Universidad de Valladolid

ESCUELA DE INGENIERÍA INFORMÁTICA

GRADO EN INGENIERÍA INFORMÁTICA
Mención en Ingeniería del Software

Desarrollo de un plugin para SemanticMerge para facilitar la integración de versiones de archivos Vensim

---

Alumno: Pablo Martínez López

Tutora: Yania Crespo González-Carvajal

A mis padres

Agradecimientos

A mi familia, por todo el apoyo durante este complicado año de mucho trabajo.

A mis amigos y compañeros de carrera, por animarme en todo momento, por ayudarme a desconectar del estrés y por motivarme a aprender y a mejorar no sólo este último año, sino durante toda la carrera.

A mi pareja Laura, por haberme apoyado constantemente durante todo este año y por ayudarme a manejar mis nervios y estrés.

A mi tutora Yania, por su labor como profesora y coordinadora de este Trabajo de Fin de Grado, que sin duda ha sido clave para mantener un trabajo constante y de calidad a lo largo de todos los meses del proyecto.

A los miembros de Plastic SCM, concretamente a Pablo Santos, Violeta Sánchez y Borja Ruiz y con especial gratitud a Míryam Gómez; por toda la ayuda en la resolución de las dudas y problemas relacionados con las herramientas de SemanticMerge y gmaster.

Al proyecto LOCOMOTION y su financiación de la beca asociada a este proyecto, y en especial a Ignacio de Blas Sanz, David González Antelo e Íñigo Capellán Pérez, miembros del GIR GEEDS, por su participación en las pruebas de aceptación de usuarios.

Resumen

`Vensim` es un lenguaje de modelado de dinámica de sistemas. Incluye una parte de definición de ecuaciones y otra parte visual. Trabajar en equipo, de forma coordinada, con control e integración de versiones, en Vensim es un gran reto. Por su parte, `SemanticMerge` es una herramienta para visualizar diferencias y para mezclar versiones (merge) con ayuda para la resolución de conflictos.

El objetivo de este Trabajo de Fin de Grado es desarrollar un *parser* externo de Vensim para `SemanticMerge`, que a su vez será utilizado por la herramienta de control de versiones `gmaster`. El *parser* funciona leyendo un archivo de entrada y generando un archivo `YAML` con la información necesaria para que `SemanticMerge` sea capaz de leerlo y reconstruir el archivo, obteniendo toda la información semántica y sintáctica del archivo en dicho proceso. Este proceso se realiza en ambas versiones de un archivo cuando se lleva a cabo una fusión o merge, para posteriormente resolver los conflictos y realizar el *merge* correctamente.

El trabajo ha sido desarrollado utilizando `ANTLR4` para el desarrollo de la gramática así como `Java` como lenguaje de programación principal. Para el desarrollo del proyecto se adaptó `SCRUM` como marco de trabajo ágil.

Este proyecto forma parte del proyecto europeo H2020 `LOCOMOTION` que tiene como objetivo el desarrollo de un modelo de evaluación integrado (*Integrated Assesment Model*, IAM), un modelo complejo basado en dinámica de sistemas, que permite la simulación de escenarios diferentes que afectan a largo plazo a diferentes indicadores, lo que permite determinar los más favorables para un futuro sostenible. El IAM desarrollado en LOCOMOTION se programa en Vensim de forma coordinada, participando 30 programadores de 13 instituciones europeas diferentes.

Abstract

Vensim is a system dynamics modeling language. It includes two main parts: an equation definition section and a sketch definition section. Coordinated teamwork using integration and version control tools when programming in Vensim is a great challenge. On the other hand, SemanticMerge is a standalone application that allows to visualize changes between different versions of a file and then merge these versions, helping to solve the conflicts that arise.

This Final Degree Project aims to develop an external parser for SemanticMerge that allows parsing Vensim files. This parser will also be used by the version control tool gmaster.

The parser reads an input file and generates a tree descriptor YAML file containing the information needed for SemanticMerge to reconstruct the file obtaining all the semantic and syntactic information of the file in the process. This process is performed in both versions of the file when a merge operation is achieved, allowing conflict resolution, and enabling a successful merge.

ANTLR4 is used for the development of the grammar as well as Java as the main programming language. SCRUM was applied as an Agile framework adapted for the development of the project.

This project is part of the European H2020 project LOCOMOTION. This project aims to develop an Integrated Assessment Model (IAM), a complex model of the World and regions, based on system dynamics, that allows the simulation of different scenarios that long term affect different indicators and to determine the more favorable ones for a sustainable future. The IAM developed in LOCOMOTION is programmed in Vensim in a coordinated way by 30 programmers from 13 different European institutions.

Índice general

Lista de Figuras

Lista de Tablas

Capítulo 1

Introducción

1.1. Contexto

Este Trabajo de Fin de Grado ha sido realizado como una beca, financiada por la Unión Europea, adscrita al proyecto de investigación del programa marco Horizonte 2020 (H2020): *"Low-carbon society: an enhanced modelling tool for the transition to sustainability"* o `LOCOMOTION` [16]. En este proyecto participan 13 instituciones europeas de 10 países diferentes, coordinado desde la Universidad de Valladolid por el GIR (Grupo de Investigación Reconocido) en Energía, Economía y Dinámica de Sistemas (`GEEDS`). Con este proyecto de investigación colaboran también algunos profesores del Departamento de Informática de esta Universidad.

Debido a la aceleración del cambio climático y todas las consecuencias que ello conlleva, en Europa se acordó tratar de diseñar un conjunto de sistemas de modelado o `IAMs` (*Integrated Assement Models*) capaz de proporcionar una visión real del futuro del planeta a aquellas instituciones que tomasen medidas relacionadas con el devenir del planeta acorde al cambio climático. Por ello se creó el proyecto `MEDEAS` [17] (*Modelling the Energy Development under Environmental And Socioeconomic constraints*), cuyo objetivo era crear un sistema computacional capaz de modelar la energía en Europa, a través de constantes físicas y sociales. Esto permitiría transicionar a un modelo energético más sostenible en un futuro. Este proyecto culminó en un modelo intermedio y ha evolucionado en el último año a su continuación natural, con el ya nombrado proyecto `LOCOMOTION`.

Tanto el anterior proyecto `MEDEAS` como el actual `LOCOMOTION` utilizan el software `Vensim` de desarrollo de modelos de simulación basados en dinámica de sistemas. Actualmente, desde el Departamento de Informática de la UVa se desarrollan varios sub-proyectos como partes del proyecto `LOCOMOTION`, tales como este Trabajo de Fin de Grado u otros destinados, por ejemplo, a la creación de un juego educativo de concienciación para adolescentes sobre el cambio climático o plugins para `SonarQube`, herramienta que realiza controles de calidad sobre el código `Vensim`.

Este Trabajo de Fin de Grado se centra en cómo mejorar el control de versiones, y la realización de *merges* o fusiones entre diferentes versiones de archivos `Vensim`.

1.2. Motivación

Los archivos con extensión `.mdl` de `Vensim` pueden dividirse en dos grandes partes, una sección de definición de ecuaciones y demás tipos de datos básicos, y una sección de definición de información de cada una de las vistas (diagramas de modelado de dinámica de sistemas) así como las definiciones de gráficos para la visualización de los resultados de las simulaciones.

En un escenario de control de versiones de archivos Vensim, al modificar un archivo `.mdl` en un proyecto, pequeños cambios como movimientos de variables en una vista generan problemas múltiples cambios en el fichero, muy desproporcionados respecto al tamaño de las modificaciones lo que introduce una complejidad enorme a la hora de intentar fusionar o *mergear* los ficheros con los que un equipo ha estado trabajando.

`SemanticMerge` [43] es un software capaz de tratar de forma sencilla y visual alteraciones de código entre ficheros, tales como movimientos, adiciones o eliminaciones de secciones de código o renombrados. Este software no está enfocado a un paradigma concreto de lenguajes de programación sino a una estructura de contenedores y nodos, con la cual es posible tratar los archivos de `Vensim`. Se ha propuesto crear un plugin, herramienta o *parser* externo para la herramienta de `SemanticMerge` que permita solucionar dichos problemas de una forma sencilla.

Se parte inicialmente de una gramática de Vensim encargada de procesar dichos archivos `.mdl` pero que sólo trataba la primera parte de los mismos (la sección relacionada con la definición de ecuaciones). Por tanto, se hace necesario ampliar la gramática para poder procesar archivos `.mdl` en su totalidad, con una complejidad mucho mayor, con múltiples vistas y definiciones de gráficos.

1.3. Introducción a Vensim

Vensim [56] es un software de simulación basado en dinámica de sistemas que permite realizar grandes modelos y comprobar su rendimiento y progreso a lo largo de un periodo de tiempo. Dicho programa es un software gráfico que nos permite representar nuestro sistema mediante diagramas en las que se relacionan variables, y otros elementos, con flechas que representan flujos, tal y como puede verse en la Figura 1.1, la cual muestra un sistema muy simple modelado con Vensim. Pese a la simplicidad del sistema ilustrado en la figura, en Vensim se pueden representar sistemas muy complejos a través de varias vistas que se hacen referencia entre sí. También es posible utilizar gráficos para ofrecer información de la simulación de forma más visual.

Los modeladores suelen trabajar de forma gráfica con los archivos que representan los modelos. La extensión más común de Vensim, `.mdl`, permite exportar el modelo a un archivo

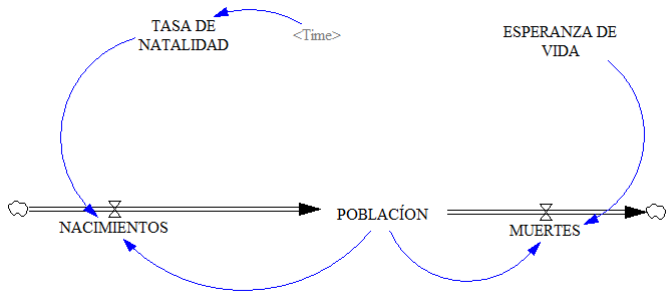

Figura 1.1: Representación de un sistema pequeño en Vensim.

completamente textual, transformando toda la parte gráfica en datos numéricos y metadatos. También se pueden guardar los archivos en formato binario, con sus respectivas extensiones: `.vpm` y `.vpf`.

Un proyecto en Vensim se corresponde con un modelo, el cual reside en un único archivo que se compone por una serie de elementos relacionados entre sí, los cuales trataremos más adelante. Además, un modelo puede estar compuesto por más de una vista, estando los elementos de todas las vistas relacionados. Las vistas suelen utilizarse para desglosar un modelo de grandes dimensiones, manteniendo una correlación entre las vistas. Es posible tratar modelos separados en cada vista, aunque es muy poco recomendable.

Al iniciar un nuevo modelo en Vensim, se lanza una ventana emergente donde configuraremos los parámetros iniciales de nuestro sistema, así como el ritmo al que evoluciona. Esto puede apreciarse en la Figura 1.2. Estas variables serán globales, por lo que serán comunes a todas las vistas. Podrán ser usadas en las definiciones de ecuaciones o en ciertos casos como variables sombra o *shadow variables*, las cuales serán explicadas en secciones posteriores de la memoria.

1.3.1. Tipos básicos en Vensim

Vensim presenta una serie de tipos básicos los cuales es importante conocer puesto que se nombrarán en múltiples ocasiones a lo largo de esta memoria [55]:

- **Variables auxiliares**. Cambian a lo largo del tiempo y suelen incluir otras variables en sus ecuaciones. Suelen ser el tipo de dato más común en la mayoría de modelos.
- **Variables de datos**. Cambian con el tiempo pero no dependen de otras variables del modelo. Se caracterizan por definir sus ecuaciones con el símbolo :=.
- **Variables de nivel**. Variables con un valor inicial establecido, que actualizan su valor a lo largo de la simulación a partir de su valor en la iteración anterior y su ecuación.

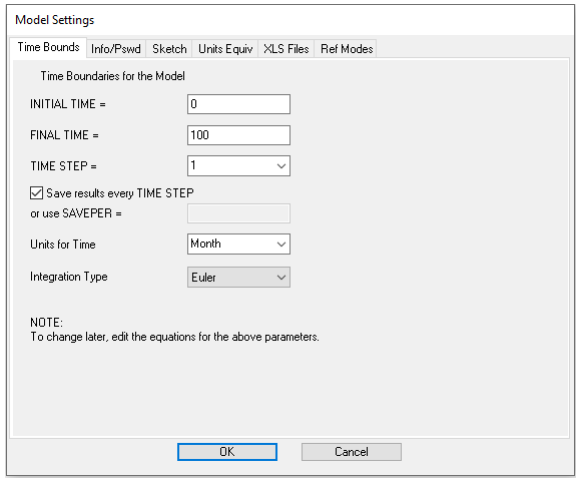

Figura 1.2: Elección de parámetros para un nuevo modelo.

- **Constantes**. Valores estáticos que no cambian con el tiempo. Sin embargo, es posible modificarlos a medida que avanza la simulación.

- **Constantes inmutables**. Valores estáticos que no cambian con el tiempo y que no pueden cambiarse durante el periodo de simulación. Su asignación es especial puesto que utilizan el símbolo ==.

- **Lookups o tablas**. Funciones no lineares definidas a través de parámetros numéricos: valores en el eje de abscisas con sus correspondientes valores en el eje de ordenadas. Cada tupla de datos se representa entre corchetes [ ] y el total del *lookup* se encuentra entre paréntesis.

- **Subscripts**. Símbolos especiales los cuales tienen definido un rango o un conjunto de valores que pueden tomar. Una variable puede estar asociada a un subscript o a un valor del subscript. En el caso de estar asociada a un subscript, cuando se referencia la variable se referencian todos los valores de dicho subscript. Estos valores suelen estar representados cada uno por su propia ecuación o conjunto de ecuaciones. Una variable puede estar relacionada hasta un máximo de ocho subscripts.

- **Elemento de un subscript**. Símbolo asociado a un subscript y es uno de los valores que lo identifican.

- **Reality checks**. Ecuaciones lógicas que comprueban que una condición se cumple. Es comparable en cierto modo a los asertos de los lenguajes de programación convencionales.

- **Macros**. Operadores especiales que el usuario puede definir para representar un concepto bien complicado o bien largo o que se repita mucho, y que es sustituido por un nombre que toma el valor de dicha expresión en todo el programa.

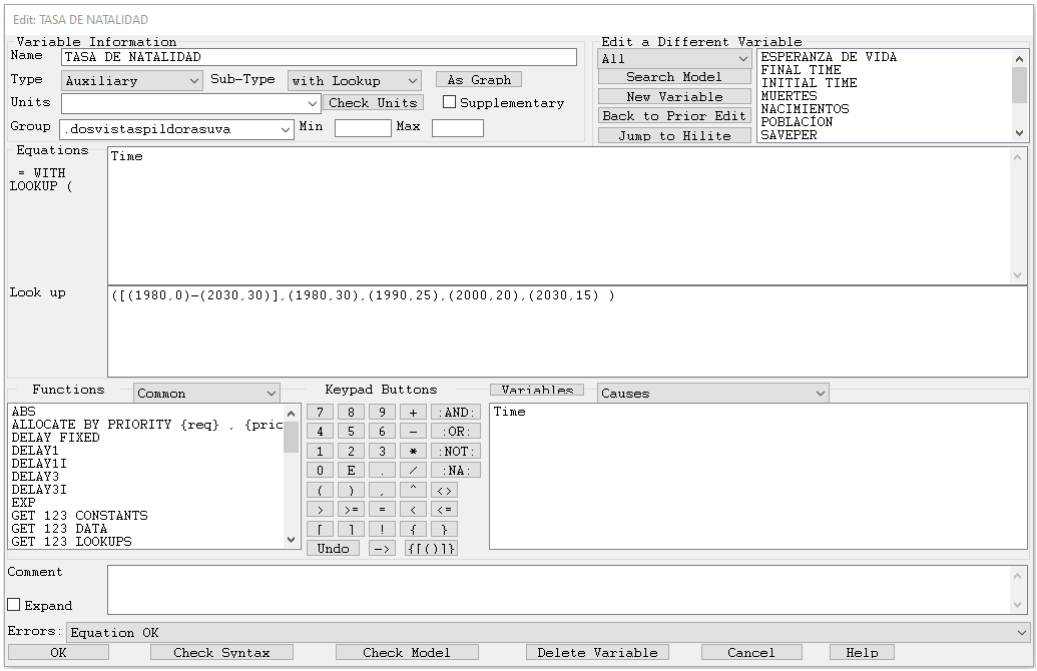

Figura 1.3: Menú de opciones de una variable.

En la interfaz gráfica de Vensim, al asociar una ecuación a una variable se despliega el menú que se muestra en la Figura 1.3, el cual resume muchos de los conceptos explicados en esta sección.

Cada ecuación tiene ciertos apartados, tal y como se puede apreciar en la Figura 1.3. En primer lugar, se encuentra el nombre de la ecuación, el cual es utilizado para referenciar a la misma. Tras ello, encontramos las unidades en las que se mide la ecuación. También podemos definir otros parámetros como valores máximo y mínimo así como el alcance de los incrementos. Posteriormente, se encuentra el campo de definición de la ecuación, donde se describe la fórmula de la misma. En el caso de las tablas o *lookups* se utiliza también para introducir los valores por pares de la misma. Finalmente, se encuentra el campo de comentario, el cual no afecta a la ecuación como tal, pero puede servir para clarificar el propósito de la misma u otros aspectos, como un mecanismo de autodocumentación.

1.4. Introducción al control de versiones

El control de versiones, y su herramienta más conocida y utilizada `git` [58], es una tecnología que como su propio nombre indica, está diseñada para permitir el trabajo simultáneo sobre un mismo archivo entre miembros de un equipo y/o trabajar con diferentes versiones de un mismo archivo. Los proyectos que utilizan esta herramienta se alojan en repositorios remotos alojados en la nube, y cada miembro del proyecto puede descargarse una copia local independiente en su ordenador.

Cada actualización de un archivo se denomina *commit*, el cual es la instancia en el tiempo del trabajo realizado hasta la fecha; y la unión sucesiva de dichas actualizaciones o *commits* forman lo que se denomina una rama. De la misma forma que en un árbol, de una rama pueden salir varias ramas, las cuales se usan para dividir y aislar zonas de trabajo, normalmente entre miembros de un equipo. Al contrario que en los árboles, las ramas pueden fusionarse de nuevo con su rama padre. Este proceso se denomina *merge*, y puede llegar a ser bastante complejo.

Esto puede ser ilustrado con el ejemplo en el que de una rama padre parten dos ramas hijas A y B, dos áreas de trabajo para dos equipos diferentes. Cuando los equipos completen sus tareas, ambos habrán partido de un conjunto de archivos inicial, los cuales habrán modificado de forma distinta. Realizando un *merge* con la rama padre, se fusionará el trabajo realizado de los dos equipos de trabajo en una única versión alojada en la rama padre.

Sin embargo, este proceso es idílico y sólo sucede en proyectos simples. Es muy corriente que dos equipos de trabajo tengan que trabajar sobre las mismas partes de un proyecto, y que al fusionar las dos partes de trabajo realizado, estas se superpongan. Esto es lo que se conoce como un conflicto. En los conflictos, se debe escoger qué parte del trabajo se desea conservar: lo que proviene de la rama A, de la rama B o incluso de ambas. Si bien resolver un conflicto puede ser algo trivial como leer unas pocas líneas de código, puede llegar a ser un proceso muy complicado y tedioso si se modifican varios archivos extensos en profundidad en ambas ramas.

Debido a que gestionar una gran cantidad de ramas y *commits* puede ser un proceso tedioso (más aún si sus nombres no son significativos), nació la metodología de trabajo *git-flow*. Esta metodología se estructura en tres tipos de ramas:

- **master/main**. Esta rama sirve de vitrina al público y sólo contendrá trabajo funcional. Cada actualización a `master` se denomina *release* y suele ir seguida del número de versión.
- **develop**. Esta rama es la rama principal del proyecto y donde salen y vuelven el tercer tipo de ramas.
- **ramas secundarias**. El nombre de estas ramas está compuesto por dos partes. La primera, es un identificador del trabajo realizado en la rama, como puede ser *feature* o *bug*. La segunda parte es una pequeña descripción del trabajo realizado en dicha rama. De estas ramas pueden salir a su vez ramas de esta categoría.

Esto puede se aprecia gráficamente en la Figura 1.4.

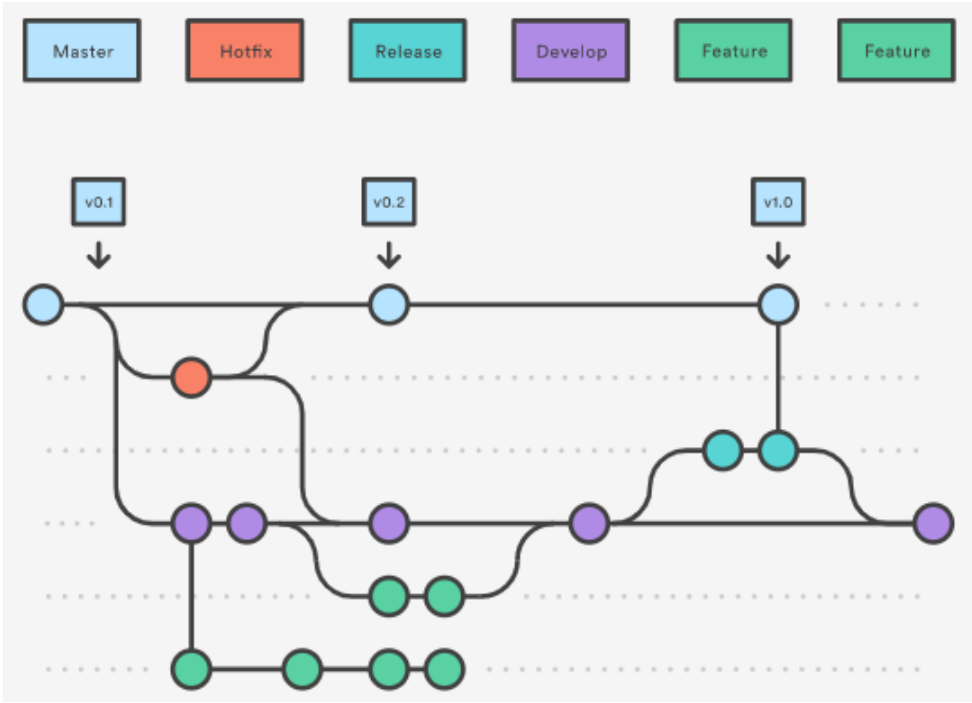

Figura 1.4: Esquema de metodología git-flow. Tomada de [6]

Esta metodología también aboga por utilizar *commits* con nombres significativos y que separen el trabajo realizado en bloques de un tamaño moderado para a la hora de solventar un problema, no desperdiciar mucho tiempo buscando en cual de ellos radica el mismo.

1.5. Introducción a PlasticSCM

`PlasticSCM` [47] es una herramienta de control de versiones desarrollada por la empresa *Códice Software*, fundada y con sede en Valladolid, la cual es cada vez más conocida internacionalmente. Esta herramienta, aparte de proporcionar las características básicas del control de versiones, proporciona también otras propias como la capacidad de trabajar con archivos de muy grandes dimensiones, trabajar con un gran número de ramas, aportar una interfaz gráfica propia para hacer las operaciones más amenas y entendibles para los usuarios e integrar la diferenciación semántica entre archivos a través de `SemanticMerge` (ver Sección 1.7, entre otras.

Al abrir dicho programa se accede directamente a un espacio de trabajo o *workspace*. En dicho *workspace* se puede navegar a través de los archivos de nuestro ordenador, como un sistema de ficheros convencional. Sin embargo, la herramienta es capaz de detectar qué directorios están asociados a un repositorio de control de versiones. Esto se puede apreciar en la Figura 1.5.

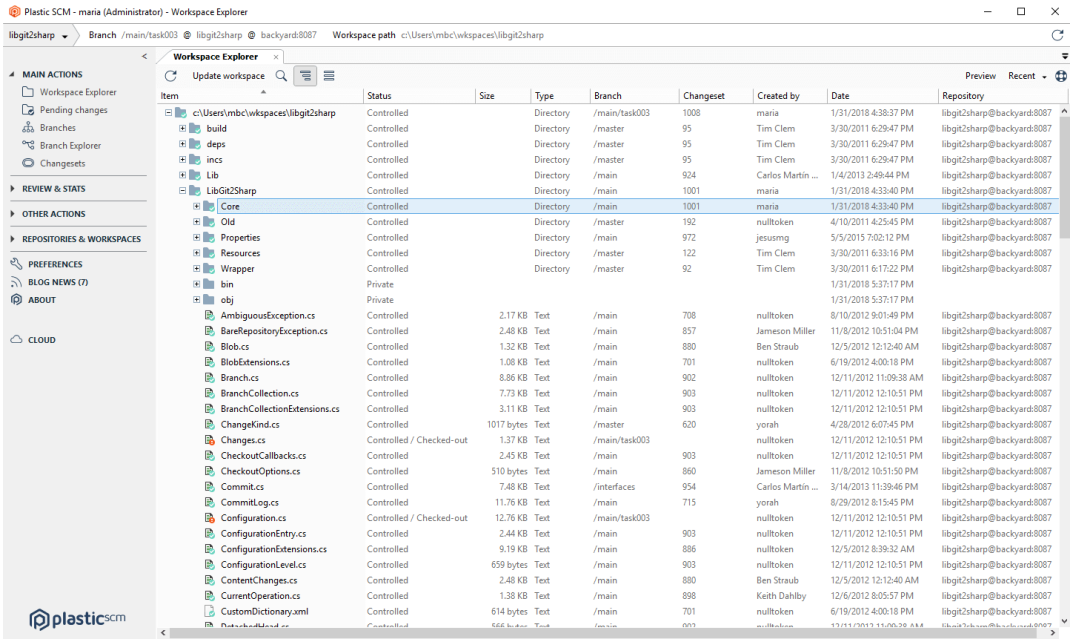

Figura 1.5: Vista de espacio de trabajo principal en PlasticSCM

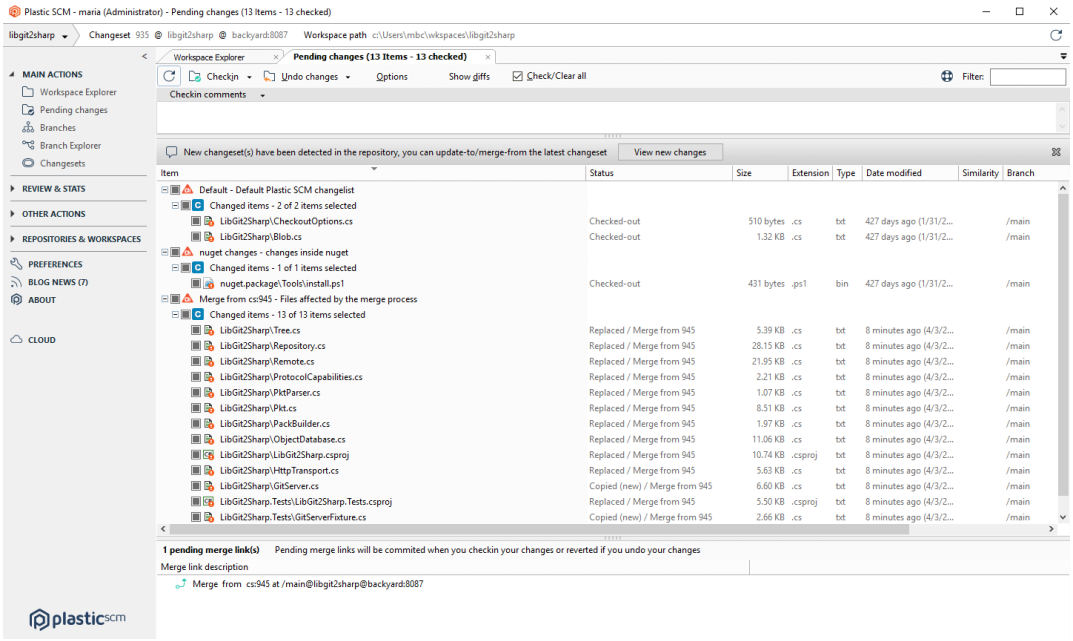

Figura 1.6: Vista de Cambios Pendientes en PlasticSCM

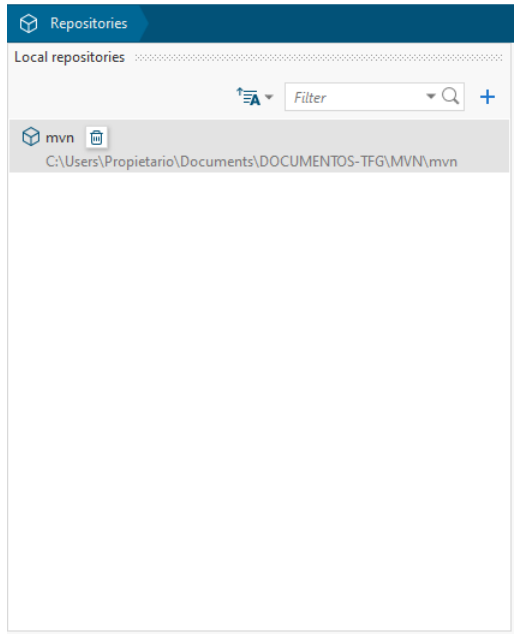

Figura 1.7: Añadir repositorio en gMaster.

En el panel izquierdo de la Figura 1.5 podemos apreciar todas las opciones de las que dispone la herramienta. Sin embargo, como en la sección inmediatamente siguiente a la actual se trata la herramienta `gMaster` y todas las características que ofrece sobre control de ramas (con las cuales también cuenta `PlasticSCM`). Nos centraremos en la característica de "Cambios pendientes" o "*Pending changes*". Esta vista es similar a la de workspace en el sentido en que expone un directorio con sus ficheros. Sin embargo, sólo muestra los archivos modificados desde el último *commit*. La herramienta incorpora la característica de crear "listas de agrupamientos", lo cual permite agrupar los archivos modificados en subgrupos de archivos que hayan tenido modificaciones similares. Esto puede ser apreciado en la Figura 1.6.

1.6. Introducción a gMaster

`GMaster` [37] es una herramienta visual de control de versiones basada en `git` desarrollada por Códice Software, la cual hace uso de la herramienta `SemanticMerge` (ver Sección 1.7) para procesar sus archivos. Por lo tanto, aparte de proporcionar al usuario una interfaz gráfica clara y limpia para comprender mejor los cambios de versiones, proporciona información semántica de los archivos que se están comparando a través de `SemanticMerge` la cual se utiliza para comparar y realizar el *merge* entre varios archivos.

Cuando abrimos la herramienta, se nos dará la opción de añadir un repositorio inicial, con el cual podremos empezar a trabajar, tal y como se puede apreciar en la Figura 1.7. Sin embargo, se pueden mantener varios repositorios activos, tanto remotos como locales.

Tras esto, accederemos a la vista principal, donde tendremos dos panales principales. El primero, situado en la parte superior de la pantalla, consiste en un mapa en el tiempo de las ramas de nuestro proyecto, el cual indica las ramas activas en este momento así como los *commits* más recientes con su correspondiente fecha. Dicho mapa también nos indica en qué rama y *commit* nos encontramos actualmente. El segundo panel, situado inmediatamente por debajo del anterior, muestra los dos archivos a comparar, permitiendo desplazarnos entre cada una de las diferencias encontradas entre los ficheros. La herramienta permite ver las diferencias tanto en formato de texto plano, como harían las herramientas de control de versiones tradicionales, o utilizar también las diferencias semánticas aprovechando las capacidades de la herramienta `SemanticMerge`. Esto puede ser apreciado en la Figura 1.8.

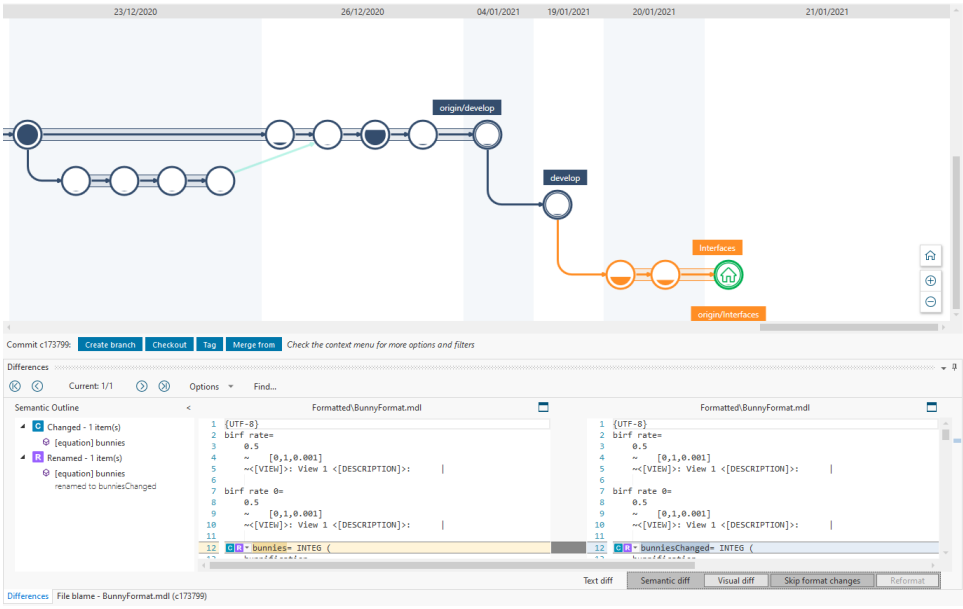

Figura 1.8: Vista principal de gMaster.

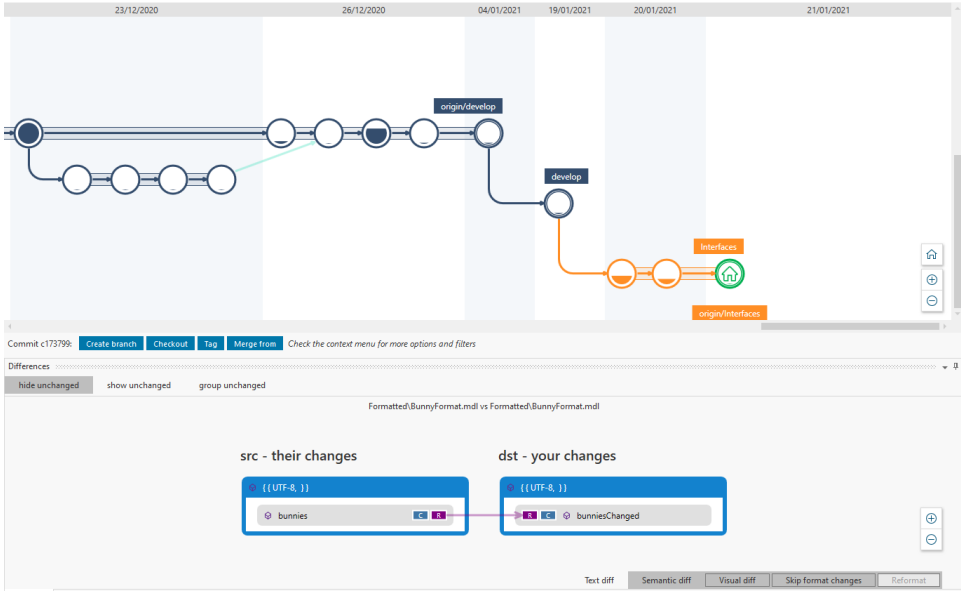

Figura 1.9: Vista principal en formato visual de gMaster.

Los cambios pueden observados de una manera más visual al pulsar el botón `Visual diff` situado en la parte inferior derecha de la figura 1.8. Con ello, los cambios se mostrarán en forma de imágenes, como puede verse en la Figura 1.9.

Al igual que en `SemanticMerge`, `gMaster` caracteriza los cambios en cinco tipos principales, estos son: Cambio, Adición, Movimiento, Renombre y Eliminación. Cada uno de estos posee un símbolo consistente en un rectángulo de color con una letra mayúscula que lo identifica y que permiten identificar los cambios de manera más visual, como puede verse en la figura anterior 1.9.

Por último, en la parte derecha de la pantalla podremos alternar entre varios archivos que queramos comparar entre sus versiones, en los casos en los que se hayan modificado varios archivos en un mismo *commit*. Esto se aprecia en la Figura, 1.10.

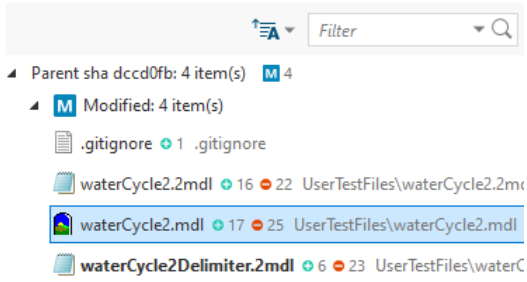

Figura 1.10: Archivos a escoger en gMaster.

1.7. Introducción a SemanticMerge

`SemanticMerge` es una herramienta desarrollada por la empresa *Códice Software*, se puede utilizar como herramienta independiente (*standalone*) o integrada en PlasticSCM o gMaster, como se ha visto en las dos secciones anteriores. Esta herramienta destaca por la gran facilidad que aporta al usuario para fusionar archivos o realizar *merges* que en un principio pudieran resultar muy complicados de resolver manualmente o mediante la ayuda de una herramienta de control de versiones basada en texto.

Esta herramienta se diferencia del resto porque no intenta tratar bloques de texto planos, sino que intenta comprender la estructura del lenguaje para simplificar la tarea. Esto lo consigue añadiendo información semántica sobre el lenguaje en el que está escrito el archivo, mediante un *parser* que analiza la gramática del lenguaje de los archivos y obtiene una representación intermedia de los mismos.

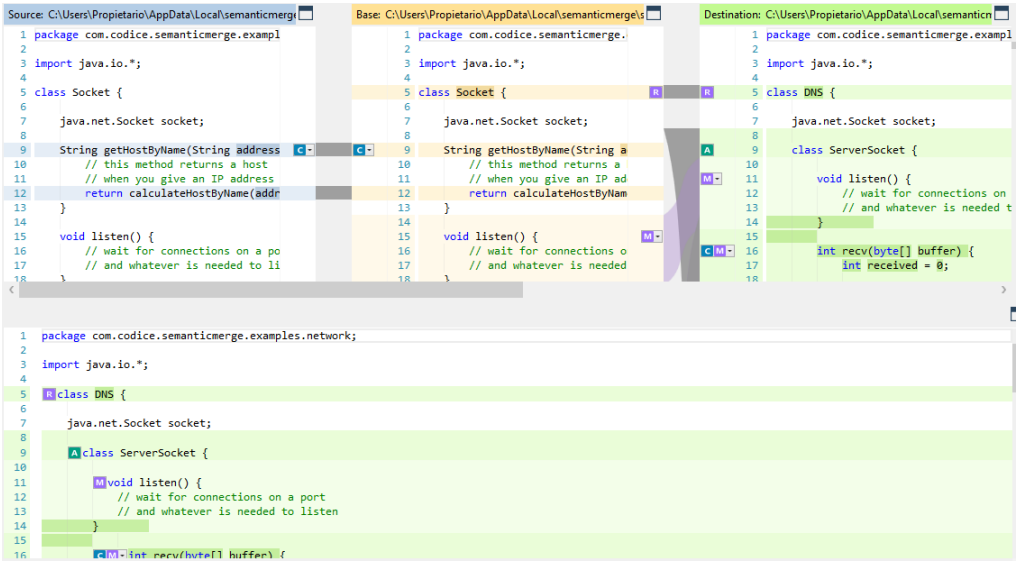

Figura 1.11: Interfaz gráfica de SemanticMerge. Tomada de [45]

El proceso por el cual los archivos se comparan viene dado por dos herramientas que la empresa desarrolló anteriormente a `SemanticMerge`: `XDiff` y `XMerge` [46]. La primera de las herramientas, `XDiff`, es capaz de comprobar si un fragmento de código dentro de un archivo ha sido desplazado mediante un algoritmo que comprueba qué partes del código han sido modificadas. Por otro lado, `XMerge` permite detectar el código que partiendo de un archivo inicial, ha sido modificado paralelamente por dos desarrolladores diferentes, con un algoritmo similar al anterior. Esto incluye también movimientos de código.

La unión de estas herramientas proporciona una metodología de trabajo al estilo *"Divide y vencerás"*, mediante el cual se intentan abordar los conflictos separándolos en módulos. Por ejemplo, pongamos que partiendo de un archivo base, dos desarrolladores modifican

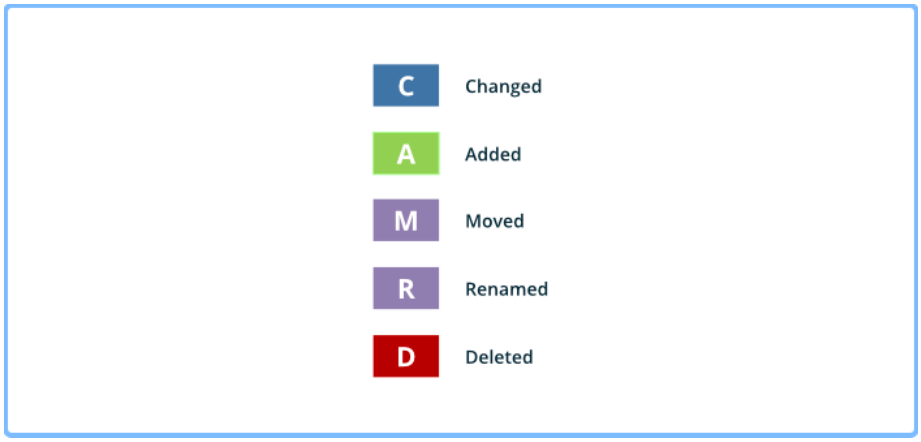

Figura 1.12: Significado de las letras en SemanticMerge. Tomado de [43]

un método. El primer desarrollador mueve el método de sitio desplazándolo líneas arriba. Por su parte, el segundo desarrollador desplaza el método líneas abajo y además lo modifica. Apoyándose en las dos herramientas, `SemanticMerge` resuelve primero el problema del desplazamiento, también llamado "movimiento divergente" o *"divergent move"*. Para ello, comprueba la estructura de los dos ficheros y localiza el método en ambos. Tras ello, realiza el primer *"sub-merge"* entre los dos archivos. Una vez solucionado ese problema, comprueba que uno de los dos métodos ha sido modificado, resuelve el conflicto y realiza un segundo *"sub-merge"*, resolviendo completamente el conflicto [44, 45].

El funcionamiento correcto de este proceso se debe a que `SemanticMerge` utiliza un método de *merge* de tres partes o *three-way merge* [41]. Esto consiste, como se muestra en la Figura 1.11, en que, además de utilizar los dos archivos a comparar en el *merge*, se utiliza un archivo inicial común a los dos otros archivos (*source* y *destination*) para ser capaz de entender cómo se ha llegado a ciertas situaciones. Por ejemplo, asegurarse de si una línea ha sido incluida por un programador, o por el contrario si ya estaba y ha sido borrada por otro.

La interfaz gráfica de la aplicación nos muestra los cambios ocurridos en los archivos respecto al original o base de forma simple en un menú, utilizando la izquierda para el archivo *source* y la derecha para el archivo *destination*.

De izquierda a derecha en la parte superior de la figura podemos ver tres paneles. El panel azul representa el fichero fuente o *source*, con una serie de cambios respecto al fichero del que se parte, el cual está representado en color verde en la parte derecha, *destination*. En color amarillo en medio se encuentra un fichero *base*, del cual han partido ambos ficheros. Este *merge* con tres archivos [41] utilizando un archivo ancestro de los archivos *source* y *destination* ayuda al usuario a elegir correctamente las partes que quiere conservar a la hora de resolver un conflicto.

Por ejemplo, si sólo se utilizasen los archivos *source* y *destination* y se encontrase una línea que sólo aparece en uno de los dos archivos, podría generar problemas pensando si hay

que mantener dicha línea o no. Al utilizar un archivo *base* ancestro común a los dos, se puede ver claramente si dicha línea existía anteriormente y el problema desaparece. En la parte inferior de la figura se muestra el fichero final, tras haber aplicado los cambios ocurridos en los ficheros *source* y *destination*, y por ende, haber realizado correctamente el *merge*.

En el caso de que cualquiera de los tres archivos no pueda ser parseado correctamente por la aplicación, el merge no podrá ser realizado. En ese caso, `SemanticMerge` lanzará un mecanismo de *fallback* donde se informará al usuario de los errores que han impedido la correcta ejecución del software [44]. Pese a ello, es posible completar su resolución mediante la herramienta básica de texto plano.

En SemanticMerge se definen cinco tipos de cambios: modificación, adición, desplazamiento, renombrado y borrado. Estos tipos de modificaciones se muestran mediante marcas con unas letras y colores en la interfaz gráfica tal y como puede verse en la Figura 1.11. En la Figura 1.12 se muestra un resumen de los colores y letras utilizados para marcar cada tipo de modificación.

`SemanticMerge` permite visualizar los cambios de tres maneras diferentes. La primera sería una manera en formato textual, al igual que lo hacen las herramientas tradicionales de control de versiones, ver Figura 1.13. La segunda forma incluye las características que hacen especial a la herramienta, y es que a través de un análisis al archivo, expone además sus diferencias semánticas, ver Figura 1.14. La tercera forma utiliza la herramienta de diferenciación semántica también, pero en vez de utilizar texto, utiliza una versión mucho más gráfica donde relaciona los cambios, Figura 1.15.

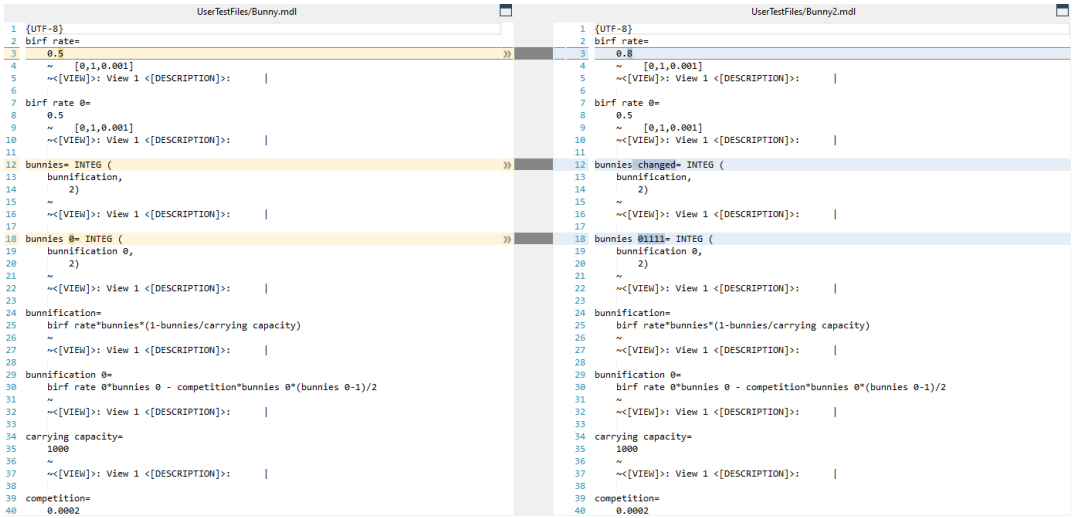

Figura 1.13: Diferenciación textual entre archivos de SemanticMerge

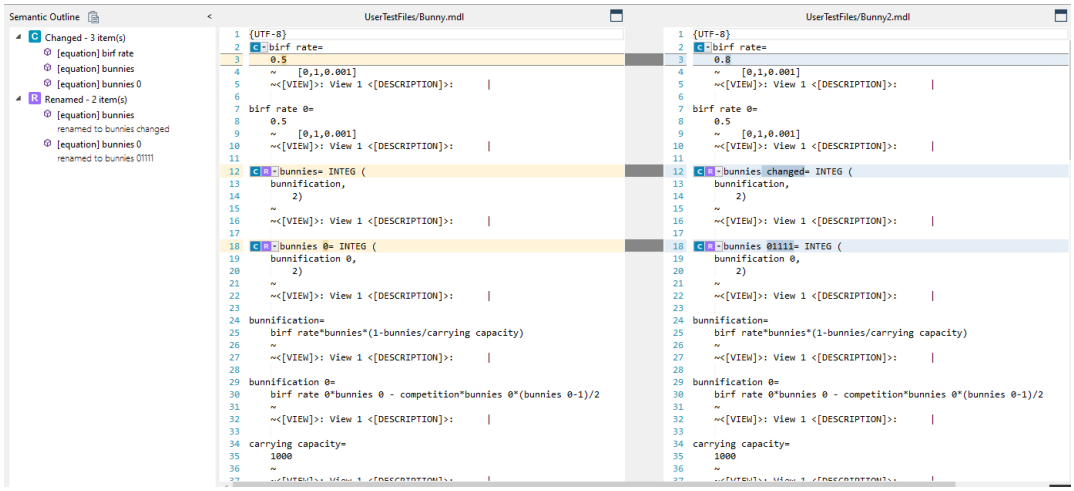

Figura 1.14: Diferenciación semántica entre archivos de SemanticMerge

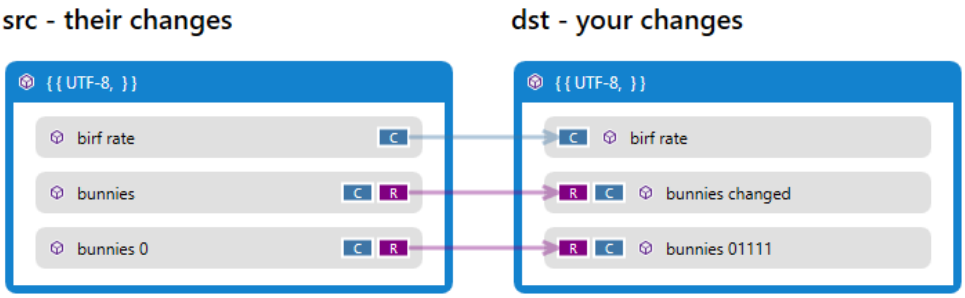

Figura 1.15: Diferenciación visual entre archivos de SemanticMerge

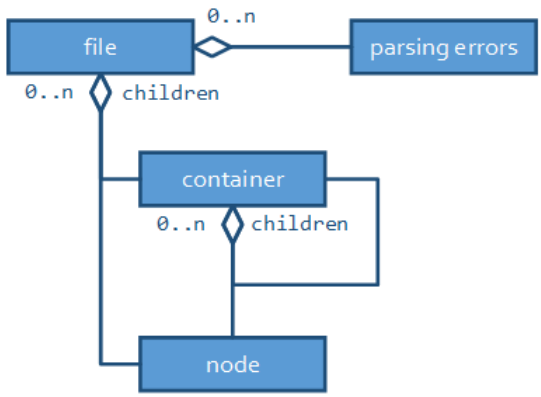

Figura 1.16: Diagrama de clases de la estructura de parseo de SemanticMerge. Tomado de [40]

1.8. Objetivos

El objetivo de este proyecto es crear una herramienta externa (plugin) para la herramienta `SemanticMerge`, y con ello ser capaz de realizar comparaciones, encontrar diferencias entre archivos `.mdl` de `Vensim` y ser capaz de fusionarlos en un único fichero (operaciones conocidas en la terminología del control de versiones como *diff and merge*). Dicha herramienta deberá ser capaz de tratar de forma completa archivos `.mdl` por lo que se definen los siguientes subobjetivos:

- construir una gramática capaz de reconocer y manejar archivos .mdl complejos con múltiples vistas y/o componentes gráficos. Dicha gramática será utilizada para leer los archivos a procesar y en el proceso, analizar la estructura y la semántica del mismo, con el fin de identificar cada una de las partes del archivo, proporcionando a los usuarios una mayor claridad sobre los cambios realizados entre dos versiones de un mismo archivo `Vensim`.
- desarrollar dos herramientas de apoyo al plugin cuyo objetivo será modificar los archivos `Vensim` a ser comparados para que su semántica sea reconocida con mayor facilidad a la hora de utilizar la herramienta, añadiendo partes adicionales que aportan más información semántica; y tras realizar la comparación/fusión con éxito, eliminar dichas partes para obtener un archivo .mdl con el estilo habitual.

1.9. Estructura de la memoria

Este documento se estructura de la siguiente forma:

Capítulo 2 Requisitos y planificación: Se describe la adaptación del marco de trabajo ágil `SCRUM` al contexto de este proyecto. Se definen los *backlogs* inicial y final así como la planificación de este trabajo y sus riesgos asociados. Por último, incluye los presupuestos asociados al mismo.

Capítulo 3 Análisis: Describe el proceso seguido en el planteamiento de los distintos problemas de los que consta este trabajo, así como explica distintos conceptos teóricos asociados al mismo.

Capítulo 4 Tecnologías utilizadas: Describe las tecnologías utilizadas, tanto para la gestión del proyecto como para el desarrollo del mismo.

Capítulo 5 Diseño: Describe el diseño de la solución basado en el análisis del proyecto. Incluye referencias al código realizado.

Capítulo 6 Implementación y pruebas: Describe el proceso de *testing* que ha sufrido el proyecto.

Capítulo 7 Seguimiento del proyecto: Describe el desarrollo del proyecto, dividido en *sprints* de dos semanas siguiendo un desarrollo ágil basado en una adaptación de `SCRUM`.

Capítulo 8 Conclusiones:

Anexo A Manuales: Incluye manuales de mantenimiento, de instalación, despliegue, y de uso.

Anexo B Resumen de enlaces adicionales: Incluye enlaces de interés sobre el proyecto, como el repositorio de código ...

Capítulo 2

Requisitos y Planificación

2.1. SCRUM y su adaptación al proyecto

Este proyecto se ha realizado mediante un desarrollo ágil basado en una adaptación de SCRUM [22]. SCRUM tiene la característica de poder dividir todo el trabajo en pequeños fragmentos denominados *sprints*, lo que permite reaccionar rápidamente a los cambios que puedan ir surgiendo en el proyecto. Esto resultó idóneo puesto que al principio del proyecto existía un cierto grado de incertidumbre acerca de ciertos aspectos del desarrollo del mismo.

2.1.1. Roles en SCRUM

- **Product Owner**. Este rol tiene como papel velar por la calidad del producto. Tiene consciencia de los requisitos y suele actuar como representante del cliente, realizando cambios y tomando decisiones que marcan el rumbo del producto final.
- **SCRUM Master**. Este rol es el encargado de gestionar todo el proceso de trabajo, gestionar los incrementos y las reuniones y agilizar al equipo de desarrollo para impedir bloqueos y aumentar la productividad.
- **Equipo de desarrollo**. Este rol puede estar formado por varias personas, y se encarga del desarrollo como tal del producto.

2.1.2. Artefactos

Existen una serie de documentos o registros de trabajo de notoria importancia dentro de un proyecto SCRUM.

- **Product Backlog**. Este documento contiene la lista de requisitos del cliente. Pese a que exista una versión inicial, esta puede evolucionar a lo largo del desarrollo del proyecto.
- **Sprint Backlog**. Este documento contiene las tareas a realizar durante un *sprint* o iteración del proyecto.
- **Incremento**. Este documento contiene el trabajo real realizado al final de un *sprint* o iteración.

2.1.3. Eventos

Una de las características de `SCRUM` es el constante control y revisiones sobre el trabajo realizado, tanto para comprobar la calidad del trabajo como para estar preparados para cambios en los requisitos [12]. Estas reuniones también tienen como objetivo facilitar la comunicación entre equipos de desarrollo, pero al estar este proyecto realizado por una persona, no se toma en cuenta dicha característica. Los eventos suelen tener una duración similar en cada una de las repeticiones y tienen una fecha y una frecuencia determinadas al inicio del proyecto.

- **Sprint**. También llamado iteración, es un periodo entre una y cuatro semanas en el que se desarrolla una parte del proyecto.
- **Sprint Planning**. Esta reunión tiene lugar al inicio del *sprint* y tiene como objetivo fijar las tareas que se deben realizar en dicha iteración.
- **Daily SCRUM**. Esta reunión es de carácter breve, entre quince y treinta minutos, y se realiza diariamente para sincronizar a los miembros de un mismo equipo de trabajo.
- **Sprint Review**. Esta reunión tiene lugar al final del *sprint* y tiene como objetivo revisar el trabajo realizado durante la iteración. El *feedback* obtenido en estas reuniones puede llegar a afectar al *Product Backlog*.
- **Sprint Retrospective**. Esta reunión se realiza entre *sprints*, por lo que se realiza después del *Sprint Review* de la iteración N pero antes del *Sprint Planning* de la iteración N+1. En este acto se debaten que aspectos han ido bien y cuales mal en el *sprint*, con el fin de realizar un mejor trabajo en la siguiente iteración.

2.2. SCRUM adaptado al proyecto

Habiendo explicado los distintos aspectos característicos de `SCRUM`, se exponen ahora las adaptaciones a este proyecto.

En el caso de los roles, la tutora del proyecto actuará tanto como *Product Owner*, haciendo de intermediario con los desarrolladores del proyecto `LOCOMOTION H2020`; como de *SCRUM*

Master, estableciendo las tareas a realizar en cada iteración. El alumno toma el papel de equipo de desarrollo.

En el caso de los *sprints*, se estableció que tendrían una duración de dos semanas. Se tendrían reuniones semanales establecidas los lunes a las 11:00. Estas reuniones actuarían como *Sprint Planning*, *Sprint Review* y *Sprint Retrospective* en las semanas de inicio de *sprint*; y como *Sprint Review* en las semanas intermedias. Este horario se mantuvo durante el primer cuatrimestre académico. En el inicio del segundo cuatrimestre, por motivos de compatibilidad de prácticas laborales del alumno, se modificó el horario de reuniones a los martes a las 18:30.

El *Product Backlog Inicial* se obtuvo a través de un primer análisis de las tareas a realizar al comienzo del desarrollo del proyecto. Tanto las historias de usuario como el *Product Backlog Final* se confeccionaron a medida que el proyecto fue desarrollándose y las tareas a realizar tomaban una forma cada vez más sólida.

2.3. Público objetivo del proyecto

Pese a la complejidad inicial que este trabajo puede transmitir, el público al que va dirigida esta solución es un público que si bien trabaja con ordenadores en su día a día, no tiene un conocimiento muy extenso en el ámbito de la informática en general, y menos aún si se compara con un Ingeniero Informático.

Los principales usuarios que harán uso de las herramientas desarrolladas son modeladores del proyecto `LOCOMOTION H2020`, que utilizan `Vensim` para programar los modelos utilizados para simular situaciones del planeta bajo varios parámetros concretos. Dichos modelos suelen ser muy complejos, llegando a tener decenas de vistas, cada una poblada con multitud de variables y/o gráficos. Por ello, a la hora de comprobar los cambios entre versiones de los archivos, los usuarios tenían grandes dificultades para entender las diferencias, puesto que los archivos `Vensim` en formato textual presentan una sintaxis complicada, especialmente en la definición de las vistas.

Por estos motivos, el desarrollo del proyecto enfocó su parte visual como su facilidad de uso en usuarios no familiarizados con el desarrollo de software. También se intentó en la medida de lo posible, facilitar la comprensión de los archivos `Vensim` en su formato textual. Por ello, se tomaron decisiones como modificar el archivo para incluir el nombre de la vista a la que una ecuación pertenece en el comentario de la misma, o utilizar parámetros numéricos de las variables para indicar su tipo, por ejemplo, distinguiendo entre variables convencionales o variables sombra (*shadow variables*). Las interfaces para modificar los archivos con el fin de facilitar la captura del valor semántico de los mismos se deben realizar de forma sencilla y con pocas opciones, con el fin de disminuir al mínimo la posibilidad de error. Estas interfaces de usuario, así como el funcionamiento del proyecto deben quedar explicados de una manera compresible para usuarios noveles en ámbitos de la informática en los anexos de **Manuales de Usuario**.

2.4. Product Backlog inicial

Un product backlog inicial se compone de aquellas historias de usuario u objetivos que debe tratar de abarcar el proyecto. Debido a la complejidad que pueden tener ciertas historias de usuario, existen algunas catalogadas como *epics* que deben ser desglosadas en tareas más sencillas.

En este proyecto, definimos el usuario promedio como un programador de Vensim perteneciente al proyecto `LOCOMOTION`, con pocos conocimientos sobre control de versiones y resolución de conflictos. Este usuario requiere resolver conflictos de archivos muy poco intuitivos de forma clara y sencilla.

A continuación se muestran las historias de usuario determinadas en un inicio, Tabla 2.1:

Número	**Descripción**
1	Como usuario quiero tener un plugin de SemanticMerge capaz de comprender los ficheros de Vensim y ser capaz de resolver los posibles conflictos generados de forma sencilla.
2	Como usuario quiero una aplicación que permita automatizar en los comentarios de las ecuaciones un código que refleje a qué vista pertenece dicha ecuación, sin inferir en el funcionamiento del control de versiones.

Tabla 2.1: Product Backlog inicial.

Se determinó que la primera tarea sería catalogada como *epic*, pues es un pequeño resumen del proyecto. A su vez, las tareas que componen la historia *epic* serían catalogadas como historias compuestas, Tabla 2.2.

Número	**Descripción**
1.1	Como usuario quiero disponer de una gramática que sea capaz de tratar todos los ficheros Vensim y cree un árbol sintáctico que desglose el fichero lo máximo posible.
1.2	Como usuario quiero que dicho plugin sea capaz de desglosar el fichero en unidades suficientemente pequeñas para que el usuario entienda los cambios de la forma más clara y concisa posible.
1.3	Como usuario quiero que el plugin sea capaz de distinguir qué partes de los archivos Vensim son triviales para el merge y cuáles producen cambios significativos.

Tabla 2.2: Desglose de la tarea 1.

A su vez, podemos desglosar las dos primeras historias de usuario en las siguientes tareas, Tablas 2.3 y 2.4:

Número	**Descripción**
1.1.1	La gramática debe ser compatible con la gramática de partida, incluyendo la definición de las vistas, gráficos y metadatos de los archivos Vensim.
1.1.2	La gramática debe soportar todos los archivos que pertenezcan a la versión 300 de Vensim, así como seguir los estándares del proyecto LOCOMOTION.
1.1.3	La gramática debe proporcionar información suficiente como para identificar cada línea de texto individualmente.

Tabla 2.3: Desglose de la tarea 1.1

Número	**Descripción**
1.2.1	El programa debe ser capaz de generar un archivo YAML compatible con SemanticMerge.
1.2.2	El programa debe proporcionar información suficiente para localizar correctamente el conflicto en un commit.
1.2.3	El programa debe superar los tests de calidad necesarios.

Tabla 2.4: Desglose de la tarea 1.2

Por otro lado, la segunda tarea del *backlog* inicial fue considerada una tarea compuesta, la cual puede dividirse en la Tabla 2.5:

Número	**Descripción**
2.1	El programa debe determinar a qué vista pertenece cada ecuación.
2.2	El programa debe de ser capaz de modificar el comentario de la ecuación para añadir la vista a la que pertenece.
2.3	El programa debe superar los tests de calidad necesarios.

Tabla 2.5: Desglose de la tarea 2.

2.5. Product Backlog final

Tras entrar en una fase avanzada del proyecto y clarificar las historias de usuario, (en la Tabla 2.6) se muestra el Product Backlog final del proyecto:

Número	**Descripción**
1	Como usuario quiero tener un plugin de SemanticMerge capaz de comprender los ficheros de Vensim y ser capaz de resolver los posibles conflictos generados de forma sencilla.
2	Como usuario quiero poder conocer a qué vista pertenece cada ecuación a la hora de resolver un conflicto.
3.1	Como usuario quiero poder comprender con detalle que ha ocasionado el conflicto entre dos archivos Vensim, en términos tanto de líneas de código como de estructuras asociadas al fichero Vensim.
3.2	Como usuario quiero poder distinguir que partes del fichero son importantes y cuales son triviales, a la hora de resolver el conflicto.
3.3	Como programador quiero que el archivo YAML generado sea compatible con la herramienta SemanticMerge.
3.4	Como programador quiero añadir delimitadores al archivo para que sea más comprensible y legible generar el archivo YAML.
3.5	Como usuario quiero poder eliminar dichos delimitadores tras la resolución del conflicto.
4	Como usuario quiero poder conservar la versión anterior a la modificación del archivo incluyendo los nombres de las vistas en las ecuaciones y delimitadores en un archivo con extensión de backup 2mdl.
5	Como usuario quiero que este plugin sea compatible con la herramienta gMaster.
6	Como usuario quiero que la experiencia de usuario sea intuitiva pese a tener muy pocos conocimientos en control de versiones.

Tabla 2.6: Product Backlog final.

2.6. Planificación

Esta beca tiene una duración inicial de 6 meses, aunque cabe la posibilidad de solicitar una prórroga hasta un periodo total de 12 meses. La beca está enfocada al desarrollo del *plugin* y no al desarrollo del TFG como tal, por lo que no incluye la redacción de la memoria. Por ello, se decidió ir tomando notas sobre las tareas realizadas mediante el `Issue Tracker` de `GitLab` [18] y realizar el grueso de la memoria una vez que el proyecto estuviese completo. El Trabajo de Fin de Grado se compone de 12 créditos ECTS [10], y según la Universidad de Valladolid, cada crédito ECTS corresponde aproximadamente a 25 horas de trabajo [11], por lo que el proyecto constaría de aproximadamente 300 horas de trabajo.

La planificación del proyecto se desarrolló en *sprints*, periodos de dos semanas donde se plantearían objetivos a cumplir. Durante esas dos semanas se estaría en contacto con la tutora para informarla del progreso del *sprint* y se tendrían dos reuniones, una al empezar el *sprint* y otra aproximadamente en la mitad del mismo.

Se planificaron un total de 12 *sprints*, comenzando el *sprint* inicial el día 14 de septiembre y finalizando el último *sprint* la última semana de febrero, aunque se prevé posible utilizar un *sprint* adicional en las primeras semanas de marzo. Como se ha comentado anteriormente, los *sprints* finales quedaron reservados para la parte de documentación y realización de la memoria del proyecto. Se debe nombrar también que durante el mes de agosto se llevó a cabo un *sprint* inicial o *sprint* 0, el cual se dedicó a familiarizarse con los entornos de `Vensim` y `SemanticMerge` así como a desarrollar pequeños *parsers* externos y a familiarizarse con `ANTLR4`.

Debido a que este Trabajo de Fin de Grado se realiza en paralelo a cinco asignaturas, existen semanas o *sprints* que han visto reducida su carga de trabajo debido a la acumulación de exámenes, entregas de prácticas u otro tipo de actividades evaluables. Estos *sprints* quedan resaltadas en gris en la Tabla de calendarización 2.7, aunque puede darse el caso de que la carga de trabajo se reduzca tan sólo en una semana del *sprint*:

Sprint 1	14/09/2020 - 27/09/2020
Sprint 2	28/09/2020 - 11/10/2020
Sprint 3	12/10/2020 - 25/10/2020
Sprint 4	26/10/2020 - 08/11/2020
Sprint 5	09/11/2020 - 22/11/2020
Sprint 6	23/11/2020 - 06/12/2020
Descanso	07/12/2020 - 17/01/2021
Sprint 7	18/01/2021 - 31/01/2021
Sprint 8	01/02/2021 - 14/02/2021
Sprint 9	15/02/2021 - 28/02/2021
Sprint 10	01/03/2021 - 14/03/2021

Tabla 2.7: Tabla de calendarización de sprints.

Se ha calificado el *sprint* 5 como complicado, debido a la cantidad de exámenes, entregas y otras actividades evaluables con fecha en ese periodo. Se previó hacer un parón en dicha etapa, aunque de encontrar tiempo, se avanzaría lo que se pudiese. Por otra parte, la segunda mitad del mes de diciembre y la primera mitad del mes de enero se han marcado como descanso debido a las entregas finales, preparación para la convocatoria ordinaria y vacaciones de navidad. Es posible que a pesar de ello se pudiese avanzar en este proyecto en dicho periodo, por lo que se destinó ese tiempo para recuperar lo no avanzado en el quinto *sprint* e incluso algo más en el caso de existir tiempo.

La fecha estimada de finalización del proyecto se planea para la segunda quincena de marzo, al acabar el décimo *sprint*, aunque podría alargarse algo más en el caso de que los requisitos cambiasen, la carga de trabajo del curso aumentase por alguna situación provocada por el `COVID-19` o alguna otra circunstancia externa.

2.7. Riesgos

En el inicio de la realización del proyecto se plantearon los posibles riesgos, puesto que este Trabajo de Fin de Grado va destinado a un cliente final y pueden existir complicaciones que deben ser notificadas desde un principio. Se han planteado las mitigaciones tomadas y las posibles contingencias según el problema [13]. A continuación, desde la Tabla 2.8 a la Tabla 2.13 se describen los posibles riesgos:

Riesgo 1	Falta de tiempo
Tipo	Personal
Probabilidad	Alta
Impacto	Medio
Descripción	Debido a que este Trabajo de Fin de Grado se realiza en el primer cuatrimestre en su mayor parte, implica que se deben realizar paralelamente cinco asignaturas más, por lo que puede darse el caso de que haya ocasiones donde la carga de trabajo del curso no deje espacio para trabajar en el proyecto todo lo deseado.
Mitigación	Planificar adecuadamente la carga de trabajo en función a las semanas con más carga del curso.
Contingencia	Replanificar las tareas a realizar por sprint y/o añadir un sprint adicional.

Tabla 2.8: Riesgo de falta de tiempo.

Riesgo 2	Actualización de Vensim
Tipo	Técnico
Probabilidad	Muy baja
Impacto	Medio
Descripción	Ventana Systems podría actualizar su software e incluir nuevas características que requiriesen realizar cambios en la gramática y derivados. Sin embargo, al haber lanzado la versión 8.1 a comienzos de 2020 y la versión 7 hace tres años, lo consideraremos altamente improbable.
Mitigación	Ampliar la gramática lo menos rígidamente posible, permitiendo incorporar futuros cambios.
Contingencia	En el caso de actualización de versiones, se debería hablar con el cliente y determinar si el software debe estar enfocado a la versión actual, a la posible versión futura o a ambas. En función de la decisión, se replantearía la gramática y sus derivados.

Tabla 2.9: Riesgo de actualización de Vensim.

Riesgo 3	Modificaciones en los requisitos
Tipo	Contractual
Probabilidad	Media
Impacto	Medio
Descripción	A lo largo del desarrollo del proyecto pueden surgir nuevos problemas que requieran implementar nuevas características o modificar algunas ya realizadas. Al estar el proyecto regulado por una organización de índole europea, la probabilidad de que en discusiones se llegue a acuerdos de modificación del proyecto es alta.
Mitigación	Tratar de mantener flexibles las partes del proyecto con mayor probabilidad a cambiar.
Contingencia	Replanificar los sprints en función del tamaño de los cambios a realizar.

Tabla 2.10: Riesgo de modificaciones de los requisitos.

Riesgo 4	Necesidad de actualizar la gramática
Tipo	Técnico
Probabilidad	Baja
Impacto	Medio
Descripción	Se pueden pasar por alto pequeños detalles o descubrir en un futuro que resulta más fácil resolver un problema haciendo cambios en la gramática que en el código.
Mitigación	Realizar la gramática de forma flexible.
Contingencia	Valorar en función de la cantidad de esfuerzo y tiempo que se invertiría en realizar cambios en la gramática si es más rentable trabajar sobre la gramática o sobre el código.

Tabla 2.11: Riesgo de necesidad de actualizar la gramática.

Riesgo 5	Modificaciones en la forma de interaccionar con SemanticMerge
Tipo	Técnico
Probabilidad	Muy baja
Impacto	Medio
Descripción	PlasticSCM podría actualizar su software y modificar la forma en que se incorporan los parsers externos a la aplicación. Sin embargo, al haber estado siguiendo el mismo método durante una gran cantidad de años, lo consideraremos altamente improbable.
Mitigación	Tratar de mantener flexibles las partes del proyecto, encapsulándolo en formato JAR.
Contingencia	En el caso de actualización de versiones, se debería hablar con el cliente y determinar si el software debe estar enfocado a la versión actual, a la posible versión futura o a ambas. En función de la decisión, se replantearía la forma de desarrollo del proyecto.

Tabla 2.12: Riesgo de modificaciones en la forma de interaccionar con SemanticMerge.

Riesgo 6	Pandemia por COVID-19
Tipo	Global
Probabilidad	Alta
Impacto	Medio
Descripción	Debido a la pandemia global del COVID-19, es posible que el Gobierno apruebe medidas excepcionales como un segundo confinamiento, lo cual obligaría a replantear los métodos de trabajo.
Mitigación	Preparar entornos de trabajo online alternativos.
Contingencia	Reorganizar las reuniones a un modo no presencial y reorganizar la carga de trabajo y la planificación, pues en caso de confinamiento, es posible que la carga de trabajo de las asignaturas del cuatrimestre suba.

Tabla 2.13: Riesgo de COVID-19.

2.8. Presupuesto simulado

Consultando el Boletín Oficial del Estado [33], en 2019 se estipuló que el sueldo medio de un programador junior a 1 de noviembre de 2020 sería de 16539,38 € anuales trabajando 1800 horas al año. Las empresas deben pagan a la Seguridad Social aproximadamente un 30 % del sueldo base del trabajador [24], por lo que el coste real para la empresa sería de 21501,2 €. Estimando el proyecto como unas prácticas de empresa, donde se emplean 300 horas, el presupuesto simulado sería de 3583,53 €.

Además, para este proyecto se requieren las herramientas de la empresa `PlasticSCMl`, concretamente `gMaster` y `SemanticMerge`. El coste mensual de la licencia de esta última es de 6,9$ (5,93 €) [36], y asumiendo que el proyecto se completaría en los seis meses estipulados, el coste de la licencia constaría de 35,58 €. El resto de software como `Vensim` o herramientas

para el desarrollo son gratuitas o se utiliza su versión gratuita.

El proyecto se realizará desde la residencia personal del alumno y la Universidad, por lo que no se incluirán costes de alquiler y similares, Tabla 2.14.

Sueldo base	2756,56€
Seguridad social	826,97€
Licencia de SemanticMerge	35,58€
Total	3619,11€

Tabla 2.14: Presupuesto simulado.

Capítulo 3

Análisis

3.1. Análisis de la gramática de Vensim

Este proyecto parte de una gramática inicial en `ANTLR4` desarrollada por Daniel Bazaco [5], antiguo alumno del Grado en Ingeniería Informática en la Universidad de Valladolid y que con su Trabajo de Fin de Grado colaboró también con en la finalización del proyecto `MEDEAS` y el inicio del proyecto `LOCOMOTION`.

3.1.1. Introducción a ANTLR4

`ANTLR4` (*ANother Tool for Language Recognition*) es una herramienta capaz de generar reconocedores de texto o *parsers* muy potentes que permiten trabajar con archivos estructurados proporcionando una gramática que lo represente. Actualmente se encuentra en su cuarta versión, tal y como se indica en el nombre.

`ANTLR4` es capaz de trabajar con dos tipos de estructuras, bien con *listeners* o bien con *visitors*. A grandes rasgos podemos diferenciarlos en que los métodos *listeners* son llamados automáticamente por `ANTLR` mientras que los métodos *visitors* son invocados explícitamente. Además, los *listeners* no pueden retornar valores y se deben utilizar variables externas para recuperar los valores necesarios, mientras que los *visitors* sí que son capaces de retornar valores. Por último, los *listeners* utilizan un espacio reservado de memoria en el *heap* mientras que los *listeners* utilizan llamadas a la pila o *stack* [50]. Por todo el control que permiten, en este proyecto se utilizan *visitors*.

`ANTLR4` se compone de dos mecanismos principales: el *lexer* y el *parser*. El *lexer* es el encargado de framentar el archivo a tratar en prqueñas etiquetas o *tokens* y el *parser* es el encargado de construir el árbol sintáctico a partir de dichos *tokens*. `ANTLR4` construye el árbol de arriba a abajo (*top-down*) utilizando un mecanismo recursivo descendente y una gramática LL, por lo que no necesita utilizar *backtracking*.

Las reglas del *parser* se encuentran al principio del archivo `ANTLR4` (con extensión `.g4`) y comienzan con letra minúscula. Estas reglas son las encargadas de formar el árbol sintáctico. Las reglas del *lexer* se encuentran posteriormente a las del *parser* y deben comenzar con letra mayúscula. Estas reglas transforman el texto de entrada en *tokens* con los que construir el árbol. En el caso de que un predicado coincida con varias reglas, siempre tomará prioridad la regla con mayor similitud de coincidencia. Es decir, si la palabra `text` tiene coincidencias con dos reglas, una que toma todas las letras como expresión regular (`[a-zA-Z]`) y otra que directamente utiliza la palabra `text`, el árbol se construirá con la segunda regla. En el caso de existir dos reglas con la máxima coincidencia para un valor de entrada, tomará prioridad la regla que esté declarada antes en el archivo `.g4` [8].

3.1.2. Gramática inicial de partida

La gramática de partida trataba exclusivamente las definiciones de ecuaciones y estaba pensada para modelos con una única vista. Pese a ello, la gramática puede resultar compleja en un inicio y a continuación se explican los pilares fundamentales de la misma:

- Un fichero se subdivide inicialmente en dos partes, el contenido del fichero y el caracter *End Of File* ($< EOF >$), que indica cuando acaba el fichero y por tanto cuando acaba el árbol.
- A su vez, la parte del modelo se subdivide en una sucesión indeterminada de definiciones de símbolos o de macros del archivo. Posteriormente se define la parte de *sketches*, la cual ha sido ampliada en este proyecto y se explicará más adelante. En la gramática inicial, la parte de *sketches* se limitaba a capturar la primera línea que declaraba que comenzaba la definición de las vistas y los gráficos en el fichero `.mdl`.
- Por su parte los símbolos se dividen en definiciones de símbolos y comentarios y unidades asociados al símbolo. En la definición de símbolos encontramos todos los tipos básicos explicados en la sección de *"Introducción a ANTLR4"*, en el capítulo de *"Introducción"*. Dichos tipos son: *lookup*, *subscript*, ecuación, constante, constante no modificable, ecuación de datos, *string* o cadena de caracteres, copia de *subscript* y *reality check*. Cada uno de estos tipos de datos tiene su propia estructura definida, apoyándose en ocasiones en más reglas.
- La gramática omite los comentarios o las definiciones de codificación, caracterizadas por estar ambos descritos entre llaves "{ }"; así como las líneas de asteriscos utilizadas para separar secciones del documento `.mdl`. También se ignoran los *backslash* (\), utilizados para indicar saltos de línea dentro de los metadatos de los símbolos; así como cualquier tipo de espacio en blanco, véase espacios, tabulaciones o saltos de línea.
- Las reglas de generación de *tokens* del *lexer* se componen de un conjunto de definiciones de operadores y un conjunto de definiciones de tipos básicos, tales como identificadores ("Id"), constantes de tipos básicos o definiciones de tipos numéricos, como números enteros, reales o racionales. Estas definiciones en su mayoría de casos se realizan a través de fragmentos o *fragments*, pequeñas reglas que sólo pueden ser utilizadas para definir otras reglas del *lexer* y que tienen como objetivo clarificar la gramática, como por ejemplo, la definición del conjunto de dígitos: $[0 - 9]$.

3.1.3. Ampliación de la gramática I

Como se ha explicado anteriormente, la gramática inicial no trataba las definiciones de múltiples vistas y los gráficos que se pueden definir. Se expandió la parte de *sketches* de la gramática inicial tal y como se explica a continuación [29]:

- Primeramente, se amplió la regla de definición del modelo, añadiendo a los *sketches* el delimitador de la sección de gráficos, los gráficos y la sección de metadatos del fichero.
- En un fichero `Vensim` pueden existir varias vistas, pero todas ellas siguen la misma estructura. Las vistas comienzan con dos líneas de texto, la primera anuncia que a continuación se muestran los datos del *sketch* y la segunda muestra el código de versión, el cual en las versiones 3, 4 y 5 será siempre el 300 (`V300`) [53]. Tras ello, la siguiente línea contendrá el nombre de la vista. Las siguientes líneas contendrán toda la información asociada a la vista. La primera línea, distinguible pues empieza por '$', indica la configuración de fuente y color de la vista, indicando el tamaño de letra y color, las características de las flechas y el fondo entre otros. Las siguientes líneas definen variables, y pueden pertenecer a uno de los siguientes tipos:
  - **Objetos**. Es el tipo más común de variable, y tienen un número indeterminado de campos. Representan variables, *valves*, comentarios, mapas de bits o metaarchivos, con los identificadores 10, 11, 12, 30 y 31 respectivamente.
  - **Flechas o *arrows***. Representan las flechas o relaciones entre las variables del modelo. Se caracterizan por tener un campo final compuesto por un número que representa el número de puntos de la flecha, seguido de las coordenadas de dichos puntos por donde pasa la flecha y permite ser trazada. Los campos de las flechas no incluyen texto, son sólo numéricos.
  - **Variables sombra o *shadow variables***. Variables que no están definidas en la propia vista, sino en otro lugar y por ello no pueden depender de otras variables, aunque otras variables pueden depender de ellas. Un ejemplo sería el tiempo.
  - **Variables de texto**. Variables objeto cuyo formato ha sido modificado, es decir, se ha alterado su color, fuente o tamaño entre otros.
- El delimitador de la sección de gráficos es una línea de texto con el formato "$///---\\\$", pero debido a que los *backslash* se omiten en la gramática, se utiliza únicamente "$///---$".
- La sección de gráficos es un tanto ambigua, puesto que al poder el usuario definir los campos que el quiere, la gran mayoría de campos son opcionales en la gramática. Sin embargo, todos los gráficos comenzarán con el campo "`:GRAPH`" que contendrá el identificador del gráfico; y el campo "`:TITLE`", que contendrá el nombre del gráfico. Todos los apartados del gráfico estarán en mayúsculas y precedidos por ':'.
- Finalmente, la sección de metadatos contiene datos generales del archivo, tales como variables, unidades escogidas en la creación del fichero u otros. Los metadatos se caracterizan por empezar con una línea con el siguiente texto: "$: L < \% \wedge E!@$". Cada línea en la sección de metadatos se caracteriza por utilizar primeramente un número, seguidamente el símbolo ':' y posteriormente un campo indeterminado que puede ser numérico, cadena de caracteres o incluso nulo.

3.1.4. Ampliación de la gramática II

Tras una reunión con el alumno Juan Herruzo, el cual se encontraba realizando también un Trabajo de Fin de Grado para el proyecto `LOCOMOTION H2020` y que también se le había encargado ampliar la gramática de Daniel Bazaco, se llegaron a las siguientes conclusiones sobre el procesamiento de las líneas que identifican los campos de cada vista:

- Sólo existen dos tipos de clasificaciones: flechas o variables. Todas las variables están representadas por hasta unos 25 campos, pero no todos son obligatorios. Varios de ellos son los campos de formato, donde en las variables comunes no aparecen, pero si modificamos su tipografía. color o tamaño sí lo hacen. Sobre las variables sombra, se determinó que el campo número 9, *bits*, podría determinar si una variable es sombra o no. El primer bit de este campo (que hace que el número en notación decimal sea par o impar), define si pueden dirigirse flechas hacia una variable. Como las variables sombra no lo permiten, se determinó que dicho número sería par en las variables sombra e impar en variables comunes.

- Se clarificó la gramática, añadiendo etiquetas para distinguir mejor los campos de las variables y añadiendo nuevas reglas para clarificar dichos campos. Además, se añadió la documentación sobre los campos de las variables, expuesta en secciones posteriores. Sobre todo, se matizó la parte de la tipografía de las vistas para poder clasificar mejor las variables.

- Existen ocasiones donde el tercer campo de una línea de definición de variable tiene un 0 en vez de contener el nombre de la variable. En esos casos, el nombre de la variable se indica por separado en la línea siguiente. Se modificó la gramática para añadir la línea separada a la anterior en vez de tratarlas como dos líneas por separado.

3.1.5. Diferencias entre gramática inicial y final

Las diferencias entre las dos gramáticas pueden ser apreciadas con mayor detalle en:

3.2. Parámetros de las vistas

En los archivos `.mdl` de `Vensim`, en la parte donde quedan definidas las vistas, podemos observar una gran cantidad de líneas de números, las cuales hay que diferenciar, saber qué significa cada parámetro y conocer qué parámetros son triviales y cuales no para a la hora de realizar un *merge* que genere conflictos graves, poder tomar las decisiones de conservar y eliminar con facilidad. Este apartado queda explicado con profundidad en el anexo de **Manuales de Usuario** de la memoria, sección A.2.4.

3.3. Consideraciones sobre la eliminación e inserción de variables

Un problema planteado en el desarrollo del proyecto es el cómo controlar las variables sombras. Las variables sombra se definen por ser variables que no se definen en la vista, sino en el proyecto en general. Son variables externas de las que pueden salir flechas, pero las flechas no pueden dirigirse hacia ellas [52]. Existen dos tipos de variables sombra, las "puras", como puede ser el concepto del tiempo; y las "derivadas", las cuales se crean a partir de las variables definidas. Las variables sombra puras tienen la característica de que no pueden eliminarse del modelo y las derivadas tienen la característica de que son eliminadas si su variable definida de la que procede es borrada.

Surge entonces el siguiente problema: si una variable se elimina (arrastrando a otras o no), y posteriormente se procede a crear una nueva variable con el mismo nombre que la anterior, ¿cómo podemos distinguir a través del parser si realmente se ha borrado y creado una nueva variables o si simplemente se ha movido la variable? Esto es posible distinguirlo a través del segundo campo que comparten tanto las flechas como las variables y que se ha explicado en la sección anterior, el id, el cual estipula el orden de aparición de la variable en la vista. Se exponen a continuación dos posibles casos:

- **La variable no es la última que se ha incluido a la vista.** En este caso, si moviésemos la variable, sus datos de posición y relativos cambiarían, pero su id permanecería intacto. En el caso de borrar dicha variable y crear una nueva, se crearía una nueva línea al final de los datos de la vista con un nuevo id con el número más alto hasta el momento.
- **La variable es la última que se ha incluido a la vista**. En este caso, moviésemos la variable cambiarían sus coordenadas, pero si borrásemos la variable y creásemos una nueva, el id sería el mismo en ambas. Sin embargo, este escenario plantea que la creación de la variable es el último movimiento que hemos hecho, por lo que ambos mover la variable y borrarla para volverla a crear significarían que de alguna forma el usuario se ha equivocado creando la variable y quiere rectificar su error, por lo que es un caso que podemos no tener en cuenta.

3.4. Análisis de SemanticMerge y gMaster

`SemanticMerge` no está enfocado a un paradigma de programación concreto como podría ser la orientación a objetos, a pesar de que los primeros lenguajes que la herramienta pudo tratar fueron C#, C++ y Java. El software es realmente efectivo ya que trata a todos los lenguajes por igual, utilizando una estructura de contenedores y nodos, o padres e hijos. Esta estructura es muy comparable al patrón de diseño *"Composite"* [15], donde los hijos de un padre pueden ser a su vez el tipo del padre, Figura 1.16. Gracias a esto, un lenguaje basado en la orientación al objeto como Java podría dividirse en clases y métodos, mientras que un lenguaje funcional como Scala podría dividirse en funciones y declaraciones. Esto

es mayormente apreciable gracias al diagrama de clases que se muestra en la Figura 1.16, extraído directamente de la página oficial de `SemanticMerge` [40].

El proceso interno que sigue `SemanticMerge` es el siguiente:

1. El archivo que es leído se fragmenta a través de una gramática que represente el formato de dicho archivo. Una vez identificadas sus partes, se genera un archivo `YAML` [1] que especifica dónde y cuando empieza cada contenedor así como la longitud en caracteres de cada uno de los nodos.
2. Tras generar este archivo, se comprueba tanto que su formato sea correcto, es decir, que se respete el indentado y la sintaxis propios del formato `YAML`; como que el archivo `YAML` represente de forma continua el archivo de entrada, esto es, que por ejemplo no pase del carácter 200 al 300 sin haber leído todos los caracteres de por medio.
3. Tras esto, se procede a la reconstrucción del archivo inicial a través del `YAML` y se comprueba que coincida con el archivo leído en primer lugar. En caso de no poder realizar la reconstrucción, la aplicación notificará al usuario de que no se ha podido reconstruir dicho fichero, y no podrá utilizarse la funcionalidad aportada por SemanticMerge.

3.4.1. Creación de un parser externo para SemanticMerge

Los *parsers* externos de `SemanticMerge` siguen la estructura de contenedores y nodos explicada en la sección anterior de forma interna, sin embargo, también se debe respetar una estructura externa a la hora de crear el *parser* [30].

El programa se debe ejecutar con dos argumentos adicionales. El primero es inmutable y es "`shell`", indicando que se quiere seguir ejecutando el programa en la terminal hasta recibir la orden "end". El segundo es el nombre de un archivo bandera o *flag*. Este archivo se utiliza para indicar al *parser* cuando se está listo para recibir los ficheros de entrada. Esto se hace escribiendo "READY" en el fichero *flag*. Borrar dicho fichero no es necesario al lanzar el programa.

```
try (PrintWriter out = new PrintWriter(args[1])) {
            out.println("READY");
        } catch (IOException ex) {
            ex.printStackTrace();
        }
```

Inicialmente y estrictamente en el orden en que se indica, el programa debe pedir al usuario el primer archivo a comparar, la codificación que utiliza (normalmente `UTF-8`) y un primer archivo de salida donde escribir el archivo `YAML` asociado al primer archivo. Si se

[1] ¿Qué son los archivos YAML? `https://www.cloudbees.com/blog/yaml-tutorial-everything-you-need-get-started/`

ha realizado correctamente, el programa imprimirá por pantalla “OK”, y en caso contrario imprimirá ”KO”. Estos pasos se repetirán para el segundo archivo. Tras completar el *parseo* de los dos ficheros, el programa deberá escribir ”end” por pantalla. Es muy importante que los ficheros `YAML` de salida sigan la estructura planteada en la página oficial de `SemanticMerge` [40] o una similar basada en contenedores y nodos para que se reconozcan dichos archivos correctamente.

```
String firstFile = scanner.nextLine();
String firstEncoding = scanner.nextLine();
String firstFileOutput = scanner.nextLine();
ParseFile(firstFile, firstFileOutput);
System.out.println("OK");
```

Como se puede observar, en este caso no se utiliza la codificación pues se asume que siempre será `UTF-8`. Sin embargo, es obligatorio pedirla por teclado para que el *parser* externo sea compatible con `SemanticMerge`.

Finalmente, se debe transformar el programa realizado a un archivo ejecutable (`.exe`) o algún formato similar que sea compatible con las opciones de línea de comandos de `SemanticMerge`. Por ejemplo, utilizando el lenguaje `Java`, el cual se utilizará en este proyecto, es posible comprimir el proyecto en un archivo `JAR`, el cual es compatible con `SemanticMerge`.

```
.\semanticmergetool.exe  --source=test1.code --destination=test2.code
--externalparser="-jar mvnparser-1.0-jar-with-dependencies.jar"
--virtualmachine="C:\Program Files\Java\jdk-11.0.8\bin\java.exe"
```

En este caso, `test1.code` y `test2.code` serían los archivos iniciales a comparar. Posteriormente se procesarán y se generará un fichero `YAML` por cada archivo, cada cual conteniendo la estructura en árbol de su respectivo archivo inicial.

3.4.2. Añadir parsers externos a gMaster

Para poder utilizar *parsers* externos en la herramienta `gMaster` debemos seguir un proceso muy sencillo [27]. Debemos situarnos en la carpeta `config` del directorio donde se encuentra ubicado `gMaster` y crear un archivo (si no ha sido creado ya) llamado `externalparsers.conf`. En este archivo debemos indicar para cada extensión de archivo que queramos procesar, la ubicación del *parser* externo en nuestro equipo. Para ello deberemos escribir su ruta absoluta, y se recomienda escribirla con los directorios comunes a todas las máquinas en inglés, pues produce menos problemas a la hora de detectar la ruta. Esto es, utilizar *Users* o *Desktop* en lugar de *Usuarios* o *Escritorio*. Tras ello, deberemos de reiniciar `gMaster` si estaba abierta previamente, y con ello, la herramienta detectará automáticamente el nuevo formato.

En el caso de este proyecto, el fichero `externalparsers.conf` tiene el siguiente aspecto:

```
.mdl=java -jar C:\Users\Propietario\AppData\Local\semanticmerge\
mvntfg-1.0-jar-with-dependencies.jar
```

Una situación que se debe tener en cuenta a la hora de integrar *parsers* externos en la herramienta `gMaster` es que dicho *parser* no funciona exactamente igual que en `SemanticMerge`. Cuando lanzamos el *parser* en `SemanticMerge`, como se ha explicado anteriormente, bien por línea de comandos o por interfaz gráfica, el usuario introduce los dos únicos ficheros que van a ser tratados.

Sin embargo, utilizando `gMaster`, la herramienta no lanzará el *parser* en cada comparación de archivos, sino que se tratará de una ejecución única. Este proceso se realiza de esta manera por términos de eficiencia. Concretamente, se inicializa el *parser* en la primera ejecución y se este queda a la escucha hasta que la aplicación termina. Esto mejora el rendimiento, sobre todo en aquellos casos de *parsers* que tienen un coste más elevado de arranque, como puede ser el caso de los *parsers* de `Java` que requieren de la `JVM` para su ejecución. Por ello, se debe modificar el parser para que acepte procesar ficheros hasta que no se reciban más órdenes, en vez de limitarse a únicamente dos ficheros cómo se realiza en `SemanticMerge`. Esto puede apreciarse en el siguiente fragmento de código:

```
while (scanner.hasNext()) {
    String firstFile = scanner.nextLine();
    String firstEncoding = scanner.nextLine();
    String firstFileOutput = scanner.nextLine();
    parseFile(firstFile, firstFileOutput);
    System.out.println("OK");
}

scanner.close();
String end = scanner.nextLine();
if (end.equals("end")) {
return;
}
```

El proceso completo puede ser apreciado en la Figura 3.1:

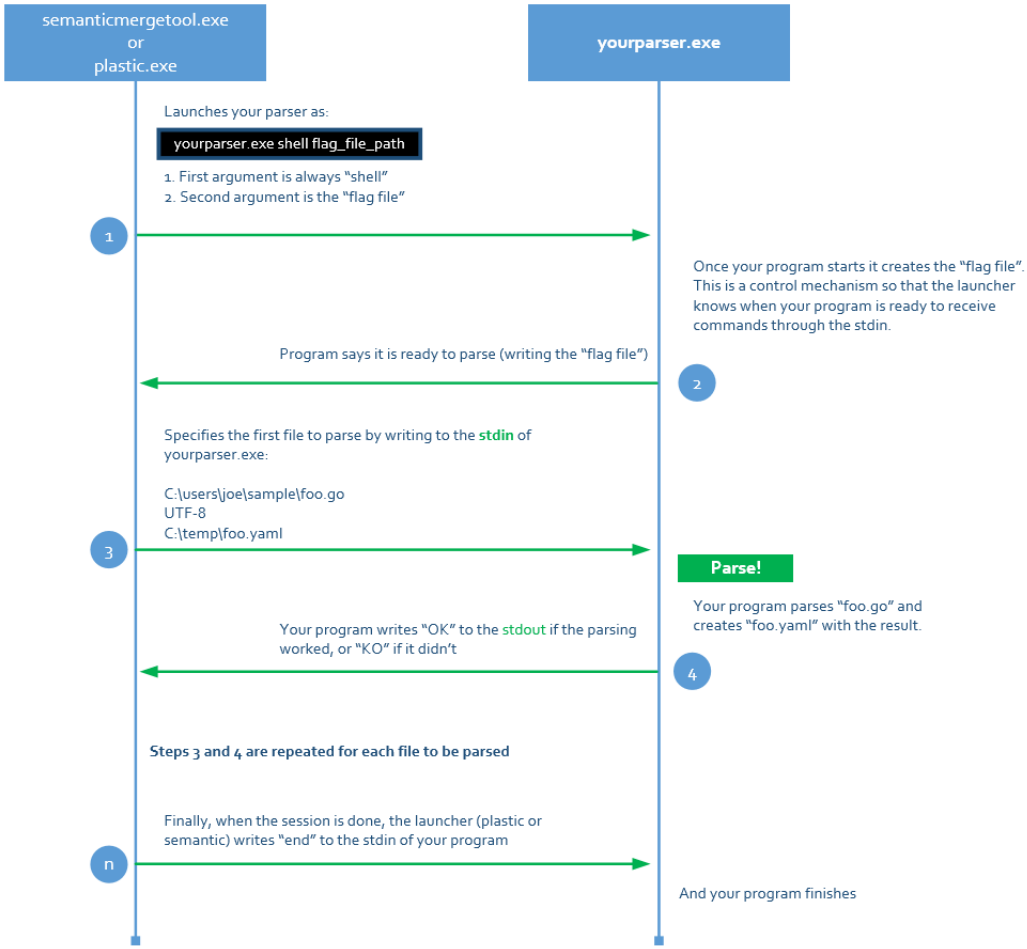

Figura 3.1: Proceso de comunicación de SemanticMerge con el parser externo, [40]

Capítulo 4

Tecnologías utilizadas

En este Capítulo se presenta un resumen de las tecnologías utilizadas tanto para la gestión del proyecto, como para el desarrollo.

4.1. Tecnologías para la gestión del proyecto

4.1.1. Rocket.chat

`Rocket.chat` [7] es una plataforma de comunicación por texto utilizada para preguntar dudas sobre el proyecto a la tutora del proyecto y para planificar las reuniones semanales de cada sprint.

4.1.2. Jitsi

`Jitsi` [25] es una plataforma de comunicación por videollamada de ámbito académico similar a otras como `Webex` o `Zoom`. La Escuela de Ingeniería Informática la ha integrado en sus principales aplicaciones, entre ellas su servidor de `Rocket.chat`. Mediante esta aplicación se han mantenido las reuniones semanales de cada Sprint.

4.1.3. Basecamp 3

`Basecamp 3` [4] es una plataforma de mensajería asíncrona que se utilizó para comunicarse con los miembros de la empresa `Códice Software` para resolver dudas acerca del funcionamiento de `SemanticMerge` o `gMaster` y relacionadas con el desarrollo de un plugin para añadir soporte a un nuevo lenguaje.

4.1.4. GitLab Issue Tracker

`GitLab` [18] es una interfaz gráfica online para el uso del software `git`. Proporciona una interfaz gráfica y muchas otras facilidades a los usuarios, y la Escuela de Ingeniería Informática cuenta con su propio servidor. Su gestor de *issues* permite organizar las tareas, añadir comentarios a las mismas, estimar su duración y anotar el tiempo real que se le ha dedicado. Resulta muy útil para calcular el tiempo real que se le ha dedicado al proyecto.

4.2. Tecnologías para el desarrollo

4.2.1. SemanticMerge

`SemanticMerge` [43] es una herramienta gráfica para la resolución de conflictos entre archivos, la cual se apoya en la semántica del archivo en vez de simplemente en la estructura del texto. Es la herramienta para la que se desarrollará el *plugin* externo y por ende, la utilizada para realizar el *testing* de dicho *plugin*.

4.2.2. SemanticMerge DEBUG

Editando el archivo `semantic.log.conf` tal y como se indica en el apartado dedicado a *parsers* externos de la web de la aplicación [40], se pueden obtener los errores que surgen al fallar el procesamiento de los archivos cuando se incluye un parser externo a `SemanticMerge`. Estos errores están disponibles en el fichero `semantic.log.txt` e incluyen errores desde lecturas incorrectas a fallos en la reconstrucción del archivo a partir del `YAML`.

4.2.3. GMaster

`GMaster` [37] es una herramienta cuyo propósito es proporcionar una interfaz gráfica al usuario para permitirles realizar todas las operaciones relacionadas con *git* y el control de versiones de una manera más visual y por tanto sencilla. La herramienta incluye toda la tecnología de `SemanticMerge`, por lo que también aporta un significado semántico a la hora de comparar los cambios realizados en los archivos. Esta herramienta se utilizó para verificar que la solución desarrollada era correcta, así como para las pruebas de usuario.

4.2.4. Git + GitLab

`Git` [58] es un software destinado a la gestión de proyectos y el control de versiones. Permite almacenar en un servidor todas las versiones del proyecto que se han ido actualizando según el transcurso del mismo. Además, permite la creación de ramas, bifurcaciones del proyecto con las cuales una o varias personas pueden tomar un punto del proyecto inicial,

ampliarlo paralelamente y posteriormente juntar los resultados, siempre con la posibilidad de volver atrás en el proyecto.

Por otra parte, como se ha explicado antes, `GitLab` [18] es una aplicación web que provee de interfaz gráfica a la tecnología de `git`. Facilita la gestión del proyecto al usuario al ser más visual y al proveer de herramientas secundarias como el *Issue Tracker* o mecanismos de despliegue continuo e integración continua.

En este proyecto se ha utilizado la metodología de trabajo *gitflow*, donde apenas se realizan cambios en la rama principal `master` (salvo aquellos que sean mínimos o cambios a ficheros de texto como el `README` o la licencia). De la rama `master` surge la rama `develop`, y de esta surgen varias ramas, una por cada *issue* propuesto. Tras concluir la tarea del *issue*, se cierra el mismo y se fusiona o *mergea* la rama del *issue* con `develop`.

Gitlab CI/CD

Una de las características que proporciona `GitLab` es la capacidad de realizar tanto integración continua como despliegue continuo de forma gratuita. La integración continua o *continuous integration* (CI) consiste en realizar el proceso de *testing* en cada *commit* que se realiza. En este proceso se evalúa el proyecto con todos los tests creados hasta la fecha. Si a medida que avanza el proyecto se realizan más pruebas, estas se añadirán a la batería de tests, comprobando que el programa los pasa correctamente en las siguientes iteraciones del proyecto desde ese momento en adelante.

Por otro lado, el despliegue continuo o *continuous deployment* (CD) consiste en desplegar el trabajo realizado hasta la fecha en una máquina virtual, con el objetivo de comprobar que el trabajo es válido y no funciona solamente en la máquina en la cual está siendo desarrollado.

4.2.5. Visual Studio Code

`Visual Studio Code` [32] es el entorno de desarrollo que se ha elegido para realizar el proyecto. Debido a su gran cantidad de plugins externos, permite una compatibilidad, debugging y otras características de integración con `Java`, `ANTLR4` o `Maven`.

4.2.6. Maven

`Maven` [2] es una herramienta utilizada para la construcción y configuración de proyectos `Java`. Este *framework* proporciona una estructura determinada y permite gestionar las dependencias externas y la configuración del proyecto a través de un archivo `pom.xml`.

4.2.7. JUnit 5 Jupiter

`JUnit 5`, versión también conocida como *Jupiter* [51], es un framework para el realizado de tests para `Java`, el cual tiene integración con `Maven`. Se utilizó para comprobar la validez del *software* creado.

4.2.8. Astah Professional

`Astah Professional` [23] es la versión profesional de una herramienta utilizado para el modelado y el diseño de software aplicando el lenguaje `UML` [20], el cual es un lenguaje universal para el desarrollo de diagramas para la ingeniería del *software*.

Capítulo 5

Diseño

Tras analizar el alcance del programa se decidió dividir la estructura en tres herramientas. La primera de todas sería el *parser* en sí, programa principal del proyecto encargado de procesar los archivos `Vensim`. Las otras dos herramientas consisten en dos modificadores de archivos de texto. La primera herramienta tiene la función de añadir tanto los nombres de las vistas en las ecuaciones como ciertos delimitadores que ayudan a procesar la semántica de los archivos. La segunda herramienta tendría el efecto contrario, eliminando del archivo tanto los nombres de las vistas como los delimitadores. Mientras que añadir los nombres de las vistas tiene una función de apoyo para los modeladores para ayudarles a entender mejor los archivos `.mdl` en su formato textual,los delimitadores son una herramienta para ayudar a transformar el archivo de partida en un archivo `YAML` con mucha menor dificultad.

5.1. Automatización de la inserción de los nombres de las vistas en los comentarios de las ecuaciones.

Inicialmente, se dividió esta tarea en tres partes debido a su potencial complejidad. Este proceso puede ser visto como una fragmentación del archivo a leer para obtener las partes clave, en este caso vistas y ecuaciones, para posteriormente volver a montar su estructura inicial. Se destacan a continuación las siguientes subsecciones:

5.1.1. Localización de las vistas.

Para obtener los nombres de las vistas, en primer lugar se divide el archivo en dos secciones claramente diferenciadas: la parte de definición de ecuaciones y la parte con los datos de las vistas. Esta segunda parte se subdivide a su vez, dejando atrás la información referida a los gráficos y a los metadatos. Tras esto, mediante la función `split()` se obtiene un *array* con

los conjuntos de datos de cada vista. Esto es posible debido a que cada vista se separa por los símbolos $\backslash\backslash\backslash - - - ///$.

Tras hacer esta separación, se obtiene el nombre de la vista a partir de la tercera línea de la separación anterior (se separan según los saltos de línea) y se añade a un *set*. Tras esto, se leen el resto de líneas que definen la lista, añadiendo el tercer campo de la línea separada por comas, o el primer campo en caso de que la línea esté definida por un único valor. Todos estos valores son añadidos a un segundo *set*, que a su vez es añadido a un *set* que contendrá todos los *sets* de valores de cada vista. De esta forma, habrá dos sets paralelos, uno proporcionará el nombre de la vista, y el otro proporcionará todos los nombres de variables de dicha vista. Esto puede ser apreciado en el siguiente fragmento de código, el cual corresponde a la función destinada a crear dichos *sets*:

```
private static Set<String> crearSets(String views) {
        String[] lines = views.split("\n");
        Set<String> set = new HashSet<String>();
        for (int i = 4; i < lines.length; i++) {
            String[] positions = lines[i].split(",");
            if (positions.length > 2) {
                set.add(positions[2]);
            } else {
                set.add(positions[0]);
            }
        }
        return set;
    }
```

5.1.2. Modificación del comentario de la ecuación.

Aprovechando la separación realizada en el apartado anterior, tomamos la parte del fichero que contiene la declaración de ecuaciones. A su vez, la dividimos en ecuaciones declaradas explícitamente y en ecuaciones declaradas por `Vensim` que representan la configuración del archivo, como `INITIAL TIME` o `TIME STEP`.

Cada ecuación termina en un símbolo |, por lo que fue fácil dividirlas. Para identificar la vista a la que pertenece la ecuación, se toma la primera línea de cada ecuación, quedándose sólo con la cadena de caracteres hasta llegar a el símbolo '='. Existen ecuaciones que pueden estar definidas por parámetros de entrada u otros argumentos, por lo que tras ello se eliminan todos los caracteres que vayan seguidos de un paréntesis '(' o un corchete de apertura '[', eliminando estos símbolos inclusive.

5.1.3. Unión de las partes divididas.

Tras esto, se compara la cadena obtenida por todos los *sets* de nombres de variables. Si se encuentra una coincidencia, se elimina dicha variable del *set* para evitar duplicaciones y se toma el índice del *set* de nombres de variables dentro del conjunto de *sets*. Con ese índice podemos obtener el nombre de la vista a través del *set* de sombres de vistas. Obtenido ese nombre, se pasa a escribir tras el segundo símbolo '∼' en la ecuación, el cual precede al apartado de comentarios de la variable. Para diferenciar el nombre de la vista con el posible comentario que existiese con anterioridad, se precede al nombre de la vista con `<[VIEW]>:` y el comentario anterior con `<[DESCRIPTION]>:`. En el caso de ser un archivo ya procesado, se obviaría la inserción de dichos diferenciadores, actualizándose simplemente el nombre de la vista. Tras este proceso, se vuelven a juntar las partes del fichero fragmentado, quedando como resultado el fichero original con los nombres de las vistas añadidas en los comentarios de las variables. En el siguiente fragmento de código se puede observar la función encargada de diferenciar si el archivo ha sido procesado previamente y en caso contrario, introducir el delimitador `<[VIEW]>:`.

```
private static String modify(String equation, String viewName) {
        int position = StringUtils.ordinalIndexOf(equation, "~", 2);
        String appendix = viewName.trim();
        if (equation.indexOf("<[VIEW]>:") == -1) {
            appendix = "<[VIEW]>: " + appendix;
            return insertString(equation, appendix, position);
        } else {
            return updateViewName(equation, appendix, position);
        }
    }
```

5.1.4. Diagrama de actividad.

A continuación, se muestra un diagrama de actividad que trata de explicar el proceso que sigue la primera herramienta a la hora de introducir los nombres de las vistas en las ecuaciones, Figura 5.1.

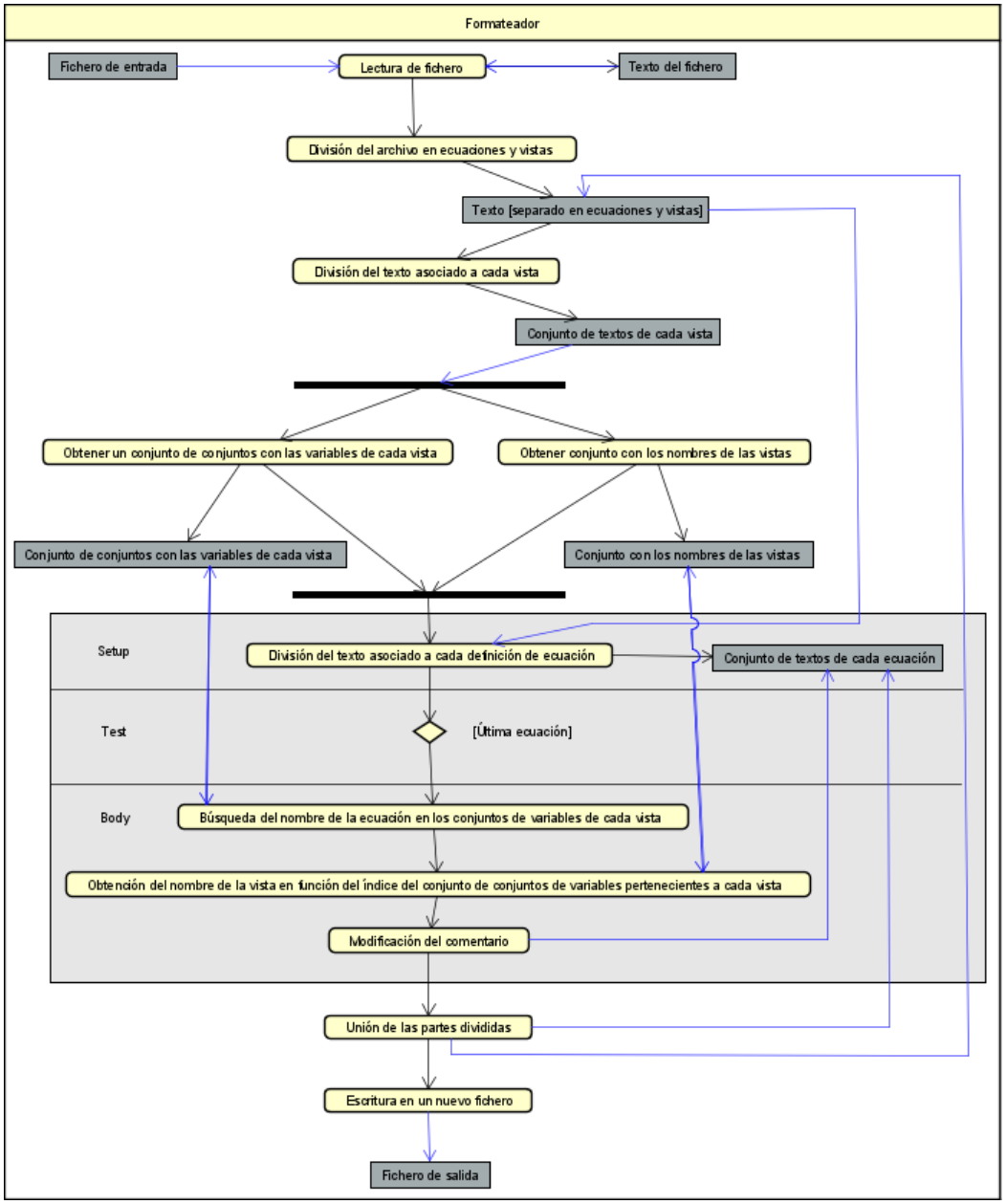

Figura 5.1: Diagrama de actividad asociado al programa

5.2. Desarrollo del primer parser externo a partir de la gramática de partida.

Debido a la posible complejidad inicial, se decidió que inicialmente se realizaría un primer *parser* básico que sólo incluyese las partes definidas en la gramática de partida, es decir, las definiciones de ecuaciones.

La estructura del árbol `YAML` se dividiría en dos principales, *equations* y *graphs*, conteniendo *equations* la definición de las ecuaciones, *lookups*, *subscript ranges* y demás elementos; y *graphs* el resto del fichero, que sería definido en posteriores *sprints*. En el caso del apartado de ecuaciones, se utilizaría como cabecera siempre la primera línea del archivo, dedicada a la definición de la codificación del texto, siendo en la mayoría de archivos, por no decir en todos, `UTF-8` (tras hablar con los desarrolladores de `LOCOMOTION H2020`, se confirmó que todos los archivos del proyecto utilizarían esta cabecera). El footer de este apartado siempre serán los dos últimos caracteres de salto de línea de la ecuación `TIME STEP`, la cual está presente siempre en los archivos al ser una ecuación que define los parámetros del propio archivo `Vensim` y no del modelo (esto de nuevo se confirmó con los desarrolladores del proyecto).

En el caso de las ecuaciones, simplemente se van añadiendo el número de líneas y de caracteres a variables globales para seguir la cuenta, pero se han de destacar dos cosas. La primera, es que normalmente la última línea de cada ecuación es un salto de línea compuesto por dos caracteres, `\r y \n`. Sin embargo, existen casos en que esto no es así, por lo que se comprueba para cada ecuación esta posibilidad y mantener correctamente la cuenta de número de líneas y de caracteres. La segunda es que existen dos clases de ecuaciones, las definidas para las diferentes vistas y las propias del fichero. Estas suelen ser cuatro en los archivos más simples, `FINAL TIME`, `INITIAL TIME`, `SAVEPER` y `TIME STEP`, aunque en archivos de gran tamaño existen gran cantidad de ecuaciones de este tipo, algunas definidas por el usuario y/o por *subscripts*. Estas variables están separadas del resto por dos líneas de asteriscos, que a través de la herramienta de `charposition` [28] se ha comprobado que ocupan 167 caracteres. Por lo tanto, la ecuación que ocurra inmediatamente antes de esta estructura, tendrá 167 caracteres adicionales.

Este programa está enfocado a crear el archivo `YAML` que representará al archivo de entrada, por lo que sus funciones principales serán escribir en un fichero la estructura general del árbol y con el fin de realizar la primera tarea, mantener la cuenta del número de línea y número de carácter en que se encuentra la lectura del fichero. Esto se realiza a través de dos variables principales que se van incrementando a lo largo del desarrollo del programa: `locationSpanStartEq`, la cual mantiene la cuenta de en qué línea del archivo se encuentra el lector; e `initCharEq`, la cual mantiene la cuenta de en qué carácter se encuentra el lector. Este parser inicial se basa en extraer el texto de cada ecuación para poder calcular las dos variables anteriores. Pese a ello, en algunos casos como espaciados distintos a lo que suele ser lo general, hacen necesario recalcular las mismas. En el siguiente fragmento de código se puede observar la escritura del fichero `YAML` de una iteración apoyándose tanto en las variables comentadas anteriormente como en otras variables utilizadas específicamente para la iteración. Posteriormente se actualizan las variables globales con las líneas y caracteres de la iteración.

```
fw.write("locationSpan : {start: [" + locationSpanStartEq + ", 0], end: ["
        + (locationSpanStartEq + equationNewLines + extraLocationSpan) +
        ", " + endColumnLocationSpan+ "]}\r\n");
locationSpanStartEq = locationSpanStartEq + equationNewLines + 1;

fw.write("span : [" + initCharEq + ", " +
    (endCharEq + initCharEq + extraCharsEq) + "]\r\n");
    initCharEq = initCharEq + endCharEq + 1;

locationSpanStartEq += extraLocationSpan;
extraLocationSpan = 0;
initCharEq += extraCharsEq;
extraCharsEq = 0;
```

La obtención del texto asociado a la ecuación incluyendo los espacios en blanco y saltos de línea debe hacerse a través de la clase `Interval`, y puede ser apreciado en el siguiente fragmento de código:

```
int a = equations.get(indexOfEquations).start.getStartIndex();
int b = equations.get(indexOfEquations).stop.getStopIndex();
Interval interval = new Interval(a, b);
String equation = ctx.start.getInputStream().getText(interval);
```

5.3. Desarrollo del segundo parser externo a partir de la gramática desarrollada.

Tras desarrollar el primer parser inicial, se aumentó su funcionalidad, basándose en la gramática completa. La estructura ahora pasaría a dividirse en cuatro hijos principales del nodo raíz *file*: *equations*, el cual es la parte desarrollada en la sección anterior y que parte de la gramática inicial [5]; *sketches*, el cual corresponde a las definiciones de las variables de las vistas en cuanto a formato; *graphs*, el cual corresponde a las definiciones de los gráficos; y por último *metadata*, el cual corresponde a aquellos campos finales que definen ciertos parámetros relativos al archivo en su conjunto.

También se mejoró la parte de *equations*, puesto que en el *parser* inicial siempre catalogaba las ecuaciones como *equation*, pero se mejoró para que fuese capaz de distinguir entre todos los tipos posibles, como *subscriptRange*, *lookupDefinition*...etc. Sin embargo, se presentó un problema al tratar de *parsear* las *macros*. Esto fue porque dichos clasificadores eran un hijo de *SymbolWithDoc* en la gramática de partida, sino que estaban a su misma altura. Se descartó modificar la gramática pues se consideró más complejo que modificar el código del *parser*. Se creó una bucle que tendría tantas iteraciones como elementos tuviesen la lista de *macros* y *SymbolWithDoc* conjuntamente. En cada iteración se comprobaría si el elemento

a observar es una definición de una *macro* y de serlo, se aplicarían ciertas modificaciones a las variables `locationSpanStartEq` e `initCharEq`. En el siguiente fragmento de código se muestra como se calcula un `ArrayList` de valores *booleanos* que identifican si el elemento del total de *macros* y *SymbolWithDoc* pertenece a una *macro*:

```
private ArrayList<Boolean> checkForMacros(String input) {
        ArrayList<Boolean> array = new ArrayList<>();
        try {
            String text = new String(Files.readAllBytes(Paths.get(input)),
            StandardCharsets.UTF_8);
            String[] noControl = text.split("\\\\\\---", 2);
            String[] equations = noControl[0].split("\\|");
            for (int i = 0; i < equations.length; i++) {
                array.add(equations[i].indexOf(":MACRO:") != -1);
            }
            for (int i = 0; i < 4; i++) { // TIME STEP equations
                array.add(false);
            }
        } catch (IOException ex) {
            ex.printStackTrace();
        }
        return array;
    }
```

En la parte de *sketches*, se definen a su vez una serie de contenedores o nodos, cada cual corresponde a una vista. A su vez, cada uno de estos sub-nodos tiene tantos hijos como variables y flechas tenga definidas, diferenciándose estas dos en el campo `type` del `YAML`. Se contemplaron aquellos casos donde el tercer campo de una variable tiene asignado el valor "cero" y su nombre viene definido en la siguiente línea, sin embargo, no se consideraron conceptos como las *shadow variables* y la relevancia de ciertos campos sobre otros debido a la complejidad del *sprint*. Sin embargo, dichas tareas quedaron fijadas para posteriores iteraciones.

De manera similar, dentro del nodo de gráficos, existen varios hijos asociados a cada gráfico. En el caso de los metadatos, cada línea es independiente y describe un parámetro concreto, por lo que son tratadas cada una como un hijo independiente.

Debido a que al intentar establecer los apartados de `headerSpan` y `footerSpan` en las secciones de *sketches* y *graphs* resultaba muy engorroso, se decidió crear un segundo formateador del texto. El problema surge de que otras secciones tienen delimitadores claros (como es el caso de *metadata*, donde la línea de : $L < \%\wedge E!@$ hace de claro delimitador para el `headerSpan` y la última línea en blanco del archivo hace de delimitador para el `footerSpan`), sin embargo estas dos no. Por ello, el nuevo formateador de texto tuvo como tarea añadir líneas de delimitado de inicio y final de dichas secciones: $< [VIEW/GRAPH\ \ START/END] >$ respectivamente. Esta labor fue fácil, ya que consistió simplemente en dividir el fichero en partes, introducir los delimitadores y volver a reconstruirlo.

No obstante, durante la integración tanto de este formateador como del anterior (`Comment`) surgió el problema de que `SemanticMerge` no era capaz de reconocer la estructura del parser, pese a aparentemente ser correcto. Después de realizar una serie de pruebas y confirmar la hipótesis de qué estaba causando el error con las ingenieras de `Códice Software`, se comprobó que el error era causado por que en la reconstrucción del `YAML` (explicado en detalle en el capítulo de Introducción) se compara el texto reconstruido con el archivo inicial. Sin embargo, el texto del archivo inicial se modificaba al añadir tanto los comentarios de las vistas en las ecuaciones como los delimitadores, por lo que al reconstruir, la estructura era distinta. La solución inicial propuesta fue realizar el formateo de los archivos en un programa distinto al parser externo.

Tras profundizar en una reunión con la tutora de este Trabajo de Fin de Grado, se acordó separar el programa en dos: uno inicial que cambiase el formato del documento y posteriormente el *parser* en sí mismo. Para poder realizar la resolución de conflictos correctamente pero aun así mantener una copia del archivo original a modo de *backup*, se decidió hacer una copia del archivo inicial y cambiarle bien el nombre o bien la extensión (utilizando una extensión propia para indicar que es un archivo de *backup*). El archivo modificado conservaría el nombre inicial para poder realizar el *merge* sin problemas.

Para poder mantener la coherencia de los archivos, se desarrolló un tercer programa cuya tarea fue eliminar aquellos delimitadores introducidos artificialmente para formar más fácilmente y con más claridad en el código el archivo `YAML`. El funcionamiento de este programa es muy simple: simplemente lee una a una todas las líneas del programa, si esta coincide con un delimitador, la ignora, y en caso contrario, la escribe en un fichero.

Tras comprobar que el *parser* funcionaba correctamente con archivos completos (es decir, con todas las partes que puede contener la gramática) y de extensión notable, se añadieron ciertas mejoras semánticas para mejorar la comprensión a la hora de realizar el *merge*. Concretamente, se añadió la posibilidad de distinguir entre variables convencionales, variables asociadas a gráficos y variables sombra o *shadow variables*. Esto se hace a través de comprobar si pueden llegar flechas a una variable para descartar las variables convencionales y posteriormente, se comprueba la `id` asociada al tipo de variable para descartar las variables asociadas a gráficos. Esto puede ser apreciado en el siguiente fragmento de código:

```
if (Integer.parseInt(viewVariablesList.get(viewVariablesIndex)
.bits.getText()) % 2 == 0) {
    if (Integer.parseInt(viewVariablesList.get(viewVariablesIndex)
    .internalId.getText()) == 10) {
        fw.write("      - type : shadow variable\r\n");
    } else {
        fw.write("      - type : graph variable\r\n");
    }
} else {
    fw.write("      - type : variable\r\n");
}
```

5.3.1. Refactor del visitor

Tras realizar un *parser* funcional que abarcase toda la gramática se procedió a dividir el *visitor* principal en varios. Se crearon tres *visitors* adicionales encargados de procesar las definiciones de ecuaciones, las definiciones de las vistas y las definiciones de gráficos y metadatos respectivamente. A mayores, se mejoró la documentación y legibilidad de dichas clases, y se tradujeron todos los comentarios útiles al inglés para aumentar su comprensión global.

En adición, se creó el paquete `utils`, el cual contiene todas las pequeñas funciones en las que se apoyan los *visitors*, como por ejemplo, funciones que calculan el número de líneas de un archivo o que diferencia las definiciones de ecuaciones que corresponden a *macros*.

5.4. Desarrollo de interfaces gráficas de las aplicaciones auxiliares

Tras desarrollar un *parser* funcional, se concluyó que se deberían realizar interfaces de usuario para los programas secundarios encargados de primeramente, introducir en el fichero los nombres de las vistas en cada ecuación y los delimitadores, y posteriormente, eliminar dichos delimitadores.

Dichas interfaces fueron desarrolladas de forma muy simple a través de la biblioteca de `Java Swing` [61], puesto que la funcionalidad de la misma es simplemente abrir un archivo a través del explorador de archivos. Ambas interfaces son prácticamente idénticas, por lo que en un principio se optó por utilizar distintos colores en el fondo de las mismas para evitar confusiones en su uso y con ello posibles errores. Por ello, la primera interfaz cuenta con un fondo amarillo claro y la segunda tiene el fondo gris predeterminado de la clase `JPanel`.

Las interfaces contaron en un principio con dos botones. El botón derecho servía para buscar el archivo, y es el que lanzaba el explorador de archivos. Al elegir un archivo cuya extensión no fuese `.mdl`, se notificaba al usuario de que el archivo seleccionado no tenía el formato de `Vensim` con una tipografía llamativa en color rojo. En caso de elegir un archivo correcto, se mostraba el nombre del archivo. Se notificaba de la misma manera si el usuario cancelaba la operación de elección de fichero en el explorador de archivos. El botón de la izquierda por su parte, tenía la función de procesar el archivo y ejecutar los programas correspondientes por debajo. Tras acabar la operación, se notificaba al usuario de que la operación se había realizado correctamente con una tipografía en color verde llamativo. Las interfaces debían de ser cerradas por el usuario, por lo que no finalizaban al procesar un archivo. Esto permitía que se pudieran procesar varios archivos en una misma ejecución de los programas.

Tras la primera prueba con usuarios se decidió modificar las interfaces para que fuesen más sencillas de usar para los usuarios. Se decidió cambiar la disposición de los botones a un único botón de abrir archivo inicialmente para posteriormente pasar a una pantalla que permitiese procesar el archivo o cancelar la selección para escoger de nuevo el archivo. A

mayores, se decidió eliminar el color amarillo de la segunda interfaz. Las etiquetas cambiaron su color a colores con menos contraste y se aumentó con creces el tamaño de la tipografía para ayudar a diferenciar los cambios a personas que pudiesen padecer daltonismo.

Además, se agregó un botón que permitía cambiar la etiqueta del archivo seleccionado entre su nombre simple y la ruta absoluta donde se encontraba el fichero en la máquina. De nuevo para mejorar la usabilidad, se agregaron iconos personalizados en cada una de las dos aplicaciones auxiliares para ayudar a diferenciarlas.

Tras la segunda prueba de aceptación de usuarios se localizaron varias posibles mejoras que se implementaron posteriormente. La primera fue restringir el explorador de archivos para mostrar únicamente los archivos `Vensim` con extensión `.mdl`. Tras ello, se decidió que sería más cómodo poder dejar abierta la herramienta en todos los *commits* realizados en vez de estar abriéndola y cerrándola cada vez. Por ello, y para evitar *bugs*, se añadieron las mismas comprobaciones de validez del archivo que se realizaban en el momento de abrir el archivo al momento en que se procesa el archivo. De forma que si se procesa un archivo dos veces seguidas, la segunda vez sería errónea y la herramienta no lo permitiría.

El resultado final de las interfaces puede ser contemplado en su totalidad en el anexo de **Manuales de Usuario**.

5.5. Diseño de la visualización de los tags de SemanticMerge y gMaster

A la hora de mostrar los cambios producidos durante un commit, `SemanticMerge` añade un *tag* entre corchetes a la izquierda del nombre del elemento modificado. Este *tag* corresponde a la etiqueta *type* que se genera en el árbol del fichero `YAML` correspondiente al archivo. Al contrario que en la visualización en formato textual (Figura 5.2), dichos *tags* sólo aparecen en la visualización en formato semántico (Figura 5.3).

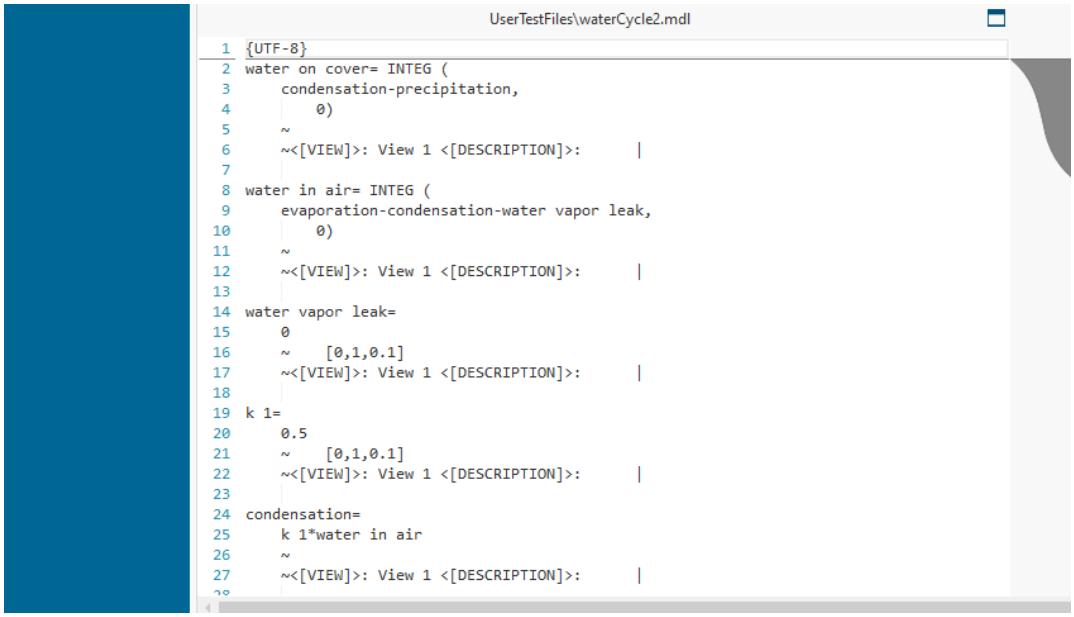

Figura 5.2: Esquema de cambios en el formato textual de gMaster

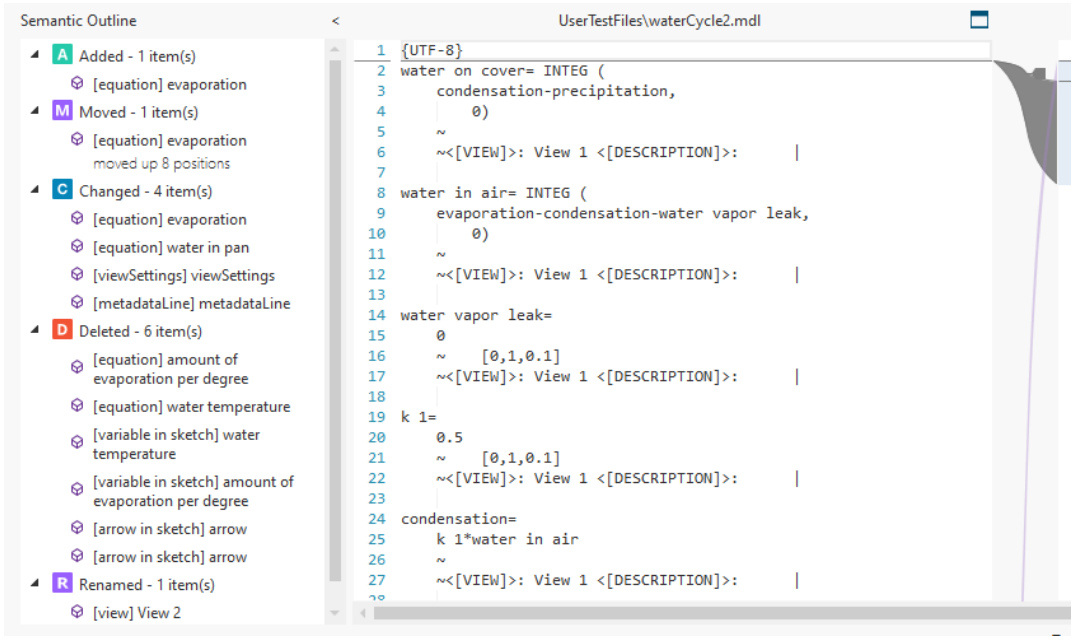

Figura 5.3: Esquema de cambios en el formato semántico de gMaster

Inicialmente se decidieron estructurar dichas etiquetas de una manera sencilla. En el caso de la primera parte, la referente a la definición de las ecuaciones, se utilizaron los ocho tipos

de variables definidas en la gramática de partida. En cuanto al nombre, se utilizó el propio nombre de la variable. En el caso de la definición de las vistas, fue más problemático, puesto que si bien existen variables que tienen su nombre definido en el tercer campo de sus líneas de definición, existen otros elementos como las *valves* que poseen un número propio de `Vensim` en ese campo. Inicialmente se les puso el nombre "Numeric", pero tras decidir que eran cambios triviales, se cambió el nombre a "Trivial". Sobre la etiqueta de dichos elementos, se obtuvo el tipo de variable a través del primer elemento en la línea de ecuación de cada elemento. Posteriormente y para distinguir estos elementos de las definiciones de ecuaciones, se agregó "*in view*" al *tag* de dichos elementos. Finalmente, en el caso de los gráficos, se indicó que eran gráficos en su etiqueta y se obtuvieron sus nombres a través de la propiedad `:NAME` de cada gráfico. En el caso de las líneas de metadatos, al ser elementos triviales, se decidió utilizar "*metadata*" en ambos nombre y *tag*.

Todos estos *tags* pueden ser apreciados con mayor claridad en la opción `Visual Diff` que permite la herramienta `gMaster`, donde como se aprecia en la Figura 5.4, todos los cambios se explicarán a través de dichas etiquetas y de los iconos propios de la herramienta `SemanticMerge` para indicar los cinco tipos de operaciones (adición, eliminación, movimiento, cambio y renombre).

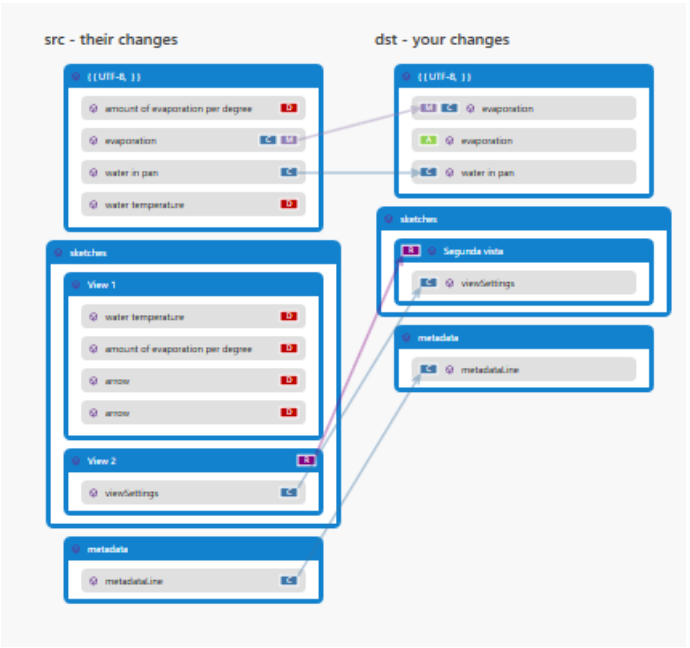

Figura 5.4: Esquema de cambios en el formato visual de gMaster

5.6. Diagramas de clases de diseño

A continuación se muestran los diagramas asociados a cada una de las tres herramientas, donde se muestran las clases utilizadas para su solución, así como sus principales atributos y operaciones. Comenzando por la herramienta principal del proyecto, el *parser*, se puede observar que comienza con un método principal `Main`, que llama a las distintas clases de `ANTLR4` para procesar el archivo leído a través de la gramática creada. Tras ello, se recorre dicho archivo a través de varios *visitors*, cada cual ocupándose de procesar una parte del fichero. Este diagrama se puede apreciar en la Figura 5.5:

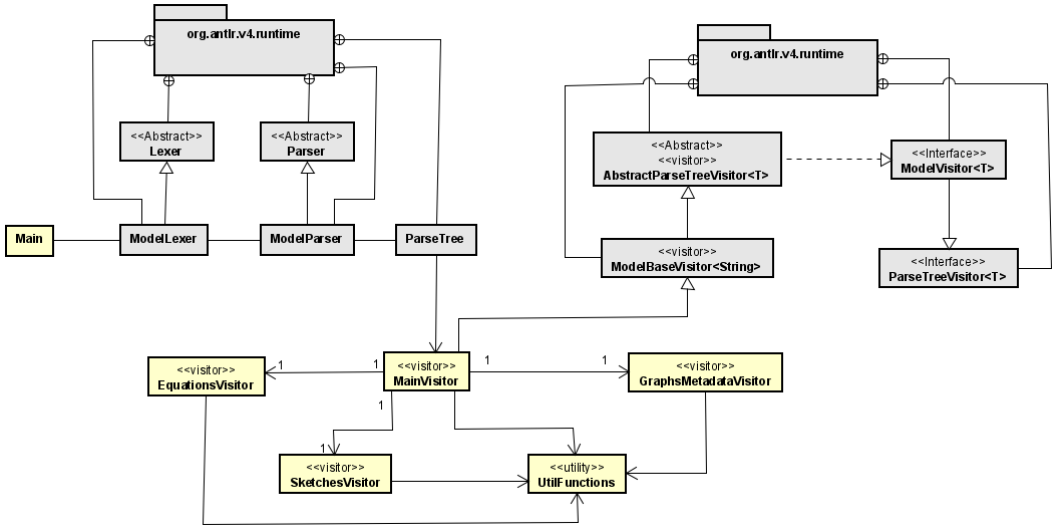

Figura 5.5: Diagrama de clases simple del parser externo para archivos Vensim.

El diagrama anterior se expone de forma simple al tener un potencial gran tamaño si se incluyesen métodos y atributos en las clases. Por ello, se muestra de forma separada el diagrama de diseño detallado de las clases desarrolladas en este proyecto a continuación, Figura 5.6:

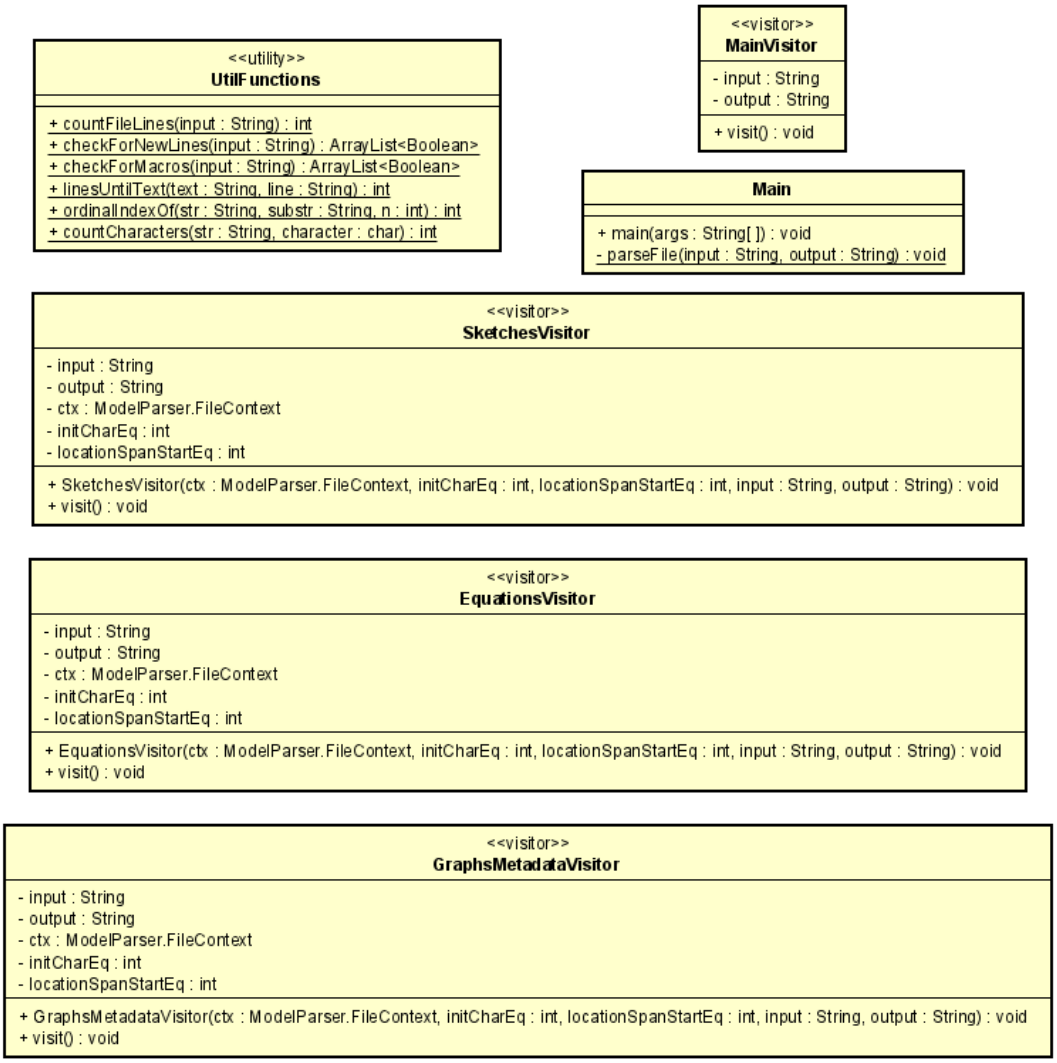

Figura 5.6: Diagrama de clases detallado del parser externo para archivos Vensim.

Por otra parte, en las Figuras 5.7 y 5.8, se pueden apreciar las clases que conforman las herramientas creadas para añadir y eliminar los nombres de las vistas y los delimitadores utilizados para *parsear* los archivos:

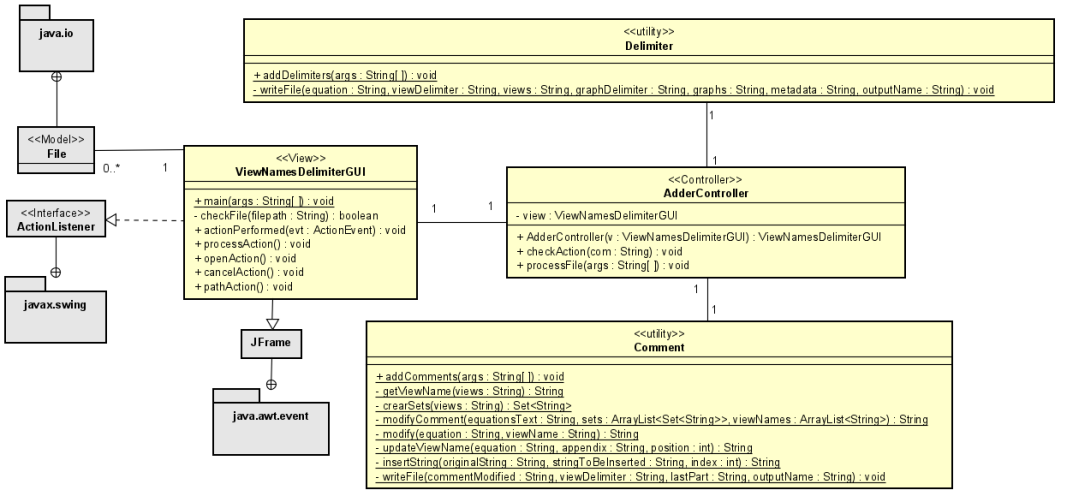

Figura 5.7: Diagrama de clases de la herramienta encargada de añadir los nombres de las vistas en las ecuaciones y delimitadores.

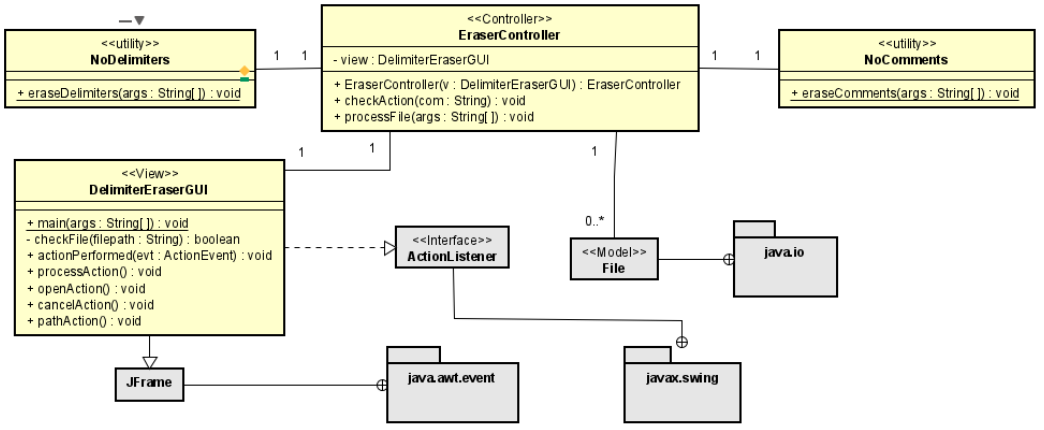

Figura 5.8: Diagrama de clases de la herramienta encargada de eliminar los nombres de las vistas en las ecuaciones y delimitadores.

Capítulo 6

Implementación y pruebas

Este capítulo está dedicado a explicar la implementación de ciertas herramientas en el proyecto, así como todas las pruebas realizadas durante el transcurso del proyecto. Las pruebas de *testing* han sido divididas en diferentes categorías, detallando en las primeras secciones pruebas específicas para cada programa desarrollado, y explicando en las secciones finales las pruebas de aceptación de usuarios.

6.1. Implementación de la integración continua y el despliegue continuo

Como se explicó en el capítulo de **Tecnologías Utilizadas**, tanto la integración continua como en despliegue continuo se realizaron a través de las herramientas de `GitLab`. La herramienta permite realizar estas actividades a través de un fichero denominado `.gitlab-ci.yml`. Respecto a la integración continua, es proceso es bastante sencillo, ya que al estar utilizando un entorno `Maven` en el proyecto, `GitLab` se limita a recrear los *tests* realizados en la máquina local y los lanza en su propia máquina virtual, comprobando que las pruebas no son correctas únicamente en la máquina local.

Respecto al despliegue continuo, a través de ese mismo archivo se conecta con una máquina virtual proporcionada por la Escuela de Ingeniería Informática. Dicha máquina tiene instalados los programas necesarios para ser capaz de lanzar el *parser* en formato `JAR` y comprobar que funciona correctamente en otras máquinas. Para ello, se instaló en la máquina virtual la herramienta `Cygwin` [42], la cual permitía realizar comunicaciones `ssh` entre dicha máquina y `GitLab`. Para garantizar la seguridad del usuario y no exponer al público sus credenciales, la contraseña de la máquina virtual fue almacenada como variable de entorno de `GitLab`.

En cuanto a la comprobación del funcionamiento en la máquina virtual desde `GitLab`, se comprobó que en caso de fallo no se lanzaba el programa en `SemanticMerge`, por lo que lo más óptimo fue lanzar la herramienta en remoto, esperar un periodo de tiempo y probar a

terminar el proceso mediante un comando `kill`. Si esto era correcto, el despliegue continuo habría funcionado correctamente.

Concretamente en este trabajo se utilizó una máquina virtual con una imagen de Windows 10, en la que estaba instalada una copia de SemanticMerge con el objetivo de probar el despliegue del *plugin* generado en forma de archivo `jar`.

Ambas acciones se describen en el fichero `.gitlab-ci.yml` situado en el directorio raíz del proyecto. En este fichero se describen las diferentes etapas por las que pasará el proyecto tras actualizarse mediante *commit* & *push*. En estas etapas se encuentran acciones asociadas tanto a la integración continua como al despliegue continuo. Estas etapas pueden ser también catalogadas como tuberías o *pipelines*, definiéndose en cada una de estas una serie de tareas. Para comenzar una etapa, todas las tareas de la etapa anterior deben haber sido completadas. En el siguiente fragmento de código se código se muestra el archivo `YAML` utilizado para esta tarea en este proyecto:

```
    image: maven:3.6.3-jdk-11

variables:
  MAVEN_CLI_OPTS: "--batch-mode"
  MAVEN_OPTS: "-Dmaven.repo.local=.m2/repository"

before_script:
  - 'apt-get update && apt-get install sshpass'

stages:
  - build
  - test
  - deploy


cache:
  paths:
    - .m2/repository
    - target/

build_job:
  stage: build
  script:
    - mvn $MAVEN_CLI_OPTS  compile

test_job:
  stage: test
  script:
    - mvn $MAVEN_CLI_OPTS test -P Unit

deploy-master:
```

```
  stage: deploy
  script:
    - mvn assembly:assembly
    - sshpass -p "$password" scp -o UserKnownHostsFile=/dev/null -o
    StrictHostKeyChecking=no -P 20012
    target/mvntfg-1.0-jar-with-dependencies.jar
    usuario@virtual.lab.inf.uva.es:C:\Users\Usuario\AppData
    \Local\semanticmerge
    - sshpass -p "PASSWORD" ssh -o UserKnownHostsFile=/dev/null -o
    StrictHostKeyChecking=no
    usuario@virtual.lab.inf.uva.es -p 20012
    "cd /cygdrive/c/Users/Usuario/AppData/Local/semanticmerge/ &&
    ./script.bat && exit"
  only:
    - master
####
deploy-develop:
  stage: deploy
  script:
    - mvn assembly:assembly
    - sshpass -p "$password" scp -o UserKnownHostsFile=/dev/null -o
    StrictHostKeyChecking=no -P 20012
    target/mvntfg-1.0-jar-with-dependencies.jar
    usuario@virtual.lab.inf.uva.es:C:\Users\Usuario\AppData
    \Local\semanticmerge
    - sshpass -p "$password" ssh -o UserKnownHostsFile=/dev/null -o
    StrictHostKeyChecking=no usuario@virtual.lab.inf.uva.es -p 20012
    "cd /cygdrive/c/Users/Usuario/AppData/Local/semanticmerge/ &&
    ./script.bat && exit"
  only:
    - develop
```

En la etapa de `test` se comprueban todos los tests asociados al proyecto. Esto se hace a través de la categorización de los tests que `Maven` provee. La etapa de `deploy` está asociada al despliegue continuo. En esta etapa se envía el fichero `jar` a la máquina virtual mediante `scp` y después de accede a ella mediante `ssh` para ejecutar un *script* que lanzará `SemanticMerge` para comprobar la validez del *parser*. En este comando podemos ver que se utiliza la variable de entorno `$password`, la cual hace referencia a la contraseña de la máquina virtual y que por motivos de seguridad no se expone en el fichero al ser el repositorio público. La variable de entorno está asociada al usuario de `GitLab` y puede ser modificada desde la página web. El *script* utilizado en la máquina virtual es el siguiente:

```
    START "" .\semanticmergetool.exe --source=Bunny1.mdl
    --destination=Bunny2.mdl --externalparser="-jar
    mvntfg-1.0-jar-with-dependencies.jar"
```

```
--virtualmachine="C:\Program Files\java\jdk-11.0.8\bin\java.exe" &&
sleep 10 && taskkill -IM semanticmergetool.exe
```

Los archivos *Bunny1.mdl* y *Bunny2.mdl* son archivos de prueba muy simples cuyo único fin es comprobar si se ha conseguido desplegar correctamente el *parser*. Como se puede apreciar, el *script* lanza el programa, espera 10 segundos e intenta finalizar el programa. En caso de fallo, no se realizarán los últimos dos comandos y el despliegue continuo fallará debido a un *timeout*. Este mecanismo debe de realizarse así puesto que las alertas de fallo se realizan por la interfaz gráfica, y no se consiguió acceder a ellas a través de la línea de comandos.

6.2. Implementación de las pruebas de eficiencia realizadas sobre el parser

En cierto punto del desarrollo del programa se comprobó que para archivos de una muy gran extensión, como puede ser 40.000 líneas de código, la ejecución del programa no llegaba a finalizar nunca, por lo que se trató de analizar la eficiencia del programa.

Concretamente se realizaron pruebas con ficheros de distinto tamaño. Primeramente se utilizaron ficheros de pequeño tamaño, con apenas 300 líneas y un tamaño entre 2 y 10 KBs. El procesamiento de estos ficheros fue instantáneo. Aumentando cuantiosamente el tamaño de los ficheros, con alrededor de unas 20.000 líneas y un tamaño de aproximadamente 550 KBs, el procesamiento del fichero se demoraba unos pocos segundos, pero no más de 5 o 6. Finalmente, utilizando un fichero de gran tamaño, concretamente unas 40.000 líneas y un tamaño de casi 2 MB, el fichero no consiguió procesarse, puesto que se cortó la ejecución de dicho procesamiento al cabo de aproximadamente treinta minutos. Este problema se encontró a lo largo del séptimo *sprint* y se arrastró el problema durante varios *sprints* consecutivos, en cada uno de ellos probando nuevas formas de analizar la eficiencia del código realizado.

Inicialmente se examinó en profundidad el código y pese que sí se encontraron ciertos puntos que podrían mejorar, no se encontró ninguna estructura que pudiera estar causando semejante consumición de tiempo. Tampoco se detectó ninguna clase de bucle infinito o problema similar donde el código pudiese encontrar un *deadlock* o quedarse atascado. Tras eso, se decidió utilizar una herramienta que analizase la eficiencia del código o *profiler*, eligiendo en este caso la herramienta `JProfiler` [57]. Sin embargo, tras configurar adecuadamente sus parámetros, no se detectó ningún problema en el código que pudiese estar causando problemas de eficiencia.

En cierto momento del desarrollo del proyecto se decidió utilizar `SonarQube` [49] para comprobar si existían brechas de eficiencia en el código. En dicha prueba se detectaron varios fallos de calidad asociados al tratamiento de ficheros y ciertas repeticiones de código asociadas a la escritura del árbol `YAML`, sin embargo, no eran problemas de eficiencia. Respecto a la complejidad cognitiva, se detectaron dos clases respectivas a los *visitors* que presentaban una gran complejidad por encima de lo permitido por la herramienta. No obstante, al revisar dichas clases, se comprobó que muchas de las cláusulas que aumentaban la complejidad eran

bloques *if-else* destinados a categorizar las estructuras de código otorgándolas tipo y nombre, en ocasiones, entre hasta ocho posibilidades. No se detectaron otros posibles problemas de eficiencia.

En cierto punto del proyecto se realizó una reunión con desarrolladores de la empresa `Plastic SCM` para analizar la eficiencia del proyecto, pues con su experiencia podrían ayudar a resolver los problemas. Sin embargo, se descubrió que el problema no era una cuestión de eficiencia, sino que existía un error invisible. Al transformar el programa para ser utilizado en `gMaster`, se cambió el funcionamiento del mismo de aceptar un número concreto de archivos a utilizar espera activa para utilizar un número ilimitado de archivos. No obstante, esto provocaba que si el fichero fallaba en el reconocimiento de la gramática, el fallo pasara desapercibido. El fallo de *parseo* se solucionó de forma sencilla, con dos líneas de código. Por otro lado, se transformó la clase principal del programa para que realizase un bloque *try-catch* en cada iteración o archivo procesado. Esto no solucionó el problema de no visualizar el error en la interfaz gráfica de `SemanticMerge`, pero hizo que el proceso de *debugging* fuese mucho más eficiente.

Durante el proceso de comprobar que el archivo de 40.000 líneas fuese correcto se encontraron varios nombrados de variable problemáticos que hacían que el *parser* no funcionase correctamente. Estos nombrados se explican en el Anexo de **Manuales de Usuario**.

6.3. Pruebas referidas a la automatización de la inserción de los nombres de las vistas en los comentarios de las ecuaciones

El proceso de *testing* fue muy similar al de la gramática, ya que se probaron la mayoría de archivos que se utilizaron para comprobar que esta era correcta. A mayores, se realizaron una serie de tests unitarios comprobando que el número de líneas del fichero no variaba y que las escrituras se realizaban correctamente y en el lugar que se debía. Los escenarios probados fueron los siguientes:

- Escenario simple básico.
- Escenario que contiene una ecuación que no aparece definida en el apartado de las vistas.
- Escenario donde las ecuaciones tienen comentarios previos.
- Escenario donde existen varias vistas.
- Escenario donde las ecuaciones tienen caracteres especiales como paréntesis, corchetes o barras bajas.
- Escenario donde el archivo ya ha sido procesado una vez y contiene los delimitadores `<[VIEW]>:` y `<[DESCRIPTION]>:`.

- Escenario donde el fichero a procesar no existe.

A continuación se muestra el ejemplo del test del escenario más básico posible, donde se comprueba que el número de líneas no ha sido modificado y que los comentarios han sido introducido en las líneas correctas:

```
@Test
    public void lecturaCorrectaSimple() {
        String[] args = new String[2];
        args[0] = "VensimExampleModels/SHODOR/Bunny.mdl";
        args[1] = "outputs/comment/test1.mdl";
        BufferedReader reader;
        try {
            Comment.main(args);
            reader = new BufferedReader(new FileReader(
            "VensimExampleModels/SHODOR/Bunny.mdl"));
            int lines = 0;
            while (reader.readLine() != null)
                lines++;
            reader.close();
            assertSame(127, lines);
            // Last newline is not counted

            reader = new BufferedReader(new FileReader(
            "outputs/comment/test1.mdl"));
            Set<Integer> positions = new HashSet<Integer>();
            positions.addAll(Arrays.asList(new Integer[] {
            4, 9, 15, 21, 26, 31, 36, 41 }));
            for (int i = 0; i < 43; i++) {
                String line = reader.readLine();
                if (positions.contains(i)) {
                    assertTrue(line.indexOf("View 1") != -1);
                }
            }
            reader.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

En el caso de este primer programa, el proceso de *testing* fue bastante rápido puesto que todos los casos probados funcionaron correctamente en la primera prueba.

6.4. Pruebas referidas al primer parser basado únicamente en la gramática de partida

La gramática de partida sólo tenía en consideración la parte de los ficheros `.mdl` dedicada a la definición de ecuaciones, por lo que no existen tests orientados a los datos definidos en las vistas y los gráficos. Se tuvo la limitación de que la herramienta utilizada para contar tanto las líneas como los caracteres de los archivos, `charposition` [28], no era capaz de leer archivos de gran tamaño, limitando el número de líneas a aproximadamente 2500. Aun así, se pudieron hacer muchas comprobaciones:

- Escenario simple básico.
- Escenario que contiene una ecuación que no aparece definida en el apartado de las vistas y con un espaciado diferente por ello.
- Escenario con palabras clave utilizando guiones bajos.
- Escenario en el que se incluyen definiciones de *subscript ranges*.
- Escenario en el que se incluyen definiciones de tablas o *lookups*.
- Escenario en el que se incluyen definiciones de *macros*.
- Escenario en el que se incluyen definiciones de *data equations*.
- Escenario en el que se incluyen definiciones de *unchangeable constants*.
- Escenario en el que se incluyen definiciones de *constraints*.
- Escenario donde `TIME STEP` no es la última ecuación declarada.

Pese a que no se encontraron archivos de tamaño lo suficientemente pequeño para probar ciertos casos como *reality checks*, se asume que son correctos al utilizar la misma sintaxis que las ecuaciones, los *subscript ranges* o las definiciones de *lookups*. Inicialmente, los tests consistieron en recorrer línea a línea el archivo e ir comprobando ciertas líneas críticas que proporcionaban información sobre si el archivo `YAML` se había generado correctamente. Esto ocasionó un problema con la integración continua en `GitLab`, puesto que `SemanticMerge` es una herramienta desarrollada para Windows, por lo que utiliza el espaciado correspondiente a este sistema operativo, `CRLF`. Sin embargo, `GitLab` utiliza el espaciado convencional de Linux, `LF`. Esto provocó que los tests asociados al *parser* funcionasen en local pero no en la integración continua. Por el momento, dicho problema se aceptó y se buscaría una solución en *sprints* posteriores del proyecto.

Este proceso de *testing* fue bastante sencillo de nuevo, obteniendo problemas únicamente en el último escenario. Esto se solucionó contemplando todas las ecuaciones en vez de parar de comprobar al reconocer `TIME STEP` como se hacía antes. Si bien en la gran mayoría de archivos de `Vensim` se utilizan siempre las mismas cuatro variables como variables de control de la simulación, en archivos de mucha mayor longitud, es posible encontrar variables definidas manualmente en este área.

Finalmente, con el objetivo de probar si el *parser* era aceptado por `SemanticMerge`, en cada test se introdujo un comando que lanzaba el `jar` del proyecto como *parser*, con el archivo que se estaba probando tanto como *source* como *destination*. Sin embargo, para permitir que los tests siguiesen siendo fluidos, este comando permanece siempre comentado salvo en las ocasiones que se quiera probar un fichero concreto. El comando se expone en el siguiente fragmento de código, utilizando el archivo del escenario base:

```
Process process = Runtime.getRuntime()
    .exec("C:\\Users\\Propietario\\AppData\\Local\\semanticmerge.
    \\semanticmergetool.exe"
     + " --source=Formatted/BunnyFormat.mdl
     --destination=Formatted/BunnyFormat.mdl
     --externalparser=\"-jar target/mvntfg-1.0-jar-with-dependencies.jar\""
    + " --virtualmachine=\"C:\\Program Files\\Java\\
    jdk-11.0.8\\bin\\java.exe\"");
```

6.5. Pruebas referidas al segundo parser basado en la gramática desarrollada

El segundo parser incluye la definición de las vistas y sus variables asociadas, los gráficos y los metadatos del programa. Como es una ampliación del *parser* desarrollado y explicado en la sección anterior, el *testing* realizado incluye los expuestos en la sección anterior. Se incluyen a mayores los siguientes casos de pruebas:

- Escenario simple básico.
- Escenario donde no se utilizan gráficos.
- Escenario donde se utilizan múltiples vistas.
- Escenario donde se utilizan múltiples gráficos.

Tras desarrollar los casos base, se probaron varios archivos de gran longitud para ver si el *parser* era capaz de procesarlos, y una vez procesados, capaz de ser utilizados por la herramienta de `SemanticMerge`. Se comenzó probando el archivo que se utilizaba como modelo base en el proyecto `LOCOMOTION`, el archivo `WILIAM.mdl`. Sin embargo se encontró que pese a que los ficheros `.mdl` suelen utilizar la codificación de texto `UTF-8` (quedando indicado en la primera línea del fichero), dicho fichero no la utilizaba, al incluir palabras con caracteres especiales como 'ñ' o 'á'. Sin embargo, dicho fichero tambíen tenía la declaración de que utilizaba el estándar de `UTF-8`, por lo que se anotó como incidencia para tratar con los desarrolladores de `LOCOMOTION`.

Probando otros ficheros se descubrieron y corrigieron dos *bugs* que aparecían por casos no contemplados. El primero de ellos ocurría porque el *parser* sólo contemplaba ecuaciones

definidas de forma compactada (sin utilizar la estructura por líneas para declarar la ecuación, las unidades y el comentario, y haciéndolo todo seguido en una única línea). Esto se debe a que en la mayoría de ficheros de prueba, las variables que controlaban parámetros de la simulación eran siempre las mismas y estaban en el mismo formato (`TIME STEP`, `FINAL TIME...`), sin embargo, se vio que en archivos de mayor complejidad se pueden añadir otro tipo de variables externas que es necesario contemplar. El segundo *bug* que se trató fue más costoso de encontrar, y es que `ANTLR4` elimina los espacios que pueden existir detrás de una coma, por lo que el `span` utilizado para contar los caracteres de las distintas partes del programa podía llegar a acumular una gran diferencia en un fichero con gran cantidad de comas. Se arregló para el caso de las comas, y se marcó como incidencia el buscar otros caracteres que produjesen este *bug*.

Tras la reunión se seguimiento asociada a este *sprint* se acordó que sería conveniente modificar los tests de forma que en vez de recorrer el archivo y comprobar las líneas críticas, se hiciese directamente un *diff* entre el archivo generado y una copia local y se comprobase que los archivos fuesen idénticos. Esto haría que los tests fuesen más sencillos y mucho más fáciles de comprender. Además, al realizar un *diff* en vez de comprobar el número de caracteres, se solucionó el problema producido por la diferencia en los saltos de línea entre Windows y Linux que ocasionaba `GitLab`. A continuación podemos observar un fragmento de código con el test utilizado para probar el escenario más simple:

```
@Test
public void lecturaCorrectaSimple() {
    try {
        CharStream charstream = CharStreams.fromFileName(
            "Formatted/BunnyFormat.mdl");
        ModelLexer lexer = new ModelLexer(charstream);
        ModelParser parser = new ModelParser(new CommonTokenStream(lexer));
        ParseTree tree = parser.file();

        EvalVisitor visitor = new EvalVisitor();
        visitor.setInput("Formatted/BunnyFormat.mdl");
        visitor.setOutput("outputs/evalVisitorTest/test1EvalVisitor.yml");
        visitor.visit(tree);

        Runtime rt = Runtime.getRuntime();
        Process proc = rt.exec(
            "powershell.exe compare-object
            (get-content outputs/evalVisitor/test1EvalVisitor.yml)
            (get-content outputs/evalVisitorTest/test1EvalVisitor.yml)");
        proc.waitFor(5, TimeUnit.SECONDS);

        BufferedReader br = new BufferedReader(
            new InputStreamReader(proc.getInputStream(), "UTF-8"));
        String result = br.lines().collect(Collectors.joining("\n"));
        br.close();
        assertEquals("", result);

    } catch (Exception ex) {
        System.err.println("Error en el proceso de diff: "
            + ex.getMessage());
    }
}
```

6.6. Corrección de bugs del parser

En el proceso de preparación de las pruebas de aceptación de usuarios, se intentó trabajar con un archivo pero este falló en el *parseo*. Tras un proceso de investigación se descubrió que el método `getText()` de `ANTLR4` que se utilizaba para recuperar los nombres de las vistas, ecuaciones y similares no recogía correctamente los espacios, por lo que los archivos `YAML` no se formaban correctamente en estos casos.

Esto se solucionó utilizando la clase `Interval`, la cual marca un punto de inicio y un punto de fin en la cadena que se lee, en este caso el fichero `Vensim`, y lo recupera sin realizar ninguna modificación. Esto se puede apreciar en el siguiente fragmento de código:

```
int intvViewLineName1 = viewVariablesList.get(viewVariablesIndex)
            .visualInfo().start.getStartIndex();
int intvViewLineName2 = viewVariablesList.get(viewVariablesIndex)
            .visualInfo().stop.getStopIndex();
Interval intervalViewLineName = new Interval(intvViewLineName1,
            intvViewLineName2);
fw.write("        name : "+ ctx.start.getInputStream()
            .getText(intervalViewLineName)
        + "\r\n");
```

En este proceso de *debugging* se confirmó que en los archivos `Vensim` a procesar no se podrían utilizar caracteres especiales fuera del alcance del sistema de codificación `UTF-8`.

Sin considerarlo realmente un *bug*, al realizar la primera prueba de aceptación de usuarios se vio que la mayoría de usuarios utilizaban la versión 8 de `Java` y la herramienta se había desarrollado en la versión 11, por lo que se tuvo que migrar entre versiones. Sin embargo, la migración solamente afectó a una función auxiliar donde se utilizaba el método `isBlank()` exclusivo de `Java` 11. Dicho método se cambió por la concatenación de métodos `trim().isEmpty()` y la migración se completó correctamente.

6.7. Pruebas de Aceptación de Usuarios

6.7.1. Contextualización

Tras haber desarrollado un *parser* funcional y las correspondientes interfaces de usuario para usuarios no familiarizados con los entornos de programación y/o las herramientas de control de versiones, se procedió a realizar varias pruebas para comprobar la satisfacción de los usuarios y detectar posibles fallos o *bugs* que hubieran pasado desapercibidos.

El proceso de la prueba comenzó con una explicación a los usuarios sobre el funcionamiento del proyecto en líneas generales. En esta explicación se explicaron cada una de los tres programas a probar, las relaciones entre sí de dichos programas y en general cómo transcurriría la prueba.

Posteriormente se entregó a los usuarios un formulario que deberían cumplimentar. Al comienzo de dicho formulario, los usuarios encontrarían un enlace que les llevaría a un repositorio externo. Dicho enlace contenía el archivo `Vensim` con el que se realizarían las pruebas del proyecto así como los manuales de usuario necesarios, concretamente, de instalación y de uso. Tras obtener todos los archivos, los usuarios realizaron todos los escenarios, los cuales estaban detallados en dicho formulario y que se describen en las subsecciones siguientes de esta memoria. Por cada escenario, los usuarios valorarían la dificultad y el tiempo. Además, pudieron proporcionar *feedback* sobre posibles mejoras o *bugs* encontrados en la prueba. El alumno responsable de este Trabajo de Fin de Grado estaría presente en la realización de dichos escenarios, tomando nota acerca de la realización de los mismos y aportando pequeñas aclaraciones a los usuarios.

Finalmente, se encuestó a los usuarios sobre qué competencias habían sido capaces de desarrollar durante el transcurso de la prueba. Dichas competencias se encuentran en subsecciones posteriores.

Se diseñaron distintos escenarios, los cuales se describen a continuación, para probar que la experiencia de usuario fuese óptima. Dichos *tests* han sido divididos en las interfaces para procesar los archivos `Vensim` y cambiar su formato, y los dedicados a probar el *parser* externo en la herramienta de `gMaster`.

6.7.2. Escenarios correspondientes a las interfaces de procesado de ficheros Vensim

Existen dos interfaces para procesar los archivos, sin embargo, estas son prácticamente idénticas, por lo que las pruebas descritas en esta sección serán válidas para sendos programas. Los escenarios a realizar se muestran en la Tabla 6.1:

Número de escenario	**Descripción**
Escenario 1	Seleccionar un archivo Vensim y procesarlo con el programa correspondiente para añadir los delimitadores y los nombres de las vistas. Tras ello, comprobar que los archivos se han modificado correctamente. Utilizar el otro programa para eliminar los delimitadores y de nuevo comprobar que los archivos se han modificado correctamente.
Escenario 2	Seleccionar un archivo no válido y tratar de procesarlo con el programa correspondiente para añadir los delimitadores y los nombres de las vistas. Realizar este mismo proceso con el programa que elimina los delimitadores.

Tabla 6.1: Escenarios a realizar en las interfaces de procesado de ficheros.

6.7.3. Escenarios correspondientes al parser externo en gMaster

El *parser* externo para archivos `Vensim` es el grueso del proyecto, y por tanto sus pruebas de aceptación de usuario serán mayores y más complejas. Antes de todo, es conveniente estar familiarizado con la herramienta de `gMaster`, la cual es descrita brevemente en el capítulo de **Introducción** de este documento. Además, se deben tener en cuenta varios aspectos antes de comenzar a probar cada uno de los escenarios:

- Este *parser* está diseñado para trabajar con archivos ya procesados e introducidos los nombres de las vistas y los delimitadores. El programa no detectará de forma **semántica** archivos no *parseados*.
- Por el mismo motivo, no deben modificarse o eliminarse dichos delimitadores durante las pruebas de usuarios.
- Añadir sentencias erróneas o modificar las ya existentes para que no respeten la estructura de Vensim, provocará que el *parser* no sea capaz de procesar **semánticamente** los archivos.
- No se permiten caracteres especiales fuera del abanico permitido por el estándar `UTF-8`.
- No se permite utilizar el carácter ':' seguido de un espacio por motivos de compatibilidad con el formato `YAML`.

Antes de realizar los distintos escenarios, se deberán subir los archivos iniciales que se deseen probar a un repositorio, descargarse el fichero `JAR` y añadirlo como extensión externa a la herramienta `gMaster`. Este proceso es descrito detalladamente tanto en la sección dedicada a `gMaster` en el capítulo de **Introducción** como en el capítulo de **Manuales de Usuario**. Para simplificar las descripciones, los elementos catalogados en la gramática como `SymbolWithDocDefinition` serán tratados como ecuaciones. Estos escenarios pueden realizarse bien realizando *merges* en la misma rama, o entre ramas distintas.

Es conveniente conocer la secuencia a seguir a la hora de trabajar con los ficheros. Partiendo de un repositorio vacío, primeramente se deben tomar los archivos que se quieran comparar y procesarlos a través de la primera interfaz gráfica. Tras ello, se realizará un *commit* con dichos archivos y se subirán al repositorio. Esta será la versión inicial. Tras ello, se tomará el archivo modificado y se aplicará la segunda interfaz gráfica para eliminar los delimitadores y poder abrir el archivo `Vensim` en formato gráfico. Se aplicarán los cambios pertinentes y tras ello, de nuevo se utilizará la primera interfaz y se subirán todos los nuevos cambios al repositorio.

Los escenarios realizados se crearon a través de trabajar con varios archivos `Vensim` de progresiva dificultad. En la primera versión entregada a los usuarios, a día 4 de febrero de 2021, se utilizó un archivo con una complejidad media, con aproximadamente 1.800 líneas. Los escenarios a realizar se enuncian en la Tabla 6.2.

Las competencias que se desea que el usuario obtenga en estos escenarios se muestran en la Tabla 6.3.

Número de escenario	Descripción
Escenario 1	Crear una rama para comenzar a realizar los cambios que proponen estos escenarios. Realizar cambios significativos (como modificar los valores de la ecuación) y triviales (como cambiar la posición y/o el color) sobre una o varias variables.
Escenario 2	Añadir y/o eliminar un gráfico o varios.
Escenario 3	Crear una *shadow variable* a partir de otra principal.
Escenario 4	Modificar los parámetros del archivo.
Escenario 5	Crear una nueva vista.
Escenario 6	Mover variables y/o flechas entre vistas.
Escenario 7	Modificar el nombre a una vista.
Escenario 8	Eliminar un conjunto de variables y/o flechas.
Escenario 9	Eliminar una vista.
Escenario 10	Hacer merge a master de los cambios realizados en los escenarios anterior.

Tabla 6.2: Escenarios a realizar en el parser externo a través de gMaster.

Tabla 6.3: Competencias a adquirir en los escenarios

Número de competencia	Descripción
Competencia 1	El usuario debe de ser capaz de distinguir entre tipos de SymbolWithDoc en gMaster.
Competencia 2	El usuario debe de ser capaz de identificar en qué vista se sitúan las variables.
Competencia 3	El usuario debe de ser capaz de distinguir entre variables normales y variables *shadow*.
Competencia 4	El usuario debe de ser capaz de distinguir entre variables y flechas.
Competencia 5	El usuario debe de ser capaz de comprender qué parte se puede modificar del comentario y cuál no.
Competencia 6	El usuario debe distinguir que cambios son triviales y cuales no en las definiciones de ecuaciones en la sección de cada vista.
Subcompetencia 6.1	Los valores 4, 5, 6 y 7 en dicha definición de variables representan posición, anchura y altura. Son valores triviales.
Subcompetencia 6.2	Los valores finales de las variables en la gran mayoría de casos son triviales.

Continúa en la página siguiente

Tabla 6.3 – *Viene de la página anterior*

Número de competencia	**Descripción**
Subcompetencia 6.3	El noveno campo de las variables puede utilizarse para conocer si una variable es *shadow variable* o no. Este es un número que representa a una serie de bits. El bit con índice cero representa si pueden entrar flechas a la variable, cosa que no está permitido en las *shadow variables*. Por tanto, si el número es impar será variable común.
Subcompetencia 6.4	En el caso de ser un número par, si el primer número de la definición es 10, será una *shadow variable*. En caso contrario, será una variable asociada a un gráfico.
Subcompetencia 6.5	Las variables suelen comenzar por el número 10 y las flechas por 1. Este es su identificador de tipo.
Subcompetencia 6.6	En el caso de las definiciones de las flechas, en la gran mayoría de los casos sólo son relevantes los cinco primeros campos.
Competencia 7	El usuario debe de ser capaz de distinguir que cuando se cambia el tipo de una variable, se produce una inserción y un borrado, y no un cambio.
Competencia 8	El usuario debe comprobar que al cambiar un nombre a una vista, los comentarios de las ecuaciones se actualizan correctamente.
Competencia 9	El usuario debe de ser capaz de distinguir inserciones y borrados de grandes cantidades de variables y/o flechas.
Competencia 10	El usuario debe de ser capaz de distinguir que cuando se mueve una variable de vista, esta se elimina en una vista y se añade en otra en la parte de ecuaciones. Sin embargo, en la parte de definiciones de vista aparecerá como modificada.
Competencia 11	El usuario debe de ser capaz de distinguir que al borrar una vista, esta aparece como eliminación en su completo, y no cada una de sus partes.
Competencia 12	El usuario debe de ser capaz de diferenciar que al borrar una vista se conservan los gráficos definidos en ella en el apartado de gráficos.
Competencia 13	El usuario debe de ser capaz de localizar los cambios que ha hecho entre cambios que haya podido realizar el *parseo*, como saltos de línea.
Competencia 14	El usuario debe de ser capaz de diferenciar los cambios importantes de los triviales.
Competencia 15	El usuario debe de ser capaz de identificar los conjuntos de elementos que ha borrado.
Competencia 16	El usuario debe de ser capaz de identificar los cambios en los nombres de las vistas.

6.7.4. Formulario para las pruebas de aceptación de usuario.

En dichas pruebas de usuario, se le proporcionó al usuario un formulario para que aporte datos sobre su experiencia con la aplicación en diversos puntos así como sugerencias de mejora. Se comenzó encuestando a los usuarios sobre la dificultad y el tiempo transcurrido en cada uno de los escenarios por separado. Además se preguntó a los usuarios sobre fallos o posibles mejoras en dichos escenarios. Para cada escenario se formularon las siguientes preguntas:

- ¿Cómo de difícil te ha resultado realizar la tarea? [Escala de 0 (muy difícil) a 10 (muy fácil)].
- ¿Cuánto tiempo te ha llevado realizar la tarea? [Escala de 0 (mucho tiempo) a 10 (poco tiempo)].
- ¿Has detectado algún error o tienes alguna sugerencia de mejora?

Tras esto, se realizaron preguntas de carácter general sobre el desarrollo de la prueba. Las preguntas realizadas fueron las siguientes:

- Explica brevemente los programas que forman parte de la solución.
- ¿Cómo de fácil te ha sido diferenciar los diferentes programas que forman parte de la solución? [Escala de 0 (muy difícil) a 10 (muy fácil)].
- ¿Cómo de fácil te ha sido utilizar como un todo los diferentes programas que forman parte de la solución? [Escala de 0 (muy difícil) a 10 (muy fácil)].
- ¿Cómo de fácil te ha sido identificar en el archivo de texto los cambios realizados en el entorno gráfico de Vensim utilizando Semantic diff? [Escala de 0 (muy difícil) a 10 (muy fácil)].
- ¿Cómo de fácil te ha sido identificar en el archivo de texto los cambios realizados en el entorno gráfico de Vensim sin utilizar Semantic diff? [Escala de 0 (muy difícil) a 10 (muy fácil)].
- ¿Cuán útiles han sido para ti las aportaciones semánticas en el análisis de las diferencias entre versiones? [Escala de 0 (nada útiles) a 10 (muy útiles)].
- ¿Qué elemento de la solución te ha resultado más útil en cuanto a analizar las diferencias?
- ¿Cuán útiles han sido para ti las aportaciones semánticas en la integración (merge)?
- ¿Qué elemento de la solución te ha resultado más útil en cuanto a realizar la integración (merge)?
- En resumen, valora la solución integrada en cuanto: Fácil de entender, fácil de aprender, útil, rápida, original, cómoda, satisfacción general. [Escala de 0 a 10].

6.7.5. Respuestas de los usuarios al formulario

Pocas personas realizaron el formulario tras realizar las pruebas de aceptación de usuario, pero las respuestas registradas reflejaron que la herramienta permitía realizar los escenarios de manera muy intuitiva y en poco tiempo, pues todas las respuestas a las preguntas de facilidad y tiempo dedicado a las tareas se calificaron de 9, siendo 10 "Muy fácil" y "Poco tiempo" respectivamente. Los errores y sugerencias de mejoras se comentaron directamente en una reunión telemática que tuvo lugar después de la tercera prueba de usuarios.

Se dio una puntuación de 7 a la tarea de identificar en el archivo de texto los cambios realizados en el entorno gráfico de Vensim utilizando `Semantic Diff`. Sin embargo, dichos problemas fueron comentados en la reunión mencionada anteriormente y se mejoró la documentación de loa **Manuales de Usuario** para evitar dichos problemas. Se calificó de muy difícil entender los cambios realizados en un fichero utilizando únicamente la diferenciación textual y se calificó de muy útil con la máxima puntuación la ayuda que proporciona este proyecto para entender los cambios realizados, destacando en especial las dos herramientas auxiliares, `Adder` y `Eraser`.

6.7.6. Problemas o advertencias de uso encontrados durante la realización de las pruebas

En las diferentes pruebas de aceptación de usuarios realizadas se produjeron varios fallos sobre los cuales, tras investigarlos por separado, se concluyó que eran debidos al funcionamiento de las herramientas de `SemanticMerge` y `Vensim`. En concreto, varios de estos fallos se encontraron al investigar por qué al realizar un cambio simple como eliminar una variable, `gMaster` llegaba en algunos casos a mostrar que se habían realizado decenas de cambios. Esta sección se encuentra detallada en profundidad en el anexo de **Manuales de Usuario**, en la sección A.2.5.

Capítulo 7

Seguimiento del proyecto

7.1. Introducción

Este Trabajo de Fin de Grado comenzó de forma oficial el 14 de septiembre de 2020, con la primera reunión organizativa del mismo. Sin embargo, durante el mes de agosto se comenzó a trabajar con las tecnologías referentes al proyecto para amenizar la curva de aprendizaje inicial y reducir la carga de trabajo en los primeros meses.

7.2. Sprint 0

Con el fin de amenizar la carga de trabajo en el inicio del desarrollo del proyecto, durante el mes de agosto se estableció como objetivo el crear un parser externo para la herramienta `SemanticDiff`, el cual fuese capaz de procesar un pequeño programa en un lenguaje imaginario aunque muy parecido a `Java`.

Para ello, se investigó acerca del funcionamiento de `ANTLR4` y su funcionamiento con otros lenguajes de programación. Al tener conocimiento de gramáticas y otro tipo de software más antiguo como `Lex` o `Yacc`, esto no supuso un gran problema. Además, tras investigar acerca de `ANTLR4`, se observó que tenía muy buena sincronización con `Java`, por lo que se utilizaría dicho lenguaje de programación para desarrollar el proyecto. A mayores, también se familiarizó con el entorno de `Vensim` y se estudió la gramática proporcionada por el Trabajo de Fin de Grado del curso anterior de Daniel Bazaco [5].

En el caso de la creación del plugin externo, existieron problemas debido a la falta de documentación y a que la documentación existente era antigua. Tras un tiempo sin lograr encontrar la solución a varios problemas que surgieron, se creó un grupo de trabajo en `BaseCamp 3` con varios ingenieros de la empresa Códice Software, los cuales tenían conocimiento sobre la herramienta `SemanticMerge`. Con su ayuda se logró desarrollar un parser externo que

utilizaba una gramática en `ANTLR4` utilizando el framework de `Maven`. Dicho proyecto era capaz de generar un archivo `.jar` que `SemanticMerge` reconoce como archivo ejecutable y proporciona las diferencias entre dos archivos de prueba. Pese a no medirse este tiempo de preparación, se estima un trabajo entre unas 40 y 45 horas.

7.3. Sprint 1 (14/9/2020 - 27/9/2020)

La Tabla 7.1 muestra las tareas realizadas durante este *sprint* así como el tiempo invertido en ellas.

Tarea	**Tiempo estimado**	**Tiempo invertido**	**Estado**
Entender un plugin de SemanticMerge de un lenguaje no orientado a objetos (Scala)	2h	1h 40min	Completado
Entender cómo SemanticMerge gestiona los diferentes tipos de diferencias	4h	3h	Completado
Ampliar la gramática **(*)**	8h	24h 20min	En proceso
Valorar qué tests son posibles y si es posible una integración continua/despliegue continuo	4h	4h 30min	Completado
Informarse de si es posible modificar las shadow variables	4h	50min	En proceso
Documentación del sprint	10h	12h	Completado
Sprint planning + sprint review + revisión semanal	2h	2h	Completado
Total	**34h**	**48h 20min**	**5/7 tareas completadas**

(*) Este tiempo incluye también el tiempo destinado a testear los cambios realizados en la gramática.

Tabla 7.1: Tareas del sprint 1.

Este *sprint* se caracterizó principalmente en la búsqueda de información y documentación de cara a futuros *sprints* del proyecto y a la ampliación de la gramática de `Vensim` ya existente, centrándose en las vistas y los gráficos. Al estar el desarrollo de la gramática explicado en profundidad en la sección de "Diseño", no se profundizará de nuevo en esta sección.

Se comenzó investigando el funcionamiento de `SemanticMerge`, pues todos los plugins externos encontrados y los lenguajes que actualmente soporta el software tienen en común que comparten una estructura típica de los lenguajes orientados al objeto. Sin embargo, `Vensim` tendría una estructura más parecida a lenguajes orientados al paradigma funcional como

`Python` o `Scala`. Tras encontrar un repositorio con un plugin externo de `Scala` y consultar la documentación oficial de `SemanticMerge`, se concluyó que la aplicación utiliza una estructura de ámbito general compatible con cualquier tipo de paradigma. Dicha estructura se caracteriza por utilizar dos elementos, contenedores y nodos, pudiendo contener los contenedores a su vez otros contenedores, estructura similar a la que utiliza el patrón de diseño "*Composite*".

En cuanto a los tests, se concluyó que testear la gramática era inviable debido a la falta de documentación y librerías, y pequeños cambios en ella podrían dejar inútiles la mayoría de los tests. Sin embargo, si que se podrían testear los visitors de la gramática por separado así como las reglas por separado, utilizando pequeños ficheros de prueba `.mdl` y comprobando que el *output* sea el esperado, puesto que todos los posibles errores encontrados al *parsear* un fichero se muestran por consola al principio. No existiría problema en realizar integración continua, sin embargo, para el despliegue continuo, si bien sí que es posible utilizar un contenedor `Docker` para instalar `SemanticMerge` y probar el *parser* en remoto, se concluyó que era mucho más fácil realizar un *script* que realizase las comprobaciones automáticas en la máquina local. Pese a ello, no se descartaron ninguna de las dos opciones.

Finalmente, se estuvo investigando acerca de las variables sombra o *shadow variables*. Existen dos tipos de variables sombra, las puras como el tiempo, y las que dependen de otras variables creadas por el usuario. Estas últimas dependen de sus variables iniciales y si estas se eliminan se eliminarán también las variables sombra asociadas. Se determinó que la gramática era capaz de tratar el problema a través de `SemanticMerge` y el futuro código implementado.

7.4. Sprint 2 (28/9/2020 - 11/10/2020)

La Tabla 7.2 muestra las tareas realizadas durante este *sprint* así como el tiempo invertido en ellas. Este *sprint* se caracterizó por una parte por el trabajo dedicado a la compresión

Tarea	**Tiempo estimado**	**Tiempo invertido**	**Estado**
Investigar cómo analizar cobertura de gramática	2h	40min	Completado
Conocer qué significan cada número/parámetro en los objetos de las vistas	2h	1h	Completado
Ampliar la gramática **(*)**	8h	2h 20min	Completado
Conocer cómo utilizar correctamente múltiples visitors	2h	30min	Completado
Informarse de si es posible modificar las shadow variables	4h	1h 50min	Completado
Comprender como funciona una tabla de símbolos	2h	1h 10min	Completado
Entender estructura del TFG de Daniel Bazaco	2h	1h 30min	Completado
Documentación del sprint	10h	3h 15min	Completado
Sprint planning + sprint review + revisión semanal	2h	2h	Completado
Total	**34h**	**14h 15min**	**9/9 tareas completadas**

(*) Este tiempo incluye también el tiempo destinado a testear los cambios realizados en la gramática.

Tabla 7.2: Tareas del sprint 2.

del proyecto anterior del que parte este trabajo, así como a la toma de contacto con los integrantes del proyecto `LOCOMOTION` gracias a una reunión por videoconferencia.

En dicha conferencia se estableció contacto con tres miembros del proyecto, los cuales expresaron sus principales problemas y dificultades a la hora de trabajar con `SemanticMerge` con archivos Vensim. Principalmente existían dos problemas, la parte dedicada a definir las vistas y los gráficos en los ficheros `.mdl` es muy confusa, ya que se emplean líneas individuales para cada variable con muchos parámetros numéricos, lo que hacía muy complicado distinguir qué significa cada campo y qué campos son triviales a la hora de hacer un *merge*; y que la gramática actual, al estar enfocada únicamente a la parte de definición de ecuaciones, potencia el problema actual, siendo los bloques que `SemanticMerge` usa para resolver los conflictos en la parte de definición de las vistas muy grandes, haciendo casi imposible resolver conflictos con la herramienta y teniéndolos en consecuencia que hacer a mano.

Estos problemas quedaron registrados en el *backlog* y enfocar el proyecto a dichos objetivos. Si bien la parte de tratamiento de las vistas por parte de la gramática ya era un objetivo previsto y desarrollado, se modificó la gramática para distinguir las partes triviales de las líneas de definición de variables en las vistas. No se profundizará en este tema pues quedó explicado anteriormente en la sección de Análisis. También en esa sección, queda planteada la solución al problema que surgió en el anterior *sprint*, el cual surgía de no saber cómo catalogar si una variable sombra ha sido borrada y creada de nuevo, o si simplemente se había movido.

Por otro lado, se realizó una labor de investigación para afianzar conocimientos sobre la base de la que parte el proyecto, así como del funcionamiento general de una gramática, su tabla de símbolos y su árbol generado.

Tras realizar las últimas modificaciones a la gramática, se amplió en una gran medida el número de tests, gracias a diferentes fuentes proporcionadas por la tutora [9] [21] [31]. Pese a ello y con el fin de estar seguros de que la gramática era correcta al 100 %, se intentó buscar alguna forma o software para realizar un análisis de cobertura de la gramática. Desgraciadamente, no se encontró ninguna fuente que especificase cómo hacerlo por lo que se descartó.

7.5. Sprint 3 (12/10/2020 - 25/10/2020)

La Tabla 7.3 muestra las tareas realizadas durante este *sprint* así como el tiempo invertido en ellas.

Tarea	**Tiempo estimado**	**Tiempo invertido**	**Estado**
Realizar un programa que añada al comentario de una ecuación la vista a la que pertenece	14h	13h 10min	Completado
Configurar máquina virtual y planificar despliegue continuo	1h 30min	1h 20min	Completado
Decidir cómo integrar el programa de resolución de vistas en el parser	1h 30min	10min	Completado
Documentación del sprint	2h	2h 10min	Completado
Sprint planning + sprint review + revisión semanal	2h	2h	Completado
Total	**21h**	**18h 50min**	**5/5 tareas completadas**

Tabla 7.3: Tareas del sprint 3.

Este *sprint* se caracterizó por el desarrollo de una de las tareas principales enunciadas en el backlog principal: desarrollar un programa que lea un fichero `.mdl`, reconozca qué variables pertenecen a una vista e introducir en el comentario de las ecuaciones de las variables a qué vista pertenece la misma. Como este proceso ha sido ya explicado en el capítulo de Diseño, no se profundizará más en él. También se dedicó tiempo a pensar cómo se integraría dicho programa en el parser final. Se llegó a la conclusión de que el parser contendrá una clase principal que leerá los archivos y les aplicará el formateo. Tras ello, escribirá el resultado en dos archivos, como por ejemplo `arch1.mdl` y `arch2.mdl`, que siempre serán leídos por el visitor, independientemente de su nombre original. Se podrá realizar una segunda escritura para reemplazar el contenido de los ficheros originales por su texto formateado.

También se trabajó en la configuración de la máquina virtual para realizar el despliegue continuo en un futuro. Se instaló el software de `Vensim` y `SemanticMerge` para poder realizar el trabajo, instalando también el `JDK-11` de `Java` para poder ejecutar el parser externo en formato `jar`, puesto que no existía ninguna versión de `Java`. Para poder acceder a la máquina, se configuró como servidor `ssh`, abriendo el puerto 22 e instalando `openssh-server`. Tras comprobar que el acceso era posible, se realizó un *script* en formato `.bat` que serviría para poder ejecutar el parser externo desde línea de comando en un futuro.

7.6. Sprint 4 (26/10/2020 - 08/11/2020)

La Tabla 7.4 muestra las tareas realizadas durante este *sprint* así como el tiempo invertido en ellas.

Tarea	**Tiempo estimado**	**Tiempo invertido**	**Estado**
Integrar TFG de partida	8h	1h 30min	Completado
Desarrollar parser básico inicial	14h	18h 5min	Completado
Incluir integración continua funcional	2h	1h 10min	En proceso
Documentación del sprint	2h 30min	3h 10min	Completado
Sprint planning + sprint review + revisión semanal	2h	2h	Completado
Total	**26h 30min**	**25h 20min**	**4/5 tareas completadas**

Tabla 7.4: Tareas del sprint 4.

Este *sprint* se caracterizó por el desarrollo de un primer *parser* basado únicamente en la gramática de partida. No se profundizará en exceso en este apartado al estar descrito extensamente en la sección de Diseño. Gracias al tiempo dedicado en verano durante el *Sprint 0*, no se obtuvieron errores propios de `SemanticMerge`, sin embargo, se obtuvieron errores que eran nuevos hasta la fecha, provenientes de una estructura errónea del archivo `YAML`. Tras contactar con los ingenieros de `PlasticSCM`, se figuró la fuente del problema y se solucionó correctamente.

Por otro lado, se comenzó el proceso de incorporar al proyecto integración y despliegue continuos. Sin embargo, pese a pasar correctamente los *tests* en la máquina local, en en proceso de despliegue continuo se obtuvieron errores desconocidos al generar los archivos `YAML` de forma diferente.

7.7. Sprint 5 (09/11/2020 - 22/11/2020)

La Tabla 7.5 muestra las tareas realizadas durante este *sprint* así como el tiempo invertido en ellas.

Tarea	**Tiempo estimado**	**Tiempo invertido**	**Estado**
Desarrollar un parser que incluya toda la gramática	18h	16h 5min	Completado
Testing avanzado del parser	4h	7h 20min	En proceso
Incluir integración continua funcional	2h	1h 40min	Completado
Investigar cómo integrar el parser en gMaster	1h 30min	15min	Completado
Integrar el programa de formateo del texto en el parser	3h	2h 40min	Completado
Documentación del sprint	3h	4h	Completado
Sprint planning + sprint review + revisión semanal	2h	2h	Completado
Total	**33h 30min**	**34h**	**6/7 tareas completadas**

Tabla 7.5: Tareas del sprint 5.

Este *sprint* se caracterizó por el desarrollo de un *parser* que tratase los archivos `.mdl` al completo, esto es, haciendo uso de la gramática desarrollada en este proyecto. Para su realización se partió de la base del realizado en el *sprint* anterior, si bien, se modificaron los tests para adaptarlos a las nuevas características del *parser*. Como ya se ha explicado en el capítulo de Diseño (y por ello no se profundizará en exceso), se obtuvieron problemas creando el *parser* puesto que al integrar el programa para añadir los comentarios de las vistas a las ecuaciones y un segundo programa cuya función es añadir ciertas líneas delimitadoras para organizar mejor el código y la estructura de la generación del *parser*, no se podía reconstruir correctamente el fichero de entrada a través del `YAML` generado, pues este había sido modificado. La solución fue dividir el proyecto en dos programas: uno que modificase el contenido del fichero y el *parser* en cuestión.

En la integración continua, se descubrió que los errores de diferencia entre los tests local y los de `GitLab` se debía a que Windows utiliza el sistema de retorno de línea `CRLF` y `GitLab`

utiliza el propio de Linux, `LF`. Tras varios intentos fallidos, se utilizo como solución temporal obviar los tests referentes a la generación del `YAML` en la integración continua. En el despliegue continuo se consiguió automatizar el envío y prueba de los *parsers*. Al ser una interfaz de comandos o `CLI`, era complicado probar `SemanticMerge` puesto que esta aplicación funciona de forma gráfica. Para probar que se había ejecutado correctamente, se ejecuta el *script* que abría la aplicación con los parámetros del *parser* externo y se utiliza un *timeout* de 10 segundos. Tras ello, se intenta cerrar el programa con la orden `taskkill`. Si el programa se ha abierto correctamente, se cierra sin problemas. En caso contrario no se cerrará y el *timeout* de `GitLab` cerrará el intento de despliegue continuo generando un error.

Sobre la integración del *parser* en `gMaster`, se descubrió que con tan sólo modificar un fichero [27], la aplicación aceptaba herramientas externas, por lo que no supuso mucho trabajo. No se realizó un proceso de *testing* tan extenso como se pensaba por falta de tiempo, pero se pospusieron los tests restantes para el siguiente *sprint*.

7.8. Sprint 6 (23/11/2020 - 06/12/2020)

La Tabla 7.6 muestra las tareas realizadas durante este *sprint* así como el tiempo invertido en ellas.

Tarea	**Tiempo estimado**	**Tiempo invertido**	**Estado**
Crear un programa para eliminar los delimitadores	2h	25min	Completado
Añadir mejoras semánticas al parser	1h	25min	Completado
Testing avanzado del parser	4h	4h	Completado
Documentación del sprint	1h	50min	Completado
Sprint planning + sprint review + revisión semanal	2h	2h	Completado
Total	**10h**	**8h 40min**	**5/5 tareas completadas**

Tabla 7.6: Tareas del sprint 6.

Este *sprint* fue marcado como débil o ligero en la planificación inicial debido a la concentración de entregas y exámenes por parte del curso universitario. Por ello, el trabajo realizado es significativamente menor que en otros *sprints*.

El trabajo realizado el este periodo consistió principalmente en el desarrollo de un programa que eliminase los delimitadores que utiliza el *parser* para no afectar a los *merges* entre versiones y a continuar con el proceso de *testing* del *parser*. Lo primero resultó mucho más fácil de lo que se esperaba, puesto que fue suficiente con leer una a una las líneas del fichero y eliminar las que coincidiesen con alguno de los delimitadores. Debido a que los delimitadores

fueron creados con una estructura especial que no podría coincidir por casualidad con alguna otra línea, el proceso fue muy sencillo y sin tener falsos positivos en la detección.

7.9. Descanso de navidades (07/12/2020 - 17/01/2021)

La Tabla 7.7 muestra las tareas realizadas durante este periodo así como el tiempo invertido en ellas.

Tarea	**Tiempo estimado**	**Tiempo invertido**	**Estado**
Limpiar el repositorio de archivos no usados	40min	20min	Completado
Refactorizar tests de EvalVisitor	2h	1h	Completado
Reestructurar EvalVisitor en varios visitors	6h	1h 50min	Completado
Completar documentación	6h	5h 5min	Completado
Sprint planning + sprint review + revisión semanal	2h	2h	Completado
Total	**16h 40min**	**10h 15min**	**5/5 tareas completadas**

Tabla 7.7: Descanso de navidades.

Este periodo tuvo una duración aproximada de un mes y fue catalogado de descanso, debido a entregas de prácticas finales, exámenes finales y festividades de Navidad. Se retomaría el ritmo normal del proyecto al terminar la convocatoria ordinaria de exámenes. Este *sprint* estaba destinado a recuperar ciertos aspectos que se pudieran haber dejado atrás en el *sprint* anterior, así como hacer un correcto *refactor* de ciertos aspectos del proyecto.

Se comenzó este periodo cambiando la forma de realizar los *tests* destinados a comprobar la funcionalidad del *visitor*. Esto consistió en modificar los *tests* para que en vez de leer línea a línea el archivo y comprobar que estas fuesen correctas, directamente se aplicaría un *diff* entre el archivo generado y una copia de lo que debería generar el archivo. Con esto se redujo la complejidad de los *tests* y los hizo mucho más fácil de comprender. Además, se eliminó el problema de salto de línea entre Windows y Linux producido en el proceso de Integración Continua.

Posteriormente, se procedió a la refactorización del hasta el momento el único *visitor* en varios, separando el mismo en un *visitor* principal y otros tres encargados de procesar los apartados de ecuaciones, vistas y gráficos y metadatos respectivamente. En dicho proceso se creó a mayores una clase externa con todas las funciones secundarias que el *visitor* utilizaba en un principio. También se añadió documentación para dichas funciones y se tradujeron todos los comentarios pertinentes al inglés. Finalmente, se eliminaron archivos redundantes o inútiles del proyecto, dejándolo mucho más claro y comprensible.

Por último, se completaron todas las secciones posibles en la documentación, realizando algunas que no se habían empezado, y completando otras que se empezaron en un momento anterior, tales como el *Product Backlog* o la sección de Introducción. En las secciones de Diseño y Pruebas e Implementación se añadieron fragmentos de código con el fin de lograr una mejor explicación del proceso realizado.

7.10. Sprint 7 (18/01/2021 - 02/02/2021)

La Tabla 7.8 muestra las tareas realizadas durante este *sprint* así como el tiempo invertido en ellas. Este *sprint* tiene una duración mayor que los demás puesto que coincidió con el cambio de cuatrimestre universitario. Esto supuso nuevos horarios para el alumno y la tutora del proyecto, por lo que las reuniones tendrían lugar los martes por la tarde en vez de los lunes por la mañana, como se había estado haciendo en todo el desarrollo del proyecto hasta este punto.

Tarea	**Tiempo estimado**	**Tiempo invertido**	**Estado**
Interfaz gráfica Formatter y Delimitter	2h	1h 55min	Completado
Crear interfaz gráfica No-Delimiters	3h	5h 35min	Completado
Añadir gMaster a CD	40min	15 min	Completado
Documentación del sprint	6h	9h 50min	Completado
Arreglar problema de parseo de gMaster	1h 30min	1h 40min	Completado
Crear formulario para pruebas de aceptación de usuarios	20min	1h 45min	Completado
Realizar escenarios de pruebas para usuarios	6h	3h 10min	Completado
Solucionar bugs de espaciados en el parser	9h	11h 50min	Completado
Analizar la eficiencia del parser	4h	1h 30min	En proceso
Sprint planning + sprint review + revisión semanal	2h	2h	Completado
Total	**34 h 30min**	**39h 30min**	**10/11 tareas completadas**

Tabla 7.8: Sprint 7.

Este *sprint* se caracterizó por el desarrollo de interfaces de usuario para los formateadores de texto y la escritura de documentación, destacando concretamente los **Manuales de Usuario**, puesto que el siguiente *sprint* estaría destinado a la realización de pruebas de usuario sobre el proyecto. Puesto que dichas interfaces están explicadas en los capítulos de Diseño y Manuales de Usuario, no se profundizará más en ello.

En cuanto a los Manuales de Usuario, se detallaron los pasos a seguir para la instalación de los archivos y se crearon varios escenarios de prueba para probar tanto las interfaces de formateo de archivos como para el *parser* externo a través de `gMaster`. Estos escenarios y su correspondiente desarrollo quedan detallados en el capítulo de Implementación y Pruebas.

Se modificó la configuración de la máquina virtual para integrar correctamente los archivos `Vensim` con `gMaster`. Sin embargo, se descartó utilizar dicha herramienta en el despliegue continuo al no tener una ejecución simple a través de línea de comandos.

Tras la reunión de mitad de *sprint* se destacaron dos puntos importantes: la modificación de las interfaces de usuario para hacerlas más usables así como la modificación de los tests de aceptación de usuarios, pues estos estaban destinados a realizarse con los ficheros abiertos en modo textual y no en modo de interfaz gráfica. Respecto a las modificaciones de las interfaces de usuarios, estas se hicieron más grandes y con menos botones por pantallas. Además, se añadieron etiquetas explicativas sobre las funcionalidades de dichas interfaces y un botón para mostrar la ruta absoluta del archivo a escoger, así como nuevas captaciones de errores. Todos estos cambios se realizaron con la finalidad de que usuarios con menos conocimiento en la informática o con un mayor de probabilidades de cometer errores humanos tuviesen una experiencia más agradable trabajando con este proyecto.

Respecto a las modificaciones de los tests, se dividieron en varios conjuntos de escenarios, los cuales incrementaban el nivel de complejidad del archivo de prueba progresivamente. Como se ha comentado antes, estos tests fueron enfocados para que el usuario modificase los archivos desde la interfaz gráfica de `Vensim`. Junto a estos tests se especificaron las competencias que el usuario debería desarrollar. Adicionalmente, se creó un formulario para que los usuarios pudieran calificar el proyecto hasta la fecha y aportar sugerencias para incorporarlas en *sprints* posteriores.

Trabajando con la herramienta de `gMaster`, se descubrió un *bug* que causaba que los ficheros fuesen *parseados* correctamente sólo en algunas ocasiones, produciéndose un error en las ocasiones restantes. Dicho error se solucionó modificando el *parser* para esperar una cantidad de archivos indeterminados en lugar de los dos únicos archivos que esperaba inicialmente. Este proceso queda explicado en mayor detalle en el capítulo de **Introducción**.

A mayores, trabajando con un nuevo fichero de gran tamaño, se descubrió un *bug* que causaba que se procesase mal el espaciado en los nombres de las vistas, variables u otros elementos. Este problema queda explicado en el capítulo de **Diseño**. Se contempló un problema con el carácter ':'. En un principio se pensó que se trataba de un *bug*, pero tras investigarlo se descubrió que si se utilizaban espacios en blanco tras dicho carácter en nombres de variables, vistas u otros elementos provocaba que el archivo `YAML` fuese incorrecto, pues este utiliza el carácter ':' para delimitar los parámetros del árbol que genera.

Otro problema encontrado es que al tratar de *parsear* archivos muy grandes (aproximadamente unas 40.000 líneas), `gMaster` no conseguía procesarlo, pues tardaba excesivo tiempo en poder procesar el archivo. Se investigó sobre la eficiencia del programa creado pero no se obtuvieron conclusiones sólidas al no detectar ninguna sentencia que pudiese causar un cuello de botella respecto a las demás estructuras del programa.

7.11. Sprint 8 (03/02/2021 - 17/02/2021)

La Tabla 7.9 muestra las tareas realizadas en este *sprint* así como el tiempo invertido en ellas.

Tarea	**Tiempo estimado**	**Tiempo invertido**	**Estado**
Hacer la aplicación compatible con Java8	1h	4h 20min	Completado
Borrar los nombres de las vistas utilizando el segundo formateador	2h	45min	Completado
Documentación del sprint	7h	5h 45min	Completado
Prueba de aceptación de usuarios	2h	3h	Completado
Analizar la eficiencia del parser	0h	0min	En proceso
Sprint planning + sprint review + revisión semanal	2h	2h	Completado
Total	**14h**	**17h 50min**	**5/6 tareas completadas**

Tabla 7.9: Sprint 8.

Este *sprint* se caracterizó por la búsqueda de soluciones y su correspondiente aplicación a todos los problemas encontrados en la primera prueba de aceptación de usuarios. En dicha prueba, los usuarios tuvieron problemas para utilizar la herramienta pues estos utilizaban la versión 8 de Java, mientras que el proyecto había sido desarrollado en la versión 11. Tras realizar varias pruebas en máquinas virtuales, se determinó que sólo hacía falta cambiar una función del paquete de `utilities` para poder migrar correctamente de versión el proyecto. Durante dichas pruebas se comprobó qué versiones de los productos a utilizar eran válidas (`gMaster`, `Vensim`) así como que no era necesario tener instalado un `JDK` de Java, bastaba sólo con un `JRE`, cuya versión mínima quedó detallada junto a otros muchos aspectos en el anexo de **Manuales de Usuario**.

Por otro lado, se tomaron en cuenta las aportaciones de los usuarios sobre las interfaces de usuario de las herramientas para formatear archivos `Vensim`. Se eliminaron los colores de fondo y se aumentó considerablemente el tamaño de la letra de la etiqueta que muestra si las operaciones fallan o tienen éxito. Además, se añadió un icono a cada ventana para poder distinguirlas fácilmente. De nuevo, esto puede apreciarse en el anexo de **Manuales de Usuario**.

Finalmente, se añadió la funcionalidad de eliminar los nombres de las vistas en los comentarios de las ecuaciones, puesto que se contempló la posibilidad de que dichas inserciones produjeran conflictos con otros proyectos paralelos como el proyecto asociado al control de calidad de archivos `Vensim` con `SonarQube`.

7.12. Sprint 9 (18/02/2021 - 02/03/2021)

La Tabla 7.10 muestra las tareas realizadas en este *sprint* así como el tiempo invertido en ellas.

Tarea	**Tiempo estimado**	**Tiempo invertido**	**Estado**
Agregar imágenes a los archivos JAR	2h	45min	Completado
Solucionar los problemas de la segunda prueba de aceptación de usuarios	4h	5h 20min	Completado
Documentación del sprint	6h	4h 25min	Completado
Prueba de aceptación de usuarios	2h	3h	Completado
Analizar la eficiencia del parser	2h	50min	En proceso
Sprint planning + sprint review + revisión semanal	2h	2h	Completado
Total	**18h**	**16h 20min**	**5/6 tareas completadas**

Tabla 7.10: Sprint 9.

Este *sprint* comenzó con la segunda prueba de aceptación de usuarios, donde esta vez se habían solucionado todos los problemas de versiones. Sin embargo, en la realización de dicha prueba se vio que `SemanticMerge` detectaba muchos más cambios de los verdaderamente realizados. Tras varios intentos, se finalizó la prueba y se anotó como tarea del *sprint* solucionar dichos problemas.

Tras analizar el problema, se comprobó que la raíz del mismo procedía de un problema de espaciados, el cual se solucionó con relativa facilidad. Aun así, se revisó toda la solución en busca de problemas similares. Se encontraron algunos problemas de nuevo relativos al espaciado, pero se lograron solucionar. A mayores, se implementaron nuevas características de usabilidad en las interfaces de usuario, tales como la capacidad de mantener abiertas las dos interfaces todo el tiempo y que su buscador de archivos sólo mostrase archivos `.mdl`. Durante este proceso de *debugging*, se encontraron varios escenarios en los que `Vensim` modificaba los archivos por cuenta propia y que producían cambios detectables por `SemanticMerge`. Dichos cambios fueron documentados en el Anexo de **Manuales de Usuario** y transmitidos a los usuarios que realizaron las pruebas de cara a futuras pruebas de aceptación de usuarios. Todos estos cambios así como otras consideraciones sobre la memoria fueron incluidas en la documentación, por lo que este *sprint* tuvo una carga significativa asociada a esta tarea.

Finalmente, con el objetivo de encontrar una causa para la baja eficiencia del programa con archivos de muy gran tamaño, se decidió utilizar la herramienta `SonarQube` para comprobar la calidad del código y con ello detectar brechas de eficiencia. Esto queda documentado en mayor profundidad en el capítulo de **Implementación y pruebas**.

7.13. Sprint 10 (03/03/2021 - 17/03/2021)

La Tabla 7.11 muestra las tareas realizadas en este *sprint* así como el tiempo invertido en ellas.

Tarea	**Tiempo estimado**	**Tiempo invertido**	**Estado**
Corregir bugs del proyecto	2h	6h 20min	Completado
Documentación del sprint	2h	50min	Completado
Sesión de profiling con desarrolladores de Plastic SCM	1h	1h 30mih	Completado
Analizar la eficiencia del parser	2h	0min	Completado
Prueba de aceptación de usuarios	2h	1h 10min	Completado
Sprint planning + sprint review + revisión semanal	2h	2h	Completado
Total	**11h**	**11h 50min**	**6/6 tareas completadas**

Tabla 7.11: Sprint 10.

Este *sprint* tuvo una menor carga de tarea, puesto que la reunión de *profiling* para analizar la eficiencia del *parser* tuvo lugar en la segunda semana del *sprint*. Pese a ello, si que se realizaron pequeñas tareas de documentación en la primera semana. Sobre la reunión para analizar el rendimiento, se concluyó que no era un problema de rendimiento, sino un problema de *parseo* que estaba siendo invisible debido a que el programa utilizaba espera activa para leer los archivos en este punto del proyecto. Esto queda explicado en profundidad en el capítulo de **Implementación y pruebas**.

A mayores se intentó buscar posibles *bugs* en los programas de adición y eliminación de delimitadores, pues los usuarios encontraron algunos problemas en la tercera prueba de aceptación de usuarios, la cual realizaron de forma independiente sin ninguna clase de reunión y tras la cual comunicaron dichos problemas por correo electrónico. No obstante, no se encontraron *bugs*, por lo que se esperó al siguiente *sprint*, donde se tendría una reunión donde se explicarían dichos problemas en profundidad.

El último día del *sprint* tuvo lugar la tercera reunión de pruebas de aceptación de usuario, aunque en esta ocasión, los usuarios ya habían realizado las pruebas por cuenta propia. Dicha reunión fue meramente de aclaración de dudas y explicación de cómo fueron las pruebas, que en esta ocasión, resultaron haber tenido mucho éxito. En ella se propuso investigar acerca del apartado de metadatos de los archivos `.mdl` para facilitar su compresión a los usuarios.

7.14. Resumen del proyecto

Tras finalizar el proyecto se echó la vista atrás para comparar las marcas establecidas inicialmente en cuanto a tiempo y dinero, y se sumarizaron todas las horas dedicadas al realizar este Trabajo de Fin de Grado así como las tareas realizadas.

7.15. Sprint 11 (18/03/2021 - Fin de proyecto)

La Tabla 7.12 muestra las tareas realizadas en este *sprint* así como el tiempo invertido en ellas.

Tarea	**Tiempo estimado**	**Tiempo invertido**	**Estado**
Documentación del sprint	6h	3h	Completado
Revisión de la documentación	4h	1h 45min	Completado
Sprint planning + sprint review + revisión semanal	2h	2h	Completado
Total	**11h**	**11h 50min**	**3/3 tareas completadas**

Tabla 7.12: Sprint 11.

Este *sprint* final se dedicó a completar la documentación y a revisar la misma en busca de errores o partes que pudiesen mejorarse o completarse. También se revisó el código realizado para limpiar comentarios, aclarar partes del código o hacer algunos *refactors* enfocados a mejorar la legibilidad del mismo.

7.15.1. Calendarización final

Pese a que se fueron cumpliendo las cuotas establecidas en el inicio del proyecto en lo referente al número de *sprints*, debido a los problemas ocurridos en las pruebas de aceptación de usuarios en el mes de febrero, la finalización del proyecto se retrasó aproximadamente un mes, que sería el equivalente a dos *sprints*. El cambio de horarios del segundo cuatrimestre hizo que dicha tabla se modificara ligeramente en las últimas etapas, al pasar las reuniones semanales del lunes al martes. Por lo tanto, la Tabla de calendarización final 7.13 terminó de la siguiente forma:

Sprint 0	01/08/2020 - 13/09/2020
Sprint 1	14/09/2020 - 27/09/2020
Sprint 2	28/09/2020 - 11/10/2020
Sprint 3	12/10/2020 - 25/10/2020
Sprint 4	26/10/2020 - 08/11/2020
Sprint 5	09/11/2020 - 22/11/2020
Sprint 6	23/11/2020 - 06/12/2020
Descanso	07/12/2020 - 17/01/2021
Sprint 7	18/01/2021 - 02/02/2021
Sprint 8	03/02/2021 - 17/02/2021
Sprint 9	18/02/2021 - 02/03/2021
Sprint 10	03/03/2021 - 17/03/2021
Sprint 11	18/03/2021 - Fin de proyecto

Tabla 7.13: Tabla de calendarización de sprints final.

Tal y como se puede apreciar, el último *sprint* se alargó de las dos semanas convencionales pues fue una etapa final dedicada a completar la documentación y a revisar la misma.

En cuanto a los riesgos establecidos en un principio, el riesgo que más se temía que ocurriese fue la falta de tiempo, en parte debida a compaginar las cinco asignaturas del primer cuatrimestre con el proyecto. Sin embargo, el hecho de utilizar metodologías ágiles en el desarrollo del proyecto y el poder asistir a las clases de forma telemática y no gastar tiempo en desplazamientos permitió que el proyecto se desarrollase en las fechas previstas con una cantidad de estrés aceptable. Si que existieron modificaciones en los requisitos o requisitos adicionales, pero por suerte, pudieron asignarse adecuadamente las tareas a los *sprints*. Muchos de estos nuevos requisitos no requirieron modificaciones del trabajo anterior, por lo que no supusieron demasiados problemas.

En cuanto al resto de riesgos, no se produjeron actualizaciones en ninguno de los programas utilizados, por lo que no hubo impacto alguno. Desde un principio se instauró el trabajo y las reuniones de forma telemática, por lo que tampoco hubieron imprevistos relacionados con el COVID-19.

7.15.2. Trabajo total realizado

En la Tabla 7.14 se recopilan las horas y las tareas totales dedicadas a la realización del proyecto:

Trabajo total estimado	**Trabajo total realizado**	**Total tareas completadas**
300h	292h5m	65/65

Tabla 7.14: Trabajo total realizado.

7.15.3. Costes reales

El coste real se ajusta a la remuneración de la beca, la cual consta de 300 euros brutos durante un periodo de seis meses, por lo que el presupuesto total sería en un principio de 1800 euros. Sin embargo, debido a que el alumno firmó un contrato de prácticas incompatible con la beca, este renunció a la misma desde el mes de febrero, llegando a percibir únicamente la remuneración correspondiente a cinco meses. Sin embargo, como el mes de septiembre se comenzó el trabajo el día 14, sólo se percibieron 16 días de trabajo. Por lo tanto, sin contar descuentos de la Seguridad Social e IRPF, se recibió un total de 1350 €.

En el caso de la licencia de `SemanticMerge`, la empresa `PlasticSCM` decidió proporcionar una licencia de usuario para el desarrollo del proyecto de forma totalmente gratuita, por lo que no existieron gastos en este ámbito.

Cabe resaltar que a pesar de que la beca comenzase en Julio (por lo que terminaría en diciembre), el proyecto se inició oficialmente el 14 de septiembre, por lo que su duración llegaría hasta el mes de marzo. Además, durante el mes de agosto se realizó el *sprint* 0, el cual consistió en documentarse acerca de la realización del proyecto y en la creación de pequeños *parsers* para familiarizarse con el entorno en que se iba a trabajar.

Capítulo 8

Conclusiones

Tras finalizar el proyecto se lograron cumplir todos los objetivos y requisitos iniciales y, adicionalmente, se pudieron cumplir los nuevos requisitos y necesidades que fueron surgiendo a lo largo del desarrollo del proyecto. Este proyecto no ha sido fácil, puesto que la tarea a realizar era laboriosa y compleja, sin embargo se logró finalizar el proyecto cerca de la fecha estimada inicialmente.

El proyecto comenzó en el mes de agosto con una fase preparatoria, pero no fue hasta la segunda quincena de septiembre donde se empezó verdaderamente el trabajo. Debido a todas las circunstancias ocasionadas por la pandemia del virus COVID-19, el trabajo se realizó completamente en remoto. Durante el primer cuatrimestre académico, este trabajo se realizó en paralelo con las cinco asignaturas correspondientes al cuarto curso de la mención de Ingeniería del Software, y durante el segundo cuatrimestre en paralelo con las prácticas de empresa del alumno. En el primer cuatrimestre se concentró el grueso del desarrollo del proyecto así como las partes más tediosas del mismo, como el desarrollo de la gramática o los problemas iniciales para realizar el *parser* para `SemanticMerge`. Pese a poder realizar parte del desarrollo en días de diario, las asignaturas y sus correspondientes prácticas hicieron que gran parte del proyecto se realizase en fines de semana. Durante la primera mitad del segundo cuatrimestre tuvo lugar una fase más relajada destinada a la refactorización, corrección de *bugs* y pruebas de usuario, y a pesar de realizarse en paralelo con las prácticas de empresa, esta parte se realizó con mucho menos estrés y presión que en el primer cuatrimestre.

Pese a todas las dificultades ocasionadas por la pandemia, se consiguió lograr un ritmo de trabajo constante y muy productivo, en gran medida gracias a utilizar metodologías ágiles en el desarrollo, concretamente `Scrum`. Las reuniones semanales para marcar objetivos y revisar el trabajo realizado fueron de gran ayuda para conseguir realizar gran parte del proyecto en el primer cuatrimestre. Modularizar lo máximo posible el proyecto y dividirlo en *sprints* y a su vez en *issues* o tareas del *sprint* lograron que la presión general disminuyera al poder centrarse en tareas más pequeñas en vez de en un gran todo global.

Existieron dos puntos principales en el desarrollo del proyecto donde se encontraron las dificultades más relevantes. El primer punto fue en el inicio del programa, pues el alumno no

había trabajado nunca con `ANTLR4`, y pese a tener conocimientos básicos sobre gramáticas como `lex` o `yacc`, completar la gramática de partida para que incluyese también las definiciones de vistas, gráficos y metadatos fue un proceso muy laborioso y tedioso, con una gran cantidad de archivos de prueba para asegurarse de que la gramática era correcta. El segundo punto de dificultad fue aproximadamente en los inicios del mes de noviembre, donde se comenzó el desarrollo del *parser* y con ello se tuvieron las primeras dificultades reales con `SemanticMerge`. Pese a haber desarrollado varios *parsers* sencillos durante el mes de agosto como preparatoria para realizar este trabajo, las dimensiones y complejidades del proyecto dieron lugar a problemas no conocidos con los que hubo que lidiar.

Cabe resaltar también dos grandes aciertos a la hora de plantear el proyecto. El primero se ha comentado anteriormente, y es el hecho de dedicar el mes de agosto previo al comienzo del proyecto a investigar acerca de `Vensim`, `ANTLR4` y `SemanticMerge` y crear pequeños *parsers* funcionales. Pese a que posteriormente sí que existieron problemas relacionados con estos tres programas, hubieran sido mucho mayores de no ser por ese mes preparatorio. El segundo acierto fue realizar la memoria del proyecto en paralelo al trabajo realizado durante todos los *sprints* en vez de dejarlo todo para el final. Aparte de lograr una carga de trabajo más equilibrada, los conceptos explicados estarían más recientes que si se hubiesen explicado ciertas partes del proyecto meses después de su desarrollo.

8.1. Líneas de trabajo futuras

Pese a que el proyecto es completo, existen ciertos puntos que podrían ser objeto de mejora en un futuro:

- **Mejorar el parser para no depender de delimitadores**. Pese a que el *parser* es funcional gracias a estos delimitadores, podría ser posible no depender de ellos si se idease un patrón o forma de conseguir separar las vistas y los gráficos, tanto entre ellos como entre sí.
- **Intentar ampliar el parser para que abarque mayor cantidad de caracteres especiales**. Pese a que la herramienta soporta acentos y similares en `UTF-8`, se podría apuntar a utilizar otro sistema de codificación como `UTF-16`.
- **Obtener más información acerca de los metadatos**. Pese a que otras secciones como los *sketches* o los gráficos tienen soporte online sobre la información de los mismos, esto no es así con los metadatos, y se debe obtener su significado mediante la comparación entre diferentes archivos. Es una tarea laboriosa, pero podría ser información muy útil para los modeladores de `Vensim`.
- **Diferenciar campos de las líneas de definición de variables en las vistas**. Pese a que el programa actual diferencia las líneas de definición de variable de las vistas y todo su significado se encuentra en los **Manuales de Usuario** de esta memoria, así como en internet, podría intentarse separar cada campo individual de las líneas de las vistas para hacer aún más precisa la herramienta.

- **Mejorar la captura de errores en caso de que la gramática falle**. Existen ciertos casos donde `SemanticMerge` se queda en segundo plano al ocurrir un error interno por parte de un fallo de procesamiento de la gramática. No debería ocurrir casi con toda seguridad, pero actualizaciones de Vensim o similares puede que aumenten dicha probabilidad.
- **Ampliar el proyecto con futuras actualizaciones de Vensim**. Es posible que futuras actualizaciones de `Vensim` incluyan nuevos elementos que la gramática no es capaz de procesar. En este caso se debería de actualizar la gramática así como el *parser*. Esto queda explicado en el Anexo de **Manual de Mantenimiento**.

Apéndice A

Manuales

A.1. Manual de instalación

A.1.1. Prerrequisitos

Se requiere contar con:

- Windows 10, arquitectura de 64 bits. [1]
- Java (versión 8 o superior) JRE. No es necesario el JDK.
- En el caso de utilizar *tags* en `gmaster`, se debe contar con la versión 0.9.289 o superior de `gmaster` [19].
- En el caso de que se desee que para las diferencias entre versiones de archivos Excel, `gmaster` lance SpreadSheet Compare tool, se debe contar con la versión 1.0.663 o superior de `gmaster` [19].
- Los archivos Vensim a procesar deben estar codificados en el formato de espaciado propio de Windows, es decir, `CRLF`.

Se debe tener en cuenta que si se va a utilizar Vensim, se deberá tener cuidado no manipular con la distribución DSS archivos creados con la distribución PLE, y viceversa.

A.1.2. Descarga del software

La solución desarrollada consta de tres archivos `.jar` que deberán descargarse de:

[1] Para comprobar la arquitectura del sistema: `https://www.computerhope.com/issues/ch001121.htm#:~:text=Press%20and%20hold%20the%20Windows,running%20the%2064%2Dbit%20version.`

`https://gitlab.inf.uva.es/pamarti/proyecto-tfg/-/tree/master/VensimPlugin4SM`.

`https://github.com/HylianPablo/VensimPlugin4SemanticMerge/tree/master/VensimPlugin4SM`

Estos archivos son:

- `AddViewNamesAndDelimiters.jar`
- `EraseViewNamesAndDelimiters.jar`
- `VensimPlugin4SemanticMerge.jar`

Los dos primeros son aplicaciones auxiliares que se explicarán más adelante, necesarias para la solución completa. El tercero es el archivo principal de la solución desarrollada para facilitar el trabajo con archivos `Vensim` en el control de versiones.

Se debe crear una carpeta para guardar estos tres archivos. En este manual asumiremos que dicha carpeta es `C:\Users\{User}\VensimPlugin4SM`, donde `{User}` debe ser sustituido por el nombre del usuario en Windows. En el caso de guardar los archivos en otra carpeta, se recomienda altamente que la ruta absoluta de dicha carpeta no contenga espacios en blanco y/o caracteres especiales. Se recalca que dicha ruta es una ruta de ejemplo y puede ser utilizaba cualquier otra que siga las convenciones de nombres y la estructura de la de ejemplo.

A.1.3. Interacciones

Este proyecto no trabaja por sí solo (*standalone*) sino integrado como *plugin* de otras herramientas.

Para poder utilizarlo el usuario necesitará hacerlo a través de alguna de las siguientes aplicaciones: `SemanticMerge` [43] o `PlasticSCM` [48] o `gmaster` [37]. La aplicación elegida deberá estar instalada previamente a los siguientes pasos de la instalación del proyecto. Para una guía de instalación de alguna de estas aplicaciones debe dirigirse a las páginas indicadas en las referencias.

La primera de estas tres, `SemanticMerge`, es una herramienta independiente del control de versiones. La funcionalidad de SemanticMerge está integrada tanto en `PlasticSCM` como en `gmaster`, y no es necesario descargarla por separado si ya se va usar una de estas dos.

A.1.4. Configuración para el uso con SemanticMerge

Si el producto desarrollado se va a utilizar con `SemanticMerge`, se debe tener en cuenta que se trata de un software con licencia. Se debe adquirir o solicitar una clave de acceso, o bien, utilizar la prueba gratuita de 30 días.

Para utilizar herramientas externas utilizando puramente la aplicación de SemanticMerge, se debe utilizar la línea de comandos de Windows, a través de las aplicaciones de CMD, Powershell o cualquier otra aplicación externa que emule una terminal.

El comando a utilizar debe seguir la estructura que se muestra en el siguiente cuadro de texto, indicando la ruta al ejecutable de la herramienta SemanticMerge, la ruta a sendos archivos a comparar, la ruta a la herramienta externa desarrollada en este proyecto, y la ruta a la máquina virtual de Java, la cual no es más que el ejecutable de Java. Si se han seguido instalaciones normales, la ruta debería ser idéntica o muy similar a la que se muestra en el cuadro de texto. Se recomienda no utilizar espacios en blanco y/o caracteres especiales en las rutas que contienen los archivos necesarios para utilizar la herramienta.

```
C:\Users\{User}\AppData\Local\semanticmerge\semanticmergetool.exe
--source={absolute path}\example.mdl
--destination={absolute path}\example.mdl
--externalparser="-jar C:\Users\{User}\VensimPlugin4SM\VensimPlugin4SemanticMerge.jar"
--virtualmachine="C:\Program Files\Java\jdk-11.0.8\bin\java.exe"
```

A.1.5. Configuración para el uso con PlasticSCM

Si el producto desarrollado se va a utilizar utilizando la herramienta de PlasticSCM el proceso es sencillo.

Suponiendo una instalación típica, se encontrará una carpeta llamada plastic4 en la ruta predeterminada:C:\Users\{User}\AppData\Local\, donde {User} debe ser sustituido por el nombre del usuario en Windows. En dicha carpeta se deberá crear un archivo con nombre externalparsers.conf, si no existía previamente. Es importante remarcar que la carpeta AppData está **oculta**, por lo que deberemos activar los elementos ocultos en la sección "Vista" de la barra de navegación del explorador de archivos.

Este archivo tiene la función de enlazar formatos no detectados de forma predefinida con sus correspondientes herramientas externas. En este caso, se debe asociar la extensión .mdl correspondiente a los ficheros Vensim a la ruta absoluta donde esté ubicada la herramienta externa, VensimPlugin4SemanticMerge.jar. Concretamente, el contenido de dicho fichero deberá de ser similar al siguiente cuadro de texto.

```
.mdl=java -jar C:\Users\{User}\VensimPlugin4SM\VensimPlugin4SemanticMerge.jar
```

Se recomienda no incluir espacios en blanco o caracteres especiales en las carpetas que conforman la ruta donde se encuentra alojado VensimPlugin4SemanticMerge.jar.

A.1.6. Configuración para el uso con `gmaster`

Actualmente `gmaster` es gratuito. En un futuro se planifica contar con una versión gratuita para uso no comercial.

Si suponemos una instalación típica de `gmaster`, se encontrará una carpeta `gmaster`, en la ruta predeterminada: `C:\Users\{User}\AppData\Local\`, donde `{User}` debe ser sustituido por el nombre del usuario en Windows. Como se ha explicado antes, se debe tener precaución, pues la carpeta `AppData` está **oculta**.

Dentro de esa carpeta se encontrará una denominada `\config`. Se debe crear en dicha carpeta un archivo con nombre `externalparsers.conf`, si no existía ya anteriormente. Este archivo tiene la función de enlazar formatos no detectados de forma predefinida con sus correspondientes herramientas externas para `SemanticMerge`.

En nuestro caso, debemos asociar la extensión `.mdl` a la ruta absoluta del *plugin* para `Vensim`. Concretamente, el contenido del archivo `externalparsers.conf` será similar al siguiente bloque de texto.

```
.mdl=java -jar C:\Users\{User}\VensimPlugin4SM\VensimPlugin4SemanticMerge.jar
```

Se recomienda no incluir espacios en blanco o caracteres especiales en las carpetas que conforman la ruta donde se encuentra alojado `VensimPlugin4SemanticMerge.jar`.

La herramienta de `gMaster` permite modificar el aspecto de la aplicación entre modo iluminado y modo oscuro, tal y como se puede apreciar en la figura A.1. A partir de el botón *Theme* mostrado en la figura, se desplegarán dos etiquetas que permitirán cambiar la visualización de la herramienta entre el modo iluminado y el modo nocturno. Como se puede apreciar en la figura A.2, en el modo iluminado es muy fácil distinguir el cambio seleccionado, sin embargo, en el modo nocturno puede generar complicaciones puesto que se distingue con más dificultad, tal y como se puede apreciar en la figura A.3. Se debe de advertir de que en ciertas ocasiones al cambiar entre los dos modos se producen ciertos *bugs* visuales.

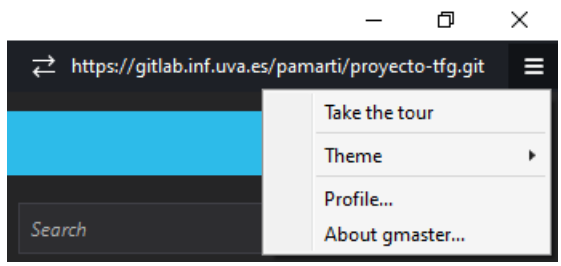

Figura A.1: Cambio de modo visual en gMaster a través del botón Theme.

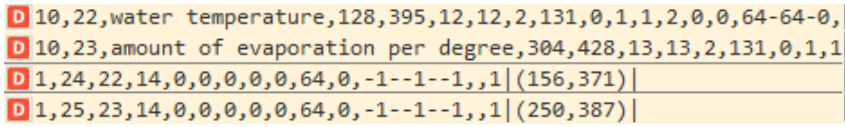

Figura A.2: Selección de cambio en modo iluminado.

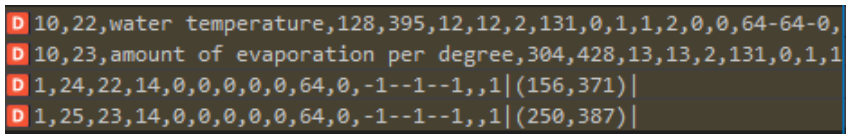

Figura A.3: Selección de cambio en modo oscuro.

A.2. Manual de usuario

Una vez descargados todos los archivos y configurada la integración con `SemanticMerge` o `PlasticSCM` o `gmaster` se puede comenzar a trabajar con la herramienta.

Este manual se centrará en describir el uso integrado en `gmaster`. El proceso hace uso de los tres programas indicados en el Manual de instalación (el Adder, el Eraser y el plugin de Vensim para SemanticMerge). En las subsecciones siguientes se explica cada programa en profundidad.

En este punto se explican de manera general los dos flujos de trabajo más habituales al usar estas herramientas. Un punto a destacar es la recomendación de utilizar ficheros `Vensim` que **no** contengan caracteres especiales y/o espacios en blanco en su nombre, pues la herramienta `SemanticMerge` puede ocasionar problemas en estos casos.

El primer flujo de trabajo describe la situación en la que se desean fusionar dos ramas (merge de una rama a otra).

En la Figura A.4 se muestra gráficamente este flujo de trabajo, suponiendo que se cuente con una rama *master* y una rama llamada *a-branch*. Si el usuario se encuentra en el punto en el que desea fusionar los cambios realizados en *a-branch* con la rama *master*, deberá:

1. Hacer *checkout* de la rama *master*.
2. Aplicar `ViewNamesAndDelimiterAdder.jar` al archivo `.mdl` (ver subsección A.2.1).
3. Hacer *commit* de los cambios realizados por el Adder al `.mdl`. El mensaje de commit podría ser "Prepare to merge". Opcionalmente, aplicar tag (semantic) al *commit* realizado.
4. Hacer *checkout* de la rama *a-branch*.
5. Realizar en esta rama los pasos (2) y (3) de este flujo de trabajo.

6. Realizar el *merge* de *a-branch* a *master* (siguiendo las indicaciones de cómo hacer *merge* en `gmaster`).
7. Para afianzar el merge se realiza un *commit*. Opcionalmente, aplicar tag (semantic) al *commit* del *merge*.
8. Aplicar `ViewNamesAndDelimiterEraser.jar` al archivo `.mdl` (ver subsección A.2.3).
9. Hacer *commit* de los cambios realizados por el Eraser al `.mdl`. El mensaje de *commit* podría ser "Ready to work with `Vensim`".
10. Hacer *push* de los cambios en *master* al repositorio remoto.

El segundo flujo de trabajo habitual describe la situación en la que se quiere comprobar los cambios entre versiones o *commits* en una misma rama.

En la Figura A.5 se muestra gráficamente este flujo de trabajo, suponiendo que se está trabajando en una rama y se desea poder utilizar `SemanticMerge` para ayudar a visualizar las diferencias entre las versiones de un archivo `.mdl`.

La propuesta de flujo de trabajo consiste en que cada modificación realizada a un archivo `.mdl` con `Vensim` de la que se quiera hacer seguimiento en el control de versiones, llevará:

1. Hacer *commit* con los cambios.
2. Aplicar `ViewNamesAndDelimiterAdder.jar` al archivo `.mdl` (ver subsección A.2.1).
3. Hacer *commit* de los cambios realizados por el Adder al `.mdl`. El mensaje de *commit* podría ser la concatenación del mensaje del *commit* anterior con "Prepared for SemanticDiff". Opcionalmente, aplicar tag (semantic) al *commit* realizado.
4. Aplicar `ViewNamesAndDelimiterEraser.jar` al archivo `.mdl` (ver subsección A.2.3).
5. Trabajar con el archivo `.mdl` de la forma habitual con Vensim hasta obtener una nueva versión de la que se quiera hacer seguimiento en el control de versiones.
6. Realizar los pasos (1), (2), (3) y (4) de este flujo de trabajo.

Cuando se desea utilizar `SemanticMerge` para visualizar cambios entre versiones siempre debe hacerse entre los *commits* intermedios (los indicados en la Figura con el tag "semantic").

Así la propuesta consiste en hacer que lo que sería un *commit* en un flujo normal de trabajo con `git` se convierte en dos *commits*, uno después de trabajar con Vensim y uno después del Adder. Después de este (el llamado intermedio o con tag "semantic"), se debe dejar el archivo listo para trabajar nuevamente con Vensim aplicando el Eraser.

Es importante resaltar que no se debe hacer *push* de los *commits* intermedios (los indicados en la Figura con el tag "semantic"), pues esto podría causar problemas con el control de calidad para los archivos `Vensim` que se puede lanzar automáticamente al hacer *push* al repositorio remoto.

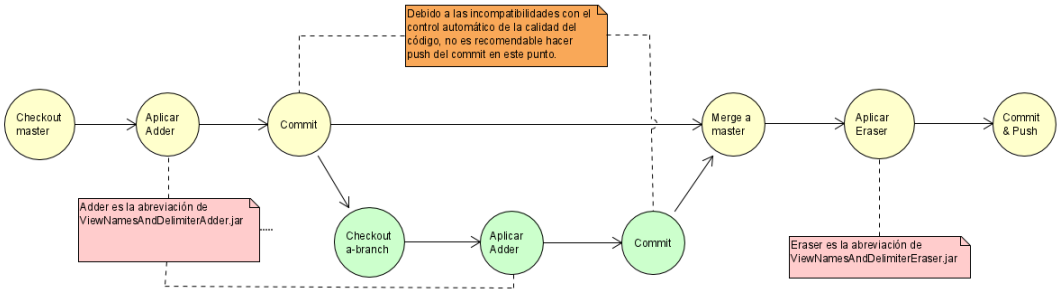

Figura A.4: Diagrama del flujo de trabajo con la herramienta en la fusión de ramas.

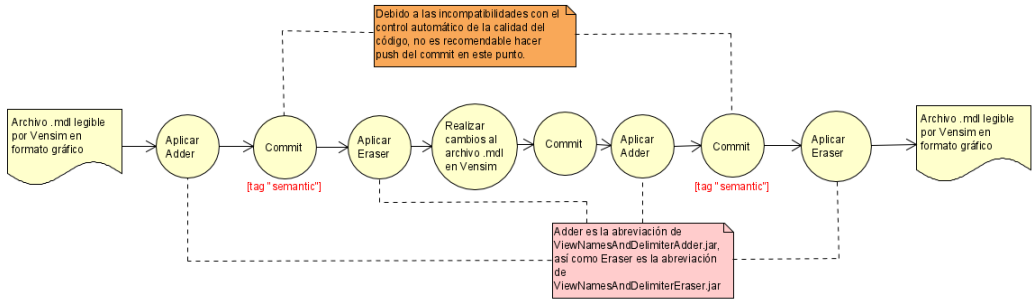

Figura A.5: Diagrama de flujo trabajo con la herramienta en la diferenciación de commits.

A.2.1. El programa auxiliar AddViewNamesAndDelimiters

`AddViewNamesAndDelimiters.jar` modifica los archivos `Vensim` que queremos utilizar, añadiendo los nombres de las vistas en la descripción de las ecuaciones y unos delimitadores de diferentes partes del archivo `.mdl`. Estos delimitadores son necesarios para que la herramienta externa `VensimPlugin4SemanticMerge` sea capaz de leer los archivos `Vensim` y diferenciar sus partes. De esta forma, a la hora de resolver un conflicto en una fusión de ramas será mucho más fácil identificar las causas del mismo y decidir la versión que debe resultar de dicha fusión (*merge*). Es importante señalar que una vez aplicado este programa, `Vensim` no será capaz de leer el archivo resultante de forma gráfica. Para revertir este efecto se utiliza el programa `EraseViewNamesAndDelimiters.jar`, que se explica más adelante (ver subsección A.2.3).

Antes de modificar el archivo, el programa genera automáticamente un archivo (con extensión `.2mdl`) a modo de *backup* del archivo antes de ser procesado. El archivo una vez modificado, conservará tanto el nombre como la extensión originales.

La pantalla inicial de `AddViewNamesAndDelimiters`, tal y como se aprecia en la Figura A.6, consta únicamente de un botón. Dicho botón, `Open file`, como su propio nombre indica, sirve para abrir el archivo a modificar. Sabremos que el archivo se ha cargado cuando la interfaz se modifique tal y como se muestra en la Figura A.7.

En caso de abrir un archivo cuyo formato no es `.mdl` o cualquier otro error, se notificará

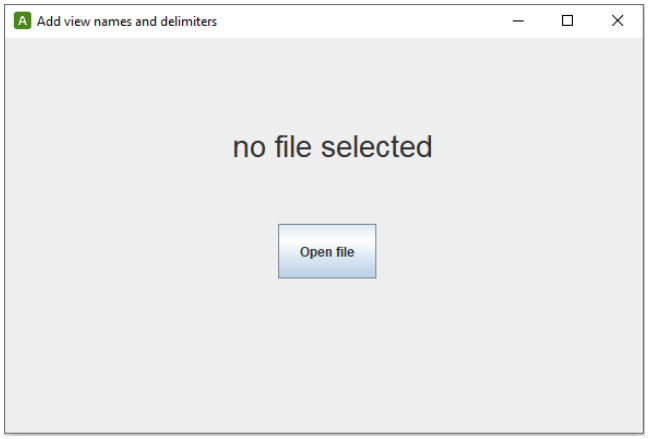

Figura A.6: Pantalla principal de la interfaz de adición de nombres de las vistas y delimitadores.

al usuario a través de un mensaje con un llamativo color rojo.

En caso de seleccionarse un archivo al que ya se la han añadido los nombres de las vistas y los delimitadores, también se indicará al usuario con un mensaje.

Una vez cargado el archivo, se debe pulsar el botón de la izquierda, `Process`, el cual procederá a modificar el archivo y a conservar el *backup* del original con la extensión `.2mdl`. Cuando haya finalizado el proceso, se notificará al usuario con un mensaje en color verde.

En caso de haber seleccionado un archivo que no era el deseado, se podrá hacer uso del botón de `Cancel` para volver a la pantalla inicial (Figura A.6).

Para permitir al usuario que pueda comprobar la ruta completa del archivo que ha seleccionado, se cuenta con el botón `Show absolute path`. Al pulsar la primera vez en dicho botón, se mostrará dicha ruta. Pulsando de nuevo en el mismo botón, se recuperará la vista del nombre del archivo sin ruta.

Finalmente, y con el objetivo de hacer la experiencia de usuario más clara y concisa, se explicará en la parte inferior del nombre del archivo seleccionado qué efecto tiene procesar el archivo con este programa.

Las dos interfaces gráficas de los programas `AddViewNamesAndDelimiters` y `EraseViewNamesAndDelimiters` son prácticamente idénticas, por lo que se debe comprobar el nombre de la etiqueta del programa.

A.2.2. El plugin de Vensim para SemanticMerge dentro de gmaster

Una vez el(los) archivo(s) `.mdl` deseado(s) son modificados añadiendo los nombres de las vistas en la descripción de las ecuaciones y los delimitadores de las partes de un `.mdl`, se

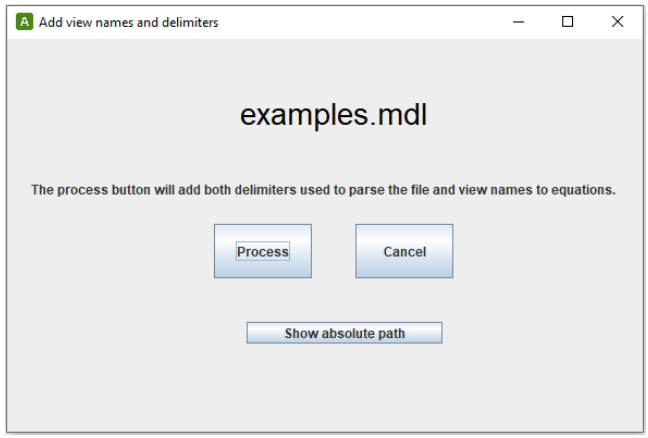

Figura A.7: Pantalla secundaria de la interfaz de adición de nombres de las vistas y delimitadores.

deberán asentar sus cambios en el repositorio mediante un *commit* (ver flujos de trabajo explicados en las Figuras A.4 y A.5). Es importante modificar los archivos con el "Adder", puesto que sin ello `gmaster` no será capaz de reconocer los archivos **semánticamente**, sino únicamente como texto plano. Es decir, se conservará la capacidad de lectura convencional de la herramienta, pero no se aplicarán las mejoras que aporta el *plugin* para `SemanticMerge`.

Ya sea en el flujo descrito para realizar fusión de ramas (*merge*), o en el descrito para permitir la visualización de diferencias entre *commits* en la misma rama, *gmaster* detectará que la extensión del archivo es `.mdl` y utilizará la herramienta externa `VensimPlugin4SemanticMerge` para realizar el análisis semántico. La herramienta nos mostrará las diferencias entre los dos archivos, proporcionándonos información sobre qué significa cada línea modificada.

La Figura A.9 muestra las diferentes partes de la interfaz gráfica de `gmaster`.

En la primera sección señalada con un recuadro rojo y el número 1, podemos observar la primera nueva característica que añade la herramienta. En esta sección se puede observar el recuento de cambios entre dos versiones de un mismo archivo. Como se puede ver en la figura, los cambios se distinguen en cinco apartados principales. Dentro de cada una de estas categorías de cambios, a su vez podemos distinguir el tipo del elemento modificado como puede ser una ecuación y una vista, así como su nombre. Los distintos tipos de cambios se explican a continuación:

- **Adiciones**. Se ha añadido un nuevo elemento al archivo. Está representado por una letra **A** de color verde, representando *added* en inglés.
- **Modificaciones**. Un elemento del archivo ha sido modificado. Está representado por una letra **C** de color azul, representando *changed* en inglés.
- **Eliminaciones**. Se ha eliminado un elemento del archivo. Está representado por una letra **D** de color rojo, representando *deleted* en inglés.

- **Movimientos**. Se ha movido un elemento del archivo dentro del formato textual del mismo. Está representado por una letra **M** de color morado, representando *moved* en inglés.
- **Renombres**. Se ha cambiado el nombre de un elemento del archivo. Está representado por una letra **R** de color morado, representando *renamed* en inglés.

Estas clases de modificaciones se pueden apreciar con mayor claridad en la siguiente figura, A.8:

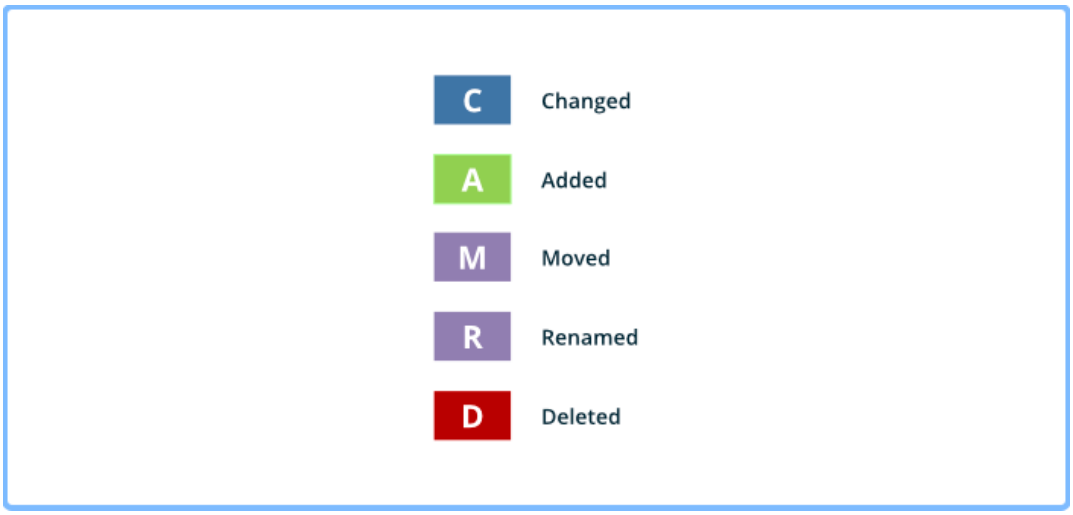

Figura A.8: Tipos de cambios en gMaster y SemanticMerge.

En la segunda sección señalada con un recuadro rojo y el número 2, se observan las dos versiones del archivo que se está analizando, mostrando en la parte izquierda la versión más antigua y en la derecha la actual (en el caso de un *merge* se mostraría la versión de la rama de origen y la versión de la rama de destino). Se observan franjas coloreadas que relacionan los cambios visualmente entre ambas versiones del archivo. Al lado de cada cambio aparece una inicial, la cual indicará la categoría de modificación de entre las cinco posibles explicadas anteriormente.

El recuadro en color azul etiquetado con 2.1 (en la parte de abajo de la sección 2), sirve para cambiar entre el modo convencional de diferencias entre versiones con texto plano, y el modo semántico propio de `SemanticMerge`.

El segundo recuadro en color azul etiquetado con 2.2, permite ver los cambios de una forma más compacta y visual, utilizando únicamente los grafismos asociados a las cinco categorías posibles de modificaciones. En la Figura A.10 se muestra el resultado de seleccionar esa forma de visualizar las diferencias.

Por último, en la tercera sección se aprecian los archivos modificados en la versión del proyecto o *commit* seleccionado.

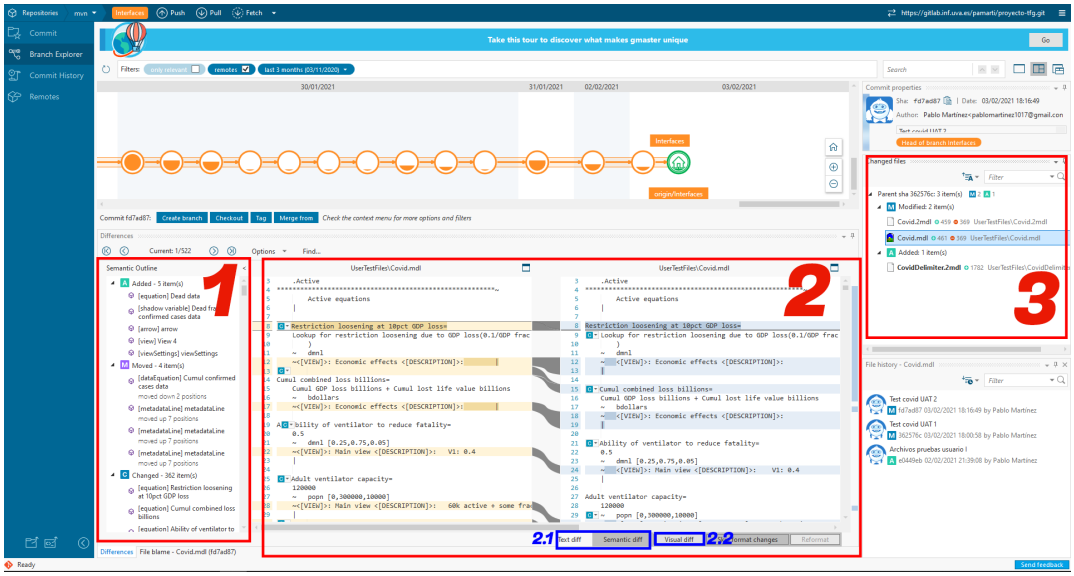

Figura A.9: Interfaz de gmaster y sus correspondientes partes.

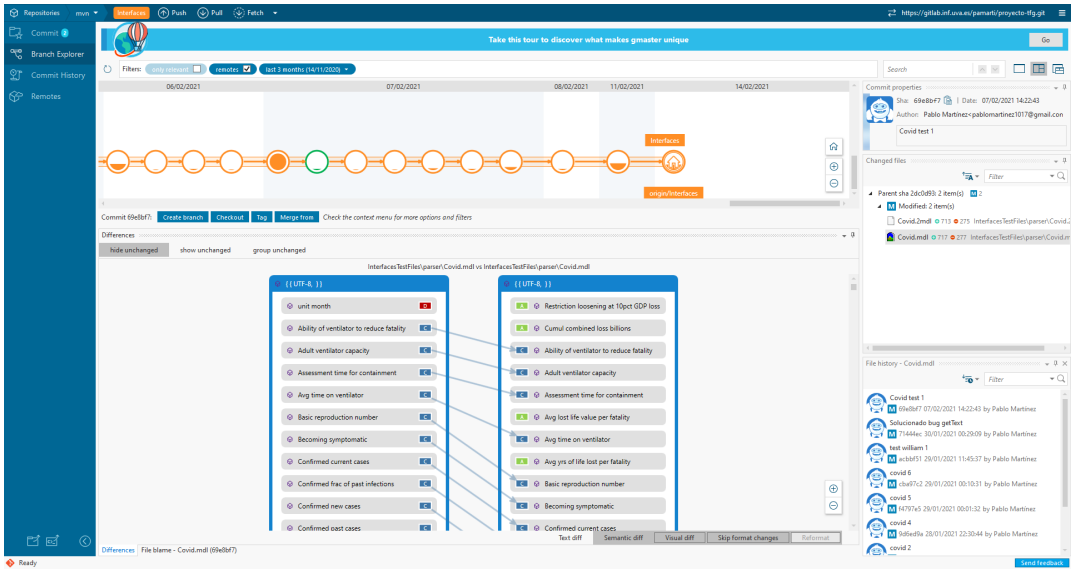

Figura A.10: Visualización de diferencias entre versiones de un .mdl basada en grafismos.

A.2.3. El programa auxiliar EraseViewNamesAndDelimiters

El programa `EraseViewNamesAndDelimiters.jar` modifica los archivos `Vensim` a los que se les haya añadido nombres de vistas en la descripción de las ecuaciones, y delimitadores de las partes de un `.mdl` (resultado de modificar un archivo `.mdl` con el programa

`AddViewNamesAndDelimiters`), eliminando todo lo que se le haya añadido extra.

Según el flujo de trabajo de fusión de ramas, una vez tengamos la rama fusionada y hecho el *commit* de *merge*, debemos utilizar `EraseViewNamesAndDelimiters` y de esta forma, los archivos vuelven a ser visibles y modificables con Vensim. De no hacerlo, no se podrá abrir correctamente el archivo con la interfaz gráfica de `Vensim`. Esto mismo sucede si el flujo de trabajo es el que se utilizaría para ayudar a visualizar las diferencias entre dos versiones de un `.mdl` en la misma rama, hay que volver siempre a un `.mdl` legible por Vensim para poder continuar con el trabajo de programación de los modelos.

La interfaz se comporta igual que la explicada con el primer programa auxiliar (`AddViewNamesAndDelimiters`). Se recuerda que son prácticamente idénticas, por lo que se debe comprobar el nombre de la etiqueta del programa. La explicación textual del efecto del botón *Process* también pretende ayudar en esta diferenciación.

`EraseViewNamesAndDelimiters` también se encarga de generar automáticamente un archivo (con extensión `.2mdl`) a modo de *backup* antes de realizar modificaciones. Pero en este caso lo llamará añadiendo la palabra Delimiter al nombre del archivo original (`NombreDeArchivoDelimiter.2mdl`) para no reemplazar el *backup* creado con `AddViewNamesAndDelimiters`. El archivo una vez modificado, conservará tanto el nombre como la extensión originales.

Las Figuras A.11 y A.12 muestran las pantallas de `EraseViewNamesAndDelimiters`.

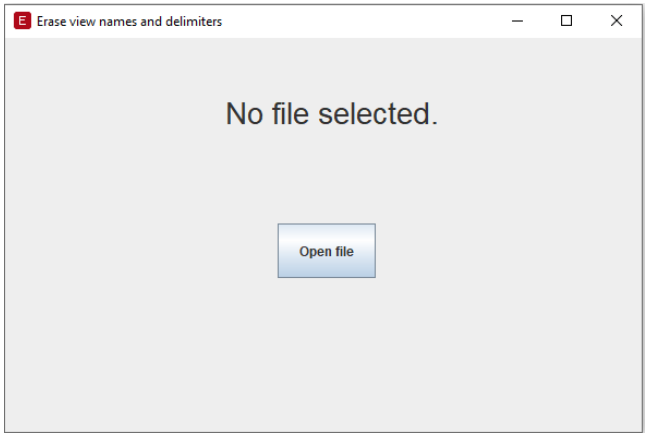

Figura A.11: Pantalla principal de la interfaz de eliminación de delimitadores.

Es muy útil en el repositorio `git` añadir el archivo `.gitignore` indicando que los archivos de *backup* no se tengan en cuenta en el control de versiones. El siguiente cuadro de texto muestra el contenido de `.gitignore` típico para un proyecto de programación de modelos con Vensim:

```
# Be careful and do not name an important file with temporal file extension
*.tmp
```

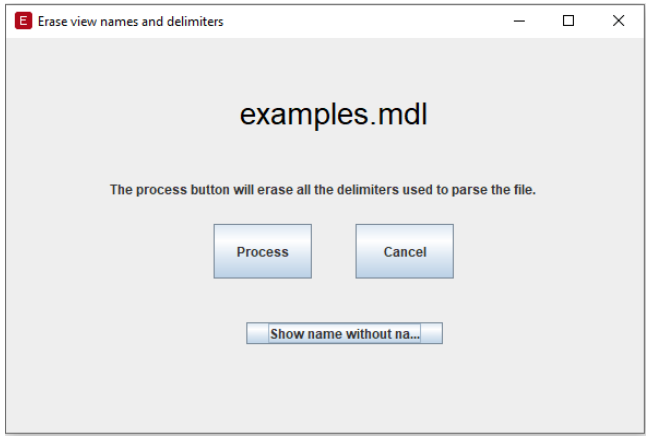

Figura A.12: Pantalla secundaria de la interfaz de eliminación de delimitadores.

```
# Word temporary
~$*.doc*

# Word Auto Backup File
Backup of *.doc*

# Excel temporary
~$*.xls*

# Excel Backup File
*.xlk

# Vensim mdl Backup File
*.2mdl

# Vensim simulation
*.vdf
```

A.2.4. Parámetros de las líneas de definición de variables en las vistas.

Aunque muchos de los parámetros de las líneas asociadas a las variables en la parte de definición de las vistas son triviales, es importante poder conocer qué representan cada uno de los campos de dichas líneas.

Podemos diferenciar dos claras estructuras diferentes [54]. Los campos de las variables marcados con **(*)** se han considerado no triviales e importantes a tener en cuenta a la hora de resolver un conflicto, ya que contienen datos importantes en la estructura de la vista y no son simples parámetros que configuran aspectos triviales del diseño como pueden ser el color o la forma de una variable:

- **Variables**. Esta estructura representa las variables, incluyendo sus variaciones como las variables sombra. También se incluye en este grupo a las válvulas o *valves* y a los comentarios. Se componen de:
  - **1**. n → Código numérico que indica el tipo de variable. Los códigos 1, 10, 11 y 12 representan a flechas, variables, *valves* y comentarios respectivamente **(*)**.
  - **2**. id → Código numérico ascendente que indica la posición respecto a otras variables en que se introdujo la variable a la vista **(*)**.
  - **3**. name → Nombre de la variable. Si fuese un '0', el nombre de la variable aparecería aislado en la línea siguiente. En el caso de las *valves* será un número irrelevante para el usuario y en el caso de los comentarios será un número que represente a la figura del comentario. En el caso de variables *triviales*, su campo es trivial, valga la redundancia **(*)**.
  - **4,5**. x, y → Posición de la variable en el plano **(*)**.
  - **6,7**. w, h → Ancho y alto de la variable en el plano **(*)**.
  - **8**. sh → Forma alrededor de la palabra y características relativas. Estas se disponen en un conjunto de bits que es representado en forma decimal.
  - **9**. bits → Conjunto de bits que indica si pueden entrar y/o salir flechas de la variable **(*)**.
  - **10.** hid → Indica si la variable está oculta. Un valor distinto de cero indica el nivel de oculta en que se encuentra la variable. **(*)**
  - **11**. hasf → Indica si la variable tiene una fuente de texto especial.
  - **12**. tpos → Indica la posición del texto respecto a la forma que encierra la variable.
  - **13**. bw → Indica el ancho del borde de la caja o forma que rodea a la variable.
  - **14**. nav1 → Indica el número de vista al que puede redireccionar la variable.
  - **15**. nav2 → En el caso de que haya más de 255 vistas, el número de vista pasa a calcularse con: $nav1 + 256 * nav2$.
  - **16**. box → Color del borde de la caja o forma que rodea a la variable.
  - **17**. fill → Color de relleno de la caja o forma que rodea a la variable.
  - **18**. font → Tipografía de la variable.
  - **19-24**. Constantes numéricas desconocidas.
  - **25.** visualInfo → Campo utilizado para anexar el nombre de la variable en caso de estar definido en la línea siguiente.
- **Flechas**. Esta estructura representa las uniones y relaciones entre el resto de variables. Se componen de:
  - **1**. n → Código numérico que indica el tipo de variable. En las flechas o *arrows* siempre será 1 **(*)**.
  - **2**. id → Código numérico ascendente que indica la posición respecto a otras variables en que se introdujo la variable a la vista **(*)**.
  - **3,4**. from, to → Ids de las variables de donde sale la flecha y a donde llega respectivamente **(*)**.

- **5**. shape → Forma de la flecha (línea recta, curvada...etc).
- **6**. hid → Indica si la flecha está oculta. Un valor distinto de cero indica el nivel de oculta en que se encuentra la flecha **(*)**.
- **7**. pol → Indica la polaridad de la flecha.
- **8**. thickness → Indica el grosor de la flecha. Un valor de más de 20 unidades indica que la flecha utiliza dos líneas paralelas.
- **9**. hasf → Indica cambios en el color y fuente que la vista trae como predeterminados.
- **10**. dtype → Indica el delay de la flecha.
- **11**. res → Valor reservado, debería ser 0.
- **12**. color → Color de la flecha o "$-1--1--1$".
- **13**. font → Fuente o color de la flecha. Este campo puede permanecer vacío.
- **14,15**. nc, pointlist → El primer valor representa el número de puntos intermedios que tiene la flecha. El segundo valor es la vista de las coordenadas de dichos puntos intermedios **(*)**.

En el caso de el apartado catalogado como "metadatos", esta sección apenas varía y su tamaño es muy reducido en comparación del tamaño total del fichero `.mdl`. No parece existir documentación sobre esta última parte, pero son líneas referentes a la configuración del archivo y similares. Debido a su pequeño tamaño y que la gramática permite separar línea por línea este apartado, se considera que no debería generar problemas de comprensión. No obstante, se enumeran a continuación ciertas líneas identificadas con su correspondiente significado:

- Las líneas que comienzan por **1** indican archivos de simulaciones.
- La línea que empieza por **5** indica la última variable seleccionada.
- Existen líneas que simbolizan los parámetros de configuración del archivo global.

A.2.5. Advertencias sobre posibles cambios que puede realizar el programa Vensim en los ficheros

Se listan a continuación una serie de advertencias acerca del funcionamiento de las herramientas de `SemanticMerge` y `gMaster` sobre cómo pueden llegar a afectar al formato textual de los archivos `.mdl`:

- En la parte donde se listan las variables asociadas a una vista, uno de los parámetros de cada variable es su orden de aparición en el texto. Por ello, ciertas acciones como eliminar una variable o moverla de vista, modificarán todas las líneas que representan a variables en esa vista con un orden de aparición mayor.

- Ciertas acciones provocan que se inserten o eliminen líneas de metadatos en el final del archivo.
- Al mover variables entre vistas, la definición de la variable en la parte de ecuaciones aparecerá como cambio, sin embargo, en la parte de definición en las vistas aparecerá como eliminación en la vista anterior e inserción en la nueva.
- En ciertas ocasiones, `Vensim` transforma la definición de una ecuación de forma extendida (en tres líneas) a forma compactada (una única línea) y viceversa.
- Borrar una vista sin borrar previamente las variables de dicha vista hace que las variables no se eliminen y queden como variables vacías en el formato textual del archivo.
- En ciertas ocasiones, `Vensim` aumenta el número de parámetros de una variable, generalmente con parámetros con valor cero.
- En ciertas ocasiones, mover variables modifica ciertos elementos que `Vensim` denomina válvulas.
- Eliminar ciertas partes de un modelo que hagan que el modelo no sea correcto respecto al funcionamiento de `Vensim` puede acarrear problemas en la definición de algunas variables.
- En el caso de las variables numéricas como comentarios o *valves*, el campo del nombre lleva un valor numérico asignado que según la documentación oficial, debe de ser ignorado [53]. Este valor cambia tras realizar ciertas operaciones como adiciones o eliminaciones de elementos.
- El renombrado de variables se debe de hacer desde el modo gráfico de Vensim. En caso de realizarlo desde el modo textual, se debe cambiar el nombre en la definición de las ecuaciones y en su aparición en su vista concreta.
- Abrir el fichero con otra versión de Vensim puede provocar movimientos o redimensiones que pueden provocar una gran cantidad de cambios al alterar los parámetros de posición de muchas variables y/o flechas.

A.2.6. Advertencias sobre el nombrado de variables

Existen ciertas formas de nombrar a las variables que hacen que el programa falle, consecuencia de una mala formación del árbol descriptor del fichero en el archivo `YAML` o ciertos caracteres especiales no soportados por el sistema de codificación del proyecto, `UTF-8`. Los nombrados de variable que causan o pueden llegar a causar problemas son los siguientes:

- Utilizar el carácter ':' seguido de un espacio en blanco.
- Acabar el nombre de la variable en uno o varios espacios en blanco.
- Utilizar comillas, salvo en los casos en los que las comillas envuelvan todo el nombre de la variable. Se recomienda utilizar comillas dobles.
- Utilizar caracteres especiales no soportados.

A.2.7. Enlaces de interés

En caso de querer conocer más acerca del funcionamiento específico de `SemanticMerge`, `PlasticSCM` y/o `gMaster` o cómo ambas herramientas incorporan herramientas o *parsers* externos, se recomienda consultar [27], [37], [39], [40], [43], [47], [53] y [54] en la bibliografía del proyecto.

A.3. Manual de mantenimiento

En esta sección se explican ciertos aspectos y pautas a seguir para posibles futuros desarrolladores que vayan a realizar labores de mantenimiento o de actualización en caso de que algunos de los programas utilizados reciba alguna actualización importante que haga a este proyecto obsoleto.

A.3.1. Actualización de la gramática y del parser

Existe la posibilidad de que se produzca una actualización de `Vensim` que incluya nuevos elementos de gran importancia y que mejoren el proceso de modelar sistemas, y que al ser algo nuevo, el proyecto no sea capaz de reconocer dichos elementos y por lo tanto no funcione. Los pasos a seguir serían los siguientes:

Actualización de la gramática ANTLR4

Inicialmente se debería ampliar la gramática existente con las nuevas incorporaciones, con cuidado de no inducir fallos en la gramática de partida. Como `ANTLR4` no tiene una forma sencilla de realizar *tests* automáticos, se recomienda probar con una variedad de archivos que tengan estructuras diferentes. Se pueden encontrar ejemplos en la carpeta `VensimExampleModels` en el repositorio del proyecto. Para hacer más fácil el proceso de depuración de errores se recomienda imprimir el árbol semántico generado, bien por salida estándar de consola o por interfaz gráfica.

En el caso de salida por consola bastaría con la línea de código:

```
System.out.println(tree.toStringTree(parser));
```

En el caso de salida por interfaz gráfica sería:

```
JFrame frame = new JFrame("Antlr AST");
JPanel panel = new JPanel();
TreeViewer viewer = new TreeViewer(Arrays.asList(parser.getRuleNames()),
```

```
                                                    tree);
viewer.setScale(0.5);  //SCALE
panel.add(viewer);
frame.add(panel);
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.pack();
frame.setVisible(true);
```

Actualización del parser

Posteriormente vendría el trabajo de la actualización del parser. En función de la zona en que se encuentre la nueva característica se deberá elegir uno de los tres *visitors* que descienden del *visitor* principal, estos son: `EquationsVisitor`, `SketchesVisitor` y `GraphsMetadataVisitor`. La clave para generar correctamente el árbol `YAML` consiste en mantener una correcta cuenta de la línea y el número de carácter asociados a cada nodo u hoja del árbol semántico generado a través de la gramática. Esto se hace con las variables `locationSpanStartEq` e `initCharEq`, las cuales se encuentran en todos los *visitors*. El resto del proceso se puede realizar basándose en cómo han sido realizados los otros tres *visitors*.

A.3.2. Proceso de testing

El proyecto se realizó utilizando el framework de `Maven`, por lo que se pudieron automatizar los tests. Para realizar el proceso de *testing* son necesarios los comandos `mvn clean compile` y `mvn test` en ese orden, y pasarán a realizarse todos los tests unitarios. En caso de que se quiera probar el correcto funcionamiento con `SemanticMerge`, se deberá descomentar el comando que lanza el programa de `SemanticMerge` al final de cada uno de los tests unitarios asociados a los *visitors*. Este comando tiene la siguiente estructura:

```
Process process = Runtime.getRuntime()
    .exec("C:\\Users\\Propietario\\AppData\\Local\\semanticmerge.
    \\semanticmergetool.exe"
    + " --source=VensimExampleModels/SHODOR/WILIAMori.mdl
    --destination=VensimExampleModels/SHODOR/WILIAMori.mdl
    --externalparser=\"-jar target/mvntfg-1.0-jar-with-dependencies.jar\""
    + " --virtualmachine=
    \"C:\\Program Files\\Java\\jdk-11.0.8\\bin\\java.exe\"");
```

En esta situación, se deberá utilizar el comando `mvn assembly:assembly` entre los dos comandos anteriores. Tras realizar el comando de `assembly`, saltará un error diciendo que no se ha encontrado el archivo `JAR`, y es normal, puesto que se ha intentado realizar el comando y generar dicho `JAR` simultáneamente. Si se ejecuta el comando `mvn test` posteriormente, el programa se lanzará correctamente. Si se desea utilizar el archivo `JAR` de forma definitiva, se

deberá buscar el archivo `mvntfg-1.0-jar.with-dependencies.jar` en la carpeta `target` y cambiarle el nombre a `VensimPlugin4SemanticMerge.jar` para poder identificarlo mejor en un futuro.

A.3.3. Actualizaciones de SemanticMerge y gMaster.

En caso de que se produzca un gran cambio en cómo realizar *parsers* externos para estas dos herramientas, se debería buscar información en la documentación actualizada, concretamente en [27] y [40].

Apéndice B

Resumen de enlaces adicionales

Los enlaces útiles de interés en este Trabajo Fin de Grado son:

- Repositorio del proyecto: `https://github.com/HylianPablo/VensimPlugin4SemanticMerge`.

Bibliografía

[1] andrew1234. Java Swing JFileChooser. https://www.geeksforgeeks.org/java-swing-jfilechooser/, 2018. Accesed: 2021-19-01.

[2] Apache. Maven. https://maven.apache.org/, 2020. Accesed: 2020-9-23.

[3] Baeldung. Java with ANTLR. https://www.baeldung.com/java-antlr, 2020. Accesed: 2020-9-15.

[4] Basecamp. Basecamp project manager software. https://basecamp.com/new, 2021. Accesed: 2021-24-03.

[5] Daniel Bazaco. Final Vensim grammar in ANTLR 4, developed for this project. https://gitlab.inf.uva.es/danbaza/tfg-sonarvensim/-/blob/master/src/main/antlr/Model.g4, 2019. Accesed: 2020-9-15.

[6] Atlassian Bitbucket. Flujo de trabajo de Gitflow. https://www.atlassian.com/es/git/tutorials/comparing-workflows/gitflow-workflow, 2021. Accesed: 2021-20-04.

[7] Rocket Chat. Rocket Chat. https://rocket.chat/, 2020. Accesed: 2020-9-23.

[8] CoronA. ANTLR 4.5 - Mismatched Input 'x' expecting 'x'. https://stackoverflow.com/questions/29777778/antlr-4-5-mismatched-input-x-expecting-x, 2015. Accesed: 2020-9-18.

[9] Departamento de Informática. Archivos Cloud Vensim. https://cloud.infor.uva.es/index.php/s/3r85R4oavbe3ivk, 2020. Accesed: 2020-9-30.

[10] Universidad de Valladolid. Guía docente de la signatura. Trabado de fin de grado (mención Ingeniería de Software). https://www.inf.uva.es/wp-content/uploads/2016/06/G46976.pdf, 2020. Accesed: 2020-10-2.

[11] Universidad de Valladolid. Preguntas frecuentes. https://www.uva.es/export/sites/uva/2.docencia/2.02.mastersoficiales/2.02.13.preguntasfrecuentes/index.html, 2020. Accesed: 2020-10-2.

[12] eclass. ¿Cuáles son los eventos de Scrum? https://blog.eclass.com/cuales-son-los-eventos-de-scrum-conocelos-aqui-0, 2020. Accesed: 2021-22-01.

[13] Norberto Figuerola. Riesgos: Plan de Mitigación vs Plan de Contingencia vs Fallback Plan. https://articulospm.files.wordpress.com/2015/06/riesgos-plan-mitigacion-vs-plan-contingencia-vs-fallback-plan.pdf, 2015. Accesed: 2020-9-26.

[14] Tobi G. Java Special Characters in RegEx. https://stackoverflow.com/questions/14134558/list-of-all-special-characters-that-need-to-be-escaped-in-a-regex, 2019. Accesed: 2020-11-11.

[15] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *European Conference on Object-Oriented Programming*, pages 406–431. Springer, 1993.

[16] Research Gate. LOCOMOTION H2020. https://www.locomotion-h2020.eu/, 2020. Accesed: 2020-9-16.

[17] Research Gate. MEDEAS. https://www.medeas.eu/#home, 2020. Accesed: 2020-9-16.

[18] GitLab.org. GitLab. https://gitlab.com/gitlab-org, 2020. Accesed: 2020-9-23.

[19] gMasterRealeaseNotes. release notes.

[20] Object Management Group. Unified Modeling Language. https://www.uml.org/, 2021. Accesed: 2021-11-02.

[21] James P. Houghton. test-models. https://github.com/SDXorg/test-models, 2020. Accesed: 2020-9-30.

[22] Joel Francia Huambachano. ¿Qué es Scrum? https://www.scrum.org/resources/blog/que-es-scrum, 2017. Accesed: 2021-22-01.

[23] Change Vision Inc. Astah Professional. https://astah.net/products/astah-professional/, 2021. Accesed: 2021-11-02.

[24] La información. ¿Cuál es el coste real de tener un trabajador para una empresa? https://www.lainformacion.com/practicopedia/cual-es-el-coste-real-de-un-trabajador-para-una-empresa/6491464/#:~:text=Cotizaciones%20a%20la%20Seguridad%20Social,26.300%20euros%20a%20la%20empresa., 2019. Accesed: 2020-9-26.

[25] Jitsi.org. Jitsi Meet. https://meet.jit.si/, 2020. Accesed: 2020-9-23.

[26] Bart Kiers. If/else statements in ANTLR using listeners. https://stackoverflow.com/questions/15610183/if-else-statements-in-antlr-using-listeners, 2013. Accesed: 2020-9-15.

[27] Sergio L. Using external parsers with gMaster. http://blog.gmaster.io/2018/03/using-external-parsers-with-gmaster.html, 2018. Accesed: 2020-14-11.

[28] Pablo Santos Luaces. charposition. https://github.com/PlasticSCM/charposition, 2016. Accesed: 2020-31-10.

[29] Pablo Martínez López. Gramática en ANTLR4 para Vensim. https://gitlab.inf.uva.es/pamarti/proyecto-tfg/-/blob/master/src/main/antlr4/es/uva/inf/grammar/Grammar.g4, 2020. Accesed: 2020-9-29.

[30] Pablo Martínez López. SimpleJavaParser-SemanticMerge. https://github.com/HylianPablo/SimpleJavaParser-SemanticMerge, 2020. Accesed: 2020-9-29.

[31] McGraw-Hill. Business Dynamics. http://www.mhhe.com/business/opsci/sterman/models.mhtml, 2001. Accesed: 2020-9-30.

[32] Microsoft. Visual Studio Code. https://code.visualstudio.com/, 2020. Accesed: 2020-9-23.

[33] migraciones y seguridad social Ministerio de trabajo. Boletín Oficial del Estado. III. Otras disposiciones. https://www.boe.es/boe/dias/2019/06/03/pdfs/BOE-A-2019-8222.pdf, 2019. Accesed: 2020-9-26.

[34] MiryamGSM. External parser sample. https://github.com/PlasticSCM/external-parser-sample/tree/master-SCM20098, 2017. Accesed: 2020-8-28.

[35] picodotdev. Cómo ejecutar un proceso del sistema con Java. https://picodotdev.github.io/blog-bitix/2016/03/como-ejecutar-un-proceso-del-sistema-con-java/, 2016. Accesed: 2020-16-12.

[36] PlasticSCM. SemanticMerge pricing. https://users.semanticmerge.com/Checkout, 2020. Accesed: 2020-9-26.

[37] Plastic SCM. gMaster - Git GUI. https://gmaster.io/, 2021. Accesed: 2021-21-01.

[38] Shodor. Vensim models. http://shodor.org/talks/ncsi/vensim/, 2005. Accesed: 2020-9-19.

[39] Códice Software. Custom languages in semantic version control. http://blog.plasticscm.com/2015/09/custom-languages-in-semantic-version.html, 2015. Accesed: 2021-15-02.

[40] Códice Software. External parsers. https://semanticmerge.com/documentation/external-parsers/external-parsers-guide, 2016. Accesed: 2020-9-19.

[41] Códice Software. Advanced version control - Pocket guide. https://www.plasticscm.com/documentation/advanced-version-control-guide#two-way-merge, 2020. Accesed: 2020-9-17.

[42] Códice Software. Cygnus Solutions. https://www.cygwin.com/, 2020. Accesed: 2021-24-02.

[43] Códice Software. SemanticMerge 2.0. https://semanticmerge.com/, 2020. Accesed: 2020-7-11.

[44] Códice Software. SemanticMerge features. http://www.semanticmerge.com/features, 2020. Accesed: 2020-9-17.

[45] Códice Software. SemanticMerge intro guide. https://semanticmerge.com/documentation/intro-guide/semanticmerge-intro-guide, 2020. Accesed: 2020-9-17.

[46] Códice Software. XDiff and XMerge. https://www.plasticscm.com/features/xmerge, 2020. Accesed: 2020-9-17.

[47] Códice Software. PlasticSCM. https://www.plasticscm.com/?utm_source=plasticscm-blog&utm_medium=blog-info&utm_content=whoweare, 2021. Accesed: 2021-15-02.

[48] Códice Software. PlasticSCM - The Distributed Version Control for Big Projects. https://www.plasticscm.com/, 2021. Accesed: 2021-2-10.

[49] Sonarqube. Code Quality and Code Security. https://www.sonarqube.org/, 2021. Accesed: 2021-27-02.

[50] Saumitra Srivastav. Antlr4 - Visitor vs Listener Pattern. https://saumitra.me/blog/antlr4-visitor-vs-listener-pattern/, 2017. Accesed: 2020-9-18.

[51] Johannes Link Matthias Merdes Marc Philipp Juliette de Rancourt Christian Stein Stefan Bechtold, Sam Brannen. JUnit 5 User Guide. https://junit.org/junit5/docs/current/user-guide/, 2020. Accesed: 2020-23-12.

[52] Ventana Systems. Defined and shadow variables. https://www.vensim.com/documentation/22890.htm, 2020. Accesed: 2020-10-6.

[53] Ventana Systems. Sketch Information. https://www.vensim.com/documentation/ref_sketch_format.html, 2020. Accesed: 2021-2-21.

[54] Ventana Systems. Sketch Objects. https://www.vensim.com/documentation/index.html?23275.htm, 2020. Accesed: 2020-10-3.

[55] Ventana Systems. Variable types. https://www.vensim.com/documentation/ref_variable_types.htm, 2020. Accesed: 2020-9-16.

[56] Ventana Systems. Vensim. https://vensim.com/, 2020. Accesed: 2020-23-12.

[57] Ej Technologies. Java Profiler - JProfiler. https://www.ej-technologies.com/products/jprofiler/overview.html, 2021. Accesed: 2021-24-02.

[58] Linus Torvalds. Git. https://git-scm.com/, 2020. Accesed: 2020-9-23.

[59] TutorialsPoint. Compiler Design - Symbol Table. https://www.tutorialspoint.com/compiler_design/compiler_design_symbol_table.htm#:~:text=Symbol%20table%20is%20an%20important,synthesis%20parts%20of%20a%20compiler., 2020. Accesed: 2020-10-7.

[60] Wangdq. How to Display ANTLR Tree GUI. https://stackoverflow.com/questions/23809005/how-to-display-antlr-tree-gui, 2014. Accesed: 2020-9-24.

[61] Wikipedia. Swing (biblioteca gráfica). https://es.wikipedia.org/wiki/Swing_(biblioteca_gr%C3%A1fica), 2021. Accesed: 2021-20-01.