Desarrollo de un Lenguaje de Dominio Específico (DSL) para programar una placa Arduino utilizando el paradigma de Ingeniería Dirigida por Modelos (MDE)

Franco Román, Jonathan Rubén y Sánchez Tapia, Benjamín Alejandro

Departamento de Eléctrica y Electrónica

Carrera de Ingeniería en Software

Trabajo de titulación, previo a la obtención del título de Ingeniero en Software

Ph.D. Jácome Guerrero, Patricio Santiago

Latacunga 17 de agosto del 2021

DEPARTAMENTO DE ELÉCTRICA Y ELECTRÓNICA

CARRERA DE INGENIERÍA EN SOFTWARE

CERTIFICACIÓN

Certifico que el trabajo de titulación**: "Desarrollo de un Lenguaje de Dominio Específico (DSL) para programar una placa Arduino utilizando el paradigma de Ingeniería Dirigida por Modelos (MDE)"** fue realizado por el/los señor/señores **Franco Román, Jonathan Rubén** y **Sánchez Tapia, Benjamín Alejandro** el cual ha sido revisado y analizado en su totalidad por la herramienta de verificación de similitud de contenido; por lo tanto cumple con los requisitos legales, teóricos, científicos, técnicos y metodológicos establecidos por la Universidad de las Fuerzas Armadas ESPE, razón por la cual me permito acreditar y autorizar para que lo sustenten públicamente.

Latacunga, 17 de agosto de 2021.

Firmado electrónicamente por:

PATRICIO
SANTIAGO JACOME
GUERRERO

.....

Ph.D. Jácome Guerrero, Patricio Santiago

C. C.: 1001689791

Curiginal

Document Information

Analyzed document	Tesis_DSL para programar una placa Arduino.pdf (D111514270)
Submitted	8/20/2021 4:45:00 PM
Submitted by	
Submitter email	psjacome@espe.edu.ec
Similarity	6%
Analysis address	psjacome.espe@analysis.urkund.com

PATRICIO SANTIAGO JACOME GUERRERO

Sources included in the report

W	URL: https://dialnet.unirioja.es/descarga/articulo/3869349.pdf Fetched: 8/20/2021 4:46:00 PM	2
SA	**Universidad de las Fuerzas Armadas ESPE / SIIPRIN_2017_paper_5_correguido-11.doc** Document SIIPRIN_2017_paper_5_correguido-11.doc (D31587905) Submitted by: psjacome@espe.edu.ec Receiver: psjacome.espe@analysis.urkund.com	4
W	URL: https://pdfcookie.com/documents/rama-desarrollo-de-software-dirigido-por-modelospdf-rv3w170n73ld Fetched: 8/18/2021 6:10:20 PM	15
SA	**Memoria_TFM_SantiagoJacome.pdf** Document Memoria_TFM_SantiagoJacome.pdf (D14890770)	5
W	URL: https://docplayer.es/76706069-Especificacion-de-un-metamodelo-para-apoyar-y-extender-la-propuesta-td-mbuid.html Fetched: 8/20/2021 4:46:00 PM	1
SA	**Tesis Revision Holger 1.0.pdf** Document Tesis Revision Holger 1.0.pdf (D46554637)	3
SA	**SIIPRIN_2017_paper_5.pdf** Document SIIPRIN_2017_paper_5.pdf (D29990173)	3
SA	**1347926.pdf** Document 1347926.pdf (D110430648)	1
W	URL: http://dis.um.es/~jmolina/Pfc/DSLvsMetaedit.pdf Fetched: 8/20/2021 4:46:00 PM	1
SA	**Universidad de las Fuerzas Armadas ESPE / PROYECTO_TITULACIÓN_MG_V10 (1) (Autoguardado)FINAL.docx** Document PROYECTO_TITULACIÓN_MG_V10 (1) (Autoguardado)FINAL.docx (D47309301) Submitted by: omiguelgp@hotmail.com Receiver: masoasti.espe@analysis.urkund.com	1
W	URL: https://github.com/Jrfranco2/circuitoV3 Fetched: 8/20/2021 4:46:00 PM	2

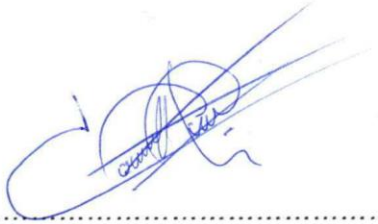
DEPARTAMENTO DE ELÉCTRICA Y ELECTRÓNICA

CARRERA DE INGENIERÍA EN SOFTWARE

RESPONSABILIDAD DE AUTORÍA

Nosotros, **Franco Román, Jonathan Rubén** con cédula de ciudadanía **N°1723486872** y **Sánchez Tapia, Benjamín Alejandro** con cédula de ciudadanía **N°0550095087**, declaramos que el contenido, ideas y criterios del trabajo de titulación: **"Desarrollo de un Lenguaje de Dominio Específico (DSL) para programar una placa Arduino utilizando el paradigma de Ingeniería Dirigida por Modelos (MDE)"** es de nuestra autoría y responsabilidad, cumpliendo con los requisitos legales, teóricos, científicos, técnicos, y metodológicos establecidos por la Universidad de las Fuerzas Armadas ESPE, respetando los derechos intelectuales de terceros y referenciando las citas bibliográficas.

Latacunga, 17 de agosto de 2021.



Franco Román, Jonathan Rubén
C.C.: 1723486872

Sánchez Tapia, Benjamín Alejandro
C.C.: 0550095087

ESPE
UNIVERSIDAD DE LAS FUERZAS ARMADAS
INNOVACIÓN PARA LA EXCELENCIA

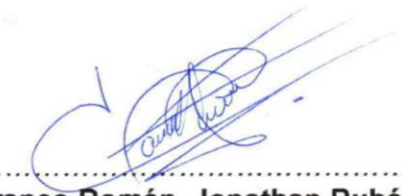
DEPARTAMENTO DE ELÉCTRICA Y ELECTRÓNICA

CARRERA DE INGENIERÍA EN SOFTWARE

AUTORIZACIÓN DE PUBLICACIÓN

Nosotros, **Franco Román, Jonathan Rubén** con cédula de ciudadanía **N°1723486872** y **Sánchez Tapia, Benjamín Alejandro** con cédula de ciudadanía **N°0550095087**, autorizamos a la Universidad de las Fuerzas Armadas ESPE publicar el trabajo de titulación: **"Desarrollo de un Lenguaje de Dominio Específico (DSL) para programar una placa Arduino utilizando el paradigma de Ingeniería Dirigida por Modelos (MDE)"** en el Repositorio Institucional, cuyo contenido, ideas y criterios son de mi/nuestra responsabilidad.

Latacunga, 17 de agosto de 2021.



.....
Franco Román, Jonathan Rubén
C.C.: 1723486872

.....
Sánchez Tapia, Benjamín Alejandro
C.C.: 0550095087

Dedicatoria

Dedico este proyecto de investigación a mi primo Víctor Andrés Ordoñez Román, quien me apoyo durante este difícil y largo camino, dándome ánimos y palabras de aliento para salir adelante, donde quiera que estés solo quiero decirte que lo logre y esto es por ti, ¡gracias primo!

Dedico con todo mi corazón mi proyecto de investigación a mi madre y a mi hermana, pues sin ellas este logro no hubiera sido posible, me brindaron su apoyo incondicional durante los días difíciles y estuvieron ahí cuando más las necesitaba.

También una dedicatoria para mi enamorada que supo cómo levantarme cuando ya no podía seguir adelante, y acompañarme a lo largo de toda esta difícil aventura.

Y, sobre todo agradezco a Dios por guiarme por el camino del bien y ayudarme a superar los fuertes obstáculos que se me cruzaron sobre mi travesía reconfortándome y dándome calma.

Jonathan Franco

Quiero dedicar el presente trabajo de titulación en primer lugar a Dios por ser mi guía espiritual en este camino, por haberme dado fortaleza en los momentos más difíciles y ayudarme a culminar con éxito.

A mi madre y abuela, pues sin ellas no lo habría logrado, por brindarme su apoyo y amor en todo momento, por haberme ensenado a tomar mis propias decisiones y seguir adelante con ellas.

A toda mi familia por el apoyo, por impulsarme a ser mejor y lograr con éxito mi carrera.

Benjamín Sánchez

Agradecimiento

Principalmente agradezco a Dios por permitirme culminar ese proyecto de investigación y seguir pudiendo compartir a diario con las personas que quiero y amo.

Agradezco a mi bella madre, a mi hermana, a mi enamorada por apoyarme a lo largo de todo de este camino sin dejar que me derrumbé o me desvié de mis objetivos.

A mis amigos y compañeros que compartimos gratos y amenos momentos donde se formaron verdaderas amistades.

A la Universidad de las Fuerzas Armadas ESPE sede Latacunga por la formación personal y profesional y a todos sus docentes, con especial agradecimiento a mi director de proyecto, al PhD. Santiago Jácome por hacer posible este trabajo.

Jonathan Franco

En primer lugar, me gustaría agradecer a Dios por permitirme vivir cada día con las personas que quiero, mi familia, mis amigos.

Segundo quiero agradecer a mi madre a todas sus bendiciones diarias que a lo largo de mi vida me protegen y me ayudan a ser mejor.

A la Universidad de las Fuerzas Armadas Espe por permitirme crecer académicamente y a sus docentes que impartieron sus conocimientos, en especial al PhD. Santiago Jácome por su guía.

Gracias a todos mis amigos que fueron un gran apoyo a lo largo de la carrera, compartimos tantos buenos momentos juntos.

Benjamín Sánchez

Tabla de contenidos

Índice de tablas

Índice de figuras

Resumen

El presente proyecto de investigación está orientado al desarrollo de un Lenguaje de Dominio Específico (DSL) para programar una placa Arduino UNO utilizando el paradigma de Ingeniería Dirigida por Modelos (MDE) a partir de la transformación de un modelo a texto, siendo en este caso código Arduino, para lograr este proceso se empieza con la definición de la sintaxis abstracta, luego se define la sintaxis concreta y por último se genera código Arduino a través de plantillas personalizadas en Acceleo, de esta manera a través de la herramienta se puede simular circuitos lógicos combinados con compuertas lógicas básicas como son AND, NOT y OR, solventando y automatizando el proceso de programación manual mediante el IDE de Arduino. La presente investigación se conforma de 4 etapas fundamentales: En la primera etapa se plantea el problema de investigación. En la segunda etapa se fundamenta el marco teórico para el desarrollo de un DSL para programar la placa Arduino UNO que contempla software implementado, la parte de hardware donde se aplica y la metodología implementada. La tercera etapa consta del desarrollo del DSL empezando por la definición de la sintaxis abstracta a través de Ecore, la definición de la sintaxis concreta a través de Sirius y por último la generación de código automático con ayuda de Acceleo. La cuarta etapa consiste en validar los resultados obtenidos del DSL desarrollado en el laboratorio de Comunicaciones perteneciente a la Universidad de las Fuerzas Armadas ESPE Sede Latacunga, comparándolos con los indicadores planteados.

Palabras clave:

- **INGENIERÍA DIRIGIDA POR MODELOS**
- **LENGUAJE DE DOMINIO ESPECÍFICO**
- **ARDUINO**
- **TRANSFORMACIÓN DE MODELO A TEXTO**
- **ECLIPSE**

Abstract

The present research project is oriented to the development of a Domain Specific Language (DSL) to program an Arduino UNO board using the Model Driven Engineering (MDE) paradigm from the transformation of a model to text, being in this case Arduino code, to achieve this process starts with the definition of the abstract syntax, Then the concrete syntax is defined and finally Arduino code is generated through predefined templates by Acceleo, in this way through the tool you can simulate logic circuits combined with basic logic gates such as AND, NOT and OR, solving and automating the manual programming process through the Arduino IDE. This research consists of 4 fundamental stages: The first stage consists to of the statement of the problem. The second stage raises the theoretical framework for the development of a Domain Specific Language for programming the Arduino UNO board that includes implemented software, the hardware part where it is applied and the implemented methodology. The third stage consists of the development of the Domain Specific Language starting with the definition of the abstract syntax through Ecore, the definition of the concrete syntax through Sirius and finally the automatic code generation with the help of Acceleo. The fourth stage consists of validating the results obtained from the DSL developed in the Communications laboratory belonging to the Universidad de las Fuerzas Armadas ESPE sede Latacunga, comparing them with the proposed indicators.

Key words:

- **MODEL DRIVEN ENGINEERING**
- **DOMAIN SPECIFIC LANGUAGE**
- **ARDUINO**
- **MODEL TO TEXT TRANSFORMATION**
- **ECLIPSE**

Capítulo I

1. Planteamiento del Problema

En el presente capitulo se abordan temas como planteamiento del problema, justificación e importancia, definición de objetivos tanto generales como específicos, variables de investigación implementadas y la hipótesis a contrastar.

1.1. Antecedentes

Actualmente la producción de software en gran mayoría aún es desarrollada por procesos manuales, casi desde cero, comparado con labores de artesanía, esto produce como consecuencia un desarrollo lento y caro con problemas de mantenimiento. (Greenfield, Jack and Short, Keith, 2003) señala que “la práctica ha demostrado que el clásico enfoque artesanal no funciona a gran escala, es beneficioso adoptar y adaptar los patrones de industrialización que han funcionado en otras industrias”.

(S. Bhardwaj and P. Larbig and R. Khondoker and K.Bayarou, 2017) afirma que “con la adopción de la red de comunicaciones tecnologías como el Internet de las Cosas (IoT) la Industria 4.0 está cambiando el enfoque de los sistemas industriales hacia la automatización, la tecnología y el intercambio de datos”.

El término industria 4.0 es nuevo, fue introducido por primera vez en 2011 como un proyecto alemán, pero antes hubo otras tres revoluciones industriales desarrolladas en siglo XVIII, XIX y XX, respectivamente. Debido a esta revolución industrial, hay mejoras en costes y rendimiento, pero también hace que estas industrias sean vulnerables a los ciberataques (S. Bhardwaj and P. Larbig and R. Khondoker and K.Bayarou, 2017).

En el inicio de la era de la automatización industrial, el diseño de sistemas automáticos para el control de procesos se caracterizaba por el planeamiento de una arquitectura centralizada y jerárquica, sin embargo, a través del tiempo con la aparición de nuevas y mejoradas tecnologías se ha obtenido que este tipo de ideas han ido mejorando en favor de tener sistemas más robustos y dinámicos (Papp, Jozsef and Tokody, Daniel and Flammini, Francesco, 2018).

En 2005 los estudiantes del instituto IVRAE Massimo Banzi proponen desarrollar “una placa de microcontroladores de bajo costo que permita incluso a un novato hacer cosas realmente asombrosas donde se pueda conectar un Arduino a todo tipo de

sensores, luces, motores y otros dispositivos" (Kushner, 2011). Por la flexibilidad de la placa Arduino que esta ofrece, se puede realizar desde relojes hasta básculas.

La Ingeniería Dirigida por Modelos (MDE) por sus siglas en ingles Model-Driven Engineering, surge como la respuesta de la Ingeniería de Software a la industrialización del desarrollo de software. Rube en (Rube, 2012) respecto a MDE, señala que "es una disciplina dentro de la Ingeniería del Software que tiene por objetivo dar soporte a las actividades del ciclo de vida del software, utilizando los modelos como principal artefacto". El uso de MDE tiene como finalidad aumentar la productividad al máximo, simplificando el proceso de diseño e incrementando el valor de los artefactos software como los modelos.

Al presente existen dos tendencias de MDE. La una mediante el empleo de los principios de Arquitectura Dirigida por Modelos (MDA), del inglés Model-Driven Architecture que se encuentra basada en el Lenguaje de Modelado Unificado (UML), mientras que la otra es mediante la utilización de Lenguajes de Dominio Específico (DSL) por sus siglas en ingles Domain-Specific Language, los cuales mejoran de forma audaz la eficiencia y la calidad, de los sistemas desarrollados al proporcionar abstracciones adecuadas, que reflejen los conceptos del dominio de la empresa (D. Ratiu and V. Pech and K. Dummann, 2017), pero "existe una gran variedad de DSL, que satisfacen las necesidades de diferentes negocios donde los ingenieros de software quieren solventar estas carencias construyendo DSL para simplificar su trabajo de desarrollo" (L. Bambaci and F. Boschetti and R. Del Gratta, 2018).

Se ha investigado la literatura existente sobre la evolución del DSL, para sorpresa no se pudo encontrar ningún documento que describiera dicha evolución. Pero (Fowler, 2010) afirma que "es difícil encontrar mucha información sobre cómo trabajar con ellos. Los Lenguajes de Dominio Específico son pequeños lenguajes, centrados en un aspecto concreto de un sistema de software. No se puede construir todo un programa con un DSL, pero a menudo se utilizan varios DSL en un sistema escrito principalmente en un lenguaje de propósito general", por lo cual se puede intuir que están desde el comienzo de la computación.

1.2. Planteamiento y formulación del problema

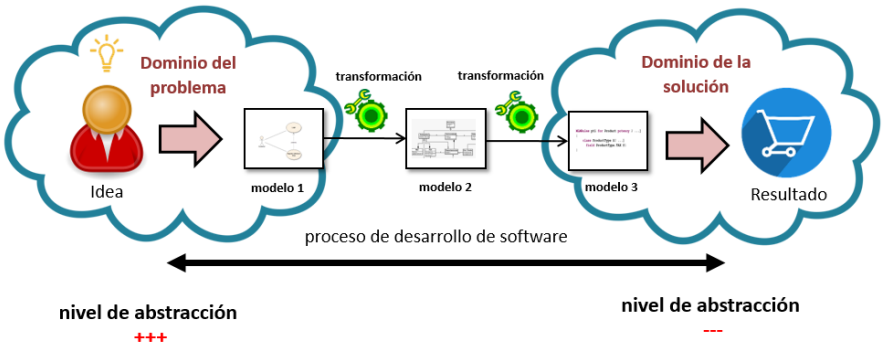
El desarrollo de software mediante MDE "es una propuesta para la construcción de software en el cual se atribuye a los modelos el papel principal de todo el proceso,

frente a las propuestas tradicionales basadas en lenguajes de programación, plataformas de objetos y componentes software" (Cueva Lovelle & García Bustelo, 2008).

Un DSL es un lenguaje de modelado con un nivel superior de abstracción optimizado para una clase específica de problemas usando conceptos y reglas de su campo y dominio. En la Figura 1, se ilustra el proceso de desarrollo de software a través del paradigma MDE con DSL.

Figura 1

Visión simplificada del proceso de desarrollo de Software usando MDE.



Nota. Tomado de (Jácome Guerrero, Propuesta de mecanismos de personalización de meta-modelos en la Ingeniería Dirigida por Modelos, 2019)

(Bézivin, 2004) señala que MDE "considera el uso sistemático de modelos en la cadena de producción de software". Un modelo es una descripción de un sistema o parte de éste, escrito en un lenguaje de modelado con sintaxis y semántica precisas, pudiendo ser interpretado automáticamente por un computador (Kleppe, Anneke G and Warmer, Jos and Warmer, Jos B and Bast, Wim, 2003). MDE considera que se debe mover el foco de desarrollo de software desde el código hacia los modelos, hasta el punto de construir modelos que puedan ser directamente compilados, transformados y ejecutados. Muy frecuentemente, MDE utiliza los DSL para la creación de modelos. Un DSL es un lenguaje especializado en modelar o resolver un conjunto específico de problemas. Es decir, un DSL se ajusta a una tarea particular en el dominio de la aplicación.

En MDE, los lenguajes de modelado, tanto de propósito general como de dominio específico, se especifican a través de su sintaxis abstracta, sintaxis concreta y semántica (Henderson-Sellers, B and Gonzalez-Perez, C, 2006) y (VÖlter Markus and Stahl,

Thomas and Bettin, Jorn and Haase, Arno and Helsen, Simon, 2013). La sintaxis abstracta está definida por un metamodelo, que describe los conceptos, propiedades, relaciones y restricciones relevantes de un determinado dominio. La sintaxis concreta está dada por la notación textual y/o gráfica utilizada para representar los elementos del metamodelo. Esta es la notación que utilizará el usuario del lenguaje para diseñar modelos. Un modelo es una instancia de su metamodelo (Bézivin, 2004). Por tanto, un metamodelo determina un conjunto posiblemente infinito de modelos válidos. La semántica del DSL está dada por el significado que expresa el modelo con relación al sistema modelado (sistema real). Un buen DSL permite especificar modelos cuyo grado de abstracción es tan cercano al problema como sea posible, y a su vez facilita el uso de los modelos para el propósito inicial (simulación, generación de código, etc.).

El mundo se encuentra controlado por todo tipo de dispositivos, que generan avances en la tecnología aumentando la productividad e incrementando las oportunidades de negocio por lo que (F. Rosique and B. Álvarez and P. Sánchez and J. A. Pastor, 2016) afirman que “Un mundo físico cada vez más digitalizado, plagado de sensores e instrumentos digitales, así como de dispositivos de control formando sistemas complejos, pueden ser accesibles desde cualquier lugar, en cualquier momento y a través de cualquier dispositivo”. Ante la evolución que se vive surgen soluciones que respaldan estas tendencias globales. Incluyendo la transformación de la tercera dimensión, el Internet de las Cosas (IOT), la Inteligencia Artificial entre otras.

Los DSL son implementados en diversas áreas tecnológicas, por ejemplo, un DSL usado por más de 140 plantas de energía pertenecientes a compañías eléctricas europeas que sirve para (Soberning, Strembeck, & Beck, 2019) “proporcionar un modelo de programación estandarizado para especificar los horarios reduciendo la redundancia de código permitiendo a los expertos en dominios (programadores) establecer y cambiar las definiciones de mercado de forma autónoma”.

En el continente americano se muestra el uso en “un lenguaje de programación de dominio específico llamado Eugene, destinado a encapsular partes biológicas, dispositivos y reglas que allanan el camino para la exploración espacial de diseño, la simulación y el ensamblaje automatizado” (Bilitchenko, y otros, 2011).

El desarrollo de un DSL indiferentemente del área a ser aplicado, ayuda al experto a realizar sus tareas de forma más rápida. La programación de la placa Arduino es la

programación de un microcontrolador, que consiste en traducir a líneas de código tareas automatizadas a través de sensores y en función de las condiciones del entorno. La automatización de este proceso por medio de una forma más simple es desarrollando un DSL. Un lenguaje que aporta semántica a la necesidad que debe ser resuelta con la ayuda del propio lenguaje de manera gráfica, de esta manera se da prioridad a otras tareas.

¿Cómo desarrollar un DSL que permita optimizar el tiempo de programación de código funcional para la placa Arduino utilizando compuertas lógicas “AND”, “OR” y “NOT”?

1.3. Justificación e importancia

Los DSL mejoran de forma audaz la eficiencia y la calidad de los sistemas desarrollados al proporcionar abstracciones adecuadas que reflejen los conceptos de la empresa dominio” (D. Ratiu and V. Pech and K. Dummann, 2017) pero “hay una gran variedad de DSL, que satisfacen las necesidades de diferentes negocios. Los ingenieros de software quieren construir sus propios DSL para simplificar su trabajo de desarrollo” (L. Bambaci and F. Boschetti and R. Del Gratta, 2018).

(L. Kumar and R. Jetley and A. Sureka, 2016) “Los controladores lógicos programables (PLC) son sistemas de control utilizados para la automatización industrial y de fábricas de procesos electromecánicos. Los PLC se programan utilizando lenguajes de dominio específico”. Ahora bien, existe software propio para el manejo de PLC, con código cerrado, dada esta razón se propone utilizar la placa Arduino ya que “Gracias a su experiencia de usuario sencilla y accesible, Arduino se ha utilizado en miles de proyectos y aplicaciones diferente” (Arduino, 2020), además “Arduino es una plataforma open source creada con el objetivo de ser funcional, fácil de programar y a bajo costo, así llegando a públicos tales como estudiantes y proyectistas aficionados” (Arduino, 2020).

Entre las ventajas que ofrece un DSL se encuentran la solución a problemas en términos abstractos, es decir facilitar la comunicación con personas que no conozcan muchos detalles de la lógica o construcción del lenguaje, así puede aumentar la calidad del producto creado: menos errores, mejor conformidad arquitectónica, mayor facilidad de mantenimiento.

Desde el lanzamiento de la iniciativa MDA por el Object Management Group (OMG) en noviembre de 2000, el interés por el desarrollo de software dirigido por modelos

es cada vez mayor, tanto por parte de la comunidad académica como de la industria. (García Molina, y otros, 2012) describen que:

"El desarrollo de MDE también tiene como objetivo elevar el nivel de abstracción con respecto al uso de lenguajes de programación lo que también conlleva un incremento de la automatización, a partir de modelos de alto nivel, que son expresados en algún lenguaje de modelado o lenguaje de un dominio (DSL), se genera código de la aplicación final".

Una mayor abstracción y automatización son claves para dominar la complejidad inherente al proceso de construcción de software, y con los modelos se pretenden obtener todas las ventajas que en su momento se consiguieron con los lenguajes de programación.

El presente proyecto de investigación propone el desarrollo de una herramienta gráfica para circuitos lógicos básicos, utilizando un DSL, con la finalidad de reducir tiempo en la programación de aplicaciones para la placa Arduino. De esta manera brindar una herramienta sencilla sin tener que detallar el lenguaje de programación propio de Arduino, es necesario tener conocimientos básicos sobre componentes electrónicos para poder desarrollar aplicaciones.

Dado que no se posee experiencia previa en el desarrollo de lenguajes de dominio específico, se plantea el uso de la metodología ágil SCRUM, puesto que es ideal al ser altamente flexible para la entrega de requisitos cambiantes, al ser iterativo incremental se pueden dividir los requisitos en algo más simple para agilitar su desarrollo, de esta manera se permite realizar pruebas funcionales en cada iteración.

1.4. Objetivos Generales y Específicos

1.4.1. Objetivo General

Desarrollar un DSL que permita optimizar el tiempo de programación de la placa Arduino UNO utilizando compuertas lógicas "AND", "OR" y "NOT".

1.4.2. Objetivos Específicos

- Formular el marco teórico para el desarrollo de un DSL que permita optimizar el tiempo de programación de la placa Arduino UNO utilizando compuertas lógicas “AND”, “OR” y “NOT”.
- Desarrollar el DSL para optimizar el tiempo de programación de código funcional para la placa Arduino UNO utilizando compuertas lógicas “AND”, “OR” y “NOT”.
- Definir la sintaxis abstracta (Metamodelo) del DSL para definir la estructura del dominio del lenguaje.
- Definir la sintaxis concreta (Representación gráfica) del DSL para especificar la notación especifica con la que los usuarios interactuarán.
- Desarrollar el generador de código a partir del DSL a código para programar la placa Arduino UNO comparando con el código realizado de forma tradicional a través del IDE de Arduino.
- Desarrollar un DSL que cumpla con las métricas de Usabilidad según el estándar ISO/IEC 9126.
- Validar los resultados obtenidos del DSL teniendo un enfoque hacia el cumplimiento de los indicadores señalados, para corroborar que el DSL genera código funcional para programar la placa Arduino UNO utilizando compuertas lógicas “AND”, “OR” y “NOT”.

1.5. Metas

- Se desarrollará el DSL para optimizar el tiempo de programación de la placa Arduino UNO utilizando compuertas lógicas “AND”, “OR” y “NOT”.
- Se definirá la sintaxis abstracta (Metamodelo) del DSL a través de Ecore.
- Se definirá la sintaxis concreta del DSL utilizando Sirius.
- Se desarrollará el generador de código a partir del modelo generado por el usuario al código de la placa Arduino utilizando Acceleo.
- Se validará los resultados obtenidos del DSL teniendo un enfoque hacia el cumplimiento de los indicadores señalados, para corroborar que el DSL genera código funcional para programar la placa Arduino UNO utilizando compuertas lógicas “AND”, “OR” y “NOT”.

1.6. Hipótesis

Si se desarrolla un DSL entonces optimizará el tiempo de programación de código funcional para la placa Arduino UNO utilizando compuertas lógicas “AND”, “OR” y “NOT”.

1.7. Variables de la Investigación

1.7.1. Variable Dependiente

Se optimiza el tiempo de programación de código funcional para la placa Arduino UNO utilizando compuertas lógicas “AND”, “OR” y “NOT”.

1.7.2. Variable Independiente

Se desarrolla un DSL.

1.8. Indicadores

- Tiempo de elaboración del código funcional para la placa Arduino UNO a través de IDE Arduino debe ser mayor que el proceso a través del DSL.
- Número de líneas de código generadas por el DSL debe ser menor en comparación a líneas de código generadas de forma manual.
- Las líneas de código generadas por parte de DSL deben ser igual o menor a las líneas generadas por un Software similar.
- Un metamodelo que permita definir la sintaxis abstracta del DSL.
- Una sintaxis concreta del DSL.
- Un generador de código a partir del modelo creado por el usuario al código para la placa Arduino UNO.

Capítulo II

2. Marco Teórico

En el presente capítulo se describe la conceptualización teórica utilizada como base para la presente investigación. Aspecto importante por ser una referencia para delimitar el problema, plantear definiciones, fundamentar la hipótesis que posteriormente tendrán que verificarse a través de los indicadores planteados.

2.1. Ingeniería Dirigida por Modelos (MDE)

(Schmidt, 2006). “MDE es una manera para manejar la complejidad a la que se enfrenta la industria del desarrollo de software al proporcionar mejores técnicas de abstracción y facilitar la automatización”.

MDE para (Rube, 2012) “es una disciplina dentro de la Ingeniería del Software que tiene por objetivo dar soporte a las actividades del ciclo de vida del software, utilizando los modelos como principal artefacto”. El criterio fundamental de MDE es la utilización de la abstracción, que permite construir modelos que representan el sistema a desarrollar. En el contexto de MDE el término abstracción especifica la medida en que un modelo o lenguaje de modelado está orientado a la tecnología de la solución. Un nivel bajo de abstracción significa que está orientado a la solución utilizando tecnologías relacionadas a la implementación de la aplicación. En contra, un nivel alto de abstracción significa que está orientado al problema, abordándose el aspecto de “qué” es hecho por el sistema, es decir que se centra en la especificación funcional del sistema, sin preocuparse por el aspecto de “cómo” se realizará.

En MDE, los modelos se utilizan para abordar el dominio de un tema, de esta manera, se puede definir MDE “como un enfoque de desarrollo de software que se fundamenta en la separación conceptual entre la especificación funcional de un sistema y su implementación, en lo que cobran vital importancia los modelos y las transformaciones entre ellos” (Jácome Guerrero, Propuesta de mecanismos de personalización de meta-modelos en la Ingeniería Dirigida por Modelos, 2019).

2.1.1. Modelo

Para (García Molina, y otros, 2012) “un modelo es una simplificación de la realidad, como resultado de un proceso de abstracción, y ayuda a comprender y razonar sobre esa realidad”. En el caso del software, es la descripción de un aspecto de un

sistema escrita en un lenguaje bien definido, si un modelo debe ser interpretado por una máquina para generar código o controlar un sistema, deberá estar expresado con una notación descrita formalmente, por ejemplo, mediante un metamodelo.

2.1.2. Metamodelo

Para (Henderson-Sellers, B and Gonzalez-Perez, C, 2006) “El metamodelo es una abstracción que destaca las propiedades del modelo, especifica conceptos de un lenguaje, reglas y las relaciones entre ellos, cada modelo es una instancia de su metamodelo”.

Los usos más comunes para metamodelos son:

- Como un esquema para los datos semánticos, que necesitan ser intercambiados y almacenados.
- Como un lenguaje que soporta un método o proceso en particular.
- Como un lenguaje para expresar semántica adicional de la información existente.
- Como mecanismo para crear herramientas que funcionan con una amplia clase de modelos en tiempo de ejecución.
- Como un esquema para modelar y explorar automáticamente oraciones de una lengua, con aplicación a la síntesis de pruebas automatizadas.

2.1.3. Desarrollo de Software Dirigido por Modelos (MDSD)

La transformación de modelos son el eje principal del desarrollo de software dirigido por modelos. Los modelos son la más alta abstracción que pueden evolucionar aplicándoles un proceso de transformación en diseños específicos.

Los dominios específicos se basan en el modelado, modelos, y en MDA, enfoques impulsados por la MDE. Enfoques que tienen como fin solventar un problema mediante herramientas que operan entre sí. (Bettin, 2004) “El desarrollo de software basado en modelos (MDSD) es un nuevo paradigma para el desarrollo de software enfocado para equipos de proyectos distribuidos que involucran a más de 20 personas, con raíces en la Ingeniería de software que es la disciplina de diseñar y construir familias de aplicaciones para un propósito específico o segmento de mercado”.

(Gronback, 2009) opina que “La idea actual de MDSD es centrarse en desarrollar y perfeccionar el modelo de un dominio en particular para proporcionar un vocabulario estándar para su uso en desarrollo”, MDSD puede hacer uso de demasiados enfoques y tecnologías, ya sean que estén basadas en estándares o no.

2.1.4. Meta-Object Facility (MOF)

Es un estándar del Object Management Group (OMG) sobre MDE. Su propósito es proporcionar un sistema de tipos para entidades en la arquitectura CORBA y un conjunto de interfaces a través de las cuales esos objetos se pueden crear y manipular. MOF es un lenguaje orientado a objetos similar a UML. La relación entre un modelo y un metamodelo es similar entre un objeto y clase, en el sentido que un modelo es una instancia de un metamodelo.

2.2. Lenguaje de Dominio Específico (DSL)

Un DSL es un lenguaje de modelado, gráfico o textual, que se usa para describir una determinada parcela de la realidad. (García Molina, y otros, 2012) aseveran que un DSL está compuesto de tres elementos principales:

> “La sintaxis abstracta que define los conceptos del lenguaje y las relaciones entre ellos, así como las reglas que establecen cuando un modelo está bien formado; la sintaxis concreta que establece la notación, y la semántica que normalmente es definida a través de la traducción a conceptos de otro lenguaje destino cuya semántica se conoce (por ejemplo, un lenguaje de programación como Java)” (p.58)

(Gronback, 2009) afirma que “Un DSL es un lenguaje diseñado para ser útil en un conjunto especifico de tareas, la especificidad del dominio está determinada por el diseñador del idioma”. Gronback plantea un ejemplo utilizando UML, el cual puede ser considerado como un lenguaje de propósito general que consta de varios lenguajes de dominio especifico como casos de uso, definición estructural, etc. Aunque también se pude considerar todo UML como un DSL, ya que cubre el dominio del desarrollo de software.

En MDE, la sintaxis abstracta de un DSL se define mediante un metamodelo, esto es, un modelo conceptual del DSL expresado con un lenguaje de meta modelado junto

con un conjunto de reglas que definen restricciones adicionales, para que un modelo se considere bien formado.

Un buen enfoque es hacer que el lenguaje de modelado sea muy específico con un marco estable, es decir comenzar poco a poco incluyendo justo lo que se necesita. Es decir, se puede ahorrar en recursos ya que con un lenguaje de propósito general el nivel de inversión es complejo y extenso, esto incluye adaptarse a las herramientas asociadas y luego integrarlas en un proceso de desarrollo que, por lo general es muy costoso, por lo tanto, cuando el metamodelo es ventajoso, pero el uso de lenguajes de modelado basados en estándares no, la alternativa es usar herramientas que faciliten la creación de un DSL.

2.2.1. Evolución DSL

Muchos lenguajes informáticos pertenecen a un dominio específico en lugar de un propósito general. “Los lenguajes de dominio específico también se denominan lenguajes orientados a aplicaciones, especiales de propósito, de tarea específica, orientada a problemas, especializada o de aplicación”. (Mernik, Marjan and Heering, Jan and Sloane, Anthony M, 2005) con esto en mente los DSL se encuentran en un dominio más limitado por notaciones propias, que permiten obtener ganancias en cuanto a la expresividad y facilidad de uso, comparados con lenguajes de dominio general.

“Hay varios problemas para escribir sobre la historia de los lenguajes de programación” (Sammet, 1972), a su vez el lenguaje de dominio específico no es un término tan nuevo, “fue desarrollado en 1957-1958 para la programación de máquinas” (Mernik, Marjan and Heering, Jan and Sloane, Anthony M, 2005) y ha recibido atención desde finales de 1960.

“La mayoría de los enfoques de ingeniería hacen hincapié en la modelización de dominios como un mecanismo importante para el desarrollo de familias de productos”. (Estublier, Jacky and Vega, Germán and Ionita, Anca, 2005) esto permite simplificar la abstracción del problema, proponiendo construcciones directamente relacionadas con los conceptos de dominio de aplicación, dando un mejor apoyo a profesionales. “En el año 2001, el OMG estableció a la MDA, como una arquitectura para el desarrollo de sistemas, representa un nuevo paradigma de desarrollo software en los que los modelos guían todo el proceso de desarrollo” (Carrillo Guambo, Carlos Enrique, 2006).

MDE tiene muchas similitudes con el DSL, pero, se centra en las relaciones que existen entre el modelo y el sistema de modelado, "una simplificación de un sistema construido con un objetivo previsto en mente. El modelo debe ser capaz de responder a las preguntas en lugar del sistema real " (Estublier, Jacky and Vega, Germán and Ionita, Anca, 2005).

2.2.2. Sintaxis Abstracta

La sintaxis abstracta es la representación textual del DSL mejor asociado como un metamodelo, que representa la estructura de los elementos y relaciones del dominio. (Pavlich-Mariscal, Veliz-Quispe, Demurjian, & Michel, 215) “EMF provee un lenguaje, llamado Ecore, para la definición de la sintaxis abstracta de modelos”.

2.2.3. Sintaxis Concreta

Una vez definida la sintaxis abstracta, el siguiente paso es la definición de la sintaxis concreta que mapea cada uno de los elementos del metamodelo a una representación textual o gráfica.

La sintaxis concreta requiere la representación del modelo de acuerdo con la estructura del metamodelo, ya sea sintaxis concreta textual y/o gráfica, y para eso existen algunas herramientas que pueden colaborar para lograr este objetivo, entre las herramientas para representaciones graficas están Graphiti y/o Sirius y para la representación textual existen herramientas como Xtext, EMFText y TCS.

La representación gráfica de un DSL se realiza mediante figuras que se mostrarán en los diagramas, paleta de herramientas del editor y la relación que existe entre las clases del metamodelo. (Pavlich-Mariscal, Veliz-Quispe, Demurjian, & Michel, 215) “La Sintaxis Concreta especifica cómo representar visualmente los modelos a través de los Diagramas”.

La representación textual proporciona un lenguaje de especificación de sintaxis a partir de un modelo del cual se pueden generar editores y componentes textuales a través de reglas que se pueden generar con las propias herramientas.

2.3. Herramientas

Existen muchas herramientas y tecnologías que se pueden utilizar para desarrollar con el paradigma de MDE, en esta sección se describen las principales herramientas

utilizadas y otras parecidas para según su conveniencia utilizarlas a lo largo del proyecto del desarrollo del DSL.

2.3.1. Eclipse IDE

Es un entorno de desarrollo integrado (IDE) que permite el diseño de aplicaciones combinando herramientas para desarrolladores en una interfaz gráfica de usuario (GUI). Eclipse permite la instalación de múltiples plug-in para integrar diversas herramientas necesarias en el proyecto que se está trabajando (Francois, y otros, 2012).

2.3.2. Plug-in

Los plug-in son pequeños programas complementarios que ayudan a la construcción del producto en desarrollo, los cuales pueden ser instalados a través del Marketplace de Eclipse.

2.3.3. Eclipse Modelling Framework (EMF)

Eclipse Modeling Framework (EMF) es una implementación del estándar de MOF. Es un framework de modelado que facilita la generación de código para construir aplicaciones basadas en modelos de datos estructurados. EMF proporciona herramientas y soporte en tiempo de ejecución para producir un conjunto de clases Java para el modelo, junto con un conjunto de clases que permiten la visualización, edición del modelo, creación y manipulación de instancias del metamodelo. (Gronback, 2009) propone que “EMF proporciona una base sólida para el desarrollo de la sintaxis abstracta e incluso incluye una sintaxis concreta basada en XML, tosca pero eficaz”.

EMF es una implementación de MOF (Meta-Object Facility) por lo cual (Francois, y otros, 2012) afirman que la implementación de EMF “facilita la construcción de herramientas y otras aplicaciones basadas en modelos de datos estructurados. Desde un modelo específico. EMF proporciona herramientas y soporte en tiempo de ejecución para crear un Lenguaje de Dominio Específico (DSL)”.

2.3.4. Eclipse Modelling Project

Eclipse Modelling Project se centra en la evolución y promoción de tecnologías de desarrollo basadas en los modelos al proporcionar un conjunto unificado de estructuras de modelado, herramientas e implementaciones de estándares.

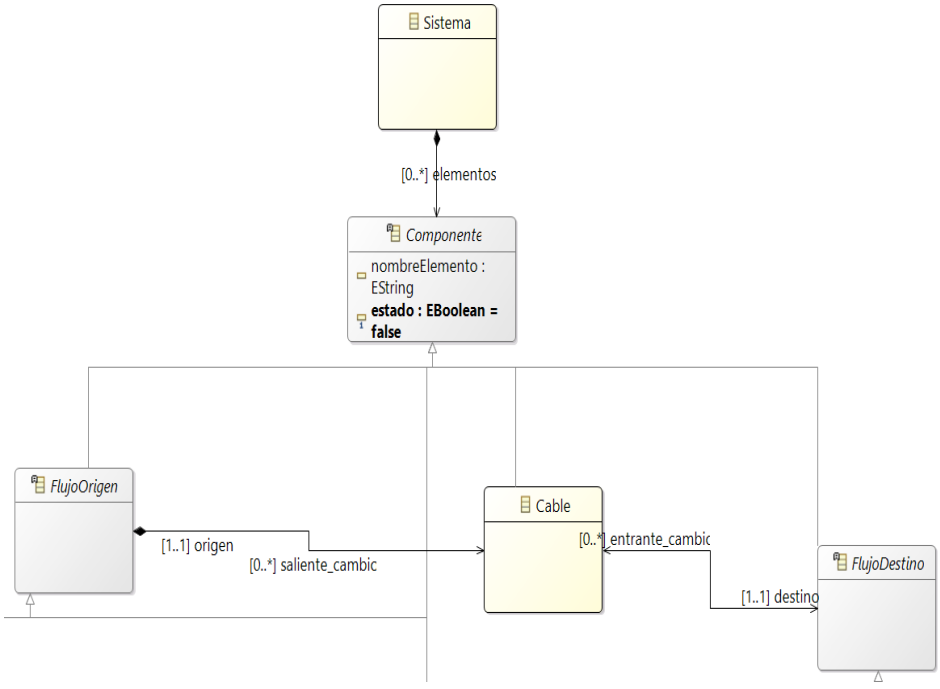
2.3.5. Ecore

Es una parte de la arquitectura de meta modelado de EMF que proporciona a Eclipse para crear entornos y herramientas a través de MDE. Ecore fue diseñado como un subconjunto de MDE con las construcciones básicas para construir metamodelos. Ecore es un lenguaje metamodelo que proporciona conceptos orientados a los objetos para la creación de metamodelos: clases, atributos y referencias; excluye las asociaciones. En un metamodelo a veces es necesario incluir reglas que determinan cuando un modelo está bien formado, esto es logrado con las reglas del estándar Meta Object Facility (MOF).

El marco central de EMF incluye un metamodelo (Ecore) para describir modelos y sus tiempos de ejecución, incluida la notificación de cambios, el soporte de persistencia con serialización XMI muy eficiente para manipular objetos de forma genérica.

Figura 2

Metamodelo



XMI

(Kovse & Harder, 2002) “XML-based Metadata Interchange (XMI) es un formato de intercambio de la metadata que está definida en términos del estándar Meta Object Facility (MOF)”.

2.3.6. Lenguaje de Restricciones de Objetos

El lenguaje de Restricciones de Objetos (OCL del inglés Object Constraint Languaje) es un lenguaje para la descripción formal de expresiones. (Richters & Gogolla, 2002) afirman que OCL “permite formalmente especificar restricciones en un modelo UML. OCL hace que el significado de las restricciones sea preciso y ayuden a eliminar ambigüedades e inconsistencias”. Lo que quiere decir que, OCL es un lenguaje que actúa sobre un modelo que presente ambigüedad o inconsistencias para especificar características adicionales haciendo su evaluación.

OCL puede especificar acciones o expresiones que, cuando se ejecutan, alteran el estado del modelo especificando restricciones específicas que ayudan a mejorar el modelo, para así no tener fragilidades en su uso.

2.3.7. Sirius

Es un proyecto Eclipse que permite fácilmente crear su propio modelo gráfico de trabajo en la playa aprovechando Tecnologías de modelado de Eclipse, incluyendo EMF y marco de modelo gráfico (GMF). Proporciona un marco de trabajo genérico para MDE, permite a los usuarios crear, editar y visualizar modelos EMF. Puede ser fácilmente adaptado a cualquier necesidad específica basada en punto de vista y está respaldada por la norma ISO/IEC 42010 la cual especifica que la misma información puede ser representada de forma sincronizada en los diagramas, mesas y árboles. Sirius se basa en Acceleo, y otros proyectos, para facilitar el establecimiento de relaciones entre los datos del modelo y su representación gráfica.

Los principales conceptos de Sirius según su documentación oficial (Sirius, s.f.).

Son:

- Punto de vista (View Point): representa a un conjunto de especificaciones y extensiones, es considerada uno de los elementos fundamentales.
- Representación (Representation): Conjunto de elementos gráficos que representan los datos del dominio, es decir la instancia del metamodelo Ecore.
- Mapeado (Mapping): Identifica un subconjunto de elementos del modelo semánticos que parecerían en la representación, a su vez se utiliza para indicar como deberían ser representados.

- Estilos (Styles): Se emplea para personalizar la apariencia de elementos definidos
- Herramientas (Tools): Agrega capacidades de edición al editor gráfico, para permitir a los usuarios finales crear, editar y eliminar elementos del modelo.

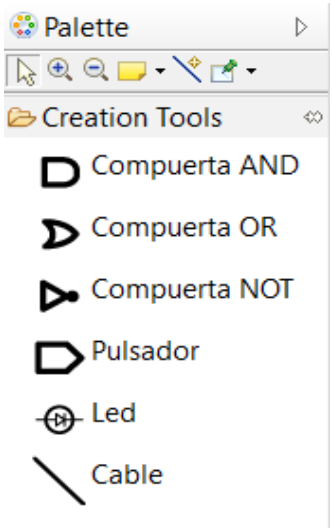
Cuando se definen elementos, bordes y herramientas en Sirius, se requiere de un intérprete de expresiones, como recomendación se utiliza Aceleo Query Languaje (AQL), porque permite crear consultas para seleccionar elementos o expresiones de propósito general para trabajar con un valor de una instancia, también permite navegar y consultar a través de un modelo EMF definido.

Otras expresiones que Sirius soporta son:

- Var: Permite obtener el valor que es contenido por la variable.
- Feature: Dispone acceso a la característica nombrada del elemento actual.
- Service: Invoca un método del elemento.

Figura *3*

Paleta de figuras creada con Sirius



Nota. Paleta creada para el DSL desarrollado.

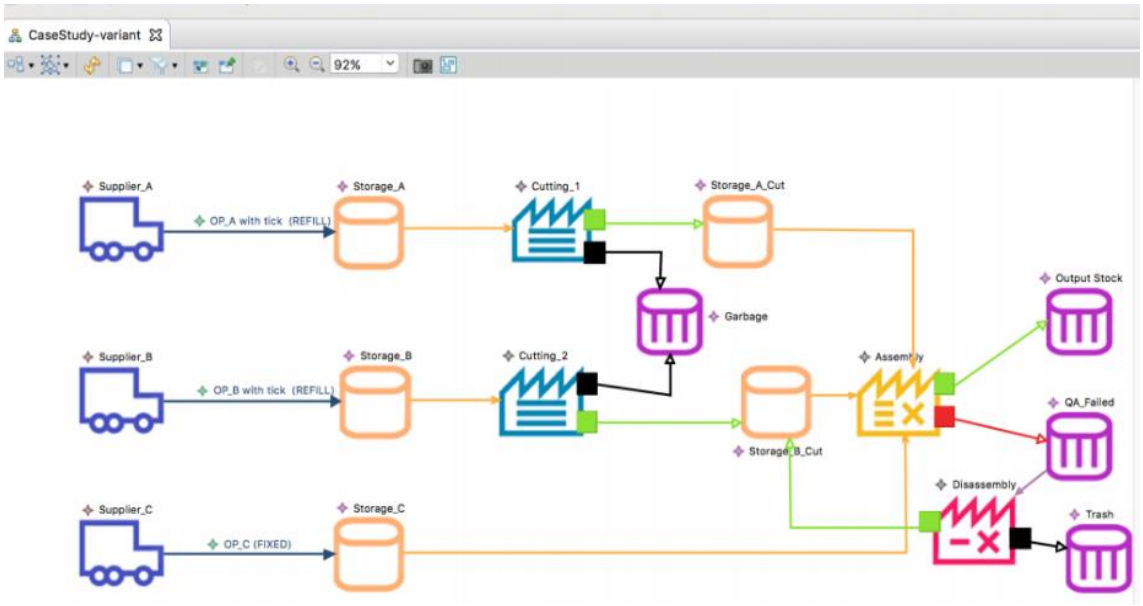
(Arkadiusz, 2020) “Sirius es una herramienta basada en modelos desarrollada por Obeo. Es una herramienta gráfica donde el usuario puede editar las propiedades de los diagramas y otras visualizaciones”. Dado que el paradigma MDE es desarrollar

aplicaciones de un dominio específico, esta herramienta se presta en gran medida para el diseño de sistemas complejos. En la figura 4 se presenta un diagrama elaborado con Sirius.

(Arkadiusz, 2020) “Sirius es una herramienta basada en modelos desarrollada por Obeo. Es una herramienta gráfica donde el usuario puede editar las propiedades de los diagramas y otras visualizaciones”. Dado que el paradigma MDE es desarrollar aplicaciones de un dominio específico, esta herramienta se presta en gran medida para el diseño de sistemas complejos. En la figura 4 se presenta un diagrama elaborado con Sirius.

Figura 4

Ejemplo de Diagrama en Sirius



Nota. Tomado de https://www.eclipse.org/sirius/gallery.html

La figura 4 muestra los elementos que posee el editor gráfico de Sirius, definidos a continuación:

Nodos: Se utilizan para representar los modelos, pude ser representado por una imagen o una figura geométrica, se definen las siguientes propiedades:

- ID: Es el identificador único de la clase a definir.

- Domain class: Define el tipo de elementos a ser representados en el nodo, es importante prevenir posibles conflictos con otros metamodelos que pudiesen ser definidos con el mismo nombre.
- Semantic candidate expression: Restringe a la lista de elementos antes de crear elementos gráficos, si no está configurada todos los modelos semánticos en sesión se validarán con la precondición de crear el elemento, caso contrario se validará solamente elementos que cumplan con la expresión de condición.

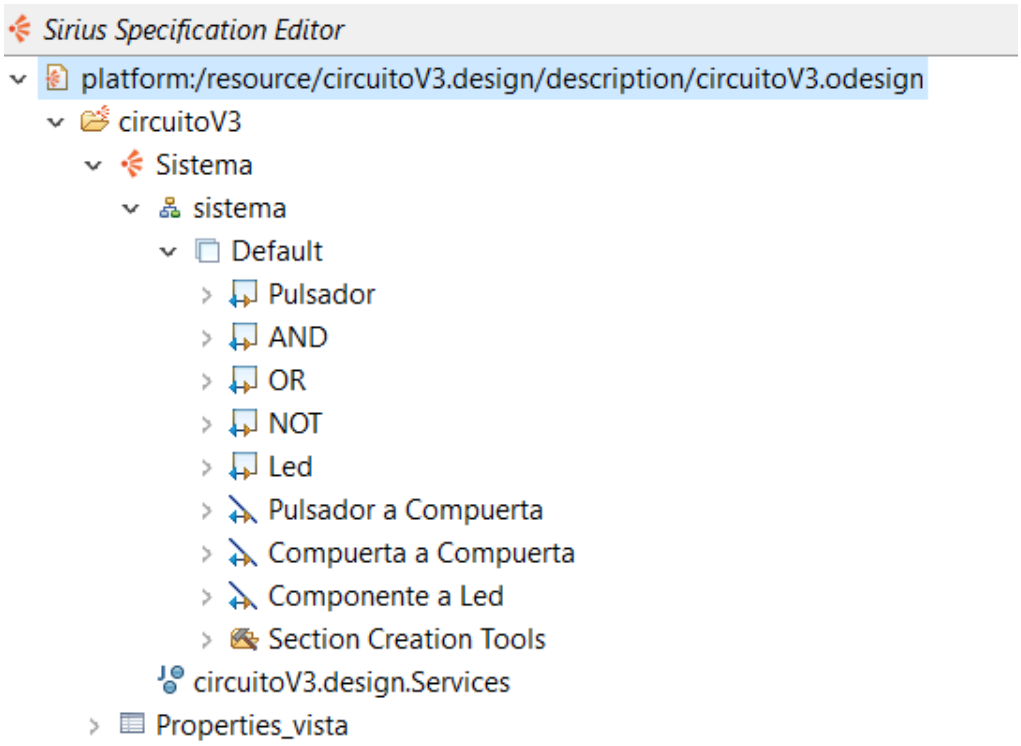
Relaciones (Defining Element-based Edges): Son responsables de la conexión de elementos relacionados por el metamodelo, representan una relación entre elementos del modelo como referencias o contención, las relaciones poseen:

- ID: Identificador único del elemento a definir.
- Domain class: Define el tipo de elementos a ser representados en la relación, es importante prevenir posibles conflictos con otros metamodelos que pudiesen ser definidos con el mismo nombre.
- Source Mapping: Elabora una ruta de elementos por donde debería empezar la relación (edge).
- Source Finder Expression and Target Finder Expression: Evalúa el contexto semántico del elemento en la relación, debería regresar los elementos actuales que conectan la relación.
- Target mapping: Elabora una ruta de elementos por donde debería terminar la relación (edge).
- Semantic candidate expression: Restringe a la lista de elementos antes de crear elementos gráficos.
- Contenedores**:** Sirven para contener subelementos que se muestra en una lista o mediante figuras, organizados según los requerimientos del usuario, puede contener sub-contenedores, sub-nodos y nodos con bordes, todos ellos definidos en las asignaciones del contenedor.

Herramientas (Defining Tools and Operations): Una vez que todos los elementos están definidos por *Nodos* y *Relaciones* las herramientas definen el comportamiento de su modelador, algunas herramientas aparecen por defecto en la paleta del diagrama, mientras que otras son llamadas automáticamente por operaciones que realiza el usuario.

Figura 5

Sirius



2.3.8. Acceleo

Es una implementación del estándar MOF de OMG, parte del proyecto Eclipse Model to Text (M2T). Ofrece varias ventajas, entre ellas: la facilidad para llevar a cabo el proceso de generación de código, alta capacidad de personalización, la interoperabilidad, la gestión de la trazabilidad. Acceleo utiliza el mecanismo de plantillas para la generación de código. La plantilla se construye para manipular modelos que se ajustan a un cierto metamodelo. La idea básica es atravesar el modelo generando un texto fijo y dejando "agujeros" a ser llenados dependiendo del modelo de entrada cuando se ejecuta la transformación.

Acceleo actualmente es un plug-in de Eclipse de código abierto, sus inicios se dieron en la empresa francesa Obeo con un crecimiento constante, este plug-in es un sistema de generación de código basado en el estándar MOF Model to Text (M2T) del Grupo de Gestión de Objetos (Object Management Group – OMG).

(Rube, 2012) “Acceleo se encuentra muy bien integrado en Eclipse e incluye las características habituales como el coloreado de sintaxis, control de trazabilidad, la cual permite encontrar los elementos del modelo, las partes utilizadas del generador y el código finalmente generado”.

Template en Acceleo

Los Template en Acceleo son un conjunto de aclaraciones que se utilizan para generar texto. Están delimitados por etiquetas [template][/template]

Figura *6*

Plantilla realizada en Acceleo para configurar las entradas del Arduino UNO

```
void setup(){
//pulsador
[for (pulsador : Pulsador | aSistema.elementos)]
                pinMode([pulsador.numero_pin/],INPUT);
[/for]
//led
[for (led : Led | aSistema.elementos)]
                pinMode([led.numero_pin/],OUTPUT);
[/for]
}
```

2.3.9. ISO/IEC 9126

Según (Jatmiko Suwawi, Darwiyanto, & Rochmani, 2015) “ISO/IEC 9126 es una evaluación de estándares reconocido internacionalmente para la evaluación de software desde la perspectiva de la Ingeniería en Software. El modelo es considero un estándar válido, confiable y eficiente para evaluar calidad de software”.

ISO/IEC 9126 es un estándar internacional realizado por ISO/IEC usado para la evaluación de software. Existen seis características de calidad en el estándar ISO/IEC 9126: funcionalidad, confiabilidad, eficiencia, usabilidad, mantenibilidad y portabilidad.

El estándar ISO/IEC 9126 para (Botella, y otros) “distingue entre calidad interna y calidad externa y los introduce al término calidad de uso”. Al unir la calidad interna y la calidad externa con la calidad en uso, se define un modelo evaluación más completo que dan como finalidad que estas características ayuden a mejorar la calidad del producto software aplicando esta serie de métricas.

Figura 7

ISO/IEC 9126-1 Modelo de calidad interno/externo

Characteristics	Subcharacteristcs
Functionality	suitability
	accuracy
	interoperability
	security
	functionality compliance
Reliability	maturity
	fault tolerance
	recoverability
	reliability compliance
Usability	understandability
	learnability
	operability
	attractiveness
	usability compliance
Efficiency	time behaviour
	resource utilisation
	efficiency compliance
Maintainability	analysability
	changeability
	stability
	testability
	maintainability compliance
Portability	adaptability
	installability
	co-existence
	replaceability
	portability compliance

Nota. Obtenido de (Botella, y otros)

2.4. Tipos de Software

2.4.1. Software propietario

El software no libre, también llamado software propietario, software privativo, o software con propietario. Según (Culebro Juárez, Gómez Herrera, & Torres Sánchez, 2006) “Se refiere a cualquier programa informático en el que los usuarios tienen limitadas las posibilidades de usarlo, modificarlo o redistribuirlo (con o sin modificaciones), o que su código fuente no está disponible o el acceso a este se encuentra restringido”.

En el software no libre una persona física o jurídica posee los derechos de autor sobre un software negando o no otorgando, al mismo tiempo, los derechos de usar el programa con cualquier propósito; de estudiar cómo funciona el programa y adaptarlo a las propias necesidades (donde el acceso al código fuente es una condición previa); de distribuir copias; o de mejorar el programa y hacer públicas las mejoras (para esto el acceso al código fuente es un requisito previo).

2.4.2. Software industrial

(Valiente) define al software industrial como “un conjunto de programas, instrucciones y reglas informáticas que se ejecutan en un Sistema digital para realizar operaciones de fabricación o gestión de la cadena de valor de la industria”.

Las actuales premisas del mercado exigen constantemente a las empresas del sector de industria una mayor productividad sin perder calidad. Hace tiempo que llegó una nueva revolución industrial: la Industria 4.0. Para mantener unos parámetros de eficacia y eficiencia a la altura de la competencia es necesario disponer de unos procesos optimizados que, hoy en día, se consiguen gracias a herramientas tecnológicas diseñadas para este ámbito empresarial, en concreto el software industrial.

Un software para industria y fabricación permite la automatización de gestiones propias de un ERP industrial; como es lo relacionado con planta de producción, stocks, presupuestos, análisis, planificación, contabilidad, etc; con otras relaciones más con sistemas de Recursos Humanos o gestión de las relaciones con clientes (CRM), como puede ser el control de horarios de los empleados, prevención y seguridad, relación e incidencias con proveedores y clientes, acciones de marketing o estrategias de ventas.

2.5. Arduino

Figura 8

Logo Arduino

Nota. Imagen obtenida de Sitio oficial de Arduino.

(Tapia Ayala & Manzano Yupa) afirman:

> "Arduino es una plataforma de hardware de código abierto, basada en una sencilla placa de circuito que contiene un microcontrolador de la marca ATMEL que cuenta con entradas y salidas, analógicas y digitales, en un entorno de desarrollo que está basado en el lenguaje de programación processing. El dispositivo conecta el mundo físico con el mundo virtual, o el analógico con el digital controlando, sensores, alarmas, sistemas de luces, sistemas de comunicación y actuadores físicos" (p. 25).

2.5.1. Arduino como PLC

Arduino también puede funcionar como un controlador lógico programable, conectándole las interfaces adecuadas para las entradas y salidas (E/S).

Técnicamente, si hablamos sólo de programar una máquina para que haga una tarea determinada, tanto un PLC como un Arduino son igualmente válidos siendo la principal ventaja de Arduino respecto al PLC es que éste es abierto, por lo que es teóricamente más fácil de aprender a programar.

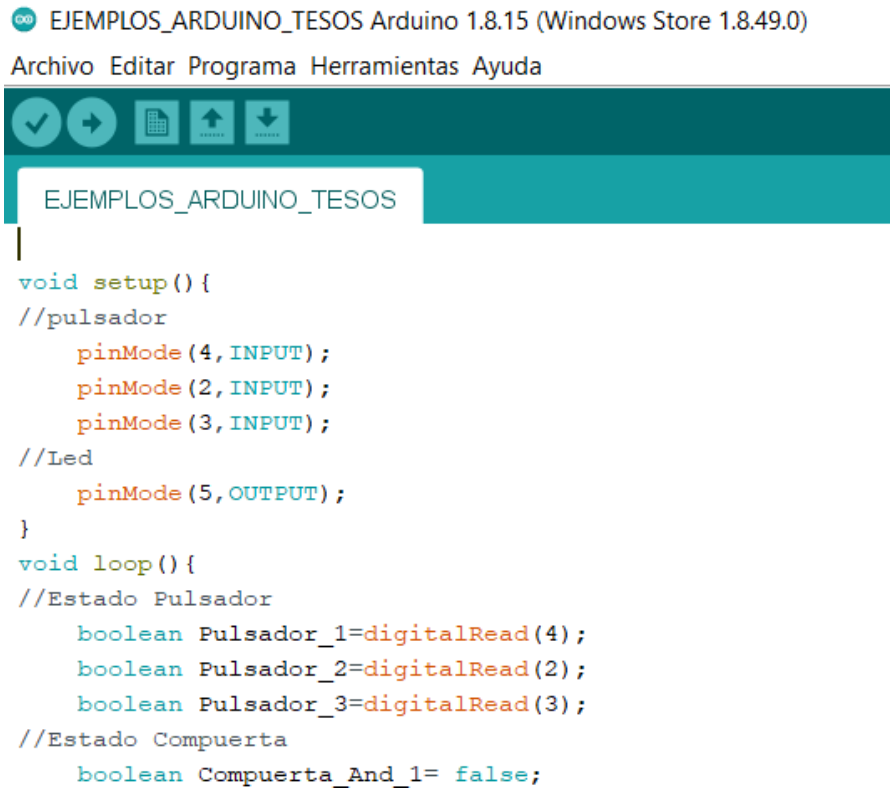
2.5.2. IDE Arduino

Es el entorno de desarrollo integrado para escribir código de manera fácil para cualquier tipo de placa Arduino, el IDE este compuesto por con un conjunto de herramientas desde

el editor de código, compilador hasta depurador. La interfaz del entorno de desarrollo integrado se puede visualizar en la figura 9.

Figura 9

Interfaz gráfica de IDE Arduino



La estructura básica de un programa Arduino está conformada por dos partes necesarias setup y loop (Ruiz Gutierrez) “La estructura básica del lenguaje de programación de Arduino es bastante simple y se compone de al menos dos partes. Estas dos partes necesarias, o funciones, encierran bloques que contienen declaraciones, estamentos o instrucciones”. La configuración mínima de estas dos funciones constituye la preparación del programa y la ejecución. (Ruiz Gutierrez) “En donde setup() es la parte encargada de recoger la configuración y loop() es la que contienen el programa que se ejecutará cíclicamente (de ahí el termino loop –bucle-). Ambas funciones son necesarias para que el programa trabaje”.

2.6. Metodología de desarrollo

2.6.1. Metodologías Ágiles

Se le denomina metodología al conjunto de herramientas, procedimientos, técnicas y documentos que le permiten a los desarrolladores de Software implementar nuevos sistemas de información. Dicha metodología se compone de distintas fases, las cuales a su vez se dividen en sub-fases, guiando a los desarrolladores del sistema para seguir las técnicas más apropiadas a lo largo del proyecto, en la planificación, gestión, control y evaluación de este.

Una metodología ágil es un procedimiento o conjunto de pasos que le permiten a los desarrolladores de Software (de un proyecto pequeño) simplificar el proceso de la programación, integrando al cliente en el equipo de trabajo, realizando entregas continuas del proyecto y permitiendo realizar los cambios que sean requeridos a lo largo del desarrollo del software para irlo adecuando a las necesidades actuales del cliente. (López Menéndez de Jiménez, 2015) “El término ágil surge como iniciativa de un conjunto de expertos en el área de desarrollo de software con el fin de optimizar el proceso de creación de este, el cual era caracterizado por ser rígido y con mucha documentación”.

2.6.2. Scrum

Scrum es un marco de trabajo que permite abordar problemas complejos para poder darles solución entregando el mayor valor posible, por eso (Schwaber & Sutherland, 2017) definen Scrum como:

> “Scrum es un marco de trabajo de procesos que ha sido usado para gestionar el trabajo en productos complejos desde principios de los años 90. Scrum no es un proceso, una técnica o método definitivo. En lugar de eso, es un marco de trabajo dentro del cual se pueden emplear varios procesos y técnicas. Scrum muestra la eficacia relativa de las técnicas de gestión de producto y las técnicas de trabajo de modo que podamos mejorar continuamente el producto, el equipo y el entorno de trabajo” (p.3).

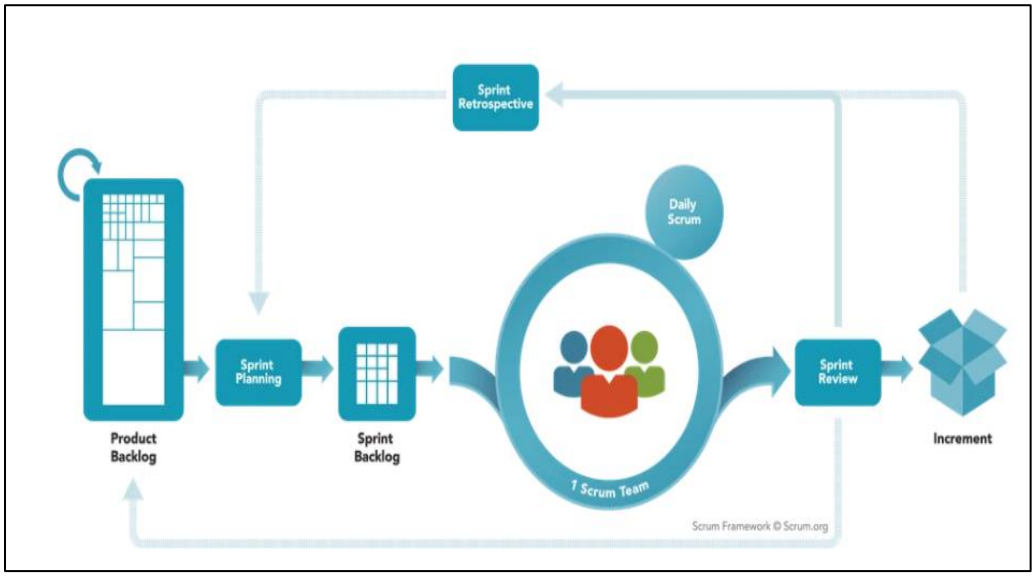
En el marco de trabajo SCRUM aplicando diferentes procesos y técnicas se muestra la eficacia de tal modo que continuamente se puede mejorar el producto.

Sprint

Los Sprints en Scrum es la parte fundamental de lo metodología, porque durante cada Sprint se entrega algo que da valor al proyecto o al cliente, pero que es Sprint, (Schwaber & Sutherland, 2017) “El corazón de Scrum es el Sprint, es un bloque de tiempo (time-box) de un mes o menos durante el cual se crea un incremento de producto “Terminado” utilizable y potencialmente desplegable. Es más conveniente si la duración de los Sprints es consistente a lo largo del esfuerzo de desarrollo. Cada nuevo Sprint comienza inmediatamente después de la finalización del Sprint anterior”.

Figura 10

Ciclo de vida de un Sprint



Nota. El gráfico representa el ciclo de vida de un Sprint aplicando la metodología Scrum desde el Sprint Planning hasta el Sprint Retrospective. Tomado de (Acosta, 2018).

Épicas

Las Épicas en Scrum son historias de usuario generales que definen un conjunto de tareas, se utiliza épicas cuando no se tienen bien definidas las tareas a realizar y mediante esta técnica se puede desglosar en historias de tamaño más pequeñas para ser gestionadas con los principios de Scrum.

2.6.3. Modelo C4

La arquitectura de software describe diversos aspectos del software donde cada uno de estos expresa el funcionamiento y la estructura. El crecimiento exponencial de la tecnología conlleva a tener sistemas complejos dificultando su mantenimiento y crecimiento a futuro. Por este motivo es importante, definir la estructura que debe tener el software a construir, los componentes que constituyen el software y como deben comunicarse e interactuar entre sí.

Una buena documentación de la arquitectura del software conlleva a generar equipos altamente efectivos y con buen desempeño. El modelo C4 significa Contexto, Contenedor, Componente y Código que sirven para describir la estructura del software.

(Vázquez Ingelmo, García Holgado, & García Peñalvo, 2020), “El modelo C4 propone cuatro niveles de abstracción, contexto, contenedores, componentes y código. Con este modelo, el sistema está dividido en partes manejables que se pueden analizar de mejor manera”. Esto ayuda a la abstracción para entender como los sistemas están diseñados y construidos. Cado uno de estos cuatro niveles del modelo C4 están orientados a cierta perspectiva del sistema.

Diagrama de Contexto

Un diagrama de contexto del sistema muestra el sistema de software que se está construyendo y cómo encaja en el mundo en términos de las personas que lo utilizan y los otros sistemas de software con los que interactúa.

(Vázquez Ingelmo, García Holgado, & García Peñalvo, 2020) afirman: “El primer nivel (contexto) se enfoca en enmarcar el software sistema a modelar representando a las diferentes personas o actores involucrados y sistemas externos que brinden cualquier tipo de servicio al sistema. Las interacciones entre componentes están representadas por líneas de flecha discontinuas anotadas por una descripción con el papel de la relación. Externo o los servicios ya implementados están coloreados con diferentes el sistema de software de destino. En este nivel, más detalles con respecto a los sistemas de software involucrados se omiten, proporcionando una vista de alto nivel del contexto en el que el objetivo el sistema se enmarcará”.

(Vázquez Ingelmo, García Holgado, & García Peñalvo, 2020) afirman: “El primer nivel (contexto) se enfoca en enmarcar el software sistema a modelar representando a

las diferentes personas o actores involucrados y sistemas externos que brinden cualquier tipo de servicio al sistema. Las interacciones entre componentes están representadas por líneas de flecha discontinuas anotadas por una descripción con el papel de la relación. Externo o los servicios ya implementados están coloreados con diferentes el sistema de software de destino. En este nivel, más detalles con respecto a los sistemas de software involucrados se omiten, proporcionando una vista de alto nivel del contexto en el que el objetivo el sistema se enmarcará".

Diagrama de Contenedores

Un diagrama de contenedor amplía el sistema de software y muestra los contenedores (aplicaciones, almacenamiento de datos, microservicios, etc.) que componen este sistema de software.

(Vázquez Ingelmo, García Holgado, & García Peñalvo, 2020) "El segundo nivel descompone el sistema de software en contenedores. Esta vista describe los componentes necesarios para proporcionar los servicios que ofrecería el sistema. De nuevo, más detalles como la estructura interna de Los componentes se omiten, proporcionando solo una función descripción de cada contenedor y las relaciones entre ellos".

Diagrama de Componentes

Un diagrama de componentes expande un contenedor individual para mostrar los componentes que contiene. Estos componentes deben asignarse a abstracciones reales en función de su código.

(Vázquez Ingelmo, García Holgado, & García Peñalvo, 2020) "Los contenedores a su vez están formados por componentes, que es la perspectiva en la que el modelo C4 se centra en el tercer nivel. La colaboración entre los diferentes componentes debe proporcionar el servicio o la funcionalidad del contenedor descrito en el segundo nivel del modelo C4".

Código

El nivel de código es el último de los diagramas que propone el modelo C4. Este nivel se centra en tomar cada uno de los componentes y hacer zoom hasta llegar a nivel de clases, objetos y sus interacciones, este modelo incluye diagramas de clases. Es un

modelo que se emplea poco, porque normalmente con los niveles anteriores es suficiente para transmitir el sistema que se desea construir.

(Vázquez Ingelmo, García Holgado, & García Peñalvo, 2020) “se puede especificar el cuarto nivel (nivel de código) a través de diagramas UML, como diagramas de clases, diagramas de secuencia, diagramas de componentes”

2.7. Implementación de un DSL

La Universidad de las Fuerzas Armadas ESPE es un centro de educación superior considerada una de las más emblemáticas del país por su constante innovación y aporte al desarrollo productivo del Ecuador. Actualmente cuenta con cuatro sedes: ESPE Sangolquí (ESPE matriz), Héroes del Cenepa en Quito, ESPE Latacunga y Hacienda Zoila Luz en Santo Domingo.

ESPE Sede Latacunga, cuenta con diferentes laboratorios adecuados según sea el fin pertinente, por esto existe el Laboratorio de Comunicaciones, el cual es apto y cuenta con los insumos necesarios para aplicar el proyecto.

El laboratorio de comunicaciones se encuentra orientado a la docencia, brinda él apoyó a las asignaturas de antenas y redes de comunicaciones de la Carrera de Ingeniería Electrónica e Instrumentación. Cuenta con equipos de medición como analizadores de espectro, generadores de señales, multímetros, osciloscopios. También cuenta con plataformas de simulación de modulaciones análogas y digitales, y equipos para prácticas de implementación de redes de datos tanto alámbricas como inalámbricas. Este laboratorio también brinda soporte para proyectos de grado y de investigación donde se requiera evaluar conectividad entre equipos o análisis de espectro radioeléctrico.

Capítulo III

2.8. Diseño e Implementación

En este capítulo se realiza el desarrollo del DSL propuesto aplicando Scrum a través de cinco Sprint divididos en la elaboración de sintaxis abstracta, sintaxis concreta, generación de código, validación e implementación.

2.9. Planificación de Backlog

Scrum permite mucha flexibilidad para entregar incrementos durante cada sprint, el presente proyecto se dividió por cinco Sprint que nos permiten tener mayor control sobre los entregables obtenidos a lo largo de cada sprint. Los actores que forman parte del proyecto desarrollado son:

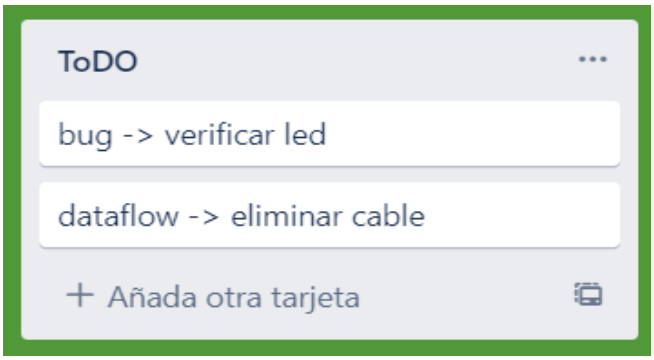
- **Dueño de Producto:** Ph.D. Rivas Lalaleo David Raimundo
- **Experto Scrum:** Ph.D. Jácome Guerrero Patricio Santiago.
- **Equipo Scrum:** Franco Jonathan, Sánchez Benjamín.
- **Usuarios:** Personas con acceso al repositorio.

Para el manejo y control de procesos iterativos se ha creado un tablero Kanban, es una herramienta ágil de gestión de proyectos diseñada para ayudar a visualizar el trabajo, limitar el trabajo en curso y maximizar la eficiencia implementado a través de la herramienta Trello, que nos sirve para la organización de tareas. Las listas de actividades implementadas para cada sprint están divididas en las siguientes secciones:

- **ToDo:** Corresponde al Product Backlog con las actividades pendientes que son creadas en cada interacción.
- **InProgress:** Se resaltan las actividades en progreso.
- **Review:** Comprueba actividades realizadas.
- **Terminadas:** Señala las actividades que han sido revisadas y aprobadas.

Figura 11

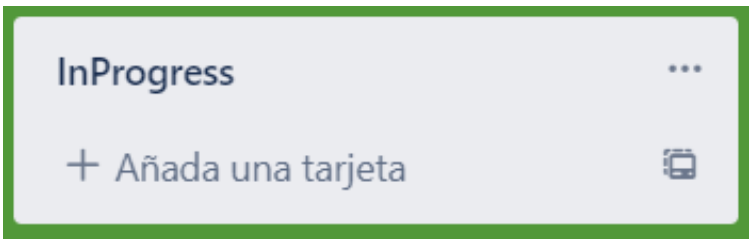
Actividades ToDo a realizar



Como se puede observar en la figura 11 las actividades que se tienen como pendiente, son actividades que describen procesos que se deben mejorar en la implementación para tener un mejor comportamiento esperado en la aplicación desarrollada.

Figura 12

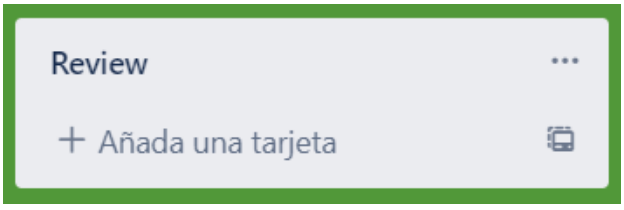
Actividades InProgress



Nota. No se encuentran actividades en la figura 12 por lo que las actividades se encuentran en la sección de actividades terminadas.

Figura 13

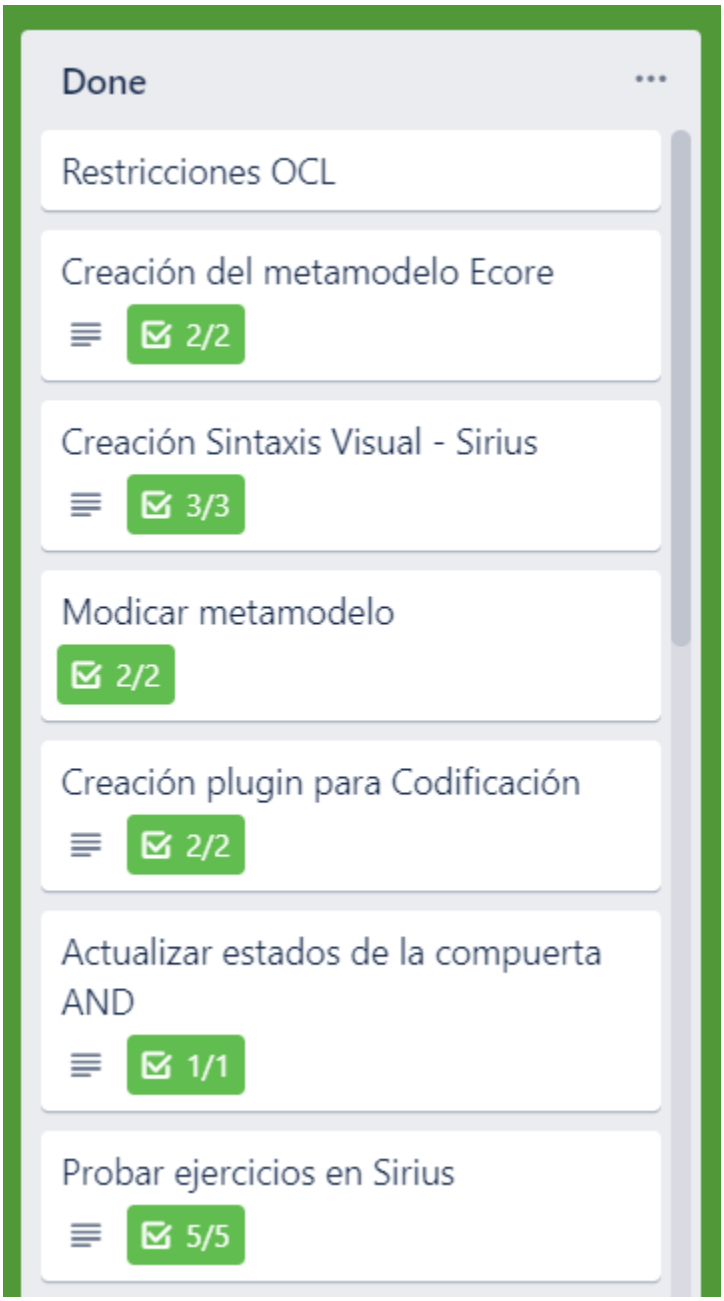
Actividades en Review



Nota. No se encuentran actividades en la figura 13, por lo que las actividades se encuentran en la sección de actividades terminadas.

Figura 14

Actividades Done



Nota. Las actividades descritas en la figura 14 representan todas las actividades realizadas y terminadas.

El backlog este compuesto por las actividades generales a implementar durante el desarrollo del DSL descritas en la figura 15.

Figura 15

Backlog para el desarrollo del DSL

2.10. Planificación Sprint No. 1

El objetivo del sprint es la definición de la sintaxis abstracta a través de Eclipse Modelling Framework – Ecore con una estimación de 4 semanas.

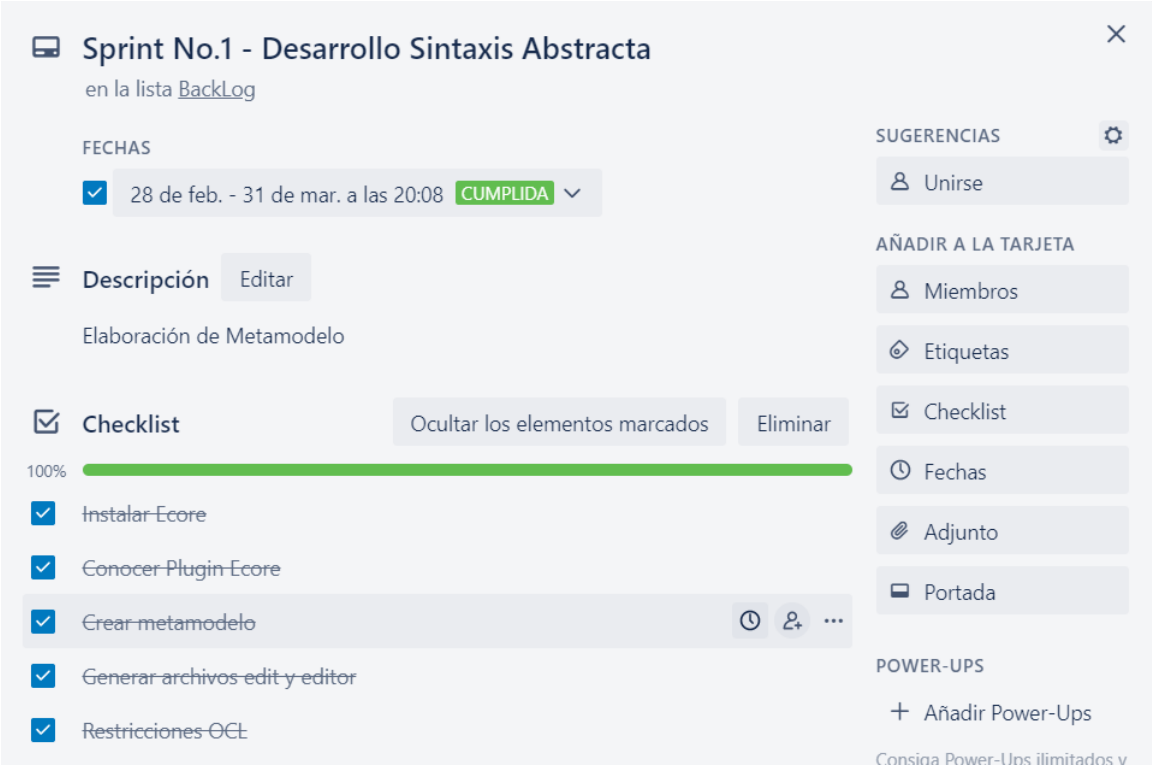
Tabla 1

Épica No. 1

Título	Descripción
Desarrollo Sintaxis Abstracta	Definición de la sintaxis abstracta (Metamodelo) del DSL a través de Ecore.

Figura 16

Planificación Sprint No.1



2.11. Planificación Sprint No. 2

El objetivo del sprint es la definición de la sintaxis concreta a través de Sirius que permite crear una interfaz gráfica, que va a hacer utilizada por el usuario para crear circuitos lógicos combinacionales con una estimación de 4 semanas.

Tabla 2

Épica No. 2

Título	Descripción
Desarrollo Sintaxis Concreta	Definición de la sintaxis concreta del DSL utilizando Sirius

Figura 17

Planificación Sprint No. 2

2.12. Planificación Sprint No. 3

El objetivo del sprint es la generación de código para la placa Arduino a través de la generación de un template en Acceleo con una estimación de 4 semanas.

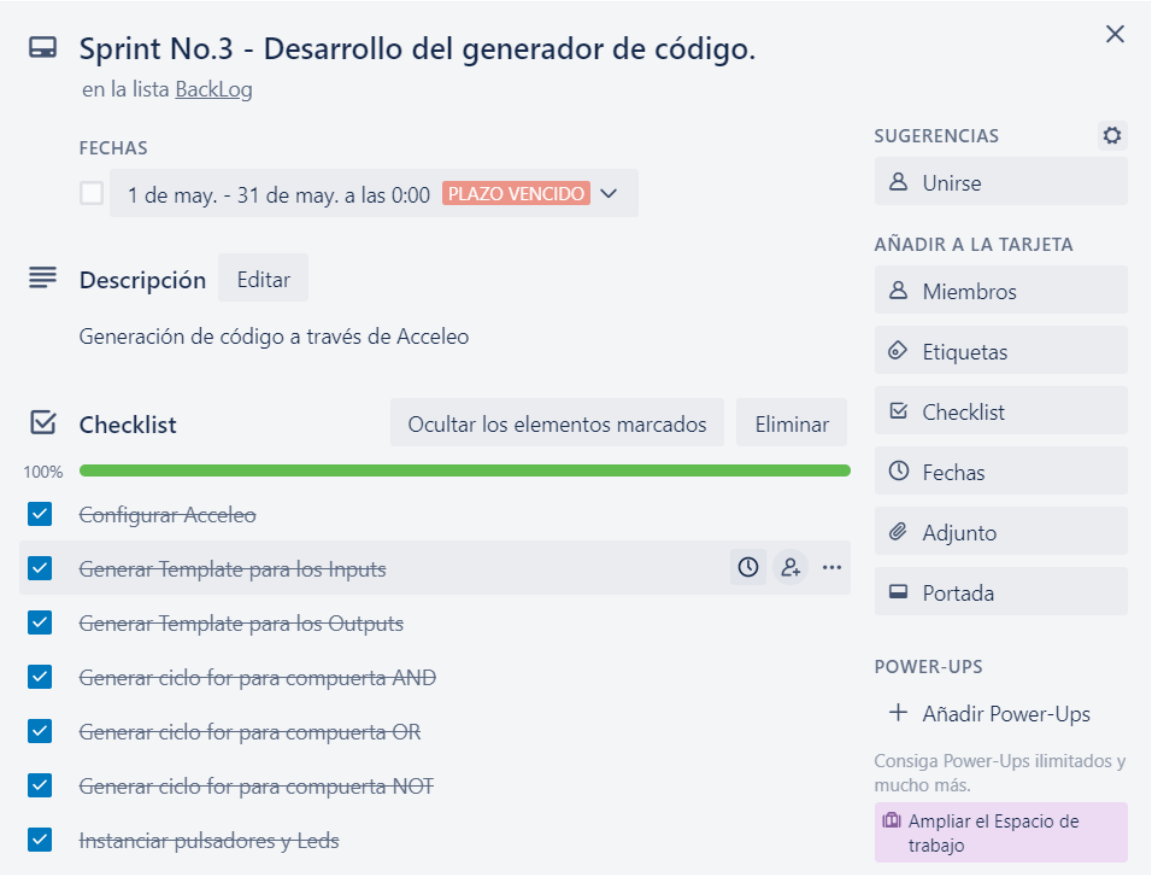
Tabla 3

Épica No. 3

Título	Descripción
Generador de código	Desarrollo del generador de código a partir del modelo generado por el usuario al código de la placa Arduino utilizando Acceleo.

Figura 18

Planificación Sprint No.3



2.13. Planificación Sprint No. 4

El objetivo del Sprint es lograr validar los resultados obtenidos a través de una evaluación empírica, donde se contrasta con otras herramientas similares y una evaluación de usabilidad, donde se determina si la herramienta cumple con las métricas de usabilidad según la norma ISO/IEC 9126 con una estimación de 4 semanas.

Tabla 4

Épica No. 4

Título	Descripción
Validar los resultados	Validación de los resultados obtenidos del DSL teniendo un enfoque hacia el cumplimiento de los indicadores señalados.

Figura 19

Planificación Sprint No.4

2.14. Planificación Sprint No. 5

Implementar el DSL desarrollado en la plataforma GitHub de manera pública para permitir la colaboración entre usuarios. El repositorio se lo puede encontrar en el siguiente enlace: https://github.com/Jrfranco2/circuitoV3 con una estimación de 1 semana.

Tabla 5

Épica No. 5

Título	Descripción
Implementación del aplicativo	Implementar el aplicativo en un repositorio público para compartir con la comunidad promoviendo la colaboración entre usuarios.

Figura 20

Planificación Sprint No. 5

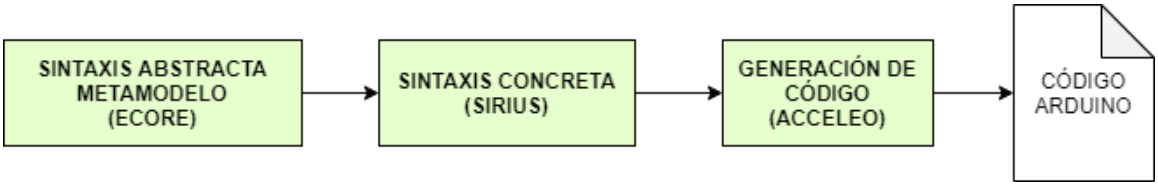
2.15. Diseño arquitectónico

El proceso de MDE empieza a partir de la conceptualización de un modelo, donde pasa por un proceso de evoluciones hasta llegar a una transformación final modelo a texto. Un problema representado en el mundo real simboliza un modelo descrito por un DSL sujeto a una sintaxis concreta y abstracta, que tiene por finalidad una transformación para el dominio semántico en un lenguaje de programación.

En la figura 21 se aborda de manera general el proceso llevado a cabo para el desarrollo del DSL.

Figura 21

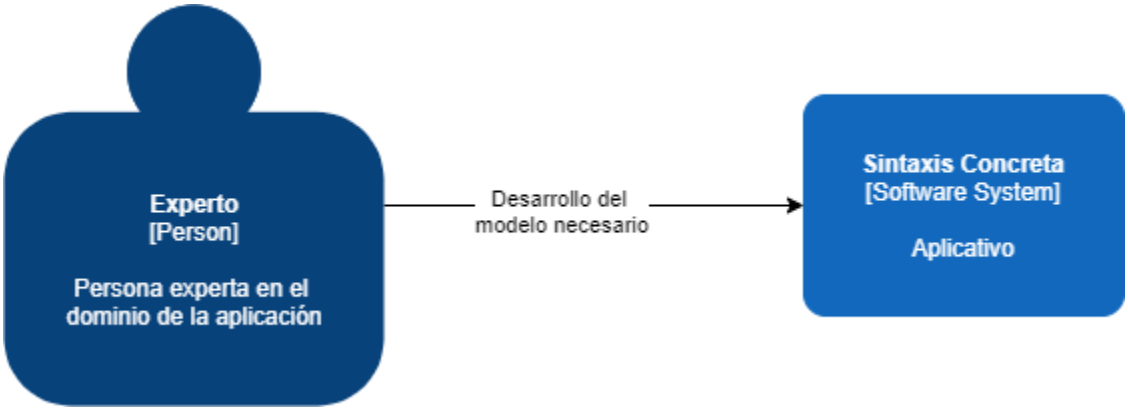
Arquitectura General para el desarrollo del DSL



La aplicación facilita el trabajo para el experto con conocimiento en el dominio específico, por esto en la figura 22 se representa como interactúa el DSL con el usuario final.

Figura 22

Diagrama de Contexto



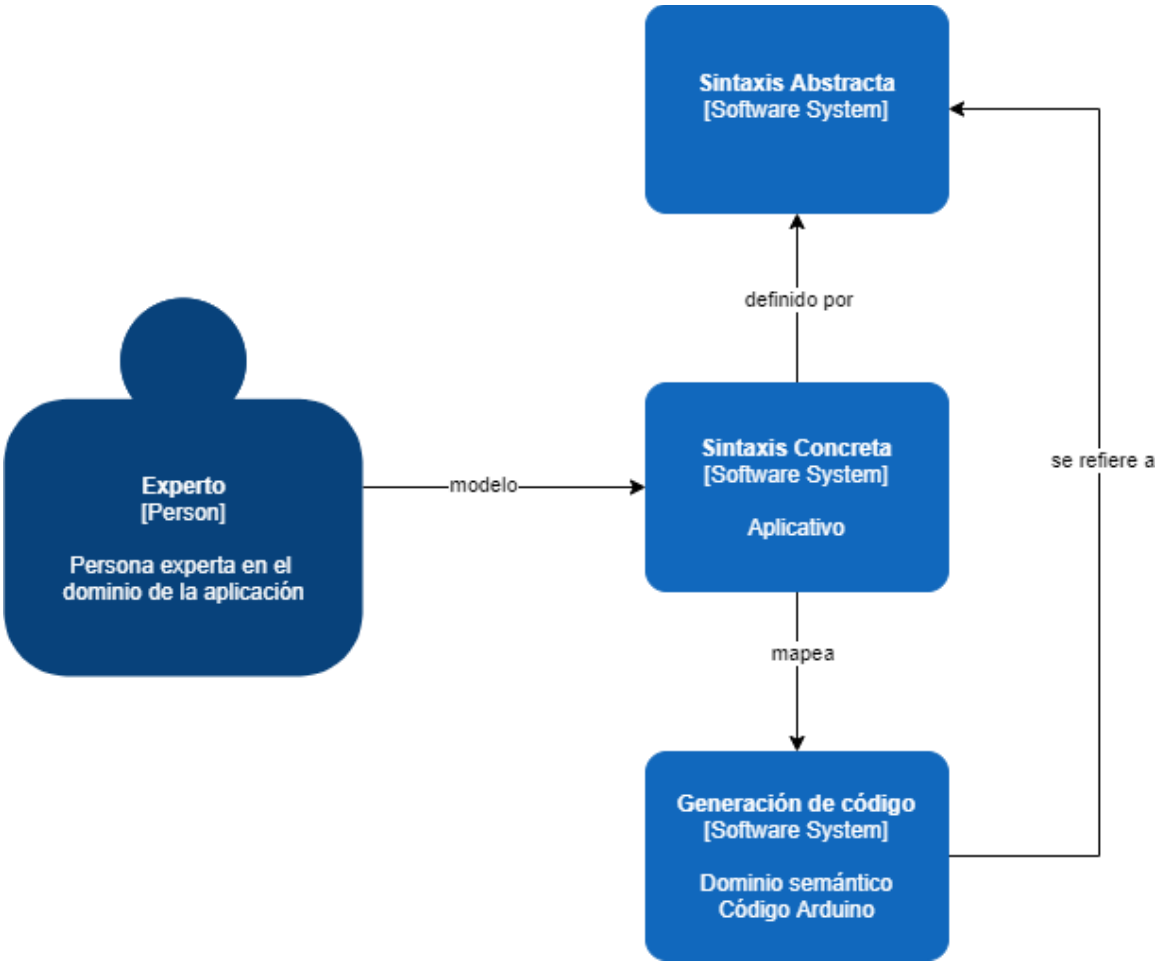
Nota. El diagrama de contexto nos permite identificar la interacción con personas y con otros sistemas que se relaciona el aplicativo desarrollado.

A través de la abstracción se puede desarrollar aplicaciones, donde se describe el problema sin detallar la solución, mediante un proceso automatizado de transformación de modelos. Este proceso de transformación se presenta en la figura 23, que permite obtener un archivo en el lenguaje de programación para la placa Arduino y simplemente compilar este código para montarlo sobre la placa.

Para lograr este proceso de trasformación existen herramientas que permiten agilizar el proceso, mediante la transformación, simulación, verificación y personalización de modelos a través de una notación gráfica realizada en Sirius hasta la generación de código realizada en Acceleo.

Figura 23

Diagrama de Componentes



Nota. La figura representa la arquitectura del DSL desarrollado

2.16. Desarrollo del DSL

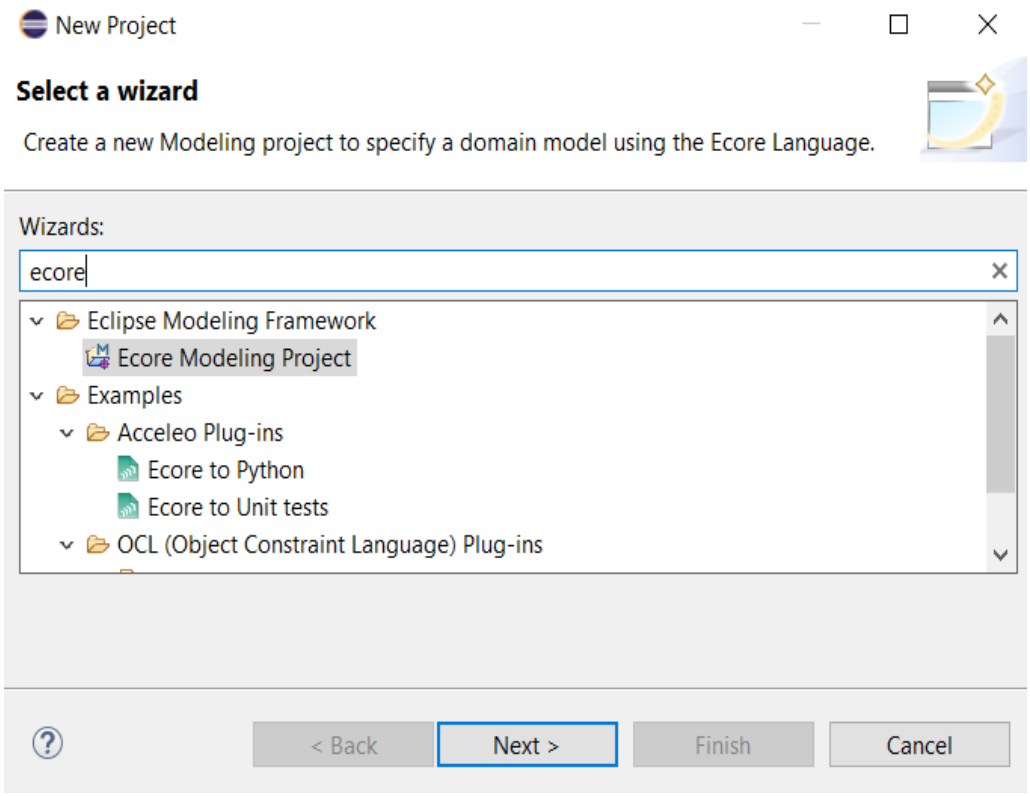
Para modelar el tema propuesto DSL se ha utilizado la herramienta Ecore para el diseño del metamodelo, OCL como lenguaje de restricciones en el diseño del metamodelo, Sirius el editor gráfico que permite trabajar con el metamodelo creado instanciando sus elementos y mediante Acceleo transformar todos los elementos creados en Sirius a texto que pueda ser interpretado por la placa Arduino, utilizando en todos los procesos el IDE Eclipse Modeling Tool que maneja el lenguaje de programación JAVA.

Como características generales que posee el trabajar en un programa con la placa Arduino UNO, se obtiene el manejo de periféricos tanto de entrada como de salida y el flujo de estados encendido/apagado de los dispositivos.

Una vez definidas las características generales que va a poseer el DSL se continua con el diseño del metamodelo, para lo cual se utiliza el IDE Eclipse Modeling 2020-12, se instancia un nuevo proyecto Ecore como muestra la figura 24, donde muestra el wizard para la creación de un proyecto con Ecore Modeling Project.

Figura 24

Eclipse Modeling Framework



2.16.1. Sintaxis Abstracta (Metamodelo)

Para empezar, se necesita de un metamodelo que haga uso de:

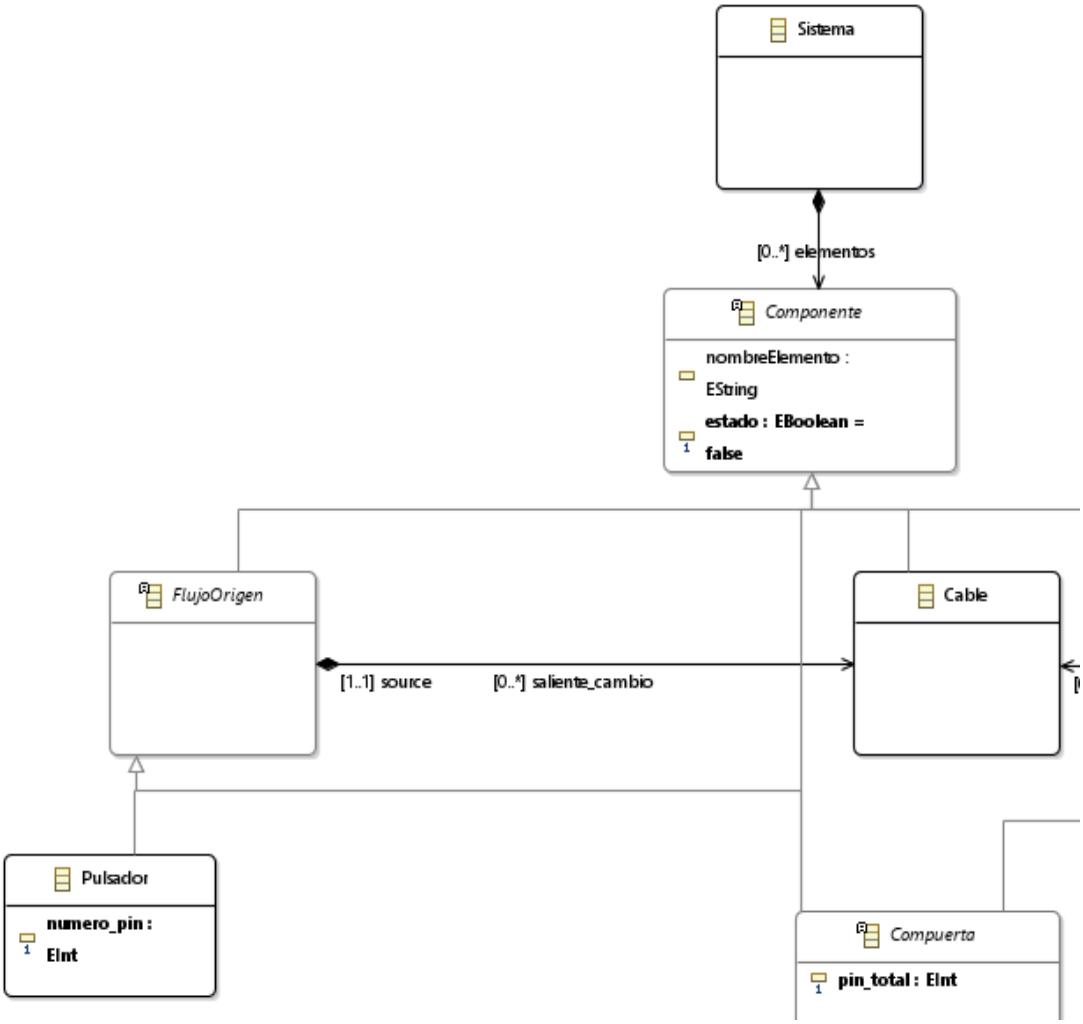
- **Clasificadores:** clase y clase abstracta.
- **Características:** atributos.
- **Relaciones:** herencia, composición y referencia de dos direcciones.

El metamodelo empleado se compone de una clase contenedora raíz nombrada *Sistema* responsable de instanciar elementos mediante la clase abstracta Componente, como parte del sistema físico a Led, Pulsador, con atributos comunes como nombreElemento, estado, pin_util, para el apartado lógico se crean clases con nombre AND, OR y NOT, que heredan de la clase abstracta Compuerta y como mediador entre

esos apartados Cable, al ser esta una clase intermediaria, sirve de ayuda para determinar el origen, destino y estado actual de los elementos. Se buscó que el metamodelo pueda ser escalable, es decir aumentar más elementos evitando que varié la base planteada, para lo cual las clases abstractas FlujoDestino y FlujoOrigen separan muy bien los flujos de entrada o salida.

Figura 25

Metamodelo – Sección I

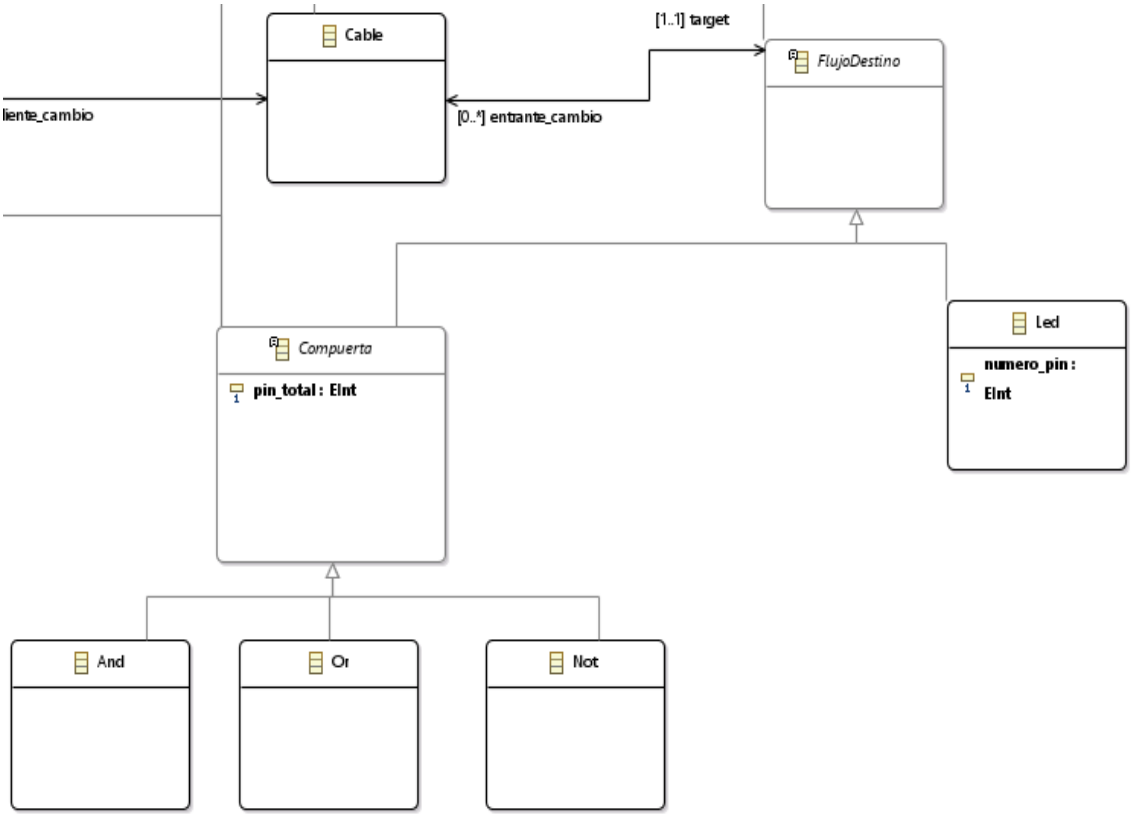


Como muestra la figura 25, el metamodelo nace a partir de una clase contenedora llamada Sistema, que instancia los diferentes elementos a través de la clase abstracta Componente que tiene como atributos nombreElemento de tipo EString y estado de tipo EBoolean, a su vez la Clase Componente está compuesta de la Clase Flujo Origen, la Clase Cable, la Clase Pulsador y la Clase Compuerta, que sirven para mantener la

dirección del flujo para tener conocimiento de donde nace el cambio y que clase está efectuando ese origen. Pulsador tiene como atributo el numero_pin de tipo EInt que nos sirve para determinar el número de pin a utilizar dentro de la placa Arduino UNO.

Figura 26

Metamodelo – Sección II



Como se puede observar en la figura 26, la Clase componente está compuesta de las Clases Cable Compuerta y Flujo Destino, la Clase FlujoDestino como la clase FlujoEntrada a través de la clase Cable sirven para saber el flujo de los datos, con esto se aborda hacia dónde y de dónde vienen las compuertas, por ejemplo, un pulsador va hacia una compuerta AND, y la compuerta AND viene de un pulsador, de esta manera sabemos el flujo bidireccional de los datos, la Clase FlujoDestino está compuesta por la Clase Compuerta Y Led que son los entes principales hacia donde se dirige un circuito Lógico. La clase Compuerta tiene como atributo pin_total y se encuentra compuesta por las Compuertas de tipo AND OR y NOT.

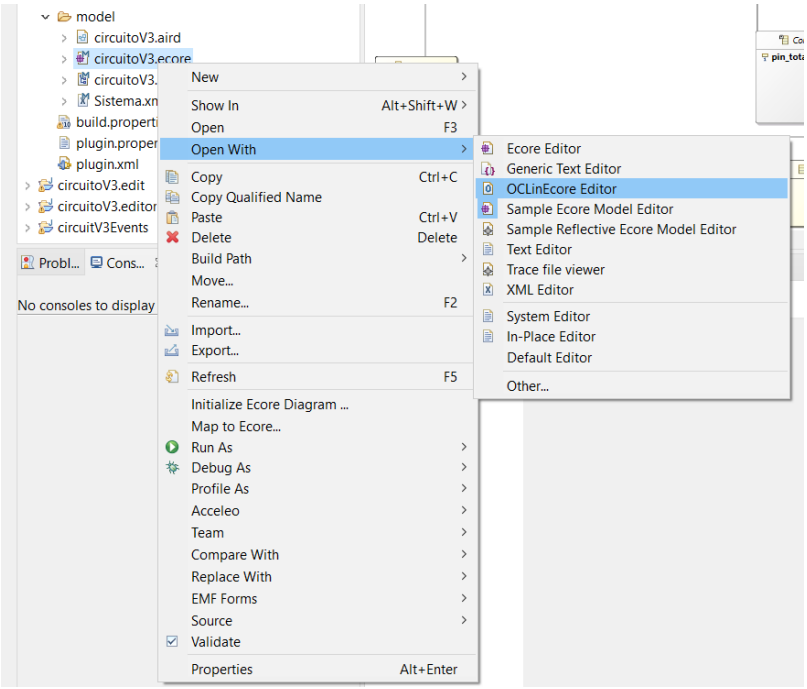
2.16.2. Restricciones del metamodelo

Análogamente es necesario crear objetos que respeten las condiciones propias de cada clase para el correcto funcionamiento del DSL, prerrequisitos como el uso correcto de un nombre, a que dispositivos puede conectarse o si el pin configurado es válido o no, por lo mismo OCL maneja restricciones a nivel del metamodelo, mediante una descripción formal de expresiones que pueden representar invariantes, precondiciones, postcondiciones, inicializaciones para determinar el estado de una clase del modelo, por esta razón es de utilidad definir un conjunto de reglas a partir de este punto, puesto que se permitirá al usar el editor gráfico realizado en Sirius tener más control sobre los circuitos realizados.

La figura 27 muestra cómo utilizar OCL para esto es necesario abrir el archivo 'circuitoV3.ecore' como OCLinEcore Editor.

Figura 27

OCLinEcore Editor



Una vez abierto el archivo como OCLinEcore se procede a definir las restricciones, mediante el uso de la palabra reservada invariant, la misma que sirve para representar reglas que deben ser verdaderas para los objetos modelados, su sintaxis es simple:

invariant [<constraint name>]: <Boolean OCL expression>

Es decir, se escribe la palabra reservada, se le asigna un nombre a la restricción y se procede a realizar la condición.

Para esta ocasión se validó que los nombres de las compuertas, leds y pulsadores no se repitan, a su vez la utilización de un número de pin valido para Arduino UNO y que tanto el número como el nombre no este vacío, con el siguiente código:

Figura 28

Validación atributo nombre

```
class Sistema
{
    property elementos : Componente[*|1] { ordered composes };
    invariant
    vacioNombre: self.elementos->forAll(c1 | c1.nombreElemento<>'' and c1.nombreElemento<>null and c1->excludes(Cable));
    invariant nombre_repetido: self.elementos->forAll(c1,c2 | c1.nombreElemento=c2.nombreElemento implies c1=c2);
}
```

La figura 28 muestra las condiciones para validar tanto que el nombre no sea repetido, como que no este vacío, es preciso señalar que son instanciadas en la clase Sistema, ya que al ser la clase raíz permite un mejor control en este atributo de manera general, conviene especificar que cada regla creada debe estar dentro de una clase, por ejemplo, si la clase hereda atributos de otra, también heredara sus restricciones.

La sentencia que corresponde a vacioNombre se encarga de tomar a los elementos instanciados, mediante self.elementos, e invocar a todos los componentes mediante la sentencia forAll(c1 | c1.nombreElemento<>'' and c1.nombreElemento<>null and c1->excludes(Cable));. Si existe alguno, la función devuelve el valor de verdad, validando así que existen componentes con un nombre vacío o invalido a excepción de la clase Cable, que sirve de conexión.

Para terminar con las validaciones generales, la sentencia que pertenece a nombre_repetido funciona de la misma manera, invoca a todos los elementos y hace una comparativa mediante: forAll(c1,c2 | c1.nombreElemento=c2.nombreElemento implies c1=c2);. Para así determinar si más de un elemento posee un nombre igual a otro.

Para empezar con reglas más específicas, la figura 29 describe, mediante su código, que la compuerta Not no puede tener más de una entrada.

Figura 29

Regla para varias conexiones.

```
class Not extends Compuerta
{
    invariant variasConexiones: Not.allInstances()->exists(self.entrante_cambio->size()<2);
}
```

No obstante, es necesario comentar el código para una mejor comprensión, la expresión Not.allInstances()->exists(), permite llamar a todas las instancias de la clase Not y preguntar si existe una clase o más, que satisfaga la condición donde self.entrante_cambio->size()<2, toma el número total de elementos conectados a la entrada que posee cada compuerta Not y compara su tamaño, si es menor a dos devuelve un valor boolean true, caso contrario devuelve el valor boolean false.

Continuando con el manejo de restricciones específicas la figura 30, señala el código implementado para validar el rango de pines digitales a utilizar.

Figura *30*

Regla para el número de pin

```
class Pulsador extends FlujoOrigen
{
    attribute numero_pin : ecore::EInt[1];
    invariant pin_fuera_rango: FlujoOrigen.allInstances()->exists(numero_pin>1 and numero_pin<13);
}
```

En esta ocasión se hace uso de la clase FlujoOrigen para poner establecer el rango numérico de los pines, estos corresponden al apartado físico de la placa, cuyo valor va de cero a trece, por consiguiente, la sentencia mostrada en la figura 30, llama a todos los elementos y valida si existe uno que posea en su atributo numero_pin un valor mayor a uno y menor que trece.

La clase FlujoDestino comparte las reglas de validación para varias conexiones y para validar que el pin no esté fuera de rango como se ve en la figura 31.

Figura 31

Reglas de la clase FlujoDestino

```
class Led extends FlujoDestino
{
    attribute numero_pin : ecore::EInt[1];
    invariant pin_fuera_rango: FlujoOrigen.allInstances()->exists(numero_pin>1 and numero_pin<13);
    invariant varias_conexiones: Led.allInstances()->exists(self.entrante_cambio->size()<2);
}
```

En resumen, la figura 32 muestra cómo debe quedar el archivo modificado con todos los cambios agregados *en la sección de manejo de restricciones.*

Figura *32*

Restricciones OCL

*circuitoV3.ecore circuitoV3

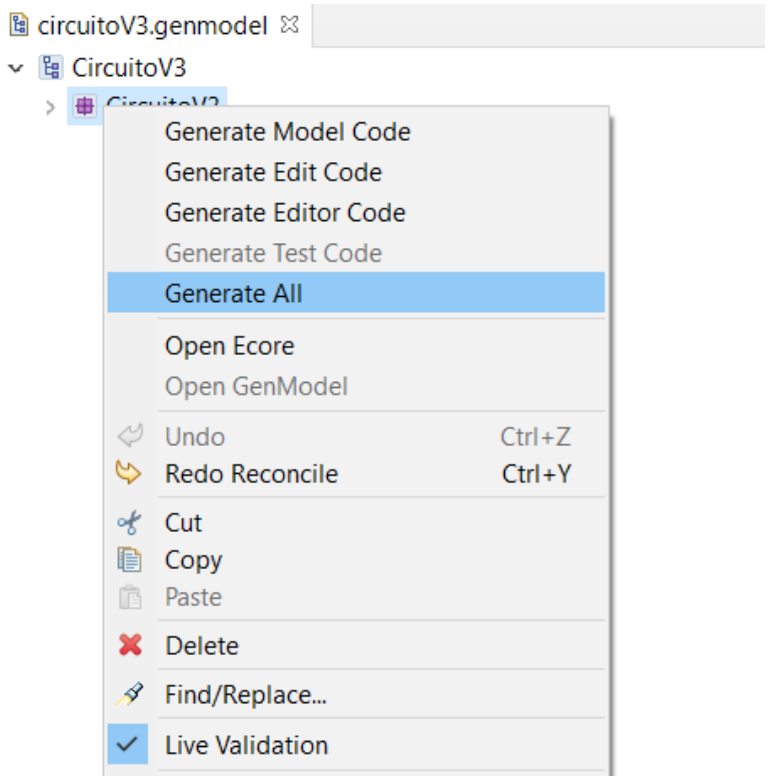
```
package circuitoV3 : circuitoV3 = 'http://www.example.org/circuitoV3'
{
    class Sistema
    {
        property elementos : Componente[*|1] { ordered composes };
        invariant
        vacioNombre: self.elementos->forAll(c1 | c1.nombreElemento<>'' and c1.nombreElemento<>null and c1->excludes(Cable));
        invariant nombre_repetido: self.elementos->forAll(c1,c2 | c1.nombreElemento=c2.nombreElemento implies c1=c2);
    }
    class Pulsador extends FlujoDestino
    {
        attribute numero_pin : ecore::EInt[1];
        invariant pin_fuera_rango: FlujoDestino.allInstances()->exists(numero_pin>1 and numero_pin<13);
    }
    abstract class Componente
    {
        attribute nombreElemento : String[?];
        attribute estado : Boolean[1];
    }
    abstract class FlujoDestino extends Componente
    {
        property saliente_cambio#source : Cable[*|1] { ordered composes };
    }
    abstract class FlujoOrigen extends Componente
    {
        property entrante_cambio#target : Cable[*|1] { ordered };
    }
    class Cable extends Componente
    {
        property target#entrante_cambio : FlujoOrigen[1];
        property source#saliente_cambio : FlujoDestino[1];
    }
    abstract class Compuerta extends FlujoDestino,FlujoOrigen,Componente
    {
        attribute pin_total : ecore::EInt[1];
    }
    class And extends Compuerta;
    class Or extends Compuerta;
    class Not extends Compuerta
    {
```

2.16.3. Sintaxis concreta (Sirius)

La elaboración de la sintaxis concreta, entendida como la sintaxis gráfica se implementó en Sirius, pero antes de ello es necesario contar con los archivos editor y edit propios del proyecto que se generan a partir del metamodelo creado con ayuda del archivo genModel como se ve en la figura 33, a partir de estos archivos generados se puede tener acceso a los iconos e imágenes para representan las diferentes clases creadas en el metamodelo, para configurar el entorno de Sirius se crea una nueva instancia de entorno de ejecución de Eclipse.

Figura 33

Generación de archivos edit y editor



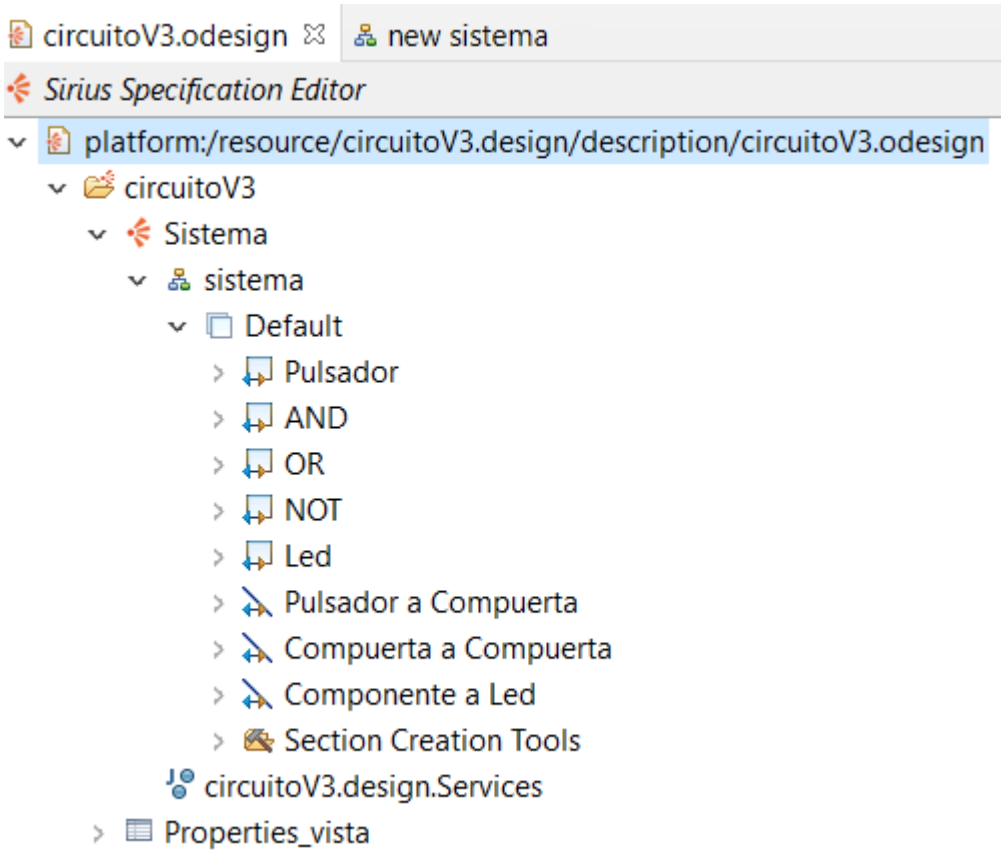
Nota. Al dar click en *Generate All* se crean los los archivos edit y editor automáticamente en el área de trabajo.

Si se necesita modificar el modelo no es necesario borrar estos archivos creados, la opción *Generate All* sobrescribe los archivos creados.

Dentro del nuevo entorno de eclipse se debe crear un nuevo proyecto de tipo *modeling Project* y se selecciona el metamodelo creado anteriormente en Ecore, una vez creado el proyecto correctamente se procede a crear tanto los nodos, relaciones, herramientas que vaya a necesitar el proyecto como se observa en la figura 34.

Figura 34

Editor Sirius – Viewpoint Specification Model



Nota. La dirección URI (Identificador de recursos uniforme) del metamodelo debe estar registrada dentro del paquete y ser seleccionada en la nueva representación.

El dominio describe a una aplicación de la industria electrónica. Se describirá los detalles de implementación relacionados con los pasos de flujo de trabajo.

El framework de trabajo open source Sirius ayuda a especificar y construir el editor del DSL. El enfoque que proporciona la herramienta permite manipular fácilmente un modelo de dominio definido por el metamodelo ya construido. Sirius facilita estilos, filtros y capas para representar las representaciones gráficas. A partir del modelo construido es

posible representar y crear los elementos definidos por el usuario. Sirius interpreta la especificación del editor DSL dinámicamente, por lo tanto, es necesario un paso de generación de código de esta representación creada en Sirius que acelera el desarrollo de herramientas de dominio específico significativamente.

Cuando se crea una nueva especificación del viewpoint con Sirius, se accede a la visualización de los elementos del modelo. La especificación del editor DSL se puede realizar de forma similar a los metamodelos basados en Ecore.

Como se observa en la figura 34, el viewpoint de la especificación del DSL creado, se crea diferentes nodos como lo son pulsados, las compuertas y los cables con lo que se une todo englobado dentro de un Sistema.

Nodos

En Sirius las notaciones gráficas se definen en el diseño de la especificación del punto de vista. Se considera a un nodo como la representación de una clase mediante un elemento gráfico, se necesita configurar a los elementos representados gráficamente, proporcionando un nombre, una figura geométrica o a su vez una imagen.

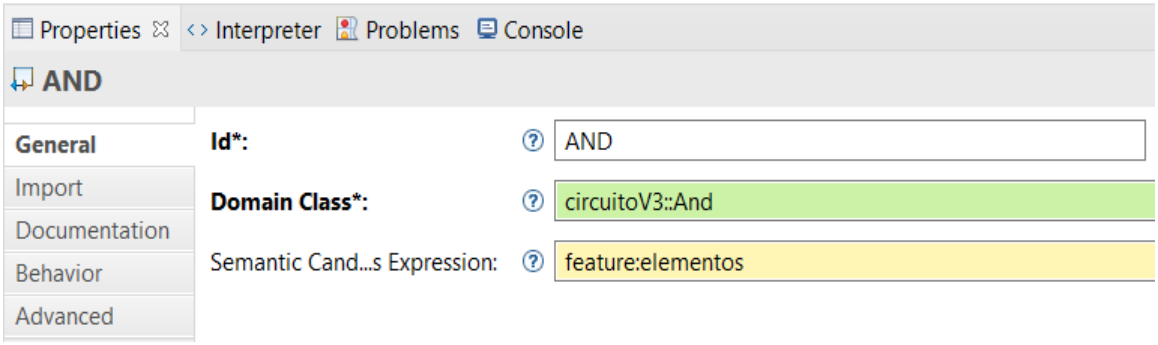
Los componentes que forman parte del DSL se definen según la especificación del metamodelo, que es:

1. Crear el elemento gráfico.
2. Determinar a qué clase de dominio y qué candidatos semánticos se representarán en ese elemento específico.
3. Personalizar el elemento a mostrar.
4. Agregar reglas de validación.

Se tomó como nodos a los componentes físicos que son el pulsador y led, y a los componentes lógicos a las compuertas AND, OR y NOT, cada una con sus atributos y relaciones ya definidas por el metamodelo.

Figura 35

Ejemplo asociación de un nodo



Nota. Nodo relacionado con compuerta lógica AND

Para la creación de un Nodo se estable:

- **ID:** Nombre único, debe tener relación con la clase que va a representar, como ejemplo AND.
- **Domain Class:** Representa la clase del metamodelo que va a ser instanciada, se necesita referenciar con el nombre del metamodelo, como muestra circuitoV3::And.
- **Semantics Candidates Expresión**: Señala a la relación existente en el metamodelo, para este caso los elementos a instanciar heredan de una clase padre Sistema su expresión semántica es feature: elementos.

Además, se agrega la imagen que representa el nodo, para ello se utiliza condiciones encargadas del estado de los componentes esta es: aql:self.estado=false, entonces, la imagen va a cambiar según se encuentre en estado encendido o apagado.

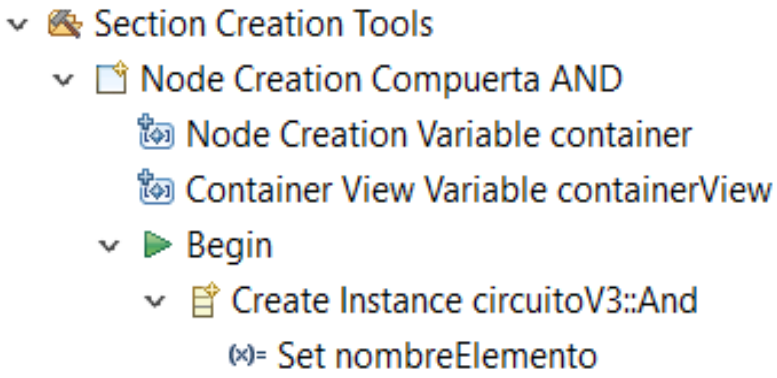
Por otra parte, para representar estos nodos de manera gráfica dentro de la paleta se utiliza la sección Section Creation Tools, cual se configura a continuación.

1. Se crea un nuevo elemento de creación de tipo nodo.
2. Se añade una ID, la misma que saldrá como nombre en el apartado de la paleta.
3. Se mapea el nodo ya creado, como ejemplo el nodo lleva la ID AND.
4. En la sección Begin se crea una nueva instancia con la referencia del metamodelo circuitoV3::And.

5. Se establece el nombre con el siguiente código: aql:'Compuerta_And_'+ self.eContainer().eContents()->filter(circuitoV3::And)->size(), el cual sirve para determinar cuántas compuertas están creadas y proporcionar su número correspondiente.

Figura 36

Creación del elemento AND en el apartado de herramientas



Por último, este proceso se repite para los componentes nombrados con anterioridad.

Relaciones

Una relación permite vincular distintos nodos entre sí, se selecciona la relación tipo *Element based Edge* dada la facilidad que ofrece hacia clases que son heredadas, como se puede ver en la figura 37 es imprescindible considerar que clases se relacionan entre si dentro del metamodelo, a fin de evitar errores, para lo cual se debe especificar:

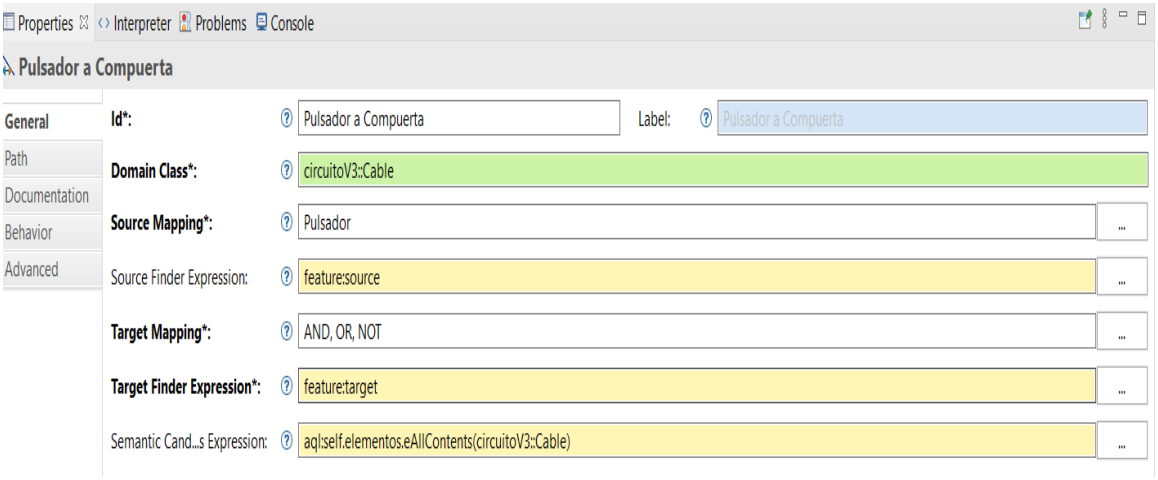
- De que clase viene la relación.
- Como se llama la relación de la clase origen.
- A que clase apunta.
- Como se llama la relación de la clase apuntada.

Por las clases presentadas en el metamodelo, se pude señalar el flujo que llevan los elementos nodos instanciados, siendo estos separados por periféricos de entrada los cuales envían la señal simbolizados por pulsadores, periféricos de salida que reciben la señal como lo son leds y periféricos mixtos representados por las compuertas, que pueden distinguir flujos de entrada y salida. De este motivo se especifica el tipo de relación será como la figura 37 representa:

- Pulsador a Compuerta: Representa la unión de pulsador a una compuerta lógica.
- Compuerta a Compuerta: Representa la unión de compuerta lógica a compuerta lógica.
- Componente a Led: Representa la unión de compuerta lógica o pulsador a un led.

Figura 37

Element based edge



Nota. se debe crear relaciones para los elementos que estén instanciados en los nodos.

Por otra parte, para representar las relaciones de manera gráfica dentro de la paleta se utiliza la sección Section Creation Tools, cual se configura a continuación.

1. Se crea un nuevo elemento de creación de tipo Edge.
2. Se añade una ID, la misma que saldrá como nombre en el apartado de la paleta.
3. Se mapea las relaciones creadas Pulsadora a Compuerta, Compuerta a Compuerta y Componente a Led.
4. En la sección Begin se crea una nuevo contexto y se le agrega el siguiente código var:source.
5. Dentro del nuevo contexto se crea una nueva instancia, de tipo circuitoV3::DataFlow, de esta manera se podrá manejar las salidas de los componentes.
6. Se establece el destino y estado, mediante la operación Set.

Figura 38

Creación del elemento Edge Cable en el apartado de herramientas

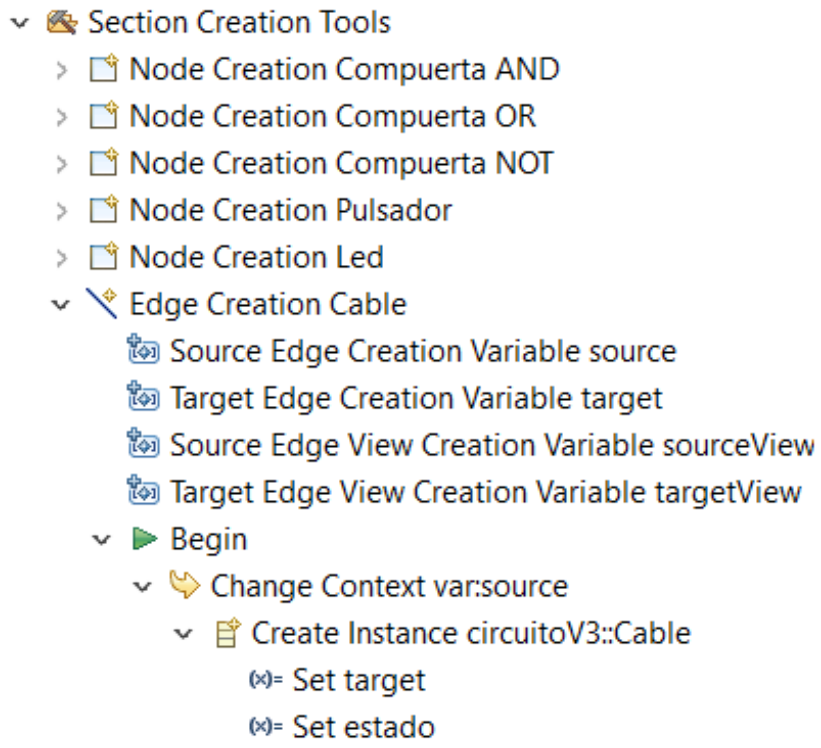
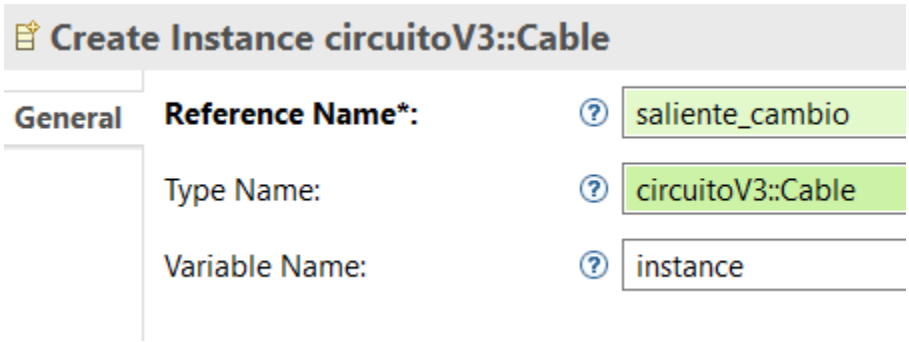


Figura *39*

Creación de la clase Cable



Por último, este proceso se repite para los componentes nombrados con anterioridad.

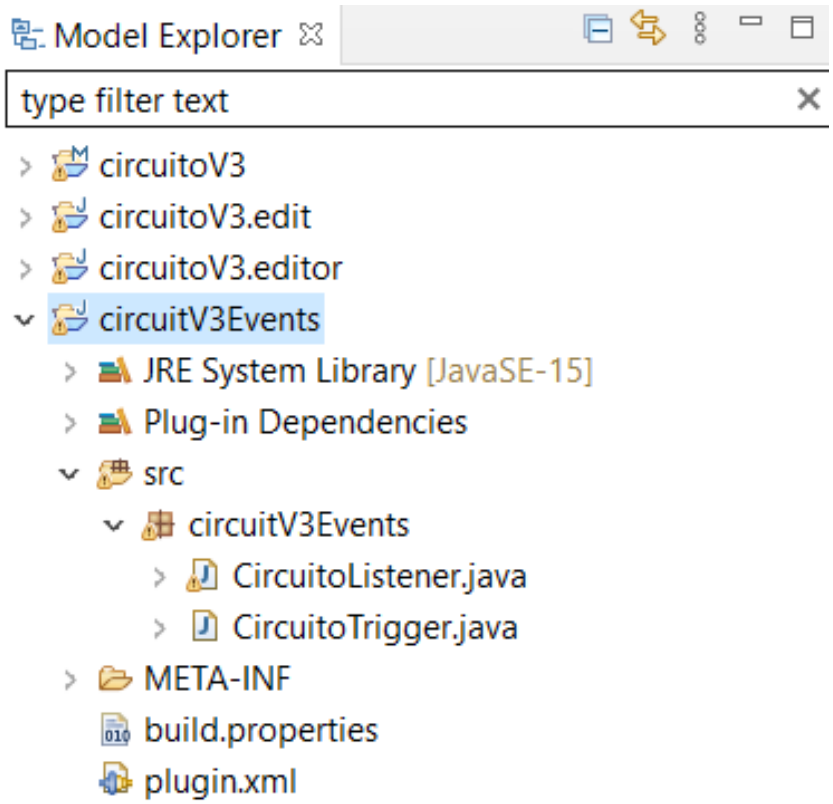
Plug-in Trigger Listener

El usuario puede iniciar acciones o tareas de manera automática, para realizar una o varias ediciones en un modelo. Estos cambios suelen tener un efecto dominó en otros componentes de una aplicación.

Por este motivo, se decide implementar un plug-in que sea capaz de escuchar los cambios de estados presentados en los nodos instanciados, a modo de si un pulsador cambia de estado este afecte directamente a los componentes conectados a él, demostrado en la figura 40. Esto permite tener los cambios reflejados en caliente, es decir, se puede observar en la interfaz gráfica como los gráficos cambian independientemente del estado de los pulsadores instanciados en el circuito que se está armando.

Figura 40

Listener y Trigger



Nota. El listener y Trigger implementado nos permite obtener los cambios reflejados en caliente al armar un circuito lógico combinacional.

2.16.4. Generador de código (Acceleo)

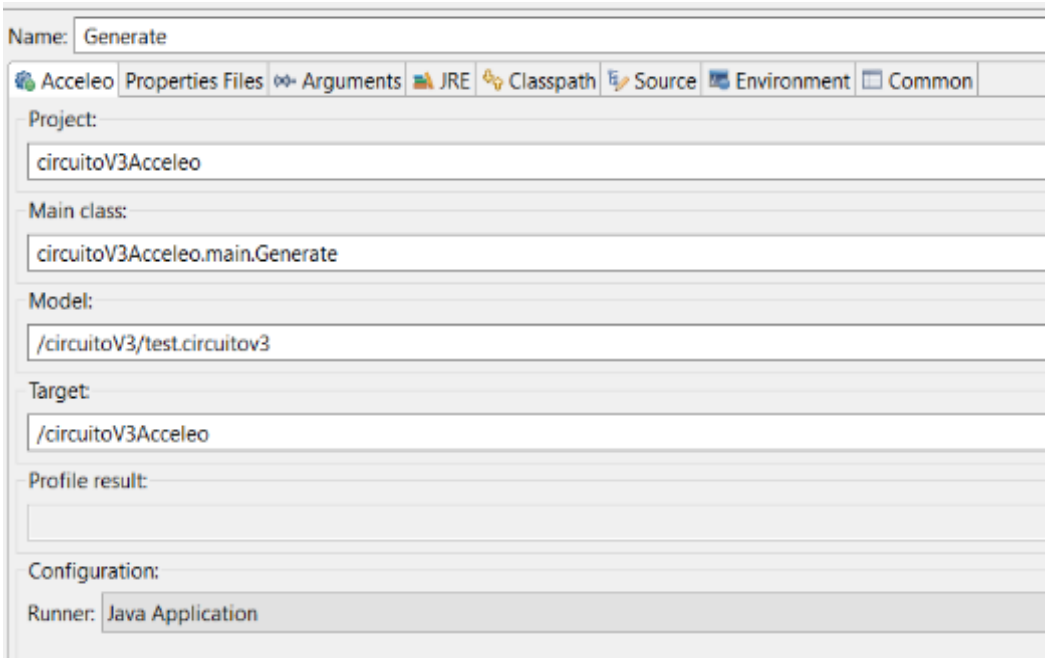
Se necesita crear código específico para cada representación creada en Sirius, por este motivo se conviene definir las reglas de transformación para la plataforma de Arduino, la misma que está dividida por tres secciones:

- Declaración de variables.
- Función setup(), donde se inicializan los parámetros.
- Función loop() en el cual se ejecuta el software programado.

Para la configuración respectiva de Acceleo, se debe crear una nueva configuración mediante la perspectiva de Acceleo que permite seleccionar el proyecto, la clase principal, el modelo que el usuario está editando, y target que es el proyecto creado.

Figura 41

Configuración de Acceleo



Acceleo utiliza sus propias sentencias para generar la plantilla preestablecida, para lograr este objetivo se sigue una serie de procesos en la plantilla generada para llegar a generar código funcional para la placa Arduino UNO.

Figura 42

Cabeceras de la Plantilla

```
[comment encoding = UTF-8 /]
[module generate('http://www.example.org/circuitoV3')]
[template public generateElement(aSistema : Sistema)]
[comment @main/]
[file ('aSistema.ino', false, 'UTF-8')]
```

Como se puede observar en la figura 42, se encuentran definidas las cabeceras de la plantilla generada, tienen que ver con el formato de codificación de caracteres UTF-8, la URI registrada donde obtiene el metamodelo, la plantilla que hace mención de la clase Sistema como clase raíz contenedora y por último como se va a llamar el archivo generado.

A continuación, se describe como se encuentra formada la función SETUP:

Figura 43

Setup para elemento físico Pulsador

```
//pulsador
[for (pulsador : Pulsador | aSistema.elementos)]
        pinMode([pulsador.numero_pin/],INPUT);
[/for]
```

Para que la placa Arduino UNO entienda que los pulsadores son entradas físicas, es necesario configurar esto definiendo el pulsador como INPUT con número de pin, de esta manera, se entiende que el pulsador es un elemento físico que tiende a cambiar.

La figura 43 describe la instanciación de todos los elementos que correspondan a la clase Pulsador mediante el ciclo For, la sentencia pinMode() es propia de Arduino, pero aquí asigna el valor numérico que posee cada pulsador y la palabra INPUT.

Figura 44

Setup para elemento físico LED

```
//led
[for (led : Led | aSistema.elementos)]
        pinMode([led.numero_pin/],OUTPUT);
[/for]
```

Para la programación de una placa Arduino es necesario definir los elementos que van a servir como entradas y salidas, en este caso los Leds con elementos físicos que sirven como salidas, por este motivo, los Leds que se implementan a través de un modelo creado se van a crear como elementos OUTPUT y la placa los pueda entender.

La figura 44 describe la instanciación de todos los elementos que correspondan a la clase Led mediante el ciclo For, la sentencia pinMode() es propia de Arduino, pero aquí asigna el valor numérico que posee cada Led y la palabra OUTPUT.

Una vez configurada todos los elementos necesarios para entradas y salidas, se procede a ejecutar el programa dentro de la función LOOP.

Figura 45

Declaración de función digitalRead para los pulsadores

```
//Estado Pulsador
[for (pulsador : Pulsador | aSistema.elementos)]
        boolean [pulsador.nombreElemento/]=digitalRead([pulsador.numero_pin/]);
[/for]
```

El estado del pulsador permite conocer cómo va a funcionar el siguiente elemento, si debe encenderse o debe apagarse, por este motivo, mediante la función digitalRead que entiende Arduino se pasa los parámetros para establecer los estados de los diferentes pulsadores creados.

La figura 45 describe mediante el ciclo For, la creación de una variable de tipo boolean que toma el nombre del elemento Pulsador y asigna un valor de falso, para que así no exista inconvenientes con otros elementos.

Figura 46

Establecimiento de estados para las compuertas AND OR y NOT

```
//Estado Compuerta
[for (compuerta : Compuerta | aSistema.elementos)]
        boolean [compuerta.nombreElemento/]= false;
[/for]
```

Antes de iniciar con el proceso de cambiar los estados de las diferentes compuertas, es necesario declarar las variables con un estado en false para no obtener errores de inicialización de variables por parte del IDE de Arduino.

Para actualizar los diferentes estados de las compuertas lógicas creadas a través del circuito, es necesario recorrer todos los nodos creados y verificar el elemento anterior con su estado. De esta manera se actualiza los estados de las compuertas y seguir con el último paso que es la actualización de los Leds.

Figura 47

Actualización de estados de las compuertas AND

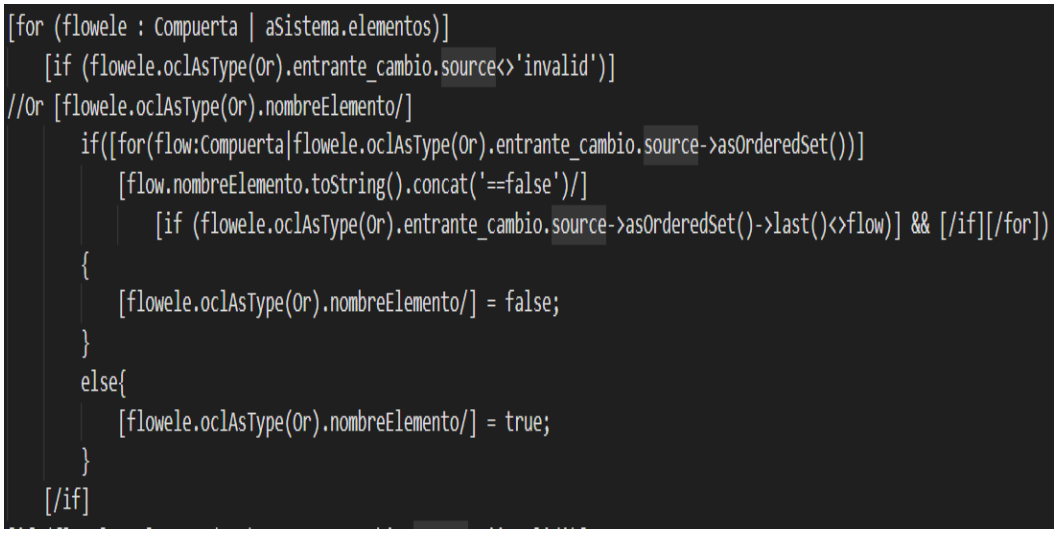
```
[for (flowele : Compuerta | aSistema.elementos)]
   [if (flowele.oclAsType(And).entrante_cambio.source<>'invalid')]
//AND [flowele.oclAsType(And).nombreElemento/]
       if([for(flow:Compuerta|flowele.oclAsType(And).entrante_cambio.source->asOrderedSet())]
           [flow.nombreElemento.toString().concat('==true')/]
               [if (flowele.oclAsType(And).entrante_cambio.source->asOrderedSet()->last()<>flow)] && [/if][/for])
       {
           [flowele.oclAsType(And).nombreElemento/] = true;
       }
       else{
           [flowele.oclAsType(And).nombreElemento/] = false;
       }
   [/if]
```

Para obtener todos los elementos de tipo Compuerta AND, es necesario recorrer todos los elementos dentro de la clase principal Sistema. Por cada compuerta de tipo AND que existe dentro del ciclo FOR, se ordena según la primera compuerta que fue creada por el usuario mediante el método asOrderedSet, luego, se concatena el nombre

de la compuerta con una igualación a true y si existen más datos del flujo de entrada si siguen concatenando caso contrario ya no se agregan más “&&”, que significa que tiene que cumplirse todas las condiciones para cumplir con el bloque de código. Por último, se establece el estado de la compuerta lógica en true o false.

Figura 48

Actualización de estados de las compuertas OR



Para obtener todos los elementos de tipo Compuerta OR, es necesario recorrer todos los elementos dentro de la clase principal Sistema. Por cada compuerta de tipo OR que existe dentro del ciclo FOR, se ordena según la primera compuerta que fue creada por el usuario mediante el método asOrderedSet, luego, se concatena el nombre de la compuerta con una igualación a true y si existen más datos del flujo de entrada si siguen concatenando caso contrario ya no se agregan otro signo de “&&” que significa que tiene que cumplirse todas las condiciones para cumplir con el bloque de código. Por último, se establece el estado de la compuerta lógica en true o false.

Figura 49

Actualización de estados de las compuertas NOT

```
[for (flowele : Compuerta | aSistema.elementos)]
[if (flowele.oclAsType(Not).entrante_cambio.source<>'invalid')]
//Not [flowele.oclAsType(Not).nombreElemento/]
        if([for(flow:Compuerta|flowele.oclAsType(Not).entrante_cambio.source->asOrderedSet())]
            [flow.nombreElemento.toString().concat('==true')/][/for])
        {
            [flowele.oclAsType(Not).nombreElemento/] = false;
        }
        else{
            [flowele.oclAsType(Not).nombreElemento/] = true;
        }
    [/if]
[/for]
```

Para actualizar el estado de las compuertas NOT implementadas en el circuito lógico diseñado por el usuario, es necesario recorrer todos los elementos dentro de nuestra clase principal Sistema, luego hay que verificar si el elemento es de tipo NOT y se añaden las diferentes validaciones que pueda tener la compuerta, dependiendo del elemento anterior. Por último, se establece el estado final de la compuerta NOT con true o false según corresponda.

Figura 50

Actualización de estado de los LEDS

```
[for (componente : Componente | aSistema.elementos)]
[if (componente.oclAsType(Led).entrante_cambio.source<>'invalid')]
//LED  [componente.oclAsType(Led).nombreElemento/]
        if([for(flow:Componente|componente.oclAsType(Led).entrante_cambio.source->asOrderedSet())]
        [flow.nombreElemento.toString().concat('==true')/][/for])
        {
            digitalWrite([componente.oclAsType(Led).numero_pin/], HIGH);
        }
        else{
            digitalWrite([componente.oclAsType(Led).numero_pin/], LOW);
        }
[/if]
[/for]
```

Arduino a través de la función digitalWrite() entiende cuando es necesario prender o apagar un Led, por este motivo mediante la plantilla se actualiza el estado del Led a través del elemento anterior, es decir, si el elemento anterior se encuentra en un estado false el Led se encuentra apagado, pero si el elemento anterior se encuentra en un estado true el Led se encuentra encendido.

Cuando el usuario ya ha armado un circuito lógico combinacional y quiere obtener el código necesario para implementarlo en la placa Arduino, es tan simple como abrir el proyecto de Acceleo y en el archivo generate.yml dar click derecho y seleccionar Run As Launch Acceleo Application. Como se puede observar en la figura 51, lo que da como resultado código funcional que debe ser compilado mediante el IDE de Arduino como se puede observar en la figura 52.

Figura 51

Run As Launch Acceleo Application

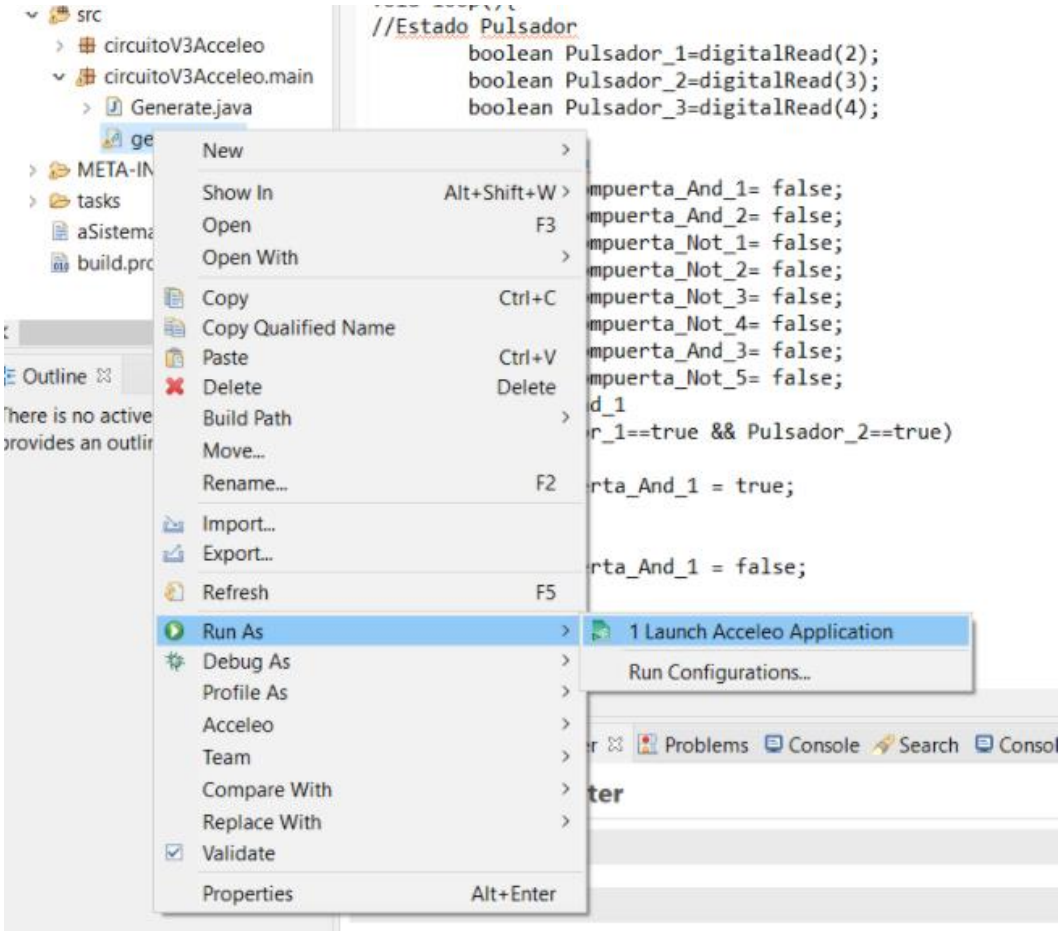


Figura 52

Código funcional generado para placa Arduino

new sistema | aSistema.ino | circuitoV3.odesign

```
void setup(){
//pulsador
        pinMode(2,INPUT);
        pinMode(3,INPUT);
        pinMode(4,INPUT);
        pinMode(5,INPUT);
//led
        pinMode(6,OUTPUT);
}
void loop(){
//Estado Pulsador
        boolean Pulsador_1=digitalRead(2);
        boolean Pulsador_2=digitalRead(3);
        boolean Pulsador_3=digitalRead(4);
        boolean Pulsador_4=digitalRead(5);

//Estado Compuerta
        boolean Compuerta_OR_1= false;
        boolean Compuerta_OR_2= false;
        boolean Compuerta_OR_3= false;
        boolean Compuerta_OR_4= false;
        boolean Compuerta_And_1= false;
        boolean Compuerta_And_2= false;
        boolean Compuerta_Not_2= false;
//Or Compuerta_OR_1
        if(Pulsador_1==false && Pulsador_2==false)
        {
            Compuerta_OR_1 = false;
        }
        else{
            Compuerta_OR_1 = true;
        }
//Or Compuerta_OR_2
        if(Pulsador_3==false && Pulsador_1==false)
        {
            Compuerta_OR_2 = false;
        }
        else{
            Compuerta_OR_2 = true;
        }
//Or Compuerta_OR_3
```

Capítulo IV

3. Validación y Pruebas del DSL

El objetivo de este capítulo es responder a las preguntas planteadas para la investigación, aplicando una evaluación empírica que contrasta con los indicadores de la investigación y una evaluación de usabilidad.

Además, se determina la validez y la efectividad del DSL para programar la placa Arduino UNO mediante compuertas lógicas AND, NOT y OR. Se realizó y validó las pruebas con 10 personas que fueron escogidas por sus conocimientos en el dominio de la programación para placa Arduino, IDE Arduino, armar circuitos lógicos, conocimiento acerca de IDE Eclipse e instalación de plug-in, herramientas de versionado de código y sobre todo que cuenten con un computador e Internet.

El cálculo del tamaño de la muestra es una función matemática que expresa la relación entre las variables, cantidad de participantes y poder estadístico, para el cálculo de la cantidad de participantes a incluirse en la investigación se utilizó un nivel de confianza del 95%, con un error del 27% a una población de 30 personas con conocimientos previos, lo que obtuvo como resultado una muestra de 10 participantes.

4.1 Evaluación empírica

Se plantean tres ejercicios de baja, media y alta complejidad a ser probados tanto en la herramienta desarrollada como en la herramienta open source Visualino y la herramienta propietaria Logo Soft, que es un entorno de desarrollo para micro PCLs Logo Siemens, midiendo tiempos de implementación en la resolución de diferentes ejercicios. A continuación, se mide las líneas generadas para la placa Arduino UNO, incluyendo la programación manual a través del IDE de Arduino y las demás herramientas propuestas, a su vez se evaluará la usabilidad de la aplicación conforme lo establecido en la norma ISO/IEC 9126 respecto a las métricas de usabilidad como lo son: métricas de entendibilidad, facilidad de aprendizaje, operabilidad, y atractividad.

3.1.1. Ejercicio No. 1

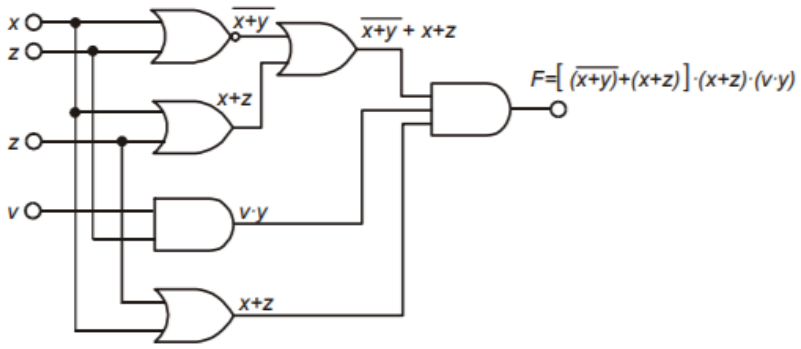
Como primera práctica a comparar se plantea un ejercicio de alta complejidad puesto que posee varios pulsadores y tres tipos diferentes de compuertas, el ejercicio de la figura 53, se pone a prueba una sola salida, misma que se ve condicionada por compuertas AND, OR y NOT, y cuatro entradas, la salida tendrá el valor de uno si:

- Todas las entradas están encendidas.
- Si x, y, v están encendidas.
- Si y, v, z están encendidas.

caso contrario la salida será igual a cero.

Figura 53

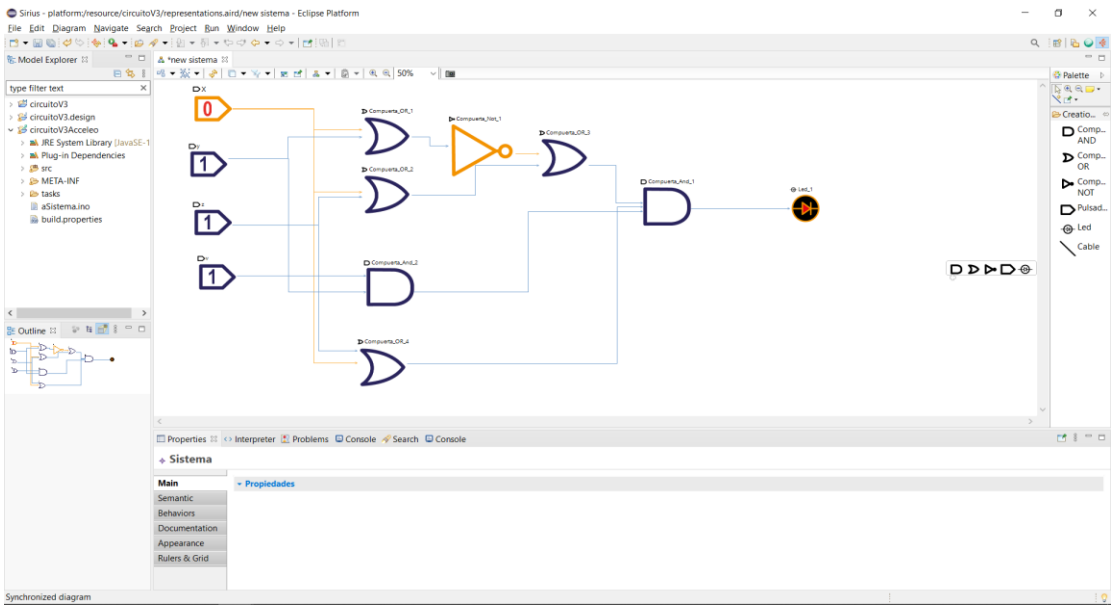
Ejercicio No. 1



Nota. Obtenido de (Prieto Torralbo & Abad)

Figura 54

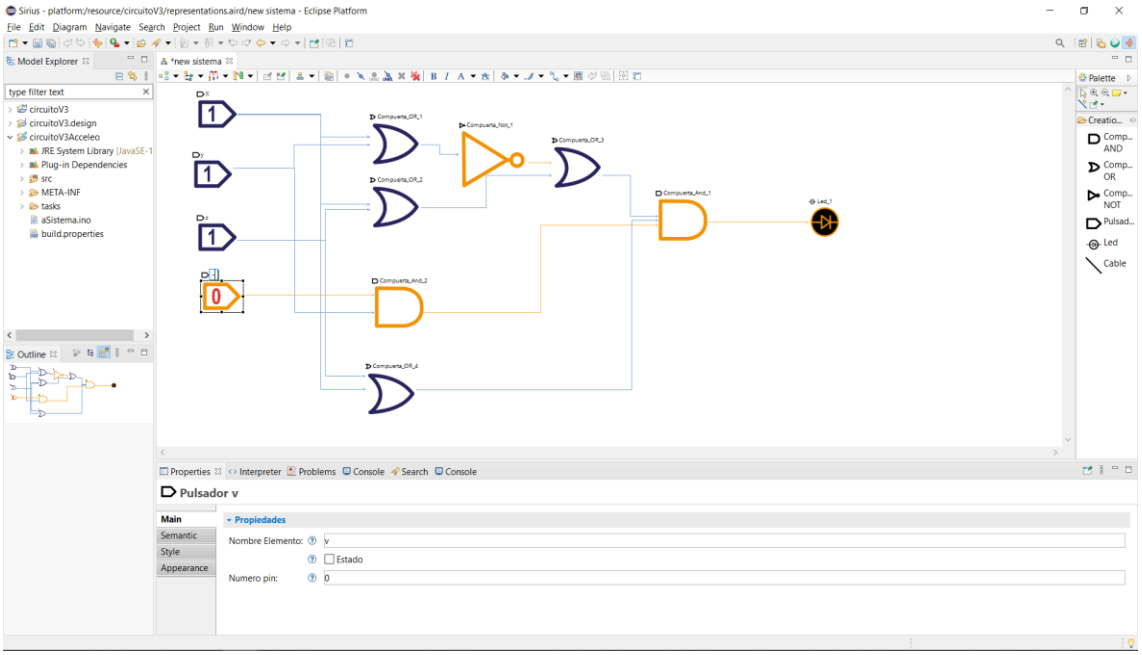
Ejercicio No.1 desarrollado con el DSL



Nota. Pulsador 1 apagado y el resto de los pulsadores encendidos

Figura 55

Ejercicio No.1 desarrollado con el DSL con diferente configuración



Nota. Pulsador 1, 2 y 3 encendidos y pulsador 4 apagado.

Figura 56

Declaración de variables de Ejercicio No.1 en Visualino

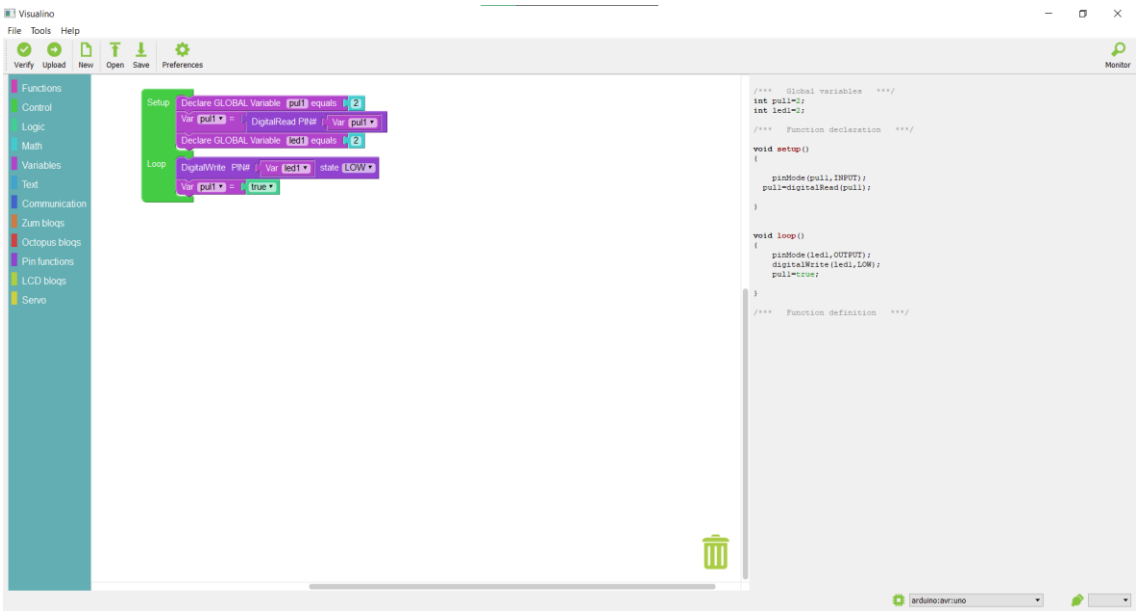


Figura 57

Desarrollo del Ejercicio No.1 en Logo Soft

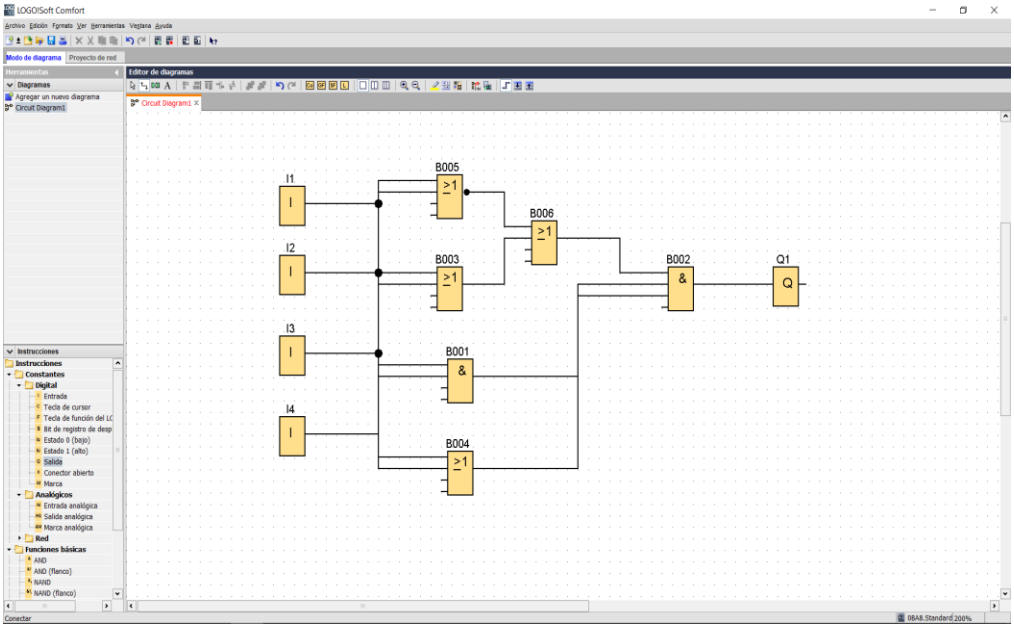


Figura 58

Programación del Ejercicio No.1 con el IDE de Arduino

Tabla 6

Tiempo empleado por herramienta - Ejercicio 1

Herramienta	**Tiempo empleado**
DSL desarrollado	5 min
Visualino	17 min
Logo Soft	3 min
IDE Arduino	8 min

Como se puede observar en la tabla 6, el tiempo empleado estimado por cada herramienta varía dependiendo de la finalidad que tiene cada herramienta, por su lado Visualino buscar mejorar la programación de Arduino, mediante gráficos desde la declaración de variables hasta la implementación. Al ejercicio No.1 se lo puede considerar como un ejercicio moderadamente sencillo, por lo que la implementación no se debe llevar a cabo en mucho tiempo. Por este motivo el desarrollo del DSL es totalmente válido, por lo que genera código para la placa Arduino en un tiempo conforme al resto de herramientas, especialmente en comparación al proceso con el IDE de Arduino donde se ahorra 3 min utilizando el DSL.

Tabla 7

Líneas de código generadas por herramienta - Ejercicio 1

Herramienta	**Líneas de código generadas**
DSL desarrollado	109 líneas
Visualino	90 líneas
Logo Siemens	N/A
IDE Arduino	60 líneas

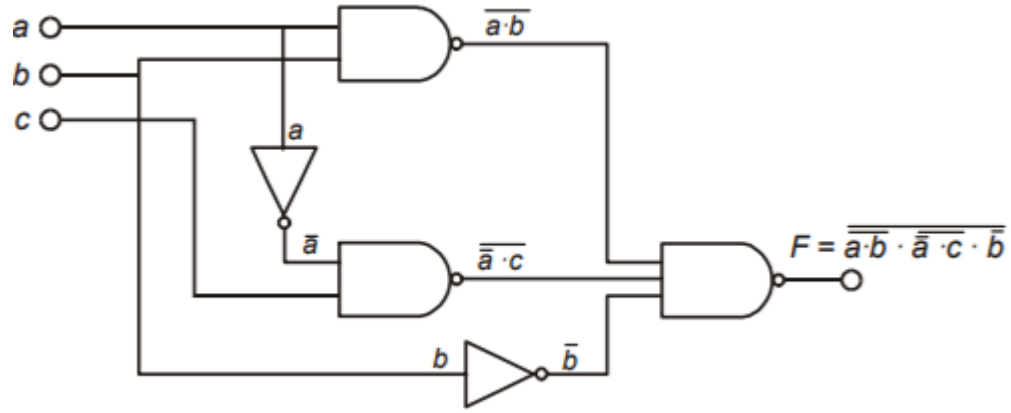
Las líneas generadas por las diferentes herramientas empleadas no tienen mucha variación, donde la herramienta desarrollada generó en mayor medida mayor cantidad de código con un 42%, Visualino con un 35% y código generado de forma Manual el 23%. De cierta manera el DSL genera mayor cantidad de código, pero es por la manera con el que está construido, es decir, la manera en que se genera código, no se tiene control de cómo va estructurado un ciclo condicional en una línea o en tres líneas que se puede hacer normalmente, indiferente de estos detalles la herramienta genera código válido y funcional en el ejercicio No.1 con todas las entradas y variaciones que puede tener.

3.1.2. Ejercicio No. 2

El siguiente ejercicio pone en práctica la compuerta NOT, la cual es colocada después de otra compuerta o de forma independiente, considerado como un ejercicio de media complejidad al poseer la compuerta AND y NOT interactuando entre sí y a su vez de tres pulsadores.

Figura 59

Ejercicio No.2



Nota. Obtenido de (López Hernández)

Figura 60

Ejercicio No.2 Desarrollado con el DSL

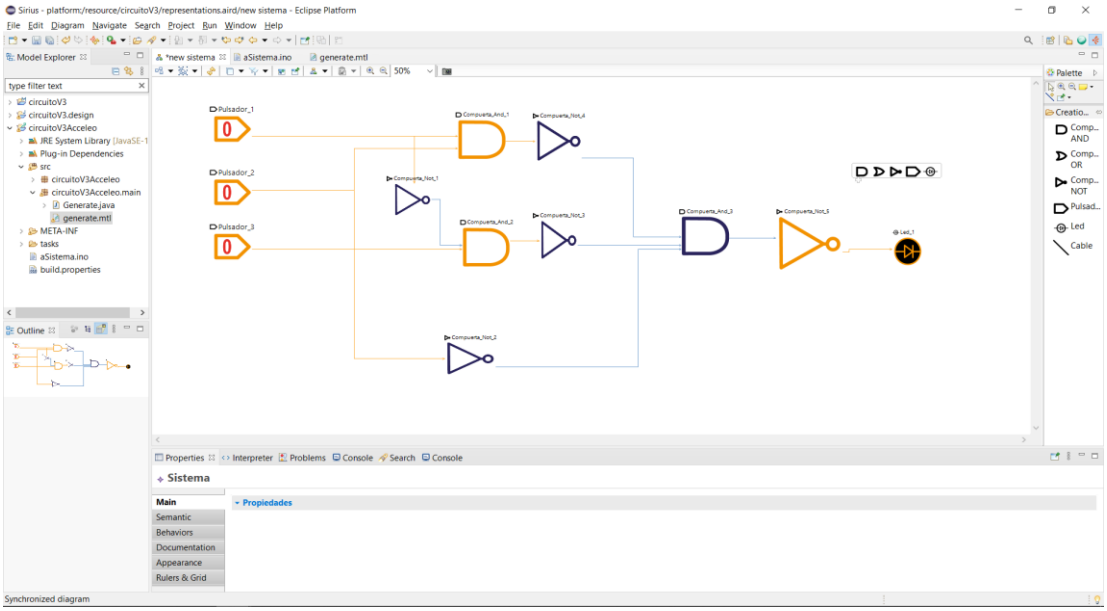


Figura 61

Ejercicio No.2 Desarrollado con el DSL con variaciones de entradas

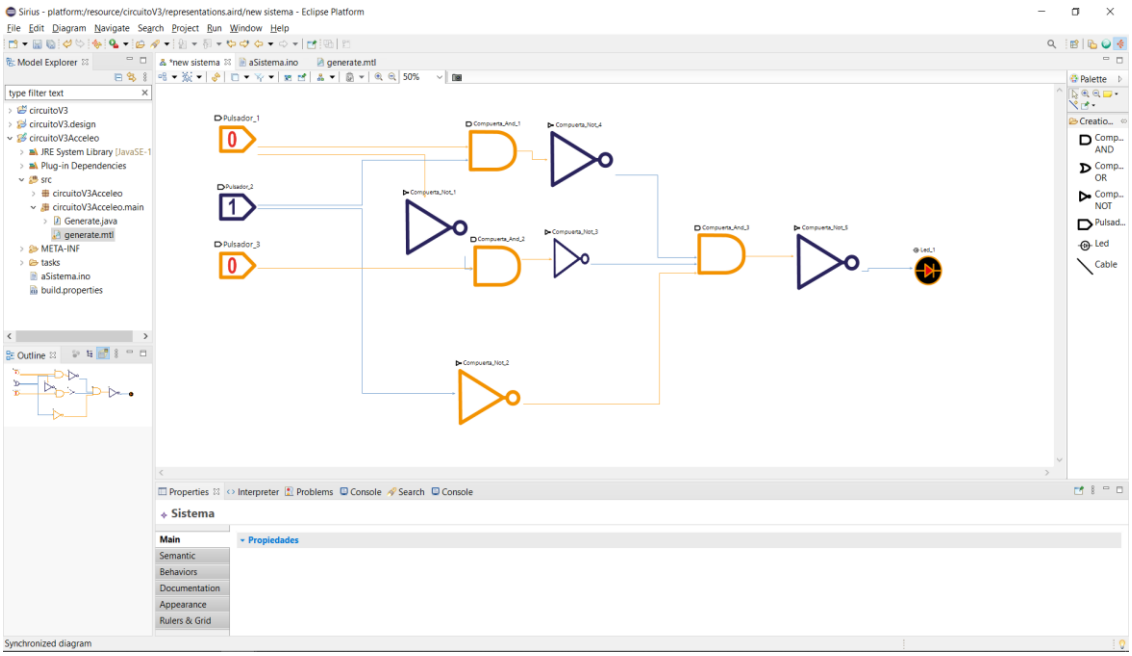


Figura 62

Implementación Ejercicio No.2 con Visualino

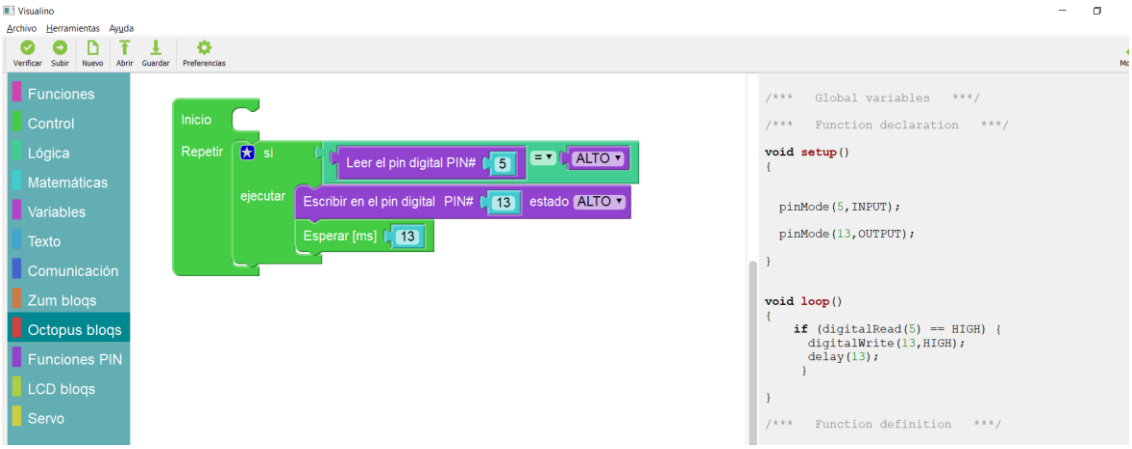


Figura 63

Implementación Ejercicio No.2 con Logo Soft.

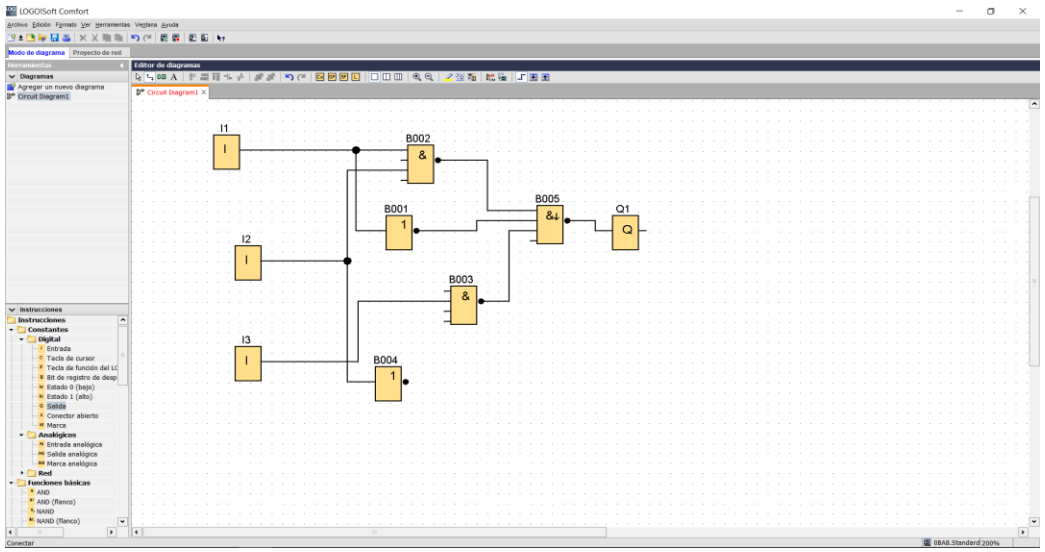


Figura 64

Implementación Ejercicio No.2 mediante la programación manual

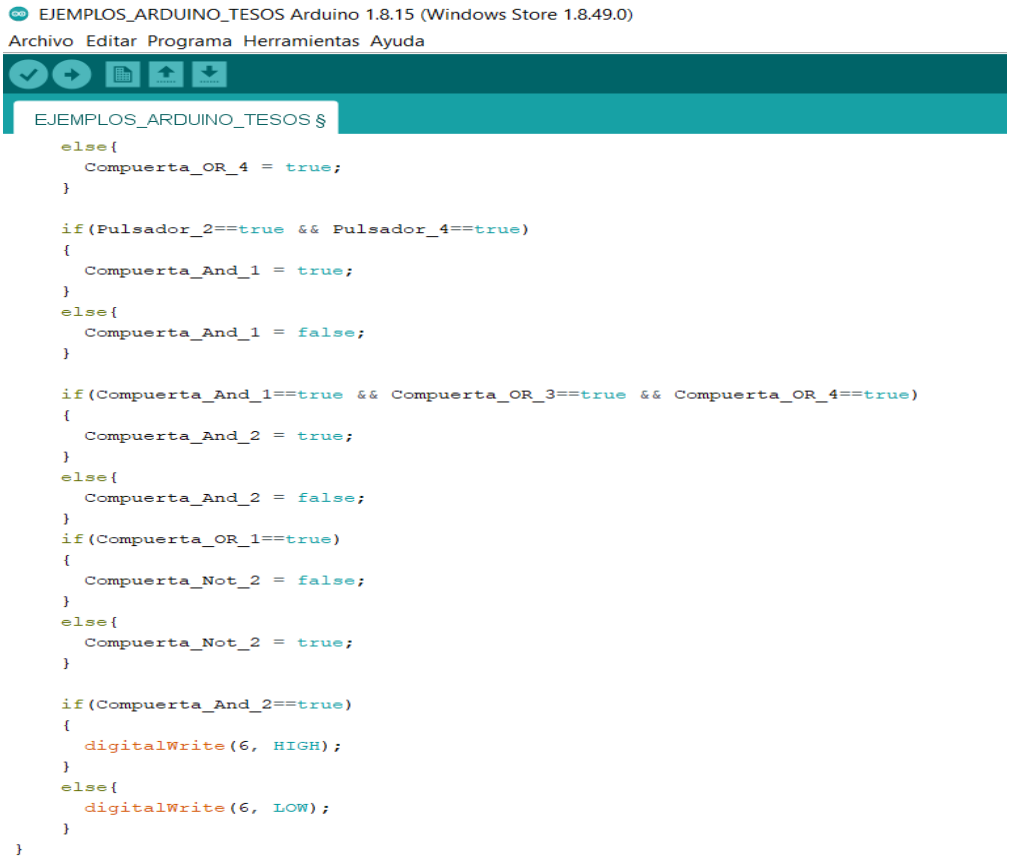


Tabla 8

Tiempo empleado por herramienta - Ejercicio 2

Herramienta	**Tiempo empleado**
DSL desarrollado	7 min
Visualino	15 min
Logo Soft	3 min
IDE Arduino	11 min

Como se puede observar en la tabla 8, el tiempo empleado estimado por cada herramienta varía dependiendo de la finalidad que tiene cada herramienta. Al ejercicio No.2 se lo puedo considerar como un ejercicio con una complejidad media, por lo que la implementación debe tener un tiempo promedio igual o mayor al Ejercicio No.1. El tiempo empleado por el DSL es 7 min, una cantidad de tiempo totalmente por debajo del promedio en comparación a los 17 min empleados por Visualino o los 11 min empleados

por el IDE Arduino. En comparación a Logo Soft, el tiempo es mucho menor, porque la compilación es directa al PCL y las compuertas ya vienen con su respectiva negación.

Tabla 9

Líneas de código generadas por herramienta - Ejercicio 2

Herramienta	**Líneas de código generadas**
DSL desarrollado	96 líneas
Visualino	89 líneas
Logo Siemens	N/A
IDE Arduino	73 líneas

Las líneas generadas por las diferentes herramientas empleadas no tienen mucha variación, donde la herramienta desarrollada generó en mayor medida mayor cantidad de código con 96 líneas generadas. Visualino con 89 líneas y código generado de forma Manual con una cantidad de líneas de código equivalentes a 73. De cierta manera, el DSL genera mayor cantidad de código, pero es por la manera con el que está generado, no se tiene control de cómo va estructurada un ciclo condicional en una línea o en tres líneas que se puede hacer normalmente, indiferente de estos detalles la herramienta genera código válido y funcional probado con en el ejercicio No.2.

3.1.3. Ejercicio No. 3

Como tercera práctica a comparar se plantea un ejercicio de baja complejidad dadas las pocas interacciones entre pulsadores y compuertas, el ejercicio de la figura 56 consiste en tener cuatro salidas, mismas que son condicionadas por 2 elementos de entrada y distintas compuertas lógicas, el resultado del ejercicio responde a la siguiente figura 65

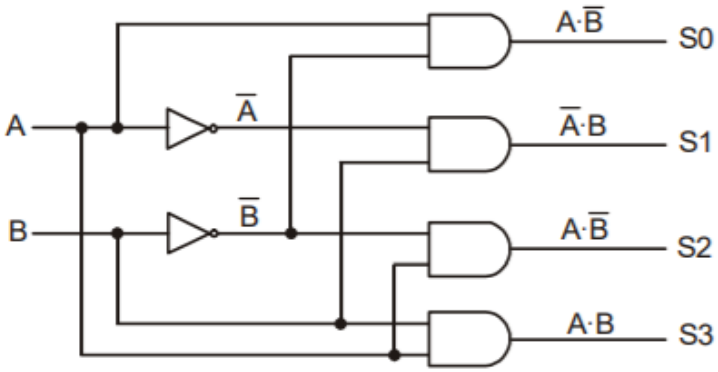
Figura *65*

Tabla de verdad Ejercicio 3.

A	B	S_0	S_1	S_2	S_3
0	0	0	0	0	0
0	1	0	1	0	0
1	0	1	0	1	0
1	1	0	0	0	1

Figura 66

Ejercicio planteado No.3



Nota. Ejercicio obtenido de (López Hernández)

Figura 67

Ejercicio No.3 Implementando con el DSL

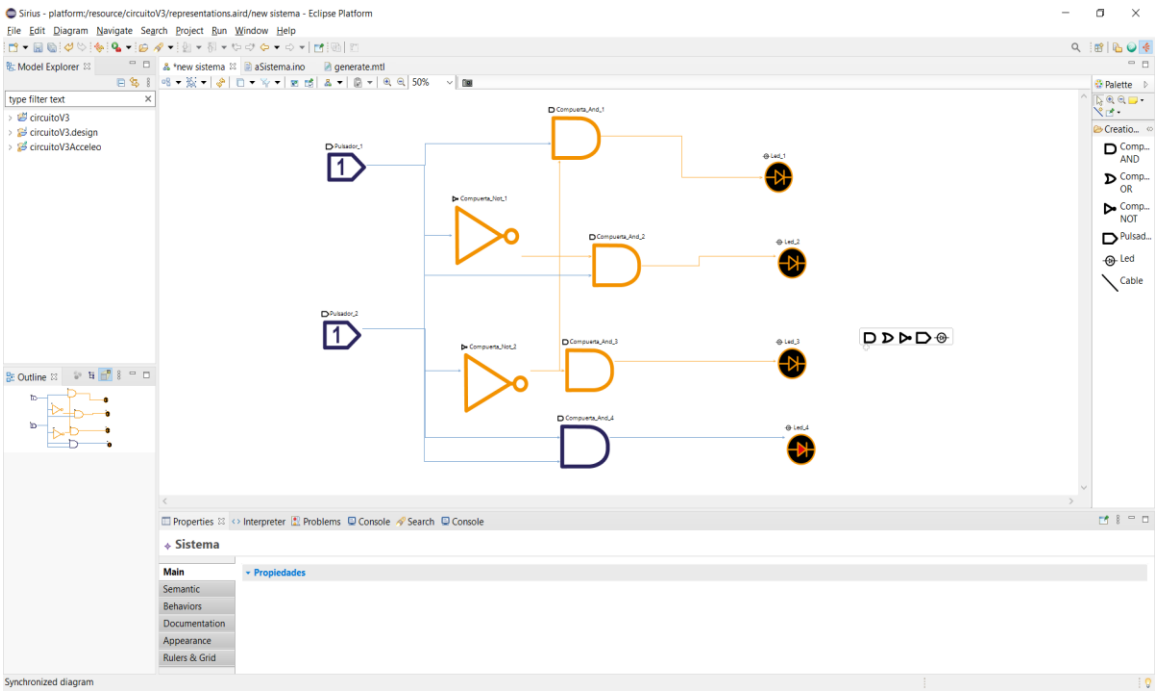


Figura 68

Creación de ejercicio No.3 con Visualino

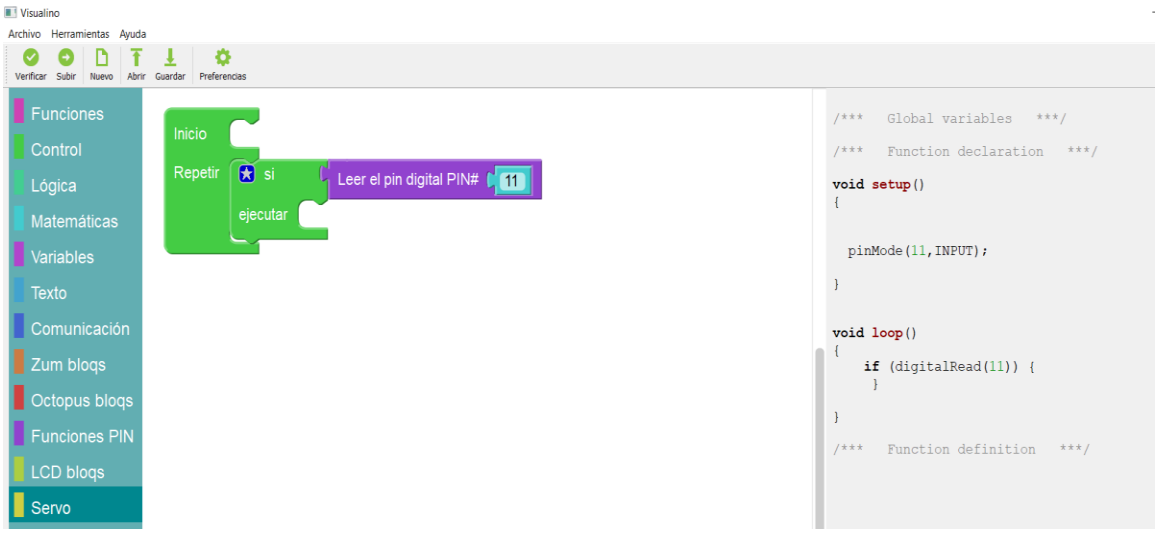


Figura 69

Implementación Ejercicio No.3 con Logo Soft

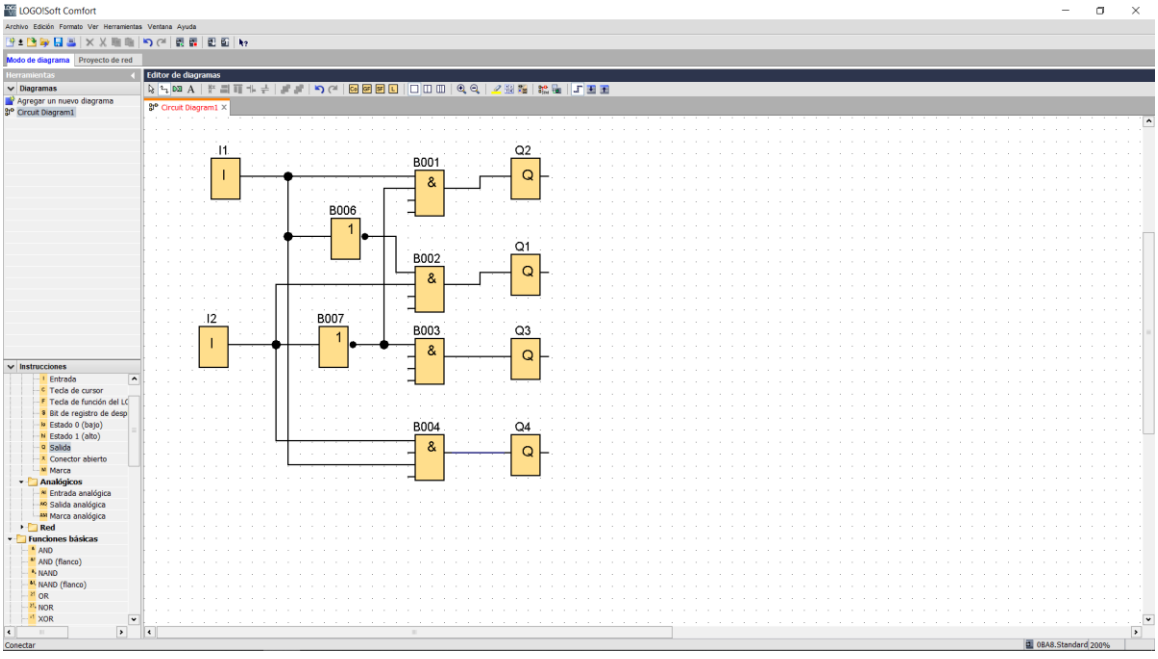


Figura 70

Compilación de Ejercicio No.3 mediante IDE Arduino

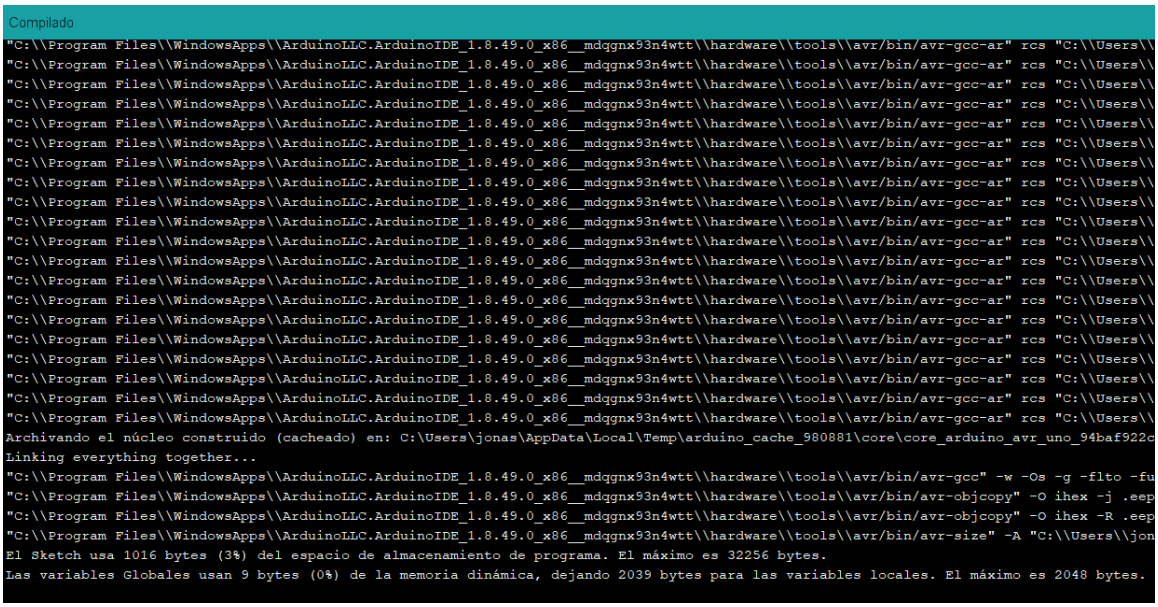


Tabla 10

Tiempo empleado por herramienta - Ejercicio 3

Herramienta	**Tiempo empleado**
DSL desarrollado	6 min
Visualino	12 min
Logo Siemens	3 min
IDE Arduino	9 min

Como se puede observar en la tabla 10, el tiempo empleado estimado por cada herramienta varía dependiendo de la finalidad que tiene cada herramienta. Por su lado, el DSL tiene como finalidad armar circuitos lógicos combinacionales con compuertas lógicas básicas. Visualino tiene como fin enseñar a programar para Arduino. Logo Soft programar sus PLCs y IDE Arduino generar código de forma natural, es decir, escribir código. La complejidad del Ejercicio No.3 varía por lo que se ha aumentado ciertas compuertas negadas y el número de compuertas necesarias para obtener un resultado, pero la implementación no debe demorarse demasiado. El tiempo empleado por el DSL es 6 min, una cantidad de tiempo totalmente por debajo del promedio en comparación a los 12 min empleados por Visualino o los 9 min empleados por el IDE Arduino. En

comparación a Logo Soft el tiempo es mucho menor por que la compilación es directa al PCL y las compuertas ya vienen con su respectiva negación.

Tabla 11

Líneas de código generadas por herramienta Ejercicio 3

Herramienta	**Líneas de código generadas**
DSL desarrollado	90 líneas
Visualino	70 líneas
Logo Siemens	N/A
Código generado manualmente	60 líneas

Las líneas generadas por las diferentes herramientas empleadas no tienen mucha variación, donde la herramienta desarrollada generó en mayor medida mayor cantidad de código con 90 líneas generadas. Visualino con 70 líneas y código generado de forma Manual con una cantidad de líneas de código equivalentes a 60. De cierta manera el DSL genera mayor cantidad de código, pero es por la manera con el que está generado, la herramienta genera código válido y funcional probado con en el ejercicio No.3.

El tiempo empleado por cada herramienta tiene que ver con la finalidad que tiene cada una de estas, Visualino por su lado tiene como objetivo aprender programación para Arduino, mediante la construcción de código con gráficos, por lo que emplea mucho más tiempo para realizar un circuito en específico, ya que se debe declarar hasta las variables de manera gráfica y esto genera cierta demora para graficar un circuito con compuertas lógicas, Logo Soft está enfocado a la programación de sus PCL de manera directa, evitando la programación de estos manualmente a través de un IDE, mientras que, por su lado el DSL desarrollado para realizar ejercicios de circuitos básicos con compuertas lógicas, toma un tiempo empleado sumamente menor, por lo que solo se debe arrastrar las figuras y generar el código, por lo cual, es sumamente sencillo utilizar esta herramienta, cabe recalcar que tanto las herramientas señaladas como el DSL desarrollado, cumplen con el objetivo de programar compuertas lógicas.

Las líneas de código generadas independientemente de las herramientas utilizadas no varían en gran cantidad, depende de la generación de cada una de estas que tiene que ver con la manera de declaración de variables, si se aplica o no comentarios, la manera de ubicar los corchetes, una herramienta puede tener una función que esta expresada en 10 líneas, pero otra herramienta puede tener la misma función

expresada en 3 líneas, pero hacen completamente lo mismo. Todas las herramientas generan código válido y similar en cierta manera, que hace lo que tiene que hacer según el ejercicio diagramado, en comparación del DSL desarrollado con las otras herramientas genera un poco más de código, porque están implementados comentarios para entender de mejor manera el código.

4.2 Evaluación de usabilidad

EL DSL implementado en un alto nivel de abstracción, tiene como objetivo diseñar circuitos lógicos, de esta manera los conceptos se acercan más al área de dominio para cada usuario. Para comprobar la correcta implementación del DSL y su usabilidad, se implementó una encuesta a 10 usuarios con conocimientos en programación de Arduino y conocimientos para instalar programas. Según la norma ISO/IEC 9126, la usabilidad permite una mayor velocidad y eficiencia en la realización de las tareas por parte del usuario, por lo que de forma general optimiza el rendimiento de la aplicación desarrollada y sobre todo la experiencia de usuario midiendo el grado en el que el software puede ser comprendido, aprendido, operado, atractivo y cumplen con las normas y directrices de usabilidad.

Las métricas de usabilidad en su gran medida son externas por lo que son probadas por los usuarios que utilizan el producto software. Este producto es evaluado bajo ciertas condiciones específicas, donde los usuarios deben realizar la prueba sin ninguna pista o ayuda externa.

Las métricas de usabilidad según la norma ISO/IEC 9126 describen las siguientes métricas, métricas de entendimiento, métricas de facilidad de aprendizaje, métricas de operabilidad, métricas de atractividad y métricas de conformidad, Según estás métricas, se ha establecido una serie de preguntas que nos permiten medir la usabilidad del proyecto desarrollado, por lo cual, se describe a continuación los resultados obtenidos:

Pregunta 1

¿Considera usted que el proyecto cumple con el objetivo de armar circuitos lógicos combinacionales a través de compuertas lógicas?

Tabla 12

Objetivo de armar circuitos lógicos combinacionales

Respuesta	Frecuencia	Porcentaje
Si	9	90%
No	0	0%
Tal vez	1	10%
Total	**10**	**100%**

El objetivo del DSL es armar circuitos lógicos combinacionales, mediante compuertas lógicas básicas a través de una interfaz que permita arrastrar y soltar objetos, por lo que, según los resultados obtenidos, se entiende cual es el fin y el objetivo de la aplicación desarrollada en el presente proyecto.

Pregunta 2

Seleccione todas las funcionalidades que pueda identificar luego de su primera interacción con la aplicación.

Tabla 13

Funcionalidades después de primera interacción con aplicación

Respuesta	Frecuencia	Porcentaje
- Armar circuitos lógicos - Exportar código - Simular circuitos lógicos	4	40%
- Armar circuitos lógicos - Simular circuitos lógicos	2	20%
- Armar circuitos lógicos - Exportar código	1	10%
- Exportar código - Simular circuitos lógicos	3	30%
Total	**10**	**100%**

Las funcionalidades que ofrece el DSL son armar circuitos lógicos, exportar código para la placa Arduino y simular circuitos lógicos como principales funcionalidades dentro del proyecto desarrollado. Estas funcionalidades fueron entendidas por la mayoría de los

usuarios que participaron de la actividad independientemente de la funcionalidad, unos no entendieron como generar código o simular, pero para este existe el manual de usuario que ayuda a comprender de mejorar manera y es así como se vuelve a entender cuál es el objetivo del DSL.

Pregunta 3

¿Considera usted que fue complicado crear un circuito lógico básico?

Tabla 14

Dificultad para crear un circuito lógico básico

Respuesta	Frecuencia	Porcentaje
1 (100% de dificultad)	1	10%
2 (75%de dificultad)	2	20%
3 (50% de dificultad)	0	0%
4 (25% de dificultad)	5	50%
5 (0% de dificultad)	2	20%
Total	**10**	**100%**

La dificultad de un circuito lógico se basa en el número de entradas y el número de compuertas lógicas que se quiere implementar. El ejercicio propuesto dentro de evaluación de usabilidad es el ejercicio No.3 que se puede evidenciar en la Figura *65*

Tabla de verdad Ejercicio 3. Por lo que, el nivel de dificultad estuvo por el 25% en promedio aplicando el DSL por la facilidad de uso que tiene al simplemente arrastrar y soltar objetos.

Pregunta 4

¿Considera usted complicado el proceso de instalación del Lenguaje de Dominio Específico?

Tabla 15

Dificultad del proceso de instalación del Lenguaje de Dominio Específico

Respuesta	Frecuencia	Porcentaje
1	0	0%
2	0	0%
3	2	20%
4	7	70%
5	1	10%
Total	**10**	**100%**

El proceso de instalación del DSL a través de Eclipse Modelling Tools fue el proceso más complejo de llevar a cabo por motivos de instalación de plug-in, instalación de proyecto y manipulación de este mismo, por lo cual ha sido necesario la implementación de un Manual de Usuario descrito en el capítulo III.

Pregunta 5

¿Considera usted el uso de un manual de usuario sobre el Proyecto DSL?

Tabla 16

Implementación de Manual de instalación del DSL

Respuesta	Frecuencia	Porcentaje
Si	8	80%
No	1	10%
Tal vez	1	10%
Total	**10**	**100%**

Según los resultados obtenidos en el proceso de instalación del DSL se planteó si es necesario crear un manual de instalación que se puede usar en casos necesarios, donde se tenga complicaciones para esto el 80% de los participantes sugirieron que, si se implemente el Manual de Usuario, mientras que el 20% se encuentran por el sí y por

el tal vez respectivamente, para la creación del manual se maneja un repositorio público donde la comunidad puede colaborar, lo cual permite el crecimiento del proyecto y la retrospectiva por parte de la comunidad que le interesa el tema.

Pregunta 6

¿Considera usted que fue complicado arrastrar y soltar una compuerta lógica?

Tabla 17

Complejidad para arrastrar objetos

Respuesta	Frecuencia	Porcentaje
1 (100% de complejidad)	2	20%
2 (75% de complejidad)	2	20%
3 (50% de complejidad)	1	10%
4 (25% de complejidad)	3	30%
5 (0% de complejidad)	2	20%
Total	**10**	**100%**

La facilidad que otorga el DSL desarrollado para simplemente arrastrar y soltar objetos mejora en la experiencia del usuario por la facilidad que se le ofrece. Un producto software siempre busca garantizar calidad y que el usuario no se aburra por no entender cómo usar, por este motivo es sumamente importante que el usuario pueda arrastrar y soltar una compuerta lógica según lo necesite.

Pregunta 7

¿Considera usted que fue complicado generar código para la placa Arduino UNO?

Tabla 18

Complejidad para generar código para placa Arduino UNO

Respuesta	Frecuencia	Porcentaje
1 (100% de complejidad)	1	10%
2 (75% de complejidad)	2	20%
3 (50% de complejidad)	1	10%
4 (25% de complejidad)	3	30%
5 (0% de complejidad)	3	30%
Total	**10**	**100%**

Otro punto importante dentro del proyecto DSL es la generación de código para la placa Arduino, uno de los objetivos primordiales para la creación del proyecto. La construcción de un circuito lógico básico termina con la generación de código necesario para la compilación para la placa Arduino. Los usuarios que participaron utilizando la herramienta pudieron generar código y observaron la facilidad que existe simplemente de compilar el código y probarlo, mediante una herramienta de simulación de Arduino o mediante la propia placa Arduino.

Pregunta 8

¿Pudo plantear y simular el ejercicio propuesto intuitivamente sin ninguna ayuda?

Tabla 19

Definición para determinar si el ejercicio propuesto es intuitivo

Respuesta	Frecuencia	Porcentaje
1 (Nada intuitivo)	0	0%
2 (Poco intuitivo)	0	0%
3 (Medianamente intuitivo)	2	20%
4 (Intuitivo)	5	50%
5 (Muy intuitivo)	3	30%
Total	**10**	**100%**

La parte instintiva de las personas es muy importante al momento de la ejecución de la actividad, por lo que dentro de los resultados se obtuvo que el 50% de los encuestados lograron realizar el planteamiento y la simulación de manera intuitiva, mientras que el 30% fue muy intuitivo en las tareas planteadas que debían realizar, con eso se demuestra que pocas son las personas que usan en mayor parte la parte de los instintos que el conocimiento que adquieren.

Pregunta 9

¿Cuánto tiempo estimado se tomó desarrollar el ejercicio planteado?

Tabla 20

Tiempo estimado para la implementación del ejercicio planteado

Respuesta	Frecuencia	Porcentaje
2 minutos	0	0%
4 minutos	4	40%
6 minutos	5	50%
8 minutos	1	10%
Total	**10**	**100%**

El tiempo empleado mediante la herramienta DSL debe ser sumamente rápido independientemente del ejercicio planteado, por lo que solo se debe armar el circuito según corresponda. Después de obtener el circuito armado se debe generar código y probarlo, ya sea en la propia placa Arduino o a través de un software.

Pregunta 10

¿Considera usted que los gráficos implementados representan adecuadamente a compuertas lógicas?

Tabla 21

Representación de gráficos adecuados

Respuesta	Frecuencia	Porcentaje
Si	9	90%
No	0	0%
Tal vez	1	10%
Total	**10**	**100%**

Para las métricas de atractividad de Usabilidad se trata de medir la capacidad del producto software de ser atractivo para el usuario, por este motivo la necesidad de gráficos adecuados es sumamente importante, por lo tanto, el 90% de los participantes si les pareció adecuados los gráficos implementados en el DSL desarrollado.

Pregunta 11

¿Considera usted que los colores implementados se ven agradables a la vista del usuario?

Tabla 22

Representación de colores adecuados

Respuesta	Frecuencia	Porcentaje
Si	9	90%
No	0	0%
Tal vez	1	10%
Total	**10**	**100%**

De acuerdo con los resultados obtenidos los colores implementados en el proyecto son agradables a la vista del usuario, lo que demuestra que la atractividad de los colores es válida y sumamente importante por la experiencia del usuario.

Como contraste de hipótesis se obtuvo que, se optimizó el tiempo de programación de código funcional para la placa Arduino Uno utilizando compuertas lógicas “AND”, “OR” y “NOT”, mediante las evaluaciones planteadas.

Capítulo V

4. Conclusiones y Recomendaciones

En este capítulo, se abordan conclusiones y recomendaciones obtenidos a lo largo del desarrollo e implementación del DSL para simular circuitos lógicos combinacionales mediante compuertas lógicas AND, OR y NOT.

5.1 Conclusiones

Se desarrolló un DSL que permite diseñar circuitos electrónicos en base a compuertas lógicas AND, OR y NOT; lo cual ha demostrado que se puede optimizar el tiempo de programación de código funcional para la placa Arduino UNO.

El alcance implementando por el desarrollo del DSL está enfocado hacia armar circuitos lógicos combinacionales, por lo cual las bases para futuros proyectos a partir del desarrollado permitirán la escalabilidad del producto software. En la recolección de información se pudo investigar sobre arquitectura de modelos, que permite la construcción de software para placas de desarrollo en este caso de Arduino UNO.

El DSL es una herramienta al usuario final (experto de dominio) que permite desarrollar software sin que este sea experto en los típicos lenguajes de programación porque le permite manejar conceptos de su cotidianidad y no sintaxis de un lenguaje de programación.

El metamodelo está planteado para separar los flujos tanto de salida como, de entrada, esto sirve para desarrollar una herramienta que sea capaz de escalar en módulos y elementos, representadas por la parte físicas, como la incorporación de periféricos de salida o entrada, o por la parte lógica ya sean clases o condiciones propias de los elementos, sin necesidad de implantar demasiados cambios si es necesario.

El editor gráfico utiliza nomenclatura electrónica propia de compuertas lógicas, pulsadores y leds, de este modo se intenta que el sistema ocupe una misma nomenclatura de sistemas ya existente, para que el usuario tenga una mejor idea de cómo armar circuitos lógicos dentro del proyecto, se busca que el editor sea lo más interactivo con el usuario, tomando en tiempo real los cambios de entradas para realizar las operaciones correspondientes , así cambiar los estados de los siguientes elementos conectados y mostrar lo más amigable posible en pantalla.

El generador de código desarrollado toma en cuenta las posibles combinaciones de los elementos y su interacción, genera código a través de una plantilla utilizada para interpretar los elementos del editor gráfico y transformar a código Arduino. El código generado resulta mayor en número de líneas de código en comparación con las herramientas contrastadas, puesto que la estructura implementada genera una serie de comentarios para que el usuario no tenga que entender el código, así mismo, el uso de corchetes se utiliza en 2 líneas lo que genera más líneas de código. El número de líneas resulta mayor pero conforme a usabilidad y funcionalidad garantiza resultados de código funcional y apto para su implementación.

Las métricas de usabilidad de entendimiento, aprendizaje, operabilidad, atractividad y complacencia de usabilidad según el estándar ISO/IEC 9126 se aplicaron en el aplicativo desarrollado y se pudo obtener que, cumple y garantiza el uso de calidad de software según las métricas aplicadas.

El DSL está disponible en un repositorio público de GitHub en la siguiente dirección: https://github.com/Jrfranco2/circuitoV3 dejando abierta la posibilidad de nuevas contribuciones que ayuden en el crecimiento del proyecto.

5.2 Recomendaciones

Para desarrollar software se recomienda la utilización de un DSL, porque permite desarrollar software utilizando un nivel más alto de abstracción, debido a que permite utilizar términos y definiciones más apegadas al problema que se quiere resolver, utilizando la notación que maneja el experto del dominio (usuario del software).

Generar la plantilla personalizada de Acceleo por segmentos, puesto que en la etapa de generación de código se trabaja con el modelo desarrollado y de esta manera no se mezclan elementos de otra sección como es el caso de flujos de entrada y flujos de salida, así aparte de ordenar el código resultante es más fácil la comprensión.

Resolver problemas de un dominio en particular o de un problema específico que requiere de un nivel de abstracción sumamente alto se recomienda como alternativa la implementación de un DSL para solventar este tipo de necesidades de la abstracción a un nivel bajo garantizando calidad y productividad en el desarrollo de aplicaciones.

Profundizar en la evaluación de calidad de productos software para mejorar la eficacia del software a través de normativas internacionales como lo es el estándar ISO/IEC 9126 que se enfoca en avaluar un software a través de métricas internas y externas que se unen en una sola obteniendo un modelo de calidad más completo.

Implementar una interfaz gráfica amigable al usuario, en la cual se respeten los símbolos propios del área del problema, porque de esta manera se reduce mucho el tiempo de aprendizaje del DSL especialmente con la técnica implementada de arrastrar y soltar objetos, resolver ejercicios de circuitos lógicos combinacionales con compuertas lógicas AND, OR y NOT se vuelve una manera más interactiva y didáctica.

Bibliografía

Prieto Torralbo, P., & Abad, P. (n.d.). Sistemas Digitales. In *Circuitos lógicos combinacionales.* Open Course Ware.

Acosta, J. (2018, Febrero 20). *Guía rápida para aprender Scrum*. Retrieved 26 May 2021 from https://openwebinars.net/blog/la-guia-para-aprender-scrum/

Arduino. (2020, 11 5). Retrieved 26 May 2021 from https://www.arduino.cc/en/guide/introduction: https://www.arduino.cc/en/guide/introduction

Arkadiusz, R. (2020). Sirius modeling tool use in electrical system.

Bettin, J. (2004). Model-driven software development. *MDA Journal*, 1-4.

Bézivin, J. (2004). In search of a basic principle for model driven engineering. *Novatica Journal, Special Issue*, 21--24.

Bilitchenko, L., Liu, A., Cheung, S., Weeding, E., Xia, B., Leguia, M., . . . Densmore , D. (2011). Eugene – A Domain Specific Language for Specifying and Constraining Synthetic Biological Parts, Devices, and Systems. *PLoS ONE*.

Botella, P., Burgués, X., Carvallo, J., Franch, X., Grau, G., Marco, J., & Quer, C. (n.d.). Measuring software product quality: A survey of ISO/IEC 9126. *IEEE*, 88-92.

Carrillo Guambo, Carlos Enrique. (2006). *Arquitectura dirigida por modelos (MDA) y su aplicación en un caso de estudio.* QUITO/EPN/2006.

Cueva Lovelle, J. M., & García Bustelo, C. P. (2008). *MDE: Ingeniería dirigida por modelos. Otra forma de construir software.* Universidad Distrital Francisco José de Caldas, Bogotá.

Culebro Juárez, M., Gómez Herrera, W. G., & Torres Sánchez, S. (2006). *Software libre vs software propietario Ventajas y Desventajas.* México: Creative Commons.

D. Ratiu and V. Pech and K. Dummann. (2017). Experiences with Teaching MPS in Industry: Towards Bringing Domain Specific Languages Closer to Practitioners. In *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)* (pp. 83-92).

Estublier, Jacky and Vega, Germán and Ionita, Anca. (2005). Composing Domain-Specific Languages for Wide-Scope Software Engineering Applications. 69-83.

F. Rosique and B. Álvarez and P. Sánchez and J. A. Pastor. (2016). U-DSL: a Domain Specific Language for Ubiquitous Systems. *IEEE Latin America Transactions*, 4416-4420.

Fowler, M. (2010). *Domain-specific languages.* Pearson Education.

Francois, F., Grégory, N., Morin, B., Daubert, E., Barais, O., Plouzeau, N., & Jezequel, J. (2012). An Eclipse Modelling Framework Alternative. *International conference*

on model driven engineering languages and systems. Springer, Berlin, Heidelberg., 87-101.

García Molina, J., García Rubio, F., Pelechano, V., Vallecillo, A., Vara, J. M., & Vicente Chicote, C. (2012). *Desarrollo de Software Dirigido por Modelos.* Madrid: RA-MA Editorial.

Greenfield, Jack and Short, Keith. (2003). Software factories: assembling applications with patterns, models, frameworks and tools. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications.* 16-27.

Gronback, R. C. (2009). *Eclipse modeling project: a domain-specific language (DSL) toolkit.* Pearson Education.

Henderson-Sellers, B and Gonzalez-Perez, C. (2006). A powertype-based metamodelling framework. *Software and Systems Modeling*, 72--90.

Jácome Guerrero, S. (2019). *Propuesta de mecanismos de personalización de meta-modelos en la Ingeniería Dirigida por Modelos.* Universidad Autónoma de Madrid, Madrid.

Jácome Guerrero, S. (2019). *Propuesta de mecanismos de personalización de meta-modelos en la Ingeniería Dirigida por Modelos.* Madrid.

Jácome Guerrero, S., Salazar Jácome, E. S., & Sánchez Ocana, W. (2018). Software development environments and tools in MDE. *KnE Engineering*, 1-15.

Jatmiko Suwawi, D., Darwiyanto, E., & Rochmani, M. (2015). Evaluation of academic website using ISO/IEC 9126. *International Conference on Information and Communication Technology (ICoICT)*, 222-227.

Kleppe, Anneke G and Warmer, Jos and Warmer, Jos B and Bast, Wim. (2003). *MDA explained: the model driven architecture: practice and promise.* Addison-Wesley Professional.

Kovse, J., & Harder, T. (2002). Generic XMI-based UML model transformations. In International Conference on Object-Oriented Information Systems. *Springer, Berlin, Heidelberg.*, 192-198.

Kushner, D. (2011). The making of arduino. *IEEE spectrum*.

L. Bambaci and F. Boschetti and R. Del Gratta. (2018). Qohelet Euporia: A Domain Specific Language to Annotate Multilingual Variant Readings. In *2018 IEEE 5th International Congress on Information Science and Technology (CiSt)* (pp. 266-269).

L. Kumar and R. Jetley and A. Sureka. (2016). Source Code Metrics for Programmable Logic Controller (PLC) Ladder Diagram (LD) Visual Programming Language. In *2016 IEEE/ACM 7th International Workshop on Emerging Trends in Software Metrics (WETSoM)* (pp. 15-21).

López Hernández, J. (n.d.). *Problemas Resueltos.* Retrieved from https://blogsaverroes.juntadeandalucia.es/sierramagina/files/2016/02/digital3.pdf

López Menéndez de Jiménez, R. E. (2015). Metodologías Ágiles de Desarrollo de Software Aplicadas a la Gestión de Proyectos Empresariales. *Escuela Especializada en Ingeniería ITCA-FEPADE*.

Mernik, Marjan and Heering, Jan and Sloane, Anthony M. (2005). When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 316--344.

Papp, Jozsef and Tokody, Daniel and Flammini, Francesco. (2018). From traditional manufacturing and automation systems to holonic intelligent systems. *Procedia Manufacturing*, 931--935.

Pavlich-Mariscal, J. A., Veliz-Quispe, H. D., Demurjian, S. A., & Michel, L. D. (215). Un ambiente de meta-modelado y visualización basado en el paradigma . *Ingeniare. Revista chilena de ingeniería*, 219-234.

Richters, M., & Gogolla, M. (2002). OCL: Syntax, semantics, and tools. In Object Modeling with the OCL. *Springer, Berlin, Heidelberg.*, 42-68.

Rube, I. R. (2012). Ingeniería Dirigida por Modelos y Calidad de Software. Cádiz.

Ruiz Gutierrez, J. M. (n.d.). *Manual de programación Arduino.* California.

S. Bhardwaj and P. Larbig and R. Khondoker and K.Bayarou. (2017). Survey of domain specific languages to build packet parsers for industrial protocols. In *2017 20th International Conference of Computer and Information Technology (ICCIT)* (pp. 1-6).

Sammet, J. E. (1972). Programming languages: history and future. *Communications of the ACM*, 601--610.

Schmidt, D. C. (2006). Model-Driven Engineering. *COMPUTER-IEEE COMPUTER SOCIETY*, 25.

Schwaber, K., & Sutherland, J. (2017). *La Guía Definitiva de Scrum: Las reglas del juego.*

Sirius. (n.d.). Retrieved 5 June 2021 from https://www.eclipse.org/sirius/doc/

Soberning, S., Strembeck, M., & Beck, A. (2019). Developing a Domain-Specific Language for Scheduling in the European Energy Sector. *International Conference on Software Language Engineering*, 19-35.

Tapia Ayala, C. H., & Manzano Yupa, H. M. (n.d.). *Evaluación de la plataforma arduino e implementación de un sistema de control de posición horizontal.* Universidad Politécnica Salesiana, Guayaquil.

Valiente, J. (n.d.). *Desarrollo y despliegue de Sotware Industrial Seguro.*

Vázquez Ingelmo, A., García Holgado, A., & García Peñalvo, F. (2020). C4 model in a Software Engineering subject to ease the comprehension of UML and the software development process. *IEEE*, 919-924.

VÖlter Markus and Stahl, Thomas and Bettin, Jorn and Haase, Arno and Helsen, Simon. (2013). *Model-driven software development: technology, engineering, management.* John Wiley \& Sons.

Anexos