UN LENGUAJE DE MODELADO ESPECÍFICO PARA AUTÓMATAS FUZZY

A specific modeling language for Fuzzy Automatons

MARÍA CASTAÑEDA LÓPEZ

MÁSTER EN INGENIERÍA INFORMÁTICA. FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo Fin Máster en Ingeniería Informática

Madrid, 6 de Septiembre de 2019

Convocatoria Septiembre 2019

Calificación: Sobresaliente - 9,5

Director/es y/o colaborador:

Mercedes García Merayo
Sonia Estevez Martín

Resumen

En la actualidad los avances médicos están ligados a los tecnológicos, pero no por ello los especialistas tienen los conocimientos necesarios para poder explotar las nuevas tecnologías. Uno de los campos en los que es necesario el uso de sistemas que traten y estudien los datos clínicos de manera precisa es la prevención de enfermedades. Los métodos formales ofrecen el rigor y la precisión necesarios para el modelado del proceso de análisis de dichos datos. En particular, los autómatas fuzzy son muy apropiados en este caso, ya que permiten representar la incertidumbre e imprecisión que se han de tener en cuenta en este tipo de análisis. En este trabajo se propone crear una herramienta gráfica que permita a los especialistas médicos, que son los que tienen el conocimiento clínico, el diseño de estos modelos de análisis de una manera sencilla e intuitiva. El objetivo principal de este trabajo es crear un entorno de diseño de modelos basados en autómatas fuzzy a través de un lenguaje de dominio específico y un editor gráfico para facilitar el diagnóstico precoz de enfermedades.

El código del proyecto puede ser descargado en https://github.com/mariacaslop/Environment-of-model-design-based-on-fuzzy-automaton

Palabras clave

Lenguaje de modelado de dominio específico, autómata fuzzy, herramientas de apoyo para el diagnóstico precoz, metamodelo, editor gráfico.

Abstract

Currently, it is undeniable that medical advances are linked to technological ones, but that this does not mean that specialists have the necessary knowledge to be able to exploit the new technologies. One of the fields in which it is necessary to use systems that process and study clinical data in a precise way is disease prevention. Formal methods can offer the rigor and precision necessary for the modeling of the process of analysis of said data. In particular, the fuzzy automatons are very suitable in this case, since they allow to represent the uncertainty and inaccuracy which must be taken into account in this type of analysis. In this paper is proposed to create a graphic tool which allows medical specialists with clinical knowledge, the design of these analysis models in a simple and intuitive way. In relation to the main purpose of this paper, it could be stated that it is to create an environment of model design based on fuzzy automaton through a specific domain language and a graphic editor to facilitate the early diagnosis of diseases.

The code of this proyect can be downloaded from https://github.com/mariacaslop/Environment-of-model-design-based-on-fuzzy-automaton

Keywords

Domain-specific modeling language, fuzzy automata, support tools for early diagnosis, metamodel, graphic editor.

Índice general

Índice I

Agradecimientos III

1. Introducción 1
- 1.1. Objetivos . . . 3
- 1.2. Plan de trabajo . . . 3
- 1.3. Estructura de la memoria . . . 4

2. Preliminares 5
- 2.1. Autómata . . . 5
  - 2.1.1. T-norms . . . 6
  - 2.1.2. Restricciones fuzzy . . . 7
  - 2.1.3. Variables y expresiones . . . 7
  - 2.1.4. El formalismo Autómata Fuzzy . . . 7
- 2.2. Ingeniería dirigida por modelos . . . 9
  - 2.2.1. Modelos, metamodelos y meta-metamodelos . . . 9
  - 2.2.2. Arquitectura de metamodelado . . . 10
  - 2.2.3. Lenguajes de modelado . . . 14

3. Entorno de desarrollo 15
- 3.1. Eclipse Modeling Framework . . . 15
- 3.2. Graphical Modeling Framework . . . 19
- 3.3. Eclipse Epsilon . . . 21

4. Diseño, implementación y evaluación 26
- 4.1. Marco de diseño . . . 26
- 4.2. Transformación Modelo Texto . . . 45
- 4.3. Producto Final . . . 47
- 4.4. Caso de Estudio . . . 51

5. Conclusiones 55

6. Introduction 57
- 6.1. Objectives . . . 59
- 6.2. Workplan . . . 59
- 6.3. Report Structure . . . 60

7. Conclusions 61

Bibliography 69

Agradecimientos

Me gustaría agradecer a mis tutoras, por darme la oportunidad de trabajar con ellas, por su ayuda y disponibilidad en todo momento. Gracias a ellas ha sido mucho más agradable la realización de este trabajo.

A mi familia, por apoyarme una vez más y darme esta oportunidad para poder seguir formándome.

Y a Pablo, por aguantar mis charlas interminables, ayudarme y apoyarme siempre.

Capítulo 1

Introducción

Los métodos formales permiten el modelado riguroso de sistemas, lo que proporciona un alto grado de fiabilidad en el desarrollo de los mismos. Una de las áreas en las que se requiere un alto grado de rigor y seguridad en los sistemas que se utilizan, los protocolos que se aplican y el análisis de la información es la medicina. Para ello sería deseable la aplicación de metodologías basadas en formalismos con base matemática que permitan proporcionar una semántica precisa.

En la literatura podemos encontrar trabajos que proponen metodologías formales en el ámbito de dispositivos y sistemas médicos. La mayoría de las propuestas se aplican a marcapasos[13,25,26,31,33,34,39–41,50], bombas de infusión[2,4,44,58] y sistemas de hemodiálisis[3,24,24,30]. Estas propuestas abordan diferentes fases del proceso de desarrollo, destacando el modelado de los sistemas[2–4,24–26,30–32,34,39–41,44,45,50,58,60], para la posterior aplicación, en la mayoría de los casos, de técnicas de verificación[2,3,13,24,31,33,34,39,44,50,58] y validación[3,30,34] de los mismos. En menor medida se aplica también a la generación de código[25,33] y testing de software[4,31]. Los formalismos más utilizados en estas propuestas son, principalmente, los *Timed Automata*[13,31,33,34,50] y otros basados en estados como *Extended Finite State Machines*[2,4], *Abstract State Machines*[3], *Z*[25,26] o *Algebraic State Transition Diagrams*[15].

Otro aspecto de la medicina en el que también se han aplicado los métodos formales es el diseño de los protocolos médicos[28,35,43,61], que facilitan y permiten entender mejor a los profesionales el proceso de atención a los clientes mejorando la seguridad de estos.

Uno de los campos en los que el desarrollo formal de sistemas empieza a cobrar importancia en el ámbito de la salud es la prevención de enfermedades. La mejora en los procesos de diagnóstico precoz requiere de sistemas que procesen y analicen de forma precisa los datos de los pacientes, incluso en estadios en los que aún son asintomáticas. En esta línea se ha propuesto recientemente un marco de análisis de información[9] que considera dos aspectos relevantes en el análisis de los datos clínicos: imprecisión e incertidumbre. En este trabajo, los autores presentan una versión *fuzzy* de los autómatas que captura estas características, permitiendo establecer ciertos niveles de tolerancia a la hora de analizar los datos, así como definir las restricciones que deben tenerse en cuenta a la hora de determinar el grado de confianza de las decisiones que se toman durante el proceso de diagnóstico. Este formalismo también facilita la representación de la correlación que debe existir entre los valores de los parámetros de interés para la detección de la patología considerada. Con ello se incorpora flexibilidad en el proceso, evitando análisis estáticos que pueden llevar a diagnósticos erróneos. Todos estos aspectos hacen que este formalismo sea muy adecuado para su uso en el desarrollo de sistemas de apoyo al diagnóstico precoz de enfermedades, basados en el conocimiento de los expertos y en el análisis de los datos obtenidos en pruebas médicas.

Sin embargo, es necesario recordar que estos sistemas se apoyan en modelos que deberían ser diseñados por especialistas médicos, que tienen el conocimiento diagnóstico requerido, pero que no tienen los conocimientos técnicos necesarios. Las herramientas utilizadas en los marcos de modelado formales previamente comentados requieren habilidades técnicas que los expertos en salud no han adquirido. Esta limitación puede ver reducido el uso de estas herramientas. Por ello, consideramos útil proporcionar a los expertos una herramienta gráfica que les permita abstraerse de complejidades técnicas y diseñar modelos de análisis de forma sencilla. Con este objetivo se ha llevado a cabo este proyecto, en el que se ha creado un marco de diseño de modelos basados en autómatas fuzzy mediante un lenguaje de modelado de dominio específico y un editor gráfico. Asimismo, se ha desarrollado un proceso de transformación de los modelos diseñados con dicho editor para que estos modelos sean utilizados

por la herramienta AUNTY (AUtomatically aNalyze daTa using fuzzY automata)[8] para el análisis de los datos correspondientes. Para la evaluación de este marco se ha considerado un caso de estudio relativo al diagnostico de enfermedad celiaca en pacientes pediátricos.

1.1. Objetivos

El principal objetivo de este trabajo, como ya se ha mencionado, es desarrollar un entorno de diseño de modelos basados en el formalismo de autómata fuzzy para el apoyo al diagnóstico de enfermedades. Este objetivo se puede desplegar en los siguientes subobjetivos.

- Diseño de un lenguaje de modelado de dominio específico para diseño de modelos basados en autómatas fuzzy: (a) definición de la sintaxis abstracta mediante un metamodelo y las reglas de validación necesarias y (b) definición de la sintaxis concreta asociada, en este caso gráfica.
- Creación de un editor gráfico que permita el diseño gráfico y validación automática de modelos conformes al lenguaje previamente definido.
- Diseño e implementación de un proceso de transformación de los modelos para su importación y explotación por la herramienta AUNTY.

1.2. Plan de trabajo

Para lograr estos objetivos, el trabajo se ha desarrollado en las siguientes fases:

- Revisión de trabajos propuestos para el uso de métodos formales en el área de apoyo al diagnóstico de enfermedades, así como de disponibilidad de herramientas de apoyo para su diseño. Selección del formalismo más adecuado para el modelado.
- Diseño del lenguaje de modelado de dominio específico gráfico para el formalismo seleccionado. Esta fase ha requerido la decisión del entorno de desarrollo, herramientas y lenguajes para llevar a cabo estas tareas.

- Diseño del editor gráfico.
- Implementación del proceso de transformación de modelos.
- Evaluación del marco de diseño mediante un caso de estudio que permita establecer la aplicabilidad del mismo.

1.3. Estructura de la memoria

En este documento podemos distinguir una parte en la que se describen los fundamentos teóricos del trabajo, y otra en la que se describe el diseño y desarrollo del proyecto.

En el Capítulo 2 se encuentran los preliminares, donde se describen los principales conceptos utilizados en este trabajo. En el Capítulo 3 se explican el entorno de trabajo, así como las herramientas y los lenguajes que se han utilizado para la implementación de este proyecto. En el Capítulo 4 se encuentra la información detallada relativa al diseño, desarrollo e implementación del marco de modelado. Además, se presenta el caso de estudio llevado a cabo para la evaluación del mismo. Por último, en el Capítulo 5, se presentan las conclusiones y posibles trabajos futuros.

Capítulo 2

Preliminares

En este capitulo se presentan los principales conceptos en los que se fundamenta este trabajo. En primer lugar, se introduce la noción de *autómata fuzzy* y se describen sus componentes. A continuación, se explican los principales aspectos asociados al metamodelado.

2.1. Autómata

Los autómatas finitos, así como diferentes variantes de los mismos, han sido utilizados para la representación formal del comportamiento de sistemas. En este trabajo nos centramos en el modelado de procesos para el análisis de información que requiere considerar cierto grado de *imprecisión* e *incertidumbre*. Por ello hemos decidido basarnos en un formalismo recientemente propuesto[7], *autómata fuzzy*, que permite tener en cuenta estas características mediante el uso de lógica difusa. Intuitivamente, la lógica difusa posibilita la definición de relaciones que proporciona una medida en una escala entre 0 y 1 del nivel de confianza en el análisis de la información. A continuación se presentan algunos conceptos básicos necesarios para la definición de los *autómata fuzzy*.

El uso de las relaciones fuzzy permite clasificar valores respecto a su comportamiento esperado. A diferencia de las relaciones clásicas cuya evaluación devuelve 0 (*true*) o 1 (*false*), las *relaciones fuzzy* proporcionan un valor en el intervalo $[0, 1]$. El valor 1 indica que la relación se cumple con confianza 1, mientras que valores menores que 1 reflejan un nivel de confianza menor. En particular, si la relación no se cumple el valor será 0. En este trabajo se

consideran relaciones sobre el conjunto de los números reales, parametrizadas por un valor, denotado por δ, que indica la máxima desviación permitida respecto al valor esperado. Por ejemplo, en el caso de que se evalúe si un valor a cumple $\alpha \leq a \leq \beta$ y dicho valor se encuentra en dicho intervalo, la relación se cumple con confianza 1. Sin embargo, si no se encuentra en este intervalo y la distancia de a a α o β es inferior a δ se obtendrá una confianza positiva, que disminuye a medida que esta distancia aumenta. Si la desviación es superior a δ el nivel de confianza será 0. A continuación se presentan las relaciones fuzzy que se han considerado en este trabajo.

$$\overline{x \leq y}^{\delta} \equiv \begin{cases} 1 & \text{if} \quad x < y \\ \frac{\delta+y-x}{\delta} & \text{if} \quad y \leq x \leq y+\delta \\ 0 & \text{if} \quad y+\delta < x \end{cases}$$

$$\overline{x \geq y}^{\delta} \equiv \begin{cases} 1 & \text{if} \quad x > y \\ \frac{\delta+x-y}{\delta} & \text{if} \quad y \geq x \geq y-\delta \\ 0 & \text{if} \quad y-\delta > x \end{cases}$$

$$\overline{x = y}^{\delta} \equiv \begin{cases} 0 & \text{if} \quad x \leq y-\delta \\ \frac{x-y+\delta}{\delta} & \text{if} \quad y-\delta < x \leq y \\ \frac{-x+y+\delta}{\delta} & \text{if} \quad y < x \leq y+\delta \\ 0 & \text{if} \quad y+\delta < x \end{cases}$$

$$\overline{x \leq y \leq z}^{\delta} \equiv \begin{cases} 0 & \text{if} \quad z < x \\ 0 & \text{if} \quad y \leq x-\delta \\ \frac{y-x+\delta}{\delta} & \text{if} \quad x-\delta \leq y \leq x \leq z \\ 1 & \text{if} \quad x < y < z \\ \frac{-y+z+\delta}{\delta} & \text{if} \quad x \leq z \leq y \leq z+\delta \\ 0 & \text{if} \quad z+\delta < y \end{cases}$$

2.1.1. T-norms

La combinación de los valores de confianza obtenidos de la evaluación de las relaciones fuzzy se realiza mediante normas triangulares ($t-norms$). Una $t-norm$ es una operación binaria que permite generalizar la conjunción de la lógica proposicional. Se pueden encontrar diferentes $t-norms$ en la literatura, cuya aplicación dependerá de la naturaleza de los datos analizados, como por ejemplo, Gödel, Hamacher, Lukasiewicz o Frank. [5,55]

2.1.2. Restricciones fuzzy

Una *restricción fuzzy* es una formula en la que se combinan *relaciones fuzzy* mediante $t-norms$ y que pueden incluir variables libres. En los autómatas las restricciones se utilizan para decidir si se permanece en un estado o si una transición puede ser ejecutada. Esto corresponde al valor obtenido de la evaluación de la restricción, es decir, 0 o 1. En el caso de los autómatas fuzzy, está evaluación corresponderá a un *grado de satisfacción* de la restricción en el intervalo $[0, 1]$.

2.1.3. Variables y expresiones

La generación de resultados a partir de los datos observados por los autómatas fuzzy incluirán un conjunto de expresiones que tomarán valores reales durante la ejecución de las transiciones etiquetadas con acciones de entrada. Además, las transiciones podrán tener asociada una *transformación de variables*.

2.1.4. El formalismo Autómata Fuzzy

A continuación se define formalmente un autómata fuzzy, formalismo que nos permitirá representar los procesos de análisis de datos en los que estamos interesados.

Un *automata fuzzy* es una tupla $(S, Acts, X, X_0, s_0, T)$ cuyos componentes corresponden a:

- S: conjunto finito de estados
- $Acts$: conjunto de acciones, que se dividen en *inputs* I y *outputs* O
- X: conjunto de variables
- $X_0 : X \longrightarrow \mathbb{R}_+$ asigna inicialmente valores a las variables
- $s_0 \in S$: el estado inicial.
- $T \subseteq S \times Acts \times \mathcal{RF} \times \mathcal{TV} \times S$: conjunto de transiciones

donde $\mathcal{RF}$ representa el conjunto de todas las restricciones fuzzy y $\mathcal{TV}$ el conjunto de transformaciones de variables.

Intuitivamente, un autómata fuzzy es un grafo dirigido cuyas transiciones están etiquetadas con restricciones fuzzy y transformaciones de variables. Se consideran dos tipos de acciones: *inputs* y *outputs*. Los inputs pueden tener asociada una tupla de variables que actúan como parámetros de entrada, mientras los outputs pueden tener asociadas expresiones de salida. Consideremos una transición (s, a, fc, vt, s'). Si el sistema se encuentra en el estado s y recibe un input a, la restricción fuzzy asociada fc es evaluada para los valores actuales de las variables del autómata. Si el grado de confianza obtenido es superior a 0 la transición puede ser ejecutada, las variables son actualizadas mediante la transformación de variables vt asociadas a la misma y el autómata pasa al estado s'. Si la transición está etiquetada con un output, esta será ejecutada si el grado de confianza de la restricción asociada es superior a 0. En la figura 2.1 se presenta un ejemplo de autómata fuzzy.

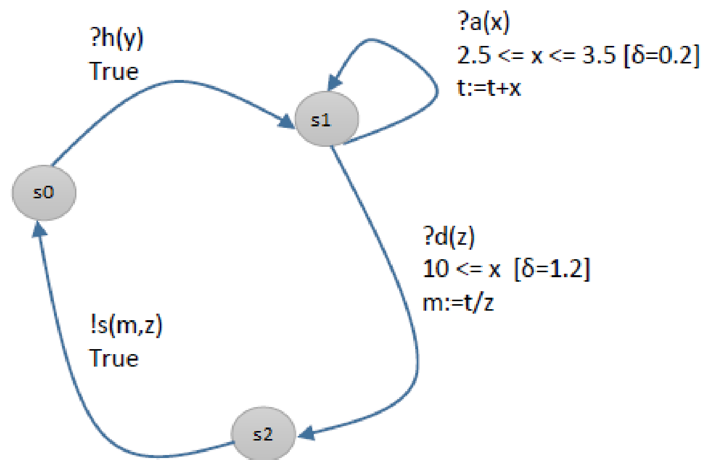

Figura 2.1: *Ejemplo autómata fuzzy*

2.2. Ingeniería dirigida por modelos

La ingeniería dirigida por modelos (en inglés Model-Driven Engineering - MDE)[6] es una disciplina dentro de la ingeniería del software, que se ocupa de la utilización de *modelos* para mejorar la productividad, la calidad y el mantenimiento del software, entre otros aspectos, especialmente cuando presta una elevada complejidad, mediante el aumento del nivel de abstracción y automatización. La ingeniería basada en modelos se puede aplicar en distintos ámbitos, desde el desarrollo de nuevas aplicaciones hasta la reingeniería basada en modelos[27,51,54].

En este contexto, un *sistema* se define como "*concepto genérico para el diseño de una aplicación, plataforma, o artefacto software*"[12]. Un sistema puede estar compuesto o estar relacionado con otros subsistemas.

Un modelo es una representación simplificada de un sistema, una abstracción, que permite entenderlo mejor. Esta representación puede ser textual o gráfica y se crea mediante un lenguaje de modelado para un dominio específico (en inglés Domain-Specific Modeling Languages - DSMLs). En el proceso de diseño se distinguen tres componentes. En primer lugar, una *sintaxis abstracta* compuesta de un metamodelo y un conjunto de reglas de validación que permiten determinar si un modelo está bien definido. En segundo lugar, una *sintaxis concreta* que incluye información sobre como representar (visual o textualmente) los conceptos de la sintaxis abstracta. Finalmente, transformaciones entre modelos que permitan la automatización del desarrollo de software, el rediseño de los mismos, su simulación, etc.

A continuación se revisan los principales conceptos y definiciones subyacentes en el ámbito de MDE.

2.2.1. Modelos, metamodelos y meta-metamodelos

Existen numerosas definiciones de *modelo*,[56 38 57] aunque existe consenso en que un modelo es una representación simplificada de un sistema. Existen algunas propiedades que

caracterizan a los modelos[42]. Un modelo debe permitir identificar los elementos de la realidad que representa. Además, un modelo debe ser útil, es decir, debe servir para un propósito final. Por último, debe ser una versión simplificada de esa realidad, en la que no tienen que estar representados todos los aspectos de la misma, tan solo aquellos relevantes para el propósito del modelo.

Un modelo se representa de acuerdo a un lenguaje de modelado, cuya sintaxis abstracta está definida por un *metamodelo*. Intuitivamente, un metamodelo es una representación de la clase de todos los modelos que pueden expresarse con dicho lenguaje, *un modelo de modelos*. El lenguaje de modelado de los metamodelos es un *meta-metamodelo*. En base a ello, se pueden definir modelos que describen metamodelos y otros modelos que describen meta-metamodelos. Podrían existir de este modo infinitos niveles de abstracción de metamodelado, pero se ha visto que los meta-metamodelos pueden ser definidos en base a si mismos.

2.2.2. Arquitectura de metamodelado

La posibilidad de que estos lenguajes puedan ser definidos respecto a si mismos permite disponer de una arquitectura finita. Por tanto, no se necesitan lenguajes que permitan definir meta-metamodelos.

La figura 2.2 presenta la arquitectura de cuatro niveles (M0,M1,M2,M3)[14] que se utiliza para describir la jerarquía del metamodelado. En esta jerarquía se asume que un modelo del nivel Mi *es conforme* al modelo del nivel M(i+1) al cual instancia. En el caso del nivel M3, los meta-metamodelos son conformes a si mismos. En las siguientes líneas se describen con más detalle estos niveles y los conceptos principales que se capturan en cada uno de ellos.

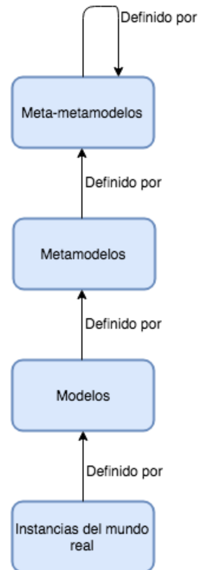

Figura 2.2: *Arquitectura del metamodelado*[6]

Nivel M3

El nivel M3 corresponde al *meta-metamodelado*. Esta capa es la responsable de proporcionar un lenguaje para especificar metamodelos, es decir, lenguajes de modelado. Los meta-metamodelos se definen en base a si mismos, es decir, son conformes a ellos mismos.

El lenguaje de metamodelado standard propuesto por la *Object Mangement Group* (OMG) es el *Meta Object Facility (MOF)*[48]. Además de este standard se han propuesto otros marcos de metamodelado. Entre ellos cabe destacar el *Eclipse Modeling Framework (EMF)*[59] que dispone de su propio lenguaje de metamodelado *Ecore*.

Nivel M2

En este nivel se encuentran los *metamodelos* definidos conforme a un meta-metamodelo. Los metamodelos básicamente constituyen la definición de un lenguaje de modelado, es decir, los conceptos del lenguaje y las relaciones entre los mismos, así como las reglas que permitan determinar si un modelo está bien formado. Un metamodelo puede verse como una descripción del conjunto de todos los modelos que pueden ser representados por ese lenguaje[6]. Tanto *Unified Modeling Language* (UML)[49] como *Common Warehouse Metamo-*

del(CWM)[47] son ejemplos de metamodelos conformes al meta-metamodelo MOF.

Nivel M1

El nivel M1 corresponde a los *modelos* definidos conforme a un metamodelo. Los modelos definen conceptos del dominio de interés en base a conceptos del lenguaje definido mediante un metamodelo.

Nivel M0

En este nivel se encuentran las instancias de los modelos que corresponden a entidades reales del dominio, como pueden ser conceptos, procesos, etc.

La Figura 2.3 muestra un fragmento de un metamodelo correspondiente al lenguaje Hypertext Markup Language (HTML), en concreto la parte de la cabecera. La clase principal es HTML que está compuesto de *Body* y *Head*. Ambos heredan de la clase *HTMLElement* la cual contiene un atributo *value* de tipo *String*. La clase *Head*, pueden tener varios atributos de tipo *HeadElement*. Estos pueden ser de tipo *Link* o de tipo *Tittle*.

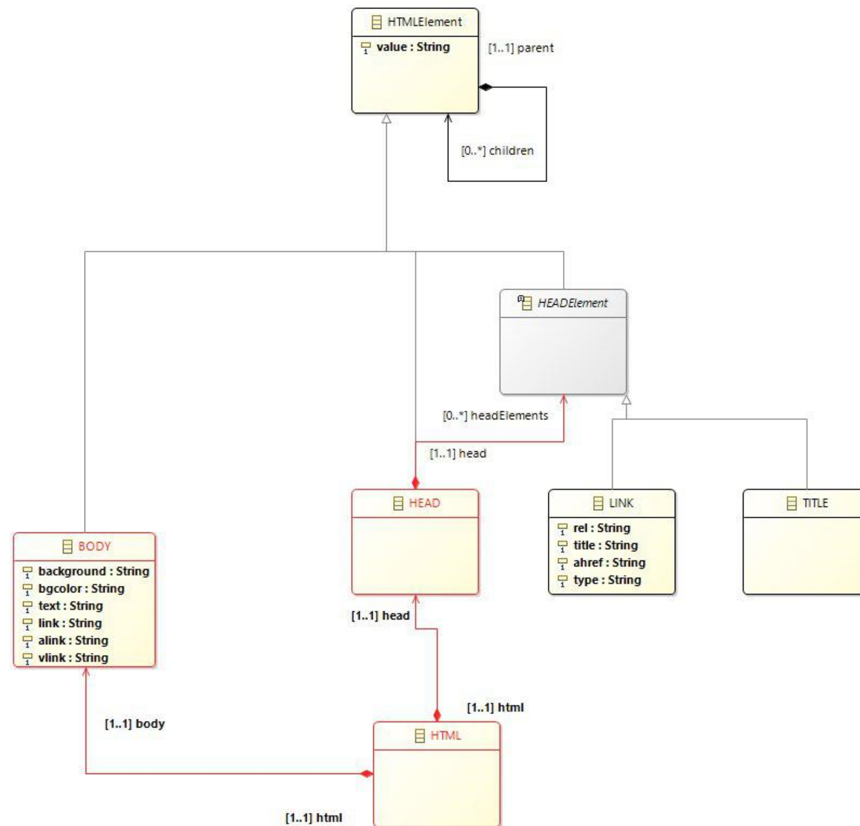

Figura 2.3: *Metamodelo HTML*

Con este metamodelo se pueden definir modelos para crear cabeceras en HTML. A continuación, se presenta un modelo conforme a este metamodelo, representado mediante su sintaxis concreta.

```
<!DOCTYPE html>
<html lang="es-ES">
  <head>
    <title>Ejemplo con 2 cabeceras</title>
  </head>
  <body>
    <h1>Esto es una cabecera h1</h1>
    <p>Esto es un parrafo.</p>
    <h2>Esto es una cabecera h2</h2>
  </body>
</html>
```

En el nivel M0 dse encontraría una página web como la que aparece en la Figura 2.4.

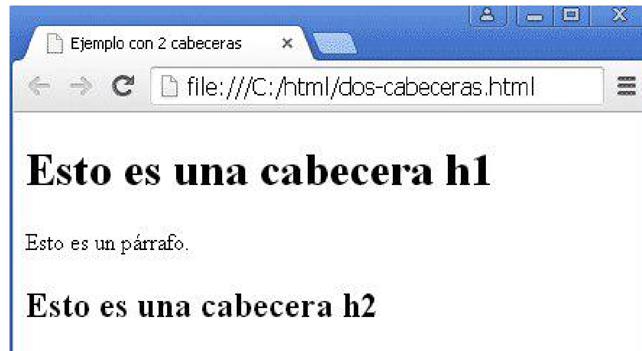

Figura 2.4: *Ejemplo cabecera html*

2.2.3. Lenguajes de modelado

Los modelos deben ser conformes al metamodelo asociado que, como ya se ha mencionado, representa la sintaxis abstracta del lenguaje de modelado. Esta sintaxis describe la estructura del lenguaje, independientemente de una notación específica. La *sintaxis concreta* asociada al lenguaje proporcionará una representación del lenguaje de modelado que permitirá las tareas de diseño de modelos conformes al metamodelo. Esta sintaxis podrá ser *gráfica*, como la asociada a UML, o *textual*, como Emfatic[16] que define Ecore.

Una sintaxis gráfica presentará símbolos gráficos, figuras, etiquetas y reglas de composición de dichos elementos gráficos. Además, se deberá establecer la correspondencia de los símbolos gráficos y los elementos de la sintaxis abstracta. Las sintaxis textuales incluirán elementos para representar la información recogida en los modelos, así como *keywords* para representar elementos de los mismos, caracteres limitadores, etc...

Los lenguajes de modelado pueden ser tanto lenguajes de propósito general como lenguajes de modelado de dominio específico (en inglés Domain Specific Modelling Languages - DSML). Estos últimos son lenguajes de modelado diseñados especificamente para un cierto dominio que incluyen sus primitivas principales y abstracciones[62]. Como ejemplos, BPMN[1] para el modelado de procesos de negocio y WebML[10] para el modelado de aplicaciones web.

Capítulo 3

Entorno de desarrollo

En este capítulo se introducen las tecnologías, entornos y herramientas que se han utilizado para desarrollar el proyecto.

El entorno de desarrollo utilizado forma parte del conjunto de herramientas que componen *Eclipse*, una plataforma de software que permite integrar diferentes aplicaciones en forma de *plug-ins* para construir un *Integrated Develoment Environment* (IDE).

3.1. Eclipse Modeling Framework

Como se ha comentado en el Capítulo 3, existen varios marcos de metamodelado, entre los cuales se encuentra *Eclipse Modeling Framework* (EMF).

EMF es el núcleo de la plataforma Eclipse para el desarrollo dirigido por modelos. Se trata de un framework de modelado y un sistema de generación de código que permite construir herramientas y aplicaciones basadas en un modelo. A partir de una especificación del modelo, descrito en *XML Metadata Interchange* (XMI), genera un conjunto de clases Java que permiten la visualización y la edición del mismo. Estos modelos se pueden especificar mediante anotación Java, documentos XML o herramientas de modelado, para después poder ser importados a EMF. En EMF se pueden distinguir tres componentes principales:

- *Ecore*: el lenguaje de metamodelado.
- *EMF.Edit*: marco que proporciona clases genéricas reutilizables para construir editores

para modelos EMF.

- *EMF.Codegen*: soporte de generación de código, capaz de construir un editor completo para un modelo EMF.

Ecore es una versión simplificada de *Meta-Object Facility* (MOF) que permite representar los metamodelos en ficheros XML con extensión *.ecore* y almacena en un fichero *.ecorediag* el diagrama gráfico del metamodelo.

Como se puede ver en la Figura 3.1, Ecore cuenta, entre otros, con los siguientes componentes[18]:

- *EClass*: corresponde a las instancias del metamodelo; estas instancias pueden extender otras instancias *EClass*. Además, contienen *EAttributes* y *EReferences*.
- *EAttribute*: representa las propiedades de las instancias.
- *EReference*: se encarga de representar las relaciones binarias entre dos instancias *EClass*.
- *EPackage*: es el componente que permite agrupar los elementos del metamodelo.
- *EDataType*: define el tipo de un atributo.

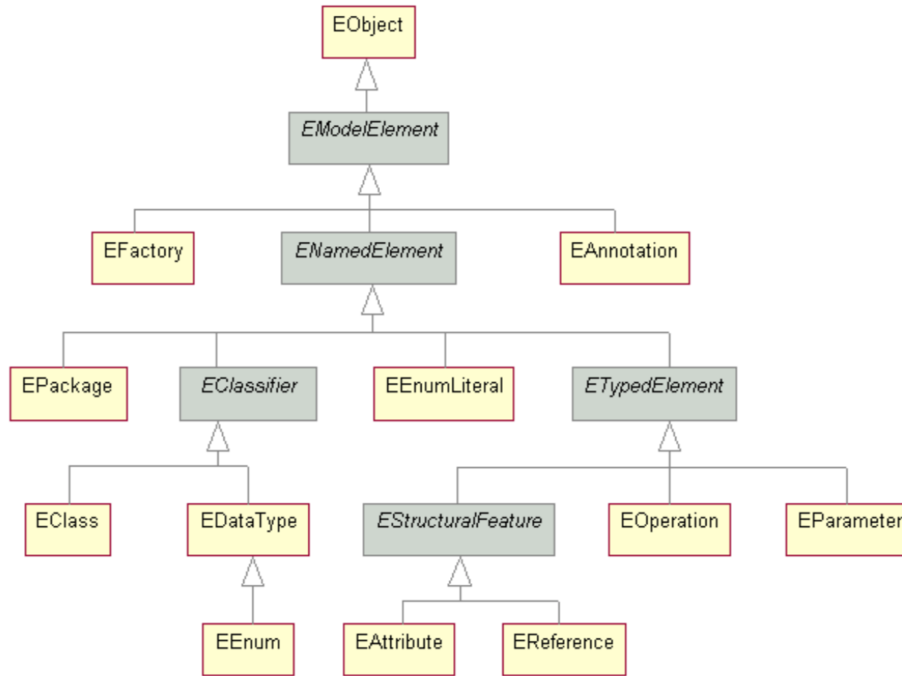

Figura 3.1: *Componentes de Ecore*[53]

Ecore cuenta con una herramienta denominada *Ecore Diagram*, la cual permite diseñar el metamodelo de dos formas distintas. En primer lugar, mediante una ventana de edición y una paleta formada por los componentes que pueden ser utilizados. Esta herramienta ofrece diferentes objetos y relaciones, como se puede ver en la Figura 3.2. Este editor permite crear clases y atributos, así como crear diferentes tipos de relaciones, como relaciones de asociación (*Association*), de agregación (*Aggregation*) y de herencia (*Generalization*). También permite incluir anotaciones mediante el uso de los objetos *EAnnotation*, *Annotation details* y *EAnnotation reference*. Algunos objetos dependen de otros, por lo que no se podrán crear si no se ha añadido previamente el objeto del que dependen. Por ejemplo, para añadir relaciones es necesario seleccionar la clase origen y la clase destino, por lo que dichas clases deben existir en el modelo previamente. Estas relaciones son unidireccionales, las relaciones bidireccionales se representan añadiendo dos relaciones, cada una en un sentido. Estas relaciones cuentan con un tipo de propiedad para fusionar ambas relaciones en una sola,

Eopposite.

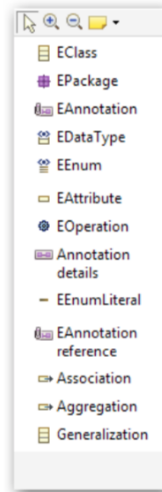

Figura 3.2: *Paleta del editor Ecore Diagram*[17]

Algunos objetos tienen asociadas una serie de propiedades editables, como los nombres de los atributos, los tipos de datos o el valor por defecto. En el caso de las relaciones, hay propiedades de definición obligatoria, como la cardinalidad mínima y máxima.

La segunda opción para diseñar el metamodelo es mediante una vista en árbol, como se muestra en la Figura 3.3. A partir del paquete principal se pueden crear otros paquetes y sus clases, los atributos de estas o las relaciones entre ellas.

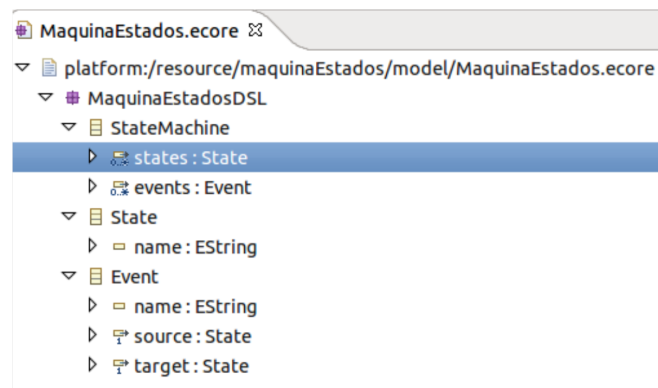

Figura 3.3: *Vista en árbol Ecore Diagram*[53]

Ecore dispone de una sintaxis textual llamada *Emfatic*, muy parecida a Java para la

definición de metamodelos. Además, permite generar código a partir de metamodelos Ecore ya existentes. El metamodelo Ecore se actualizará automáticamente cada vez que el archivo Emfatic sea modificado. En el Capítulo 5 se detalla la sintaxis empleada en este proyecto.

Una vez creado el diagrama, Ecore genera un fichero XML, con extensión *.ecore*, encargado de asociar el diagrama al modelo y generar el código.

EMF permite generar automáticamente cuatro proyectos para el modelo que se ha creado. Estos proyectos son[52]:

- *Model*: almacena las entidades, paquetes y clases utilizadas para crear objetos instanciados del modelo. Corresponde a la implementación en Java de los elementos del modelo.
- *Edit*: incluye las clases que se necesitan para visualizar y editar los modelos.
- *Editor*: proporciona los elementos necesarios para la interfaz del editor de modelos.
- *Test*: permite crear casos de prueba para el modelo.

3.2. Graphical Modeling Framework

Graphical Modeling Framework (GMF) es el marco de modelado gráfico de Eclipse, utilizado para el desarrollo de editores gráficos basados en EMF. GMF permite modelar de manera independiente los elementos del dominio y definir la paleta de herramientas. Esta forma de modelar de manera independiente hace que se puedan reutilizar los elementos gráficos que ya han sido creados para distintos dominios y aplicaciones. Además, GMF proporciona una definición de *mapping* que sirve para asociar cada entidad del modelo a su componente gráfica y a su herramienta.

GMF establece un proceso para construir herramientas gráficas de modelado que consta de los siguientes pasos: definición del metamodelo o establecimiento de uno ya creado, generación del código, creación de los elementos gráficos, definición de las herramientas del

modelo, especificación de la relación entre los elementos del modelo y los elementos gráficos y, por último, la generación de la herramienta.

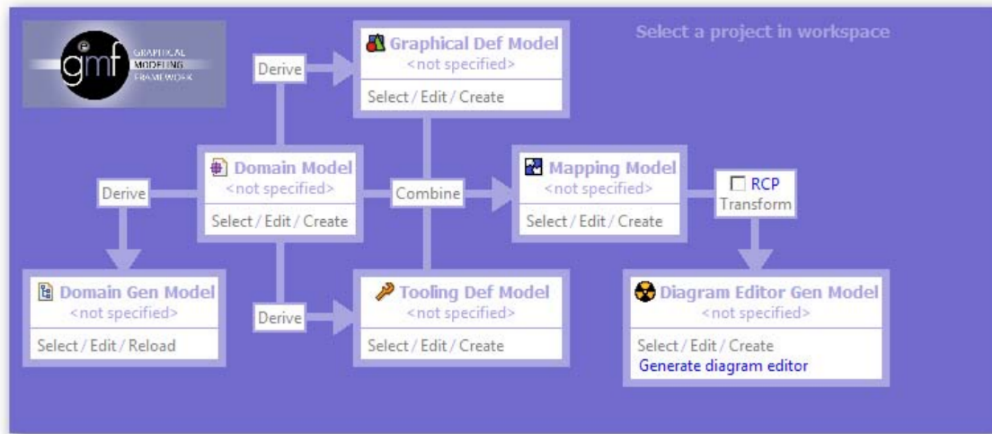

Figura 3.4: *Flujo del proceso GMF*

En Figura 3.4 podemos ver el flujo del proceso para construir una herramienta de modelado.[46]

El origen del proceso corresponde al modelado del dominio o definición del metamodelo, del que parten el resto de procesos. Específicamente se debe proporcionar el archivo *.ecore* previamente definido. En la Figura 3.4 está representado como *Domain Model.*

El segundo paso consiste en la definición de los elementos gráficos, es decir, el tipo que corresponde a cada entidad del modelo (enlaces, nodos, etiquetas...). En la Figura 3.4 corresponde a *Graphical Def Model*,

El siguiente paso consiste en la definición de las herramientas, que la Figura 3.4 representa como *Tooling Def Model.* En este paso el usuario deberá definir la paleta de herramientas, además de crear los iconos que aparecerán en la paleta para representar cada uno de los elementos del modelado.

Antes de comprobar que todos los pasos se han realizado correctamente y que la herramienta se genera correctamente, se debe definir la relación entre los elementos del modelo, los elementos gráficos y las herramientas, *Mapping Model* en la Figura 3.4.

Por último, cuando ya se han completado todos los pasos anteriores, se debe generar el archivo *.gmfgen* que en la Figura 3.4 corresponde al *Diagram Editor Gen Model*, dando lugar al código generado.

El proceso de creación de herramientas del dominio se debe realizar en el orden descrito anteriormente, en otro caso el proceso perderá coherencia y no se creará la herramienta correctamente. Todos los pasos anteriores se pueden generar de manera automática, pero es necesario revisarlo y modificar cada paso para adaptarlo a la herramienta que se quiere crear.

3.3. Eclipse Epsilon

Este proceso puede ser simplificado mediante el uso de *Eclipse Epsilon* que facilita, en parte, el trabajo realizado por EMF y GMF.

Eclipse Epsilon está formado por una familia de lenguajes y herramientas que sirven para generar código, validar modelos o transformarlos.[21,37]. Entre los lenguajes integrados en Eclipse Epsilon se encuentran: *Epsilon Object Language* utilizado para crear, consultar o modificar modelos EMF; *Epsilon Transformation Language*, lenguaje empleado para la transformación de modelo a modelo, basado en reglas y creado sobre EOL; *Epsilon Validation Language*, utilizado para las validaciones, donde las restricciones son muy similares a las de OCL.

Además, Eclipse Epsilon también cuenta con una serie de herramientas entre las que se encuentra *Eugenia*, que genera automáticamente una serie de modelos necesarios para implementar un editor GMF desde un metamodelo Ecore. Eclipse Epsilon utiliza otras herramientas como *EUnit*, un marco de test unitarios o *Exeed*, el cual permite personalizar los iconos y etiquetas de los elementos del modelo de una forma sencilla.

Eugenia

Eugenia es una de las herramientas con las que cuenta Eclipse Epsilon y que sirve para simplificar el desarrollo de editores GMF, ya que genera de manera automática los modelos intermedios: *gmfgrafph*, *gmftool* y *gmfmap*.[22]

Eugenia también proporciona anotaciones para los ficheros *.emf*. Cada tipo de anotación tiene a su vez una serie de propiedades que permiten detallar algunas propiedades. Para poder utilizar estas anotaciones se debe incluir la anotación *@gmf* en el paquete superior para así poder validar correctamente el fichero Ecore.

Las anotaciones que proporciona Eugenia son las siguientes[23,36]:

- *@gmf.diagram*: esta anotación se debe situar en la raíz del metamodelo, es decir en la *EClass* principal. Permite añadir detalles como *diagram.extension*, para indicar la extensión del archivo del diagrama.

- *@gmd.node*: se asocia a los componentes de tipo *EClass* y se utiliza para indicar que son de tipo nodo. Esta anotación permite personalizar características de la apariencia en el diagrama, como por ejemplo: *border.color*, define el color del borde del nodo; *color*, indica el color del fondo del nodo; *tamaño*, tamaño del nodo; *tool.small.path*, ruta del icono, que aparecerá tanto en la paleta de creación como en el diagrama.

- *@gmf.link*: se utiliza para indicar los componentes de tipo enlace. Además, cuenta con una serie de detalles que se pueden especificar para definir, entre otros: *width* indica el grosor del enlace, *color*, color del enlace, *label*, texto que se muestra como la etiqueta del enlace y *source.decoration* que define el estilo de las puntas de las flechas de los enlaces.

- *@gmf.compartment*: esta anotación permite señalar los nodos que están incluidos dentro de otros nodos. Un ejemplo sencillo son los atributos de una clase, estos se encuentran dentro de una clase específica.

- *@gmf.affixed*: se utiliza para crear nodos que se sitúan anexionados al borde de un nodo que contiene. Aparecerán uno junto a otro en el diagrama.

- *@gmf.label*: se emplea para añadir etiquetas adicionales a un nodo. Los nodos normalmente tienen una etiqueta, las etiquetas adicionales aparecerán en el diagrama justo debajo de la etiqueta propia del nodo.

Epsilon Object Language

Epsilon Object Language (EOL), es un lenguaje de programación utilizado para crear, consultar y modificar modelos EMF. Se puede considerar como una mezcla de Javascript [11] y OCL [20]. Es muy similar a OCL, pero mejorado, gracias a la mezcla con Javascript.

EOL puede acceder o modificar a la vez muchos modelos de distintos metamodelos. También permite llamar a métodos de objetos Java, lo que es muy útil para realizar funciones más complejas. Está característica ha sido utilizada en el proyecto y en el Capítulo 5 se detalla su funcionamiento.

Los programas se organizan en módulos, cada uno de los cuales está formado por un cuerpo y unas operaciones. El cuerpo es un bloque de sentencias que se evalúan cuando se ejecuta el módulo. Las operaciones definen el tipo de objetos sobre los que son aplicables, un nombre, un conjunto de parámetros y un tipo para el valor de retorno, aunque esto último es opcional. Los módulos pueden importar otros módulos y así acceder a sus operaciones.

EOL, cuenta con cuatro tipos primitivos (*String, Integer, Boolean y Real)* con operaciones propias para cada uno de ellos. También proporciona cuatro tipos de colecciones y un tipo de mapa. Los tipos de colecciones son: el tipo *Bag* (colecciones no únicas y no ordenadas), el tipo *Sequence* (colecciones no únicas y ordenadas), el tipo *Set* (colecciones únicas y no ordenadas) y por último el tipo *OrderedSet* (colecciones únicas y ordenadas).

Epsilon Validation Language

Epsilon Validation Language (EVL), es un lenguaje de validación construido sobre EOL. Las restricciones EVL son muy parecidas a las restricciones OCL pero EVL tiene una serie de

características que lo mejoran. Por ejemplo, EVL permite que existan dependencias entre restricciones, mensajes de error personalizados para mostrar al usuario, se pueden crear correcciones para reparar cuando no se cumple una restricción, evalúa restricciones entre modelos y durante la validación distingue entre errores y advertencias. Además, cuenta con las características de EOL. Los ficheros *.evl* siguen una estructura con bloques obligatorios y opcionales.

```
1  context nameContext{
2
3      constraint nameConstraint {
4
5          guard: Bloque EOL
6
7          check: Bloque EOL
8
9          message: Bloque EOL
10
11         fix: Bloque EOL
12
13     }
14
15 }
```

Figura 3.5: *Estructura básica de Epsilon Validation Language*

La Figura 3.5 muestra un ejemplo de la estructura básica de un fichero *.evl* cuyos principales elementos se detallan a continuación.

- *nameContext*: En la línea 1 se define el nombre del tipo de instancia sobre la cual se van a evaluar las restricciones.
- *nameConstraint*: En la línea 3 se indica el nombre que se quiere dar a la restricción.
- *guard*: Este bloque, que aparece en la línea 5, no es obligatorio, tan solo se utiliza cuando la restricción depende de otra restricción. Si la restricción a la que se hace referencia en este bloque no se cumple, la restricción no se ejecuta. La nomenclatura es la siguiente:

```
guard : self.satisfies("primeraRestriccion")
```

- *check*: Este bloque, en la línea 7, es el único obligatorio. En este se define la restricción que se quiere evaluar, por ejemplo:

```
check {
        if (self.tFRelation == 3 and self.expression3.isDefined()) {
            return true;
        }
        return false;
    }
```

- *message*: En la línea 9 aparece este bloque, también opcional. En este bloque se define el mensaje que se muestra al usuario en el caso en el que la restricción no se cumpla.

```
message : 'Expression3 has to be defined'
```

- *fix*: Por último, en la línea 11 aparece el último bloque de la estructura que es opcional. En este bloque se define el código para reparar los errores detectados. Este bloque corrige el modelo para que se cumpla la restricción.

Esta estructura se replica tantas veces como restricciones se quieran crear para cada tipo de instancia.

Epsilon Generation Language

Epsilon Generation Language (EGL) es un generador de código basado en plantillas, es decir los programas EGL se asemejan al texto que generan. Está definido sobre Eclipse Epsilon y utiliza el lenguaje EOL. EGL proporciona un lenguaje adaptado para la transformación de modelo a texto (M2T). Un programa EGL está formado por una o más secciones que pueden ser de dos tipos: estáticas y dinámicas. Las secciones estáticas son aquellas cuyo contenido aparece textualmente en el texto que se genera. En las secciones dinámicas se utiliza el lenguaje EOL, permitiendo realizar bucles u operaciones personalizadas. En Capítulo 5 se detallará la parte de la sintaxis empleada en este proyecto.

Capítulo 4

Diseño, implementación y evaluación

En este capitulo se explica de manera detallada el desarrollo del proyecto, describiendo como se han utilizado las tecnologías explicadas en los puntos anteriores.

El propósito de este proyecto, como se ha comentado inicialmente, es crear una herramienta gráfica para poder diseñar modelos de análisis de datos en el ámbito de la diagnosis de enfermedades, aunque sería de aplicación en cualquier área en la que el formalismo utilizado fuese adecuado.

4.1. Marco de diseño

El marco de diseño desarrollado para modelos basados en autómatas fuzzy consiste en un lenguaje de modelado de dominio específico y un editor gráfico.

El lenguaje de modelado diseñado se divide en dos partes. Por una parte, una sintaxis abstracta definida mediante un metamodelo y un conjunto de reglas de validación que permitirán determinar si el modelo está bien definido o no. Por otra parte, una sintaxis concreta gráfica, que permite representar de modo gráfico los conceptos de la sintaxis abstracta.

Metamodelo

Este proyecto ha sido desarrollado en el marco de *Eclipse Epsilon*. El diseño del metamodelo ha sido realizado mediante la creación de un *Graphical editor project* en el que se ha definido un fichero *.emf*.

En este fichero el metamodelo ha sido implementado mediante el lenguaje *Emfatic*. Emfatic es un lenguaje de diseño que se utiliza para representar los modelos EMF Ecore en forma de texto.[19]

En la Figura 4.1 se puede ver parte del fichero *.emf* correspondiente al metamodelo del autómata fuzzy definido en el Capítulo 2.1. A continuación, se detallan los principales componentes incluidos en el metamodelo.

fuzzyAutomaton.emf

```
@namespace(uri="fuzzyAutomaton", prefix="fuzzyAutomaton")
package fuzzyAutomaton;

@gmf.diagram(model.extension="fza", diagram.extension="fza diagram")
class FuzzyAutomaton {
  attr String[1] name;
  attr TNormType[1] tNorm;
  val State[+] states;
  val Transition[+] transitions;
  val VariableSet[1] variableSet;
  val TransitionFeature[+] transitionFeatures;
}

enum TNormType {
  HAMACHER = 0;
  GODEL =1;
}

@gmf.node(label.placement="none", margin="10", figure="ellipse", resizable="false", tool.name="State", border.width="2", border.color="154,205,50", color="154,205,50")
class State {
  attr Boolean[1] isInitial = false;
  ref Transition[*]#target incoming;
  ref Transition[*]#source outgoing;
}

@gmf.link(source="source", target="target", target.decoration="arrow",
color="255,140,0", width="3", tool.name="Transition", tool.description="Add a transition")
class Transition {
  ref State[1]#outgoing source;
  ref State[1]#incoming target;
  ref TransitionFeature[1]#featureToTransition feature;
}

@gmf.node(label.readOnly= "true", size="125,150", label="name", tool.name="Variable Set")
class VariableSet {
  readonly attr String[1] name="Variable Set";

  @gmf.compartment(layout="list")
  val Variable[+] variables;
}

@gmf.node( label="name", tool.name="Variable", resizable="false")
class Variable {
  attr String[1] name;
  attr Double[1] value= 0;
}

@gmf.node(label.readOnly= "true",label="name",size="160,160", tool.name="Transition Features")
class TransitionFeature {
  readonly attr String[1] name="Transition Feature";

  @gmf.link(style="dash", width="3", color="105,105,105")
  ref Transition[+]#feature featureToTransition;

  @gmf.compartment(layout="list", collapsible="false")
  val Action[1] action;

  @gmf.compartment(layout="list", collapsible="false")
  val FuzzyConstraint[0..1] fuzzyConstraint;
```

1 3 6 4 5 2

Figura 4.1: *Fichero .emf*

- En primer lugar, es necesario añadir un *package* y darle un nombre (1). El programa, cuando se compile, creará un modelo con un paquete denominado con el nombre que se ha definido, en nuestro caso *fuzzyAutomaton*. El resto de elementos que se definan

se crearán dentro de este paquete.

- A continuación, se definen todos los elementos que constituyen nuestro formalismo. Para ello se crean clases que corresponden con cada uno de estos elementos, específicamente: las clases que representan los estados, el conjunto de variables, las variables, las acciones y las transiciones. En cada clase se añaden los atributos correspondientes a cada elemento. Por ejemplo, en la clase *Variable* (2) se han incluido los atributos nombre (de tipo *String*) y valor (de tipo *Double*). Cada atributo tiene definidas entre corchetes [ ] las expresiones de multiplicidad. En este caso solo existe un atributo de cada tipo para cada variable. Además los atributos se pueden inicializar, mediante un valor por defecto, como el atributo valor, que está inicializado a 0.

- Además de los atributos, las clases pueden *contener* otras clases. Este es el caso de la clase *FuzzyAutomaton* (3) que contiene entre uno y varios elementos (+) de tipo *State*, *Transition* y *TransitionFeature* y un único *VariableSet*. Sí se elimina la clase *FuzzyAutomaton* las clases que contienen dejan de existir. La forma para referenciar que un elemento contiene a otros se indica mediante *var*.

- Las clases también pueden *referenciar* a otras, mediante la palabra reservada *ref*. A diferencia del caso anterior, si se elimina la clase que referencia, no se eliminan las clases referenciadas. En nuestro metamodelo, la clase *State* (4) tiene dos referencias a la clase *Transition*. El símbolo # seguido de un identificador nombra la referencia opuesta que se declara. Estas dos referencias tienen una multiplicidad entre cero y varios elementos (*) de cada tipo de transición.

- Emfatic también permite utilizar modificadores de los atributos, como el utilizado en la clase *VariableSet* (5), *readonly*, que sirve tanto para atributos como referencias.

- Emfatic también permite la creación de enumerados, como en nuestro caso para los tipos de *TNorms* (6).

Reglas de validación

Como ya he mencionado, la sintaxis abstracta es uno de los dos componentes que forman el lenguaje de modelado. Esta sintaxis se define mediante el metamodelo, definido anteriormente, y un conjunto de reglas de validación. Estas reglas se utilizan para especificar restricciones que deben ser satisfechas por los modelos y que no pueden ser reflejadas en el metamodelo.

Aunque estas reglas de validación podrían ser asociadas al metamodelo mediante su definición como restricciones OCL, en nuestro caso hemos decidido utilizar el lenguaje EVL proporcionado por Epsilon. EVL permite establecer dependencias entre diferentes restricciones, mostrar mensajes de error e incluso establecer reparaciones de posibles errores detectados en los modelos.

Para poder utilizar EVL es necesario crear un nuevo *Plug-in* en el proyecto, como aparece en la Figura 4.2.

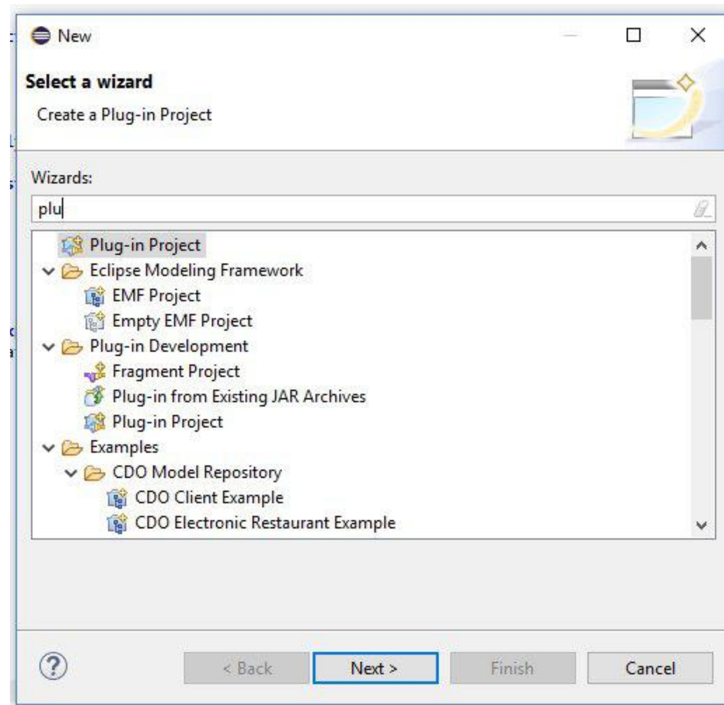

Figura 4.2: *Creación Plug-in EVL*

Una vez creado el Plug-in, se insertan en la pestaña *MANIFEST.MF* las dependencias *org.eclipse.ui.ide* y *org.eclipse.epsilon.evl.emf.validation*. A continuación, se crea una nueva carpeta, en nuestro caso la hemos llamado *validation* y dentro de esta se crea un fichero *fuzzyautomaton.evl*. Es en este dónde se añaden las reglas de validación en lenguaje EVL. Por último, se enlazan las restricciones al editor. Para ello hay que agregar la extensión *org.eclipse.epsilon.evl.emf.validation* y editarla como aparece en la Figura 4.3.

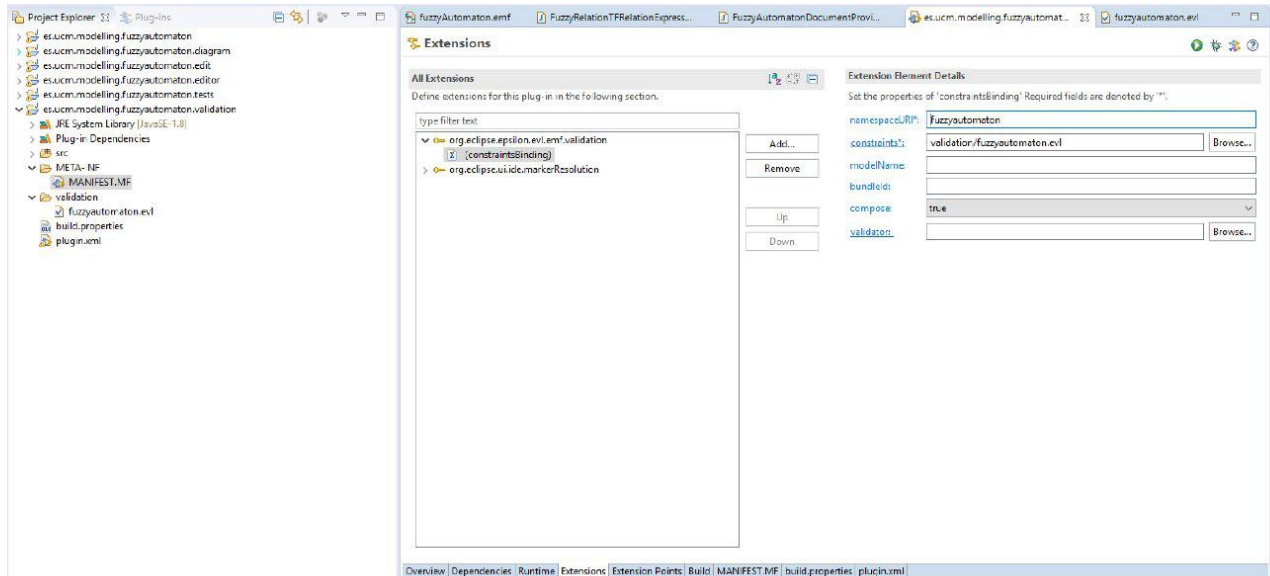

Figura 4.3: *Extensión EVL*

Finalmente se agrega la extensión *org.eclipse.ui.ide.markerResolution* y se crean dos *markerResolutionGenerator* con los detalles que se muestran en las Figuras 4.4 y 4.5.

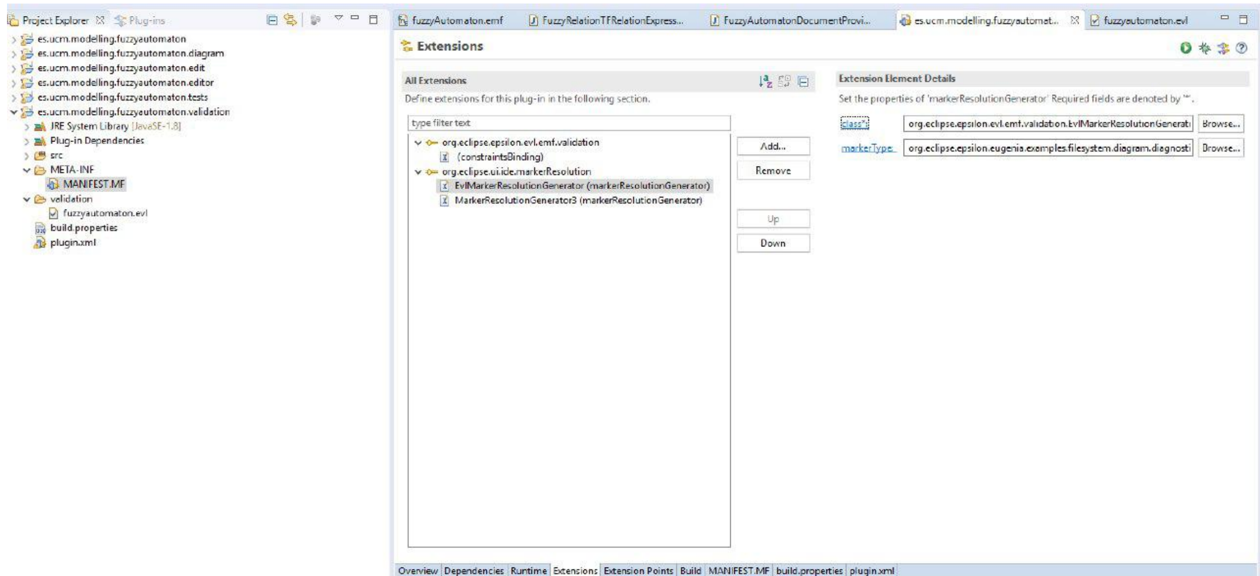

Figura 4.4: *Extensión EVL*

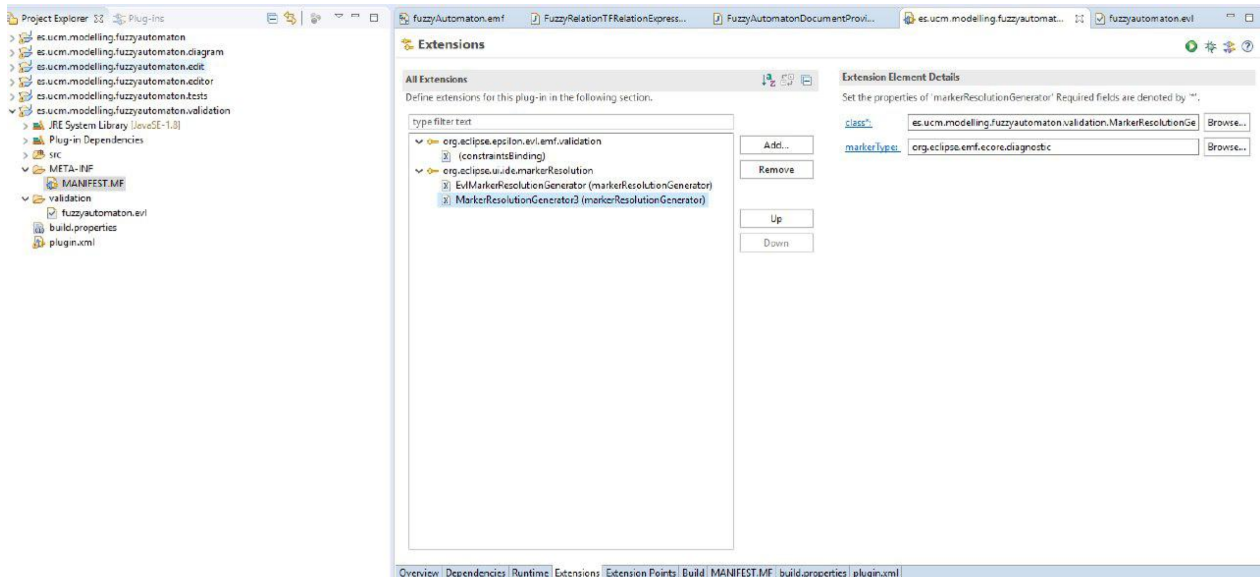

Figura 4.5: *Extensión EVL*

A continuación, se detallan las restricciones que se han creado para este proyecto:

- La primera de las restricciones de nuestro modelo es que debe existir un único estado inicial. Para esta validación se utilizan dos reglas, que se muestran en la Figura 4.6.

En primer lugar se chequea que no haya más de un estado inicial y en segundo lugar que se ha incluido al menos uno. Si alguno de los dos casos no se cumple aparecerán los mensajes de error correspondientes.

```
15 // The automaton only presents an initial state
16
17 context FuzzyAutomaton {
18
19
20     constraint InitialStateValidation1 {
21
22
23         check:  self.states.select(s|s.isInitial).size()<=1
24
25         message : 'The model presents more than one initial state'
26
27     }
28
29     constraint InitialStateValidation2 {
30
31
32         check:  self.states.select(s|s.isInitial).size()=1
33
34         message : 'The model must include an initial state'
35
36     }
37 }
38
```

Figura 4.6: *Restricción EVL sobre FuzzyAutomaton*

- Otra de las validaciones que se deben realizar es que los nombres de las variables no empiezan por un dígito. Para ello se ha creado una restricción, que se muestra en la Figura 4.7, en la que se comprueba que los identificadores de las variables encajan en el patrón indicado.

```
38
39 // Variable's name does not begin with a digit
40
41 context Variable {
42
43    constraint VariableNameValidation {
44
45         check : self.name.matches("^[^0-9][A-Za-z0-9_]*")
46
47         message : 'Variable name cannot begin with a digit '
48
49
50     }
51 }
52
```

Figura 4.7: *Restricción EVL sobre Variable*

- En lo referente a los identificadores de las variables, es necesario comprobar que nos se repiten. Cada variable, dentro del conjunto de variables, debe tener un nombre único. La restricción que se ha creado para este caso es la que aparece en la Figura 4.8.

```
55 // Variable set does not include duplicate variables
56
57 context VariableSet {
58
59
60     constraint DiffVariableNameValidation {
61
62
63         check {
64
65             var duplicate : Integer;
66
67             for (variable in self.variables) {
68
69                 if (not(self.variables.one(v| v.name=variable.name)))   {duplicate+=1;}
70             }
71
72             return duplicate = 0;
73         }
74
75
76         message : 'Different variables have the same name'
77
78     }
79
80 }
```

Figura 4.8: *Restricción EVL sobre VariableSet*

- Algunas restricciones pueden contener una condición previa en el bloque *Guard*. Si no se cumple la condición indicada la restricción no se comprueba, como ocurre en el caso de la Figura 4.9. Esta restricción corresponde a las instancias *Output* y comprueba que tenga definida alguna *Expression*. Si este es el caso, comprueba que solo contenga operadores aritméticos.

```
//Expressions associated to outputs only contain arithmetic operators

context Output {

    constraint arithmeticExpressionValidation{

        guard:  self.expression.isDefined()

        check {

            var correct : Boolean= true;

            for (expr in self.expression) {

                if (not expr.matches("^[^<>=&!]*$"))    {correct=false;}
            }

            return correct;

        }

        message : 'Expressions only can contain arithmetic operators '

    }

}
```

Figura 4.9: *Restricción EVL sobre Output*

- Cuando se crea un nuevo elemento de tipo *State*, este debe tener asociada al menos una *incoming transition* o una *outgoing transition*, como se refleja en la restricción de la Figura 4.10

```
context State {


    constraint StatesWithTransitionValidation {

        check: (self.incoming.size()>=1) or (self.outgoing.size()>=1)

        message : 'The states must include at least an incoming transition or an outgoing transition'

    }
}
```

Figura 4.10: *Restricción EVL sobre State*

No todas las restricciones se pueden definir mediante EVL debido a su complejidad, como el caso de la validación de expresiones aritméticas. EOL permite instanciar objetos Java usando el tipo de datos *Native* y así poder invocar a sus métodos. Para validar las expresiones aritméticas que se declaran en el modelo se ha utilizado la biblioteca *parsii*[29], que permite indicar el conjunto de variables validas. En la Figura 4.11 se puede ver el contenido de esta biblioteca. También, cuenta con un control de errores que durante la

validación no lanza ninguna excepción sino que recoge los errores detectados y los reporta todos de una sola vez.

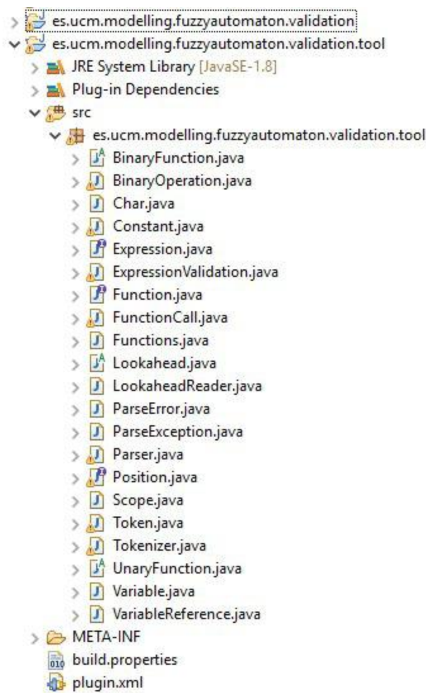

Figura 4.11: *Contenido de la biblioteca parsii*

En nuestro caso, decidimos utilizar este control de errores para validar las expresiones, creando una clase llamada *ExpressionValidation* en la que definimos una serie de métodos, como se puede ver en la Figura 4.12.

```java
package es.ucm.modelling.fuzzyautomaton.validation.tool;

import java.util.Collection;

public class ExpressionValidation {


    Scope scope;
    Expression expr;

    public ExpressionValidation() throws ParseException{

        scope = new Scope();

    }

    public String valExpression(String expression) throws ParseException {

        String errorMsg="";

        try {
            expr = Parser.parse(expression, scope);
        }

        catch (ParseException e) {
            errorMsg= e.getMessage();
        }

        return errorMsg;
    }

    public void setVariable(String variable) {

        Variable a = scope.create(variable);

    }


}
```

Figura 4.12: *Clase ExpressionValidation*

Para poder utilizar estos métodos es necesario incluir mediante EOL, una precondición, definida en la Figura 4.13, que instancia un objeto de la biblioteca *parsii* cada vez que se compile el fichero *.evl* y además, recorre todas la variables que se han definido en el modelo para que, mediante la invocación del método *setVariable(varible.name)*, sean incluidas en el *Scope* de validación.

```
fuzzyautoma...  Scope.java  ExpressionV...  Token.java  Tokenizer.java  Parser.java  VariableRef...  Map.class  Concurrent

pre ValidExpression {

    var expressionValidation := new Native('es.ucm.modelling.fuzzyautomaton.validation.tool.ExpressionValidation');

// All the variables declared in the model are included in the ExpressionValidation scope

    for (variable in Variable.getAllOfType()) {

        expressionValidation.setVariable(variable.name);
    }

}
```

Figura 4.13: *Precondición EOL, creación objeto parsii*

Las siguientes restricciones que se han incluido corresponden a la validación de las instancias de las entidades *VarUpdate* y *FuzzyRelation*. En la Figura 4.14 se encuentra una de estas restricciones, que valida que las *FuzzyRelations*, binarias presentan dos expresiones.

Una vez comprobado, se invoca al método *valExpression(expression)*, definido en la Figura 4.12, que verifica que las expresiones son correctas.

```
241    constraint CorrectExpressionFuzzyRelationBIN {
242
243        guard: self.tFRelation <> FuzzyRelationType#TERN and self.expression1.isDefined() and self.expression2.isDefined()
244
245
246        check   { var errorMsg : String="";
247
248                  errorMsg= expressionValidation.valExpression(self.expression1);
249                  errorMsg=  errorMsg + expressionValidation.valExpression(self.expression2);
250
251                  return errorMsg ="";
252
253                  }
254
255        message : "Incorrect expression. "  + errorMsg
256
257        }
258    }
```

Figura 4.14: *Restricción EVL sobre FuzzyRelation*

Sintaxis gráfica

Después de detallar la sintaxis abstracta y el conjunto de reglas de validación, se especifica la sintaxis concreta, en este caso, que se ha creado para diseñar los modelos conformes al metamodelo de los autómatas fuzzy.

Para ello se han añadido a nuestro fichero *.emf* una serie de anotaciones proporcionadas por Eugenia. A continuación, con ayuda de la Figura 4.15, se van a detallar las anotaciones que se han incluido en nuestro metamodelo.

```
fuzzyAutomaton.emf
1  @namespace(uri="fuzzyAutomaton", prefix="fuzzyAutomaton")
2  package fuzzyAutomaton;
3
4  @gmf.diagram(model.extension="fza", diagram.extension="fza_diagram")   1
5  class FuzzyAutomaton {
6    attr String[1] name;
7    attr TNormType[1] tNorm;
8    val State[+] states;
9    val Transition[+] transitions;
10   val VariableSet[1] variableSet;
11   val TransitionFeature[+] transitionFeatures;
12 }
13
14 enum TNormType {
15   HAMACHER = 0;
16   GODEL =1;
17 }
18                                                                      2
19 @gmf.node(label.placement="none", margin="10", figure="ellipse", resizable="false", tool.name="State", border.width="2", border.color="154,205,50", color="154,205,50")
20 class State {
21   attr Boolean[1] isInitial = false;
22   ref Transition[*]#target incoming;
23   ref Transition[*]#source outgoing;
24 }
25
26 @gmf.link(source="source", target="target", target.decoration="arrow",
27 color="255,140,0", width="3", tool.name="Transition", tool.description="Add a transition")   3
28 class Transition {
29   ref State[1]#outgoing source;
30   ref State[1]#incoming target;
31   ref TransitionFeature[1]#featureToTransition feature;
32 }
33
34 @gmf.node(label.readOnly= "true", size="125,150", label="name", tool.name="Variable Set")
35 class VariableSet {
36   readonly attr String[1] name="Variable Set";
37
38   @gmf.compartment(layout="list")   4
39   val Variable[+] variables;
40 }
41
42 @gmf.node( label="name", tool.name="Variable", resizable="false")
43 class Variable {
44   attr String[1] name;
45   attr Double[1] value= 0;
46 }
47
48 @gmf.node(label.readOnly= "true",label="name",size="160,160", tool.name="Transition Features")
49 class TransitionFeature {
50   readonly attr String[1] name="Transition Feature";
51
52   @gmf.link(style="dash", width="3", color="105,105,105")
53   ref Transition[+]#feature featureToTransition;
54
55   @gmf.compartment(layout="list", collapsible="false")   5
56   val Action[1] action;
57
58   @gmf.compartment(layout="list", collapsible="false")
59   val FuzzyConstraint[0..1] fuzzyConstraint;
60
```

Figura 4.15: *Anotaciones GMF*

- En primer lugar, se define cual es la raíz de nuestro metamodelo añadiendo la anotación *@gmf.diagram*, en este caso la clase *FuzzyAutomaton* (1). Las anotaciones pueden tener propiedades que permiten incluir más detalles acerca de las mismas. En el caso de la clase principal se ha definido la extensión del modelo (*model.extension* = *"fza"*) y la extensión del archivo del diagrama (*diagram.extension* = *"fza_ diagram"*).

- Otra de las anotaciones utilizadas es *@gmf.node*, para indicar que elementos son de tipo nodo. Este es el caso de la clase *State* (2). Mediante el uso de las diferentes propiedades se ha definido la apariencia que estos elementos tendrán en el diagrama: *label.placement* = *"none"* (define donde se coloca la etiqueta con respecto al nodo,

en este caso como se iguala a *none* no habrá etiqueta), *margin* = *"10"* (número de unidades en el margen de inserción para el nodo), *tool.name* = *"State"* (nombre en la herramienta), *figure* = *"ellipse"* (forma en la herramienta, en este caso de elipse), *resizable* = *"false"* (si se puede redimensionar el nodo), *border.width* = *"2"* (define el ancho del borde), *border.color* = *"154,205,50"* (color RGB establecido para el borde) y *color* = *"154,205,50"* (color RGB establecido para el fondo del nodo).

- La anotación *@gmf.link* indica que componentes son de tipo enlace, como el caso del elemento *Transition* (3). Entre las propiedades asociadas a esta anotación podemos distinguir: *source*= *"source"* (indica la referencia origen del enlace) o *target* = *"target"* (define la referencia destino), *target.decoration* = *"arrow"* (forma que tendrá el destino del enlace, en este caso se ha definido como una flecha), *width* = *"3"* (grosor de la forma) o *tool.description* = *"Add a transition"* (para añadir la descripción que aparecerá en la herramienta de creación).

- La componente *VariableSet* tiene asociada una lista de variables (4). Para ello se ha utilizado la anotación *@gmf.compartment*. Esta crea un compartimento donde se colocan los elementos, en este caso de tipo *Variable*, con la propiedad *layout* = *"list"* (permite ordenarlos en forma de lista).

- Las clases de tipo nodo pueden hacer referencia a elementos que a su vez sean enlaces o compartimentos, es el caso del elemento *TransitionFeature* (5). Este hace referencia a un elemento de tipo enlace (*Transition*), que contiene la propiedad *style* = *"dash"* (dibuja una línea discontinua). Además creará dos compartimientos en los que se encontrarán los elementos *Action* y *FuzzyConstraint*.

Editor gráfico

Como comentamos al principio de esta sección, nuestro entorno de diseño proporciona un editor gráfico que permitirá a los usuarios el modelado de autómata fuzzy de forma visual,

así como la posterior validación automática de los modelos conforme al metamodelo y las reglas de validación definidas.

Después de añadir las anotaciones en el fichero *.emf*, utilizamos la herramienta *Eugenia*, para generar una serie de modelos intermedios, como se muestra en la Figura 4.16.

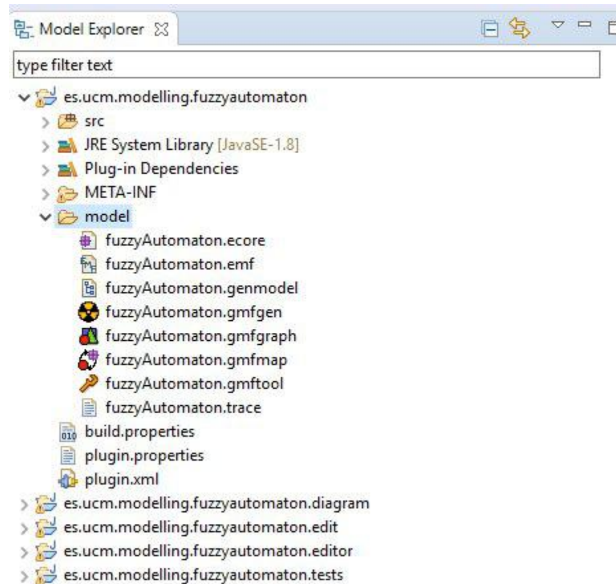

Figura 4.16: *Generación GMF Editor*

Uno de los ficheros generados es el *.ecore* del que parten el resto de procesos. En la Figura 4.17 se muestra su vista en árbol, donde aparecen todas las entidades creadas en el metamodelo anteriormente definido, con sus anotaciones y propiedades.

Figura 4.17: *Fichero .ecore*

A partir del *.ecore* podemos inicializar el *Ecore Diagram*. Este representa el metamodelo en forma de diagrama, como aparece en la Figura 4.18.

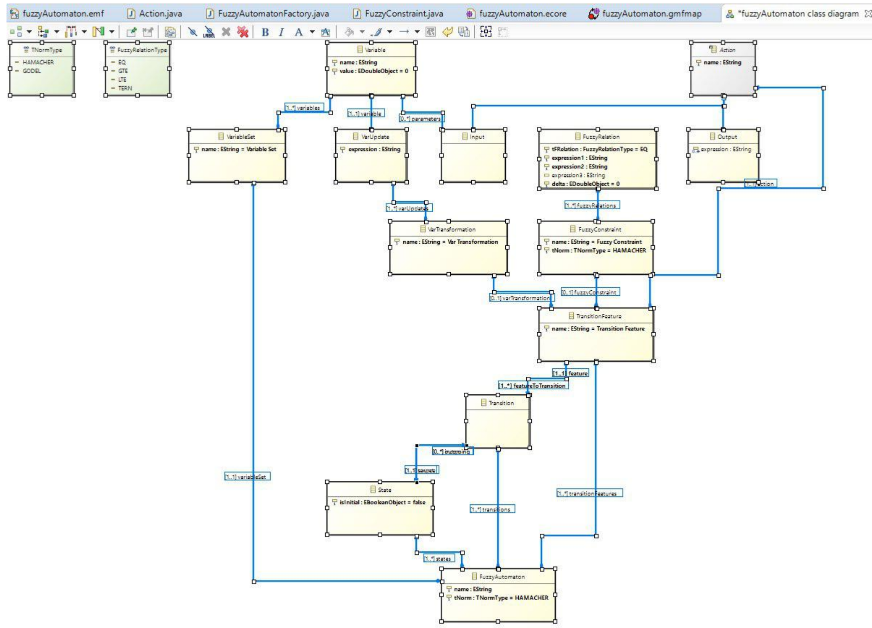

Figura 4.18: *Ecore Diagram*

Además de la generación de los modelos intermedios se crean los proyectos: *Edit* y *Tests*, mencionados en el Capítulo 3.1.

Cuando se modifica manualmente uno de estos modelos se debe volver a regenerar todos los modelos intermedios mediante Eugenia. Esto implica que todas las modificaciones realizadas manualmente en ellos son sobrescritas. La solución que hemos dado a este problema es la creación de *scripts EOL*, que junto al metamodelo creado en el fichero *.emf* personalizan los modelos GMF que se generan. Los cambios se han añadido en el script *Ecore2GMF.eol*, consiguiendo con ello que tras la generación de los modelos intermedios mediante Eugenia, los cambios que se desean realizar en los mismos vuelvan a ser incorporados. También se ha creado un script *FixGMFGem.eol*, en el que se ha personalizado el *modelo.gmfgen*. En otros casos la solución que se ha aplicado es el uso de la anotación @generated NOT a los métodos que no deseábamos que fueran sobrescritos.

Una vez que se han generado todos los modelos y hemos creado los scripts para personalizar el resultado, se compila y genera el editor gráfico. En la Figura 4.19 se muestra

la apariencia del mismo. En la parte izquierda de la imagen aparecen los dos ficheros que definimos: el modelo (*.fza*) y el diagrama (*fza_ diagram*) y en la parte derecha la paleta de herramientas con todas las entidades que se han creado, divididas entre nodos (Objects) y enlaces (Connections).

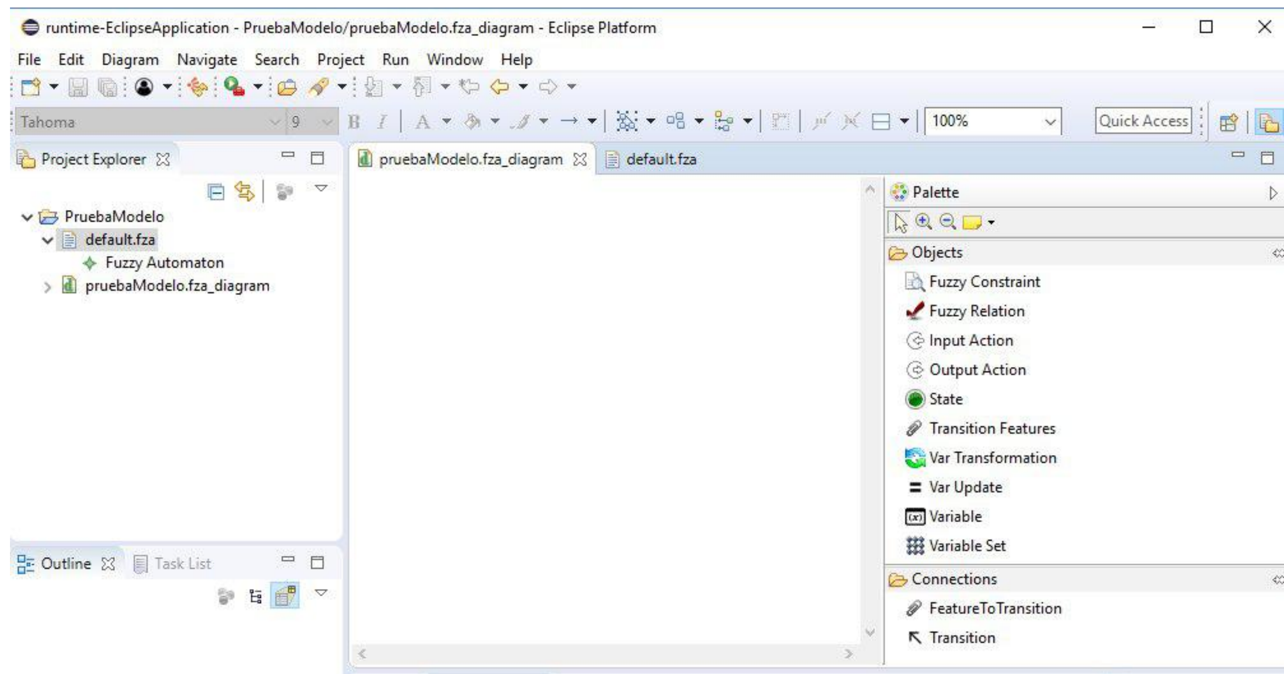

Figura 4.19: *Editor gráfico*

Desde la paleta del editor gráfico se pueden incluir los distintos elementos que conforman los modelos que se deseé diseñar. Al guardar el modelo se validan automáticamente todas las restricciones que definimos anteriormente, permitiendo solo crear modelos bien formados. En la Figura 4.20 se muestra un ejemplo de un modelo que hemos creado y de los errores que se han producido:

- *Expressions only can contain arithmetic operators*: este error ocurre al realizar la validación de la restricción que se ha creado utilizando biblioteca *parsii*. Existe un error debido a que una de las expresiones que contiene la *Fuzzy Relation* se ha definido con operadores que no son aritméticos (5&5). En este caso también se muestra otro

mensaje de error más: *Incorrect expression. 1:2: Unexpected token: &. Expected an expression.* Este mensaje es una de las excepciones que se lanzan durante la validación.

- *The required feature 'name' of '<FuzzyAutomaton>' must be set*: este error se produce al validar una de las restricciones de cardinalidad. Al crear el elemento *FuzzyAutomaton* se definió que este debe tener obligatoriamente un nombre.

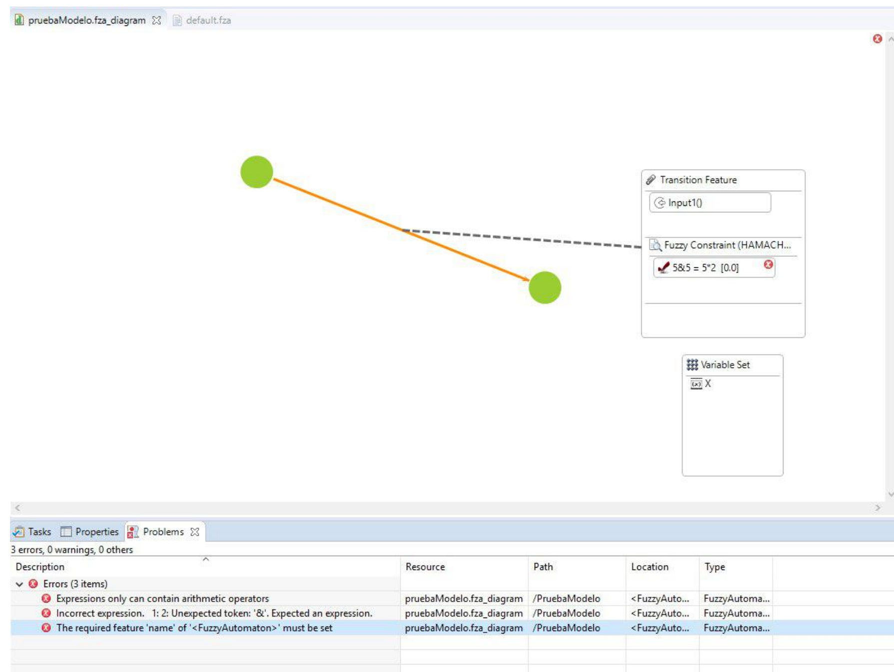

Figura 4.20: *Ejemplo de modelo*

En la Figura 4.21 se muestran las ventanas de propiedades de los elementos *FuzzyRelation* y *FuzzyAutomaton*. Desde estas vistas se pueden modificar todas las propiedades de los elementos y corregir los errores.

Tasks | Properties | Problems

Undefined

Core · Rulers & Grid · Appearance

Property	Value
Name	
TNorm	HAMACHER

Tasks | Properties | Problems

FuzzyRelation

Core · Appearance

Property	Value
Delta	0.0
Expression1	5&5
Expression2	5*2
Expression3	
TF Relation	EQ

Figura 4.21: *Vista propiedades*

4.2. Transformación Modelo Texto

Con el objetivo de poder utilizar los modelos en la herramienta AUNTY y así poder analizar los datos correspondientes, se ha diseñado un proceso de transformación de modelo a texto. El fichero generado presentará el formato de importación requerido por dicha herramienta.

Este proceso requiere el diseño de un conjunto de reglas de transformación. Estas reglas se han creado en el lenguaje *EGL* detallado en el Capitulo 3.3. A continuación se detallan algunas de las reglas de transformación diseñadas. En la Figura 4.22 aparece la regla de transformación que comprueba si el *fuzzyAutomaton* tiene definido una *tNorm* y si no es así indica el valor por defecto correspondiente a *HAMACHER*.

```
1  [* Finite Automaton TNorm *]
2  [%
3      var faNorm: String="HAMACHER";
4      var fa = t_fuzzyAutomaton.all.first();
5
6      if (fa.a_tNorm.isDefined()) {faNorm=fa.a_tNorm;}
7  %]
8  [%=faNorm%]
```

Figura 4.22: *Regla de transformación T-Norms*

La siguiente transformación permite generar la lista de variables definidas en el modelo.

```
[* Set of variables. decVars stores the index of the variables and the corresponding names*]
[%
   var listVars: String=""; var decVars: Map;  var iString: String; var i: Integer=0;

   for (variable in t_variables.all) {

       if (listVars="") {listVars= variable.a_name;}
       else {listVars=listVars + "," + variable.a_name;}

       iString=""+ i;
       decVars.put(iString,variable.a_name);
       i++;
}%]
[%=listVars%]
```

Figura 4.23: *Regla de transformación Variables*

La especificación de las *Fuzzy Relations*, se realiza mediante la regla de transformación de la Figura 4.24, en la que recuperamos las *fuzzyRelations* y en función del tipo de relación se asocian las expresiones correspondientes.

```
[* Fuzzy relations associated with the transition*]
[%

       var frelations = fconstraint.children.select(a|a.tagName = "fuzzyRelations");

       for (fr in frelations) {

           switch (fr.a_tFRelation){
               case "LTE": op="<=";
               case "TERN": op="<=";
               case "GTE": op=">=";
               default: op="=";
           }

           listFRel=listFRel + fr.a_expression1 + op + fr.a_expression2;
           if (fr.a_tFRelation="TERN") {listFRel=listFRel + op + fr.a_expression3;}
           listFRel=listFRel + " [" + fr.a_delta + "]"  + ",";
       }
       lastPos= listFRel.length()-1;
       listFRel=listFRel.substring(0,lastPos);
    }
    else {listFRel="True";}
%]
[%=listFRel%]
```

Figura 4.24: *Regla de transformación Fuzzy Relations*

En la Figura 4.25 se muestra un ejemplo del fichero obtenido del proceso de transformación de un modelo gráfico.

```
export.txt: Bloc de notas
Archivo  Edición  Formato  Ver  Ayuda
HAMACHER
Fe,Hgb,Ca,TG2,Hto,GOT,GPT,GGT,numH,Neutrof,Eosinof,Basof,Mono,Linfo
s0,s1,s2,s3,s4,s5,s6,s7,s8,s9,s10,s11,s12,s13
s8
s0 s1
?Hierro Fe
HAMACHER
40<=Fe<=100 [3.0]
ID
s0 s2
?Hierro Fe
HAMACHER
40>=Fe [3.0]
ID
s3 s4
?GOT GOT
HAMACHER
10<=GOT<=50 [2.0]
ID
s4 s5
?GPT GPT
HAMACHER
9<=GPT<=39 [1.5]
ID
s5 s6
?GGT GGT
HAMACHER
5<=GGT<=36 [1.5]
ID
s8 s0
?NumHistoria numH
True
ID
s1 s9
?Hgb Hgb
HAMACHER
12<=Hgb<=15 [0.15]
```

Figura 4.25: *Ejemplo fichero de transformación*

4.3. Producto Final

Para mejorar la usabilidad de la herramienta decidimos crear un producto para facilitar la instalación del entorno de diseño. La exportación del producto crea un ejecutable de Eclipse, de modo que, el usuario sólo necesitará lanzar el ejecutable para disponer de la herramienta.

Para conseguir este producto es necesario crear un nuevo plugin, *Feature Proyect* e insertar todas las dependencias del proyecto en el fichero *feature.xml*, como aparece en las Figuras 4.26 y 4.27.

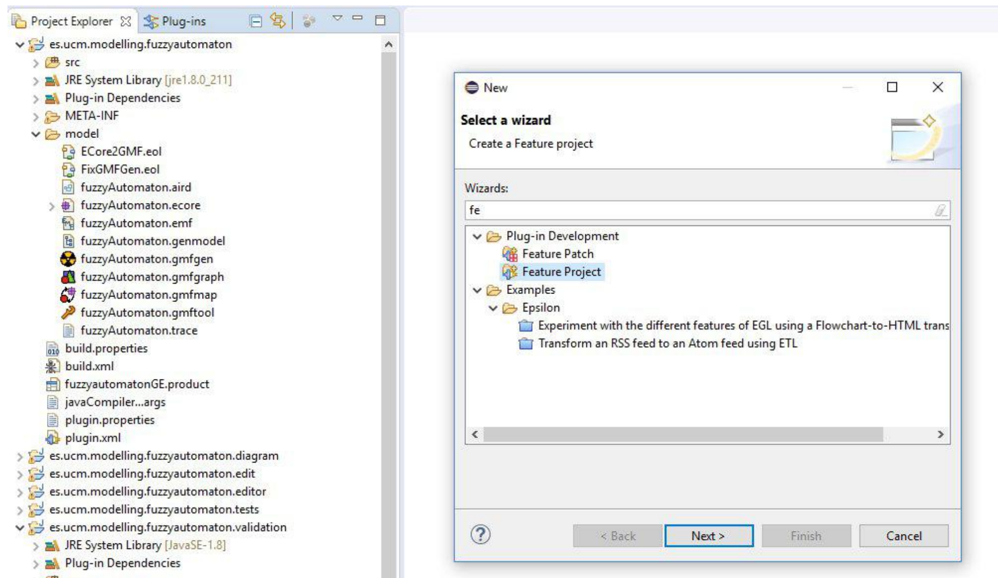

Figura 4.26: *Creación plugin Feature*

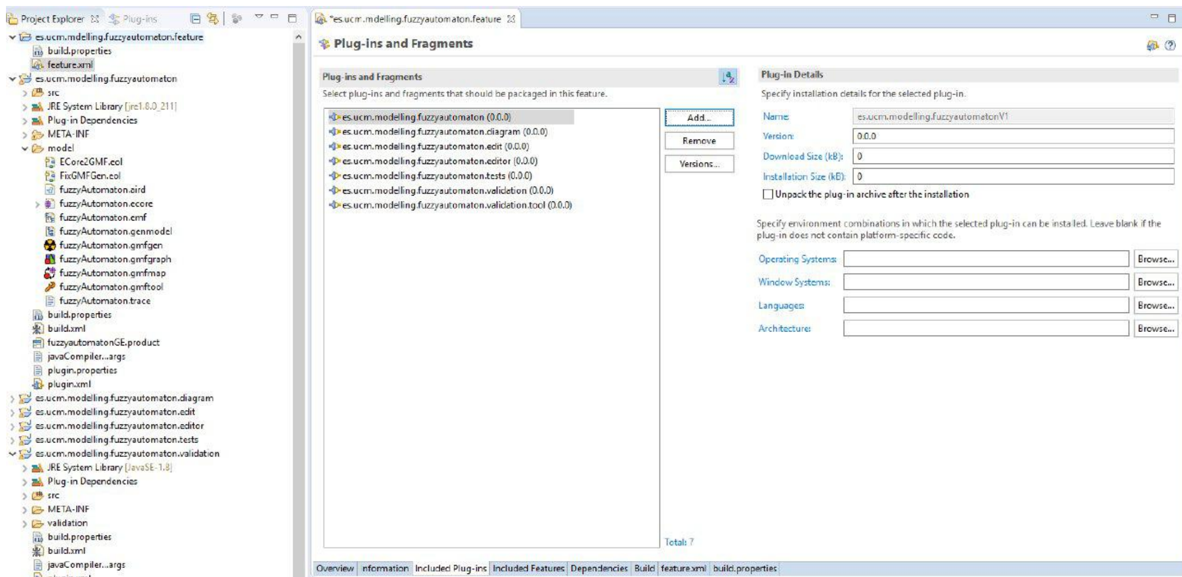

Figura 4.27: *Creación plugin Feature*

A continuación, se crea un nuevo plugin *Product*, en el que añadiremos un nuevo fichero, *configuration* en el que se basará la exportación y que incluirá como dependencia el plugin *feature*, tal y como se muestra en las Figuras 4.28 y 4.29.

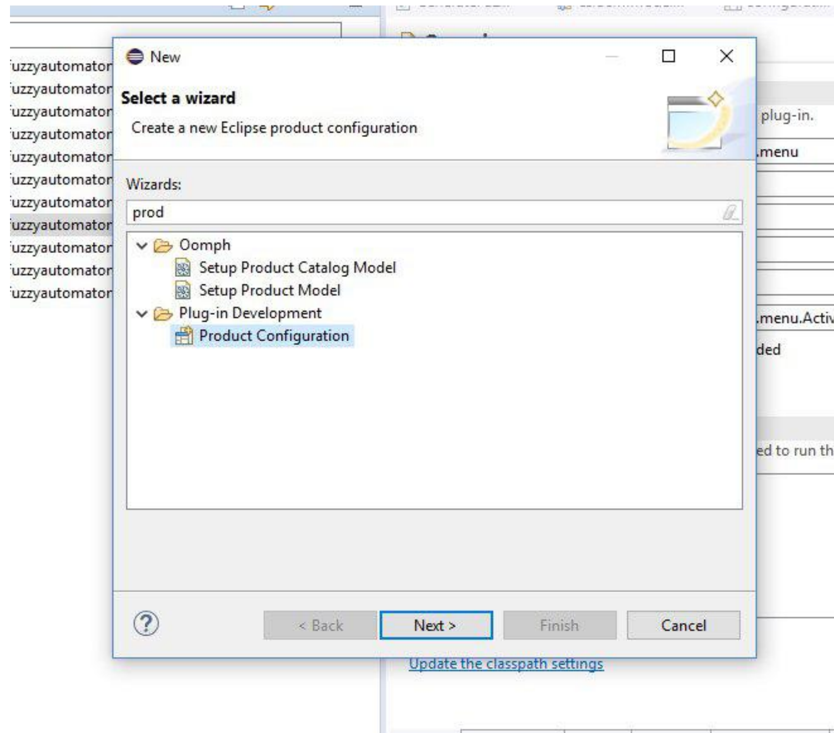

Figura 4.28: *Creación plugin Product*

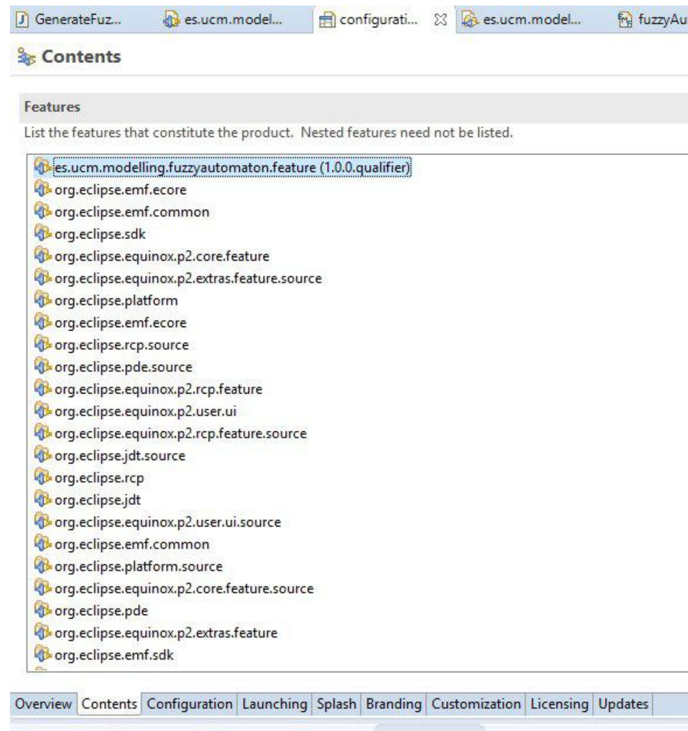

Figura 4.29: *Creación plugin Product*

También se ha incluido el plugin encargado de realizar la transformación del modelo (conforme al metamodelo *FuzzyAutomaton*), a texto.

Por último, se decidió crear una entrada de menú para que desde el editor, el usuario pudiera generar el fichero correspondiente a la transformación del modelo en el directorio seleccionado.

En la Figura 4.30 aparece la entrada de menú.

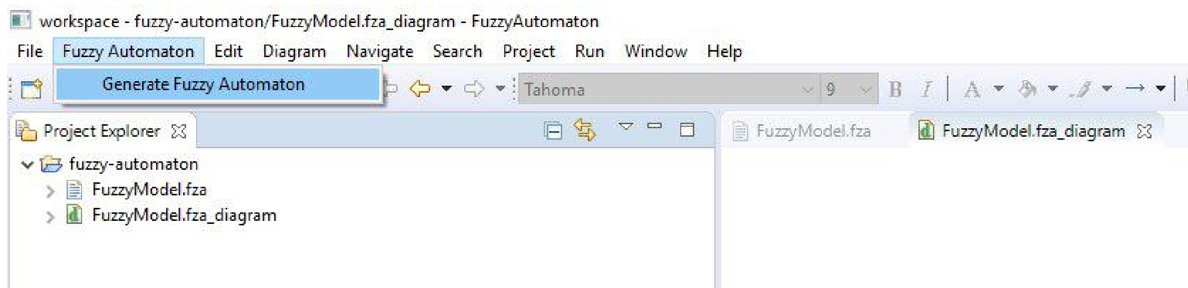

Figura 4.30: *Menú para la transformación m2t*

4.4. Caso de Estudio

Con el objetivo de evaluar la aplicabilidad de nuestro editor gráfico se ha llevado a cabo un caso de estudio en el ámbito del diagnóstico precoz de enfermedades. En concreto, y con la colaboración de doctores de la Facultad de Medicina de la Universidad Católica de Valencia, se ha desarrollado un modelo para soporte en la toma de decisiones a la hora de realizar pruebas diagnósticas a enfermos pediátricos con sospecha de padecer enfermedad celiaca.

La enfermedad celiaca se puede manifestar mediante numerosos y variados síntomas. Sin embargo, el posible diagnóstico de la enfermedad debería ser considerado en niños, que aun siendo asintomáticos, pertenezcan a un grupo de riesgo (por ejemplo genético), o presenten valores anormales en una serie de datos analíticos. Una vez se presentan los síntomas, y existe sospecha de la enfermedad, esta puede ser diagnosticada mediante análisis serológicos, análisis genéticos y biopsias intestinales. No obstante, es deseable, predominantemente en el caso de los niños, un diagnostico basado en datos clínicos y serológicos, con el fin de evitar la biopsia intestinal.

En nuestro caso de estudio se ha diseñado un protocolo de análisis de datos analíticos que ayude a los médicos a determinar la necesidad de realizar pruebas diagnósticas adicionales. Este protocolo se basa en el estudio de la combinación de valores de parámetros fuera de rango con cierto grado de imprecisión y certidumbre. Este protocolo ha sido implementado en nuestro marco de diseño. En la Figura 4.32 se muestra un fragmento del modelo desarrollado. Este modelo fue exportado mediante la utilidad desarrollada para probar la correcta importación por parte de la herramienta AUNTY[8]. Como ya se mencionó, esta herramienta permite el análisis de datos mediante procesos de análisis implementados como autómatas fuzzy. En la figura 4.31 se encuentra parte del fichero generado por la transformación del modelo diseñado en el formato requerido por la herramienta para su importación por AUNTY. Como podemos apreciar, el diseño del autómata fuzzy en este formato tiene un nivel de complejidad que difícilmente podría ser manejable para profesionales médicos sin

los conocimientos técnicos necesarios. Esto demuestra que el marco de diseño desarrollado cumple con el objetivo principal de este trabajo, proporcionar a los expertos una herramienta gráfica que les permita abstraerse de complejidades técnicas y diseñar modelos de análisis de forma sencilla.

export.txt: Bloc de notas

Archivo Edición Formato Ver Ayuda

```
HAMACHER
Fe,Hgb,Ca,TG2,Hto,GOT,GPT,GGT,numH,Neutrof,Eosinof,Basof,Mono,Linfo
s0,s1,s2,s3,s4,s5,s6,s7,s8,s9,s10,s11,s12,s13
s8
s0 s1
?Hierro Fe
HAMACHER
40<=Fe<=100 [3.0]
ID
s0 s2
?Hierro Fe
HAMACHER
40>=Fe [3.0]
ID
s3 s4
?GOT GOT
HAMACHER
10<=GOT<=50 [2.0]
ID
s4 s5
?GPT GPT
HAMACHER
9<=GPT<=39 [1.5]
ID
s5 s6
?GGT GGT
HAMACHER
5<=GGT<=36 [1.5]
ID
s8 s0
?NumHistoria numH
True
ID
s1 s9
?Hgb Hgb
HAMACHER
12<=Hgb<=15 [0.15]
ID
s9 s3
?Calcio Ca
HAMACHER
Ca<=9.8 [0.2]
ID
s6 s10
?Neutro Neutrof
HAMACHER
1800<=Neutrof<=8100 [300.0]
ID
s10 s11
?Eosin Hgb,Fe
HAMACHER
50<=Eosinof<=300 [10.5]
ID
s11 s12
?Baso Hgb,Hgb
```

Figura 4.31: *Fichero generado por la transformación*

Aunque hubiéramos deseado utilizar datos analíticos para ser analizados con esta herramienta y permitir comprobar la validez del modelo diseñado desde el punto de vista funcional, esto no ha sido posible. La Universidad Católica de Valencia no ha recibido el permiso correspondiente para su uso. No obstante, este aspecto queda fuera de nuestro marco de desarrollo, ya que este se centra en el diseño del modelo y su exportación. El análisis de los datos se enmarcaría dentro de las funcionalidades proporcionadas por AUNTY.

El diseño de este modelo real nos ha permitido comprobar la facilidad de uso para

usuarios sin conocimientos técnicos y también recabar sugerencias para futuras mejoras tanto en la interfaz del editor gráfico como en las propiedades del autómata fuzzy. En particular, extender el metamodelo con nuevos componentes que permitan representar características adicionales en los modelos, como por ejemplo, variables de tipo no numérico.

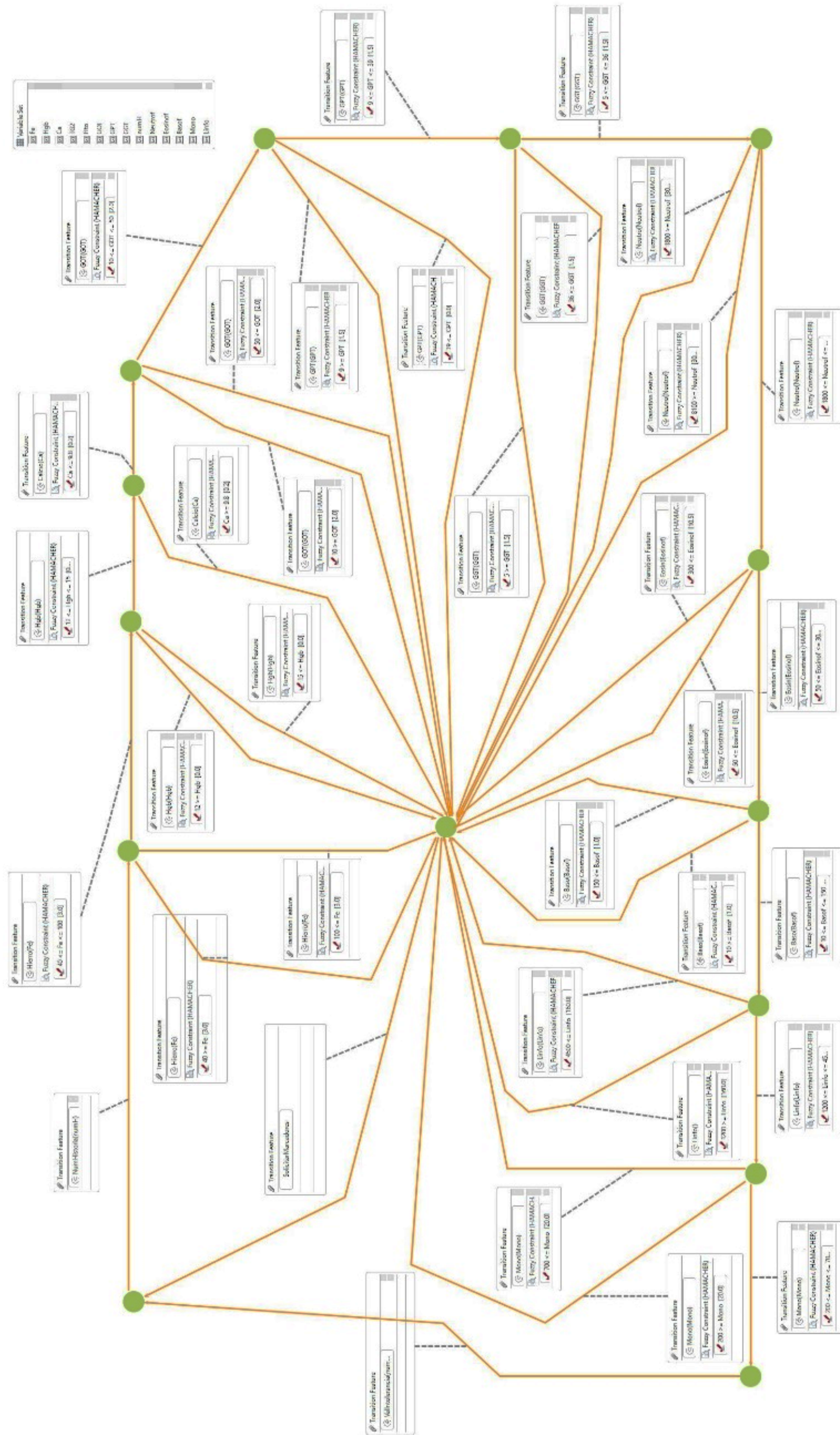

Figura 4.32: *Fragmento del modelo desarrollado*

Capítulo 5

Conclusiones

En este proyecto se ha abordado el desarrollo de un entorno de diseño de modelos basados en el formalismo de autómatas fuzzy. Aunque este desarrollo estaba inicialmente orientado al diseño de protocolos para el análisis de datos clínicos, con el fin de apoyar los diagnósticos precoces de enfermedades, podría ser aplicado en cualquier área en la que los modelos se ajusten a las características proporcionadas por dicho formalismo.

Este marco de diseño se basa en la definición de un metamodelo que representa la sintaxis abstracta correspondiente al autómata fuzzy. También se han especificado las reglas de validación necesarias para poder establecer la conformidad de los modelos con respecto a este formalismo. Teniendo en cuenta que se pretendía la simplificación del diseño de estos modelos, con el objetivo de que pudieran ser definidos por especialistas médicos que no disponen de los conocimientos técnicos necesarios, se ha desarrollado un entorno gráfico para su definición. Para poder explotar estos modelos se ha desarrollado un proceso de transformación de modelo a texto que puede ser importado y por tanto utilizado por la herramienta AUNTY para el análisis de los datos correspondientes. Para la evaluación de este marco se ha considerado un caso de estudio relativo al diagnóstico de enfermedad celiaca en pacientes pediátricos. La evaluación por parte de los usuarios ha sido satisfactoria, proporcionando información que ha permitido el refinamiento del modelo inicialmente diseñado.

Como trabajo futuro y como consecuencia del caso de estudio realizado, se pretende mejorar el entorno gráfico de diseño simplificando aún más el diseño. Asimismo, se exten-

derá el metamodelo para que permita establecer condiciones disyuntivas asociadas a las transiciones, así como el uso de variables de datos no numéricos.

Capítulo 6

Introduction

Formal methods allow rigorous modeling of systems, which provides a high degree of reliability in their development. One of the areas which demands a high degree of rigor and security in the systems which are used, the protocols which are applied and finally the analysis of the information is medicine. For this reason, it would be convenient the application of methodologies based on mathematical-based formalisms which allow us to provide a precise semantics.

In the literature, we can find works which propose formal methodologies in the field of medical devices and systems. Most of proposals are applied to pacemakers[13,25,26,31,33,34,39–41,50], infusion pumps[2,4,44,58] and hemodialysis systems[3,24,24,30].

These proposals deal with different phases of the development process, highlighting the modeling of the systems[2–4,24–26,30–32,34,39–41,44,45,50,58,60], for the subsequent application, in most of the cases, verification techniques[2,3,13,24,31,33,34,39,44,50,58] and validation[3,30,34]. To a lesser extent, it is also applied to code generation[25,33] and software testing[4,31]. The most used formalisms in these proposals are, principally, the *Timed Automata* [13,31,33,34,50] and others based on states such as *Extended Finite State Machines*[2,4], *Abstract State Machines*[3], *Z*[25,26] o *Algebraic State Transition Diagrams*[15].

Indeed, another aspect of medicine in which formal methods have also been applied is the design of the medical protocols[28,35,43,61]. This facilitates and allows the professionals a better understanding of the customer service process and improving their safety.

One of the fields in which the formal development of systems begins to gain importance in the field of health is disease prevention. The improvement in the processes of early diagnosis demands systems which process and analyze data of patients in a precise way, even in phases where they are still asymptomatic. In this line, it has been recently proposed an information analysis framework[9] that considers two relevant aspects in the analysis of clinical data: inaccuracy and uncertainty. In this paper, the authors present a *fuzzy* version of the automata that captures these characteristics, establishing certain tolerance levels when analyzing data as well as defining the restrictions which must be taken into account when determining the degree of confidence in the decisions made during the diagnostic process. This formalism also facilitates the representation of the correlation which must exist between the values of the parameters of interest for the detection of the pathology considered. This incorporates flexibility in the process, avoiding static analysis that can lead to wrong diagnoses. All these aspects make this formalism suitable for the use in the development of support systems for the early diagnosis of diseases, based on the knowledge of experts and the analysis of the data obtained in medical tests.

However, it is important to remember that these systems rely on models that should be designed by medical specialists, who have the required diagnostic knowledge, but they do not have the necessary technical knowledge. The tools used in the formal modeling frameworks previously discussed require technical skills which health experts have not acquired. Consequently, this limitation may reduce the use of these tools. Therefore, it is useful to provide experts with a graphic tool which allows them to abstract from technical complexities and design analysis models in simple way. With this objective this project has been carried out, in which it has been created a model design framework based on fuzzy automatons using a modeling language of specific domain and a graphic editor. In addition, it has been developed a transformation process of the models designed with this editor so that these models could be used by the AUNTY tool (AUtomatically aNalyze daTa using fuzzY automata)[8] for the analysis of the corresponding data. For the evaluation of this framework,

it has been considered a case of study related to the diagnosis of celiac disease in pediatric patients.

6.1. Objectives

The main purpose of this paper, as it has been already mentioned, is to develop an environment of design of models based on the formalism of fuzzy automaton to support the disease diagnosis. This aim can be displayed in the following sub-objectives.

- Design of a specific domain modeling language for fuzzy model design automata: (a) definition of abstract syntax using a metamodel and rules of necessary validation and (b) definition of the specific syntax associated, in this case graphic.
- Creation of a graphic editor which allows the graphic design and automatic validation of models conforming to the previously defined language.
- Design and implementation of a model transformation process for its import and exploitation by the AUNTY tool.

6.2. Workplan

To achieve these objectives, this paper has been developed in the following phases:

- Review of proposed works for the use of formal methods in the support are to the diagnosis of diseases as well as the availability of support tools for its design. Selection of the most suitable formalism for modelling.
- Design of graphic domain specific modelling language design for the selected formalism. This phase has required the decision of the development environment, tools and languages to carry out these tasks.
- Design of graphic editor.

- Implementation of the model transformation process.
- Evaluation of the design framework through a case study which allows establishing the applicability of it.

6.3. Report Structure

In this document we can distinguish a part in which the theorists foundations of work are described, and another in which the design and development of the project are described. Chapter 2 contains the preliminaries, where the main concepts used in this work are explained. Chapter 3 shows the work environment as well as the tools and languages which have been used for the implementation of this project. Chapter 4 deals with detailed information regarding the design, development and implementation of the modeling framework. In addition, it is introduced the case study carried out for the evaluation of the developed framework. Last but not least, in Chapter 5, the conclusions and possible future works are presented.

Capítulo 7

Conclusions

This project has dealt with the development of a model design environment based on the formalism of fuzzy automatons. Although this development was initially oriented to the design of protocols for the analysis of clinical data, with the aim of supporting early diagnosis of diseases, it is important to point out that it could be applied in any area where the models conform to the characteristics provided by said formalism.

This design framework is based on the definition of a metamodel which represents the abstract syntax that corresponds to the fuzzy automaton. Furthermore, the validation rules necessary to establish the conformity of the models with respect to this formalism have been specified. It is important to take into account that the simplification of the design of these models was intended, with the objective that they could be defined by medical specialists who do not have the necessary technical knowledge, a graphic environment has been developed for their definition. In order to exploit these models, it has been developed a model-to-text transformation process which can be imported and therefore used by the AUNTY tool for the analysis of the corresponding data. For the evaluation of this framework, it has been considered a case study regarding the diagnosis of celiac disease in pediatric patients. Fortunately, the evaluation made by the users has been satisfactory, providing information which has allowed the refinement of the initially designed model.

As a future work and as a consequence of the case study, it is intended to improve the graphic design environment by simplifying even more the design. Additionally, the metamo-

del will be extended in order to allow the establishment of disjunctive conditions associated with transitions, as well as the use of non-numerical data variables.

Bibliografía

[1] T. Allweyer. *BPMN 2.0*. BoD, 2010.

[2] R. Alur, D. Arney, E. L. Gunter, I. Lee, J. Lee, W. Nam, F. Pearce, S. Van Albert, and J. Zhou. Formal specifications and analysis of the computer-assisted resuscitation algorithm (cara) infusion pump control system. *International Journal on Software Tools for Technology Transfer*, 5(4):308–319, 2004.

[3] P. Arcaini, S. Bonfanti, A. Gargantini, and E. Riccobene. How to assure correctness and safety of medical software: the hemodialysis machine case study. In *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 344–359, 2016.

[4] D. Arney, R. Jetley, P. Jones, I. Lee, and O. Sokolsky. Formal methods based development of a pca infusion pump reference model: Generic infusion pump (gip) project. In *2007 Joint Workshop on High Confidence Medical Devices, Software, and Systems and Medical Device Plug-and-Play Interoperability (HCMDSS-MDPnP 2007)*, pages 23–33, 2007.

[5] S. Borgwardt, M. Cerami, and R. Peñaloza. The complexity of fuzzy el under the łukasiewicz t-norm. *International Journal of Approximate Reasoning*, 91:179–201, 2017.

[6] M. Brambilla, J. Cabot, and M. Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool Publishers, 1st edition, 2012. ISBN 1608458822, 9781608458820.

[7] I. Calvo, M. Merayo, and M. Núñez. An improved and tool-supported fuzzy automata framework to analyze heart data. In *Asian Conference on Intelligent Information and Database Systems*, pages 694–704, 2018.

[8] I. Calvo, M. Merayo, and M. Núñez. Aunty: A tool to automatically analyze data using fuzzy automata. In *2018 3rd International Conference on Computational Intelligence and Applications (ICCIA)*, pages 102–106, 2018.

[9] I. Calvo, M. Merayo, and M. Núñez. Ia methodology to analyze heart data using fuzzy automata. *Journal of Intelligent Fuzzy Systems*, En Prensa:1–11, 2019.

[10] S. Ceri, P. Fraternali, and A. Bongio. Web modeling language (webml): a modeling language for designing web sites. *Computer Networks*, 33(1):137 – 157, 2000.

[11] D. Crockford. *JavaScript: The Good Parts: The Good Parts.* .º'Reilly Media, Inc.", 2008.

[12] A. R. da Silva. Model-driven engineering: A survey supported by the unified conceptual model. *Computer Languages, Systems & Structures*, 43:139 – 155, 2015.

[13] A. David, M. O. Möller, and W. Yi. Formal verification of uml statecharts with real-time extensions. In *International Conference on Fundamental Approaches to Software Engineering*, pages 218–232, 2002.

[14] Ecured. Metamodelado-EcuRed. Available at https://www.ecured.cu/Metamodelado, 2019.

[15] T. Fayolle, F. Frappier, M.and Gervais, and R. Laleau. Modelling a hemodialysis machine using algebraic state-transition diagrams and b-like methods. In *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 394–408, 2016.

[16] E. Foundation. Emfatic. Available at https://wiki.eclipse.org/Emfatic, 2012.

[17] E. Foundation. Eclipse. Available at https://www.eclipse.org, 2019.

[18] E. Foundation. org.eclipse.emf.ecore (EMF javadoc). Available at https://download.eclipse.org/modeling/emf/emf/javadoc/2.6.0/org/eclipse/emf/ecore/package-summary.html, 2019.

[19] E. Foundation. Emfatic. Available at https://www.eclipse.org/epsilon/doc/articles/emfatic/, 2019.

[20] E. Foundation. Epsilon object language. Available at https://www.eclipse.org/epsilon/doc/eol/, 2019.

[21] E. Foundation. Epsilon. Available at https://www.eclipse.org/epsilon/, 2019.

[22] E. Foundation. EuGENia. Available at https://www.eclipse.org/epsilon/doc/eugenia/, 2019.

[23] E. Foundation. EuGENia GMF tutorial. Available at https://www.eclipse.org/epsilon/doc/articles/eugenia-gmf-tutorial/, 2019.

[24] A. O. Gomes and A. Butterfield. Modelling the haemodialysis machine with circus. In *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 409–424. Springer, 2016.

[25] A. O. Gomes and M. Oliveira. Formal development of a cardiac pacemaker: from specification to code. In *Brazilian Symposium on Formal Methods*, pages 210–225, 2010.

[26] A. O. Gomes and M. V. M. Oliveira. Formal specification of a cardiac pacing system. In *International Symposium on Formal Methods*, pages 692–707, 2009.

[27] D. M. Groenewegen and E. Visser. Integration of data validation and user interface concerns in a dsl for web applications. *Software & Systems Modeling*, 12(1):35–52, 2013.

[28] P. Groot, A. Hommersom, P. Lucas, M. Balser, and J. Schmitt. Experiences in quality checking medical guidelines using formal methods. 2007.

[29] A. Haufler. How to write one of the fastest expression evaluators in java. Available at https://www.javacodegeeks.com/2014/01/how-to-write-one-of-the-fastest-expression-evaluators-in-java.html, 2014.

[30] T. Hoang, C. Snook, L. Ladenberger, and M. Butler. Validating the requirements and design of a hemodialysis machine using iuml-b, bmotion studio, and co-simulation. In *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 360–375, 2016.

[31] E. Jee, S. Wang, J. K. Kim, J. Lee, O. Sokolsky, and I. Lee. A safety-assured development approach for real-time software. In *2010 IEEE 16th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 133–142, 2010.

[32] Z. Jiang, A. Connolly, and R. Mangharam. Using the virtual heart model to validate the mode-switch pacemaker operation. In *2010 Annual International Conference of the IEEE Engineering in Medicine and Biology*, pages 6690–6693, 2010.

[33] Z. Jiang, M. Pajic, A. Connolly, S. Dixit, and R. Mangharam. Real-time heart model for implantable cardiac device validation and verification. In *2010 22nd Euromicro Conference on Real-Time Systems*, pages 239–248, 2010.

[34] Z. Jiang, M. Pajic, and R. Mangharam. Cyber–physical modeling of implantable cardiac medical devices. *Proceedings of the IEEE*, pages 122–137, 2011.

[35] G. Jun, J. Ward, P. Clarkson, et al. Mapping the healthcare process in order to design for patient safety. In *ICED 05: 15th International Conference on Engineering Design: Engineering Design and the Global Economy*, page 2189, 2005.

[36] D. S. Kolovos, A. García-Domínguez, L. M. Rose, and R. F. Paige. Eugenia: towards disciplined and automated development of GMF-based graphical model editors. 16: 229–255, 2017.

[37] D. S. Kolovos, A. García-Domínguez, L. M. Rose, and R. F. Paige. *The Epsilon Book*. 2018.

[38] T. Kühne. Matters of (meta-) modeling. *Software & Systems Modeling*, 5(4):369–385, 2006.

[39] M. Kwiatkowska, H. Lea-Banks, A. Mereacre, and N. Paoletti. Formal modelling and validation of rate-adaptive pacemakers. In *2014 IEEE International Conference on Healthcare Informatics*, pages 23–32, 2014.

[40] B. R. Larson. Formal semantics for the pacemaker system specification. In *ACM SIGAda Ada Letters*, volume 34, pages 47–60, 2014.

[41] J. Leemans and N. Amálio. Modelling a cardiac pacemaker visually and formally. In *2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 257–258, 2012.

[42] J. Ludewig. Models in software engineering–an introduction. *Software and Systems Modeling*, 2(1):5–14, 2003.

[43] C. Mahulea, L. Mahulea, J. García-Soriano, and J. Colom. Petri nets with resources for modeling primary healthcare systems. In *2014 18th International Conference on System Theory, Control and Computing (ICSTCC)*, pages 639–644, 2014.

[44] P. Masci, A. Ayoub, P. Curzon, I. Lee, O. Sokolsky, and H. Thimbleby. Model-based development of the generic pca infusion pump user interface prototype in pvs. In *International Conference on Computer Safety, Reliability, and Security*, pages 228–240, 2013.

[45] D. Méry and N. Singh. Formalization of heart models based on the conduction of electrical impulses and cellular automata. In *Foundations of Health Informatics Engineering and Systems*, pages 140–159. Springer Berlin Heidelberg, 2012.

[46] D. Musat Salvador, J. Pérez, and P. Alarcón. Tutorial de introducción a EMF y GMF. 2019.

[47] Object Management Group (OMG). Common warehouse metamodel. Available at https://www.omg.org/spec/CWM/, 2003.

[48] Object Management Group (OMG). Meta object facility. Available at http://www.omg.org/spec/MOF/2.0/PDF/, 2006.

[49] Object Management Group (OMG). Unified modeling language. Available at https://www.omg.org/spec/UML/, 2017.

[50] M. Pajic, Z. Jiang, I. Lee, O. Sokolsky, and R. Mangharam. From verification to implementation: A model translation tool and a pacemaker case study. In *2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium*, pages 173–184, 2012.

[51] A. Popovic, I. Lukovic, V. Dimitrieski, and V. Djukic. A dsl for modeling application-specific functionalities of business applications. *Computer Languages, Systems & Structures*, 43:69 – 95, 2015.

[52] I. R. Rube. Construccion de editores de modelos con emf. 2013.

[53] I. R. Rube. Desarrollo de metamodelos con emf. 2013.

[54] H. B. Saritas and G. Kardas. A model driven architecture for the development of smart card software. *Computer Languages, Systems & Structures*, 40(2):53 – 72, 2014.

[55] P. Sarkoci. Domination in the families of frank and hamacher t-norms. *Kybernetika*, 41(3):349–360, 2005.

[56] E. Seidewitz. What models mean. *IEEE software*, pages 26–32, 2003.

[57] B. Selic. The pragmatics of model-driven development. *IEEE software*, 20(5):19–25, 2003.

[58] N. K. Singh, H. Wang, M. Lawford, T. S. Maibaum, and A. Wassyng. Stepwise formal modelling and reasoning of insulin infusion pump requirements. In *International Conference on Digital Human Modeling and Applications in Health, Safety, Ergonomics and Risk Management*, pages 387–398, 2015.

[59] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework 2.0.* Addison-Wesley Professional, 2nd edition, 2009.

[60] T.Chen, M. Diciolla, M. Kwiatkowska, and A. Mereacre. Quantitative verification of implantable cardiac pacemakers over hybrid heart models. *Information and Computation*, 236(C):87–101, 2014.

[61] M. Ten Teije, A.and Marcos, M. Balser, J. van Croonenborg, C. Duelli, F. van Harmelen, P. Lucas, S. Miksch, W. Reif, K. Rosenbrand, et al. Improving medical protocols by formal methods. *Artificial intelligence in medicine*, 36(3):193–209, 2006.

[62] M. Voelter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L.Kats, E. Visser, and G. Guido. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages.* dslbook.org, 2013.