Universidad de Valladolid

ESCUELA DE INGENIERÍA INFORMÁTICA

GRADO EN INGENIERÍA INFORMÁTICA
Mención en Ingeniería del Software

Definición y comprobación de estándares de calidad en la programación de IAMs en Vensim

---

Alumno: Daniel Bazaco Velasco

Tutor: Yania Crespo González-Carvajal

A mis padres, por todo el apoyo que me han dado

Agradecimientos

A mis padres, que siempre me apoyaron. Sin sus consejos no habría sobrevivido a mi estancia en Madrid durante la cuarentena.

A mis amigos y compañeros de carrera, con los que he crecido día a día y que me han motivado a aprender más y a ser más curioso.

A mi tutora Yania, por todo el esfuerzo que dedica como profesora y por su ayuda como mediadora a lo largo de este proyecto.

Resumen

El objetivo de este TFG es desarrollar un plugin para Sonarqube que analice la calidad de Integrated Assessment Models (IAMs) creados con Vensim. El TFG se realiza como parte de los proyectos europeos H2020 MEDEAS y LOCOMOTION, que tratan de desarrollar un modelo para facilitar la transición a un mundo más sostenible.

El plugin comprueba una serie de estándares de calidad, que fueron acordados con los desarrolladores del modelo. Los resultados de los análisis ejecutados se almacenan en la plataforma de Sonarqube, desde donde pueden ser consultados por los modeladores/programadores.

El proyecto fue desarrollado con Java, ANTLR4 y las librerías de Sonarqube, siguiendo una metodología ágil basada en Scrum.

Abstract

The purpose of this proyect is to develop a plugin for Sonarqube that analyzes the quality of Integrated Assesment Models (IAMs) created with Vensim. This final degree project is part of the European projects H2020 MEDEAS and LOCOMOTION, which aim to develop a model to support the transition to sustainability.

The plugin checks several quality standards, that were agreed with the developers of the model. The results of the analysis are stored in the Sonarqube platform, where they can be consulted by modellers and programmers.

The project was developed with Java, ANTLR4 and the Sonarqube libraries following an agile methodology based on Scrum.

Índice general

Lista de Figuras

Lista de Tablas

Capítulo 1

Introducción

1.1. Contexto

Este TFG se realizó como parte de una beca de colaboración con el proyecto europeo MEDEAS [23] (Modelling the Energy Development under Environmental And Socioeconomic constraints), que forma parte del programa Horizon 2020 [9].

El objetivo de MEDEAS es crear un modelo que permita realizar simulaciones para analizar el efecto de diferentes políticas energéticas, y así facilitar una transición a las energías renovables. Este tipo de modelos se denominan IAMs [17] (Integrated Assessment Model), y combinan modelos económicos, energéticos y tecnológicos para realizar simulaciones y previsiones.

En el proyecto MEDEAS participan grupos de investigación de múltiples países. A lo largo de este TFG se ha estado en contacto con el GEEDS [12] (Grupo de Energía, Economía y Dinámica de Sistemas) como cliente principal. Este grupo de investigación forma parte de la UVa y se encargó de la creación y programación del modelo MEDEAS.

El trabajo de fin de grado se realizó en un periodo de transición del proyecto MEDEAS al proyecto LOCOMOTION [21] (Low-carbon society: an enhanced modelling tool for the transition to sustainability), cuyo objetivo es mejorar los modelos de MEDEAS para hacerlos más precisos, robustos y transparentes. La Universidad de Valladolid es el socio coordinador del nuevo proyecto LOCOMOTION, liderado por el grupo GEEDS, con la colaboración de profesores del Departamento de Informática.

Los modelos de MEDEAS y LOCOMOTION se crean usando Vensim [77], un software gráfico para el desarrollo y análisis de modelos. Aunque con el objetivo de promover el acceso libre y la transparencia se pretende traducir el modelo a Python.

LOCOMOTION está dividido en varios subproyectos, denominados working packages. El TFG se desarrolló como parte del working package 9: Technical coordination and quality as-

surance (Coordinación técnica y garantía de la calidad). El modelo está creado por diferentes grupos, por ello para mantener la calidad del modelo es necesario determinar una serie de reglas y estándares. Para facilitar esta tarea es necesaria una herramienta que compruebe de manera automática los estándares establecidos por la coordinación técnica con el acuerdo de todos los socios. Como parte de la propuesta de TFG y de la descripción de las tareas a realizar en la convocatoria de beca asociada estaba establecido que se usaría Sonarqube.

1.2. Introducción a Vensim

Vensim es un programa que permite crear modelos de dinámicas de sistemas. Se caracteriza por su eficiencia, flexibilidad y uso de algoritmos avanzados. Los modelos se crean mediante una interfaz gráfica y se pueden almacenar en formato binario (`.vpm`,`.vpm` y `vpf`) o en texto plano(`.mdl`). Los modelos se suelen desarrollar en texto plano y se publican en formato binario [78].

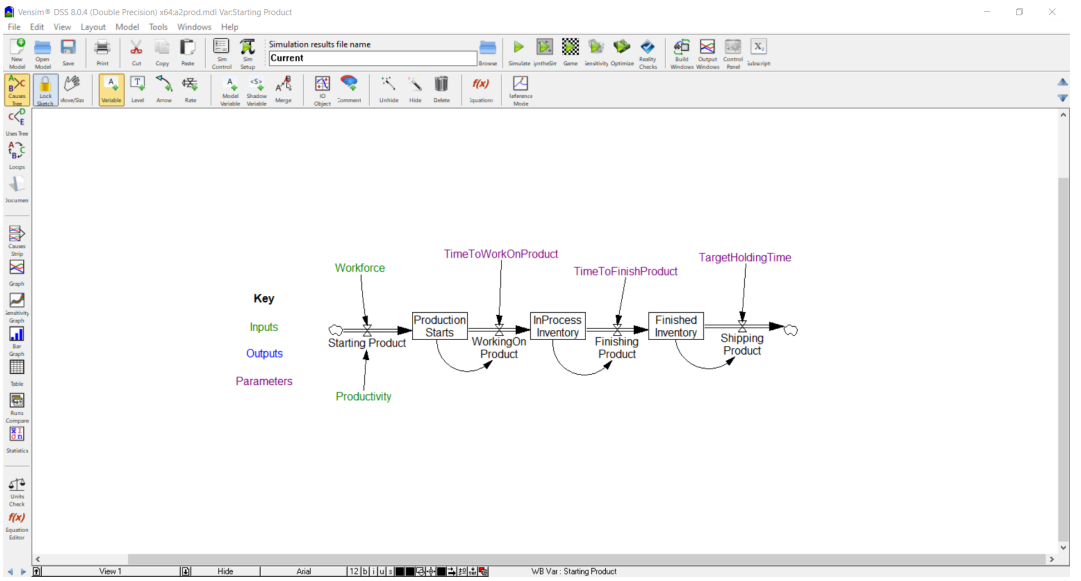

Figura 1.1: Ejemplo de un modelo Vensim.

El modelo está compuesto por módulos. Cada uno es un fichero `.mdl`. Los módulos están formados por una serie de símbolos interconectados. Cada símbolo representa un aspecto del modelo, y está definido por una o varias ecuaciones. Los modelos muestran cómo evoluciona un sistema durante un periodo. Por ello se ejecutan de forma iterativa. En cada iteración el tiempo avanza una cantidad fija y se recalcula el valor de algunos de los símbolos (ver 1.2.1).

Los modelos pueden estar compuestos por diferentes vistas. Estas permiten dividir el modelo en módulos, donde solo algunos símbolos se comunican con símbolos de otras vistas. Los símbolos también contienen una serie de metadatos, como las unidades, comentarios, grupo, etc.

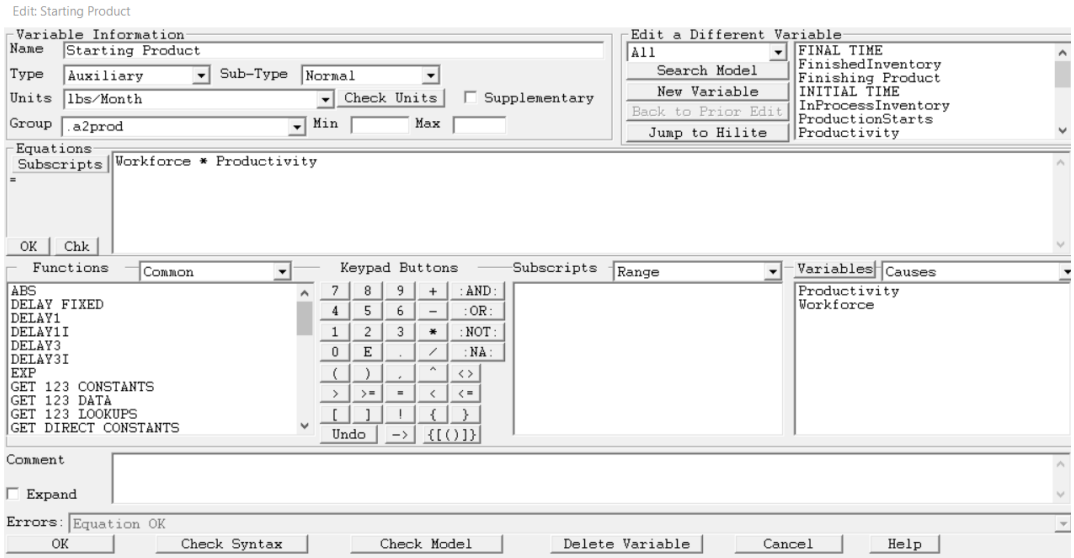

Figura 1.2: Ventana de definición de un símbolo en Vensim.

1.2.1. Tipos básicos

Los símbolos de un modelo pueden ser de varios tipos [75]. Estos son relevantes para los estándares de nombrado y programación que se establecen y se harán múltiples referencias a estos a lo largo de la memoria. Los tipos son:

- **Constantes**: Símbolos cuyo valor no varía en función del tiempo u otros parámetros. Durante el modo simulación Vensim permite al usuario modificar su valor para ver sus efectos en el modelo.
  - **Constantes inmutables (Unchangeable constants)**: Son un subtipo de constantes. A diferencia de las normales no se permite modificarlas dentro del modo simulación.
- **Variables/Auxiliares**: Símbolos cuyo valor cambia a lo largo de la simulación.
  - **Nivel (Level)**: Variables cuyo valor se calcula a partir del valor que tenían en la iteración anterior.
  - **Data**: Variables cuyos valores no dependen de otras variables del modelo. Suelen depender de información externa al modelo. Se usan para comparar el modelo con el sistema real.
- **Lookups**: Funciones numéricas no lineales. Se definen mediante una serie de puntos en los ejes X e Y. Sirven para inferir el valor que tomará la función dado un X. Se pueden definir en un símbolo o usar de forma anónima con la función `WITH LOOKUP`.

- **Subscripts**: Símbolos que pueden tener una serie de valores finitos. Son similares a las enumeraciones en lenguajes de programación como Java. Las constantes y variables pueden estar indexadas por subscripts o valores de subscripts. Esto provoca que el símbolo indexado pase de ser un solo valor a ser una matriz de valores, de tantas dimensiones como índices tenga. Cada combinación de índices puede tener su propia ecuación. Cuando una variable está indexada por un subscript se hace referencia a todos los valores de dicho subscript a la vez.
- **Subscript Values**: Los valores que puede tomar un subscript. Los subscripts pueden ser copias o subcojuntos de otros subscripts, en cuyo caso comparten algunos valores. Pero en estos casos no pueden tener más valores propios o de un tercer subscript.
- **Reality checks**: Ecuaciones que comprueban que una restricción se cumple. Son similares a los asertos en lenguajes de programación comunes. Se usan para asegurar que los símbolos no toman valores irreales, como tiempo negativo o temperatura negativa en Kelvin, o para asegurar condiciones que deben cumplirse a lo largo de toda la simulación, como una ley de conservación de energía, por ejemplo.

1.3. Introducción a Sonarqube

Sonarqube [51] es una plataforma web que permite analizar código en varios lenguajes de programación para detectar bugs y malas prácticas, algunas de ellas conocidas como *code smells*.

La plataforma se divide en dos partes: La web y el escáner, ambas son open source. La web muestra los resultados de los análisis y permite configurar los proyectos (ver Figura 1.3). El escáner es el programa que se encarga de analizar el código. Para ello se comunica con la web para obtener la configuración del proyecto y devolver los resultados. El escáner puede ser el programa oficial (`sonar-scanner` [53]) o puede ser un escáner embebido dentro de un entorno de desarrollo.

En un análisis Sonarqube comprueba el cumplimiento de una serie de reglas. Por ejemplo no tener métodos deprecados, o comparar strings con `equals` en Java en vez de con `==`. Cuando una regla no se cumple se lanza una **issue**. Estas pueden ser de varios tipos: Bugs, code smells o vulnerabilidades. Las issues están asociadas a un fragmento de un fichero, normalmente a una o varias líneas consecutivas. Cada issue también tiene una gravedad asociada, para poder priorizar la resolución de issues. En la Figura 1.4 se muestra una fragmento de una vista en SonarQube donde se ven los detalles de una issue.

Cada regla está asociada a un lenguaje de programación. Los lenguajes pueden tener varios **Quality Profiles**, que son un subconjunto de las reglas que tiene un lenguaje. Cada proyecto está asociado a un Quality Profile, lo que permite configurar las reglas que se quieren aplicar para un proyecto concreto.

En Sonarqube las reglas están programadas de forma modular, mediante plugins. Cada plugin contiene una serie de reglas para un lenguaje de programación. Ejemplos de estos plugins serían SonarPython [52] y SonarJS [50].

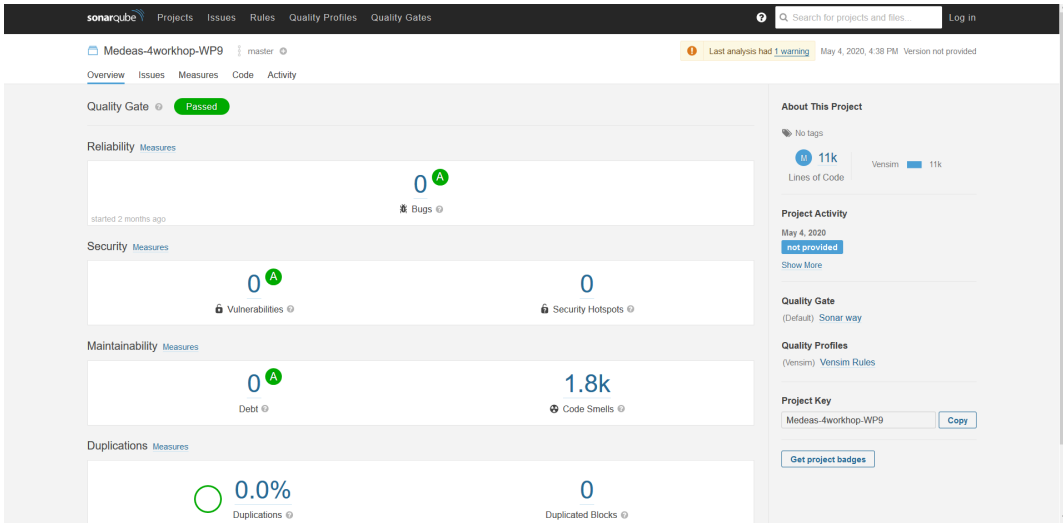

Figura 1.3: Ejemplo de análisis en Sonarqube.

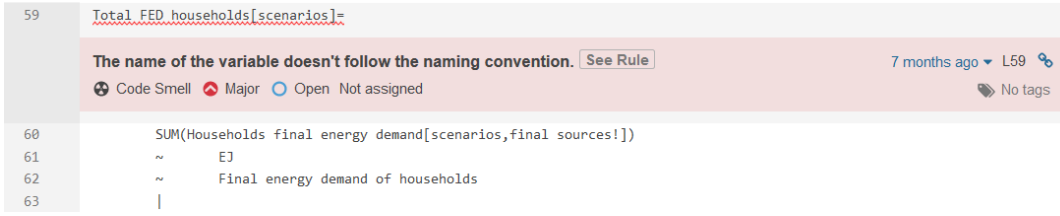

Figura 1.4: Ejemplo de una issue en Sonarqube.

1.4. Objetivos

El objetivo de este trabajo es crear un plugin de Sonarqube para analizar la calidad de modelos Vensim en su formato textual (archivos `.mdl`). Se consideran los siguientes subobjetivos:

- Definir unos estándares de calidad para los modelos Vensim que se puedan comprobar de manera automática.
  - El estándar se creará en colaboración con el GEEDS.
- Crear un plugin de Sonarqube capaz de comprobar los estándares acordados.
- El plugin deberá comunicarse con un diccionario de datos, que contendrá metadatos de los símbolos. El plugin debe comprobar que los metadatos del diccionario son consistentes con los del fichero `.mdl` en análisis.

Para este proyecto se usará la versión de 8.0.4 Vensim y la versión 7.9 LTS de Sonarqube. Además, el plugin asumirá que los modelos a analizar son correctos. Vensim internamente

permite realizar una comprobación del modelo, y determinar si es correcto. Esto se hace a petición del usuario o antes de ejecutar una simulación. En el plugin desarrollado, cuando se esté ante un `.mdl` cuya corrección no haya sido comprobada en Vensim, se tratará de generar mensajes de error útiles.

1.5. Estructura de la memoria

Este documento se estructura de la siguiente forma:

Capítulo 2 Requisitos y planificación: Describe el proceso de desarrollo utilizado y su adaptación. También incluye los requisitos, su evolución a lo largo del proyecto y la planificación realizada.

Capítulo 3 Tecnologías utilizadas: Describe las tecnologías utilizadas, tanto para la gestión del proyecto como para el desarrollo.

Capítulo 4 Estándares de Programación a cumplir: Presenta los estándares de calidad acordados.

Capítulo 5 Desarrollo y despliegue de Plugins para SonarQube: Describe las herramientas y código necesario para crear un plugin de Sonarqube que añade un nuevo lenguaje con sus propias reglas.

Capítulo 6 ANTLR como solución al análisis de Vensim: Describe por qué se utilizó ANTLR para el parseo de los ficheros `mdl` y explica cómo se creó la gramática.

Capítulo 7 Seguimiento del proyecto: Describe la evolución del proyecto, en orden cronológico y dividido en sprints.

Capítulo 8 Diseño e implementación de la Solución: Describe detalles de la implementación del plugin. Detalla algoritmos y soluciones a problemas que surgieron durante el desarrollo. También muestra la estructura del código.

Capítulo 9 Testing: Describe los tests que se han realizado y la cobertura obtenida.

Capítulo 10 Conclusiones: Describe los resultados del proyecto. Presenta además qué problemas ha habido, qué decisiones han sido buenas y malas, y líneas de trabajo futuras.

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 o un enlace a la instancia de Sonarqube con el plugin instalado.

Capítulo 2

Requisitos y Planificación

2.1. Scrum y su adapción al proyecto

El proyecto se ha desarrollado siguiendo el marco de desarrollo ágil Scrum [33]. Se escogió usar este marco para poder reaccionar rápidamente a los cambios, ya que al inicio del proyecto los requisitos no estaban completamente concretados. Además, se sabía que a lo largo del desarrollo habría varias reuniones en las que los clientes podrían cambiar o añadir requisitos.

2.1.1. Artefactos

Los artefactos [4] que se crean en Scrum son:

- **Product Backlog**: Es el documento que contiene la lista de requisitos de usuario. Al principio se crea una versión inicial, pero esta evoluciona a lo largo del desarrollo.
- **Sprint Backlog**: Contiene la lista de tareas que el equipo de desarrollo debe realizar en un sprint.
- **Incremento**: Es la parte del producto que se ha desarrollado durante un sprint.

2.1.2. Roles

Los roles [6] que tiene Scrum son:

- **Product owner**: Trata de aumentar el valor del producto. Normalmente actuará como representante de los clientes y se encarga de tomar las decisiones que afectan al producto.

- **Scrum master**: Es la persona que se encarga de gestionar todo el proceso de Scrum y de ayudar al equipo para que no haya impedimentos o bloqueos.
- **Equipo de desarrollo**: Está formado por varios profesionales, que se encargan de desarrollar el producto.

2.1.3. Eventos

Scrum tiene una serie de eventos [32] que ocurren de forma regular. Estos tratan de evitar reuniones demasiado largas y facilitan la comunicación regular entre miembros del equipo.

Los eventos tienen una duración prefijada y se realizan de forma periódica. Son:

- **Sprint/Iteración**: Es un periodo de tiempo, normalmente de entre una y cuatro semanas, en las que se desarrolla un incremento del producto.
- **Sprint Planning**: Es una reunión que se realiza al inicio de un sprint. Trata de responder a qué trabajo se va a realizar en ese sprint y cómo se va a realizar.
- **Daily Scrum**: Es una reunión diaria que dura entre quince y treinta minutos. Sirve para que los miembros del equipo se sincronicen y para analizar el progreso realizado.
- **Sprint Review**: Es una reunión que se realiza al final de cada sprint. En esta se revisa lo que se ha realizado en el sprint, y si es necesario se ajusta el Product Backlog.
- **Sprint Retrospective**: Se realiza después del sprint review y antes que el sprint planning. El objetivo es identificar qué ha ido bien y qué no, e intentar implementar mejoras para el siguiente sprint.

2.2. Adaptación de Scrum al proyecto

Para poder usar Scrum hay que adaptarlo a las características de este proyecto.

En cuanto a los participantes, el alumno formará el equipo de desarrollo, y la tutora actúa como product owner, haciendo de intermediario con el GEEDS, y como Scrum master, guiando y asesorando al alumno.

Se estableció que los sprints tendrían una duración de dos semanas. Como el equipo de desarrollo está formado por una sola persona se decidió realizar una weekly en vez de reuniones diarias. Por tanto, por cada sprint se realizan dos. En la primera weekly se revisa el progreso realizado hasta el momento, y en la segunda se hacen el sprint review, sprint retrospective y sprint planning.

El product backlog original se obtuvo a partir de las actividades establecidas en la convocatoria de la beca y reuniones con la tutora. Las historias se fueron detallando según avanzaba el proyecto a partir de conversaciones y reuniones con el GEEDS, así como con la tutora actuando como product owner.

2.3. Product Backlog inicial

En la Tabla 2.1 se muestra el product backlog inicial. Un product backlog inicial puede contener varias historias compuestas o historias epics. Una historia de usuario epic es una historia demasiado grande que no puede entregarse como está definida en un único sprint, o se considera suficientemente grande como para poder subdividirse en historias de usuario más pequeñas.

Número	Descripción
1	Como usuario quiero tener un plugin de Sonarqube para poder analizar la calidad de mis modelos Vensim.
2	Como usuario quiero que el plugin de Sonarqube sea capaz de analizar la representación textual de los modelos de Vensim almacenada en archivos `.mdl` para verificar que cumplen algunas reglas de calidad.
3	Como usuario quiero poder comprobar si mi modelo Vensim sigue el estándar de nomenclatura `snake_case`.
4	Como usuario quiero poder comprobar si hay "números mágicos" en mi modelo para poder definir en su lugar constantes y eliminar el uso del número mágico.
5	Como usuario quiero poder comprobar que los nombres de las variables y constantes en mis modelos Vensim forman parte de un diccionario común, para que en la integración de diferentes módulos se garantice que se utilizan los mismos nombres o que estos se encuentran documentados en dicho diccionario.
6	Como usuario habitual de Vensim en plataformas como Windows y OsX quiero poder analizar la calidad de mis modelos Vensim sin usar la línea de comandos, para que sea más fácil de usar y no requiera conocimientos avanzados de desarrollo de software.

Tabla 2.1: Product backlog inicial

Se determinó que las historias 3, 5 y 6 son epics y se dividirían antes de empezarlas, cuando hubiera menos incertidumbre y se acuerden los requisitos concretos con el GEEDS.

Finalmente la historia número 3 se dividió en como se muestra en la Tabla 2.2.

Al principio la historia de usuario 3.3 incluía los estándares de todos los tipos, pero se dividió en dos historias (3.3.1 y 3.3.2), dado que la primera regla de nombrado que se realizara requeriría más esfuerzo (modificar las estructuras de datos para generar issues, realizar los tests de issues por primera vez, etc). Para poder encajar parte del 3.3 con la tareas ya planificadas del sprint 3 se decidió dividirlo.

La historia 5 se acabó dividiendo como se muestra en la Tabla 2.3. Se decidió sistemáticamente descomponer cada regla en dos historias de usuario, una para conseguir que se comprueba el cumplimiento de la regla, y otra para conseguir generar la issue con toda su información en Sonarqube. Esto permite hacer una mejor descomposición de las tareas y un mejor seguimiento del proyecto.

Número	Descripción
3.1	Como programador quiero construir una tabla de símbolos y parsear un archivo `.mdl` poblando dicha tabla de símbolos y anotando los tipos de símbolos.
3.2	Como programador quiero construir un algoritmo que resuelva los tipos de los símbolos que no se pueden determinar en una sola pasada del parser.
3.3.1	Como usuario quiero poder comprobar si el modelo Vensim representado en el `mdl` cumple el estándar de nomenclatura que se detalla Subscripts
3.3.2	Como usuario quiero poder comprobar si el modelo Vensim representado en el `mdl` cumple los estándares de nomenclatura para constantes, variables, lookups y reality checks

Tabla 2.2: División de la historia 3.

A lo largo del proyecto se modificaron y añadieron historias dentro de la 5. También se añadió la historia de usuario 7, que consistía en inyectar los símbolos en el diccionario. Para poder acabar la mayoría de la carga de trabajo antes de Marzo y para lidiar con los cambios y nuevos requisitos que surgieron se decidió eliminar la historia de usuario 6.

Tabla 2.3: División de la historia 5.

Número	Descripción
5.1	Como usuario quiero poder consultar un servicio con los nombres de variables recogidos de Vensim y saber si están definidas en el diccionario común del proyecto.
5.2	Como usuario quiero que se cree una issue en SonarQube cuando una variable presente en el archivo Vensim no está definida en el diccionario común.
5.3	Como usuario quiero poder consultar si un símbolo en el modelo Vensim es del mismo tipo que el definido en el diccionario.
5.4	Como usuario quiero que se cree una issue en SonarQube cuando una variable en el modelo Vensim no sea del mismo tipo que su definición en el diccionario.
5.5.1	Como usuario quiero poder comprobar que un subscript está asociado a los mismos o a un subconjunto de los subscript values que los que se indica en el diccionario de datos.
5.5.2	Como usuario quiero que se cree una issue en Sonarqube cuando un subscript en el modelo Vensim no tiene los mismos o a un subconjunto de los subscript values que se indica en el diccionario de datos.
5.5.3	Como usuario quiero poder comprobar que una variable o constante indexada usa los índices a los que está asociada en el diccionario de datos, bien sea el nombre del index, o uno de los index values asociados al index.
5.5.4	Como usuario quiero que se cree una issue en Sonarqube cuando una variable o constante indexada usa índices que no están asociados en el diccionario de datos, bien sea el nombre de un subscript o de uno de los valores asociados al subscript.

Continúa en el siguiente página

Viene de la página anterior

Número	Descripción
5.6	Como usuario quiero poder comprobar que un símbolo (constante, variable o lookup table) tiene asociado comentario y unidades.
5.7	Como usuario quiero que se cree una issue en SonarQube cuando un símbolo no tiene comentario o unidades
5.8.1	Como usuario quiero poder comprobar que los comentarios y unidades de un símbolo coinciden con los del diccionario de datos.
5.8.2	Como usuario quiero que se cree una issue en Sonarqube cuando las unidades o comentarios de un símbolo no coinciden con los del diccionario de datos.
5.9	Como usuario quiero poder "parametrizar" el servicio que permite consultar el diccionario de datos, ya sea para poder adaptarse a diferentes proyectos, cada uno con su diccionario de datos desplegado en una determinada URL, o ya sea para no realizar ninguna comprobación relativa a diccionario de datos si un proyecto desea adoptar las reglas de programación pero no usará o no tiene desplegado aún el diccionario de datos.

Fin de la Tabla División de la historia 5.

2.4. Product Backlog Final

Finalmente, después de todo el proceso, las subdivisiones de historias, la eliminación, modificación e inclusión de nuevas historias, en la Tabla 2.4 se muestra el Product Backlog final de este proyecto.

Tabla 2.4: Product backlog final

Número	Descripción
1	Como usuario quiero tener un plugin de Sonarqube para poder analizar la calidad de mis modelos Vensim.
2	Como usuario quiero que el plugin de Sonarqube sea capaz de analizar la representación textual de los modelos de Vensim almacenada en archivos `.mdl` para verificar que cumplen algunas reglas de calidad.
3.1	Como programador quiero construir una tabla de símbolos y parsear un archivo `.mdl` poblando dicha tabla de símbolos, anotando los tipos de símbolos.
3.2	Como programador quiero construir un algoritmo que resuelva los tipos de los símbolos que no se pueden determinar en una sola pasada del parser.
3.3.1	Como usuario quiero poder comprobar si el modelo Vensim representado en el `mdl` cumple el estándar de nomenclatura que se detalla Subscripts
3.3.2	Como usuario quiero poder comprobar si el modelo Vensim representado en el `mdl` cumple los estándares de nomenclatura para constantes, variables, lookups y reality checks
4	Como usuario quiero poder comprobar si hay "números mágicos" en mi modelo para poder definir en su lugar constantes y eliminar el uso del número mágico.

Continúa en el siguiente página

Viene de la página anterior

Número	Descripción
5.1	Como usuario quiero poder consultar un servicio con los nombres de variables recogidos de Vensim y saber si están definidas en el diccionario común del proyecto.
5.2	Como usuario quiero que se cree una issue en SonarQube cuando una variable presente en el archivo Vensim no está definida en el diccionario común.
5.3	Como usuario quiero poder consultar si un símbolo en el modelo Vensim es del mismo tipo que el definido en el diccionario.
5.4	Como usuario quiero que se cree una issue en SonarQube cuando una variable en el modelo Vensim no sea del mismo tipo que su definición en el diccionario.
5.5.1	Como usuario quiero poder comprobar que un subscript está asociado a los mismos o a un subconjunto de los subscript values que los que se indica en el diccionario de datos.
5.5.2	Como usuario quiero que se cree una issue en Sonarqube cuando un subscript en el modelo Vensim no tiene los mismos o a un subconjunto de los subscript values que se indica en el diccionario de datos.
5.5.3	Como usuario quiero poder comprobar que una variable o constante indexada usa los índices a los que está asociada en el diccionario de datos, bien sea el nombre del index, o uno de los index values asociados al index.
5.5.4	Como usuario quiero que se cree una issue en Sonarqube cuando una variable o constante indexada usa índices que no están asociados en el diccionario de datos, bien sea el nombre de un subscript o de uno de los valores asociados al subscript.
5.6	Como usuario quiero poder comprobar que un símbolo (constante, variable o lookup table) tiene asociado comentario y unidades.
5.7	Como usuario quiero que se cree una issue en SonarQube cuando un símbolo no tiene comentario o unidades
5.8.1	Como usuario quiero poder comprobar que los comentarios y unidades de un símbolo coinciden con los del diccionario de datos.
5.8.2	Como usuario quiero que se cree una issue en Sonarqube cuando las unidades o comentarios de un símbolo no coinciden con los del diccionario de datos.
5.9	Como usuario quiero poder "parametrizar" el servicio que permite consultar el diccionario para poder adaptarse a diferentes proyectos, cada uno con su diccionario de datos. O para no realizar ninguna comprobación relativa a diccionario de datos.
7	Como usuario quiero poder inyectar los símbolos definidos en un programa Vensim en el diccionario de datos pendiente de validación.

Fin de la Tabla Product backlog final

2.5. Planificación

La duración de la beca es de 6 meses a partir de junio, prorrogable hasta 12. Como la beca no incluye escribir la memoria se decidió que la parte de programación era prioritaria,

pero que se tomarían notas de forma continua para poder redactar la memoria al final. El TFG está valorado en 12 créditos ECTS [62], cada crédito equivale a 25 horas de trabajo [61]. Por tanto, el proyecto deberá ajustarse a 300 horas de trabajo.

Se planificó un desarrollo de 7 sprints, 6 más un sprint como medida de contingencia ante retrasos. De forma adicional habría 3 sprints para documentación. Cada sprint tiene una duración de 2 semanas. La fecha de inicio oficial fue el 23 de septiembre y el final planificado sería el 22 de marzo.

Aunque los sprints empezaron en septiembre, durante el verano se trabajó en un 'Sprint 0'. El objetivo de este periodo fue estudiar cómo se podía crear un plugin básico de Sonarqube para generar issues, y familiarizarse con Vensim, tanto en su formato gráfico como con su sintaxis.

Además, la mayor parte del TFG se realizaría durante el primer cuatrimestre, en paralelo con otras 5 asignaturas. Por ello se estimó que habría periodos en los que fuera necesario reducir la carga de trabajo para poder afrontar los estudios. Se estableció que habría una semana libre para estudiar y avanzar otros trabajos. Tras consultar la planificación de las asignaturas se decidió que sería la semana del 18 de noviembre. También se estableció que no se trabajaría en el proyecto durante navidades para poder preparar los exámenes finales.

Por otra parte, se amplió la duración del sprint 5 a tres semanas, ya que contiene un puente de 5 días (del 5 al 9 de diciembre). Se puede ver el diagrama de Gantt de la planificación inicial en la figura 2.1.

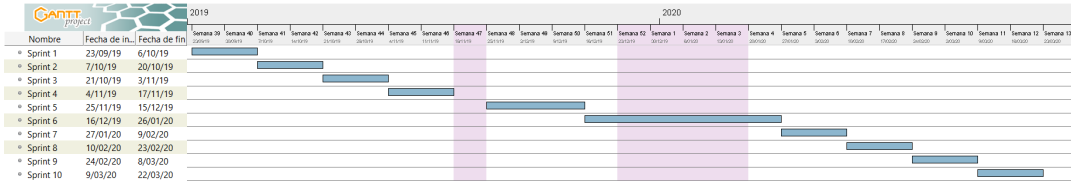

Figura 2.1: Diagrama de Gantt de la planificación original

2.6. Riesgos

Como parte del Sprint 0 se realizó un análisis de riesgos. Debido a la naturaleza del proyecto existen una serie de riesgos que no son tan comunes en los TFGs, como los derivados de tener clientes reales.

Las tablas desde la 2.5 hasta la 2.15 muestran los riesgos identificados, su probabilidad, impacto, y medidas que se puede tomar para reducir la probabilidad e impacto.

Riesgo 1	Gramática de Vensim no proporcionada
Tipo	Técnico
Probabilidad	Media
Impacto	Medio
Descripción	Puede que 'Ventana Systems', la empresa desarrolladora de Vensim, no nos proporcione la gramática que se necesita para analizar sintácticamente los ficheros `.mdl`
Contingencia	Si no se dispone de una gramática, se podría partir de una gramática encontrada en Github [11] con licencia MIT. También se podría obtener una gramática mirando el código del proyecto PySD [18], en el que se traduce código Vensim a Python. En el peor caso se podría crear una gramática a mano partiendo de la documentación de Vensim y de los ficheros de ejemplo.

Tabla 2.5: Riesgo de no obtener una gramática para Vensim

Riesgo 2	Actualización de Vensim
Tipo	Técnico
Probabilidad	Muy baja
Impacto	Bajo-Medio
Descripción	Durante el desarrollo puede haber una actualización de Vensim que modifique la gramática o las asunciones realizadas sobre esta.
Mitigación	Reducir el acoplamiento entre el parser y las reglas de Sonar
Contingencia	En el caso de que la gramática actual no fuese compatible con los ficheros generados en la nueva versión se hablaría con los clientes para determinar si se van a actualizar a la nueva versión. En base a cuándo los vayan a actualizar habría que determinar si actualizar la gramática es una prioridad. También existe la posibilidad de que quieran que el plugin sea compatible con ambos formatos, en cuyo caso se estudiaría la mejor forma de realizarlo.

Tabla 2.6: Riesgo de actualización de Vensim

Riesgo 3	Gramática no completa
Tipo	Técnico
Probabilidad	Media-Alta
Impacto	Bajo
Descripción	Es posible que la gramática no esté completamente definida y no permita analizar modelos que usen características poco frecuentes de Vensim.
Mitigación	Obtener una gramática lo más completa posible a partir de la documentación y los ficheros de ejemplo.
Contingencia	Una vez detectado un caso en la gramática, se puede corregir.

Tabla 2.7: Riesgo de una gramática no completa

Riesgo 4	Falta de validación de un requisito
Tipo	Proceso
Probabilidad	Media
Impacto	Medio
Descripción	Debido a que el TFG debe seguir la planificación acordada, es posible que haya que empezar a implementar un requisito sin que se haya podido validar o acordar sus características concretas.
Mitigación	Mantener vías de comunicación con los clientes para preguntar dudas. Implementar los requisitos de forma flexible para facilitar cambios en las reglas.
Contingencia	Una vez obtenido el feedback de los clientes, hacer los cambios pertinentes.

Tabla 2.8: Riesgo de falta de validación de requisitos

Riesgo 5	Flexibilidad del parser
Tipo	Técnico
Probabilidad	Baja
Impacto	Alto
Descripción	Es posible que el parser que se use no sea lo suficientemente flexible y no permita extraer cierta información con facilidad.
Mitigación	Estudiar en profundidad las diferentes tecnologías que se pueden usar para parsear el fichero. Programar de forma que las estructuras de datos, algoritmos y reglas sean independientes del algoritmo de parseo.
Contingencia	Se valoraría si merece la pena hacer el cambio a otra tecnología, el coste que esto tendría y si hay otras alternativas mejores.

Tabla 2.9: Riesgo de falta de flexibilidad del parser

Riesgo 6	Integración entre la api del plugin y el parser
Tipo	Técnico
Probabilidad	Baja
Impacto	Medio
Descripción	Si se decide no usar el parser "oficial"para los plugins de Sonarqube (SSLR) es posible que no se pueda integrar correctamente o que algunas características no se puedan utilizar.
Mitigación	Estudiar previamente cómo funcionan los plugins de Sonarqube y cómo se realizaría la integración.
Contingencia	En caso de que el parser no se pueda integrar, tendrá que ser reemplazado por otro. Si se puede hacer una integración parcial, en la que solo funcionen algunas características del parser, habría que valorar si merece la pena cambiar de parser o no.

Tabla 2.10: Riesgo de integración entre el plugin y el parser

Riesgo 7	Cambios en los requisitos
Tipo	Contractual
Probabilidad	Alta
Impacto	Medio
Descripción	A lo largo del desarrollo puede haber cambios en los requisitos y/o estándares. Hay muchas organizaciones que forman parte de este proyecto, por lo que es más probable que haya cambios ya que tienen que coordinarse y llegar a un acuerdo, que puede que cambie con el tiempo.
Mitigación	Usar una metodología ágil, y tratar de aislar las partes del programa que es probable que cambie (convenios de nombres, excepciones en las reglas, etc).
Contingencia	Replanificar y realizar los cambios.

Tabla 2.11: Riesgo de cambios en los requisitos

Riesgo 8	Falta de tiempo
Tipo	Personal
Probabilidad	Alta
Impacto	Medio
Descripción	Gran parte del proyecto se realiza durante el primer cuatrimestre, junto con 5 asignaturas. Puede haber momentos en los que la carga de trabajo sea demasiado grande y no se pueda avanzar en el TFG.
Mitigación	Añadir un sprint más a la planificación.
Contingencia	Priorizar las tareas y replanificar.

Tabla 2.12: Riesgo de falta de tiempo

Riesgo 9	Limitaciones de los plugins de Sonarqube
Tipo	Técnico
Probabilidad	Muy baja
Impacto	Alto
Descripción	Ya que el proyecto depende de las librerías de Sonarqube, es posible que haya limitaciones en los plugins o en la propia plataforma de SonarQube.
Contingencia	Si hay alguna tarea que no se pueda realizar, habría que analizar soluciones alternativas.

Tabla 2.13: Riesgo de limitaciones de los plugins de Sonarqube

Riesgo 10	Problemas con el hardware
Tipo	Técnico
Probabilidad	Bajo
Impacto	Medio
Descripción	Si fallara el disco o hubiera algún problema con el hardware, se podría perder parte del progreso realizado.
Mitigación	Usar git y hacer push de forma frecuente a un repositorio remoto.
Contingencia	Reemplazar el hardware.

Tabla 2.14: Riesgo de problemas con el hardware

Riesgo 11	Mala planificación
Tipo	Planificación
Probabilidad	Media
Impacto	Medio
Descripción	Puede haber retrasos si se realiza una mala planificación o se estima de manera incorrecta.
Mitigación	Se ha añadido un sprint más a la planificación.
Contingencia	Ampliar el plazo de entrega o aumentar el trabajo diario.

Tabla 2.15: Riesgo de mala planificación

2.7. Presupuesto simulado

Consultando el Boletín Oficial del Estado [25], se observa en las tablas salariales que el sueldo para un Programador Junior (categoría E I) a partir del 31/12/2019 es de 15.860,56€ brutos al año, y su jornada laboral es de 1.800 horas/año. Las empresas pagan a la Seguridad Social aproximadamente un 31 % [30] del salario base del trabajador. Por tanto, el coste total

para la empresa es de 20.777,33€. Por tanto el coste por hora es de 11,54€. Se estimó que el proyecto serían 300 horas por lo que en total serían 3.462,9€.

En cuanto a software, se usará siempre la versión gratuita de los productos, ya que este proyecto no necesita usar ninguna característica de pago.

No se van a tener en cuenta costes generales, como luz, agua y alquiler ya que el proyecto se desarrolla desde la universidad y desde el alojamiento del alumno.

Para el desarrollo se usará un portátil HP Pavilon 15-ak003ns, valorado en 1.199€. Los ordenadores se consideran equipos para procesos de información, por lo que tienen un coeficiente lineal máximo de un 25 %[1]. La duración del proyecto es de 6 meses, por lo que el coste sería de 1.119€*0.25 anual* 6 / 12. Esto son: 149,87€.

Para afrontar costes inesperados o retrasos se aplica un colchón del 25 %. La Tabla 2.16 muestra el presupuesto calculado.

Sueldos	3.462,9€
Material	149,87€
Suma	3.612,77€
Colchón 25 %	903,19€
Total	**4.515,96€**

Tabla 2.16: Presupuesto simulado inicial del proyecto

2.8. Presupuesto real

El presupuesto real está compuesto únicamente por la remuneración de la beca, ya que ya se disponía del portátil. La duración de ésta es de seis meses, con un sueldo de 300€brutos al mes. Por tanto el presupuesto real es de 1800€.

Cabe resaltar que los seis meses de duración de la beca no concuerdan con los seis meses de desarrollo. La beca se inició en julio y acaba en diciembre, y la planificación del desarrollo es de finales de septiembre a finales marzo. De julio a septiembre se realizó el Sprint 0, y de enero a marzo se completarán los últimos sprints de desarrollo y documentación, que ya no forman parte de la beca.

Capítulo 3

Tecnologías utilizadas

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

3.1. Tecnologías para la gestión del proyecto

3.1.1. Trello

Trello [2] es una herramienta de gestión de proyectos en el que cada proyecto se representa mediante un tablero, que muestra el estado de las tareas. Un tablero está compuesto por una o varias listas, y cada lista puede tener tarjetas, que representan a las tareas.

En este proyecto se uso Trello como herramienta para planificar las tareas dentro de un sprint, sobretodo las subtareas más pequeñas dentro de cada historia o tarea. Se siguió una estructura similar al método Kanban, en el que las tareas se dividen en tres listas: Por hacer, haciendo y completadas.

3.1.2. Rocket.Chat

Rocket.Chat [31] es una plataforma de comunicación similar a Slack. En este proyecto se utilizó para comunicar información sin tener que esperar a la siguiente weekly.

3.1.3. Skype

Skype [24] es una aplicación de telecomunicación que permite hacer videollamadas. Se utilizó como herramienta para realizar las weeklies debido al confinamiento (ver sección 7.13).

3.1.4. Gitlab issue tracker

Gitlab [14] es una plataforma que incluye múltiples herramientas para el desarrollo y planificación de proyectos.

En cuanto a planificación se utilizó el gestor de issues, que permiten crear tareas, estimar su duración y guardar el tiempo real que se ha dedicado a la tarea.

3.2. Tecnologías para el desarrollo

3.2.1. IntelliJ

IntelliJ [19] es un entorno de desarrollo para Java con features avanzadas como auto-completado, linting, refactoring, debugging, integración con git. Se usó para desarrollar el plugin.

3.2.2. Sonarqube y VM

Para poder ejecutar el plugin y hacer tests de integración se instalaron dos instancias de Sonarqube [51]: Una en local, para testing, y otra en una máquina virtual, para hacer demostraciones y hacer tests en una instancia en remoto.

Ambas están desplegadas en linux y usan una base de datos PostgreSQL [60], con el objetivo de que el entorno de desarrollo se parezca lo más posible al entorno de despliegue.

3.2.3. Git y Gitlab

Git [13] es un sistema de control de versiones con el que se pueden almacenar y ver los cambios realizados a lo largo de un proyecto. El repositorio git se almacenó en el Gitlab de la escuela https://gitlab.inf.uva.es como repositorio remoto. Gitlab [14] es una plataforma para, entre otras cosas, gestionar repositorios Git y expande su funcionalidad a diferentes ámbitos del desarrollo.

Como flujo de trabajo se usó el modelo de ramas Gitflow [7], por el cual hay dos ramas principales: master y develop. Cada feature se programa en una rama que parte de develop, y tras completarla se mergea de nuevo con develop. Cuando en develop se tiene un incremento funcional, se mergea en master, garantizando que en master siempre hay un una versión funcional, lista para una demo.

Continuous Integration y Continuous Deployment

Otra de las características de Gitlab que se usó en este proyecto fue la integración continua (CI). Esta práctica consiste en automatizar una serie de tareas habituales a lo largo del desarrollo, desde la compilación, generación de código, la ejecución de los tests cada vez que se hace un push al repositorio en gitab, trasladando los nuevos commmits que se hayan realizado desde el push anterior.

A su vez, cuando se hace un commit en la rama master además de los tests se despliega el plugin automáticamente en instancia de Sonarqube de la máquina virtual. Esto se denomina Continuous Deployment (Despliegue Continuo).

Ambas se realizaron mediante el mecanismo de CI/CD de Gitlab, definiendo las acciones en un fichero `.gitlab-ci.yml` en la carpeta raíz del proyecto. La integración y despliegue continuo en gitlab se basa en la definición de una tubería (pipeline) de etapas, dentro de cada etapa se definen trabajos. Los trabajos dentro de cada etapa pueden ser concurrentes, para iniciar una etapa deben haber finalizado todos los trabajos de la etapa anterior.

Ejecutar los tests de forma automatizada es sencillo. Solo hay que definir un trabajo que se encargue de ejecutar los tests, y hacer que forme parte del pipeline.Esto se muestra en el Fragmento de código 3.1. Para ejecutar solo los tests que no requieren de una instancia de Sonarqube se usa el perfil **Unit**.

```
stages:
  - build
  - test
  - deploy

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

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

Fragmento de código 3.1: Parte del fichero `.gitlab-ci.yml` correspondiente al CI

El despliegue es más complicado, ya que requiere acceder a la máquina virtual por ssh y ejecutar los comandos necesarios para actualizar, compilar y desplegar el plugin dentro de la máquina.

Para poder acceder por ssh se usará una autenticación por claves ssh. El primer paso es generar las claves con `ssh-keygen` [15] como se ve en el fragmento de código 3.2.

```
ssh-keygen -o -t rsa -b 4096
```

Fragmento de código 3.2: Parte del fichero `.gitlab-ci.yml` correspondiente al CI

Para evitar problemas no se usó contraseña, y se dejó la ruta por defecto. Esto generará dos ficheros dentro de la carpeta ∼`/.ssh`: La clave pública (`id_rsa.pub`) y la clave privada (`id_rsa`).

Una vez creado hay que almacenar la clave privada en una variable en Gitlab. Esto se realiza dentro de la configuración del CI/CD del repositorio. La variable se llama `SSH_PRIVATE_KEY`.

Después hay copiar la clave pública al fichero ∼`/.ssh/authorized_keys` **de la máquina a la que se quiere conectar**, en este caso a la máquina virtual que contiene la instancia de Sonarqube.

Finalmente, hay crear una tarea previa que se encargue de configurar el ssh a partir de la variable creada. Esta tarea se llama `before_script` para que se ejecute la primera [16].

```
before_script:

  ## Install ssh-agent if not already installed, it is required by
      Docker.
  - 'which ssh-agent || ( apt-get update -y && apt-get install
      openssh-client -y )'


  ## Run ssh-agent (inside the build environment)
  - eval $(ssh-agent -s)

  ## Add the SSH key stored in SSH_PRIVATE_KEY variable to the agent
       store
  - echo "$SSH_PRIVATE_KEY" | tr -d '\r' | ssh-add -

  ## Create the SSH directory and give it the right permissions
  - echo "$SSH_PRIVATE_KEY" > key.prv
  - chmod 600 key.prv
  - mkdir -p ~/.ssh
  - chmod 700 ~/.ssh
```

Fragmento de código 3.3: Tarea del `.gitlab-ci.yml` que se encarga de configurar el ssh

Con esto ya se puede crear una tarea que se conecte a la máquina por ssh y ejecute

cualquier cosa. El funcionamiento es sencillo: Entrar en la máquina, ir a la carpeta en la que está el repositorio (que se debe de descargar a mano la primera vez), moverse a la rama máster (por si por alguna razón hubiera cambiado), hacer git pull, compilar el plugin, desplegarlo y reiniciar SonarQube. La tarea se puede ver en el fragmento de código 3.4.

```
deploy:
  stage: deploy
  script:  ssh -i key.prv  -oStrictHostKeyChecking=no
      usuario@tfg2020.virtual.lab.inf.uva.es -p 20201
  "cd tfg-sonarvensim &&
  git checkout master &&
  git pull origin master &&
  mvn package -DskipTests &&
  cp target/sonar-vensim-1.0-SNAPSHOT.jar   ../sonarqube-8.0/
      extensions/plugins/sonar-vensim-1.0-SNAPSHOT.jar  &&
  sh ../sonarqube-8.0/bin/linux-x86-64/sonar.sh restart &&
  exit"
  only:
    - master
```

Fragmento de código 3.4: Tarea de `.gitlab-ci.yml` que se encarga de hacer el despliegue automático.

Al añadir el despliegue automático hubo problemas por los que Gitlab no se podía conectar a la máquina virtual por ssh. Tras hablar con los técnicos de la escuela se llegó a la conclusión que era probable que el firewall de la red de la escuela estuviera bloqueando la conexión. Una vez lo solucionaron el despliegue funcionó correctamente.

3.2.4. Maven

Maven [58] es una herramienta para la gestión de dependencias y para la construcción de proyectos Java. La configuración del proyecto se almacena en un fichero `pom.xml` que se encuentra en la carpeta raiz del proyecto.

3.2.5. ANTLR4

ANTLR4 [27] es una herramienta que permite definir una gramática y a partir de esta genera un parser y las estructuras de datos necesarias para recorrerlo.

Como se explica en la Sección 6.1, se consideró que esta era la herramienta más adecuada para el parseo de los ficheros generados por Vensim.

3.2.6. Dyson

Dyson [20] es una herramienta que permite crear rápidamente un servidor de node para hacer mocks de peticiones al backend. Se usó para hacer los tests de integración con Sonarqube usando un mock en vez del servicio real del diccionario de datos.

3.2.7. JUnit y Mockito

JUnit [59] es un framework de testing para Java. Se utilizó para realizar los tests del plugin junto con Mockito [10], un framework para simular objetos, con el objetivo de realizar pruebas en aislamiento.

Capítulo 4

Estándares de Programación a cumplir

Para evaluar la calidad de los modelos Vensim se acordaron una serie de estándares a cumplir. Estos cubren varios aspectos del modelo. Se debe seguir un estándar de nombrado consistente a lo largo de todos los ficheros, se debe de evitar usar números mágicos y los símbolos del fichero tienen que tener los mismos metadatos que los que establece el diccionario de datos compartido por todos los modeladores.

4.1. Reglas de nombrado

El estándar de nombrado sigue una serie de reglas que cambian dependiendo del tipo del símbolo. El objetivo es que se siga un nombrado similar al estándar de Python, para facilitar una conversión automática de Vensim a Python.

Los símbolos pueden estar compuestos por una o varias palabras, separadas por una sola barra baja (_). El nombre solo puede contener caracteres alfanuméricos. Dependiendo del tipo las reglas son:

- Las variables deben de estar en minúsculas.
- Las constantes deben de estar en mayúsculas.
- Las constantes y variables pueden definirse en una equation o en una data equation. En ambos casos siguen el mismo estándar.
- Los lookups son funciones por lo que tienen que estar en minúscula. Además para resaltar que son lookups deben tener el sufijo `_lt`. Por ejemplo: `historic_demand_lt`.

- Los subscripts y sus valores se consideran constantes, por lo que sus nombres deben estar en mayúsculas. Para diferenciar los subscripts de los valores y las constantes, deben de tener el sufijo `_I`. Por ejemplo `ESCENARIOS_I: BAU, GREEN_GROWTH`.
- Los reality checks son como funciones, aunque actuen como tests. Por ello se escriben en minúscula y se usa el sufijo `_check`.
- Las macros también son funciones, por tanto deben de estar en minúscula, y no tienen un sufijo propio.

Se decidió poner sufijos en vez de prefijos a los símbolos ya que Vensim tiene una lista de palabras en el que se ordenan alfabéticamente, y un buscador para localizar símbolos que busca por el inicio del nombre. Poner un prefijo habría hecho más difícil la búsqueda de símbolos a los desarrolladores de modelos.

Cabe resaltar que el estándar de nombrado de los modelos es más completo que el aquí indicado. Existen una serie de reglas semánticas que el plugin no puede detectar. Por ejemplo que las variables deberían de tener solo palabras en inglés. Las comprobaciones del plugin son solamente sintácticas, y por ello solo se han indicado las reglas que se comprueban. El documento con el estándar completo es el Deliverable 9.1 del proyecto LOCOMOTION, que está pendiente de publicación [5].

4.1.1. Funciones no puras

Para poder diferenciar entre constantes y variables se definió una lista de funciones no puras. Todo símbolo que dependa de una de estas funciones se considera variable.

La lista de funciones no puras es: `INTEG`, `STEP`, `DELAY`, `DELAY1I`, `DELAY3`, `DELAY3I`, `FORECAST`, `SMOOTH3`, `SMOOTH3I`, `SMOOTHI`, `SMOOTH`, `TREND`, `RAMP`, `RANDOM 0 1`, `RANDOM BETA`, `RANDOM BINOMIAL`, `RANDOM EXPONENTIAL`, `RANDOM GAMMA`, `RANDOM LOOKUP`, `RANDOM NEGATIVE BINOMIAL`, `RANDOM NORMAL`, `RANDOM PINK NOISE`, `RANDOM POISSON`, `RANDOM TRIANGULAR`, `RANDOM UNIFORM` y `RANDOM WEIBULL`.

4.1.2. Unidades y comentarios

Aunque no forman parte estrictamente del estándar de nombrado, se decidió que todos los símbolos deberían de tener unidades y descripción (comentario). Las únicas excepciones son aquellos símbolos en los que no se pueden poner en Vensim : Funciones, macros y valores de subscript.

4.2. Números mágicos

Los números mágicos son números que aparecen en medio del código y no queda claro de dónde provienen. No hay un estándar del número de veces que tiene que aparecer un número

para que se considere mágico, ni de cuándo no se considera mágico.

Por ello se acordó que el número de apariciones de un número sería configurable a partir de un parámetro de regla (Sección 5.3.1). Por defecto tiene que aparecer 3 veces para que sea un número mágico. Además se añadieron una serie de excepciones en las que los números no contarían. Las siguientes excepciones hacen referencia a las situaciones en las que una aparición no se tiene en cuenta a la hora de determinar si el número es o no es mágico.

4.2.1. Excepciones obligatorias

Hay sitios en los que no se puede evitar usar literales numéricos, ya que su uso es obligatorio por la sintaxis de Vensim.

Asignación de constantes

Para poder reemplazar los números mágicos por constantes debe de estar permitido definir constantes. Las asignaciones de un símbolo a un literal numérico no cuentan como número mágico, por ejemplo `INITIAL_YEAR = 1970`. En el caso de ser una expresión con varios literales numéricos, sí que contaría como números mágicos. Por ejemplo en `A = 3*4` ambos números serían candidatos a número mágico.

Definición de arrays

Al definir un tabbed arrays, arrays bidimensionales y unidimensionales solo se pueden usar literales numéricos. Se puede ver un ejemplo de definición de estos tipos en los Fragmentos de código 4.1, 4.2 y 4.3.

```
tabbed_array = TABBED ARRAY(1    2    3    4
                            5    6    7    8
                            9    10   11   12)~~|
```

Fragmento de código 4.1: Definición de un tabbed array

Reality checks

Aunque en los reality checks sí se pueden usar símbolos, es preferible realizar las comprobacciones con literales. Usar literales es más legible y, además, añadir constantes al modelo únicamente para testear solo añadiría ruido.

```
initial_population[country,blood_type] = 1,2,3,4;
                                         5,6,7,8;
                                         9,10,11,12; ~~|
```

Fragmento de código 4.2: Definición de un array bidimensional

```
variable = 3,1,1,2,3,9,10,11,12; ~~|
```

Fragmento de código 4.3: Definición de un array unidimensional

Lookups

Al definir un lookup no se pueden usar símbolos, ni al definirlo con nombre, ni de forma anónima. Aunque al usar la función WITH LOOKUP, el primer argumento sí podría ser un número mágico. Se pueden ver ejemplos en los Fragmentos de código 4.4 y 4.5.

```
variable( [(3,0)-(10,10)],(1.10092,2.41228),(2.72171,3.24561)
    ,(6.11621,5.96491),(3,8.37719))~~|
```

Fragmento de código 4.4: Definición de un lookup

```
shipping_time= WITH LOOKUP (
        TIME, ([(0,0)-(100,20)],(0,10),(20,10),(25,20),(60,20)
            ,(70,10),(100,10) ))~~|
```

Fragmento de código 4.5: Llamada a la función `WITH LOOKUP` con un lookup anónimo.

4.2.2. Números compuestos

Hay una serie de números que aunque no sean literales actúan de una forma similar. Un ejemplo sería raíz de dos. Según las reglas descritas en la sección anterior, en `"a = SQRT(2)"` el 2 se consideraría un candidato a número mágico.

Para evitar que esto sea considerado como número mágico, se ha definido que estos casos forman números compuestos. Estos números siguen las mismas reglas descritas hasta ahora, solo que en vez de considerar que el número es el `2`, se considera que el número es `SQRT(2)`.

No todas las funciones permiten formar números compuestos. Se determinó la lista a partir de aquellas funciones puras en las que tiene sentido considerar el número en su conjunto, en vez del literal numérico.

La lista completa es: `SQRT`, `TAN`, `TANH`, `SIN`, `SINH`, `COS`, `COSH`, `ARCTAN`, `ARCSIN`, `ARCCOS`, `ABS`, `LN`, `GAMMA LN`, `INTEGER`, `GAME`, `EXP`, `POWER`, `LOG`, `MODULO`, `QUANTUM`.

Para que una llamada a estas funciones se considere un número compuesto, todos sus argumentos deben de ser literales numéricos u otros números compuestos. Por ejemplo `SQRT(SQRT(3))` es un número compuesto, pero `POWER(TIME,2)` no.

4.2.3. Números permitidos

Los números 0, 1 y 100 se usan de forma habitual y en muchos casos su uso no tiene un significado relevante, por ejemplo se usan para inicializar valores, calcular porcentajes, etc.. Por ello se acordó que estos números no se considerarían mágicos, independientemente del número de veces que aparezcan.

4.2.4. Switches

El modelo de MEDEAS permite seleccionar entre varios escenarios o estados iniciales. Para seleccionar un estado se usa una constante que se denominó **switch**, y para inicializar las variables en el estado correspondiente se usa una serie de funciones IF THEN ELSE anidadas.

En estas condiciones el switch se compara con una serie de literales numéricos. Se decidió que no se quería que estos literales contaran como números mágicos.

Esta verificación es más semántica que sintáctica, ya que no se puede diferenciar entre una comparación normal y una comparación de un switch. La solución fue acordar que los switches tendrían el prefijo `SWITCH_`. Por lo que todas las comparaciones dentro de una función IF THEN ELSE entre un literal y un símbolo con dicho prefijo no cuentan como números mágicos.

4.2.5. Numéricos mágicos por valor y no por token

Como parte del estándar se estableció que lo que importa no es el token de un número, sino su valor. Es decir casos como 1.0, 1 y $--1$ se consideran el mismo. Por tanto aunque se almacenen como cadenas a la hora de comprobar si son números mágicos se tiene que tener en cuenta su valor.

La única excepción serían los números compuestos, en los que el número se considera toda la cadena que lo forma.

4.2.6. Issues generadas

Al principio se decidió que el plugin solo crearía una issue cuando un candidato a número mágico apareciera el número de veces que indica el parámetro de la regla o más.

Sin embargo tras una reunión el GEEDS consideró más útil generar una issue para cada número candidato. Si el número supera el parámetro de la regla generará una issue con gravedad `MAJOR`, y si no lo supera solo tendrá gravedad `INFO`.

De esta forma pueden filtrar las issues para solo ver los números mágicos, o si quieren pueden ver todos los números candidatos y verificar a mano si quieren quitarlo o no.

4.3. Consistencia con el diccionario de datos

Como es posible que haya varios programadores trabajando en un mismo modelo a la vez, es importante mantener la consistencia entre modelos. Por ello existe una base de datos con información relativa a los símbolos, y el plugin debe de verificar que coincide con lo leído en el fichero `mdl`. Este diccionario de datos consta de una base de datos, que se denomina el diccionario, una aplicación web para facilitar su uso y unos servicios REST para permitir uso programáticamente, por ejemplo desde el plugin de Sonarqube.

En concreto, hay que comprobar que:

- Todos los símbolos que se definen en el fichero están definidos en el diccionario.
- El tipo, comentarios y unidades de los símbolos coinciden con los que indica el diccionario.
- Los subscripts tienen los mismos o un subconjunto de los valores que se establecen en el diccionario.
- Los símbolos indexados tienen los mismos o un subconjunto de los índices que se establece en el diccionario.
- Los símbolos solo pueden aparecer en los módulos que indica el diccionario de datos. En el diccionario de datos se define que un símbolo se crea un módulo (módulo principal) y puede usarse en ese mismo módulo u otros módulos (módulos secundarios).

En el caso de los subscripts y los símbolos indexados solo es necesario que sean un subconjunto en vez de que sean iguales. Esto se debe a que un símbolo puede aparecer en varios módulos, y en uno es posible que no sea necesario usar todos los subscripts o índices.

Capítulo 5

Desarrollo y despliegue de Plugins para SonarQube

Para dar soporte a un nuevo lenguaje en SonarQube hay que crear un plugin que analice el código, o en este caso, los ficheros `mdl` generados por Vensim.

Los plugins se desarrollan con Java y permiten usar Maven o Gradle [40]. En este proyecto se decidió usar Maven ya que se encontró más documentación y ejemplos de plugins que lo utilizaban. Se debe de usar como mínimo la versión 8 de Java, aunque en este proyecto se usó la versión 11, ya que es la versión mínima para ejecutar el servidor de Sonarqube [41].

Para desarrollar el plugin hay que seguir una serie de pasos establecidos en la documentación [57]. Las partes esenciales del plugin son:

1. Interacción con Sonarqube. Se encarga de:
   - Establecer la configuración para que Sonarqube reconozca el programa como un plugin y cree las reglas en el servidor.
   - Llamar a los otros módulos para que analicen el fichero.
   - Enviar a Sonarqube los resultados del análisis.
2. Parser. Se encarga de leer el fichero y generar un árbol.
3. Reglas. Analiza la salida del parser para comprobar si se cumple la condición de la regla.

En la parte de interacción con Sonarqube es necesario seguir la estructura que establece la API, pero el parser y las reglas se pueden implementar de forma libre, aunque al finalizar el análisis habrá que pasar los resultados en el formato establecido por Sonarque.

5.1. Interacción con Sonarqube

5.1.1. Dependencias necesarias

La interacción con Sonarqube se realiza usando la librería `sonar-plugin-api`. Además para generar el `.jar` de forma correcta y que se detecte como un plugin válido hay que añadir a maven el plugin `org.sonarsource.sonar-packaging-maven-plugin`. En el fragmento de código 5.1 se ve lo que hay que añadir al pom.xml.

```
<dependencies>
    <dependency>
      <groupId>org.sonarsource.sonarqube</groupId>
      <artifactId>sonar-plugin-api</artifactId>
      <version>7.9.1</version>
      <scope>provided</scope>
    </dependency>
</dependencies>

<build>
    <plugins>
      <plugin>
        <groupId>org.sonarsource.sonar-packaging-maven-plugin</
            groupId>
        <artifactId>sonar-packaging-maven-plugin</artifactId>
        <version>1.18.0.372</version>
        <extensions>true</extensions>
        <configuration>
          <pluginKey>vensim</pluginKey>
          <pluginClass>es.uva.locomotion.VensimPlugin</pluginClass>
        </configuration>
      </plugin>
    </plugins>
</build>
```

Fragmento de código 5.1: Fragmento del pom.xml

Es importante que la dependencia tenga el ámbito `provided` o no lo detectará de forma correcta. La versión de la dependencia indica la versión mínima de Sonarqube en la que funcionará el plugin.

En la configuración del build plugin hay que indicar la clave del plugin, que tiene que ser única, y la clase que representa al plugin, que se verá a continuación.

5.1.2. Clases necesarias

Para que el plugin funcione hay que añadir una serie de clases que definen la configuración del plugin y sirven de punto de anclaje cuando este ejecuta.

VensimPlugin

`VensimPlugin` es una clase que se encarga de definir los módulos/extensiones que tiene el plugin. La usa el build plugin de maven para configurar el .jar generado.

Un plugin básico para un lenguaje nuevo tendrá 4 extensiones:

- `VensimLanguage`: Define un lenguaje. Es necesario porque Sonarqube requiere que todas las reglas estén asociadas a un lenguaje.
- `VensimRuleRepository`: Define y configura las reglas que añade el plugin.
- `VensimQualityProfile`: Un perfil es el subconjunto de reglas que están activas en un proyecto. Se pueden definir varios pero tiene que haber al menos un perfil por defecto.
- `VensimSquidSensor`: Es el módulo que se ejecuta al lanzar un análisis usando sonar-scanner.

```
public class VensimPlugin implements Plugin {

    @Override
    public void define(Context context) {

        context.addExtensions(VensimLanguage.class,
                VensimQualityProfile.class,
                VensimSquidSensor.class,
                VensimRuleRepository.class
        );

    }

}
```

Fragmento de código 5.2: Clase `VensimPlugin`

VensimLanguage

Esta clase se encarga de definir el nombre y una clave única para el lenguaje. Dado que en los plugins oficiales [43] se pone como clave la extensión del lenguaje, se ha seguido la

misma convención. El fragmento de código 5.3 muestra el código de esta clase.

```
public class VensimLanguage extends AbstractLanguage {

    public static final String KEY = "mdl";
    public static final String NAME = "Vensim";

    private static final String[] DEFAULT_FILE_SUFFIXES = { ".mdl"
        };

    public VensimLanguage(Configuration configuration) {
        super(KEY, NAME);
    }

    @Override
    public String[] getFileSuffixes() {

        return VensimLanguage.DEFAULT_FILE_SUFFIXES;
    }


}
```

Fragmento de código 5.3: Clase `VensimLanguage`

VensimRuleRepository

Esta clase se encarga de definir las reglas. Hay varias formas de hacer esto:

- Cargando los metadatos de las reglas a partir de un fichero XML. Es la opción que suelen usar los plugins oficiales [44].
- Definiendo los metadatos a mano.
- Mediante anotaciones.

Para definir los metadatos a mano se pueden usar métodos. Se puede ver un ejemplo en el Fragmento de código 5.4.

La opción que se ha seguido en este proyecto es usar anotaciones, debido a que no hay tantas reglas como para que sean necesario moverlo a un fichero. Además así los metadatos están incrustados junto a los datos.

Para ello hay que crear un `RuleDefinitionAnnotationLoader` y pasarle un array con todas las clases que representan a una regla. Se puede ver en el Fragmento de código 5.5.

```
repository.createRule(LowerCaseCheck.CHECK_KEY).setName(
    LowerCaseCheck.NAME).setMarkdownDescription(LowerCaseCheck.
    DESCRIPTION);
```

Fragmento de código 5.4: Código necesario para definir una regla mediante métodos.

Además, las reglas deben de tener la anotación `Rule`. La sección 5.2 describe las reglas con más detalle.

```
RulesDefinitionAnnotationLoader loader = new
    RulesDefinitionAnnotationLoader();

List<Class> checks = new ArrayList<>();
getChecks().forEach(checks::add);

loader.load(repository, checks.toArray(new Class[0]));
repository.done();
```

Fragmento de código 5.5: Código necesario para definir reglas mediante anotaciones.

VensimQualityProfile

Esta clase se encarga de definir un perfil de calidad de Sonarqube. Un perfil establece el subconjunto de reglas que están activas en un proyecto en un momento determinado [46]. Todo lenguaje debe de definir al menos un perfil por defecto.

Esta clase tiene un único método que marca las reglas activas. Para activar una regla hay que indicar el repositorio y la clave única. Si la clave no está asociada al repositorio salta una excepción y Sonarqube no arranca.

Para indicar que el perfil esté seleccionado por defecto en el lenguaje hay que llamar al método `setDefault`. Una vez acabada la definición hay que llamar al método `done()` para que se registre el perfil.

VensimSquidSensor

Las clases vistas hasta ahora sirven para configurar la integración con Sonarqube. Sin embargo, la lógica del plugin se encuentra en el sensor. Esta clase tiene una parte de definición,

```
public final class VensimQualityProfile implements
    BuiltInQualityProfilesDefinition {

    @Override
    public void define(Context context) {
        NewBuiltInQualityProfile profile = context.
            createBuiltInQualityProfile("Vensim Rules",
            VensimLanguage.KEY);
        profile.setDefault(true);

        String REPO_KEY = VensimRuleRepository.REPOSITORY_KEY;

        profile.activateRule(REPO_KEY, SubscriptNameCheck.CHECK_KEY);
        ...

        profile.done();
    }
}
```

Fragmento de código 5.6: Fragmento de la clase `VensimQualityProfile`.

que sería el método `describe`, y una parte de ejecución, que es el método `execute`. Se puede ver la estructura básica en el fragmento de de código 5.7

El método `describe` sirve para establecer en qué condiciones se debería de ejecutar el plugin. Sirve para logging y optimizaciones [48]. Entre ellas evitar que el sensor se ejecute si no se cumplen ciertas condiciones. Por ejemplo, se puede usar `onlyOnLanguage` para que no se ejecute si no se encuentran ficheros del lenguaje del plugin.

La clase hereda de `Sensor`, que tiene la anotación `@ScannerSide`. Esta anotación permite que se pueda añadir al constructor cualquier argumento cuyo tipo tenga la misma anotación. Sonarqube se encarga internamente de inyectar la referencia en tiempo de ejecución.

Para este proyecto solo se necesita `CheckFactory`, que permite crear las instancias de las reglas. Todas las reglas tienen que tener una misma interfaz pero, como esta no depende de Sonarqube, es necesario indicarle cuál es la interfaz mediante esta factoría.

Otras clases que se pueden inyectar son `NoSonarFilter`. Esta clase permite registrar las líneas que contienen la palabra `NOSONAR` [38] para que se ignoren las issues en estas. También existe `FileLinesContextFactory` [35], que permite almacenar métricas en las líneas, como el tipo (código, comentario, texto), el autor, etc.

El método `execute` se ejecuta cuando se lanza el escáner y se cumplen las condiciones establecidas en `describe`. La implementación de este método es completamente libre.

```
public class VensimSquidSensor implements Sensor {

    private static final String NAME = "Vensim Squid Sensor";

    private final Checks<VensimCheck> checks;

    public VensimSquidSensor(CheckFactory checkFactory) {
        checks = checkFactory .<VensimCheck>create(
            VensimRuleRepository.REPOSITORY_KEY)
            .addAnnotatedChecks(VensimRuleRepository.getChecks());

    }

    @Override
    public void describe(SensorDescriptor sensorDescriptor) {
        sensorDescriptor.onlyOnLanguage(VensimLanguage.KEY)
                .name(NAME)
                .onlyOnFileType(InputFile.Type.MAIN);

    }

    @Override
    public void execute(SensorContext sensorContext) {
        ...
    }
}
```

Fragmento de código 5.7: Fragmento de la clase `VensimSquidSensor`.

Algunos plugins oficiales extraen toda la información del contexto necesaria del objeto `SensorContext` y delegan la lógica a una clase llamada `VensimScanner` [37, 45], mientras que otros implementan la lógica en la propia clase [36].

Para este proyecto se decidió dividirlo en dos clases, debido a que reduce el acoplamiento y reduce el número de responsabilidades de cada clase. La definición de ambas clases se explica en detalle en la Sección 8.2.

5.2. Parser y reglas

El parser se encarga de transformar el fichero en un árbol. Los plugins de Sonarqube permiten usar cualquier tipo de parser. Los plugins oficiales usan SSLR [55] (SonarSource Language Recognizer). Sin embargo, como se describe en la Sección 6.1, se decidió usar ANTLR4 [56]. En los capítulos 6 y 8 se desarrolla en detalle cómo se realizó el parser.

Todas las reglas tienen que tener la misma interfaz. La interfaz `VensimCheck` declara un método, que toma como argumento un objeto contexto y no devuelve nada. Esto se debe a que las issues se almacenan en el propio contexto.

```
public interface VensimCheck {
     void scan(VensimVisitorContext context);
}
```

Fragmento de código 5.8: Interfaz `VensimCheck`.

El contexto debe de contener toda la información necesaria para todas las reglas. Por ejemplo, el contexto incluye tanto la tabla de símbolos como una referencia al árbol del fichero. Esta referencia es necesaria en caso de que haya reglas que no tengan información suficiente con la tabla de símbolos y necesiten extraer más información por su cuenta. Un ejemplo de este tipo de reglas sería la regla de los números mágicos, que ignora la tabla de símbolos y recorre el árbol creando su propia tabla.

El objeto contexto solo hace referencia a un fichero. Por tanto la regla analizará los ficheros uno a uno. Sonarqube no garantiza un orden de los ficheros, por lo que el resultado de la regla debe de ser independiente del orden de análisis.

La estructura básica de una regla se puede ver en el fragmento de código 5.9.

Como se mencionó en el apartado anterior, las reglas se cargan mediante anotaciones. Por ello todas las reglas tienen que definir su clave, nombre y descripción con la anotación `@Rule`. En caso de que se usara otro método de los descritos para cargar los metadatos, no sería necesario usar `@Rule`.

El método `scan` dependerá de la regla, pero en la mayoría se recorre la tabla de símbolos y se comprueba la validez de cada uno. Si se localiza un símbolo que incumple la regla se añade una issue al contexto indicando la línea, la regla y el mensaje de la issue.

La clase `Issue` no es parte de Sonarqube, por lo que al completar el análisis hay que transformarla al tipo de Sonarqube, como se vió en el apartado anterior. Esta contiene toda la información necesaria para describir una issue: La regla infringida, la línea en la que se detectó, el mensaje, y la gravedad de la issue.

5.3. Parámetros de análisis y parámetros de regla

Sonarqube permite añadir dos tipos de parámetros: de análisis y de regla. Cada uno tiene su función y los plugins pueden definir parámetros de ambos tipos.

```
@Rule(key = SubscriptNameCheck.CHECK_KEY, name = SubscriptNameCheck.
    NAME, description = SubscriptNameCheck.HTML_DESCRIPTION)
public class SubscriptNameCheck implements VensimCheck {
    public static final String CHECK_KEY = "subscript-convention";
    public static final String HTML_DESCRIPTION = ...;
    public static final String NAME = "SubscriptNameCheck";


    @Override
    public void scan(VensimVisitorContext context) {
        SymbolTable table = context.getParsedSymbolTable();

        for(Symbol symbol:table.getSymbols()){

            if(symbol.getType()== SymbolType.Subscript && !
                checkSubscriptNameFollowsConvention(symbol.getToken()
                )){

                for(int line: symbol.getDefinitionLines()) {
                    Issue issue = new Issue(this,line,"The name of
                        the subscript doesn't follow the naming
                        conventions");
                    context.addIssue(issue);
                }
            }

        }
    }

    ...

}
```

Fragmento de código 5.9: Fragmento de la regla SubscriptNameCheck.

5.3.1. Parámetros de regla

Los parámetros de regla sirven para configurar las reglas, por ejemplo para establecer el número de veces que tiene que repetirse un número para ser considerado un número mágico. Esta configuración se almacena en el perfil y es común para todos los proyectos que usen dicho perfil.

Para añadir un parámetro a una regla hay que definir un atributo como parámetro. Para esto se usa la anotación `RuleProperty`, indicando la clave única del parámetro, el valor por defecto y la descripción.

```
@RuleProperty(
    key = "minimum-repetitions",
    defaultValue = DEFAULT_REPETITIONS,
    description = "Minimum times a number must appear to be
        considered a magic number. Must be greater than 0.")
public String repetitions = DEFAULT_REPETITIONS ;
```

Fragmento de código 5.10: Anotación utilizada para definir un parámetro de regla.

El valor por defecto se usa en el perfil por defecto. Para cambiarlo es necesario crear un nuevo perfil. Al activar la regla aparecerá una ventana que permite sobrescribir el valor, como muestra la Figura 5.1.

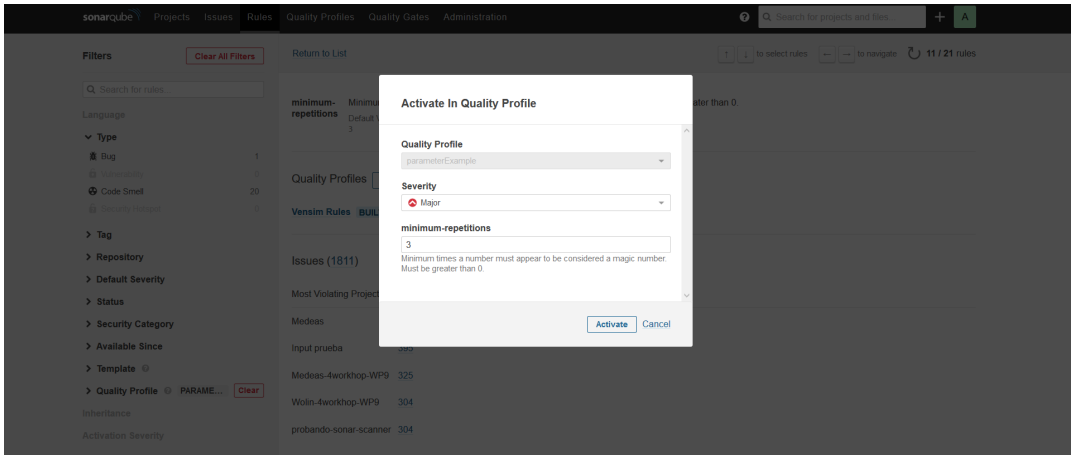

Figura 5.1: Ventana de Sonarqube que se muestra al activar una regla con parámetros.

Cuando se realice un análisis, Sonarqube inyectará dicho valor en la instancia de la regla y se podrá usar el atributo directamente. Los parámetros siempre son cadenas, por lo que el plugin debe de encargarse de validarla y parsearla, si es necesario.

No es recomendable usar este tipo de parámetros para configuraciones globales o que afecten a múltiples reglas, ya que solo se puede acceder a su valor desde la regla que lo definió.

5.3.2. Parámetros de análisis

Los parámetros de análisis [34] sirven para configurar el escaneo. Existe una lista de parámetros que Sonarqube implementa por defecto, entre ellos `sonar.host.url` para indicar

la dirección del servidor de Sonarqube o `sonar.projectKey` para indicar el proyecto que se está analizando.

No es necesario declarar nuevos parámetros. Cualquier parámetro que se ponga se considera válido. El plugin recibe una lista con todos los parámetros que ha introducido el usuario y decide los que usa y los que no.

La desventaja es que no saltará un error si el usuario comete una falta al introducir el parámetro, por ejemplo si se equivoca en una letra. `Sonar-scanner` lo acepta como un parámetro, pero el plugin no lo reconocerá.

Para acceder a la lista, en el método `execute` se llama a `sensorContext.config()`. Esto devuelve un objeto de tipo `Configuration` que tiene una interfaz similar a la de `Map`. Los objetos que devuelve `Configuration` son siempre del tipo `Optional<E>`, aunque se pueden usar los métodos `getInt`, `getFloat`, etc para evitar tener que parsearlos a mano. Se puede ver un ejemplo en el Fragmento de Código 5.11.

```
String DICTIONARY_SERVICE_PARAMETER = "vensim.dictionaryService";
String dictionaryService = sensorContext.config().get(
    DICTIONARY_SERVICE_PARAMETER).orElse("");
```

Fragmento de código 5.11: Ejemplo de acceso a un parámetro de análisis.

Algunos posibles usos para este tipo de parámetros sería añadir tokens de autenticación, o establecer un servidor, como es el diccionario de datos del proyecto.

Es posible configurar reglas usando parámetros de análisis, pero no es recomendable ya que permitiría que los usuarios pongan parámetros diferentes para una misma regla, dando lugar a resultados diferentes dependiendo de quién realice el análisis.

5.4. Despliegue

Para generar el `.jar` del plugin hay que hacer `mvn package -DskipTests`. Los tests se ignoran debido a que los integración fallan si el plugin no está ya desplegado. Esto generará el fichero `sonar-vensim-1.0-SNAPSHOT.jar` dentro de la carpeta `target`.

Para instalar el plugin en Sonarqube hay que copiar el `.jar` en la carpeta `extensions/plugins/` de la instalación de Sonarqube. En caso de que Sonar estuviera en ejecución, habría que reiniciarlo para que se instale el plugin.

Los ejecutables de Sonarqube están en la carpeta `bin`. Por ejemplo los de linux se encuentran en `bin/linux-x86-64/`.Para iniciarlo hay que ir a dicha carpeta y ejecutar `sh sonar.sh start`. También se puede reiniciar con `sh sonar.sh restart`.

Se puede ver una guía más detallada sobre cómo instalar concretamente el plugin desarrollado en este TFG en el anexo A.1.

5.4.1. Debugging del arranque

En caso de que hubiera algún error con el plugin, saltaría un error y Sonarqube no se iniciaría. Para ver a qué se debe hay que estudiar los logs, que se encuentran en la carpeta `logs` de la instancia de Sonarqube.

El fichero `sonar.log` suele contener lo mismo que se muestra al iniciar Sonarqube, así que no es de gran ayuda. Los más útiles son `web.log` y `ce.log`.

Un error que ocurrió al principio del desarrollo fue que Sonarqube no reconocía el plugin porque no se había configurado el pom.xml correctamente. Otro error común fue tratar de activar una regla en `VensimQualityProfile` sin haberla añadido al método `getChecks` de `VensimRuleRepository`, o sin haber puesto la anotación `@Rule`.

Capítulo 6

ANTLR como solución al análisis de Vensim

La primera etapa para poder analizar un fichero es crear un parser capaz de transformarlo en un árbol. Después hay que establecer un mecanismo para recorrer el árbol y extraer la información necesaria.

Para ello hay que decidir qué tecnología se va a usar. Para este proyecto se consideraron dos opciones: SSLR y ANTLR4.

6.1. ANTLR4 vs SSLR

Los plugins oficiales de Sonarqube usan una librería propia de Sonarqube para parsear los ficheros: SSLR [55] (SonarSource Language Recognizer).

Una de las primeros puntos que se analizaron tras completar el plugin de prueba (Sprint 1, ver Sección 7.3) fue cuánto acoplamiento había entre la librería de creación del plugin y SSLR.

Se determinó que las librerías eran independientes y por tanto es posible usar otras librerías, siempre y cuando se puedan usar en Java, ya que es el lenguaje en el que está hecho el plugin.

Debido a que el alumno tenía más conocimiento de ANTLR que de SSLR, se realizó un análisis para determinar cuál se debería usar en el proyecto.

A continuación se resumen las ventajas de utilizar uno u otro.

6.1.1. Ventajas de ANTLR

- Mayor documentación y soporte en foros.
  - Define mejor cómo aborda los problemas clásicos de una gramática, como la prioridad entre reglas, resolución de ambigüedades y recursión por la izquierda.
- Rápida implementación de la gramática formal.
  - No es necesario traducir la gramática a un formato muy diferente, como en SSLR, donde hay que escribir la gramática en Java.
  - Para una gramática compleja como la de Vensim, poder traducirla rápido ahorra mucho tiempo y permite realizar cambios rápidamente.
- ANTLR4 permite que las gramáticas tengan recursión por la izquierda directa. Esto permite una gramática más expresiva y se ahorra el tiempo en eliminar la recursión.
- ANTLR4 implementa el patrón visitor por defecto. Mientras que en cada plugins que usa SSLR lo reimplementan.
- Experiencia previa con ANTLR4.

6.1.2. Ventajas de SSLR

- Más rápido que ANTLR4
  - SSLR es un parser PEG sin recursión por la izquierda [54]. Este tipo de parsers se suele implementar mediante `packrat parsing` y tiene una complejidad lineal [80]. Sin embargo no se ha encontrado cuál es la implementacion concreta de SSLR por lo que no se puede garantizar que sea lineal.
  - ANTLR4 es un parser `LL(*)` que en el peor caso tiene una complejidad `O(n^2)` [29].
- A diferencia de ANTLR4, SSLR no requiere un programa externo para generar el parser.
- Todos los plugins oficiales de Sonarqube usan SSLR.
  - Se pueden usar los plugins de código abierto para obtener ejemplos tanto del parser como de las reglas.

6.1.3. Conclusiones

Se decidió que ANTLR4 es una mejor opción para realizar el plugin. Es una herramienta más potente y que permite acelerar el desarrollo de la gramática.

Además las ventajas de SSLR no son tan relevantes para este proyecto. La eficiencia no es determinante y aunque pudiera utilizar la misma implementación de los visitors de los plugins de código abierto, esta no es tan flexible como los visitors que genera ANTLR4.

6.2. Desarrollo de la gramática

6.2.1. Preparación del entorno

Para poder usar una gramática en el proyecto hay que configurar Maven. Por un lado hay que añadir el plugin `antlr4-maven-plugin` para generan el parser, y por otro hay que añadir las dependencias de ANTLR4 para poder compilar los ficheros Java que se crean al generar el parser.

```
<plugin>
    <groupId>org.antlr</groupId>
    <artifactId>antlr4-maven-plugin</artifactId>
    <version>${antlr4-version}</version>

    <configuration>
      <sourceDirectory>${basedir}/src/main/resources</
          sourceDirectory>
      <outputDirectory>src/main/java/es/uva/locomotion/parser</
          outputDirectory>
      <arguments>
        <argument>-package</argument>
        <argument>es.uva.locomotion.parser</argument>
        <argument>-no-listener</argument>
        <argument>-visitor</argument>
      </arguments>

    </configuration>
    <executions>
      <execution>
        <goals>
          <goal>antlr4</goal>
        </goals>
      </execution>
    </executions>
</plugin>
```

Fragmento de código 6.1: Fragmento del pom.xml necesario para compilar la gramática.

El fragmento de código 6.1 muestra cómo se ha configurado el plugin de compilación. Se asume que hay un único fichero para la gramática. En caso de que haya una gramática principal que importe varias subgramáticas habría que excluirlas todas excepto la principal. Esto se puede hacer poniendo `<excludes>Fichero.g4</excludes>` dentro de la configuración del plugin.

Como se explica más tarde en la sección 6.2.3 se decidió recorrer el árbol mediante un visitor en vez de un listener. Por ello se han añadido los argumentos `-no-listener` y `-visitor`.

La versión del plugin se guarda en una variable tanto porque es una buena práctica, como porque es necesario que comparta versión con las dependencias de ANTLR4 que se puede ver en el fragmento de código 6.2.

```
<dependency>
<groupId>org.antlr</groupId>
<artifactId>antlr4</artifactId>
<version>${antlr4-version}</version>
</dependency>
```

Fragmento de código 6.2: Fragmento del pom.xml para descargar las dependencias necesarias en tiempo de ejecución.

6.2.2. Creación de la gramática

En un primer momento se intentó obtener una gramática oficial de los archivos `.mdl` recurriendo el soporte técnico de Vensim. Sin embargo, afirmaron no tener una gramática. Así que hubo que desarrollar una.

Para ello se tomó como base una gramática que se encontró en Github [11] con licencia MIT, hecha en ANTLR4. La gramática no era completa y fallaba tanto con los ficheros de MEDEAS [22], como con muchos de los ejemplos que se descargan al instalar Vensim.

Con estos ficheros de ejemplo se creó un script que ejecutaba el parser contra todos los ejemplos. Si funcionaba se ignoraba el resultado, pero si fallaba se mostraba dónde había fallado. El script usaba la herramienta `grun/TestRig` [28] de ANTLR4, que también se usó para debuggear la gramática.

Los cambios más notables que se realizaron a la gramática de partida fueron:

- Se juntó la gramática en un solo fichero, ya que el plugin de Maven solo detectaba los cambios de la gramática raíz (`Model.g4`) y no en la dependencia (`Expr.g4`), por lo que cuando ésta cambiaba el plugin no recompilaba la gramática y hacía el desarrollo más lento.
- Se dividió la regla `equation` en `equation`, `dataEquation`, `lookupDefinition` y `unchangeableConstant`, para poder diferenciar en el código entre tipos de definiciones de símbolos diferentes.
- Se añadieron dos sintaxis alternativas para la definición de lookups: Una en la que se ponían todos los números separados por comas, y otra en la que se usa la función `GET XLS LOOKUPS` [68].
- Se añadió la sintaxis de los reality checks y las constraints.

- Se añadió la sintaxis de las macros.
- Se parsean los metadatos de los símbolos en vez de ignorarlos, ya que se necesitan para algunas reglas. Estos incluyen: Unidades, comentario, y si es `supplementary`. Los símbolos se pueden marcar como `supplementary` para indicar no se usan [74].
- Se añadió una regla `OTHER` para que lea los caracteres no alfanuméricos como tokens, en vez de ignorarlos y poner un warning. Si se ignorase se podrían formar árboles incorrectos cuando hubiera caracteres inesperados en el fichero.
- Se dividió la regla original `Id` en `Id` y `subscriptId` para reflejar mejor que solo los nombres de los subscripts pueden acabar en exclamación [76].
- Se añadió soporte para diferentes alfabetos en unicode.
- Se añadió soporte para el operador de asignación de cadenas `:IS:` [73].
- Se añadió soporte para la sintaxis de copia de subscripts (`<->`) [71].
- Se añadió soporte para las funciones `DELAYP` y `TABBED ARRAY`, cuya sintaxis es diferente a la del resto.
- Se añadió soporte para wildcards [63] y los keyword `:NA:` [72] e `:IGNORE:` [70].
- Se obligó a que los ficheros acaben en `EOF` para evitar que al parsear un fichero incorrecto solo se lea la parte válida y se ignore la no válida.
- Se añadió soporte para los subscripts cargados de un archivo usando la funciones `GET DIRECT SUBSCRIPT` [66] y `GET XLS SUBSCRIPT` [69].
- Se añadió soporte para las Unchangeable Constants que se cargan de un fichero usando la funciones `GET DIRECT CONSTANTS` [65] y `GET XLS CONSTANTS` [67].
- Se ignoraron los backslash (\), porque se usan para indicar un salto de línea dentro de los metadatos de un símbolo.

Para ver en detalle los cambios se puede consultar la gramática original [11] y la gramática final [3].

6.2.3. Visitor vs Listener

ANTLR4 dispone de dos mecanismos para recorrer los árboles que genera: Listeners y Visitors [28].

Un listener está compuesto por métodos `enterX` y `exitX`, donde `X` es una regla o subregla etiquetada. Si se usa este patrón, el árbol siempre se recorre siguiendo un algoritmo primero en profundidad. Al entrar al nodo ejecuta el método `enterX`, y cuando acaba de recorrer todos los nodos hijos, llama a `exitX`.

Por otro lado, el visitor permite controlar el recorrido, cambiando el orden o ignorando ramas. En este patrón se genera un método `visitX` por cada regla o subregla etiquetada. Para que el recorrido avance, cada método tiene que llamar explícitamente al método `visitX` del nodo que quiera visitar. También se puede llamar al método `visit`, que es válido para cualquier tipo de nodo, y simplemente redirige la llamada al método adecuado.

No es necesario implementar todos los métodos del visitor. Todos los visitors heredan de `YBaseVisitor`, dónde `Y` es el nombre de la gramática. Esta clase implementa por defecto todos los métodos `visitX` para que recorran todos los hijos. Por tanto si se quieren recorrer todos se puede llamar directamente a `super.visitX(ctx)`.

Otra ventaja de los visitors es que pueden devolver valores, mientras que en un listener sería necesario almacenarlo en una variable.

Se decidió que en este proyecto se usarían visitors, ya que permite mucho más control y flexibilidad que los listeners.

Capítulo 7

Seguimiento del proyecto

7.1. Introducción

Este proyecto comenzó de forma oficial a finales de septiembre del 2019. Sin embargo durante el verano de ese mismo año se dedicó un tiempo a aprender a crear un plugin para Sonarqube. Esta etapa previa al proyecto forma un Sprint 0.

7.2. Sprint 0

Para poder empezar el proyecto con más rapidez se acordó que a lo largo del verano se investigaría cómo funciona la creación de plugins. El objetivo planteado para este periodo fue crear un plugin muy sencillo y comprobar que genera issues.

La mayor dificultad que hubo fue la falta de documentación. Se encontró una página [57] en la que se explican los pasos que hay que realizar a alto nivel pero no cómo realizarlos. Por ello para aprender se estudió el código de los plugins SonarPython [52] y SonarJS [50], publicados en Github. Estos proyectos se analizaron para averiguar cuáles son las mínimas clases necesarias para que un plugin funcione.

Todos los plugins oficiales de Sonarqube, includos los dos analizados en este proyecto, usan el parser SSLR [55](SonarSource Language Recognizer). Sin embargo este no parecía tan flexible como ANTLR4, por lo que también se estudió cuánto acoplamiento había entre el plugin y el parser. Se llegó a la conclusión de que sí era posible separarlos. Aún así la elección final del parser se dejó para cuando se tuviera más información sobre la gramática de Vensim.

En este periodo se logró cumplir el objetivo, y se creó un plugin muy sencillo. Usaba una gramática en SSLR que solo aceptaba un número. Si este era positivo el plugin generaba una

issue y si no, no hacía nada. Para probar el plugin se usó una instancia de Sonarqube que ya se tenia instalada en local.

También hubo problemas para que Sonarqube aceptara el .jar generado como un plugin. El problema era que no se habían declarado las dependencias bien en el pom.xml. Se solucionó gracias a la documentación de Sonarqube [39] y a un proyecto de prueba [49]. Se puede ver en más detalle en la sección 5.1.1.

Este Sprint 0 también se aprovechó para realizar alguna tarea menor, como instalar Vensim y familiarizarse con su interfaz. No se midió el tiempo dedicado a estas tareas. Sin embargo se estima un trabajo de veinticinco horas.

7.3. Sprint 1 (20/9/2019-4/10/2019)

La Tabla 7.1 muestra el desglose de las tareas realizadas en este sprint y las historia de usuario abordadas en el mismo.

Tarea	Tiempo estimado	Tiempo invertido	Estado
1.1 - Configurar el entorno de desarrollo	4h	2h 20m	Completado
1.2 - Obtener un plugin funcional para una gramática que acepta cualquier archivo textual y comprueba que lo que se le pasa no está vacío	5h	3h 26m	Completado
2.1 - Decidir si la gramática se gestionará con ANTLR o SSLR	2.5h	1h 57m	Completado
2.2 y 2.5 - Definir una gramática para archivos `.mdl` que describen textualmente modelos vensim y analizar el árbol para ver cómo detectar el tipo de los símbolos	10h	8h 36m	Completado
2.3 - Implementar en SSLR o ANTLR la gramática definida	1h	1h 11m	Completado
2.4 - Integrar la gramática en el plugin	3h	1h 48m	Completado

Tabla 7.1: Tareas del sprint 1

Durante este sprint se configuró el entorno de desarrollo. Para que se pareciera al entorno de producción se actualizó la version de Sonarqube a la última LTS, y se configuró para que usara la base de datos PostgreSQL. También se configuró la integración continua con Gitlab CI.

Al principio se iba a usar como base de datos MySQL. Sin embargo no tenía soporte desde la versión LTS 7.9 [47], que es la que se estaba usando. Por ello se optó por PostgreSQL [60], debido a que es gratuito y open source.

Tras esto se empezó a desarrollar un plugin básico para Sonarqube. Este era una versión simplificada y mejor organizada del plugin desarrollado durante el sprint 0. A diferencia de dicho plugin, este no tendría gramática. En su lugar que generaba una issue si el fichero analizado estaba vacío.

Tras esto, se analizó el funcionamiento de SSLR y sus ventajas y desventajas frente a ANTLR4. Finalmente se decidió usar ANTLR4. El razonamiento se detalla en la sección 6.1.

Durante las semanas previas a este sprint se estuvo en contacto con el soporte técnico de Vensim para obtener una gramática. Sin embargo afirmaron que no disponían de una gramática. Por tanto para continuar con el proyecto hubo que desarrollar una gramática.

Comenzar con una gramática de cero habría requerido un gran esfuerzo. Por suerte se encontró una gramática [11] en Github para ANTLR4. Esta gramática era muy simplificada, y al probarla con los ficheros de prueba daban muchos fallos. Para probar la gramática se usaron los modelos que se instalan junto con Vensim. Estos eran muy variados y fueron de mucha ayuda a la hora de detectar los fallos de la gramática. Se realizaron una larga serie de cambios, que se detallan en la Sección 6.2.2.

La tarea 2.5 se realizó a la vez que la 2.1. Se determinó que no se podía diferenciar entre constantes y variables mediante la sintaxis. Además no se podía inferir el tipo en una sola pasada. Serían necesarias dos: En la primera se crea una tabla de símbolos y en la segunda se aplica un algoritmo de inferencia de tipos.

Las tareas de este sprint se pudieron completar gracias a lo aprendido durante el sprint 0. Si este no se hubiera realizado habrían sido necesarios uno o varios sprints únicamente para obtener un plugin funcional.

El día 2 de octubre de 2019 se realizó una reunión con los miembros del GEEDS para determinar los estándares de nombrado y recibir feedback. Se discutió sobre los diferentes tipos que había en Vensim y las reglas que debía de cumplir cada uno.

7.4. Sprint 2 (5/10/2019-20/10/2019)

La Tabla 7.2 muestra el desglose de las tareas realizadas en este sprint y las historia de usuario abordadas en el mismo.

Al analizar la historia de usuario número tres se llegó a la conclusión de que era una epic. Se dividió en tres historias y durante este sprint se abordaron las dos primeras.

Durante la división en tareas se planteó un posible algoritmo para completar las dependencias indirectas de los símbolos. Sin embargo no fue necesario, ya que se determinó que en la tabla de símbolos las dependencias son referencias a otros símbolos. Por lo que se pueden consultar las dependencias indirectas sin necesidad de transformarlas en dependencias directas.

Tarea	Tiempo estimado	Tiempo invertido	Estado
3.1.1 - Definir la estructura de datos de la tabla de símbolos	2h	45m	Completado
3.1.2 - Definir un algoritmo de *aliasing* para anotar dependencias entre símbolos	4h	1h 20m	Completado
3.1.3 - Obtener del árbol parse los símbolos y almacenarlos en la tabla de símbolos	7h	3h 33m	Completado
3.1.5 - Definir tests automáticos	7h	6h 22m	Completado
3.2.1 y 3.2.2 - Inferir los tipos de cada símbolo a partir de los dato almacenados en la tabla y almacenarlo en la tabla de símbolos	4h	43m	Completado
3.2.3 - Añadir tests automáticos y refactor	8h	4h 37m	En progreso

Tabla 7.2: Tareas del sprint 2

La tarea 3.2.2 se realizó a la vez que la 3.2.1. Para el algoritmo de inferencia se usó un algoritmo de punto fijo que se describe en la sección 8.4. Este algoritmo funcionaría siempre y cuando no hubiera dependencias cíclicas. Durante la segunda parte del sprint se descubrió que en las variables de tipo Level puede haber dependencias cíclicas. Por ello se llegó a un acuerdo sobre cómo tratar estos casos. Esto se describe en la sección 8.4.2.

Durante el sprint review se llegó a la conclusión de que era necesario hacer refactoring y añadir más tests. Por ello se añadió como tarea para el siguiente sprint.

Se decidió que la semana del 18 de noviembre se tomaría de vacaciones del proyecto. Estas permitirían al alumno descansar y ponerse al día con el resto de asignaturas. También fue necesario una replanificación debido a que el alumno iba a participar en la final de la Primera Liga Nacional Interuniversitaria de Retos en el Ciberespacio. Esta tendría lugar entre los días 13 y 16 de noviembre en Madrid. Además de la competición se realizarían una serie de charlas y actividades que no iban a permitir al alumno trabajar en el proyecto.

7.5. Sprint 3 (21/10/2019-3/11/2019)

La Tabla 7.3 muestra el desglose de las tareas realizadas en este sprint y las historia de usuario abordadas en el mismo.

Los tests y el refactor pendientes del sprint anterior llevaron más tiempo del esperado, pero permitió acelerar la creación de las siguientes reglas.

Durante este sprint se realizaron los primeros tests de integración con Sonarqube. En estos se ejecutaba el plugin de forma indirecta con el `sonar-scanner`. El problema de esto es que cuando el escáner acaba hay un periodo de tiempo en el que se procesa el análisis. Este periodo es variable y depende de varios factores, entre ellos la cantidad de ficheros analizados

e issues generadas. Los tests de integración piden los resultados del análisis a la web de SonarQube, por ello se debe enviar la petición una vez el procesamiento ha finalizado. No se encontró una manera sencilla de detectar esto, por lo que se hizo una pausa de diez segundos antes de enviar las peticiones. Esta cifra debería de ser más que suficiente para completar el procesamiento, que normalmente tarda entre medio y dos segundos.

Tarea	Tiempo estimado	Tiempo invertido	Estado
3.2.3 - Añadir tests automáticos y refactor	6h	8h 28m	Completado
3.3.1.1 - Adaptar el contexto de Sonarqube a la tabla de simbolos y refactorizar Vensim-Check	1h	20m	Completado
3.3.1.2 - Comprobar las reglas de nombrado para los subscripts y generar la issue en caso de incumplimiento	2h	2h 20m	Completado
3.3.1.3 - Testear que genera las issues correctamente	6h	5h 9m	Completado

Tabla 7.3: Tareas del sprint 3

Por otra parte, durante este sprint se revisó y discutió el estándar de nombrado con diversos miembros del proyecto. Se realizó mediante un documento compartido en Google Drive.

Durante el sprint review se plantearon varias tareas de interés para planificarlas en sprints posteriores:

- Se detecta la necesidad de categorizar los tests para que el pipeline de integración continua pueda separar los tests que no requieren de interacción con Sonarqube.
- Se analiza la posibilidad de que en aquellos tests que se integran con el sonar-scanner se pueda detectar algún tipo de evento para saber que el análisis se ha completado y no depender de un delay de tiempo, que puede variar según van a aumentando los casos de prueba.
- Buscar si hay alguna imagen docker con sonarqube para ejecutar los tests de integración en la integración continua.

7.6. Sprint 4 (4/11/2019-12/11/2019)

La Tabla 7.4 muestra el desglose de las tareas realizadas en este sprint y las historia de usuario abordadas en el mismo.

Durante este sprint surgió un nuevo requisito por parte del GEEDS. El plugin debe de crear un fichero json con el contenido de las tabla de símbolos de los ficheros analizados. Esto permitiría que comprobaran de forma más cómoda que la información es correcta y les permite procesar esta información si lo requirieran.

Además, se completó el documento de estándares de nombrado y se actualizaron las historias de usuario para que reflejen los últimos cambios. También se descubrió que un símbolo podía estar definido varias veces, y se adaptó el plugin para que lo permitiera.

Tarea	Tiempo estimado	Tiempo invertido	Estado
Extra 1 - Refactorizar los tests de la API y categorizarlos para separar los que dependen de sonar-scanner de los que no	1h 30m	1h 53m	Completado
Extra 2 - Actualizar las historias de usuario a partir del documento de Google Drive	1h	42m	Completado
3.3.2.1 - Añadir las reglas restantes del estándar de nombrado	6h	1h 44m	Completado
3.3.2.2 - Testear las reglas añadidas y comprobar que se suben a Sonar	6h	4h 33m	Completado
Extra 3 - Hacer un fichero json con la tabla de símbolos	2h	45m	Completado
Extra 4 - Testear que se genera el json correctamente	4h	1h 34m	Completado
Extra 5 - Actualizar la clase `Symbol` y las reglas para permitir que un símbolo esté definido varias veces	1h	50m	Completado
Extra 6 Preparar la demo	1h 30m	2h	Completado

Tabla 7.4: Tareas del sprint 4

Todas las tareas del sprint se realizaron durante la primera semana, ya que la liga de ciberseguridad empezaba el miércoles de la segunda. Se decidió aprovechar el fin de semana para realizar otra tarea extra, que consistió en crear una instancia de Sonarqube en una máquina de la virtual de la escuela. Esto permitiría que los miembros de GEEDS puedan usar la máquina como demo, y que aporten feedback.

Por falta de tiempo durante este sprint solo se instaló Sonarqube con la base de datos en la máquina virtual. Pero no se llegó a instalar el plugin ni a probarlo con los ficheros de MEDEAS.

7.7. Sprint 5 (25/11/2019-15/12/2019)

Este sprint comenzó después del parón tras la liga de ciberseguridad y la semana de vacaciones. La duración de este sprint fue de tres semanas, debido a que los días 5, 6, y 9 no fueron lectivos.

La Tabla 7.5 muestra el desglose de las tareas realizadas en este sprint y las historia de usuario abordadas en el mismo.

Tarea	Tiempo estimado	Tiempo invertido	Estado
4.1 - Detallar la regla de números mágicos y sus excepciones	3h	2h 45m	Completado
4.2 - Implementar la regla de los números mágicos y sus excepciones	8h	7h 20m	Completado
4.3 - Testear la regla de los números mágicos	3h	5h	En progreso
Extra 7 - Instalar el plugin en la VM por despliegue automático, crear un usuario y ejecutar el scanner con MEDEAS	2h	3h 37m	Completado
Extra 8 - Corregir un bug encontrado en los lookups tras analizar el fichero `MEDEAS_w_v1_3.mdl`	2h	1h 44m	Completado

Tabla 7.5: Tareas del sprint 5

Durante este sprint se configuró el CI de Gitlab para desplegar de forma automática al hacer commit en la rama master. Hubo problemas a la hora de conectarse por ssh a la máquina virtual desde Gitlab debido a que no era capaz de alcanzar la VM. Tras hablarlo con los técnicos de la Escuela se logró resolver el problema.

Al analizar el fichero `MEDEAS_w_v1_3.mdl` se detectó que en el lookup `"Historic energy industry own-use"` no se generaba una issue, a pesar de no seguir el estándar de nombrado de los lookups. El símbolo era detectado como una función y no como un lookup.

Se llegó a la conclusión de que se debía a que se habían juntado dos condiciones que no se habían tenido en cuenta:

1. El símbolo es un lookup definido mediante una función. Por tanto no se podía determinar que era un lookup en la primera pasada.
2. Se realiza una llamada al lookup antes de su definición. Por tanto se definía como `FUNCTION` y se consideraba que el tipo estaba resuelto.

Tras analizarlo se determinó que no es posible determinar si una llamada es una función o un lookup en la primera pasada. Se añadió el tipo `UNDETERMINED_FUNCTION` para marcar las llamadas y se modificó el algoritmo de resolución de tipos para que los resolviera.

Al hacer este cambio se detectó que algunos de los tests relacionados con la inferencia de tipos fallaban a veces. No seguía un patrón determinado, por lo que parecían condiciones de carrera. Se solucionó al resolver otro bug en la misma clase. Sin embargo, este bug no parecía ser la causa del probema, por lo que se añadió como una tarea para descubrir la causa.

En cuanto a la regla de los números mágicos, se determinaron y programaron todas las excepciones iniciales. El resultado final está en la Sección 4.2.

La única tarea pendiente fue la detección de los números mágicos compuestos de un solo argumento. Esta tarea pasó al siguiente sprint. Además, en el sprint review se decidió que era necesario refactorizar la regla y hacer más tests.

Se decidió que no se continuaría con el proyecto hasta el día 17 de enero, debido al periodo de exámenes. Durante este tiempo si se diese la posibilidad se completarían las tareas pendientes de este sprint y algunas tareas menores.

7.8. Período del 16/12/2019 al 16/1/2020

En la Tabla 7.6 se muestran las tareas realizadas durante el período de vacaciones y de exámenes en convocatoria ordinaria.

Tarea	Tiempo estimado	Tiempo invertido	Estado
Extra 9 - Corregir la excepción de números compuestos de funciones con un solo argumento	1h 30m	1h 5m	Completado
Extra 10 - Completar los tests de la regla 4 y mejoras generales de testing	6h	6h 19m	Completado
Extra 11 - Investigar y corregir las supuestas condiciones de carrera	2h	51m	Completado
Extra 12 Mejorar el despliegue automático	1h 30m	28m	Completado
Extra 13 - Añadir un error admisible para que 5.0000000000001 sea igual a 5	2h	15m	Completado
Extra 14 - Refactors y pequeños cambios	10h	2h 55m	Completado

Tabla 7.6: Tareas realizadas en el periodo 16/12/2019-16/1/2020

El trabajo durante este periodo se realizó de forma voluntaria para ir adelantando pequeñas tareas. Así en el siguiente sprint se podría continuar con las historias de usuario.

Se averiguó por qué los tests fallaban a veces. Se debía a dos razones:

- El algoritmo de inferencia era sensible al orden de los símbolos. Dependiendo del orden en el que se resolvían algunos símbolos se consideraban funciones o lookups.

- La tabla de símbolos está implementada mediante un HashMap, que no garantiza un orden.

Se solucionó este problema por ambos frentes. Por un lado, se añadió una condición que faltaba para que el algoritmo de inferencia fuera independiente del orden. Por otro lado, se modificó el método `getSymbols` para que devolviera los símbolos ordenados siempre de la misma forma. Para que fuera más sencillo se ordenó por orden alfabético del token, pero la ordenación concreta es irrelevante para la solución. Lo que importa es que siempre se ordene de la misma forma.

En los números mágicos, para añadir un error admisible, se decidió que los números fueran floats y no doubles. El algoritmo ya estaba programado con anterioridad para transformar los candidatos a números mágicos de tokens a valores. Por tanto el error admisible ocurriría si un token representa un número que no puede alcanzar la precisión permitida por los float.

Finalmente, se realizaron una serie de refactors. Se esperaba dedicar mucho más tiempo del que finalmente fue necesario, ya que no hacía falta realizar tantos cambios.

7.9. Sprint 6 (17/1/2020-30/1/2020)

La Tabla 7.7 muestra el desglose de las tareas realizadas en este sprint y las historia de usuario abordadas en el mismo.

Tarea	Tiempo estimado	Tiempo invertido	Estado
5.1 - Definir el servicio para acceder al diccionario	4h	5h 48m	Completado
Extra 15 - Trabajar en la memoria	3h	1h 16m	En progreso
5.2.1 - Crear la regla que determina si los símbolos existen en el diccionario	5h	2h 49m	Completado
5.2.2 - Testear regla que comprueba si los nombres existen en la DB	7h	9h 41m	Completado

Tabla 7.7: Tareas del sprint 6

Aunque el comienzo oficial del sprint fue el viernes 17, no se retomó el trabajo hasta el día 21. Ya que el último examen de la convocatoria ordinaria era el día 20.

En primer lugar se determinó que la historia de usuario número 5 era una epic, y se dividió en nueve. Y se asignaron las dos primeras a este sprint.

Los días 28 y 29 se realizaron varias reuniones con miembros del GEEDS. Se acordaron diversas modificaciones en los estándares de nombrado, se añadieron nuevas excepciones a

los números mágicos y se acordó dividir la issue de los números mágicos en dos dependiendo de cuántas veces apareciera cada número.

Además se acordaron nuevas historias de usuario:

- **5.6**: Como usuario quiero poder comprobar que un símbolo (constante o variable o lookup table) tiene asociado comentario y unidades.
- **5.7** Como usuario quiero que se cree una issue en SonarQube cuando un símbolo no tiene comentario o unidades
- **5.8.1** Como usuario quiero poder comprobar que los comentarios y unidades de un símbolo coinciden con los del diccionario de datos.
- **5.8.2** Como usuario quiero que se cree una issue en Sonarqube cuando las unidades o comentarios de un símbolo no coinciden con los del diccionario de datos.
- **7** Como usuario quiero poder inyectar los símbolos definidos en un programa Vensim en el diccionario de datos pendiente de validación.

El nuevo requisito más importante fue la inyección de símbolos. Tras analizar un fichero `mdl`, el plugin debe de enviar los símbolos que sean válidos al diccionario de datos. Con esto el GEEDS trata de invertir el flujo de trabajo. En lugar de primero definir el símbolo en la base de datos y después usarlo, pueden usarlo directamente o con una mínima alteración.

Debido a las restricciones de tiempo se decidió que la inyección de símbolos estaría muy simplificada, y se ignorarían algunos casos. Además para poder afrontar el resto del proyecto se decidió eliminar la historia de usuario 6, ya que no era tan prioritaria.

Como no se había establecido todavía cómo sería la interfaz de la API del diccionario de datos se asumió una interfaz simplificada. Pero se programó minimizando el acoplamiento con el resto del plugin, ya que se asumió que en el futuro cambiaría. Para que la API pudiera comenzar se realizo una reunión con el alumno encargado de esta API para discutir la estructura que tendría que tener tanto la base de datos como el formato de intercambio de datos en los servicios REST.

Finalmente, se decidió que era hora de empezar a escribir la memoria. Aunque fue una tarea poco prioritaria, debido a que los nuevos requisitos requerirían una replanificación y más tiempo.

7.10. Sprint 7 (31/1/2020-13/2/2020)

La Tabla 7.8 muestra el desglose de las tareas realizadas en este sprint y las historia de usuario abordadas en el mismo.

Tarea	Tiempo estimado	Tiempo invertido	Estado
5.9 - Añadir un parámetro de análisis para indicar la url del servicio del diccionario	3h	44m	Completado
Extra 16 - Refactors y pequeños cambios	3h	2h 42m	Completado
Extra 17 - Añadir los números 0, 1 y 100 como excepciones en los números mágicos	1h	30m	Completado
Extra 18 - Dividir las issues de los números mágicos en 2 (INFO y MAYOR)	3h	1h 24m	Completado
Extra 19 - Cambiar el estándar de nombrado de los subscripts	1h 30m	47m	Completado
Extra 20 - Añadir los Switch como excepción a los números mágicos	1h 30m	42m	Completado
Extra 21 - Seguir trabajando en la memoria	5h	3h 47m	En progreso
5.6. - Detectar comentarios y unidades	5h	4h 9m	Completado

Tabla 7.8: Tareas del sprint 7

Durante este sprint las tareas estuvieron centradas en cumplir con los nuevos requisitos, establecidos en las reuniones del sprint anterior.

7.11. Sprint 8 (14/2/2020-27/2/2020)

La Tabla 7.9 muestra el desglose de las tareas realizadas en este sprint y las historia de usuario abordadas en el mismo.

Tarea	Tiempo estimado	Tiempo invertido	Estado
5.7.1 - Crear y testear la issue que salta si los símbolos no tienen comentarios	3h	1h 31m	Completado
5.7.2 - Crear y testear la issue que salta si los símbolos no tienen unidades	2h	1h 10m	Completado
Extra 22 - Seguir trabajando en la memoria	5h	4h 49m	En progreso
Extra 23 - Almacenar en el Json generado los comentarios y unidades	2h	1h 20m	Completado
Extra 24 - Refactors y otras comprobaciones de la gramática	1h	23m	Completado
5.5.3.1 - Almacenar los índices de los símbolos	8h	3h 20m	Completado

Tabla 7.9: Tareas del sprint 8

Durante este sprint se descubrió que dentro de las macros se podía hacer referencia a variables externas usando el sufijo $.

Para evitar problemas de scope, se estableció que los símbolos definidos dentro de una macro no se incluirían en el diccionario de datos y, por tanto, no había que sacar las issues de consistencia con el diccionario. Además, se decidió que como parte del estándar los símbolos definidos en las macros deberían tener nombres únicos en el ámbito global del fichero. Se determinó que estas tareas no eran prioritarias y solo se realizarían en caso de que sobrara tiempo.

La detección de índices fue más rápido de lo estimado debido a que se decidió que la regla no compararía subscripts, sino valores de subscripts. Esto permitió que no fuera necesario un algoritmo para inferir el subscript a partir de sus valores. Las razones para esta decisión se detallan en la Sección 8.7.

También se planteó el problema de qué hacer si un símbolo tiene comentarios o unidades diferentes dependiendo de la ecuación. Esto no puede ocurrir de forma natural en Vensim. Solo puede surgir si se modifica el fichero `mdl` arbitrariamente de forma externa a Vensim. Como el plugin asume un modelo correcto, se decidió que se cogería la primera aparición y se ignoraría el resto.

7.12. Sprint 9 (28/2/2020-12/3/2020)

La Tabla 7.10 muestra el desglose de las tareas realizadas en este sprint y las historia de usuario abordadas en el mismo.

Tarea	Tiempo estimado	Tiempo invertido	Estado
5.3 y 5.4 - Crear una issue si el tipo del diccionario no coincide con el de la base de datos	3h	1h 27m	Completado
5.8 - Crear issue si el comentario no coincide con el del diccionario	2h 30m	1h 42m	Completado
Extra 25 - Cargar el diccionario a partir del JSON acordado en la reunión	5h	5h 25m	Completado
Extra 26 - Trabajar en la memoria	3h	1h 49m	En progreso

Tabla 7.10: Tareas del sprint 9

El día 4 de marzo se realizó una reunión con el alumno encargado de realizar la API del servicio del diccionario de datos. En esta se dejó establecida su interfaz, que se implementó en el plugin a lo largo del sprint.

La carga de este sprint fue más reducida debido a que el alumno se mudó a Madrid el día 9 de marzo, para realizar las prácticas en empresa. Como estas eran a jornada completa se

decidió que los sprints siguientes tendrían una carga de trabajo menor. También se acordó el uso de Skype para realizar las weeklies.

Durante el sprint review se determinó que faltaba hacer tests de integración de las reglas que necesitan el diccionario de datos, usando mocks del servicio.

7.13. Sprint 10 (14/3/2020-26/3/2020)

El día 14 de marzo el Gobierno decretó el estado de alarma debido a la crisis del COVID- 19 [26]. Como parte del decreto se inició una cuarentena de duración indefinida. Esto no afectó de forma directa al proyecto, ya que el alumno se había mudado a Madrid la semana anterior, y ya estaba planificado trabajar y hacer las reuniones semanales en remoto. Sin embargo debido a este imprevisto las condiciones de vida y de trabajo del alumno empeoraron, dificultando el avance del proyecto.

La Tabla 7.11 muestra el desglose de las tareas realizadas en este sprint y las historia de usuario abordadas en el mismo.

Tarea	Tiempo estimado	Tiempo invertido	Estado
Extra 27 - Hacer mocks del servicio y acabar los tests de integración	2h	1h 27m	Completado
5.5.1 - Almacenar en la tabla de símbolos los valores de los subscripts que se definen mediante secuencias	4h 30m	3h 10m	Completado
5.5.2 - Crear la issue que comprueba si los valores de un subscript coinciden con los de la base de datos	2h	1h 31m	Completado
Extra 28 - Trabajar en la memoria	3h	1h 17m	En progreso
5.5.3.2 - Determinar el algoritmo de emparejamiento de subscripts	1h	30m	Completado
5.5.4 - Implementar el algoritmo de emparejamiento y crear la issue si un símbolo no está indexado de la forma que indica el diccionario	3h	1h 29m	Completado

Tabla 7.11: Tareas del sprint 10

Para poder detectar la issue que comprueba que los valores de los subscripts coincide con los del diccionario de datos fue necesario parsear las secuencias de subscripts. Estas secuencias son de la forma `subscriptName: (valueName81 - valueName95)`. En este ejemplo se generarían todos los subscripts values entre `valueName81` hasta `valueName95`, ambos incluidos. Hasta ahora no había sido necesario que la tabla de símbolos almacenara todos los

valores, ya que se puede comprobar el estándar de nombrado solo con el primero y el último valor. Este tema se detalla más en la Sección 8.3.

En el algoritmo de emparejamiento se plantearon diversas propuestas. La necesidad de este algoritmo surge porque el servicio no garantiza el orden en el que devuelve los índices. Por tanto, tiene que compararse la lista de subscripts sin tener en cuenta el orden. Al principio se intentó buscar criterios de ordenación que permitieran un emparejamiento correcto. El más prometedor era tratar de emparejar los subscripts con menos valores. Este criterio se discutió durante el sprint planning y parecía funcionar en la gran mayoría de casos. Sin embargo, a lo largo del sprint se encontró un contraejemplo con el uso de secuencias de subscripts.

Si se tiene un fichero como el fragmento de código 7.1 lo que se lee del fichero sería `[[YEAR_2017,YEAR_2017], [YEAR_2016, YEAR_2017]]`. Del servicio llegaría algo como `[YEARS_ONE_I, YEARS_TWO_I]`. El orden en el que los devuelve el servicio no importa, ya que al ordenarlo acabaría estando primero `YEARS_ONE_I`, porque su secuencia es tiene un año menos.

```
YEARS_ONE_I: (YEAR_2016~ YEAR_2018)
YEARS_TWO_I: (YEAR_2017 ~ YEAR_2020)

var[YEAR_2017,YEAR_2016] = 3
var[YEAR_2017,YEAR_2017] = 4
```

Fragmento de código 7.1: Fichero `.mdl` para el contraejemplo al algoritmo de emparejamiento de primero el subscript con menos valores

Según este criterio en primer lugar se trataría de emparejar `[YEAR_2017,YEAR_2017]` con `YEARS_ONE_I`. Esta comparación encajaría ya que `YEAR_2017` es un valor de `YEARS_ONE_I`. A continuación trataría de emparejar `[YEAR_2016, YEAR_2017]` con `YEARS_TWO_I`. Esta comparación fallaría, ya que `YEARS_TWO_I` no incluye `YEAR_2016`. Por tanto, el resultado final sería que no encaja. Sin embargo sí debería de encajar, solo que en el orden contrario (`[YEARS_TWO_I,YEARS_ONE_I]`).

Tras analizar otras posibles soluciones, se determinó que la única solución válida era usar un algoritmo de *backtracking*. Se puede encontrar más información sobre este en la sección 8.7.1.

Durante este sprint también se determinó que el servicio REST del diccionario de datos solo devuelve los símbolos que ya han sido validados. Por tanto, no es necesario que el plugin tenga en cuenta los casos en los que al comparar un símbolo con el símbolo del servicio, este último no estuviera validado.

La única tarea pendiente de este sprint fue la realización de tests para el algoritmo de emparejamiento, que se movieron al siguiente.

7.14. Sprint 11 (27/3/2020-9/4/2020)

La Tabla 7.12 muestra el desglose de las tareas realizadas en este sprint y las historia de usuario abordadas en el mismo.

Tarea	Tiempo estimado	Tiempo invertido	Estado
Extra 29 - Hacer tests del algoritmo de emparejamiento	2h	4h 29m	Completado
Extra 30 - Actualizar la regla del algoritmo de emparejamiento para que no genere issue si el modelo no es válido	4h 30m	3h 10m	Completado

Tabla 7.12: Tareas del sprint 11

Los tests del algoritmo de emparejamiento llevaron más tiempo del esperado debido a la complejidad y a la aparición de bugs.

Durante el sprint se detectó que los símbolos podían tener un subscript y un subscript value en la misma columna del índice. En dicho caso el modelo no sería válido. Se prefirió lanzar un mensaje de aviso pero no sacar issue, ya que el mensaje de la issue podría confundir al usuario.

Durante este sprint también se debería de haber trabajado en la memoria, sin embargo por la carga de trabajo de las prácticas en empresa y otras razones personales no se pudo avanzar.

7.15. Sprint 12 (10/4/2020-23/4/2020)

La Tabla 7.13 muestra el desglose de las tareas realizadas en este sprint y las historia de usuario abordadas en el mismo.

Tarea	Tiempo estimado	Tiempo invertido	Estado
7.1.1 - Inyectar los símbolos válidos en el servicio	3h	2h 8m	Completado
7.2 - Testear la inyección de símbolos	6h	5h 48m	Completado
Extra 31 - Intercambiar los mocks con los del servicio para probar la integración	3h	1h 12m	Completado

Tabla 7.13: Tareas del sprint 12

Para poder probar la integración entre el plugin y el servicio se intercambiaron mocks con el desarrollador del servicio REST del diccionario de datos. También se modificó la interfaz del servicio debido a unos cambios de requisitos que hubo en la base de datos.

Se decidió que al inyectar un símbolo se enviarían todos los metadatos del símbolo (nombre, tipo, unidades, definición, valores de subscripts y si está indexado). No se incluyeron los índices del símbolo porque se determinó que sería difícil que el servicio los relacionara en la base de datos, y sería mejor que se añadieran y revisaran a mano.

También se ignoró la categoría del símbolo, ya que no es un metadato de Vensim, sino un concepto de MEDEAS. Durante las reuniones de enero se discutió que los modeladores querían que los símbolos tuvieran una categoría, y que al inyectar se podría usar como categoría la vista en la que está el símbolo. Sin embargo esta información no es accesible de forma sencilla en el fichero `.mdl` y requeriría un análisis sobre como obtenerlo. Además era muy probable que cuando los modeladores validaran el símbolo en el diccionario sobreescribieran la categoría, ya que la vista de un símbolo no aporta información sobre este. Por ello se desestimó pasar la categoría.

En el momento del sprint la petición de la inyección no devolvía nada. Pero se programó de forma que fuera sencillo añadir una gestión de errores si en el futuro fuera necesario.

7.16. Sprint 13 (24/4/2020-7/5/2020)

La Tabla 7.14 muestra el desglose de las tareas realizadas en este sprint y las historia de usuario abordadas en el mismo.

Tarea	Tiempo estimado	Tiempo invertido	Estado
7.1.2 - Acabar los tests	2h	1h 15m	Completado
Extra 32 - Continuar con la memoria	8h	8h 2m	En progreso

Tabla 7.14: Tareas del sprint 13

Este sprint se consideró el último sprint de programación. A partir de este se trabajaría exclusivamente en la memoria del proyecto.

7.17. Sprints finales (8/5/2020 - 13/7/2020) de documentación

Después del sprint 13 se siguieron realizando sprints al igual que se habían ido realizando. En todos la única tarea era seguir con la memoria. Debido a la carga de trabajo adicional

del alumno no se estimaron las horas que se dedicarían por sprint, sino que se dejó libertad. Aunque se seguía controlando el avance mediante weeklies.

Las prácticas en empresa del alumno acabaron el día 18 de junio. Sin embargo, como le prorrogaron el contrato tuvo que seguir compaginando el TFG con la jornada completa.

El día 21 de junio el estado de alarma finalizó [8], y se entró en la última fase de la desescalada, la nueva normalidad. Esto permitió de nuevo los viajes entre comunidades. Ese mismo día el alumno se mudó de vuelta a Valladolid, lo que le permitió trabajar en unas condiciones mucho mejores que las que tuvo en Madrid. Además pudo descansar más, y le permitió dedicarle más horas a la memoria de forma consistente.

Este periodo estuvo formado por cinco sprints, durante los cuales se dedicaron 78 horas para completar la memoria. Durante el último sprint, el método de autenticación en el servició cambió a propuesta de uno de los socios del proyecto (Agencia Griega de Energía) que serán los responsables de mantener el diccionario de datos en funcionamiento en sus servidores. Además, los desarrolladores del servicio sacaron una versión de prueba, por lo que se hicieron pruebas de integración con el servicio. Todo esto llevó 5 horas adicionales.

7.18. Resumen

La duración del proyecto ha acabado siendo mucho más larga de lo esperado. Esto se debe principalmente a los cambios en los requisitos y a la elevada carga de trabajo que el alumno ha tenido aparte del proyecto.

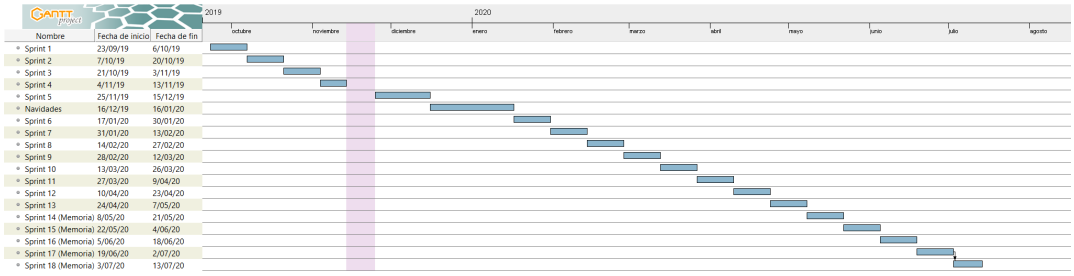

Figura 7.1: Diagrama de Gantt final del proyecto.

Los siete sprints de programación se expandieron a trece, aunque en los últimos ya se combinaba programación y documentación. Los tres sprints exclusivos de documentación acabaron siendo cinco. Al final se dedicaron algo más de 300 horas al proyecto. La figura 7.1 muestra el calendario final del proyecto.

7.19. Costes simulados finales

La duración del proyecto acabó siendo de nueve meses, por lo que los costes por la amortización del portátil aumentan. Las horas trabajadas son las mismas que se estimaron. Por ello, los costes son ligeramente superiores a los costes base previstos, pero no superan el colchón del 25 % que se estableció.

Los costes simulados finales se pueden ver en la tabla 7.15.

Sueldos	3.462€
Material	209,81€
Suma	3.671,81€
Total	**4.515,96€**

Tabla 7.15: Presupuesto simulado final del proyecto

7.20. Costes reales finales

Debido al aumento en los requisitos, la beca también se prolongó hasta marzo. No se prolongó hasta junio ya que a partir de abril la mayoría de carga de trabajo era la escritura de la memoria.

Como la duración final de la beca fue de nueve meses, el coste real ascendió a 2700€.

Capítulo 8

Diseño e implementación de la solución

8.1. Introducción

Para llevar a cabo el proyecto el primer paso es obtener una gramática y crear un parser capaz de leer los ficheros `.mdl`. Los cambios realizados a la gramática de partida se describieron en la Sección 6.2. A continuación, hay que crear la base del plugin, que de soporte a añadir todas las reglas necesarias.

Tras esto, hay que implementar las reglas, para lo cual hace falta definir varios algoritmos, como la obtención de la tabla de símbolos, la obtención de números mágicos o el emparejamiento de índices.

8.2. Implementación de VensimScanner

La lógica de los plugins de Sonarqube se encuentra en el sensor, implementado en la clase `VensimSquidSensor`. Su método más importante es `execute(SensorContext sensorContext)` que se ejecuta cuando se realiza un análisis con SonarScanner.

Para reducir el acoplamiento, se decidió que este método se encargaría de extraer la información necesaria del contexto y delegaría el análisis en una segunda clase: `VensimScanner`.

El método `execute` toma como argumento un objeto que representa el contexto de la ejecución. A partir de este se pueden obtener, entre otros, los parámetros del análisis y los ficheros que SonarScanner está analizando.

El contexto contiene una referencia a todos los ficheros, no solo a los del lenguaje del plugin, por ello existen métodos como `hasLanguage` o `hasType` para filtrarlos. No es necesario

```
@Override
public void execute(SensorContext sensorContext) {
    FilePredicates p = sensorContext.fileSystem().predicates();

    String dictionaryService = sensorContext.config().get(
        DICTIONARY_SERVICE_PARAMETER).orElse("");

    Iterable<InputFile> files = sensorContext.fileSystem().
        inputFiles(p.hasLanguage(VensimLanguage.KEY));
    List<InputFile> list = new ArrayList<>();
    files.forEach(list::add);
    List<InputFile> inputFiles = Collections.unmodifiableList(list);

    VensimScanner scanner = new VensimScanner(sensorContext, checks,
         new JsonSymbolTableBuilder(), new ServiceController(
        dictionaryService));
    scanner.scanFiles(inputFiles);
}
```

Fragmento de código 8.1: Método `execute` de `VensimSquidSensor`.

que la lista de ficheros sea inmutable, pero se consideró una buena práctica para poder detectar rápidamente si se intenta modificar.

Para la implementación de esta clase se tomó como base la clase `JavaScriptSensor` [36] de `SonarJS`. Esta clase mezcla el Sensor y el Scanner, pero los métodos `analyseFiles` y `analyze` sirvieron para entender el funcionamiento del plugin, y acabaron resultando en los métodos `scanFiles` y `scanFile`.

El método `scanFiles` es muy sencillo, tan solo itera los ficheros llamando al método `scan` en cada uno. También comprueba si el usuario ha cancelado el análisis, en cuyo caso el plugin finaliza. Este se puede ver en el fragmento de código 8.2.

El método `scanFile` se encarga de llamar al módulo de parseo para obtener la información necesaria, que almacena en una estructura uniforme para todas las reglas. Se decidió que esta estructura sería un objeto contexto, `VensimVisitorContext`, al igual que se realiza en `SonarJS` [36]. La estructura básica del método se muestra en la figura 8.3.

Cada fichero tiene su propio objeto contextual. Este almacena la tabla de símbolos, el árbol obtenido durante el parseo, y las issues que se han encontrado en el fichero.

Una vez obtenido el contexto, se ejecutan todas las reglas. Las reglas deben de ser independientes, ya que Sonarqube no garantiza un orden, y además pueden ser desactivadas o activadas por los administradores de un proyecto.

La última parte es mandar las issues al servidor. Las issues que generan las reglas se

```
public void scanFiles(List<InputFile> inputFiles) {
        for (InputFile vensimFile : inputFiles) {
            if (context.isCancelled()) {
                return;
            }
            try {
                scanFile(vensimFile);
            } catch (Exception e) {
                LOG.warn("Unable to analyze file '{}'. Error: {}",
                    vensimFile.toString(), e);
            }
        }
    }
}
```

Fragmento de código 8.2: Método `scanFiles` de `VensimScanner`.

```
public void scanFile(InputFile inputFile) {

    try {

        String content = inputFile.contents();
        ...

        VensimVisitorContext visitorContext = new
            VensimVisitorContext(root, symbolTable);

        checkIssues(visitorContext);
        saveIssues(inputFile,visitorContext.getIssues());

    } catch (IOException e) {
        LOG.error("Unable to analyze file '{}'. Error: {}",
            inputFile.toString(), e);
    }catch (ParseCancellationException e){
        LOG.error("Unable to parse the file '{}'. Message {}",
            inputFile.toString(),e.getLocalizedMessage());
    }

}
```

Fragmento de código 8.3: Fragmento del método `scanFile` de `VensimScanner`.

almacenan en un formato independiente de Sonarqube, por tanto hay que transformarlas

al formato que establece la librería de Sonarqube. Es posible almacenar las issues directamente en el formato de Sonarqube, pero se decidió que era mejor separarlo para reducir el acoplamiento.

```
  private void saveIssues(InputFile file, List<Issue> issues){
     for (Issue preciseIssue : issues) {
          RuleKey ruleKey = checks.ruleKey(preciseIssue.getCheck());

          NewIssue newIssue = context
                  .newIssue()
                  .forRule(ruleKey);


          TextRange range = file.selectLine(preciseIssue.getLine());
          NewIssueLocation newLocation = newIssue.newLocation()
                  .on(file);
          newLocation.message(preciseIssue.getMessage());
          newLocation.at(range);

          newIssue.at(newLocation);
          newIssue.overrideSeverity(preciseIssue.getSeverity());
          newIssue.save();
     }
}
```

Fragmento de código 8.4: Método `saveIssues` de `VensimScanner`.

Como se puede ver en el fragmento de código 8.4, las issues no se asocian a las instancias de las reglas sino a sus claves únicas, por ello se llama al método `ruleKey`.

Hay varias formas de indicar la localización de una issue. En este proyecto las issues saltan en la línea en la que se encuentran. Sin embargo también se podría hacer que ocupasen varias líneas, o fragmentos de una línea.

Las reglas tienen una gravedad/severidad por defecto, que se puede indicar con el atributo `priority` de la anotación `@Rule`. Sin embargo se puede sobreescribir usando `overrideSeverity`, lo cual permite que varias issues de una misma regla tengan gravedades diferentes.

8.2.1. Número de líneas

Si se crea un plugin siguiendo lo descrito en esta sección y se ejecuta, se obtiene un panel vacío, como el de la Figura 8.1, en el que tan solo aparece `The main branch has no lines of code`. Esto no significa que el plugin no funcione. Las issues se generaran correctamente, pero el escáner no aporta la suficiente información para crear el panel completo.

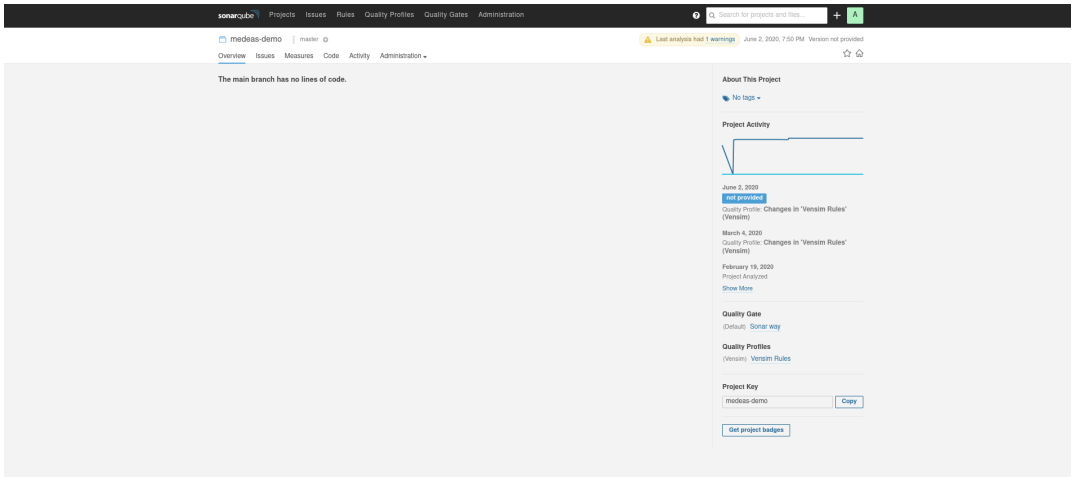

Figura 8.1: Panel vacío de un proyecto Sonarqube.

Para que aparezca el panel estándar de Sonarqube hay que pasarle a Sonar el número de líneas analizadas en cada fichero. Sonarqube permite diferenciar entre diferentes tipos de líneas, por ejemplo código y comentarios. Como en este proyecto el objetivo de añadir las líneas es únicamente mejorar el panel, se considera que todas las líneas del fichero son líneas de código.

La implementación es sencilla, hay que añadir las línas que se muestran en el Fragmento de código 8.5 en `scanFile`.Una vez hecho el cambio, el panel será similar al de la Figura 8.2.

```
int lines = content.split("[\r\n]+").length;
context.<Integer>newMeasure().forMetric(CoreMetrics.NCLOC).on(
    inputFile).withValue(lines).save();
```

Fragmento de código 8.5: Código necesario para calcular el número de líneas de un fichero y guardarlo en Sonarqube.

8.3. Obtención de la tabla de símbolos cruda

En el Capítulo 6 se explicaron los cambios que se hicieron a la gramática de partida y por qué se decidió usar visitors para recorer el árbol. Tras esto hay que definir el algoritmo para crear la tabla de símbolos.

Sin embargo, esto no es tan sencillo como recorrer el árbol mediante un visitor, ya que no se puede saber el tipo de un símbolo en una sola pasada.

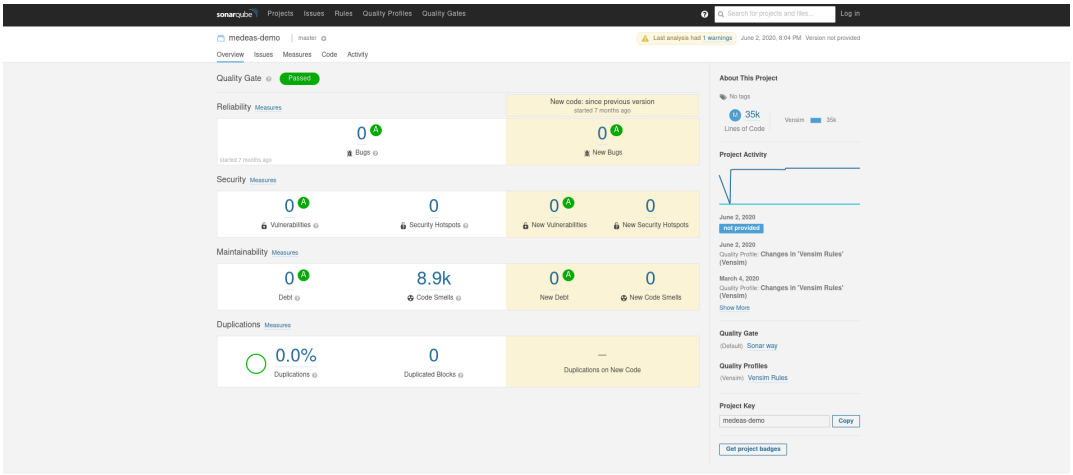

Figura 8.2: Panel de un proyecto Sonarqube tras añadir el conteo de líneas al plugin.

Es necesario realizar una segunda pasada por dos razones:

1. Las constantes y variables no tienen una sintaxis que permita diferenciarlos. Si un símbolo tiene una referencia a una variable o a una función no pura, entonces es variable. De lo contrario, es constante.
2. Vensim no garantiza que la definición de un símbolo aparece en el fichero antes de ser referenciado. Lo cual hace que en el momento en el que se recorre el árbol no se tenga información suficiente como para determinar si es un símbolo es constante o variable.

La solución es dividir la obtención de la tabla de símbolos en dos fases: La obtención de una tabla de símbolos cruda, y la inferencia o resolución de dicha tabla.

Para obtener la tabla cruda se recorre el árbol mediante el visitor. Esta tabla contiene la información que se puede obtener durante la primera pasada: línea, unidades, dependencias, comentario, etc. El único dato que no se puede determinar es el tipo. Aunque en algunos casos, como las `unchangeable constants` o los `reality checks` sí se puede determinar por la sintaxis.

Al recorrer el árbol se marca como `UNDETERMINED_FUNCTION` a las llamadas a una función nueva (que no se había encontrado hasta ahora). Esto es necesario porque no es posible diferenciar en todos los casos entre una llamada a una función y una llamada a un lookup. El resto de símbolos en los que no se puede inferir el tipo se marcan como `UNDETERMINED_TYPE`.

Además, si un símbolo está indexado puede tener múltiples ecuaciones. Se considera que si al menos una es variable, todo el símbolo lo es. Por ello, la tabla de símbolos no diferencia entre ecuaciones, y junta todas las dependencias en una sola lista.

Para obtener la tabla cruda hay que instanciar `RawSymbolTableVisitor` y llamar al método `getSymbolTable` pasando la raíz del árbol del fichero. Este método crea una tabla

de símbolos vacía y la almacena como un atributo, tras esto empieza a recorrer el árbol, que irá rellenado dicha tabla.

Dependencias de un símbolo

Para poder capturar las dependencias, cada método del visitor devuelve un símbolo, o una lista de los símbolos que ha generado. Por ello, se han implementado varias reglas que no definen símbolos para que devuelvan una lista vacía, porque si no provocarían un `NullPointerException`.

Al llegar a una ecuación se almacena la información inmediata del símbolo y tras esto se visitan los nodos hijo, y se almacena el resultado del método como dependencias del símbolo que se estaba analizando.

Hay que tener en cuenta que no se almacena la relación entre las dependencias. En vez de formar un árbol se almacenan aplanadas. Si un símbolo tiene una sola ecuación en la que aparece `ABS(SQRT(foo))+ABS(var)` la lista almacenaría algo similar a `[ABS, SQRT, foo, var]`. Varias ecuaciones pueden generar las mismas dependencias. Por ejemplo `ABS(var) + SQRT(foo)` daría el mismo resultado. Al hacer esto se pierde información, pero no es necesaria para el análisis. Para determinar el tipo es suficiente con saber si `foo` o `var` son constantes o variables, y si `ABS` y `SQRT` son funciones puras. La relación entre las dependencias es irrelevante.

La dependencias que se almacenan son referencias a otros objetos de tipo `Symbol`, por tanto los métodos del visitor no pueden crear directamente los objetos cada símbolo cada vez que encuentran uno, porque si el símbolo ha aparecido antes habría dos instancias diferentes para el mismo símbolo, lo cual provocaría que cada símbolo tuviera información parcial. Por ello, llaman a un método `getSymbolOrCreate` que busca en la tabla de símbolos si ya existe, y si no lo crea el propio método y lo devuelve. De esta forma es transparente, ya que los métodos del visitor no tienen por qué saber si el símbolo es nuevo o ya existía.

La única excepción son las llamadas a funciones. En dicho caso si el símbolo no existía, se marca como `UNDETERMINED_FUNCTION`, pero si ya existía entonces no se sobreescribe. Esto se debe a que como las funciones no tienen una sintaxis en las que se define, hay que establecer el tipo cuando se lee una llamada. Pero puede haber llamadas a lookups que se han definido antes y por tanto ya se sabe que son un lookup. Si se sobreescribiera el tipo se perdería la información y el resultado sería erróneo.

Los valores de los subscripts se almacenan como dependencias del subscript. Aunque también pueden tener dependencias que no sean valores, por ejemplo si se carga el subscript mediante una llamada a `GET DIRECT SUBSCRIPT`.

Secuencias de subscripts

Todos los símbolos se pueden extraer directamente del fichero `.mdl`, a excepción de las secuencias de subscripts. Estas son del formato `subscriptName: (valueName81 - valueName95)`.

En este ejemplo el subscript `subscriptName` tendría todos los entre `valueName81` hasta `valueName95`, ambos incluidos. La sintaxis debe cumplir las siguientes condiciones:

- Si el nombre del valor tiene varias palabras, solo puede haber un espacio entre cada una.
- Si hay texto, tiene que ser igual en el valor inicial que en el final.
- Si hay texto, el número tiene que ir después del texto.
    - Esto también incluye si hay varios números. El que marca el rango sería el último número, y el resto tiene que ser constante (`valueName 80 0 - valueName 80 100`) es válido. Pero ni (`valueName 70 0 - valueName 80 10`) ni (`valueName 0 10 - valueName 80 10`) lo son.
- El segundo número del rango tiene que ser mayor que el primero.

Para generar los valores del subscript, se usa una expresión regular para extraer tanto el texto como el número. Después se comprueba que ambos textos son iguales, y si encajan, se generan todos los valores entre medias y se añaden como valores del subscript.

En caso de que la secuencia no sea válida se lanza un warning, y solo se toman como los que están escritos explícitamente en el fichero: el primero y el último de la secuencia.

Copias de un subscript

Vensim tiene el operador `<->` que permite copiar un subscript. La copia contendrá exactamente los mismos valores que el original. El problema es que es posible definir la copia antes del original, por tanto no se puede simplemente copiar la lista de dependencias (que en los subscript contiene los valores).

La solución que se tomó es, en vez de copiar la lista pasar una referencia de la lista original a la copia del subscript. De esta forma cuando se recorran los nodos que definen al subscript original se añadirán a la lista común, por lo que al final del recorrido ambos tienen los valores adecuados.

8.4. Inferencia del tipo de un símbolo

Una vez obtenida la tabla de símbolos cruda es necesario resolver los tipos que han quedado sin determinar. Hay dos tipos de símbolos sin determinar: `UNDETERMINED_TYPE` y `UNDETERMINED_FUNCTION`. El segundo, como se ha explicado previamente, es necesario porque cuando se encuentra una llamada a una función, no se sabe si es un lookup o si es una función normal.

Los lookups se pueden definir de dos formas: mediante su sintaxis única o cargándolo de un fichero. En el primer caso sí es posible diferenciarlo del resto de las funciones y no genera problemas. Pero si se carga de un fichero con las funciones `GET XLS LOOKUP` o `GET DIRECT LOOKUP`, entonces tiene la misma sintaxis que las constantes y variables y no se puede diferenciar de estas.

Además hay que tener en cuenta que dependiendo de la situación, un lookup podría ser un `UNDETERMINED_FUNCTION` o un `UNDETERMINED_TYPE`. Si el símbolo se define pero no se llama, será un `UNDETERMINED_TYPE`, pero en la mayoría de situaciones sí se llamará, y por tanto se leerá como un `UNDETERMINED_FUNCTION`.

Para almacenar la información de la primera pasada se usa la tabla de símbolos, los atributos clave para este algoritmo son el tipo y las dependencias del símbolo, que son los símbolos a los que se hace referencia en la ecuación.

La segunda pasada consiste en realizar un algoritmo de punto fijo sobre la tabla de símbolos, para resolver los tipos sin determinar.

En cada iteración se trata de resolver todos los símbolos pendientes. Resolver una función es sencillo: Si depende de `GET XLS LOOKUP` o de `GET DIRECT LOOKUP`, entonces es un lookup. Si no, es una función.

Para resolver el resto de símbolos hay que recorrer sus dependencias. El criterio para determinar el tipo de un símbolo es:

```
Para cada dependencia:
    Si la dependencia es variable o una función no pura:
        El símbolo es variable.
    Si la dependencia depende de una función generadora de lookups:
        El símbolo es un lookup.
    Si la dependencia no está determinada:
        No hacer nada.

Si no se ha resuelto todavía:
    Si hay alguna dependencia no determinada:
        Intentarlo otra vez en la siguiente iteración.
    Si todos los símbolos están determinados:
        El símbolo es una constante.
```

Con que una dependencia sea variable está garantizado que el símbolo también lo es. Pero si no hay ninguna dependencia variable y quedan símbolos sin determinar hay que esperar a que se determinen, puesto que podrían ser variables o constantes.

Este algoritmo garantiza que si no hay ciclos en las dependencias, el algoritmo se completa.

Vensim no permite ciclos de dependencias. Sin embargo, para evitar entrar en un bucle infinito al analizar un fichero `.mdl` incorrecto, el algoritmo para si durante una iteración no se ha resuelto ningún símbolo.

8.4.1. Origen de la variabilidad

El algoritmo de resolución de tipos permite propagar las variables, pero es necesario que surjan de algún sitio. Si no hay nada que cause que un símbolo sea variable, todos serían constantes.

Por ello a partir de la documentación se obtuvo una lista con la mayoría de funciones no puras, que se confirmaron preguntando al GEEDS. Esta lista se encuentra en la sección4.1.1.

Además antes de iniciar el algoritmo se añade la variable `Time` a la tabla de símbolos. De esta forma se propaga la variabilidad a todos los símbolos que dependan del tiempo.

8.4.2. Dependencias cíclicas

No puede haber dependencias cíclicas en las constantes ni variables. Sin embargo, las variables de tipo `Level`/Caja son una excepción. Un ejemplo de esto se muestra en la Figura 8.3.

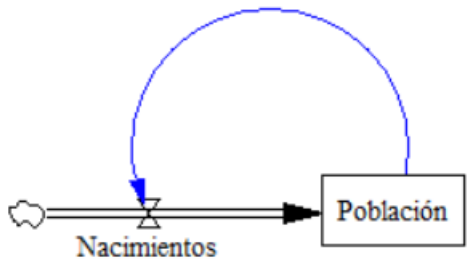

Figura 8.3: Representación gráfica en Vensim de una dependencia cíclica.

En estos casos hay dos opciones: o bien todos los símbolos involucrados son constantes, o bien todos son variables. Debido a que es poco probable que todos sean constantes, se decidió hacer una simplificación y asumir que siempre son variables.

Las variables de tipo caja se identifican por llamar a la función `INTEG`. Para implementar esta decisión se consideró que esta función no es pura. De hecho, no lo es porque va modificando el valor de los símbolos por cada paso de los algoritmos de Euler o Runge-Kutta para resolver sistemas de ecuaciones diferenciales. Así, el tipo se puede determinar de manera inmediata y el ciclo se rompe.

8.5. Detección de números mágicos

La regla de detección de números mágicos funciona de forma diferente al resto, ya que no usa la tabla de símbolos. En su lugar recorre el árbol con otro visitor y genera su propia tabla de símbolos, solo que en vez de capturar las constantes y variables almacena los números. Para evitar confusión entre la tabla de símbolos mencionada hasta ahora, y la tabla de símbolos que contiene números mágicos, se utilizará a partir de ahora el término "tabla de números"para referirse a la última. Aunque ambas están implementadas con la clase `SymbolTable`

El visitor no puede simplemente anotar cada vez que se recorre un nodo de tipo número, ya que entonces no detectaría las excepciones, explicadas en la Sección 4.2. El visitor se encarga de recorrer solo aquellos segmentos del árbol en el que hay números que podrían ser mágicos, e ignora los caminos en los que no hay números mágicos.

Hay que tener en cuenta que el número de veces que tiene que aparecer un número para ser considerado mágico depende de un parámetro de regla, que se explican en la sección 5.3.1. Por tanto, la tabla de números almacena todas las apariciones de un número en los contextos en los que podría ser mágico. Y después la regla decide en base al número de veces si es un número mágico o si no.

El visitor tiene comprobaciones para cumplir las excepciones. Por ejemplo, al llegar a un nodo de tipo `Equation` se comprueba si la expresión que se asigna es un número (fragmento de código 8.6). Si lo es, entonces es una definición de una constante y no se considera un número mágico. Por tanto no se visitaría el nodo de la expresión. Pero si no es un número entonces es una expresión que podría contener números, y estos sí serían números mágicos.

```
@Override
public Void visitEquation(ModelParser.EquationContext ctx) {
    if(ctx.expr()==null || exprIsANumber(ctx.expr()))
        return null;

    return super.visit(ctx.expr());
}
```

Fragmento de código 8.6: Método `visitEquation` de `MagicNumberTableVisitor`.

En la excepción de los tabbed arrays, los arrays bidimensionales y los unidimensionales la solución es directa: cortar el recorrido en cuanto se llega al nodo. En los lookups es similar pero es posible que en su definición haya una llamada a la función `GET XLS LOOKUPS`. Si es una llamada válida, no debería de contener números pero, por si acaso, se recorre.

La excepción de `WITH LOOKUP` requiere que cuando se recorre un nodo de tipo `Call`, se compruebe si la llamada es a dicha función. Si lo es, solo se visita el primer argumento. En caso contrario se visitan todos.

La detección de switches es algo más complicada, ya que la diferenciación entre switch

```
@Override
public Void visitTabbedArray(ModelParser.TabbedArrayContext ctx) {
    return null;
}

@Override
public Void visitLookupDefinition(ModelParser.
    LookupDefinitionContext ctx) {
    if(ctx.call()!=null)
        super.visitCall(ctx.call());
    return null;
}

 @Override
public Void visitConstList(ModelParser.ConstListContext ctx) {
    return null;
}
```

Fragmento de código 8.7: Métodos de `MagicNumberTableVisitor` que contienen las excepciones de los lookups y arrays unidimensionales y bidimensionales

y constante es más semántica que sintáctica. Por ello se llegó al estándar de que todos los switches tendrían el prefijo `SWITCH_` como se explicó en el Capítulo 4.

Siguiendo este convenio, cuando se llega a un nodo de tipo `ExprOp` se comprueba si la operación es una comparación. Si lo es, se mira si alguno de los dos lados es un switch (porque contiene el prefijo acordado). Si al menos uno lo es, se ignora la expresión. En caso de que no lo sean, se recorren ambos lados de la expresión.

```
@Override
public Void visitExprOperation(ModelParser.ExprOperationContext ctx)
    {
    if(ctx.op.getType() == ModelLexer.Equal)
        if(isASwitch(ctx.expr(0).getText()) || isASwitch(ctx.expr(1)
            .getText()) )
          return null;

    return super.visitExprOperation(ctx);

}
```

Fragmento de código 8.8: Método `visitExprOperation` de `MagicNumberTableVisitor`.

Finalmente, cuando se recorre un literal entero o flotante, se añade una entrada a la tabla de números. Debido a las excepciones mencionadas se implementan ignorando ramas del árbol, solo se llegará a un nodo numérico cuando sea un candidato a número mágico.

8.5.1. Valores en vez de cadenas

Para valorar si un número es mágico no se tiene en cuenta el token, sino su valor. Por ejemplo 1 y 1.0 son tokens diferentes, pero son el mismo número. Lo mismo pasa con --2 y 2.

Para evitar ambos casos, al entrar a un nodo literal se realizan dos acciones:

- Se traduce el número de cadena a flotante o entero, dependiendo del nodo concreto. Para esto se usa una función propia, ya que `Integer.parseInt` no permite que los números tengan varios signos seguidos (como `--1`), pero Vensim sí.
- Cuando se lee un flotante, se comprueba si es equivalente a un número entero, en cuyo caso se transforma.

La tabla de símbolos almacena tokens, que son de tipo string. Por tanto una vez obtenido el valor, se vuelve a pasar a cadena. Sin embargo, gracias a haber obtenido el valor, tanto --2 como 2.0 acabarán leyéndose como un token '2'.

8.5.2. Números compuestos

Como se explica en la Sección 4.2, las llamadas a algunas funciones puras se consideran números compuestos, por ejemplo `SQRT(4)`. Los números compuestos siguen las mismas reglas que los números normales, pero hay que diferenciarlos de las llamadas a funciones normales. Además, pueden estar anidados. Por ejemplo `SQRT(SQRT(4))` también es un número compuesto.

Para que una llamada se considere compuesta debe cumplir dos condiciones:

- Todas las llamadas tienen que ser a funciones que permiten formar números compuestos. La lista completa está en la Sección 4.2.
  - Es importante que todas las llamadas anidadas sean compuestas. `SQRT(SQRT(4))` es anidado, pero `SQRT(RANDOM UNIFORM(0,1,2))` no.
- Todos los argumentos tienen que ser literales. No puede haber referencias a símbolos. `A = LOG(3,CONST)` no es un compuesto, y el 3 se consideraría candidato a número mágico, aunque sea una asignación de un valor constante.

Comprobar estas condiciones es difícil, ya que requiere recorrer subárboles y habría que mantener un estado para diferenciar cuando el visitor está buscando un número compuesto

y cuando está recorriendo el árbol de forma normal. Por ello se consideró que lo maś sencillo era crear un segundo visitor: `CompoundMagicNumberVisitor`. Su única responsabilidad es verificar si una llamada es un número compuesto o no.

Los métodos de este visitor devuelven booleans: True si el nodo que se está leyendo podría formar parte de un número mágico compuesto, y false si rompe una de las dos condiciones. Los nodos que tienen varios subnodos realizan la operación `AND` con el resultado que obtengan de estos. De forma que con que un solo nodo rompa una condición, se devolverá false.

Si se encuentra un número compuesto candidato a número mágico, la tabla de números contendría el texto con la llamada entera. Es decir para `SQRT(4)` se almacenaría toda la cadena "`SQRT(4)`" en vez de solo el número `4`.

8.6. Obtención de la tabla de símbolos remota

El plugin tiene una serie de reglas que comprueban la consistencia entre los símbolos del fichero `.mdl` y la información almacenada en el diccionario de datos compartido por todo el proyecto. Esto se realiza mediante una API desarrollada por otro estudiante de la UVa. La estructura de la API se acordó durante una reunión como se explicó en el Capítulo 7.

Para saber la dirección de la API se usa el parámetro de análisis `dictionaryService`. Al entrar en el método `execute` de `VensimSquidSensor` se extrae el parámetro, se crea una clase de tipo `ServiceController` con el parámetro, y se pasa este objeto al escáner.

El parámetro se almacena en `ServiceController` para abstraer la información del escáner. Además si hay varios ficheros a escanear se puede compartir el mismo controlador, por lo que no es necesario que el escaner indique varias veces cuál es el parámetro.

El plugin de Sonarqube debe de pedir al diccionario de datos la información sobre los símbolos que ha encontrado en un fichero `.mdl` analizado. Por tanto, la petición se realiza después de obtener la tabla de símbolos pero antes de ejecutar las reglas. Para ello, llama a `getSymbolsFromDb` pasando una lista con los símbolos que ha encontrado. Este método se encarga de filtrar los símbolos para eliminar las funciones y los símbolos por defecto. Además, se encarga de loggear los errores.

La obtención de la tabla en sí se delega en `DBFacade.getExistingSymbolsFromDB`. Este método a su vez llama `ServiceConnectionHandler.sendRequestToDictionaryService`, que se encarga de mandar la petición. Una vez obtenido el resultado, lee el JSON obtenido para crear la tabla de símbolos.

Para almacenar el estado esperado de los símbolos se usan las mismas clases que forman la tabla de símbolos local: `SymbolTable` y `Symbol`. La única diferencia es que solo almacenan la información que tiene la base de datos que soporta el diccionario de datos compartido. Por ejemplo, esta no contiene información sobre la línea en la que se encuentra el símbolo, por lo que estará vacío. Además, en la base de datos un símbolo puede estar en varios módulos, mientras que en local como hay una tabla de símbolos por fichero todos los símbolos tendrán un solo módulo.

Si el servicio no está activo o hay algún error, la tabla de símbolos remota tendrá valor `null`. En este caso todas las reglas que lo necesitan se ignorarán.

8.6.1. Interfaz del servicio

La interfaz acordada para obtener la información de los símbolos es la siguiente. Se manda una petición de tipo `POST` al endpoint `qaGetSymbolDefinition`. La url base se establece en el parámetro de análisis. Como datos se pasa la lista de símbolos que se quiere obtener, con el formato `{"symbols":["symbolA", "symbolB"]}`, y como salida devuelve el json con la información que tiene el diccionario. El Fragmento de código 8.9 muestra un ejemplo de lo que devuelve el servicio.

Como el servicio no está creado exclusivamente para el plugin, contiene información que este no usa. Por ejemplo la lista de módulos y categorías. Además los símbolos tienen dos tipos: `ProjectTypeOfValue` y `ProgrammingSymbolType`. El primero se refiere al tipo semántico, que se usa dentro del proyecto, por ejemplo `Historical data series`. Y el segundo hace referencia al tipo de Vensim, que es común para todos los modelos creados con esta herramienta. La tabla de símbolos solo almacena el tipo de Vensim, ya que es el único que se puede detectar. El tipo del proyecto es semántico, ya que varios tipos de proyecto pueden hacer referencia a un mismo tipo de símbolo Vensim y al revés.

```
{
   "symbols":[
      {
         "name":"SymbolExample1",
         "definition":"definition example for symbol 1 example",
         "unit":"Kg",
         "is_indexed":"false",
         "indexes":[],
         "modules":{ "main":"IAM Number One", "secondary":[]},
         "category":"CategoryExampleTopLevel",
         "ProjectTypeOfValue":"Constant",
         "ProgrammingSymbolType":"Constant"
      },{
         "name":"SymbolExample2",
         "definition":"definition example for symbol 2 example",
         "unit":"N",
         "is_indexed":"true",
         "indexes":[ IndexExample1","IndexExample2"],
         "modules":{ "main":"IAM Number One", "secondary":["
             Testing_modules"]},
         "category":"CategoryExampleBottomLevel",
         "ProjectTypeOfValue":"Scenario_Parameter",
         "ProgrammingSymbolType":"Lookup"
      }
    ],
   "modules":[
      "IAM Number One",
      "Testing_modules"
    ],
   "indexes":[
      {
         "name":"IndexExample1",
         "definition":"definition example 1",
         "values":["ValueExample1"]
      },
    ],
   "categories":[
      {
         "name":"CategoryExampleTopLevel",
         "super_category":"null"

      }]
}
```

Fragmento de código 8.9: Ejemplo de la respuesta que da el servicio para la obtención de la tabla de símbolos remota.

8.7. Emparejamiento de índices

La regla `DictionaryIndexMismatchCheck` debe de comprobar si los índices de un símbolo concuerdan con los almacenados en la base de datos que soporta el diccionario de datos compartido. Pero no vale con una simple comparación, por varios motivos:

1. El diccionario de datos no garantiza el orden de los índices, por lo que el fichero .mdl puede contener `foo[A,B,C]` y que el servicio devuelva `[C,A,B]`
2. Se considera válido que un símbolo tenga como índices un subconjunto de los índices que indica la base de datos. Ya que puede que en algunos módulos haya índices que no son necesarios.
3. El servicio REST devuelve subscripts, pero en el fichero los índices pueden ser subscripts o valores de subscripts. Si un símbolo tiene un solo índice, este puede ser un subscript o un valor. Si es un valor entonces el símbolo tendrá una ecuación por valor del subscript. Si es un subscript solo habrá una ecuación, que se aplicará a todos sus valores. Si el símbolo tiene más índices, puede haber combinaciones, algunos índices pueden ser subscripts y otros valores. El problema es que no se puede inferir el subscript al que pertece un valor.
4. Hay cierta ambigüedad a la hora de tratar con copias. Si existe un subscript llamado `ESCENARIOS_I` y su copia `ESCENARIOS_COPY_I`, y el diccionario de datos dice que un símbolo está indexado por el original, pero en el fichero está indexado por la copia, ¿debería de lanzar una issue?

El punto 3 es muy importante, ya que un valor puede estar asociado a varios subscripts, y en muchos casos no se puede diferenciar a cuál pertenece:

- Un subscript y su copia comparten los mismos valores, y Vensim los trata como si fueran el mismo.
- Es posible definir un subscript con los mismos valores que otro sin usar la sintaxis de copia, lo que dificulta detectar si un subscript es una copia de otro.
- Se pueden crear subscripts que tengan un subconjunto de valores de otro subscript. Por lo que en ciertos casos no se puede diferenciar a qué subscript pertenece un valor.
- No puede haber subscripts que tengan un subconjunto de valores de otro subscript y que además tengan valores únicos. Con una excepción: si se crean mediante secuencias. En el caso de tener `firstSubscript: (value1-value50)` y `secondSubscript: (value25-value75)` ambos subscripts compartirían los valores entre `value25` y `value50`, pero no el resto.

Para resolver todos estos problemas se decidió que la regla no compararía directamente el subscript, sino sus valores. Es decir comprueba si los valores que se encuentran en el

símbolo del fichero concuerdan con los valores de los subscripts que dice el diccionario de datos compartido.

Para ver cómo se hace esta comprobación, se empezará explicando qué se almacena en cada tabla de símbolos. Ambas tablas están implementadas con la clase `SymbolTable`, y almacenan los índices de un símbolo en el atributo `indexes` de `Symbol`. Este atributo es una lista de listas de símbolos. Sin embargo hay una pequeña diferencia en cómo se usa este atributo en ambas tablas.

En la tabla de símbolos local cada lista dentro de `indexes` representa una columna de índices. La primera columna serán todos los índices que se encuentran en la primera posición de un símbolo. Se puede ver un ejemplo en el Fragmento de código 8.10

En este ejemplo la primera columna hace referencia a un subscript. En una columna solo se puede o hacer referencia a un subscript completo, o hacer referencia a varios valores de un solo subscript. Por ello en este caso la primera columna siempre contiene el mismo símbolo.

```
---- .mdl content ----
var[SUBSCRIPT_I, SUBSCRIPT_VALUE_1] = 3
var[SUBSCRIPT_I, SUBSCRIPT_VALUE_2] = 5
var[SUBSCRIPT_I, SUBSCRIPT_VALUE_3] = 9
---- Attribute 'indexes' of the symbol 'var' ----
List[List[SUBSCRIPT_I, SUBSCRIPT_I, SUBSCRIPT_I],
     List[SUBSCRIPT_VALUE_1, SUBSCRIPT_VALUE_2, SUBSCRIPT_VALUE_3]]
```

Fragmento de código 8.10: Ejemplo del contenido del atributo `indexes` de `Symbol`

Al almacenarlo por columnas en vez de por filas se mantienen juntos todos los valores de un mismo subscript. Lo que favorece el procesamiento posterior.

Si el fichero no es correcto, podría ocurrir que una ecuación tuviera más índices que el resto. En el proyecto se asume que todos los ficheros son correctos, por lo que podría haberse ignorado o lanzar un error. Pero en este caso se decidió permitirlo y crear una columna extra, que tendría menos valores que el resto.

La idea era crear una clase encargada de validar los índices. Esta comprobaría que cada columna tiene el mismo número de valores, y asegurándose de que cada columna tiene valores válidos. Por falta de tiempo no se llegó a hacer, sin embargo la propia regla sí realiza algunas validaciones, con el objetivo de evitar algunos casos problemáticos.

Volviendo a la estructura, en la tabla de símbolos remota los índices son iguales, pero con una diferencia: como todo son subscripts cada columna solo tiene un valor.

8.7.1. El algoritmo de emparejamiento

Con las dos estructuras preparadas es necesario un algoritmo capaz de verificar si se cumplen las condiciones mencionadas al principio de esta sección. Como no se puede garantizar el orden de los subscripts que devuelve el servicio REST que consulta el diccionario de datos compartido, se implementó un algoritmo de *backtracking* mediante recursión. El número de índices no debería superar los 10, así que la eficiencia no es un problema.

El algoritmo trata de emparejar todos los subcripts del símbolo del diccionario de datos con los índices del símbolo local. Si al completarse el algoritmo, todos los índices se han asociado, entonces los índices son los esperados y no se genera issue. Pero si sobra alguno, ya sea porque no se ha podido emparejar o porque hay más índices en local que en el diccionario de datos, lanzará una issue indicando los índices no esperados. Se puede ver en el Fragmento de código 8.11.

```
private boolean tryToMatchIndexes(List<List<Symbol>> foundIndexes,
    List<List<Symbol>> dbIndexes) {
    for(List<Symbol> index:foundIndexes){

        for(List<Symbol> dbIndex:dbIndexes){
            if(matchIndex(index,dbIndex.get(0))){
                ArrayList<List<Symbol>> newFound = new ArrayList<>(
                    foundIndexes);

                newFound.remove(index);
                ArrayList<List<Symbol>> newDb = new ArrayList<>(
                    dbIndexes);
                newDb.remove(dbIndex);

                if(tryToMatchIndexes(newFound,newDb))
                    return true;
            }
        }
    }
    return false;
}
```

Fragmento de código 8.11: Algoritmo de backtracking para el emparejamiento de índices

Para comparar un subscript del diccionario de datos con el índice de un símbolo primero se comprueba si dicho índice es un subscript. Si lo es, se comparan directamente sus valores. Si no lo es, se comprueba si los valores del índice son un subconjunto de los valores del subscript.

La comparación que se realiza es por tokens y no por un `.equals` por las diferencias entre los símbolos locales y los remotos. Como ya se ha explicado anteriormente almacenan

los índices de forma diferente, además los locales incluyen la línea y un módulo, mientras que los remotos pueden indicar varios módulos y no tienen línea. Esta comparación puede observarse en el Fragmento de código 8.12.

```
private boolean matchIndex(List<Symbol> foundIndex, Symbol dbIndex){
    Symbol firstFoundValue = foundIndex.get(0);
    Set<Symbol> uniqueFoundIndexes = new HashSet<>(foundIndex);

    // Si el índice es un subscript, se compara los valores con el
        subscript de la base de datos
    if(firstFoundValue.getType()== SymbolType.Subscript){
        if(uniqueFoundIndexes.size()==1) {
            Set<String> dbValues = dbIndex.getDependencies().stream
                ().map(Symbol::getToken).map(String::trim).collect(
                Collectors.toSet());
            Set<String> foundValues = firstFoundValue.
                getDependencies().stream().map(Symbol::getToken).map(
                String::trim).collect(Collectors.toSet());
            return dbValues.containsAll(foundValues);
        }

        throw new IllegalStateException("A symbol can have either a
            subscript or several subscript values, not both");
    }

    // Si el índice son varios valores, se comparan con los valores
        del subscript de la base de datos
    List<String> dbIndexTokens = dbIndex.getDependencies().stream().
        map(Symbol::getToken).map(String::trim).collect(Collectors.
        toList());
    for(Symbol index:foundIndex){
        if(!dbIndexTokens.contains(index.getToken().trim())) {
            return false;
        }
    }
    return true;
}
```

Fragmento de código 8.12: Método de comparación entre un índice local y un índice de la base de datos

8.8. Inyección de símbolos

Otro de los requisitos del proyecto pedía que la información obtenida sobre los símbolos del archivo .mdl se inyectase programáticamente desde el plugin al diccionario de datos, para

automatizar lo máximo posible la población del diccionario de datos compartido. Pero solo se deben enviar los símbolos válidos, es decir aquellos que no incumplan las reglas de nombrado.

Para saber si un símbolo es válido los símbolos tienen un atributo `isValid`, que se inicializa a true. Si una regla genera una issue con un símbolo, lo marcará como no válido llamando al método `symbol.setAsInvalid()`.

Los símbolos se envían mediante el mismo servicio de obtención de la tabla de símbolos remota. Por ello comparten una estructura similar. El Fragmento de código 8.13 muestra el formato del input, y el Fragmento de código 8.14 el formato del output. El output muestra los símbolos que no se han podido inyectar y el por qué ha fallado la inyección.

Cuando se ha acabado de comprobar todas las reglas en un fichero, se llama al método `injectNewSymbols` de `ServiceController`. Este método filtra todos los símbolos que hay que inyectar, eliminando los no válidos, los que ya están en el diccionario de datos y los símbolos por defecto de Vensim. Además, esta función se encarga de loggear los errores que haya, pero delega el envío en `DBFacade.injectSymbols`.

`InjectSymbols` se encarga de transformar los símbolos de la clase `Symbol` a formato `JSON`, siguiendo el estándar que se estableció. Y después ejecuta`ServiceConnectionHandler`, que se encarga de hacer la petición en sí.

```
{
    "module":"",
    "symbols":[
        {
            "name":"symbol_1",
            "definition":"Example definition",
            "unit":"Dml",
            "is_indexed":"false",
            "programmingsymboltype":"variable"
        },
        {
            "name":"symbol_n",
            "definition":"Another example definition",
            "unit":"Kg",
            "is_indexed":"true",
            "programmingsymboltype":"variable_subscripted"
        }
    ],
    "indexes":[
        {
            "indexName":"INDEX_EXAMPLE1_I",
            "definition":"Index definition",
            "values":[ "VALUE1", "VALUE2"]
        },
        {
            "indexName":"INDEX_EXAMPLE2_I",
            "definition":"Index definition number two",
            "values":["CO2","CO"
            ]
        }
    ]
}
```

Fragmento de código 8.13: Ejemplo del formato a seguir para inyectar los símbolos en el servicio del diccionario.

```
{
   "symbols":[
      {
         "name":"symbol_1",
         "message":"There is a symbol with this name in the data
             dictionary."
      },
      {
         "name":"symbol_n",
         "message":"Other reason ... "
      }
   ],
   "indexes":[
      {
         "indexName":"INDEX_1",
         "message":"There is an index with this name in the data
             dictionary."
      },
      {
         "indexName":"INDEX_N",
         "message":"Other reason ... "
      }
   ]
}
```

Fragmento de código 8.14: Ejemplo del output que devuelve el servicio al inyectar los símbolos de un módulo.

8.9. Estructura general del código

Todo el código está dividido en cinco paquetes:

- **parser**: Contiene el código generado por ANTLR, los visitors y las clases que forman la tabla de símbolos.
- **plugin**: Contiene las clases básicas de un plugin de Sonarqube.
- **rules**: Contiene todas las reglas del plugin.
- **service**: Contiene las clases relacionadas con el acceso al diccionario de datos compartido.
- **utilities**: Contiene clases de utilidad, como una clase con constantes y funciones comunes, el generador del fichero de output intermedio, y más.

Las Figuras desde la 8.4 hasta la 8.9 muestran el diagrama de paquetes y los diagramas de clases de los diferentes paquetes.

La Figura 8.10 muestra la secuencia que se realiza cuando se ejecuta el escáner y se llama al método `execute` de `VensimSquidSensor`. En esta se presupone que los objetos de las reglas se crean antes del inicio de esta secuencia. La instanciación de esta clase y de otras clases de configuración es interna, por lo que se desconoce su uso y no se han representado en el diagrama.

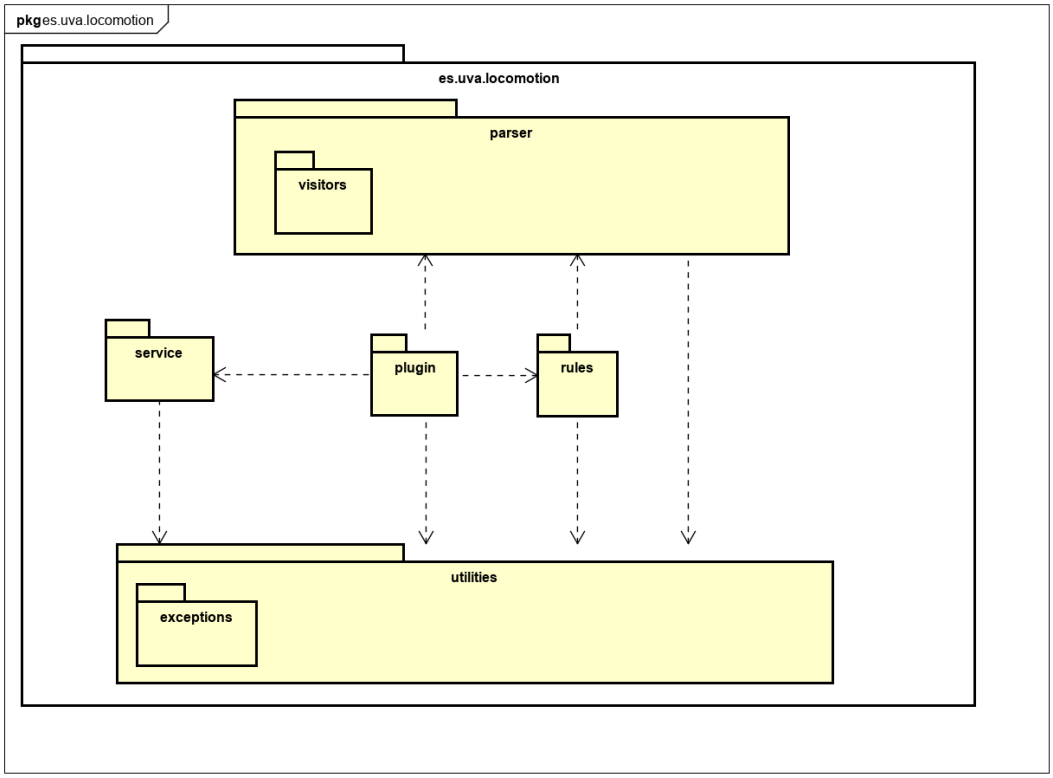

Figura 8.4: Diagrama de paquetes

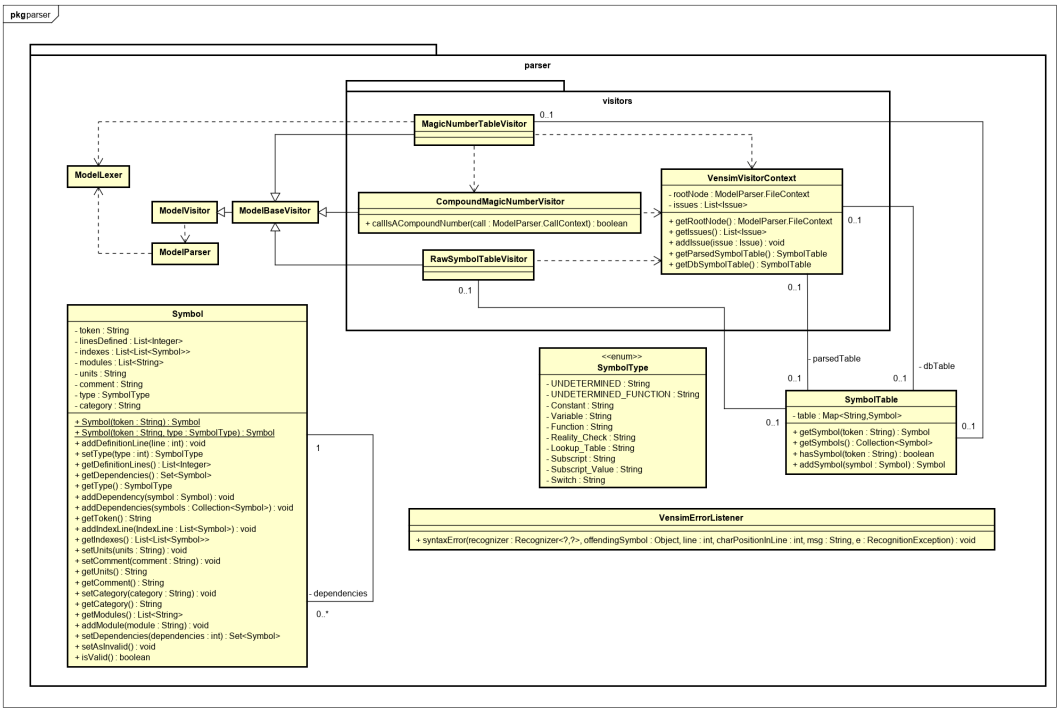

Figura 8.5: Diagrama de clases del paquete `es.uva.locomotion.parser`

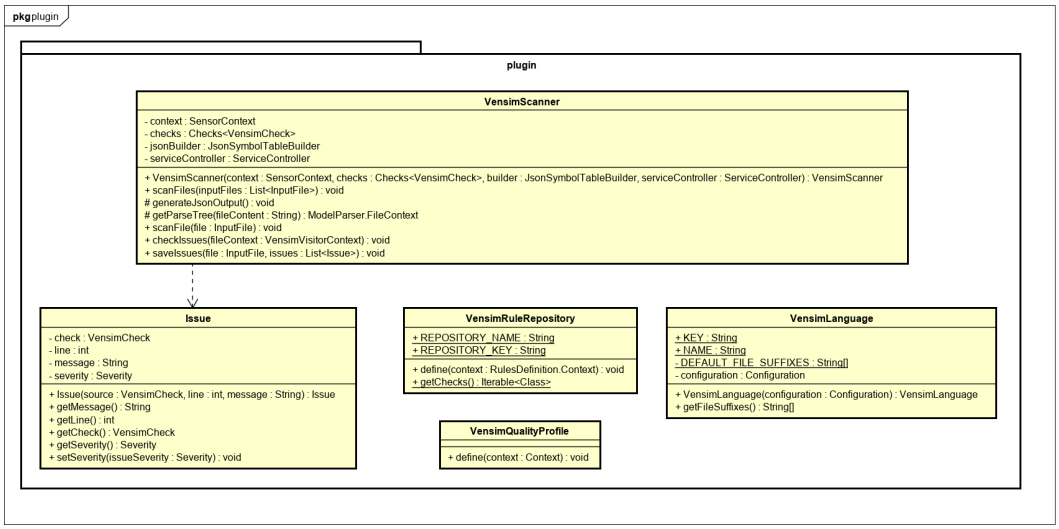

Figura 8.6: Diagrama de clases del paquete `es.uva.locomotion.plugin`

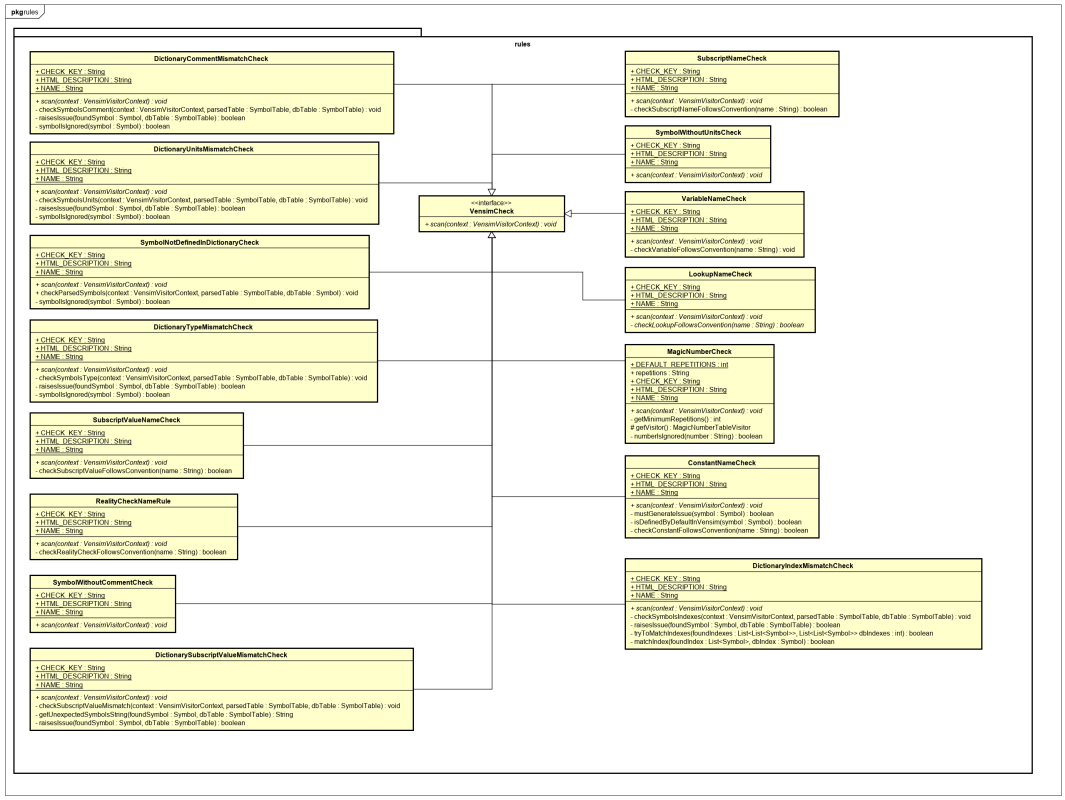

Figura 8.7: Diagrama de clases del paquete `es.uva.locomotion.rules`

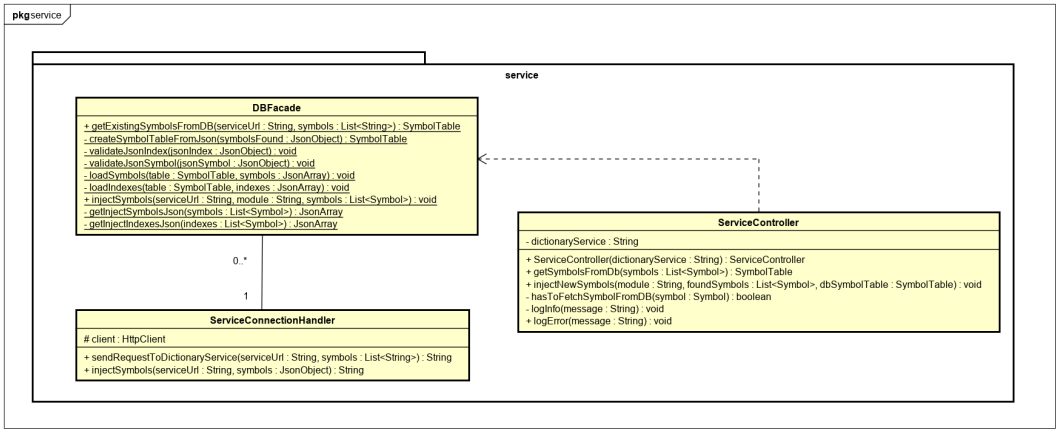

Figura 8.8: Diagrama de clases del paquete `es.uva.locomotion.service`

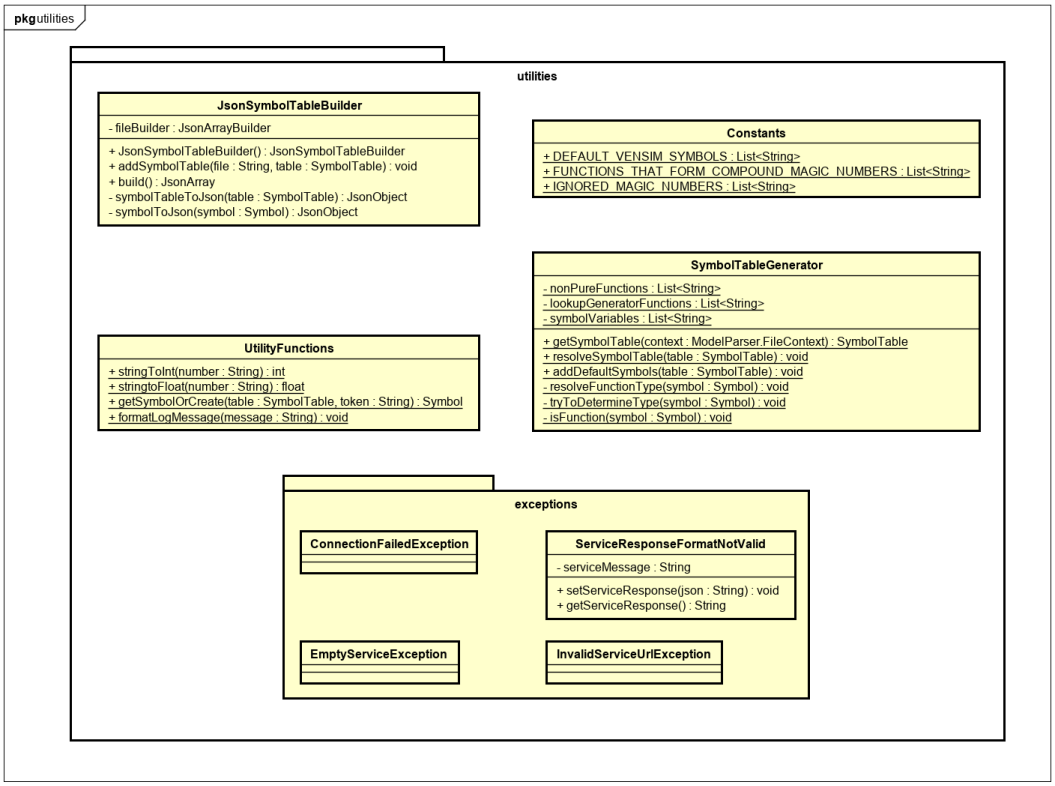

Figura 8.9: Diagrama de clases del paquete `es.uva.locomotion.utilities`

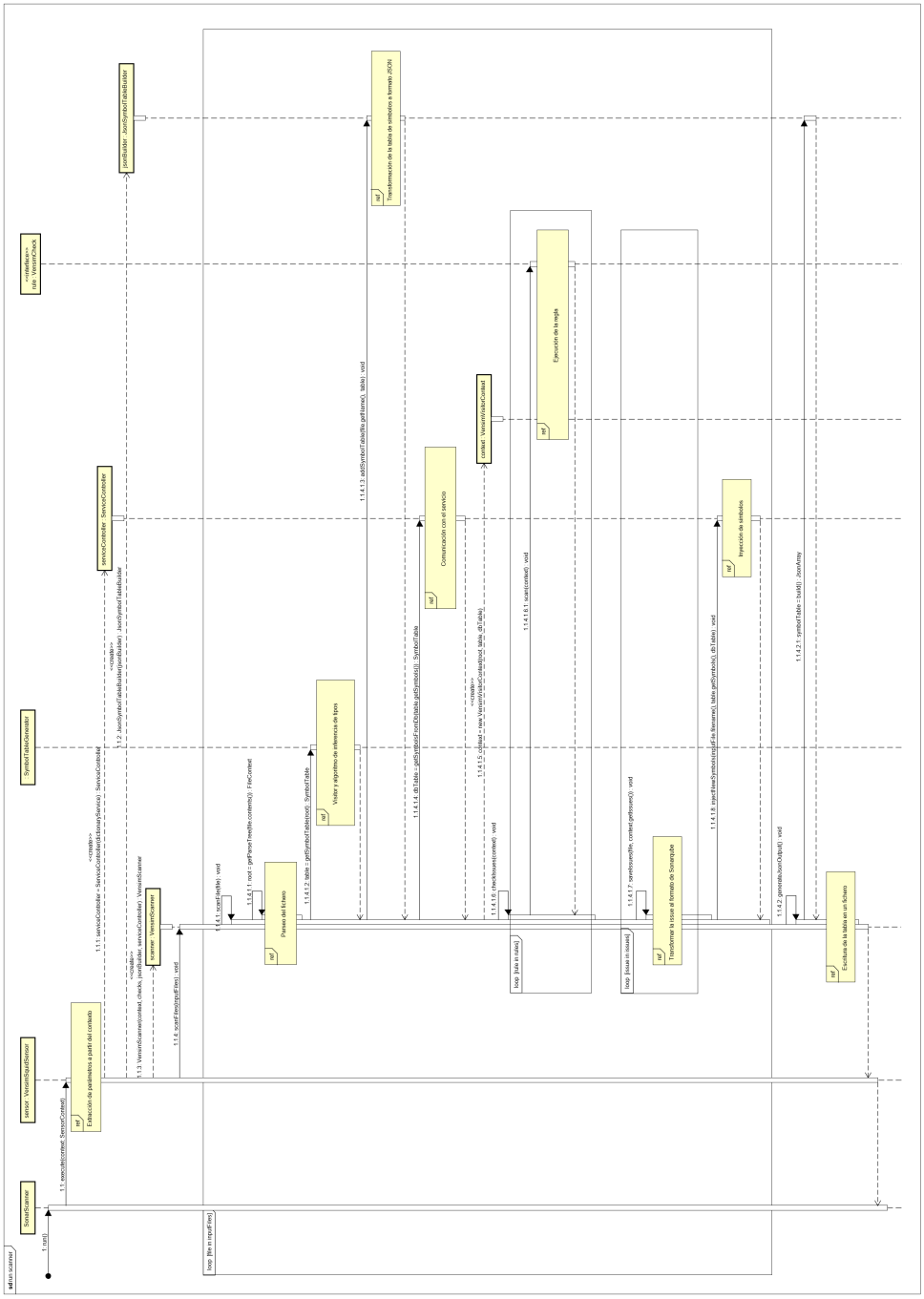

Figura 8.10: Diagrama de la secuencia que se desencadena cuando se ejecuta un análisis usando `sonar-scanner`

Capítulo 9

Testing

Para la automatización de los tests se usó JUnit y Mockito. Los tests se podrían subdividir en:

- Tests unitarios de visitors y otras clases de utilidad
- Tests de gramática
- Tests de reglas
- Tests de acceso al servicio del diccionario de datos
- Tests de integración con Sonarqube

Los tests de los visitors y otras clases de utilidad son tests unitarios que no tienen demasiado interés. Solo merece la pena resaltar que para evitar duplicación de código en tareas que se repetían mucho, como puede ser la obtención de una tabla de símbolos a partir de una cadena, se crearon muchos métodos de utilidad. Estos se encuentran en el paquete `testutilities`.

9.1. Tests de gramática

El árbol generado por una gramática es muy sensible a cambios. De forma que un pequeño cambio en la gramática, como añadir un nodo intermedio, puede provocar que todos los tests fallen. Además, no existen librerías para testear ANTLR que faciliten el proceso.

Por ello se ha preferido dar mayor prioridad a los tests de las reglas, que testean la gramática de forma indirecta. No comprueban la forma del árbol, pero comprueban que el output generado tras recorrer dicho árbol es válido.

Aún así, a lo largo del desarrollo se encontraron algunos bugs relacionados con la generación del árbol, y tras corregirlos se decidió que merecía la pena mantener algunos tests para actúen como tests de regresión. Estos se encuentran en el fichero `TestGrammar.java`.

Además, se incluyeron tres tests que parsean tres de los deliverables de MEDEAS con el objetivo de comprobar que no ocurre ninguna excepción. En estos casos sería imposible comprobar el árbol entero, por lo que simplemente se intenta asegurar que el parser es capaz de generar el árbol a partir de los ficheros.

9.2. Tests de reglas

Al crear los primeros tests de las reglas se decidió que estos tests se acercarían más a tests de integración que a tests unitarios. En vez de comprobar únicamente la clase de la regla, cada test analizaría un pequeño fichero, de manera similar a la que se ejecuta el plugin sobre un fichero.

Esto se debe a dos razones: por un lado la dificultad de testear árboles, mencionada en la sección anterior; y por otro lado para comprobar que se genera la issue correctamente a partir de un fichero.

Según los ficheros fueron aumentando, se movieron adentro de los propios tests. Normalmente suelen estar compuestos por una o dos líneas, así que no dio problemas. Además esto permite leer el contenido que se parsea dentro del test, lo cual permite entender el test más fácilmente.

La ejecución de cada test es muy similar a lo que ocurre al realizar un análisis con `sonar-scanner`: Se crea un objeto `VensimScanner`, solo que de forma manual y con un contexto de tipo `SensorContextTester`, que forma parte de la librería de Sonar. Una vez obtenido el escáner, se genera un `VensimVisitorContext` y se llama al método `checkIssues` de `VensimScanner`.

Este método se encarga de analizar la cadena como si fuera un fichero Vensim. Ejecutando todas las reglas que existen.

Realizar todos estos pasos en cada test haría que hubiera mucha duplicación, y que el objetivo del test acabara oculto entre tanto código. Por ello se crearon muchos métodos para facilitar el testing, tanto para generar árboles como diversos asertos.

En el fragmento de código 9.1, se puede ver un ejemplo de cómo es un test que comprueba una regla. En este se aprecia que estos métodos de utilidad permiten que los tests sean mucho más expresivos. En este ejemplo, además, se ha añadido un salto de línea al principio de la cadena para verificar que la issue se genera en la línea correcta.

Como este método ejecuta todas las reglas, podrían aparecer issues que no son relevantes para el test. Por ello en el aserto se indica el tipo de la issue.

A diferencia del resto, los tests de números mágicos están divididos en dos: Por un lado

```
@Test
public void testSeveralUnderscore(){
    String program = "\nexpected__consumption_2020 = Time ~~|";

    VensimVisitorContext visitorContext =
        getVisitorContextFromString(program);
    VensimScanner scanner = getScanner();

    scanner.checkIssues(visitorContext);
    assertHasIssueInLines(visitorContext, VariableNameCheck.class, 2);
}
```

Fragmento de código 9.1: Ejemplo de un test de reglas

`TestMagicNumberCheck`, que comprueba solo la regla, y por otro lado hay tests similares al resto de reglas, en la clase `TestIntegrationMagicNumberCheck`. Esta división es necesaria porque en esta regla hay algunos tests que solo se pueden hacer de forma unitaria, debido a que no se puede configurar el parámetro de la regla si se testea de forma global.

Los tests que comparan el estado del fichero .mdl con el del contenido del diccionario de datos también son unitarios, ya que no comprueban casos de parseo concretos sino que prueban la comparación entre dos tablas de símbolos.

9.3. Tests de acceso al servicio del diccionario

Todos los tests de las clases que se comunican con la API REST se realizan mediante mocks. No se realizaron tests de integración debido a que la API no existía en el momento del desarrollo. Para escribir y testear estas clases se usó la interfaz que se acordó en una reunión con la persona encargada de desarrollar la API.

Las tres clases que se encargan de la comunicación están escritas de forma incremental y secuencial. `ServiceController` realiza un filtrado de la información que se va a pedir y delega en `DBFacade`, que a su vez traduce la información al formato adecuado y delega el envío de la petición en `ServiceConnectionHandler`.

Por tanto para testear estas clases se realiza un mock de la clase a la que llaman, para testear únicamente las responsabilidades de una clase. En el caso de `ServiceConnectionHandler` se hace mocks de la clase de Java `HttpClient`, que es la que envía la petición.

9.4. Tests de integración con Sonarqube

Para testear que el plugin en conjunto funciona correctamente se crearon una serie de tests de integración. Estos tests no ejecutan código del plugin de forma directa, sino que llaman a `sonar-scanner` para que analice unos ficheros, y después usan la API REST de Sonarqube para comprobar si se han generado los resultados esperados.

Debido a la complejidad de los tests estos solamente comprueban un caso en el que se genera issue y uno o varios en los que no se genera. Su objetivo es ver que la regla está correctamente configurada en Sonarqube y que es capaz de generar issues.

Los ficheros que se analizan están en la carpeta `integrationTests`. Para evitar poner la configuración del `sonar-scanner` dentro de los propios tests se usó un fichero de configuración `sonar-project.properties` dentro de la carpeta que contiene los ficheros de prueba. A su vez para poder identificarse en el servidor debe de existir una variable de sistema llamada `SONAR_TOKEN` que contenga el token de autenticación.

Como el código para llamar al escáner y para llamar a la API de Sonarqube es bastante largo se movió a métodos dentro de una clase de utilidad. Además, se crearon asertos que permiten que los tests sean transparentes al formato en el que Sonarqube devuelve las issues.

Los tests de integración también comprueban las reglas que interactúan con el diccionario de datos. Para ello se usa Dyson [20] para hacer un mock de la API REST. Los mocks se encuentran en `mocksServicios\consultaSimbolos.js`

Comprobar la inyección de símbolos no es tan fácil como las reglas. Para comprobar una regla le pides a Sonarqube que te devuelva lo que ha procesado, y compruebas si es lo que esperabas. Pero en el caso de la inyección de símbolos no se puede preguntar a la API si ha recibido lo que se esperaba.

Se discutieron varias soluciones para este problema. Para evitar complicar mucho las cosas se decidió que cuando el servidor recibiera la petición esperada, este crearía un fichero que actuaría como flag. Después el test comprobaría si el fichero existe. Para evitar problemas, el fichero se borra antes de ejecutar los tests.

9.4.1. Tests del output intermedio

Además de probar la integración con Sonarqube, estos tests comprueban que el output intermedio se ha generado correctamente. Antes de empezar los tests el fichero se borra, y acabar el análisis se comprueba que el fichero se ha vuelto a crear, y que contiene los datos esperados.

Para testear que se genera de forma correcta se añadió un fichero de prueba pequeño pero que cubre la mayoría de casos.

9.5. Cobertura alcanzada

La cobertura obtenida por paquetes se puede ver en la Tabla 9.1.

Paquete	% Clase	% Método	% Línea
Total	95,9 % (93/97)	68,4 % (453/662)	80,2 % (2319/2892)
es.uva.locomotion	0 % (0/2)	0 % (0/5)	0 % (0/17)
es.uva.locomotion.parser	100 % (60/60)	59 % (263/446)	73,7 % (1301/1765)
es.uva.locomotion.parser-visitors	100 % (4/4)	87,5 % (70/80)	88,7 % (258/291)
es.uva.locomotion.plugin	60 % (3/5)	68,2 % (15/22)	59,5 % (78/131)
es.uva.locomotion.rules	100 % (15/15)	100 % (61/61)	100 % (310/310)
es.uva.locomotion.service	100 % (3/3)	95,2 % (20/21)	99,6 % (233/234)
es.uva.locomotion.utilities	100 % (4/4)	85 % (17/20)	96,2 % (125/130)
es.uva.locomotion.utilities.exceptions	100 % (4/4)	100 % (7/7)	100 % (14/14)

Tabla 9.1: Cobertura obtenida en los tests

Se observa una clara diferencia entre paquetes. Algunos alcanzan una cobertura alta mientras que en otros el resultado es mucho más bajo. Esto se debe a dos factores que reducen la cobertura total del programa:

- Las clases de configuración de Sonarqube no se pueden testear o solo de forma parcial. Por ello, el paquete raíz y el paquete `plugin` tienen una cobertura muy baja.
- La tabla incluye la cobertura de las clases generadas por ANTLR4, cuyos métodos se sobrescriben o no se usan durante el parseo de los tests. De los 183 métodos sin testear del paquete `parser`, 181 pertenecen a clases autogeneradas por ANTLR.

Capítulo 10

Conclusiones

Tras completar el proyecto se ha logrado realizar la mayoría de objetivos iniciales. El único que no se logró cumplir fue la interfaz gráfica, que se remplazó por nuevos requisitos que surgieron a lo largo del proyecto. El proyecto no ha sido fácil, y ha tenido dos problemas principales: el tiempo, y los cambios en los requisitos.

No se ha podido dedicar tanto tiempo como se esperaba al proyecto, ya que en todo momento se ha estado realizando en paralelo con otra carga de trabajo. Durante el primer cuatrimestre se realizó a la vez que otras cinco asignaturas y durante el segundo cuatrimestre a la vez que el alumno realizaba las prácticas en empresa a jornada completa y completaba una asignatura. Las prácticas se realizaron en Madrid, por lo que el alumno tuvo que vivir de manera independiente. Tras el inicio del confinamiento trabajó en remoto pero no regresó a Valladolid hasta el fin del estado de alarma. Todo esto provocó que las condiciones de trabajo fueran poco óptimas y ralentizaran el progreso.

El otro problema fueron los requisitos. Al principio del proyecto los requisitos eran demasiado genéricos. Se quería un estándar de nombrado, pero no se sabía exactamente cuál. Se quería detectar los números mágicos, pero no se había establecido una definición de números mágicos. Establecer estos requisitos no era sencillo, ya que había que acordarlo con el GEEDS. Para seguir la planificación en varios casos se tuvo que concretar los requisitos y empezar a trabajar en ellos sin tener todavía el visto bueno de los clientes. En el acceso al diccionario se tuvo que programar sin conocer la interfaz que tendría del servicio, que hubo que cambiar un mes más tarde cuando se estableció el estándar.

Adicionalmente, en muchos casos los estándares no eran fijos e iban evolucionando. A medida que avanzaba el proyecto los clientes cambiaban de opinión o añadían nuevos requisitos. La interfaz del servicio al que se conecta el plugin también cambió. Todo esto provocó que no se pudiera dedicar tanto tiempo a requisitos nuevos, porque había que modificar y mantener los antiguos.

Para poder reducir estos problemas, se diseñó y programó tratando de reducir el acoplamiento. Lo cual evitó que fuera necesario hacer modificaciones a la estructura del código. El

uso de una metodología ágil también ayudó a lidiar con los problemas que fueron surgiendo. Aún así, no se pudo evitar que los cambios ralentizaran el progreso del proyecto.

También se debería de haber dado más prioridad a la memoria. Es cierto que debido a la duración de la beca había que acelerar la programación. Pero los requisitos siguieron aumentando y por tanto la beca se extendió. Habría sido mejor recortar requisitos y favorecer un poco más la memoria, ya que los sprints finales de documentación se hicieron muy duros.

Por otro lado, el trabajo realizado durante el sprint 0 fue muy útil. Permitió un comienzo de proyecto mucho más rápido y fluido. El tiempo que se ahorró vino muy bien para poder crear la gramática, ya que al principio se asumió que se obtendría una gramática oficial. En cambio, tener que mejorar una gramática requirió un conocimiento más avanzado de Vensim, que fue útil más adelante a la hora de implementar los visitors y crear los estándares.

El uso de ANTLR4 para implementar la gramática fue un gran acierto. La flexibilidad que aporta permitió hacer cambios en la gramática con facilidad. Los visitors permitieron afrontar problemas como la detección de números compuestos sin añadir demasiada complejidad.

10.1. Líneas de trabajo futuras

La base del proyecto está completada, pero se podrían realizar una serie de mejoras y nuevas funcionalidades. Algunas de estas mejoras serían:

- **Crear una interfaz gráfica**: Esta es la historia de usuario que fue descartada. Para los creadores de modelos sería más fácil realizar los análisis mediante una interfaz que actue como wrapper del **sonar-scanner**. Además esta podría filtrar los logs que produce el escáner para mostrar solo la información relevante al usuario.
- **Inyección de símbolos en condiciones no ideales**: En este momento solo se inyectan los símbolos que son completamente válidos. Sin embargo sería posible inyectar solo la parte del símbolo que sea válida. Por ejemplo si un símbolo no tiene comentario se podría inyectar pero dejando el comentario vacío.
- **Permitir acrónimos en los estándares de nombrado**: Se discutió con el GEEDS la posibilidad de que el estándar permita acrónimos en mayúscula incluso cuando el símbolo debería estar en minúscula. Para esto habría que obtener una lista de acrónimos, a definir en el diccionario de datos compartido, y modificar las reglas.
- **Diferenciar constantes de switches**: A lo largo del proyecto se añadió al modelo de datos del diccionario de datos compartido el tipo Switch. El plugin tiene en cuenta los switches para los números mágicos, pero no en la tabla de símbolos. Por tanto, si le llega un switch de la tabla de símbolos remota, siempre lanzará una issue, aunque sea un switch.
- **Inyección de símbolos ya existentes**: La inyección solo se realiza para símbolos nuevos y válidos. Esto hace que para modificar las propiedades de un símbolo se necesario cambiar primero la base de datos y luego el modelo. Se podría programar una opción

o parámetro en el plugin para enviar símbolos en los que han cambiado datos. Esto requeriría un cambio en el backend del servicio para que funcione de forma correcta.

- **Errores en la inyección**: Cuando se creó la interfaz del servicio para inyectar símbolos se dejó en el aire la posibilidad de que el backend devolviera mensajes de error para informar al plugin de si algo ha pasado. Esto no se llegó a realizar, y tendría que hacerse junto con el backend.

- **Mejora de los logs**: El sistema actual de logs es demasiado simple. Todas las excepciones lanzadas durante el análisis las captura el catch del método `scanFile` de `VensimScanner`. Sin embargo, en muchos casos se debería de tratar las excepciones de forma previa para que los errores sean más claros. Se puede mejorar mucho en este aspecto, tanto a nivel de estructura como de los mensajes generados. También se debería de dar la opción de almacenar los logs en un fichero, ya que en muchos casos son muy largos y difíciles de analizar.

- **Parsear las macros y los alcances**: Los símbolos definidos en una macro tienen su propio alcance. En el estado actual del plugin se incluyen como parte del alcance global, lo cual puede generar resultados incorrectos. Se debería de implementar un sistema que permita tener un alcance global y varios subalcances. No es necesario formar un árbol porque no se pueden definir macros dentro de macros. También hay que tener en cuenta que dentro de una macro se puede hacer referencia a un símbolo del alcance global, si acaba con el sufijo `$`.

- **Detección de funciones externas**: Vensim permite programar funciones, que se compilan en un DDL y se importan desde Vensim [64]. El plugin podría tratar de inferir qué funciones son externas y aplicar una regla de nombrado para estas funciones. Localizar el dll y ver las funciones que implementa sería muy difícil. La manera más sencilla sería contrastando las funciones del fichero con una lista de las funciones por defecto de Vensim.

- **Soporte para submodelos**: El proyecto se desarrolló a partir de la versión 8.0.4 de Vensim. Sin embargo en junio del 2020 salió la versión 8.1, que da soporte a submodelos [79]. Lo submodelos pueden compartir parámetros con el padre. Habría que analizar las implicaciones que esto tiene para el plugin. Si se quisiera realizar un análisis entre modelos relacionados habría que detectar cuáles son los submodelos y acceder a las tablas de símbolos de estos durante el análisis del modelo padre.

- **Refactor de la gramática**: La gramática se programó mediante pequeños incrementos de la gramática original. Por ello se pueden mejorar muchas cosas, como la forma en la que gestiona los signos en las expresiones. Si se cambia la gramática también habría que actualizar los visitors de forma acorde.

Apéndice A

Manuales

A.1. Manual de despliegue

En esta sección se detalla lo necesario para desplegar el plugin desarrollado en una instancia de Sonarqube.

A.1.1. Requisitos previos

- Una instancia con Sonarqube 7.9 LTS
- Java 11
- Maven 3.5.4
- Sonar Scanner 4.0.0.1744 o una aplicación que contenga un escáner de Sonarqube embedido.
- ANTLR4 4.7.2

El plugin se desarrolló con estas versiones de las herramientas, pero se pueden usar versiones superiores si son compatibles.

A.1.2. Descarga y compilación

Lo primero es descargar el repositorio, el enlace se encuentra en el anexo B. Una vez descargado hay que compilar el plugin. Para ello hay que ejecutar el comando que se muestra en el fragmento de código A.1 en la carpeta raíz del proyecto. Para evitar que sea necesario tener la instancia de Sonarqube activa se usa la opción `-Dmaven.test.skip=true` para ignorar los tests.

```
mvn -Dmaven.test.skip=true clean package
```

Fragmento de código A.1: Compilación del plugin

A.1.3. Instalación

Tras compilar el plugin se habrá generado un fichero `sonar-vensim-X.jar` dentro de la carpeta `target`. Donde `X` se refiere a la versión actual del plugin. Se va a usar `$SONARQUBE_PATH` para referirse a la carpeta raíz de la instancia. Los pasos para instalarlo son:

1. **Parar la instancia de Sonarqube**. La forma depende del sistema operativo que se use. Para Linux hay que ir a la carpeta `$SONARQUBE_PATH/bin/linux-x86-64/` y ejecutar `sh sonar.sh stop`.
2. **Mover el artefacto generado a** `$SONARQUBE_PATH/extensions/plugins/`
3. **Comprobar que no hay otras versiones del plugin instaladas**. Si existen varios ficheros `sonar-vensim-X.jar` dentro de la carpeta de los plugins es probable que cause problemas.
4. **Encender la instancia de Sonarqube**. Muy parecido a la forma de pararlo, pero cambiando parámetro `stop` por `start`: `sh sonar.sh start`.

Para comprobar que el plugin se ha ejecutado correctamente hay que entrar a la instancia de Sonarqube, pulsar en *Rules* y ver si Vensim aparece en la lista de lenguajes. La Figura A.1 muestra un ejemplo de esta ventana.

A.2. Manual de mantenimiento

En esta sección se detallan algunos aspectos necesarios a tener en cuenta por los desarrolladores que vayan a realizar labores de mantenimiento al software desarrollado en este proyecto.

A.2.1. Aspectos a tener en cuenta al crear una regla

Aunque en el Capítulo 5 se menciona cómo se crea un plugin merece la pena indagar en los pasos para añadir una regla y otros detalles a tener en cuenta.

Para crear una regla:

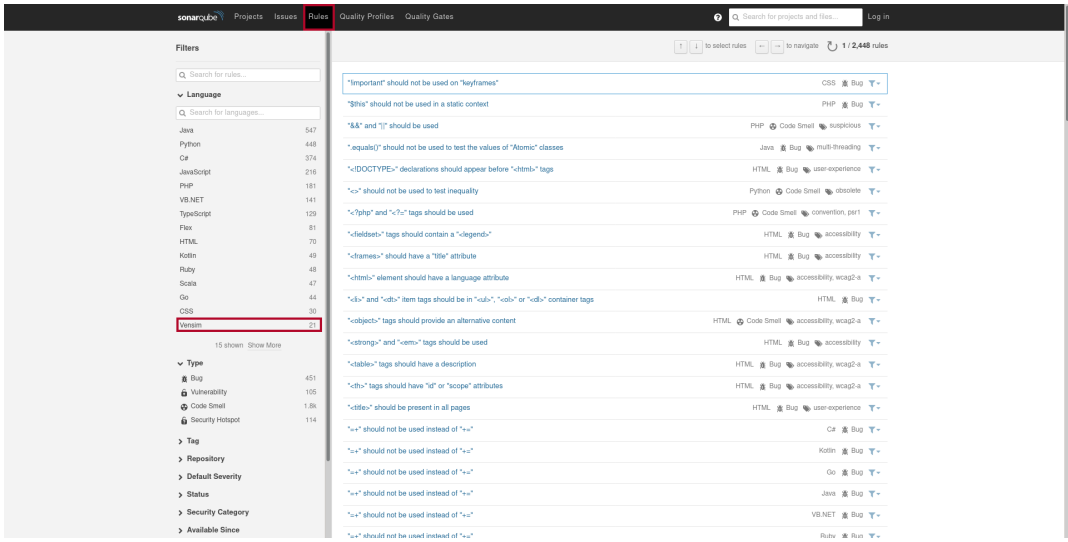

Figura A.1: Ventana de reglas con el plugin funcionando.

1. Crear una clase que implemente la interfaz `VensimCheck`.

    - Todas las reglas se ejecutan mediante el método `execute` de esta interfaz. Por tanto, toda la información que las reglas necesiten debe de estar dentro de la clase `VensimVisitorContext`. Si se necesita más información habría que añadirlo como un nuevo atributo del contexto.

2. Añadir el decorador `@Rule` con el identificador único (key), el nombre y la descripción.

3. Añadir la regla al repositorio (poniéndola en el método `getChecks` de la clase `VensimRuleRepository`), y activar la regla en el quality profile por defecto (en el método `define` de la clase `VensimQualityProfile`).

Algunos puntos a tener en cuenta sobre las reglas y el plugin en general:

- Las reglas no devuelven nada, sino que guardan las issues en el objeto contexto.
- Las reglas no deben depender de otras reglas. Sonarqube no garantiza el orden en el que se ejecutan las reglas, y los usuarios podrían desactivar una regla necesaria para poder ejecutar otra.
- Los símbolos pueden tener varias ecuaciones, que pueden estar en varias líneas. Por ello, si un símbolo no cumple el estándar se genera una issue por ecuación.
- La tabla de símbolos incluye símbolos por defecto de Vensim (`TIME, INITIAL TIME`, etc). Por ello, en muchas reglas se ignoran las funciones y los símbolos por defecto (almacenados en `Constants.DEFAULT_VENSIM_SYMBOLS`)

  - En algunos casos no se ignoran las funciones, porque como no están definidas en ningún sitio la lista de líneas está vacía, y no se genera issue.

- Si la regla necesita la tabla de símbolos que se obtiene de consultar el diccionario de datos a través del servicio REST, se tiene que comprobar si es **null**. En ese caso se ignoraría la regla (poniendo un simple `return`).

- Una tabla de símbolos no puede tener varias veces el mismo símbolo, ni aunque sean instancias diferentes. Esto se debe a que los símbolos se almacenan en un `HashMap` con el token como clave. Además, las reglas están ideadas de forma que solo pueda haber un objeto símbolo por token.

A.2.2. Debuggear la gramatica

Una herramienta útil para ver si se generan los árboles de forma correcta es la opción gráfica del `grun`. Esta herramienta viene instalada con ANTLR4 [27], y permite visualizar el árbol generado a partir de un fichero `.mdl`.

Para ejecutarlo hay que compilar el programa con `mvn clean compile -DskipTests`, ir a la carpeta `./target/classes` y ejecutar el comando que muestra el Fragmento de código A.2. Otra opción interesante es `-tokens`, que muestra una lista con los tokens que ha generado el lexer. Esto puede ser útil en casos en los que el parser forma un árbol mal porque detecta los tokens de forma incorrecta.

```
grun es.uva.locomotion.parser.Model model -gui $PATH_AL_FICHERO/
    ficheroDePrueba.mdl
```

Fragmento de código A.2: Comando para ejecutar la herramienta gráfica `TestRig` de ANTLR4

Una buena fuente de ficheros de prueba son los modelos que se instalan junto con Vensim. En Windows por defecto se encuentra en la carpeta `C:\Users\Public\Vensim\Models`.

A.2.3. Ejecutar los tests de integración

Para poder ejecutar los tests de integración es necesario tener instalado una instancia de Sonarqube en local. Esta deberá de usar el puerto 9000, ya que por defecto los tests hacen llamadas a `http://localhost:9000/`, aunque se podría cambiar el código para que sea un parámetro. También es necesario obtener un token de autenticación (ver la Sección A.3), y guardarlo en la variable de entorno `SONAR_TOKEN`. La forma varía dependiendo del OS. En linux se puede añadir la línea `export SONAR_TOKEN="......."` en el fichero `~/.bash_profile`.

Una vez configurado, hay que arrancar la instancia de Sonarqube. También es necesario ejecutar el servidor de mocks del servicio. Para ello es necesario instalar Dyson [20], ir a la carpeta `./mocksServicio/` y ejecutar `dyson . 9999`.

De momento no se puede ejecutar únicamente los tests de integración, pero para ejecutarlos todos hay que hacer `mvn test` en la carpeta raíz.

A.3. Manual de uso

En esta sección se explica cómo utiliza el plugin creado un usuario final.

A.3.1. Creación y configuración de un proyecto

El primer paso para analizar ficheros es crear y un proyecto en Sonarqube. Para ello hay que ir a la plataforma web, autenticarse, pulsar en el icono de la esquina superior izquierda y dar a *Crear un proyecto nuevo*.

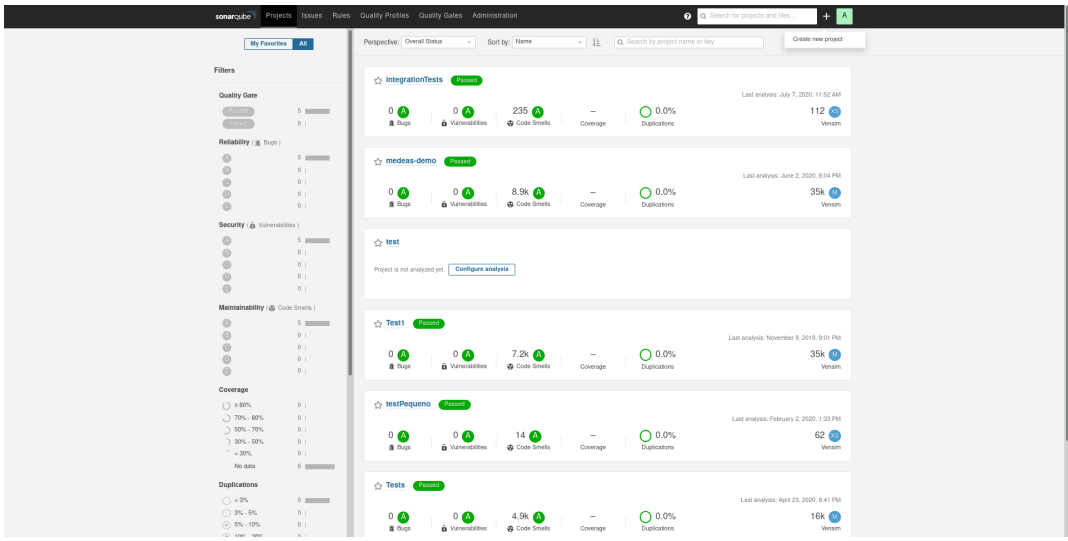

Figura A.2: Ventana que muestra el botón para crear un nuevo proyecto.

Después aparecerá una pantalla en la que hay que poner el identificador y el nombre del proyecto. Tras esto aparecerá una ventana que pregunta si se quiere crear un token de autenticación o usar uno ya existente. Es mejor escoger crear el token aquí, pero como este token es personal se explica en la Sección A.3.3

Al introducir un nombre para el token, este se generará de forma automática. Después la ventana avanza al paso 2, en el que pregunta qué lenguaje se está usando. En este caso hay

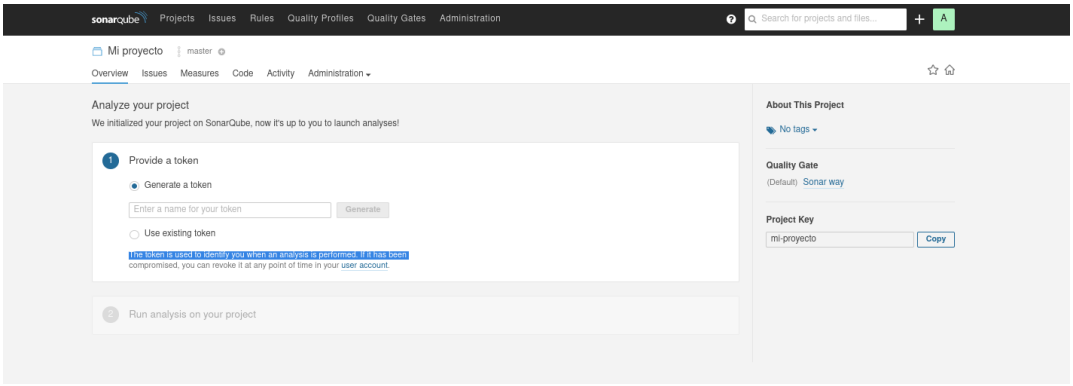

Figura A.3: Ventana de configuración del proyecto 1.

que pulsar en *Other* e indicar el sistema operativo. En ese momento la ventana mostrará un enlace para descargar `sonar-scanner` y el comando que permite ejecutarlo.

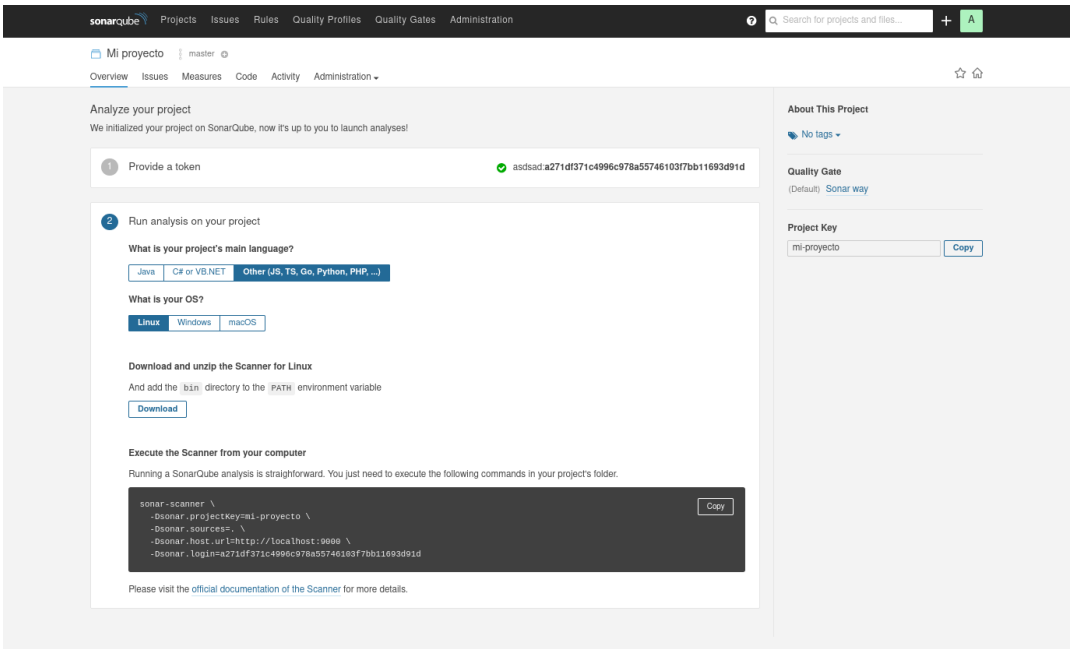

Figura A.4: Ventana que muestra cómo ejecutar un análisis.

El comando que muestra contiene las propiedades básicas del proyecto. Para poder conectar con el servicio será necesario indicar su url en un nuevo parámetro, que se denominó `vensim.dictionaryService`. Se puede ver un ejemplo completo en el Fragmento de código A.3. Además, se pueden añadir o modificar los parámetros de análisis [34] para personalizarlo.

```
sonar-scanner \
  -Dsonar.projectKey=mi-proyecto \
  -Dsonar.sources=. \
  -Dsonar.host.url=http://localhost:9000 \
  -Dsonar.login=a271df371c4996c978a55746103f7bb11693d91d \
  -Dvensim.dictionaryService=http://localhost:9999/
```

Fragmento de código A.3: Código para ejecutar un análisis en linux contra una instancia en local

En lugar de indicar todos los parámetros en cada análisis se puede usar un fichero de configuración, llamado `sonar-project.properties`. Este debe de estar en la carpeta que en la que se va a realizar el análisis. Su contenido está compuesto por un parámetro por línea, separado de su valor por un igual. El fragmento de código A.4 contiene el fichero de configuración equivalente al comando del fragmento A.3. En este ejemplo no se ha incluido el parámetro `sonar.login`, ya que no es una buena práctica almacenar el token en un fichero.

```
sonar.projectKey=mi-proyecto
sonar.sources=.
sonar.host.url=http://localhost:9000
vensim.dictionaryService=http://localhost:9999/
```

Fragmento de código A.4: Contenido del fichero `sonar-project.properties` para un proyecto

Al llamar al escáner todos los parámetros que contenga el fichero se incluirán en el análisis, por lo que no es necesario indicarlos de nuevo.

A.3.2. Ejecución de un análisis y sus resultados

Si ya se tiene `sonar-scanner` instalado, todo lo que hay que hacer es ir a la carpeta en la que están los modelos a analizar y ejecutar el comando `sonar-scanner`, ya sea con un fichero de configuración o indicando los parámetros a mano. Durante la ejecución se mostrará una serie de logs, como los de la figura A.5.

Para ver el resultado del análisis hay que entrar a la plataforma web y pulsar en el proyecto. Esto mostrará un panel con los datos generales del proyecto y una comparación con el análisis anterior. A partir de aquí se puede pulsar en *Issues* para ver una lista con todas las issues, o se puede pulsar directamente en el número de issues que muestra el panel. También se puede ir a la pestaña *Code* para ver los ficheros analizados y ver las issues junto

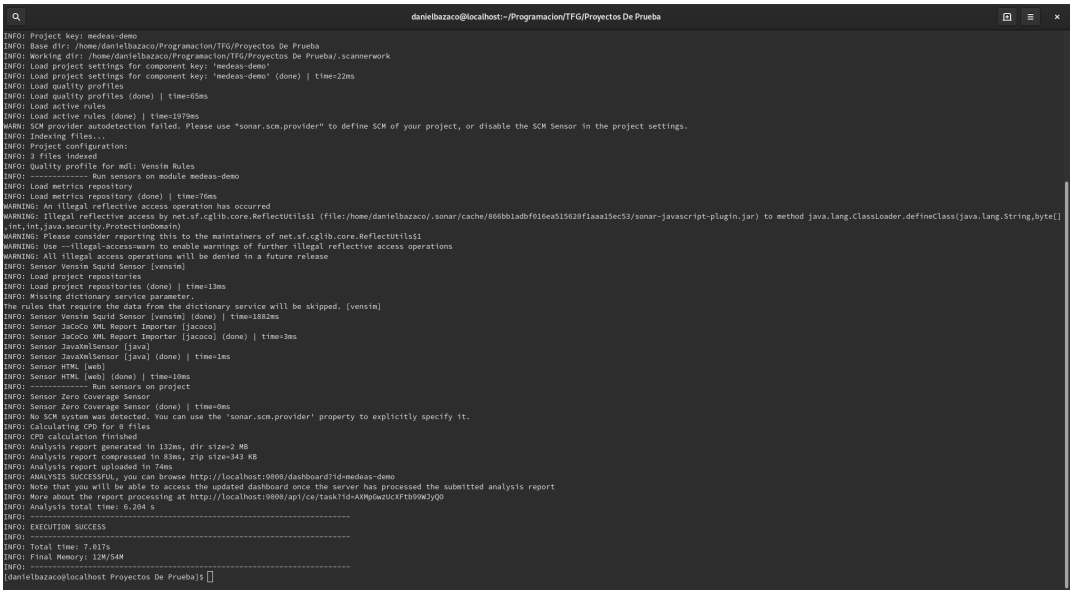

Figura A.5: Logs generados por `sonar-scanner` durante un análisis.

al código.

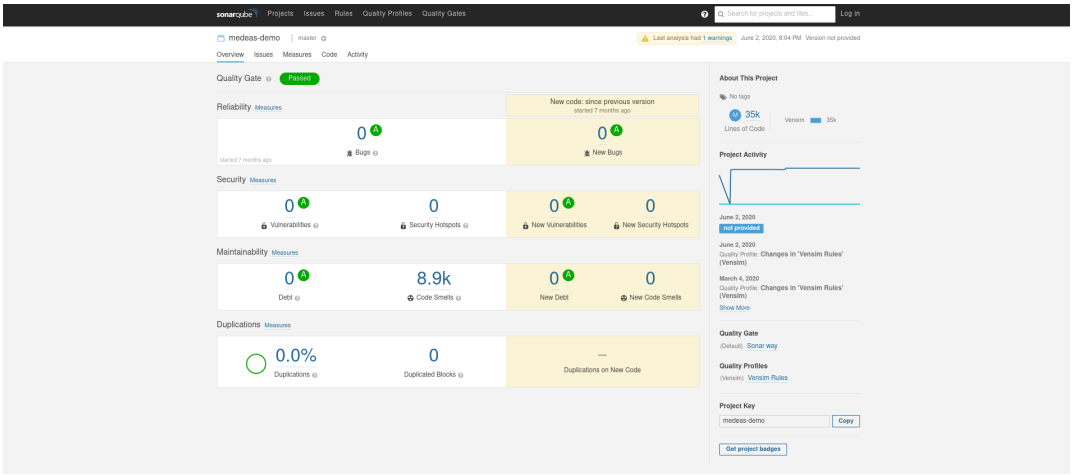

Figura A.6: Ejemplo del panel de un proyecto.

Para ver más en detalle cómo usar Sonarqube se puede consultar la documentación en línea [42].

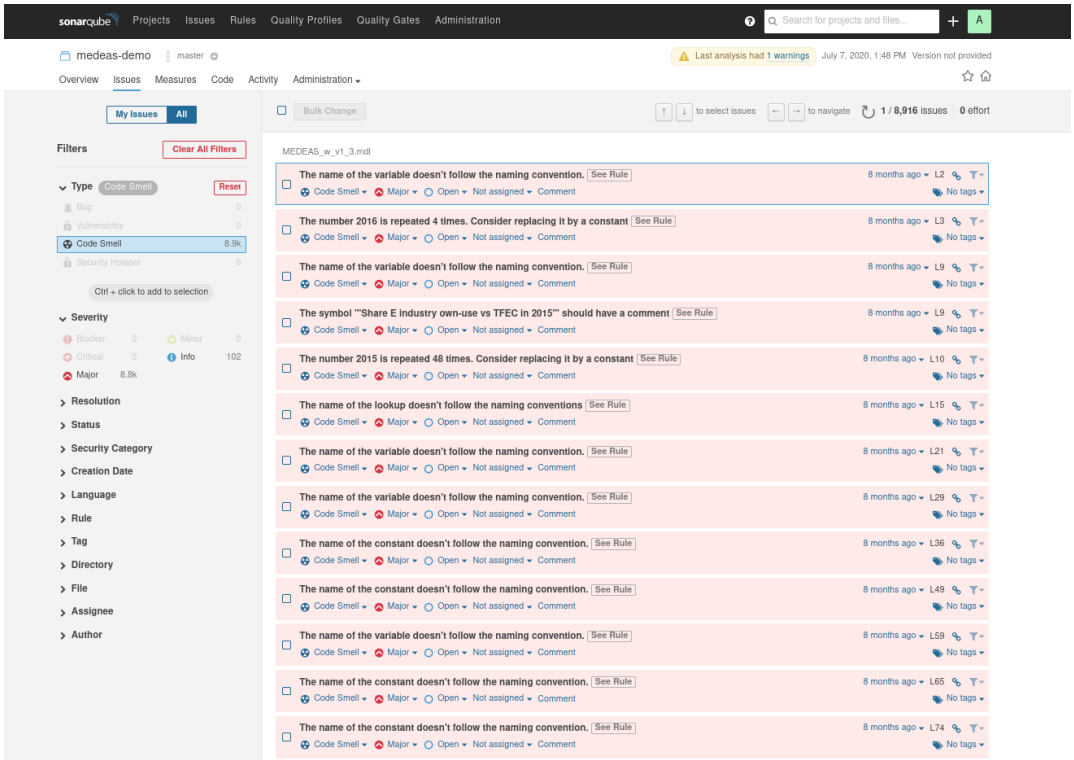

Figura A.7: Ventana con la lista de issues de un proyecto.

A.3.3. Creación de tokens de autenticación

El token de autenticación determina qué usuario está haciendo el análisis. Por ello es importante que cada usuario ejecute análisis usando sus propios tokens. Para ello hay que autenticarse en la plataforma web, pulsar en el icono del usuario que hay arriba a la izquierda, pulsar en *My Account* e ir a la pestaña de seguridad. En esta pestaña se puede crear un token indicando el nombre del token y pulsando en *Generar*.

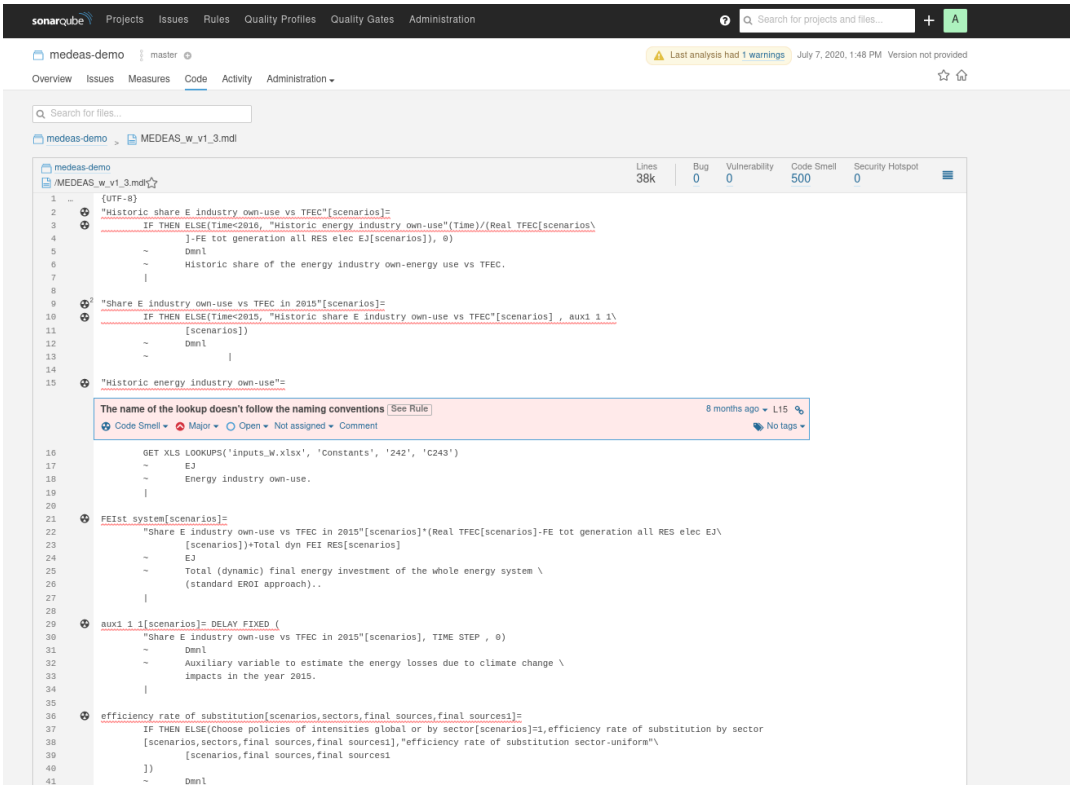

Figura A.8: Ventana con el código de un proyecto junto a sus issues.

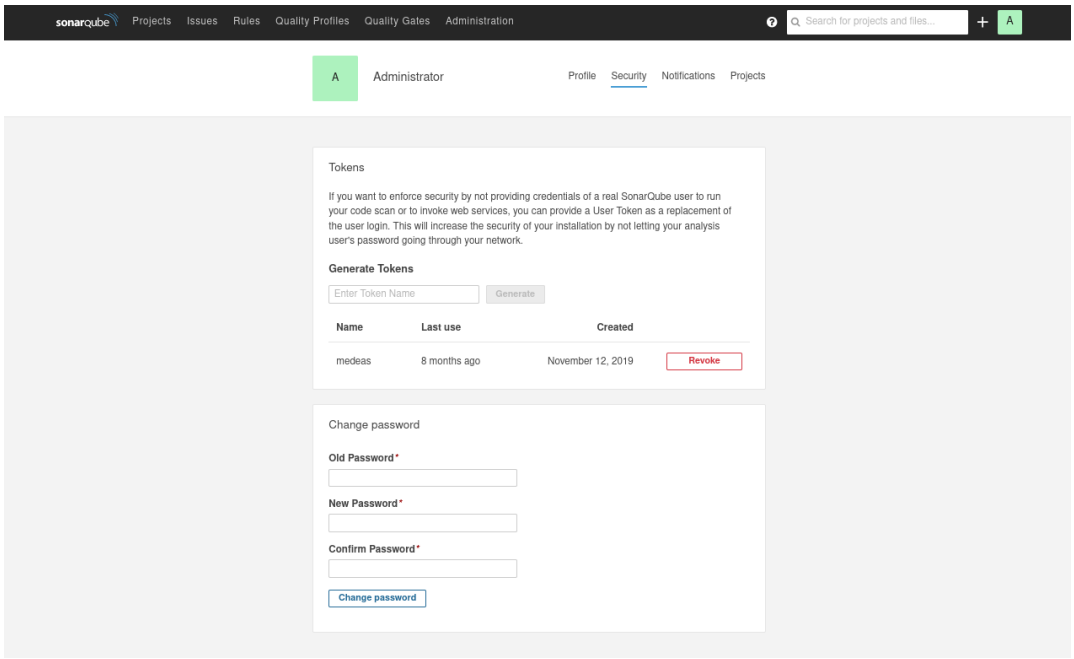

Figura A.9: Pestaña en la que se crean tokens de autenticación.

Apéndice B

Resumen de enlaces adicionales

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

- Repositorio del código: `https://gitlab.inf.uva.es/danbaza/tfg-sonarvensim`.
- Plugin desplegado para pruebas en instancia de SonarQube. Contiene ejemplos de proyectos analizados: `http://tfg2020.virtual.lab.inf.uva.es:20202/`.
- Plugin desplegado en instancia de SonarQube para dar soporte al desarrollo del IAM del proyecto Locomotion: `https://sonarqube-locomotion.infor.uva.es/`

Bibliografía

[1] Agencia Tributaria. Tabla de coeficientes de amortización lineal. https://www.agenciatributaria.es/AEAT.internet/Inicio/_Segmentos_/Empresas_y_profesionales/Empresas/Impuesto_sobre_Sociedades/Periodos_impositivos_a_partir_de_1_1_2015/Base_imponible/Amortizacion/Tabla_de_coeficientes_de_amortizacion_lineal_.shtml. Accessed: 2020-7-7.

[2] Atlassian. Trello. https://trello.com/. Accessed: 2019-8-25.

[3] 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. Accessed: 2019-9-25.

[4] Scrum Manager BoK. Scrum: roles y responsabilidades. https://www.scrummanager.net/bok/index.php?title=Artefactos. Accessed: 2020-6-28.

[5] Yania Crespo, Ignacio de Blas, Jesús Vegas, Luis Javier Miguel González, Iñigo Capellán-Pérez, David Álvarez, Martin Baumann, Margarita Mediavilla, Roger Samsó, César Llamas, Carmen Hernández, Luis Fernando Lobejón, Iñaki Arto, Ignacio Cazcarro, and Gonzalo Parrado. Report of the common modeling framework. https://cordis.europa.eu/project/id/821105. version 1: May 2020, version 2: July 2020, web publication: forthcoming.

[6] Deloitte. Scrum: roles y responsabilidades. https://www2.deloitte.com/es/es/pages/technology/articles/roles-y-responsabilidades-scrum.html. Accessed: 2020-6-28.

[7] Vincent Driessen. A successful Git branching model. https://nvie.com/posts/a-successful-git-branching-model/. Accessed: 2019-10-30.

[8] Antonio G. Encinas. El 21 de junio volverá la movilidad libre, pero la junta podrá restringir actividades. https://www.elnortedecastilla.es/castillayleon/junio-volvera-movilidad-20200612205600-nt.html. Accessed: 2020-7-7.

[9] European Commission. What is Horizon 2020? https://ec.europa.eu/programmes/horizon2020/what-horizon-2020. Accessed: 2020-7-5.

[10] Szczepan Faber. Mockito. https://site.mockito.org/. Accessed: 2019-8-25.

[11] Todd Fincannon. Original Vensim grammar in ANTLR 4. `https://github.com/climateinteractive/antlr4-vensim/commit/703c85da8f0e9a66ca4f2cd17622e6d5b89d1155`. Accessed: 2019-9-25.

[12] GEEDS. Group of Energy, Economy and Systems Dynamics. `https://geeds.es/en/`. Accessed: 2020-7-5.

[13] Git Community. Git. `https://git-scm.com/`.

[14] GitLab Inc. GitLab. `https://about.gitlab.com/`. Accessed: 2019-8-25.

[15] GitLab Inc. GitLab and SSH keys. `https://docs.gitlab.com/ee/ssh/`. Accessed: 2020-1-19.

[16] GitLab Inc. GitLab CI/CD pipeline configuration reference. `https://docs.gitlab.com/ee/ci/yaml/#before_script-and-after_script`. Accessed: 2020-1-19.

[17] Bill Hare, Robert Brecha, and Michiel Schaeffer. Integrated Assessment Models: what are they and how do they arrive at their conclusions? `https://climateanalytics.org/media/climate_analytics_iams_briefing_oct2018.pdf`, 2018. Accessed: 2020-7-5.

[18] James Houghton. PySD - system dynamics modeling in python. `https://github.com/JamesPHoughton/pysd`. Accessed: 2019-9-18.

[19] Jetbrains. Intellij. `https://www.jetbrains.com/idea/`. Accessed: 2019-8-25.

[20] Lars Kappert. Dyson - Github. `https://github.com/webpro/dyson`. Accessed: 2019-8-25.

[21] LOCOMOTION Consortium. LOCOMOTION website. `https://www.locomotion-h2020.eu/`. Accessed: 2020-7-5.

[22] MEDEAS Consortium. Deliverables - medeas. `https://www.medeas.eu/deliverables`. Accessed: 2019-9-30.

[23] MEDEAS Consortium. MEDEAS website. `https://www.medeas.eu/`. Accessed: 2020-7-5.

[24] Microsoft. Skype. `https://www.skype.com/en/`. Accessed: 2019-8-25.

[25] Ministerio de Empleo y Seguridad Social, Gobierno de España. XVII Convenio colectivo estatal de empresas de consultoría, y estudios de mercado y de la opinión pública. `https://www.boe.es/boe/dias/2018/03/06/pdfs/BOE-A-2018-3156.pdf`. Accessed: 2020-7-7.

[26] Ministerio de la Presidencia, Relaciones con las Cortes y Memoria Democrática. Real Decreto 463/2020, de 14 de marzo, por el que se declara el estado de alarma para la gestión de la situación de crisis sanitaria ocasionada por el COVID-19. `https://www.boe.es/diario_boe/txt.php?id=BOE-A-2020-3692`. Accessed: 2020-7-7.

[27] Terence Parr. ANTLR. `https://www.antlr.org/`. Accessed: 2019-8-25.

[28] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, second edition, 2013.

[29] Terence Parr and Kathleen S. Fisher. LL(*): The foundation of the ANTLR parser generator. https://www.antlr.org/papers/LL-star-PLDI11.pdf. Accessed: 2019-9-26.

[30] Elvira Ramajo. ¿Cuánto le cuesto a mi empresa?". https://www.ramajoasesores.com/cuanto-le-cuesto-a-mi-empresa/. Accessed: 2020-7-7.

[31] Rocket.Chat. Rocket.Chat. https://rocket.chat/. Accessed: 2019-8-25.

[32] SaraClip. Eventos en Scrum I. https://www.saraclip.com/eventos-en-scrum/. Accessed: 2020-6-28.

[33] Scrum.org. What is Scrum? https://www.scrum.org/resources/what-is-scrum. Accessed: 2020-6-28.

[34] SonarSource. Analysis Parameters - Sonarqube Documentation. https://docs.sonarqube.org/latest/analysis/analysis-parameters/. Accessed: 2019-2-4.

[35] SonarSource. Filelinescontext.java - Sonar-Plugin-API Source Code. https://github.com/SonarSource/sonarqube/blob/master/sonar-plugin-api/src/main/java/org/sonar/api/measures/FileLinesContext.java. Accessed: 2019-7-24.

[36] SonarSource. Javascriptsensor.java - SonarJS Source Code. https://github.com/SonarSource/SonarJS/blob/master/sonar-javascript-plugin/src/main/java/org/sonar/plugins/javascript/JavaScriptSensor.java. Accessed:2019-7-25.

[37] SonarSource. Javasquid.java - SonarJava Source Code. https://github.com/SonarSource/sonar-java/blob/master/java-frontend/src/main/java/org/sonar/java/JavaSquid.java. Accessed:2019-7-25.

[38] SonarSource. Nosonarfilter.java - Sonar-Plugin-API Source Code. https://github.com/SonarSource/sonarqube/blob/master/sonar-plugin-api/src/main/java/org/sonar/api/issue/NoSonarFilter.java. Accessed: 2019-7-24.

[39] SonarSource. Plugin basics. https://docs.sonarqube.org/8.2/extend/developing-plugin/. Accessed: 2019-9-23.

[40] SonarSource. plugin basics - sonarqube documentation.

[41] SonarSource. Prerequisites and Overview - Sonarqube Documentation. https://docs.sonarqube.org/8.2/requirements/requirements/#header-3. Accessed: 2019-7-20.

[42] SonarSource. Project Page. https://docs.sonarqube.org/7.9/user-guide/project-page/. Accessed: 2019-7-20.

[43] SonarSource. Python.java - SonarPython Source Code. https://github.com/SonarSource/sonar-python/blob/master/sonar-python-plugin/src/main/java/org/sonar/plugins/python/Python.java. Accessed: 2019-7-22.

[44] SonarSource. Pythonrulerepository.java - SonarPython Source Code. https://github.com/SonarSource/sonar-python/blob/master/sonar-python-plugin/src/main/java/org/sonar/plugins/python/PythonRuleRepository.java. Accessed: 2019-7-22.

[45] SonarSource. Pythonscanner.java - SonarPython Source Code. https://github.com/SonarSource/sonar-python/blob/master/sonar-python-plugin/src/main/java/org/sonar/plugins/python/PythonSensor.java. Accessed:2019-7-25.

[46] SonarSource. Quality Profiles. https://docs.sonarqube.org/8.2/instance-administration/quality-profiles/. Accessed: 2019-7-23.

[47] SonarSource. Release upgrade notes - 7.9 lts. https://docs.sonarqube.org/7.9/setup/upgrade-notes/. Accessed: 2019-21-9.

[48] SonarSource. Sensordescriptor.java - SonarPython Source Code. https://github.com/SonarSource/sonarqube/blob/master/sonar-plugin-api/src/main/java/org/sonar/api/batch/sensor/SensorDescriptor.java. Accessed: 2019-7-25.

[49] SonarSource. Sonar custom plugin example. https://github.com/SonarSource/sonar-custom-plugin-example. Accessed: 2019-9-23.

[50] SonarSource. SonarJS. https://github.com/SonarSource/SonarJS. Accessed: 2019-9-15.

[51] SonarSource. Sonarqube. https://www.sonarqube.org/. Accessed: 2019-8-25.

[52] SonarSource. SonarQube Python Plugin. https://github.com/SonarSource/sonar-python. Accessed: 2019-9-15.

[53] SonarSource. SonarScanner. https://docs.sonarqube.org/latest/analysis/scan/sonarscanner/. Accessed: 2019-8-25.

[54] SonarSource. SSLR Github code - leftrecursivegrammar.java. https://github.com/SonarSource/sslr/blob/master/sslr-examples/src/main/java/org/sonar/sslr/examples/grammars/LeftRecursiveGrammar.java. Accessed: 2019-9-26.

[55] SonarSource. SSLR (SonarSource Language Recognizer). https://github.com/SonarSource/sslr. Accessed: 2019-9-20.

[56] SonarSource. SSLR (SonarSource Language Recognizer). https://github.com/antlr/antlr4. Accessed: 2019-9-20.

[57] SonarSource. Supporting New Languages - Sonarqube Documentation. https://docs.sonarqube.org/8.2/extend/new-languages/. Accessed: 2019-7-20.

[58] The Apache Software Foundation. Apache Maven Project. https://maven.apache.org/. Accessed: 2019-8-25.

[59] The JUnit Team. Junit. https://junit.org/. Accessed: 2019-8-25.

[60] The PostgreSQL Global Development Group. Postgresql. https://www.postgresql.org/. Accessed: 2019-8-25.

[61] Universidad de Valladolid. Preguntas frecuentes. https://www.uva.es/export/sites/uva/2.docencia/2.02.mastersoficiales/2.02.13.preguntasfrecuentes/index.html. Accessed: 2019-7-6.

[62] Universidad de Valladolid. Proyecto docente del trabajo de fin de grado 2019-2020 (Mención Ingeniería de Software). https://alojamientos.uva.es/guia_docente/uploads/2019/545/46976/1/Documento.pdf. Accessed: 2019-7-6.

[63] Ventana Systems. Constraints - vensim documentation. https://www.vensim.com/documentation/20970.htm. Accessed: 2019-9-30.

[64] Ventana Systems. Defining external functions. https://www.vensim.com/documentation/25725.htm. Accessed: 2020-7-5.

[65] Ventana Systems. GET DIRECT CONSTANTS - vensim documentation. https://www.vensim.com/documentation/fn_get_direct_constants.htm. Accessed: 2019-9-30.

[66] Ventana Systems. GET DIRECT SUBSCRIPT - vensim documentation. https://www.vensim.com/documentation/fn_get_direct_subscript.htm. Accessed: 2019-9-30.

[67] Ventana Systems. GET XLS CONSTANTS - vensim documentation. https://www.vensim.com/documentation/fn_get_xls_constants.htm. Accessed: 2019-9-30.

[68] Ventana Systems. GET XLS LOOKUPS - vensim documentation. https://www.vensim.com/documentation/fn_get_xls_lookups.htm. Accessed: 2019-9-30.

[69] Ventana Systems. GET XLS SUBSCRIPT - vensim documentation. https://www.vensim.com/documentation/fn_get_xls_subscript.htm. Accessed: 2019-9-30.

[70] Ventana Systems. Ignoring variables - vensim documentation. https://www.vensim.com/documentation/ref_ignore.htm. Accessed: 2019-9-30.

[71] Ventana Systems. Mapping of subscript ranges - vensim documentation. https://www.vensim.com/documentation/ref_subscript_mapping.htm. Accessed: 2019-9-30.

[72] Ventana Systems. :NA: - vensim documentation. https://www.vensim.com/documentation/na.htm. Accessed: 2019-9-30.

[73] Ventana Systems. Operators - vensim documentation. https://www.vensim.com/documentation/operators.htm. Accessed: 2019-9-30.

[74] Ventana Systems. Variable information - vensim documentation. https://www.vensim.com/documentation/variable_information.htm. Accessed: 2019-9-30.

[75] Ventana Systems. Variable types. https://www.vensim.com/documentation/ref_variable_types.htm. Accessed: 2020-7-6.

[76] Ventana Systems. Vector functions - vensim documentation. https://www.vensim.com/documentation/21230.htm. Accessed: 2019-9-30.

[77] Ventana Systems. Vensim. https://vensim.com/. Accessed: 2020-7-5.

[78] Ventana Systems. Vensim Model Reader - vensim documentation. `https://vensim.com/vensim-model-reader/`. Accessed: 2019-9-30.

[79] Ventana Systems. What is a parent/sub-model? `https://vensim.com/documentation/what-is-a-sub-model_.htm`. Accessed: 2020-7-6.

[80] Alessandro Warth, James R. Douglass, and Todd Millstein. Packrat parsers can support left recursion. `http://www.vpri.org/pdf/tr2007002_packrat.pdf`. Accessed: 2019-9-26.