comando definido. En este proyecto se utiliza una clase que hereda de *ICommand* (Figura 33) para poder enlazar y ejecutar los comandos deseados como por ejemplo: Abrir, Guardar, Cerrar, Eliminar, Acerca de. (Figura 34)

```

public class Command : ICommand
{
    public Action Action { get; set; }

    public void Execute(object parameter) { if (Action != null) Action(); }

    public bool CanExecute(object parameter) { return IsEnabled; }

    private bool _isEnabled;
    public bool IsEnabled {
        get { return _isEnabled; }
        set
        {
            _isEnabled = value;
            if (CanExecuteChanged != null)
                CanExecuteChanged(this, EventArgs.Empty);
        }
    }

    public event EventHandler CanExecuteChanged;
    public Command(Action action) { Action = action; }
}

```

Figura 33 Clase Command

```

private Command _aboutCommand;
public Command AboutCommand
{
    get { return _aboutCommand ?? (_aboutCommand = new Command(AboutProject)); }
}
/// <summary>
/// Launches the About Window
/// </summary>
private void AboutProject()
{
    About w = new About();
    w.Show();
}

```

Figura 34 About Command

DataContext: En WPF tenemos dos secciones dentro de una vista: la sección UI y la sección de datos. La sección de datos empieza inicializada a null y se puede asignar con la propiedad *DataContext* (Figura 35). Todos los objetos UI heredarán su *DataContext* a través del elemento padre al menos que los especifiquemos de otra forma. Cuando utilizamos el patrón MVVM, el *DataContext* representa la aplicación mientras que los objetos UI como botones, etiquetas, tablas o ventanas son simplemente objetos visuales intuitivos que permiten interactuar con el *DataContext*, que se encuentra en los *ViewModels* y *Models*.

Desarrollo de una aplicación gráfica para la edición de diagramas Grafcet y ladder

En cuanto utilizamos un binding en WPF, enlazamos con el *DataContext*. Por ejemplo cuando declaramos en el XAML:

```
<Label Name="miEtiqueta" Content="{Binding Path=Nombre}"/>
```

Enlazamos a *miEtiqueta.DataContext.Nombre*, y no a *miEtiqueta.Nombre*.

Se pueden utilizar otras propiedades como *ElementName* o *RelativeSource* para hacer un *binding* a una propiedad que está localizada fuera del *DataContext* actual.

```
public MainWindow()
{
    InitializeComponent();
    DataContext = new MainViewModel(this);
}
```

Figura 35 Asignación del DataContext

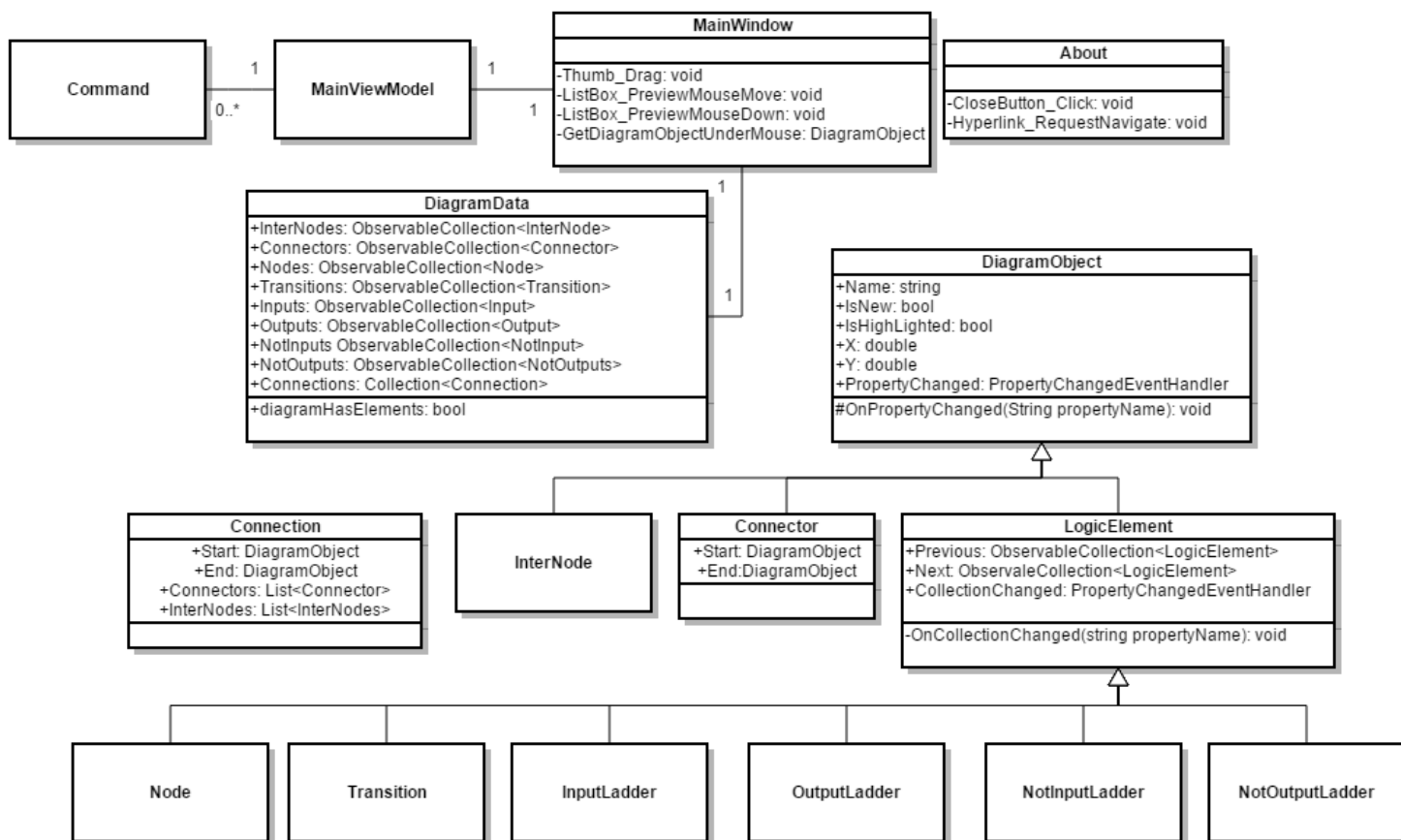
La repartición de clases y la repartición de capas del proyecto sería:

- **Capa de vista:** *MainWindow*, *About*.
- **Capa de modelo-vista:** *MainViewModel*.
- **Capa de modelo:** Todos las clases de Elementos gráficos y Modelos de datos.

Tenemos una ventana principal que se encarga de posicionar los elementos gráficos y consulta el modelo-vista para saber en qué modo de edición estamos y que acciones podemos hacer. Una vez realizada una acción, una parte de la lógica se resuelve en la ventana principal y la consistencia de datos es asegurada por el *binding* y también los métodos del modelo-vista.

5.2 Diagrama de Clases implementado

En el siguiente diagrama de clases tenemos el resultado de la implementación con todas las clases utilizadas para el proyecto. La clase *MainWindow* se encarga de refrescar la posición de los elementos en función de la posición del ratón con los métodos *drag* cuando estamos seleccionado un elemento en concreto. También se encarga de refrescar la pre-visualización y el clic que se hace sobre el *listbox* que será nuestra área de trabajo. La clase *About* corresponde a la ventana de 'Acerca de' y contiene simplemente información: texto, hiperenlaces e imagen. La clase *Command* es una implementación simple de la clase *ICommand* de WPF que sirve para gestionar acciones de botones con *binding*. En la clase *DiagramData* disponemos de todas las colecciones de elementos gráficos en función de su tipo y también una colección de *Connections*. Como dicho previamente, las *Connections* no son elementos gráficos, sino un conjunto de ellos. Cada *Connection* tiene una referencia *Start* y *End* que indican a que elementos lógicos conecta así como una colección de *InterNode* y de *Connector*.



La clase *DiagramObject* tiene los siguientes campos: *Name* (Nombre), *isNew* (booleano que indica si estamos pre-visualizando), *isHighLighted* (booleano que indica si estamos subrayando), *X* e *Y* (coordenadas del objeto en el *listbox*) y por último el manejo de eventos *PropertyChanged*.

La clase *LogicElement* indica que elementos lógicos conectados a este son previos o siguientes con las colecciones *Previous* y *Next*. También dispone de un manejador de eventos para indicar la alteración de estas colecciones: *CollectionChanged*.

La clase *MainViewModel* dispone de acceso a las colecciones de un *DiagramData* asociado al *ViewModel*. En esta clase tenemos diversos comandos como *OpenProjectCommand*, *NewProjectCommand*, etc... Esta clase se encarga de gestionar la visibilidad de ciertos atributos, lanzar comandos (abrir, guardar, borrar), crear objetos, permitir el sistema de *scroll*, localizar un objeto en el interfaz y monitorizar la selección actual (que *DiagramObject* está siendo seleccionado).



5.3 Serialización XML y reconstrucción del *DiagramData*

Para poder exportar e importar Diagramas en fichero XML vamos a utilizar *XmlSerializer* que está disponible en la librería System.XML. En la Figura 36 podemos observar como serializamos un *DiagramData*. La clase *DiagramData* es simplemente el conjunto de datos que componen el Diagrama y que se crea de forma única con patrón *singleton*.

De esta forma podemos serializar el tipo *DiagramData* y guardar fácilmente todas las colecciones necesarias. Sin embargo hay atributos que no se guardan en el fichero (explicado más en detalle en el apartado de implementación) con la etiqueta *[XmlIgnore]* como se ve en la Figura 37.

```
private void SaveProject()
{
    SaveFileDialog saveFileDialog1 = new SaveFileDialog();
    saveFileDialog1.Filter = "XML File|*.xml";
    saveFileDialog1.Title = "Save XML File";
    saveFileDialog1.ShowDialog();

    if (saveFileDialog1.FileName != "")
    {
        FileStream fs = (FileStream)saveFileDialog1.OpenFile();
        new XmlSerializer(typeof(DiagramData)).Serialize(fs, dd);
        fs.Close();
    }
}
```

Figura 36 Serialización XML

```
[XmlIgnore]
public ObservableCollection<LogicElement> Previous
{
    get
    {
        if (previous != null) return previous;
        previous = new ObservableCollection<LogicElement>();
        previous.CollectionChanged += Previous_CollectionChanged;
        return previous;
    }

    set { }
}
```

Figura 37 Etiqueta XmlIgnore

El proceso de des-serializar es más complejo ya que requiere una reconstrucción de la lógica y del posicionamiento de los objetos en el área de diseño. Para ello utilizaremos un *DiagramData* auxiliar donde recuperamos los objetos y colecciones del fichero XML. En primera parte rellenaremos los elementos lógicos del auxiliar al *DiagramData* asociado al *ViewModel*. En segundo lugar reconstruimos las *Connections* (Figura 38):

- Añadimos las *Connections* en el *DiagramData* del *ViewModel*
- Buscamos los elementos lógicos que ya están presentes en las colecciones del *DiagramData* del *ViewModel* y que corresponde a las coordenadas *Start* y *End* de la *Connection*. Esto es debido a que las colecciones de la *Connection* indican la referencia a los elementos existentes, están repetidos debido al proceso de serialización que no permite identificarlos como únicos. Actualizamos de forma parecida los *InterNodes* que son los puntos intermedios del arco.
- Añadimos los *connectors* y asignamos las referencias *Start* y *End* de cada uno con los objetos del diagrama válidos. Se añaden los *connectors* desde las *connections* porque en el *DiagramData* se ignora la serialización de los *connectors*. Ya disponemos de todos los *connectors* a partir de las colecciones de los *connections*.

```
// Rebuilding connections
foreach (Connection n in auxDD.Connections)
{
    dd.Connections.Add(n);
    n.Start = GetDiagramObject(n.Start);
    n.End = GetDiagramObject(n.End);
    Collection<InterNode> auxCollection = new Collection<InterNode>();
    // For each internode, we need to make sure
    // it is one of those already initialized and present in dd.InterNodes
    foreach (InterNode inter in n.InterNodes){ auxCollection.Add(GetDiagramObject(inter) as InterNode); }
    foreach (InterNode inter in auxCollection){ n.InterNodes.Add(inter); }
    foreach (Connector c in n.Connectors)
    {
        dd.Connectors.Add(c);
        c.Start = GetDiagramObject(c.Start);
        c.End = GetDiagramObject(c.End);
    }
}
```

Figura 38 Reconstrucción de las *Connections*

Y en tercer lugar reconstruimos las referencias lógicas *Previous* y *Next* (Figura 39) que no se serializan en la función de guardado. Para ello iteramos entre todas las *connections* y referimos los elementos lógicos *Start* y *End* entre ellos.

```

LogicElement auxLogicElement;
foreach (Connection n in dd.Connections)
{
    // We need to fill the Next and Previous lists of each logicElement
    // that is connected to a valid connection
    if (n.Start != null && n.End != null)
    {
        auxLogicElement = n.Start as LogicElement;
        auxLogicElement.Next.Add(n.End as LogicElement);
        auxLogicElement = n.End as LogicElement;
        auxLogicElement.Previous.Add(n.Start as LogicElement);
    }
}

```

Figura 39 Reconstrucción de la lógica

5.4 Flujo de la aplicación

En la barra de botones tenemos dos categorías: *Button* y *ToggleButton*. Los *Button* están asociados a comandos, es decir unos métodos que se ejecutan en cuanto el usuario pulsa uno de estos botones. Para los *ToggleButton* tenemos que el clic activa un estado ‘*Checked*’ (Marcado) que se mantiene hasta que acabe la funcionalidad del botón y se desactive (Figura 40).

```

</ToggleButton>
<!-- Button : Create New Transition -->
<ToggleButton IsChecked="{Binding CreatingNewTransition}" Tooltip="Insert Transition">
  <Rectangle Height="5" Width="20" Stroke="Black"
    StrokeThickness="1" Fill="Black">
  </Rectangle>
  <!-- Button : Create New Connection -->
</ToggleButton>

```

Figura 40 ToggleButton Binding

Los *ToggleButton* tienen un *binding* de su estado *IsChecked* con un booleano definido especialmente para cada uno. Esto permite poder influir desde la lógica sobre el estado de estos botones alterando el valor del booleano. Si el booleano cambia, el estado *IsChecked* también y por lo tanto el estado visual del botón también. De la misma forma si el estado *IsChecked* se altera, el valor booleano se sincronizará con él.

Cuando pulsamos el botón de Nuevo, se comprueba si hay elementos inicializados en las colecciones observables. En el caso de que hayan, entonces se preguntará con un dialogo si el usuario desea guardar, después de resolver la duda se vaciaran todas las colecciones para comenzar un nuevo diagrama.

Con el botón de Abrir, se comprueba otra vez si hay elementos para proponer guardar. A continuación se vacían las colecciones y se abre un dialogo para buscar el archivo XML para abrirlo y reconstruir la lógica una vez las colecciones rellenas de nuevo. Y

con el botón de Guardar serializamos como explicado previamente. Y con el botón *About* abrimos el dialogo de información.

En el proyecto tenemos definido la clase *LogicElement* que hereda de la clase *DiagramObject*. *LogicElement* tiene referencias lógicas como *Next* y *Previous* que son colecciones de otros *LogicElement* que tienen conexiones en común entre sí. Un *LogicElement* corresponde a todo elemento gráfico que no sea *Connector* o *InterNode*. Las *Connections* no son elementos gráficos ya que son clases lógicas que contienen elementos gráficos (*Connectors* e *InterNodes*).

Cuando pulsamos un botón de insertar un elemento gráfico que hereda de *LogicElement* entonces activamos el booleano correspondiente y creamos el objeto pero le asignamos la propiedad *IsNew* al valor *true*. Esto permite pre-visualizar el elemento gráfico e indicarle al usuario que estamos posicionando un elemento que podría ser confirmado por el usuario si decide clicar en una posición determinada del área de diseño.

Cuando el usuario desplaza el ratón mientras esta el booleano activo que indica el modo de inserción, la pre-visualización del objeto sigue al ratón y se va saltando a la posición del *grid* más cercana, esta colocación se calcula internamente en la clase *DiagramObject*.

Si el usuario decide cancelar la operación de inserción clicando de nuevo el *ToggleButton* que había activado esta misma operación, entonces el elemento se queda en estado de pre-visualización y desaparecerá una vez se haya confirmado una creación de otro elemento. Si justamente el usuario hace clic en el área de diseño, entonces el elemento seleccionado (el que se acaba de crear y que tiene propiedad *IsNew* al valor *true*) se confirmará y ya no se considera elemento nuevo y deja el estado de pre-visualización, cogiendo las coordenadas de la posición en la que se encuentra en el momento de clic. Además se borrarán todos los elementos nuevos que no hayan sido confirmados por el usuario.

Cuando insertamos una conexión, necesitamos previamente haber seleccionado un elemento gráfico conectable. Al entrar en el modo conexión, creamos un inter-nodo nuevo que se pre-visualiza con además un conector conectado a él y se puede confirmar por el usuario. Si el usuario hace clic sobre una posición vacía entonces se crea el inter-nodo y seguimos en modo conexión para que el usuario pueda dar el siguiente punto de conexión hasta que llegue a un elemento lógico, ya que la conexión simboliza un flujo entre dos elementos lógicos. Si el usuario cancela la operación de conexión haciendo clic en el *ToggleButton* correspondiente entonces se quedara la conexión inacabada y será eliminada en próxima conexión exitosa. Si el usuario hace clic sobre un elemento lógico, entonces se crea una conexión y se apunta sus referencias *Start* y *End*, así como se apuntan las referencias *Next* y *Previous* de los elementos lógicos unidos por la conexión. Cada conexión representa un arco entre dos elementos lógicos. Cada vez que añadimos un inter-nodo se van añadiendo los conectores y los inter-nodos a la *newConnection* que es la referencia auxiliar de la conexión que se está creando. Cuando se intenta conectar un elemento lógico consigo mismo se anula la conexión y se elimina. Si el usuario hace clic sobre otro inter-nodo mientras se está en modo



conexión, se anula y elimina la nueva conexión ya que no se pueden conectar conexiones juntas, las conexiones solo pueden tener un inicio y un fin (*Start & End*).

Cuando pulsamos el botón de eliminar se procede en función de que elemento tenemos seleccionado:

- Elemento lógico : buscamos las conexiones que llegan o salen del elemento, las eliminamos y después eliminamos al elemento
- Conector: se busca la conexión que contiene el conector y se elimina la conexión que a su vez elimina todos los conectores e inter-nodos que contiene.
- Inter-nodo: se busca la conexión que contiene el inter-nodo y se elimina la conexión que a su vez elimina todos los conectores e inter-nodos que contiene.

A la hora de editar los atributos *Width* y *Height* el usuario puede determinar cuál es el área de *scroll*, es decir el área por la que se activarán las barras de desplazamiento vertical y/o horizontal.

Con el *check-box ShowNames* se active o desactiven las etiquetas que muestran los nombres. Los *RadioButtons* permite la misma funcionalidad con las coordenadas de los elementos.

Cada vez que seleccionamos un elemento el editor propiedades se actualiza con la señal que avisa el cambio de selección. Entonces en función del tipo del objeto seleccionado se puede mostrar ciertas propiedades. En esta versión los elementos lógicos muestran sus coordenadas mientras que los conectores muestran los elementos inicial y final del conector y sus respectivos campos. Ambos editores se pueden plegar o desplegar.

6. Detalles de Implementación

En este capítulo vamos a comentar unos detalles complementarios con respecto al desarrollo y la implementación del proyecto.

6.1 WPF 2D Graphics

Al inicio del desarrollo del proyecto, se trató de trazar manualmente los gráficos a través de código en vez de WPF. Es una alternativa posible para tener un modelo que no requiere flexibilidad. Sin embargo, se puede hacer incluso más complejo que con XAML ya que no hay redibujado dinámico por defecto.

Para utilizar los gráficos 2D de WPF se pueden utilizar las clases: *Shape*, *Brush*, *DrawingImages*, *DrawingContext*.

Durante el desarrollo de esta alternativa se acabó optando por el uso de XAML gracias a varios ejemplos introductorios [22] que muestran el poder y la flexibilidad que ofrece WPF combinado con XAML para dibujar de forma dinámica todos los elementos gráficos.

6.2 ObservableCollection

En WPF y MVVM es importante disponer de señales que avisan cuando tenemos cambios de propiedades. Lo mismo ocurre con las colecciones y por ello necesitamos las *ObservableCollections*. Estas son colecciones clásicas pero que dispone de señales que avisan cuando el contenido de la colección ha cambiado, por ejemplo cuando quitamos, añadimos o vaciamos la colección. Sin embargo no tiene señal para cuando se modifica una propiedad de uno de los objetos que contiene. Se puede implementar y podría servir para extender la lógica del proyecto. Una implementación posible se propone en la Figura 41.

```
ObservableCollection<INotifyPropertyChanged> items =
    new ObservableCollection<INotifyPropertyChanged>();
items.CollectionChanged +=
    new System.Collections.Specialized.NotifyCollectionChangedEventHandler(
        items_CollectionChanged);

static void items_CollectionChanged(object sender,
    System.Collections.Specialized.NotifyCollectionChangedEventArgs e)
{
    foreach (INotifyPropertyChanged item in e.OldItems)
        item.PropertyChanged -= new
            PropertyChangedEventHandler(item_PropertyChanged);

    foreach (INotifyPropertyChanged item in e.NewItems)
        item.PropertyChanged +=
            new PropertyChangedEventHandler(item_PropertyChanged);
}

static void item_PropertyChanged(object sender, PropertyChangedEventArgs e)
{
    throw new NotImplementedException();
}
```

Figura 41 ObservableCollection extendida

6.3 DataTemplate

Con el uso de *DataTemplate* se dibuja cada uno de los elementos gráficos del área de diseño. La ventaja principal es poder aplicar setter sobre los objetos gráficos que componen el *DataTemplate*. Con esto podemos cambiar el color o el tamaño de una forma específica y con su nombre propio definido. Hacemos esto por ejemplo para mostrar cuál es el elemento seleccionado gracias al uso de *Setter* o *Triggers* en XAML dentro del *DataTemplate*. Vemos un ejemplo en la Figura 42 donde se rellena de color Rojo en elemento llamado "Rectangle" o bien se rellena de color Verde si se está arrastrando.

```
<Rectangle Height="40" Width="40" Stroke="Black" StrokeThickness="1"
  x:Name="Rectangle">
</Rectangle>
<TextBlock Canvas.Top="10" Canvas.Left="-30" Width="100"
  TextAlignment="Center" Text="{Binding Name}" FontWeight="Normal"
  IsHitTestVisible="False"
  Visibility="{Binding DataContext.ShowNames,
    RelativeSource={RelativeSource FindAncestor,
    Converter={StaticResource BoolToVisible}}"/>
<TextBlock Canvas.Left="30" Canvas.Top="10"
  Text="{Binding X, StringFormat='{0}X = {0}'}"
  IsHitTestVisible="False"
  Visibility="Collapsed" x:Name="XText"/>
<TextBlock Canvas.Left="30" Canvas.Top="25"
  Text="{Binding Y, StringFormat='{0}Y = {0}'}"
  IsHitTestVisible="False"
  Visibility="Collapsed" x:Name="YText"/>
</DataTemplate>
<ControlTemplate.Triggers>
<DataTrigger Binding="{Binding IsSelected, RelativeSource={RelativeSource FindAncestor,
  Converter={StaticResource BoolToVisible}}"/>
  <Setter TargetName="Rectangle" Property="Fill" Value="Red" >
</DataTrigger>
<Trigger Property="IsDragging" Value="True">
  <Setter TargetName="Rectangle" Property="Fill" Value="Green" >
</Trigger>
</ControlTemplate.Triggers>
```

Figura 42 DataTemplate Triggers

Otra ventaja es la reutilización del código. Pero una desventaja es la poca flexibilidad que tiene con respecto a polimorfismo.

6.4 Grid Snapping

Dentro de la clase *DiagramObject* tenemos un set que establece la fórmula de la Figura 43 para posicionar los elementos en forma de *grid* de 50 x 50 pixeles por posición.

```
private double _x;
public double X
{
    get { return _x; }
    set
    {
        // "Grid Snapping"
        // this actually "rounds" the value so that it will always be a multiple of 50.
        _x = (Math.Round(value / 50.0)) * 50;
        OnPropertyChanged("X");
    }
}
```

Figura 43 Grid Snapping

6.5 Referencias cíclicas y Serialización XML

Durante el desarrollo de la serialización XML surgió un error que daba un descripción incompleta: “Error al generar el fichero XML”. Se trataba de una excepción disparada por el método *Serialize* pero que a su vez era activada por otra excepción interna. Entonces se investigó hasta descubrir que la versión Express de Visual Studio no dispone de manejador visual de excepciones pero se puede con un try catch poner un *breakpoint* y abrir la instancia de la excepción y abrir en arborescencia las excepciones internas.

El error completo precisa que el *XMLSerializer* no puede serializar cuando hay referencias cíclicas. Por lo tanto se tuvo que desactivar la serialización de las etiquetas *Previous* y *Next* de los elementos lógicos cuando el diagrama tenía un bucle. Así es como se tuvo que implementar entonces la reconstrucción de la lógica al des-serializar un fichero XML.

7. Conclusiones

A lo largo de la titulación no hemos tenido la ocasión de tomar tantas decisiones en un solo proyecto y con tanta libertad de acción y decisión. Es una experiencia muy interesante que nos hace plantearnos de forma continua cual es la mejor herramienta, la mejor solución o implementación, así como la gestión del propio proyecto. Al fin de al cabo este tipo de proyecto incita a interpretar que disponer de conocimientos especializados en todas las tecnologías no es tan importante como tener la capacidad de abarcar las diferentes alternativas y escoger la mejor para un cierto problema o situación para obtener la mayor eficiencia.

En este proyecto en concreto, las conclusiones a nivel de diseño son varias. En primer lugar, usar un buen patrón de diseño para la arquitectura del software es esencial para poder utilizar múltiples librerías y apoyo de estándares de desarrollo previamente probados y soportados. Por otra parte el poder utilizar una documentación completa del proyecto es una gran ventaja, ya que con el tamaño cada vez más grande de este, se puede incluso olvidar el funcionamiento de un método desarrollado previamente. Es evidente que la documentación es una clave para poder compartir un proyecto entre varios desarrolladores. Pero con la amplitud de este proyecto, incluso a nivel personal es imprescindible tener una documentación completa que abarca la mayoría de clases, métodos y propiedades para poder utilizar *IntelliSense* y los *Tooltips* dentro del *IDE*.

El *databinding* utilizado entre las capas de Modelo y Vista resulta ser muy poderoso pero requiere tener un conocimiento de los conceptos suficiente para aprovechar su uso. Aunque parezca un esfuerzo superior tener que incluir el patrón de diseño MVVM, a largo plazo es evidente que el mantenimiento y la flexibilidad son los puntos clave de este paradigma de desarrollo.

Para un trabajo futuro se podría ampliar el proyecto con un intérprete lógico de los diagramas y añadir los estados de activación, estados iniciales y condiciones en el editor de propiedades usando el mismo tipo de *binding* usado para las coordenadas.

Glosario

Add-on: extensión de un programa.

Framework: estructura conceptual y tecnológica de soporte definido, normalmente con artefactos o módulos de software concretos, que puede servir de base para la organización y desarrollo de software.

Debug / Debugging: modo de ejecución orientado a detectar y analizar errores, fallos o excepciones.

Assembly references: variables de entorno que se pueden añadir para acceder a ciertas librerías.

Databinding: marcado de enlace de datos que sincroniza el contenido de un elemento de la interfaz de usuario con un dato característico de la capa de modelo.

Grid: distribución de elementos en función filas y columnas.

Layout: “disposición” o “plan” y tiene un uso extendido en el ámbito de la tecnología.

Tooltip: Mensaje emergente que da información de un elemento gráfico cuando el cursor permanece encima de este mismo durante un periodo determinado de tiempo.

Scroll: Función de desplazar la zona visible de un área más amplia con por ejemplo una barra de desplazamiento.

Checkbox: Casilla de marcado.

XAML: *eXtensible Application Markup Language* es el lenguaje de formato para la interfaz de usuario para la Base de Presentación de Windows (WPF por sus siglas en inglés), el cual es uno de los "pilares" de la interfaz de programación de aplicaciones .NET

Singleton: Patrón que garantiza una única instancia creada posible de un objeto característico.

IntelliSense: es la aplicación de autocompletar, mejor conocido por su utilización en Microsoft Visual Studio entorno de desarrollo integrado. Además de completar el símbolo de los nombres que el programador está escribiendo, *IntelliSense* sirve como documentación y desambiguación de los nombres de variables, funciones y métodos de utilización de metadatos basados en la reflexión.

IDE: Un ambiente de desarrollo integrado o entorno de desarrollo interactivo, es una aplicación informática que proporciona servicios integrales para facilitarle al desarrollador o programador el desarrollo de software.

Bibliografía

Libros

- [1] “Automática Industrial y Control”, A. Cuenca, J. Salt. ISBN:84-9705-783-X. SPUPV: 2005.349
- [2] “Automatización de Procesos Industriales”, E. García SPUPV-4116, 2005.
- [3] “Prácticas de Automatización Industrial. Programación del Autómata Programable TSX37 de Télémecanique”, J.L Díez, E.J. Bernabeu, A. Esparza. SP-UPV-406, 2005
- [4] “Autómatas Programables Industriales: Arquitectura y Aplicaciones”. J. C. Bossy. 1995.
- [5] “Grafcet. Práctica y Aplicaciones”. J.-C. Bossy
- [6] “Ingeniería de la Automatización Industrial”. R. Piedrafita, 2004.
- [13] “Enciclopedia de Microsoft Visual C#” Ceballos Sierra, Francisco Javier 2010
- [14] “C# 5 - Desarrolle aplicaciones Windows con Visual Studio 2013” HUGON, J. [2-7460-8987-4] 2014
- [19] Pro WPF and Silverlight MVVM: Effective Application Development with Model-View-ViewModel [1-4302-3162-9; 1-4302-3163-7] Hall, Gary McLean, 2010

Referencias de Fuentes Electrónicas

[7] Di-Meglio Stéphane – Automatisme Industriel

<http://stephane.dimeglio.free.fr/sfcedit/en/>

[8] REEA – Revista de Electricidad, Electrónica y Automática.

<http://reea-blog.blogspot.com.es/p/freesfc-diseno-de-graficos-grafcet.html>

[9] Grafcet Designer for LabView

<http://www.grafcetview.com/en>

[10] National Instruments - License

<http://sine.ni.com/nips/cds/view/p/lang/es/nid/212492>

[11] Classic Ladder

<https://sites.google.com/site/classicladder/>

[12] SoapBox Snap

<http://soapboxautomation.com/products/soapbox-snap/>

[15] GhostDoc

<http://submain.com/products/ghostdoc.aspx>

[16] GhostDoc en la VisualStudioGallery

<https://visualstudiogallery.msdn.microsoft.com/46A20578-F0D5-4B1E-B55D-FO01A6345748>

[17] Sandcastle

<https://sandcastle.codeplex.com/>

[18] Sandcastle en Github

<https://github.com/EWSsoftware/SHFB>

[19] Creating documentation in C# using Visual Studio and Sandcastle

<http://blogs.msdn.com/b/msgulfcommunity/archive/2014/04/22/creating-documentation-in-c-using-visual-studio-and-sandcastle.aspx>

[20] The MVVM Pattern

<https://msdn.microsoft.com/en-us/library/hh848246.aspx>

[21] WPF 2D Graphics Tutorial

<https://www.youtube.com/watch?v=wTxktpA5GUM>

[22] WPF Samples

<https://github.com/High-Core/WPFSamples>

Referencias generales

MSDN – Microsoft Developer Network - Contiene una gran cantidad de información técnica de programación, incluidos código de ejemplo, documentación, artículos técnicos y guías de referencia:

<https://msdn.microsoft.com/es-es/default.aspx>

Wikipedia – La enciclopedia libre:

<https://es.wikipedia.org/>

Stack Overflow - Es un sitio web desarrollado por Jeff Attwood, este sitio web es utilizado por una comunidad de desarrolladores informáticos, en la cual otros desarrolladores pueden encontrar soluciones a problemas de programación en diferentes lenguajes:

<http://stackoverflow.com/>

Real Academia Española:

<http://www.rae.es/>

Repositorio del proyecto

<https://github.com/cloporte/TFG-Grafcet-Ladder-Editor>

Anexo – Manual de Usuario

En esta guía para el usuario se va explicar el buen uso de los controles de la aplicación y su funcionalidad. También se comenta las directivas de diseño a seguir.

Barra de herramientas:



Nuevo (*New*): Con este botón se borra y reinicia el estado del área de diseño, pero se comprueba si el usuario tiene elementos gráficos inicializados. Si el usuario tiene un diagrama empezado se abre un dialogo simple donde se pregunta al usuario si desea guardar el/los diagrama(s) actuales.



Guardar (*Save*): Lanza la función de guardado en formato XML. Un dialogo aparece entonces para que el usuario decida de la ruta y el nombre del archivo que quiere guardar.



Abrir (*Open*): De la misma manera que la función Nuevo, se comprueba si el usuario tiene trabajo empezado y si lo desea guardar. Una vez se ha resuelto ese aspecto, este botón abre un dialogo para abrir un fichero de formato XML para importarlo en el área de diseño.



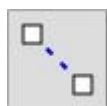
Acerca de (*About*): Abre un dialogo con información relativa al proyecto: autor, fecha, organización, enlace hacía la organización y enlace del manual de usuario.



Insertar etapa (*Node*): Activa el modo de inserción de una nueva etapa en la zona de diagrama.



Insertar transición (*Transition*): Permite insertar una transición en el área de diseño



Conectar (*Connection*): Este botón esta deshabilitado por defecto y se habilita cuando el usuario selecciona un elemento gráfico. Una vez activado el modo de inserción de puntos intermedios del arco.



Insertar contacto (*Input*): Crea un nuevo contacto para que el usuario lo pueda posicionar en el lienzo.



Insertar contacto negado (*Not Input*): Inserta una entrada negada en el diagrama.



Insertar bobina (Output): Activa el modo de inserción de bobina en el área de trabajo.



Insertar bobina negada (Not Output): Añade una bobina negada, que invierte el valor lógico de la función de la combinación de contactos.

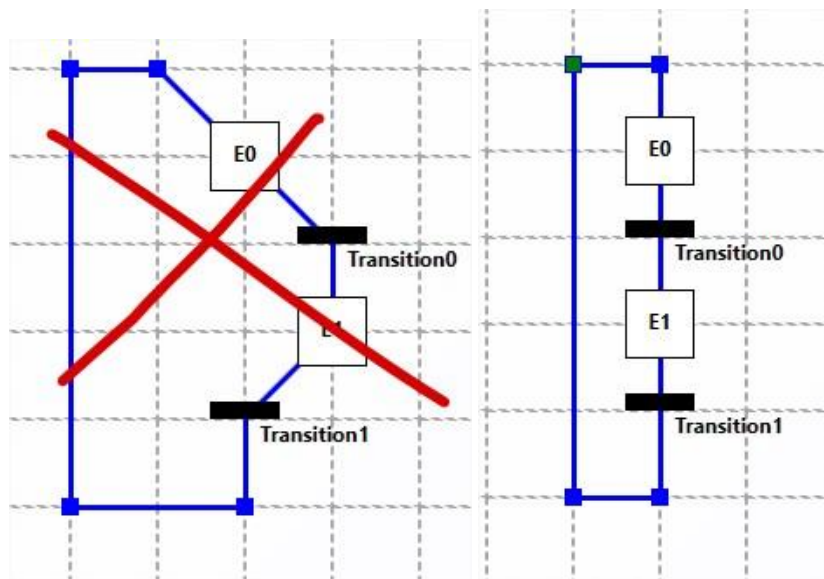


Eliminar Elemento (Delete): Se habilita este botón cuando tenemos seleccionado un elemento gráfico cualquiera y nos permite eliminarlo. Si el elemento tiene conexiones, estas también son eliminadas. También permite eliminar arcos.

Directivas de diseño Grafcet:

Las conexiones deben tener segmentos horizontales y verticales. Evitar segmentos en diagonal y cruzar los arcos entre-sí.

Las conexiones se deben de hacer en orden descendente, es decir que clicamos un elemento lógico (Etapa o Transición) para conectarlo con su siguiente. Si queremos representar un enlace ascendente bajaremos en línea recta y subimos por un lateral.



Los elementos lógicos deben estar posicionados verticalmente y no puede estar conectados horizontalmente, esa disposición es para esquemas Ladder.

No se debe dejar una conexión cortada o tratar de reconectar una conexión cortada.

No superponer elementos en las mismas coordenadas.

Directivas de diseño Ladder:

Las conexiones con segmentos verticales tienen que trazarse primero, las conexiones horizontales trazan después.

Los elementos van de orden lógico de izquierda a derecha, de los previos a los siguientes. Los contactos en la parte izquierda y las bobinas en la parte derecha.

No puede haber una bobina con en la misma columna que un contacto. Una bobina no debe estar a la izquierda de un contacto.

