Universidad de Valladolid

ESCUELA DE INGENIERÍA INFORMÁTICA

GRADO EN INGENIERÍA INFORMÁTICA
Mención en Ingeniería del Software

Evolución y mejora en la traducción de código Vensim a código Python:

Ingeniería inversa y mantenimiento de PySD

Alumna: María Robles del Blanco

Tutores: Yania Crespo González-Carvajal
David Escudero Mancebo

A mis padres, por todo su apoyo

Agradecimientos

A mi familia, por su gran amor y continuo apoyo.

A mis amigos, por ayudarme a crecer día a día y acompañarme en todo momento.

A mi tutora Yania, por su gran supervisión y guía durante el proyecto y, sobretodo, por su gran labor como profesora.

A mi tutor David, por sus sugerencias y ayuda.

Al grupo GEEDS de la UVa, por su aportación con la realización de modelos y compartir sus conocimientos.

Al proyecto LOCOMOTION, por la financiación de la beca para este trabajo.

Resumen

El objetivo de este Trabajo de Fin de Grado es mejorar el proyecto de PySD para poder traducir IAMs (*Integrated Assessment Models*) creados con Vensim al lenguaje Python. Este Trabajo de Fin de Grado se engloba dentro del proyecto europeo LOCOMOTION [27], el cual está basado en desarrollar modelos sofisticados que permiten evaluar el impacto ambiental y socieconómico que tendrá la toma de ciertas decisiones, para poder llegar a obtener un futuro sostenible con bajas emisiones de carbono.

Más concretamente, este proyecto tiene como objetivo la traducción del modelo WILIAM, basado en modelos MEDEAS [47] existentes, que simulan las relaciones altamente complejas entre el ser humano y el entorno. El citado modelo se desea traducir del lenguaje Vensim al lenguaje Python, para lo que se utilizará el proyecto PySD, desarrollando funcionalidades adicionales necesarias.

Este trabajo se ha desarrollado bajo el marco de trabajo Scrum.

Finalmente, se ha conseguido agregar funcionalidad nueva a PySD, corregir ligeros fallos y colaborar con otros *contributors* mediante tres *pull requests* realizadas al repositorio central de PySD. Además, se ha creado documentación en inglés sobre el funcionamiento interno de PySD, para reducir la complejidad de la curva de aprendizaje de futuros desarrolladores.

Abstract

The purpose of this project is to improve the PySD project by translating IAMs (*Integrated Assessment Models*) created with Vensim into the Python language. This final degree project is part of the European LOCOMOTION project [27], which is based on the development of sophisticated models that will assess the socioeconomic and enviromental impact of certain decisions in order to achieve a sustainable future with low-carbon emisions.

Specifically, this project aims to translate the WILIAM model, based on MEDEAS models [47], which simulate the complex relationship between the humans and the enviroment. This model is to be translated from the Vensim language into the Python language, for which the PySD project will be used, developing the necessary additional functionalities.

This project has been developed under the Scrum framework.

Finally, it has been possible to add new functionalities to PySD, fix minor bugs and collaborate with other contributors through three pull requests made to the PySD central repository. In addition, documentation on the inner workings of PySD has been created in English to reduce the complexity of the learning curve for future developers.

Índice general

Lista de Figuras

Lista de Tablas

Capítulo 1

Introducción

1.1. Motivación

Este Trabajo de Fin de Grado se ha realizado como parte de una beca de colaboración con el proyecto europeo LOCOMOTION [27]. Con la financiación del programa Horizonte 2020 [19] de la UE, LOCOMOTION pretende diseñar un conjunto de IAM (Integrated Assessment Models) que permitirá obtener un sistema de modelado confiable con el que se podrá evaluar las diferentes opciones relacionadas con políticas de sostenibilidad. LOCOMOTION ayudará a la toma de decisiones en base a los resultados del modelado dinámico con variables ambientales, económicas, sociales, tecnológicas y biofísicas para poder llegar a una sociedad con menores emisiones de carbono.

En el ámbito de LOCOMOTION se desarrolla el modelo integrado de evaluación WILIAM (*"within limits"*), basado en modelos MEDEAS existentes. Se trata de un modelo a través del cual se simulan complejos escenarios medioambientales permitiendo ver las consecuencias que tendría la toma de ciertas decisiones. Más allá de eso, se aspira a poder permitir a todo el mundo que quiera la posibilidad de simular y tomar sus propias decisiones, intentando así concienciar de la importancia que tienen éstas. Actualmente, el modelo está desarrollado en el lenguaje Vensim, utilizando unas funcionalidades específicas de la licencia de pago del programa.

Se determinó poder convertir el modelo a un lenguaje gratuito, como es el caso del lenguaje Python. La traducción del modelo a Python permitirá que sea utilizado en otros proyectos y su código pueda ser publicado. Asimismo, cualquier persona que lo desee podrá ejecutar el citado modelo y obtener los resultados en base a las decisiones tomadas.

Por tanto, la motivación que dirige este Trabajo de Fin de Grado es realizar una traducción automática del modelo WILIAM a un lenguaje abierto.

1.2. Objetivos

El objetivo fundamental de este TFG es lograr la traducción del modelo WILIAM en el lenguaje Vensim al lenguaje Python. Para poder alcanzarlo, se divide en los siguientes subobjetivos:

- Comprensión del lenguaje Vensim. La alumna no había trabajado anteriormente con este software ni con ningún simulador de sistemas, por lo que es necesario estudiar el funcionamiento de Vensim y de algunas de sus funcionalidades para poder reproducir su comportamiento en Python.
- Comprensión del proyecto PySD. Estudiar el funcionamiento interno con el que cuenta el proyecto ya comenzado de PySD para poder realizar mejoras en él. Además, el proceso de estudio sobre este proyecto será plasmado como documentación del mismo para disminuir la curva de aprendizaje a futuros desarrolladores.
- Contribución al repositorio de PySD. Los cambios realizados sobre el proyecto de PySD, deberán añadirse al repositorio central para que los colaboradores y todo el que lo desee pueda acceder a él.

1.3. Proyectos open source

Un proyecto *open source*, también conocido como código abierto, es aquel que presenta licencias abiertas que permiten a desarrolladores y usuarios acceder al código fuente del proyecto y a los documentos de diseño [58]. Así, la gente que lo desee, puede involucrarse en un proyecto *open source*. Normalmente, los principales motivos por los que alguien quiere formar parte de un desarrollo *open source* son diferentes pero, al final, contribuye una gran cantidad de gente que da como resultado un amplio rango de talento volcado en el proyecto. Esta modalidad de desarrollo software se enfoca más en los beneficios prácticos que puede llegar a tener el código fuente que en cuestiones éticas y de libertad que se tratan en el movimiento de *software libre* [51].

Existen tres términos básicos relacionados con *open source*: contribuidores o *contributors*, *fork* y *pull request*.

Normalmente, en un proyecto de este estilo, se parte de un proyecto raíz al que se le suman diferentes **contribuidores** que añaden funcionalidad nueva y/o mejoras al código o a la documentación del proyecto raíz. Para ser un contribuidor más, es necesario realizar un ***fork*** de la rama del proyecto en la que se quiera añadir o mejorar funcionalidad. Este *fork* es una copia del repositorio y en él se podrán efectuar los cambios que se crean convenientes sin modificar el proyecto original. Además, a través de las ***issues*** de GitHub, se suelen debatir las deficiencias del proyecto y se especifica qué falta por mejorar. Cada contribuidor puede añadir las *issues* que crea conveniente o los problemas que encuentre y, a su vez, puede intentar solventar alguna *issue* ya presentada.

Posteriormente, cuando un contribuidor haya terminado de realizar los cambios que crea necesarios, deberá hacer una ***pull request*** al dueño del repositorio desde el cual se ha hecho previamente el *fork*. La aceptación o denegación de dicha solicitud está guiada por una serie de requisitos impuestos, así como puede exigirse un cierto porcentaje de cobertura de tests sobre la funcionalidad añadida para corroborar la mejora que supone al proyecto. Cuando se realiza una *pull request* se abre un debate en el que los demás contribuidores pueden pedir que se modifiquen ciertas partes del código que se ha añadido o pueden realizar preguntas sobre fragmentos de código que no entiendan o que falte documentación al respecto. Una vez resueltas las posibles dudas, el dueño de la rama donde se ha hecho el *fork*, si acepta la *pull request*, se añadirán los cambios realizados a la rama principal.

1.4. Ingeniería Inversa

El origen de la ingeniería inversa se encuentra en la ingeniería mecánica. La ingeniería se ocupa de diseñar elementos de un producto para poder obtener un aparato funcional, la ingeniería inversa invierte este proceso [26]. Este tipo de desarrollo permite obtener información acerca de cuáles son los componentes de un determinado sistema y de qué manera interactúan entre sí.

La **ingeniería inversa** [50] es un método de resolución, que supone una gran profundización en el estudio de un proyecto, hasta el punto de que se pueda llegar a entender, modificar y mejorar el modo de funcionamiento de dicho proyecto.

Para realizar el mantenimiento del proyecto PySD, teniendo en cuenta que la alumna no tenía ningún conocimiento previo sobre él, ha sido necesario realizar una comprensión exhaustiva del software, conocer cómo funciona, cuáles son los objetivos y cómo se estructura internamente. Para esto se ha utilizado el método de **ingeniería inversa**.

1.4.1. Ingeniería Inversa de software

El Instituto de Ingenieros Eléctricos y Electrónicos (IEEE), en 1990, definió la **ingeniería inversa** como el proceso de analizar un sistema para identificar sus componentes y sus interrelaciones y crear representaciones del sistema en otra forma o en un nivel superior [1]. Es un proceso de examen únicamente, ya que el sistema de software en consideración no se modifica, o bien se rediseña o se reestructura. La ingeniería inversa se puede realizar desde cualquier etapa del ciclo de vida del producto, no necesariamente cuando éste haya finalizado.

La aplicación a un proyecto de ingeniería inversa nunca debe cambiar la funcionalidad del mismo. Al poner en práctica esta metodología se pueden encontrar algunas ventajas: se puede reducir la complejidad del sistema, generar diferentes alternativas como representaciones gráficas, recuperar y/o actualizar la información perdida, detectar efectos laterales y facilitar la reutilización de sistemas existentes.

1.5. Proyectos de mantenimiento

En muchas ocasiones el software necesita ser modificado, ya sea por detección de errores, nuevas exigencias o necesidades del usuario del sistema. A esta fase se le conoce fase de mantenimiento [67]. Entre las características sobresalientes de mantenimiento de software destacan:

- El software no envejece realmente pero se puede degradar o quedar obsoleto.
- El mantenimiento de software supone adaptar el paquete o sistema objeto del mismo a nuevas situaciones como cambio del hardware o del sistema operativo.
- Todo sistema software conlleva mejoras o añadidos indefinidamente.

Existen 4 tipos de mantenimiento: correctivo, adaptativo, perfectivo y preventivo, explicados a continuación. Este Trabajo de Fin de Grado se enmarca como un proyecto de **mantenimiento perfectivo y preventivo** de PySD.

1.5.1. Mantenimiento correctivo

Localiza y elimina los posibles defectos del software que pueden provocar fallos en el mismo. Estos fallos pueden originar un comportamiento diferente al esperado. Estos fallos pueden ser de procesamiento (salidas incorrectas de un programa), rendimiento (tiempo de respuesta demasiado alto), programación (inconsistencias en el diseño) o documentación (inconsistencias entre la funcionalidad de un programa y el manual de usuario).

1.5.2. Mantenimiento adaptativo

Tiene lugar cuando se realizan cambios en el entorno en el que se ejecuta, tanto hardware como software, para mantener la plena funcionalidad del sistema. Estos cambios varían desde cambios en el sistema operativo, en la arquitectura física del sistema informático hasta cambios en el entorno de desarrollo software. Se trata de un tipo de mantenimiento cada vez más frecuente debido a la tendencia actual de actualización de hardware y sistemas operativos cada poco tiempo.

1.5.3. Mantenimiento perfectivo

Conjunto de actividades para mejorar o añadir nuevas funcionalidades requeridas por el usuario. Puede ser de ampliación, en el cual se incorporan nuevas funcionalidades al sistema, o de mejora de eficiencia de la ejecución.

1.5.4. Mantenimiento preventivo

Tiene como objetivo principal mejorar las propiedades del software (calidad y mantenibilidad) sin alterar sus especificaciones funcionales. Algunos ejemplos de mantenimiento preventivo pueden ser: incluir sentencias que comprueben la validez de los datos de entrada, reestructuración de los programas para aumentar su legibilidad o incluir nuevos comentarios. Este tipo de mantenimiento utiliza las técnicas de ingeniería inversa y reingeniería.

1.6. Estructura de la memoria

En adelante, este documento se estructura de la siguiente forma:

Capítulo 2. Requisitos y planificación: Describe el marco de trabajo utilizado en este proyecto y su adaptación al mismo.

Capítulo 3. Marco teórico: Describe todos los conocimientos teóricos que forman la base y han sido necesarios para desarrollar y entender el proyecto.

Capítulo 4. Ingeniería Inversa: Describe el proceso de ingeniería inversa que se ha llevado a cabo para comprender el funcionamiento de la versión de PySD desde la que se partía.

Capítulo 5. Tecnologías utilizadas: Describe todas las tecnologías necesarias para la gestión y desarrollo del proyecto.

Capítulo 6. Seguimiento del proyecto: Describe la evolución del proyecto dividida en sprints.

Capítulo 7. Diseño de las modificaciones: Describe las modificaciones y mejoras realizadas sobre PySD.

Capítulo 8. Implementación y pruebas de las modificaciones: Describe los tests realizados, la organización de cada modificación junto con enlaces a los cambios comentados y la cobertura final obtenida.

Capítulo 9. Conclusiones: Describe los resultados, el aprendizaje que ha supuesto el proyecto y las líneas de trabajo futuras.

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

Anexo B Resumen de enlaces adicionales: Incluye enlaces de interés sobre el proyecto, como el repositorio de código de pysd en GitLab y GitHub, el repositorio de código de los tests en GitHub y las issues que se han abierto durante el desarrollo del Trabajo de Fin de Grado.

Capítulo 2

Requisitos y Planificación

2.1. Scrum

Para el desarrollo del proyecto se va a utilizar el marco de desarrollo ágil **Scrum**.

Scrum [61] es un marco de trabajo iterativo e incremental para desarrollo ágil de software consistente en aplicar de manera regular un conjunto de buenas prácticas para trabajar en equipo y así poder obtener el mejor resultado posible para el proyecto. Se definen una serie de roles para los integrantes del equipo, especifica también los artefactos necesarios para los procesos, los bloques de tiempo preestablecidos y las reuniones que se deben respetar.

Este marco de trabajo está especialmente indicado para proyectos que se encuentran en entornos complejos donde se necesitan obtener resultados tempranos, o aquellos proyectos en los que los requisitos son cambiantes o poco definidos y el proyecto cuenta con cierta incertidumbre.

Los elementos principales en Scrum son los artefactos, eventos y roles.

2.1.1. Artefactos

Los artefactos definidos por Scrum fomentan la transparencia de la información, de manera que todos los integrantes tengan el mismo concepto de lo que se está llevando a cabo y del progreso [16]. Estos artefactos son:

- ***Product Backlog***
  Lista con los requisitos iniciales del proyecto que se va a desarrollar. Irá evolucionando y cambiando a medida que lo hace el producto y el entorno del proyecto. Dicha lista, contiene una descripción de cada tarea y subtarea que se necesitan realizar para la ejecución de cada requisito, organizadas en base a la prioridad de cada una.

- ***Sprint Backlog***
  Subconjunto de objetivos y requisitos seleccionados del *Product Backlog*, acordados para realizarlos en el sprint actual.

- ***Incremento***
  Son las iteraciones o productos entregables que se han desarrollado durante un sprint. El concepto de sprint o iteración se define a continuación.

2.1.2. Eventos

Scrum define varios eventos que establecen una rutina de reuniones, fomentando así la comunicación y colaboración del equipo. De esta forma se consigue reducir el tiempo empleado en reuniones extensas. Todos los eventos son bloques de tiempo que tienen definidos una duración máxima [17]. Estos son los eventos de Scrum:

- ***Sprint***
  Iteraciones sucesivas en las que se realiza el proyecto. Cada sprint tiene una duración fijada previamente, normalmente entre una y cuatro semanas. En cada sprint se debe declarar un objetivo y al final de éste, se debe haber aportando cierto valor tangible al producto. Las iteraciones son cortas y permiten corregir el rumbo. Sin embargo, si ocurriese algún inconveniente no se intenta solucionar durante el sprint, se espera para intentar solucionarlo en el siguiente sprint.

- ***Sprint Planning***
  Reunión que tiene como propósito definir el objetivo del sprint actual, negociando los ítems del *Product Backlog* que serán ejecutados o desarrollados en ese sprint. Participa todo el *Scrum Team*. Se determina qué es lo que se entregará en el incremento resultante del sprint que se va a comenzar y el trabajo necesario para ello. Posteriormente, el equipo de desarrollo revisa los ítems con mayor prioridad y cuáles pueden finalizarse en ese sprint. La duración de esta reunión no debe durar más de 2 horas por cada semana de sprint.

- ***Daily Scrum***
  Reunión diaria fijada. Participa el equipo de desarrollo reuniéndose un máximo de 15 minutos. Se realiza una breve inspección y adaptación, donde cada integrante del equipo de desarrollo comenta lo que consiguió el día anterior, qué va a realizar en el día actual y los obstáculos que está encontrando.

- ***Sprint Review (Revisión de Sprint)***
  El propósito de esta reunión es revisar el trabajo realizado por el Equipo de Desarrollo durante el sprint. Se realiza al final de cada sprint y puede afectar al *Product Backlog*, incluyendo nuevos requisitos, mejoras o incluso eliminaciones si fuera necesario. Es organizada por el *Product Owner* y requiere la presencia del todo el equipo de Scrum.

- ***Sprint Retrospective (Retrospectiva de Sprint)***
  Reunión que busca la mejora del equipo de Scrum. En ella se expone todo lo que ha ocurrido en el sprint y se hace hincapié sobre lo que no está funcionando o lo que se puede mejorar.

En la Figura 2.1 se muestra la relación entre los diferentes artefactos y eventos de Scrum.

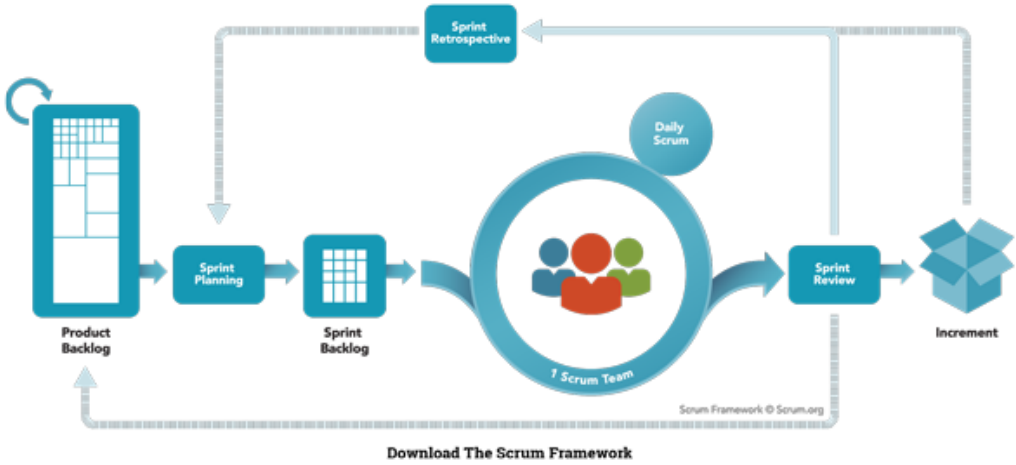

Figura 2.1: Artefactos y Eventos de Scrum. [30]

2.1.3. Roles

Scrum define tres roles bien diferenciados. Un equipo de Scrum está compuesto por un grupo de 5 a 11 personas, dentro del cual se diferencian: el **equipo de desarrollo**, formado por un mínimo de 3 personas y con un máximo de 9, el **Product Owner** y el **Scrum Master**.

- ***Product Owner***:
  Es la única persona autorizada para definir el *Product Backlog*, lista donde se definen las funcionalidades y características del proyecto. Representa al cliente y usuarios del software, así como a todas aquellas partes interesadas en el producto.

- ***Scrum Master***:
  Encargado de gestionar el *Sprint Backlog*, lista ordenada que contiene las tareas e historias del *Product Backlog* que se ha decido realizar en el sprint actual.

- ***Equipo de desarrollo***:
  Grupo de personas que se encargan de gestionar y realizar las tareas del *Sprint Backlog*, así como de cumplir las tareas y objetivos del *Product Backlog*. Asisten a los *daily meeting* para plasmar el progreso, resolver los problemas que puedan surgir y comentar lo que se va a desarrollar o implementar en ese día.

2.2. Adaptación de Scrum al proyecto

Habiendo tratado las principales características con las que cuenta *Scrum*, hay que adaptar este marco de trabajo al contexto del proyecto en el que se aplique.

En este Trabajo de Fin de Grado, se ha determinado que la tutora actúe como *Scrum Master* y *Product Owner*, mientras que la alumna será el único miembro que forme el equipo de desarrollo.

Se establecen sprints de dos semanas de duración, con aproximadamente 30 horas de trabajo por cada sprint. Además, se ha decidido establecer reuniones semanales o *weeklies* en lugar de *dailies*. En este conjunto, se obtienen un total de tres reuniones por cada sprint. En la primera reunión del sprint se realiza el *Sprint planning*, estimándose los requisitos del Sprint Backlog que se van a llevar a cabo durante ese sprint. En la segunda *weekly* se revisa el avance de las historias de usuario. Y para finalizar el sprint, se realiza la tercera *weekly* que consta de una *Sprint Review*. En esta última reunión semanal se realizará también el inicio del siguiente sprint, por lo que coincidirá con el *Sprint Planning* siguiente.

2.3. Product Backlog inicial

En la Tabla 2.1 se muestra el Product Backlog con los requisitos iniciales. En él se han listado todas las necesidades del proyecto, que han ido evolucionando a medida que ha avanzado el desarrollo del mismo. La Tarea 6 se ha decidido dividir en sub-tareas debido a su complejidad, lo que hará que no pueda realizarse en un único sprint.

2.4. Planificación inicial

La duración de la beca es de 6 meses a partir de septiembre, prorrogable hasta 12 meses. En la guía docente del Trabajo de Fin de Grado se establece una carga de trabajo de 12 ECTS equivalente a 300 horas. Como cada sprint tiene una duración de 2 semanas, se puede establecer una relación de 30 horas/sprint o lo que sería equivalente, 15 horas semanales. De esta forma, el requisito de 300 horas de trabajo estaría cumplido en 10 sprints. En la Tabla 2.2 se muestra la planificación de los sprints previstos.

Cabe destacar que una gran parte del TFG se realiza en el primer cuatrimestre, en paralelo con otras 5 asignaturas pendientes. De esta forma se han estimado periodos de inactividad o de reducción de la carga de trabajo para poder afrontar las asignaturas. Se establecieron dos semanas, del 18 de noviembre al 2 de diciembre, para poder afrontar la carga de trabajo de los exámenes parciales y las prácticas. Así como también se estableció que no se trabajaría en el proyecto durante el periodo de vacaciones de Navidad ni en la convocatoria ordinaria.

Número	Descripción
1	Como modelador de Vensim quiero poder traducir y ejecutar modelos que contengan la función Random 0 1 de Vensim para mantener la compatibilidad hacia atrás en PySD.
2	Como modelador de Vensim quiero poder trabajar con modelos Vensim cuya extensión de archivo se encuentre en mayúsculas para poder traducirlos y ejecutarlos en Python.
3	Como integrante del proyecto LOCOMOTION quiero poder utilizar modelos Vensim que contengan sentencias con varias comillas en los comentarios para poder traducirlos y ejecutarlos con el sistema PySD.
4	Como integrante del proyecto LOCOMOTION quiero poder utilizar la función Sample If True de Vensim en modelos para poder traducirlos y que sean ejecutados con PySD.
5	Como desarrollador del proyecto PySD quiero que el sistema tenga documentación sobre su arquitectura acompañada de los diagramas pertinentes para poder entender con mayor facilidad el funcionamiento del mismo.
6	Como integrante del proyecto LOCOMOTION quiero poder utilizar las operaciones de subscripts en modelos Vensim para que sean traducidas y ejecutadas en Python.
6.1	Como integrante del proyecto LOCOMOTION quiero poder utilizar la operación de copia de subscripts en modelos Vensim para que se traduzca y ejecute en Python.
6.2	Como integrante del proyecto LOCOMOTION quiero poder utilizar la operación de mapeo de subscripts en modelos Vensim para que se traduzca y ejecute en Python.
6.3	Como integrante del proyecto LOCOMOTION quiero poder utilizar la definición de subscripts a partir de un rango numérico para que se traduzca y ejecute en Python.

Tabla 2.1: Product Backlog Inicial

2.5. Riesgos

Un riesgo es un evento futuro que afecta negativa o positivamente a los objetivos del proyecto. Los riesgos están relacionados con la incertidumbre del futuro. Hay que tener siempre en cuenta que hay riesgos que pueden desencadenar otros.

Es necesario realizar una buena gestión de riesgos para poder evitar posibles efectos adversos que puedan ocasionar en el proyecto o que estos causen el menor daño posible una vez que sucedan.

Sprint	Fecha de comienzo	Fecha de finalización	Observaciones
Sprint 0	14/09/2020	21/10/2020	
Sprint 1	21/10/2020	4/11/2020	
Sprint 2	4/11/2020	18/11/2020	
Exámenes parciales			
Sprint 3	2/12/2020	16/12/2020	
Navidad y convocatoria ordinaria			
Sprint 4	20/1/2021	3/2/2021	
Sprint 5	3/2/2021	17/2/2021	
Sprint 6	17/2/2021	3/3/2021	
Sprint 7	3/3/2021	17/3/2021	
Sprint 8	17/3/2021	31/3/2021	
Sprint 9	31/3/2021	14/4/2021	
Sprint 10	14/4/2021	28/4/2021	
Sprint 11	28/4/2021	12/5/2021	Sprint de refuerzo
Sprint 12	12/5/2021	26/5/2021	Sprint de refuerzo

Tabla 2.2: Calendarización inicial del proyecto

Para llevar a cabo una buena gestión de riesgos es necesario seguir una serie de pasos:

1. **Identificar** los posibles riesgos del proyecto
2. **Analizar y priorizar** los riesgos, teniendo en cuenta el impacto y la probabilidad de cada uno
3. **Planificar** cómo se lidia con cada riesgo.
4. **Monitorizar y controlar** los riesgos, si han ocurrido o a aumentado su probabilidad.

Desde la Tabla 2.3 a la Tabla 2.11, se han presentado los diferentes riesgos analizados, acompañados de una breve descripción del mismo, su probabilidad e impacto, un plan de mitigación y un plan de contingencia.

Riesgo 1	Enfermedad de desarrollador
Descripción	Un miembro del equipo de desarrollo enferma y es incapaz de realizar sus tareas asignadas
Probabilidad	Baja
Impacto	Alto
Plan de mitigación	Añadir un sprint a mayores a la planificación para poder realizar las tareas que no haya dado tiempo
Plan de contingencia	Aumentar las horas de trabajo diarias

Tabla 2.3: Riesgo de enfermedad

Riesgo 2	Cambios en los requisitos
Descripción	A lo largo del desarrollo del proyecto, se necesitan cambiar los requisitos establecidos en un primer momento ya que se han encontrado diferentes necesidades
Probabilidad	Alta
Impacto	Medio
Plan de mitigación	Usar un marco de desarrollo ágil para el desarrollo del proyecto, como Scrum, hará más fácil lidiar con los posibles cambios en los requisitos que pueda haber, reduciendo así el alcance que este riesgo pueda tener
Plan de contingencia	Replanificar y llevar a cabo los cambios necesarios

Tabla 2.4: Riesgo de cambios en los requisitos

Riesgo 3	Falta de formación
Descripción	Debido a la naturaleza del proyecto, el desarrollador puede no tener el conocimiento necesario sobre el mismo
Probabilidad	Media
Impacto	Medio
Plan de mitigación	Sprint 0, sprint dedicado a realizar el trabajo de ingeniería inversa sobre el proyecto para aumentar los conocimientos de los desarrolladores sobre el proyecto
Plan de contingencia	Consultar a profesionales o contribuidores

Tabla 2.5: Riesgo de falta de formación

Riesgo 4	Falta de conocimiento del software Vensim para pruebas
Descripción	El desconocimiento del software Vensim puede derivar en problemas al intentar implementar pruebas para poder probar el código desarrollado
Probabilidad	Alta
Impacto	Bajo
Plan de mitigación	En el Sprint 0 se dedica un cierto número de horas a comprender mejor el lenguaje Vensim
Plan de contingencia	Preguntar a los expertos de Vensim, pidiendo las pruebas que sean necesarias

Tabla 2.6: Riesgo de falta de conocimiento Vensim para pruebas

Riesgo 5	Fallo en la planificación
Descripción	Puede haber retrasos si alguna tarea se estima de manera incorrecta, lo que puede afectar a la duración del proyecto
Probabilidad	Media
Impacto	Alto
Plan de mitigación	Al utilizar Scrum como marco de trabajo, se reduce el impacto si existiese la necesidad de replanificar. Además, se ha incluido dos sprints de refuerzo al final de la planificación para posibles retrasos que puedan existir
Plan de contingencia	Aumentar las horas de trabajo diarias o aumentar el plazo de entrega

Tabla 2.7: Riesgo de fallar en la planificación

Riesgo 6	Falta de tiempo
Descripción	Debido a que una parte del proyecto se realiza durante el primer cuatrimestre y la alumna cuenta con 5 asignaturas a mayores, puede darse el caso de falta de tiempo por exámenes o entregas de prácticas
Probabilidad	Alta
Impacto	Medio
Plan de mitigación	Añadir un sprint a mayores para poder recuperar el tiempo que no se ha podido invertir
Plan de contingencia	Replanificar y/o aumentar el plazo de entrega

Tabla 2.8: Riesgo de falta de tiempo

Riesgo 7	Problemas con el hardware
Descripción	Si hubiera cualquier problema con el hardware, se podría perder el progreso y los cambios realizados en el proyecto
Probabilidad	Baja
Impacto	Medio
Plan de mitigación	Utilizar plataformas de control de versiones como *Git* y tener actualizado el repositorio frecuentemente
Plan de contingencia	Reemplazar el hardware estropeado

Tabla 2.9: Riesgo asociado al hardware

Riesgo 8	Confinamiento por COVID
Descripción	Dada la situación sanitaria actual, puede haber retrasos por confinamiento a causa de la COVID-19
Probabilidad	Media
Impacto	Medio
Plan de mitigación	Sprints añadidos a mayores para lidiar con posibles retrasos
Plan de contingencia	Trabajar un mayor número de horas cuando sea posible, replanificar

Tabla 2.10: Riesgo de confinamiento

Riesgo 9	Funcionalidad limitada a la versión profesional de Vensim
Descripción	Alguna funcionalidad que presenta Vensim está solo disponible en versiones de pago. La versión que utiliza la alumna es la gratuita para estudiantes. Alguna funcionalidad no se puede probar debido a la falta de la licencia profesional, como los *subscripts*.
Probabilidad	Alta
Impacto	Medio
Plan de mitigación	
Plan de contingencia	La UVa cuenta con licencia para el software de Vensim, por lo que se podría acceder a esa funcionalidad exclusiva desde ciertos equipos de la Escuela

Tabla 2.11: Riesgo de funcionalidad limitada Vensim

2.6. Presupuesto simulado

Para calcular el presupuesto asociado a un proyecto debemos fijarnos en los costes de plantilla, gastos generales y los cargos por uso de un suministro [31].

Los **costes de plantilla** incluyen los sueldos de los desarrolladores y otros costes directos, como la contribución por trabajador a la Seguridad Social. Para calcular este coste, se ha consultado el Boletín Oficial del Estado [11] y en las tablas salariales se ha observado que el sueldo de un Programador Junior (categoría E I) a partir del 31/12/2019 es de 15.680,56 € brutos al año y su jornada laboral es de 1.800 horas/año. Contando con un 30 % [42] del salario a mayores para cubrir los gastos sociales, el coste total para la empresa sería 20.384,73 € por trabajador. Sabiendo esto, se puede calcular el coste por horas, que serían aproximadamente 11,32 €.

Teniendo en cuenta los valores obtenidos anteriormente y que el proyecto está estimado

que se desarrolle en un total de 300 horas, el **coste de plantilla** relativo al proyecto sería **3.397,45 €**.

Los **gastos generales** incluyen aquellos costes como el alquiler de las oficinas, la luz y el agua. Normalmente, estos gastos se calculan aplicando un cargo fijo a los departamentos de desarrollo o mediante un cargo porcentual adicional sobre los costos directos de plantilla.

Para este proyecto, los costes generales pueden corresponder al consumo que tiene un portátil y la tarifa de la luz actual. Un ordenador portátil consume como promedio un total de 220Wh [23]. Si la duración del proyecto es de 300 horas, el portátil consumirá en ese periodo 66 kWh. Si el precio medio al día de la luz es 0.11059 €/kWh [63], el precio de consumo de la luz en base a lo que consume un ordenador alcanzará **7,30 €**, siendo estos los **gastos generales** del proyecto.

Durante el desarrollo del proyecto se usará un MacBook Pro de 2019, valorado en 2.699 €. Para calcular el coste que aporta este dispositivo es necesario calcular su depreciación a lo largo del tiempo y tener en cuenta la duración del proyecto. Debemos contar con el **valor inicial** o el precio de su adquisición, su **vida útil** y su **valor residual** o el valor estimado cuando su vida útil haya terminado [45]. Contando con estos tres valores se obtiene la cuota de depreciación anual con la Fórmula 2.1.

$$Cuota\,de\,depreciación\,anual = \frac{valor\,inicial - valor\,residual}{vida\,útil\,en\,años} \tag{2.1}$$

Teniendo en cuenta que la vida útil estimada para el ordenador en cuestión sería de 5 años aproximadamente y su valor residual 100 €, podemos sustituir estos datos en la Fórmula anterior (2.1) y obtendríamos el valor representado en la Fórmula 2.2.

$$Cuota\,de\,depreciación\,anual = \frac{2.699 - 100}{5} = 519,8 \tag{2.2}$$

Se obtiene una depreciación de 519,8 € al año. Teniendo en cuenta que la duración del proyecto son 6 meses, **la depreciación del ordenador** en este periodo es de **259,9 €**.

También se debe contar con los costes asociados a los servicios utilizados, como licencias de software. Para desarrollar el proyecto se necesitan las licencias de *Vensim*, *Astah* y *Visual Paradigm*. Otras tecnologías utilizadas, como *GitLab*, *GitHub*, *Visual Studio Code* y *Overleaf*, no necesitan ninguna licencia para su uso, por lo que su coste asociado es 0 €.

Comenzando con la licencia de *Vensim*, se ha utilizado la versión *PLE* o gratuita, pero con funcionalidad limitada. En un equipo de la Escuela de Ingeniería Informática se cuenta con una licencia para *Vensim DSS*, ya que esta versión aporta funcionalidad con la que se ha trabajado, como *subscripts*, y ha sido necesaria para ejecutar determinados modelos a lo largo del proyecto. El costo de esta licencia es 1995$/persona al año [37], equivalente a 1636,73 € [76]. Como la duración de este trabajo es de 6 meses, la licencia de ***Vensim DSS*** tiene un coste de **818,37 €** para este proyecto.

Se utiliza también la versión *Professional* de *Astah*. Se necesita una licencia para esta versión que conlleva un costo de 40$ al año [2]. Convirtiendo esta cantidad a euros, obtenemos

una licencia de 33,35 € al año [75]. Si la duración del proyecto son 6 meses, la licencia de ***Astah Professional*** para este proyecto son **16,67 €**.

Para el uso de *Visual Paradigm* se utiliza la versión *standard* que tiene un coste de 19$ al mes [53], es decir, 15,85 € al mes [74]. Teniendo en cuenta la duración del proyecto (6 meses) la licencia ***standard de Visual Paradigm*** finalmente tendría un costo de **95 €**.

Entre las licencias de *Visual Paradigm*, *Astah* y *Vensim*, se obtendría un coste total de **930,04 €** relativo a las licencias software requeridas para el desarrollo del proyecto.

Finalmente, en la Tabla 2.12 se recogen todos los costes anteriormente calculados, además del coste total del proyecto. Obteniendo un coste simulado de **4.594,69 €** para este proyecto.

Costes de Plantilla	3.397,45 €
Gastos Generales	7,30 €
Material	259,9 €
Licencias software	930,04 €
Total	**4.594,69 €**

Tabla 2.12: Costes simulados

2.7. Presupuesto real

Este proyecto está remunerado con una beca, la cual tiene una duración de 6 meses con un sueldo de 300 € al mes. Realmente, los **costes de plantilla** del proyecto están compuestos por dicha remuneración, siendo un total de **1.800 €**.

Los **gatos generales** calculados son análogos a los calculados en la Sección 2.6, es decir, **7,30 €**, correspondiendo a la electricidad que consume un portátil durante el periodo de desarrollo de este proyecto.

Los gatos relativos al material con el que se realiza el desarrollo del proyecto siguen teniendo el mismo valor que el calculado en la Sección 2.6, es decir, **259,9 €** por la **cuota de depreciación del ordenador** durante la duración del proyecto.

Las licencias de *Astah*, *Visual Paradigm* y *Vensim* las proporciona la Universidad de Valladolid, por lo que su coste no se tiene en cuenta en el presupuesto real.

Finalmente, el presupuesto real del proyecto es el que se presenta en la Tabla 2.13, alcanzando el valor de **2.067,2 €**.

Costes de Plantilla	1.800 €
Gastos Generales	7,30 €
Material	259,9 €
Licencias software	0 €
Total	**2.067,2 €**

Tabla 2.13: Costes reales

Capítulo 3

Marco teórico

3.1. Dinámica de Sistemas

La dinámica de sistemas [68] es una técnica para **analizar** y **modelar** el comportamiento temporal de los sistemas. Está basada en herramientas extraídas de la ingeniería de control como la simulación por ordenador. Se puede aplicar a sistemas de diferentes áreas como pueden ser sistemas económicos, sociales, tecnológicos, industriales, etc. De esta manera se puede estructurar, a través de modelos matemáticos, la dinámica del comportamiento de estos sistemas [44].

Este tipo de modelos de simulaciones, ayuda a la comprensión de los sistemas complejos y a la toma de decisiones sobre los mismos. Permite analizar y comparar los supuestos y modelos mentales acerca de cómo funcionan las cosas, obtener una **visión cualitativa** sobre el funcionamiento de un sistema, conocer las consecuencias de una decisión o reconocer arquetipos de sistemas disfuncionales en la práctica diaria.

Los modelos permiten simular el impacto de diferentes políticas relativas a la situación a estudiar ejecutando simulaciones que permitirán ver las **consecuencias** a corto y medio plazo, además de ayudar a comprender cómo los cambios en un sistema se ven afectados por el tiempo. Se utiliza en especial para investigar la dependencia de los recursos naturales y los problemas resultantes del creciente consumo a nivel global. Existe una gran variedad de marcas de software en el mercado que ayudan a aplicar esta herramienta como son: Vensim, Stella, ithink, Powersim, y Dynamo, entre otras.

3.2. Vensim

Vensim [69] es una herramienta visual de modelización que permite conceptualizar, documentar, simular, analizar y optimizar modelos de dinámica de sistemas [15].

Vensim es el software elegido por miles de analistas, consultores e investigadores de todo el mundo debido a que integra en un solo entorno un poderoso conjunto de herramientas que permiten desarrollar, probar, interpretar y distribuir modelos [39]. Provee una forma simple y flexible de construir modelos de simulación mediante diagramas de influencias y diagramas de Forrester.

Cuando se realiza un modelo en Vensim, se genera un archivo con extensión *.mdl*. Este tipo de archivo se puede abrir con el entorno gráfico de Vensim o con un editor de texto. Leído como texto plano se encuentran tres partes bien diferenciadas. Comienza con la definición de todas las funciones y constantes creadas por el usuario, seguido de la inicialización de los parámetros de control de la simulación y finalmente, cuenta con una serie de caracteres alfanuméricos útiles para la representación del modelo en el entorno gráfico.

Una sentencia en Vensim habitualmente tiene la estructura representada en el Fragmento de Código 3.1

```
1 Nombre =
2     Ecuacion
3     ~ Unidades y limites
4     ~ Comentario
5     |
```

Fragmento de código 3.1: Ejemplo de sentencia Vensim

Se distinguen 5 elementos: nombre de la variable, ecuación que describe el comportamiento de la variable, unidades y límites y un comentario describiendo la variable. Las unidades, límites y comentario son opcionales. Dichos elementos se encuentran separados por un igual y dos virgulillas, finalizando la sentencia con una barra lateral.

Un ejemplo de una variable real sería el representado en el Fragmento de Código 3.2, donde se presenta una constante con nombre *Characteristic Time* cuyo valor es 10. Se mide en minutos con límites [0, inf) y está acompañada de un comentario.

```
1 Characteristic Time =
2     10
3     ~ Minutes [0.0, inf]
4     ~ How long will it take the teacup to cool 1/e of the way to
        equilibrium?
5     |
```

Fragmento de código 3.2: Ejemplo real de sentencia Vensim

3.2.1. Estructuras específicas de Vensim

En Vensim se pueden presentar en un modelo gran variedad de símbolos [35]. Los más útiles para este proyecto y que aparecerán a lo largo de la memoria son los explicados a continuación.

- **Subscripts**

  Los subscripts [34] son unas estructuras de datos específicas de Vensim que permiten representar en una sola variable más de un valor, equivalente a una matriz. Un posible ejemplo de subscript se presenta en el siguiente Fragmento de Código 3.3, donde se define un subscript de países.

```
1       paises : MEXICO, USA, CANADA ~~|
```

Fragmento de código 3.3: Ejemplo de Subscript

Los subscripts se dividen en dos partes, el nombre y los valores de un subscript. La primera parte se llama *subscript name* y es el nombre con el que se identifica al subscript. En el ejemplo anterior el *subscript name* es "paises". Además, contienen otra segunda parte la cual se encarga de dotar de valor al subscript name y se denomina *subscript values* o los valores del subscript, que en este ejemplo serían "MEXICO, USA y CANADA", 3 *subscript values*.

Para poder utilizar los subscripts, es necesario que se incluyan entre corchetes ([ ]), tanto el *subscript name* como los *subscript values*, después del nombre de una variable. Una variable puede contener un máximo de 8 subscripts separados por comas (,). En el Fragmento de Código 3.4 se puede apreciar el uso del anterior subscript definido.

```
1       nacimientos [paises] =
2           poblacion[paises] * factor de natalidad[paises]
3           ~ personas / año
4           ~ Total de nacimientos en un país específico.
5           |
```

Fragmento de código 3.4: Ejemplo de uso de Subscript

3.2.2. Compatibilidad de Subscripts

Los *subscripts*, definidos anteriormente, cuentan con una serie de operaciones asociadas para su manejo. Las más relevantes para este proyecto se explican a continuación.

- **Subscripts numeric range (Rango numérico de Subscripts)**

  Normalmente, los subscripts se definen como se muestra en el Fragmente de código 3.3. Sin embargo, en algunos modelos es necesario trabajar con un gran número de *subscript values* y el lenguaje Vensim presenta un atajo para poder definir un subscript a partir de rangos numéricos, sin tener que escribir todos los elementos que conforman el subscript. Un ***numeric range*** o rango numérico [70] es una representación simplificada para definir varios *subscript values* similares entre sí asociados a un subscript. En un rango numérico solo se especifican los límites (inferior y superior) de los *subscript values*,

pero finalmente el subscript se compone de todos los valores encontrados dentro de ese rango. Estos límites deben estar formados por la misma cadena de texto seguida de un número que define el inicio y fin del rango. Se presenta un ejemplo en el Fragmento de Código 3.5 que equivale a realizar una definición de un subscript como en el Fragmento de Código 3.6.

```
1     upper :
2         (Layer1 - Layer4)
```

Fragmento de código 3.5: 1. Subscript Numeric Range

```
1     upper :
2         Layer1, Layer2, Layer3, Layer4
```

Fragmento de código 3.6: 1. Subscript equivalente

Además, Vensim permite que se defina un subscript a partir de varios rangos y saltándose algunos elementos, como ocurre en el ejemplo presentado en el Fragmento de Código 3.7. En él se define el subscript "*lot*" a partir de dos rangos numéricos y un *subscript value* aislado entre ambos. Su sentencia equivalente, sin utilizar rangos numéricos, es la presentada en el Fragmento de Código 3.8.

```
1     lot : (LOT1-LOT3),LOT12,(LOT14-LOT16)
```

Fragmento de código 3.7: 2. Subscript Numeric Range [70]

```
1     lot : LOT1,LOT2,LOT3,LOT12,LOT14,LOT15,LOT16
```

Fragmento de código 3.8: 2. Subscript equivalente [70]

- **Subscript mapping**

Trabajando con subscripts, puede surgir la necesidad de utilizar diferentes familias de subscripts pero que posean el mismo significado. Para ello, se utiliza **subscript mapping** [33]. El mapeo de subscripts consiste en la asociación entre dos o más *subscript names* para poder utilizar dichos nombres de los subscripts indistintamente en una misma ecuación.

Para ello, se debe escribir en una sentencia Vensim el nombre del subscript que se quiere mapear, seguido de dos puntos " : " y los *subscript values*, como normalmente se define un subscript. Seguido, se añade " -> " (símbolo del *subscript mapping*) y el *subscript name* que se quiera mapear. El número de elementos entre ambos subscripts debe coincidir para que se pueda realizar el mapeo elemento a elemento.

```
1     Quality [product] = work quality[worker type] ~~|
2
```

```
3       product : clay, plastic, wood ~~|
4       worker type : wtclay, wtplastic, wtwood -> product ~~|
```

Fragmento de código 3.9: Subscript mapping

En el Fragmento de Código 3.9 se presenta un ejemplo del uso de susbscript mapping. Normalmente, una ecuación como la de la 1ª línea generaría un error en el entorno Vensim, ya que el subscript que aparece a la izquierda de la ecuación no coincide con el que se utiliza en la derecha (*product* y *worker type*). Sin embargo, realizando el mapeo entre ambos subscripts (línea 4), esa excepción se vería solucionada, ya que significa que cada *product* es elaborado por un solo *worker type*. En la ecuación de la primera línea, se expresa que la calidad de un *product* es igual a la calidad del trabajo del *worker type* asociado, como por ejemplo, la calidad del producto arcilla (en inglés *clay*) sería igual a la calidad del trabajo del trabajador de la arcilla. Esto se debe a que se ha creado un mapeo entre el subscript que representa a los productos y el subscript que representa al tipo de trabajador y, por consiguiente, se ha creado un mapeo entre los *subscript values* de ambos. Es decir, *wtclay* se mapea con *clay*, *wtplastic* con *plastic* y *wtwood* con *wood*, relación uno a uno entre los distintos rangos de los subscripts.

Para que el mapeo sea correcto es necesario que el orden y el número de los *subscript values* coincidan, en este ejemplo se trata de 3 elementos en cada subscript.

Cuando un subscript cuenta con muchos valores, el mapeo de subscripts se hace indispensable. Gracias al mapeo de subscripts (" -> ") se necesita una sola ecuación para poder mapear todos los valores entre los dos subscripts, independientemente del número de valores que tuviesen los subscripts. Sin el mapeo, deberíamos definir tantas ecuaciones como valores de subscripts hubiera.

Cabe destacar, que en el Fragmento de Código 3.9 se ha determinado un mapeo del subscript *product* a *worker type* y no a la inversa. Esto implica que el *subscript name product* o los *subscript values* por los que está formado puedan ser utilizados a la izquierda de una ecuación y el *subscript name worker type* o sus *subscript values* se utilicen a la derecha, es decir, solo se establece el mapeo en una única dirección.

- **Subscript copy**

A veces, es necesario que un conjunto de Subscripts se mapeen entre ellos. Suele ser útil cuando se desea utilizar dos veces un mismo subscript en dos variables diferentes. Esto se consigue con **subscript copy**, representado por el símbolo <->. Mediante este símbolo se permite copiar dos subscripts y, posteriormente, permite utilizarlos indistintamente. El *subscript copy* tiene el mismo resultado que si se utilizase el *subscript mapping* en ambos sentidos sobre dos subscripts.

```
1       Task : Clear, Dig, Build ~~|
2
3       Ptask <-> Task ~~|
```

Fragmento de código 3.10: Subscript copy

En el ejemplo presentado en el Fragmento de Código 3.10, se copian los Subscripts *Ptask* y *Task*. Obteniendo dos variables que contienen los mismos *subscript values*.

3.2.3. Funciones Vensim

Vensim cuenta con una amplia lista de funciones. Las más relevantes para este proyecto y sobre las que se ha trabajado son las presentadas a continuación.

- **RANDOM 0 1**

  Se trata de una función que genera un número aleatorio en el rango de 0 a 1 [65]. Actualmente, esta función está obsoleta y se suele utilizar *RANDOM UNIFORM (0,1,0)* en su lugar, que posee el mismo funcionamiento. *RANDOM 0 1* se mantiene para evitar problemas de compatibilidad con versiones anteriores.

  En el Fragmento de código 3.11 se representa una sentencia que utiliza la función *RANDOM 0 1*. Se ha definido la variable *"White noise"* cuyo valor depende del que devuelva la función *RANDOM 0 1* menos 0,5.

```
1       White noise = RANDOM 0 1() - 0.5
2       ~~ |
```

Fragmento de código 3.11: Random 0 1

- **SAMPLE IF TRUE**

  La función *SAMPLE IF TRUE* [71] suele ser utilizada como una variable de nivel en los modelos Vensim. Cuenta con tres parámetros: una condición, un valor y un valor inicial. Esta función devuelve el valor del segundo parámetro introducido cuando la condición es cierta y, además, almacena ese valor. Si la condición no se satisface, *SAMPLE IF TRUE* mantiene constante el último valor con el que se ha cumplido la condición.

  El valor inicial, tercer parámetro de la función, permite únicamente inicializar el valor almacenado antes de que se compruebe por primera vez la condición.

  Esta función se utiliza como "retenedor" de un evento particular durante el tiempo que dure la simulación.

  En el Fragmento de código 3.12 se presenta el uso de la función *SAMPLE IF TRUE*. En la línea 1 se define la variable *"Workforce"*, cuyo valor oscila entre 0 y 50 debido a la distribución uniforme que proporciona la función *Random Uniform*. En la línea 6 se define la variable *"Max workforce"* que almacena el valor máximo que toma la fuerza de trabajo durante la ejecución del modelo. Para ello, se utiliza la función *SAMPLE IF TRUE*. *"Max workforce"* se inicializa con el valor de la variable *"Workforce"* (3^{er} parámetro de la función *Sample If True*). En cada *step* o iteración del modelo en tiempo de ejecución se evalúan ambas variables. El primer parámetro de la función *Sample If True* es la condición que determina si la variable *"Max workforce"* se actualiza con el valor del segundo parámetro, *"Workforce"*, o si se devuelve el valor que está almacenado de iteraciones anteriores. Por lo que el valor que almacena la variable *"Max workforce"* se actualizará cuando el valor de *"Workforce"* sea mayor que el máximo almacenado.

```
Workforce=
    RANDOM UNIFORM(0,50,0)
    ~   people
    ~           |

max workforce=SAMPLE IF TRUE(
    Workforce > max workforce, Workforce, Workforce)
    ~   people
    ~   retains the maximum value of workforce
    |
```

Fragmento de código 3.12: Sample If True

Posiblemente, esta función puede llegar a crear confusión con la función *IF THEN ELSE*, ya que ambas cuentan con los mismos parámetros. Sin embargo, la principal diferencia radica en el valor devuelto. *IF THEN ELSE* puede devolver uno de los dos parámetros con los que cuenta, mientras que *SAMPLE IF TRUE* puede devolver el valor almacenado o el valor actual que se pasa como parámetro. Es decir, la desemejanza entre *SAMPLE IF TRUE* e *IF THEN ELSE* es que la primera permite almacenar valores siempre que se cumpla la condición, y la función *IF THEN ELSE* siempre devolverá uno de dos los valores que tenga por parámetro además de que no almacenará ningún valor.

3.3. PySD

Simulating System Dynamics Models in Python (PySD) [40] es un proyecto *open source* alojado en la plataforma GitHub iniciado por **James Houghton** y **Michael Siegel**. Fue un proyecto creado en 2013, con el objetivo de crear una librería en Python para poder ejecutar modelos de dinámica de sistemas, con el propósito de mejorar la integración del *Big Data* y *Machine Learning* en el flujo de trabajo de dinámica de sistemas. La principal razón del proyecto es la inmensa cantidad de herramientas computacionales que se están desarrollando en la comunidad científica de datos en general. Los dinamistas de sistemas deberían usar directamente las herramientas que otras personas están construyendo, en lugar de replicar su funcionalidad en software específico de dinámica de sistemas.

PySD traduce los archivos de modelos de **Vensim** o **XMILE** en módulos de **Python** y proporciona métodos para modificar, simular y observar esos modelos traducidos. Sin embargo, aún la librería no es capaz de traducir correctamente algunos modelos más complejos.

Al tratarse de un proyecto *open source*, PySD cuenta con muchos contribuidores. Los más significativos para este proyecto son: **James Houghton**, creador de PySD, el usuario **julienmalar**, **DVRodri8** (Diego Rodrigo Verdugo) y **enekomartinmartinez**. La última versión de **DVRodri8** es un fork del proyecto de **julienmalar**, añadiendo aportaciones para lograr la traducción del proyecto MEDEAS, modelo desarrollado en Vensim capaz de simular el efecto que tendría en el ecosistema y en el sector energético las diferentes decisiones políticas. **Enekomartinmartinez** ha realizado cambios en la versión de **DVRodri8**, añadiendo

mejoras y documentación. Finalmente, con la versión final de **enekomartinmartinez**, se han *mergeado* esos cambios a la rama *master* de PySD, la cual ha servido como base en este proyecto.

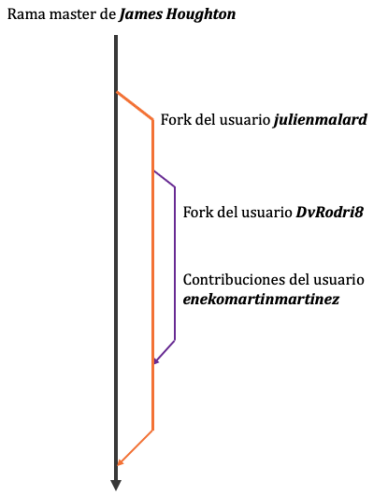

Figura 3.1: Ramas principales de PySD

3.4. Parsimonious

Parsimonious [9] es el analizador sintáctico que se utiliza en PySD. Se basa en analizar gramáticas de expresión [64] también conocidas como **PEG** (Parsing expression grammar). Son un tipo de gramáticas formales analíticas, es decir, describen un lenguaje formal en términos de un conjunto de reglas para reconocer cadenas en el lenguaje. Fue introducido en 2004 por *Bryan Ford* y se encuentra estrechamente relacionado con la familia de lenguajes de análisis de arriba hacia abajo. Los PEG no distinguen entre lectura y análisis, todo se realiza al mismo tiempo. Además, una gran característica con la que cuentan, es la no ambigüedad, es decir, si una cadena se analiza, tiene exactamente un árbol de análisis válido. Por lo tanto, con este tipo de analizadores, la ambigüedad se resuelve dando siempre el primer reconocimiento exitoso.

Parsimonious fue diseñado para ser un analizador rápido, con un uso moderado de RAM y que sus gramáticas fueran legibles y extensibles.

Para utilizar *parsimonious* como gramática primero se debe comenzar definiendo la gramática deseada. A ésta se le dota de cierta lógica con el **patrón visitor** asociado a cada regla. Cada vez que se cumple una regla, se ejecuta el método *visitor* que ésta tenga asignado. Posteriormente, se utiliza la subsclase **nodes.NodeVisitor** para recorrer el árbol y poder realizar las funciones deseadas sobre él.

3.4.1. Gramática

En la Tabla 3.1 se muestran las principales reglas de sintaxis de Parsimonious.

3.4.2. Patrón visitor

El patrón *Visitor* o Visitante es una forma de separar la lógica u operaciones de la estructura de datos sobre la que opera [5]. En ocasiones, las estructuras de datos son muy complejas y requieren que se realice operaciones sobre ellas. Esas operaciones pueden ser muy variadas y se pueden desarrollar nuevas a medida que la aplicación crece. Esto puede dar lugar a que la estructura de datos junto a sus operaciones sea muy complejo y de un tamaño considerable, lo que dificulte su compresión y modificación.

Debido a esta razón, el patrón de diseño *Visitor* propone la separación de las operaciones en clases independientes llamadas Visitantes. De esta forma se permite definir una operación sobre objetos de una clase sin llegar a modificar las clases sobre las que opera. Es un patrón muy útil cuando existen varias clases de objetos con interfaces diferentes y se desea realizar operaciones que dependen de la cada clase concreta. También se utiliza cuando existen diversas operaciones en una jerarquía de objetos y no es conveniente recargar las clases con estas operaciones. Es conveniente utilizar este patrón cuando las clases de la jerarquía no cambian pero se añaden con una mayor frecuencia operaciones a la estructura.

La estructura típica de este patrón de diseño se ilustra en la Figura 3.2.

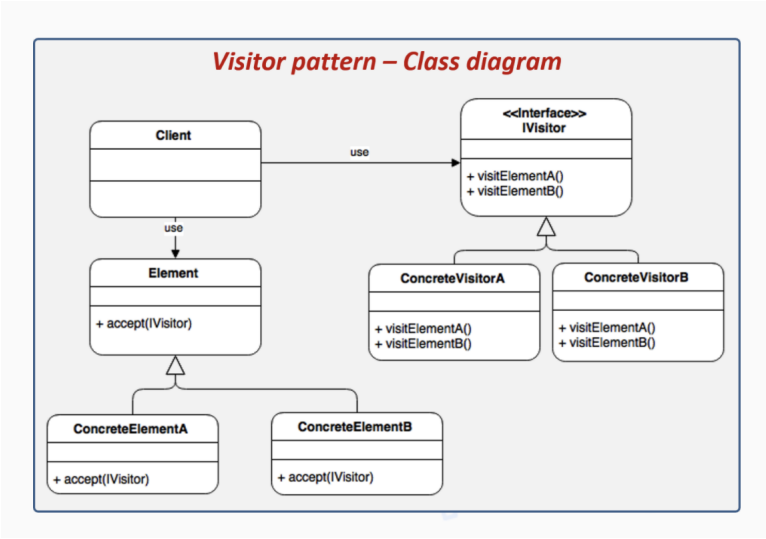

Figura 3.2: Estructura del patrón Visitor. [5]

Los componentes que forman este patrón son:

- **Cliente**: interactúa con la estructura (*element*) y con el Visitor. Es el encargado de crear los visitors y los envía al elemento para su procesamiento.

- **Element**: es la raíz de la estructura, sobre la que se utiliza el Visitor. Este objeto, por lo general, es una interfaz que define el método **accept** y que deben implementar todos los objetos de la estructura.

- **ConcreteElement**: representa un hijo de la estructura. La estructura completa puede estar compuesta por varios *ConcreteElement* y cada uno debe implementar el método **accept**.

- **IVisitor**: interfaz que define la estructura del visitor. Ésta deberá tener un método por cada *element* que se quiera analizar.

- **ConcreteVisitor**: representa la implementación del *Visitor*. Realiza una operación sobre un *element* concreto.

Para poder entender mejor las relaciones y la interacción entre las clases se ha proporcionado en la Figura 3.3 un diagrama de secuencia sobre cómo se relacionan las clases en el patrón Visitor.

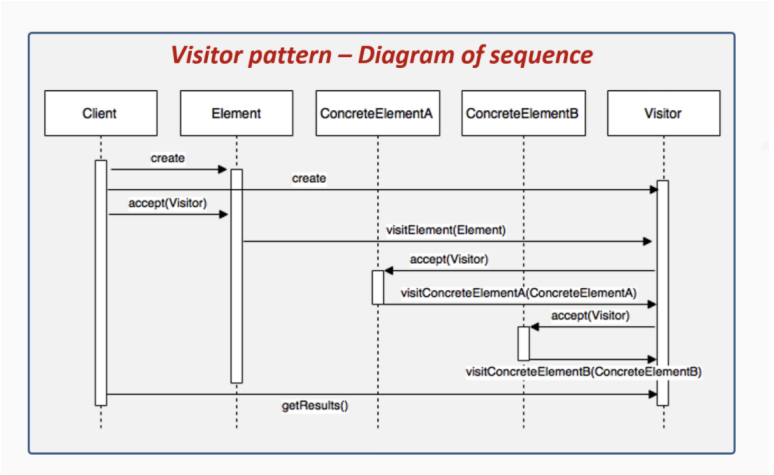

Figura 3.3: Diagrama de secuencia del patrón Visitor. [5]

Primero, el *cliente* crea la estructura (*element*) y la instancia del *visitor* asociado. Cuando el cliente ejecuta el método *accept* de un elemento le pasa como argumento el *visitor*. *Element* envía un método al *visitor* pasándose a sí mismo como parámetro para poder distinguir la operación asociada a ese *element* concreto. Cada *visitor* deberá tener un método para cada tipo de *element*.

3.4.3. Patrón visitor en Parsimonious

El patrón Visitor, como se ha visto anteriormente, permite aportar nueva funcionalidad sin modificar el código. En Parsimonious esto se utiliza para aportar funcionalidad a las diferentes reglas de la gramática. Para ello, se crea una clase que herede de **NodeVisitor** y se crea en ella todos los métodos correspondientes a la gramática que Parsimonious visitará.

En Parsimonious, se agrega un *visitor* por cada regla de la gramática, es decir, se distingue entre diferentes instancias de un mismo objeto no entre diferentes objetos.

Se presenta la siguiente gramática en Parsimonious, Fragmento de código 3.13, que define un dato como nombre o DNI y, a su vez, define nombre y DNI [60].

```
dato = nombre / dni
nombre = ~"[a-zA-Z]"
dni = ~"[0-9]+" ~"[A-Z]"IU
```

Fragmento de código 3.13: Gramática Simple Parsimonious

Se le asocia una lógica a cada regla heredando de la clase **NodeVisitor** y definiendo una función con nombre: **visit_*nombre-de-la-regla***, como se muestra en el Fragmento de código 3.14.

```
class logicaEjemplo(NodeVisitor):
    def visit_nombre(self, nodo, hijos):
        # Aquí lógica asociada a nombre

    def visit_dni(self, nodo, hijos):
        # Aquí lógica asociada a DNI

    def generic_visit(self, nodo, hijos):
        # Lógica de cualquier otro elemento
```

Fragmento de código 3.14: Lógica Parsimonious

Finalmente, cuando se parsee un texto con Parsimonious y se satisfaga la regla "nombre", visitará la función *visit_nombre* y si confronta con dni, se ejecutará la función *visit_dni*. La función *generic_visit* es útil cuando se quiera tratar reglas de manera especial.

Cabe destacar, como se ha presentado en el Fragmento de Código 3.14, cada método *visitor* tiene tres parámetros. Estos son:

- Self: primer argumento de los métodos de instancia en Python. Corresponde a una regla de nomenclatura del lenguaje Python [20]. Este parámetro hace referencia a la instancia actual de la clase y se utiliza para acceder a las variables que pertenecen a la clase.

- Nodo: estructura de tipo *Node* (clase que se encuentra en el módulo *parsimonious.nodes*). Representa el nodo que se está visitando en el árbol de análisis.
- Hijos: lista que contiene los resultados de visitar a los nodos hijos del nodo actual.

3.5. Patrón Decorator

El patrón *Decorator* [22], es un patrón de diseño estructural que permite añadir funcionalidad a un objeto dinámicamente. También conocido como *Wrapper*, proporciona una alternativa flexible a las subclases permitiendo extender la funcionalidad.

Cuando se tiene que añadir o modificar el comportamiento de un objeto tendría cabida utilizar la herencia entre clases. Sin embargo, hay que tener en cuenta que la herencia es estática y esto no permite alterar la funcionalidad de un objeto durante el tiempo de ejecución. Se puede únicamente sustituir el objeto completo por otro creado [56]. Además, una gran limitación se ocasiona cuando las subclases solo pueden tener una única clase padre, que ocurre en la mayoría de lenguajes.

Al contrario que la herencia, el patrón *decorator* permite añadir un comportamiento a un objeto individual, sin afectar a los demás objetos de la misma clase, es decir, dinámicamente. Un *wrapper* es un objeto que se vincula a otro objeto. El *wrapper* cuenta con los mismos métodos que el objeto al que se vincula y le delega todas las solicitudes. Sin embargo, no tienen exactamente el mismo comportamiento ya que el *wrapper* altera el resultado antes o después de pasar la solicitud al objeto.

En la Figura 3.4 se presenta un ejemplo de un diagrama de clases y de secuencia utilizando el patrón *Decorator*. Utilizando este patrón se pueden agregar responsabilidades a un objeto de forma dinámica y ampliar la funcionalidad en tiempo de ejecución [73]. Se crean objetos *decorators* separados que permiten agregar responsabilidades a un objeto ya existente.

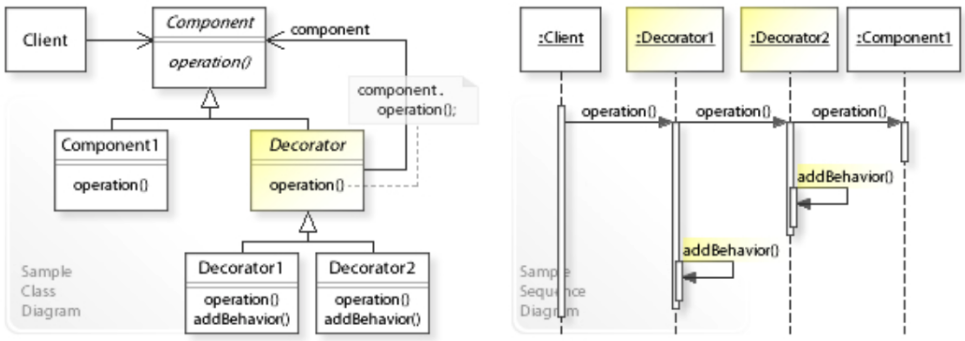

Figura 3.4: Diagrama de clases y de secuencia: Patrón *Decorator* [73]

A través de estos objetos *decorator* se puede ampliar la funcionalidad de un objeto en tiempo de ejecución. En la Figura 3.4 se define una clase *Decorator* que implementa la interfaz

Component de forma trasparente y así, el cliente solo conoce e interacciona con dicha interfaz. La clase *Decorator* mantiene una referencia a un objeto *Component* y le reenvía las solicitudes a este componente. Posteriormente, las subclases de *Decorator* (*Decorator1* y *Decorator2*) añadirán funcionalidad adicional que se realizarán antes y/o después de reenviar la solicitud al componente.

Así es como este patrón crea una clase decoradora, que envuelve la clase original y proporciona la funcionalidad adicional esperada manteniendo intactos los métodos de la clase. Además, se pueden anidar varios *decorators* de forma recursiva y no hay restricciones sobre las posibles combinaciones a la hora de agregar responsabilidades con *decorators*. En la Figura 3.4, en el diagrama de secuencia cabe destacar el anidamiento del *Decorator2* y *Decorator1* para alterar la funcionalidad del objeto *Component*.

El uso de este patrón puede ser más eficiente que la creación de subclases, ya que de esta manera el comportamiento de un objeto puede aumentarse sin tener que definir un objeto completamente nuevo.

En Python, el uso del patrón *decorator* es bastante común. La funcionalidad que se quiere agregar o el comportamiento que se quiere modificar se escribe en una función. Posteriormente, donde se quiera añadir el comportamiento adicional se escribe el nombre de la función decoradora con el carácter @ delante y encima de la función a la que se le quiere añadir funcionalidad. En PySD se utiliza este patrón para implementar una caché de dos niveles, más adelante se explica en detalle.

Regla	Descripción
“literal”	Se utiliza para que el texto que aparezca entrecomillado se considere literalmente.
[espacio]	Para definir espacios o tabuladores en las reglas.
a/b/c	Alternativas. El primero en tener éxito de *a, b o c* gana
thing?	Expresión opcional, es decir, “*thing*” se consume si es que existe.
&thing	Afirmación anticipada. Asegura que “*thing*” coincida con la posición actual pero no se consume.
!thing	Negativa. Coincide si “*thing*” no se encuentra en la posición actual, no consume ningún token.
thing*	Cero o más tokens. Asegura que “*thing*” aparezca cero o más veces en la posición actual y consume tantos tokens como haya.
thing+	Uno o más tokens. Asegura que “*thing*” aparezca una o más veces en la posición actual y consume tantos tokens como haya.
(thing)	Los paréntesis se utilizan simplemente para agrupar.
∼ r“regex” ilmsux	Las expresiones regulares se definen con el símbolo ∼ delante y se citan como literales. Se puede utilizar “r” para tratar el texto de la expresión regular como crudo sin tener que escapar caracteres, aplicable también a la primera regla. Después de la expresión regular se pueden utilizar los caracteres *i, l, m, s, u y x* que facilitan el trabajo. **“i”**: *ignore_case*, no distingue entre mayúsculas y minúsculas. **“l”**: *locale*, hace que los caracteres {\w, \W, \b, \B} dependan de la configuración local. **“m”**: *multiline*, los caracteres ^ y $ corresponden al principio y al final de cada línea en vez de al string. **“s”**: el carácter “.” sustituye al carácter de nueva línea. **“u”**: *unicode*, hace que los caracteres {\w, \W, \b, \B} dependan de la codificación de caracteres. **“x”**: *verbose*, permite insertar comentarios dentro de la expresión regular seguidos del carácter #.

Tabla 3.1: Reglas de sintaxis

Capítulo 4

Ingeniería Inversa

En este capítulo se expondrán los conocimientos sobre PySD tras realizar el proceso de ingeniería inversa sobre él. Para conocer más detalles sobre el mismo es necesario que se comprendan las 5 gramáticas que posee PySD y la función de cada una en el proyecto, además de la estructura del proyecto, mediante un modelo 4+1 vistas que ha sido el más conveniente para poder comprender toda la funcionalidad que encierra PySD.

4.1. Gramáticas de PySD

PySD tiene dos fases principales, la traducción del fichero Vensim a un fichero Python equivalente, y posteriormente, la ejecución del fichero Python creado. La traducción de PySD es compleja, debido a que no cuenta con un parser "tradicional". En PySD, no se ha creado una única gramática donde se encuentren todas las reglas correspondientes a Vensim con sus clases *visitors*, si no que PySD cuenta con un total de 5 gramáticas para poder traducir un archivo Vensim. Estas son: **file_structure_grammar**, **model_structure_grammar**, **component_structure_grammar**, **expression_grammar** y **lookup_grammar**, cuyo funcionamiento y razón se explican a continuación.

4.1.1. file_structure_grammar

Es la primera gramática que analiza el código Vensim. Se encarga de separar las Macros de Vensim y el código principal. Una Macro [38] permite representar una estructura de modelo sin tener que volver a escribir ecuaciones. Primero, la macro se define escribiendo ecuaciones y se invoca de la misma forma que cuando se llama a una función. Se debe encontrar definida antes de usarla.

Esta gramática como entrada tiene el archivo **.mdl** convertido a una cadena de texto. Devuelve una lista de diccionarios, en la que cada diccionario representa una sección, una

macro o el *main* (código principal) del archivo pasado como parámetro. A su vez, cada diccionario se compone de varios elementos. Estos son:

- ***returns***: lista de strings. Representa lo que contiene una macro o está vacía cuando se trata del *main*.
- ***params***: lista de string. Representa los parámetros que recibe una macro o está vacía si se trata del *main*.
- ***name***: es un string. Contiene el nombre de la macro o "main" cuando se trata del *main* del modelo.
- ***string***: string. Contiene el código Vensim.

Si se trata del código principal del modelo, el diccionario generado se compondrá por: en el elemento **name**, el nombre *main* y las listas de **returns** y **params** se encontrarán vacías. El código Vensim se encontrará en **strings**.

Se toma como ejemplo el código Vensim del Fragmento de código 4.1 que consta de 3 sentencias Vensim [60]. En la línea 1, se define un subscript, "países", con 3 subscripts values cuya unidad es país y el comentario asociado, listado de países. Posteriormente, se define la variable "prueba" utilizado el subscript "países" cuyo valor es el valor de la función, en este caso el máximo de 3 valores. La última sentencia, comienza en la línea número 7, solo aporta un comentario útil para el desarrollador.

```
1 países: italia, francia, alemania
2     ~ país
3     ~ listado de países |
4 prueba[países] = MAX(1, MAX(2, 3))
5     ~ unidades
6     ~ comentario |
7 .Control ~
8 Simulation Control Parameters |
```

Fragmento de código 4.1: Ejemplo código Vensim

Si se parsea el código Vensim previo (4.1) con la gramática **file_structure_grammar**, ésta generaría un diccionario como el representado en el Fragmento de código 4.2. El ejemplo de código Vensim no contiene ninguna Macro, por lo que, como se ha comentado anteriormente, las listas *returns* y *params* están vacías, además de asignar a *name* el valor "main". El código de la sentencia Vensim se almacena en el elemento *string*.

```
1 [{
2     'returns' : [],
3     'params' : [],
4     'name' : 'main',
5     'string' :  'países: italia, francia, alemania
6                     ~ país
```

```
7                       ~ listado de países |
8                   prueba[países] = MAX(1, MAX(2, 3))
9                       ~ unidades
10                      ~ comentario |
11                  .Control ~
12                  Simulation Control Parameters |'
13  }]
```

Fragmento de código 4.2: Salida de ***file-structure-grammar***

4.1.2. model_structure_grammar

Model_structure_grammar es la segunda gramática que parsea el código Vensim en PySD. Se encarga de parsear un string que representa el código principal del modelo y organiza su contenido. Separa ecuaciones de secciones y de comentarios, etiquetándolos de manera diferente. La principal distinción entre ecuaciones y comentarios radica en que los comentarios no afectarán posteriormente a la ejecución, ya que su única función es aportar información para los programadores del modelo.

Como salida, esta gramática proporciona una lista de diccionarios, en el que cada diccionario contiene los componentes de un elemento del modelo separados en ecuación, unidades, documentación, límites y tipo. Más en detalle, en cada diccionario se encuentran las siguientes *keys*:

- ***doc***: documentación, comentarios asociados a la sentencia.
- ***eqn***: parte de la sentencia que contiene la ecuación.
- ***units***: las unidades con las que se opera.
- ***lims***: valores entre los que se acotan las unidades de la ecuación.
- ***kind***: tipo de la sección: comentario o ecuación.

En la Sección 3.2 se explican las diferentes secciones con las que cuenta una sentencia de Vensim. Resumidamente, estas son: ecuación, unidades y comentario. La gramática ***model_structure_grammar*** se encarga de separar la sección de unidades en unidades y límites de éstas, ya que se pueden agregar intervalos entre los cuales la variable tome valores.

Si partimos del ejemplo del Fragmento de código de ejemplo (4.1) y parseamos la 2ª sentencia presentada (línea 4 a 6) obtendríamos como salida el diccionario representado en el Fragmento de código 4.3. Este diccionario almacena en la key *'doc'* el comentario de la sentencia, en la key *'eqn'* la ecuación, en *'units'* las unidades con las que se opera en la ecuación y en la key *'limits'* el rango de valores entre los que puede tomar valor la variable. Finalmente, el diccionario cuenta con la key *'kind'* a la que se le asigna el valor *'entry'* debido a que se trata de una ecuación.

```
[{
    'doc' : 'comentario',
    'eqn' : 'prueba[países] = MAX(1, MAX(2,3))',
    'units' : 'unidades',
    'limits' :  '[1,3]',
    'kind' : 'entry'
}]
```

Fragmento de código 4.3: Salida de ***model-structure-grammar*** Ecuación

En el ejemplo previo se ha presentado una sentencia correspondiente a una ecuación. Sin embargo, si se tratase de una sentencia que pertenezca a un comentario, la gramática *model_structure_grammar* generaría un diccionario similar al presentado en el Fragmento de código 4.4, cuyos elementos *'eqn'*, *'units'* y *'limits'* se encuentran vacíos. En el campo *'doc'* se almacena el texto del comentario en cuestión y se indica que es un comentario asignando el valor *'section'* al elemento *'kind'*.

```
[{
    'doc' : 'aquí el texto del comentario',
    'eqn' : '',
    'units' : '',
    'limits' :  '',
    'kind' : 'section'
}]
```

Fragmento de código 4.4: Salida de ***model-structure-grammar*** Comentario

4.1.3. component_structure_grammar

La tercera gramática con la que cuenta PySD, *component_structure_grammar*, parsea únicamente las ecuaciones, es decir, aquellas sentencias que poseen en su diccionario el valor *'entry'* en el campo *'kind'* que se ha asignado previamente en la gramática *model_structure_grammar*. Además, no parsea todas las entradas del diccionario creado en la gramática anterior, si no que solo parsea el elemento *'eqn'*. En resumen, la gramática *component_structure_grammar* divide la cadena que representa la parte de la ecuación de un elemento del modelo.

La función en la que se define esta gramática toma como entrada dos parámetros:

- ***equation_str***: la ecuación.
- ***root_path***: ruta raíz del archivo Vensim, necesaria para resolver rutas de archivos de datos externos.

El objetivo de esta gramática es dividir la parte izquierda de la ecuación en *real_name* y *subs* y la parte derecha en *'expr'*. Generando un diccionario con los siguientes elementos:

- ***'real_name'***: nombre.
- ***'subs'***: subscripts si incluye.
- ***'expr'***: parte derecha completa.
- ***'kind'***: puede ser *component*, *lookup*, *subdef* o *data*, dependiendo del tipo de la ecuación.

Cabe destacar que se asocia un nuevo tipo al elemento *'kind'*, mientras que anteriormente se había definido *'kind'* con el valor *'entry'* para ecuaciones. Esto se debe a que la función para la que está el elemento *'kind'* es conducir cada sentencia por diferentes caminos durante la ejecución.

El elemento *'kind'* (en la gramática *component_structure_grammar*) depende del tipo de la ecuación pudiendo obtener 4 valores:

- **component**: una expresión de modelo normal o una constante.
- **lookup**: una definición de lookup.
- **subdef**: una definición de subscript.
- **data**: una variable de datos.

Para poder comprender mejor esta gramática, se toma el ejemplo de antes, presentado en el Fragmento de código 4.1. Si solo se analizan las ecuaciones, de este ejemplo solo se analizaría las dos líneas presentadas en el Fragmento de código 4.5. Nótese que los comentarios y unidades de la sentencia Vensim ya no se consideran.

```
1 países: italia, francia, alemania
2
3 prueba[países] = MAX(1 ,MAX(2, 3))
```

Fragmento de código 4.5: Entradas de ***component-structure-grammar***

Los diccionarios creados para cada una de estas ecuaciones serían los presentados en el Fragmento de código 4.7 para el subscript definido en la línea 1 (4.5) y en el Fragmento de código 4.6 para la variable "prueba", línea 3 (4.5). La primera ecuación se etiqueta como *subdef* (4.6) debido a que es la definición de un *subscript* y la segunda como *component* (4.7) debido a que se trata de la definición de una variable.

```
1 {
2     'real_name': 'países',
3     'subs': [],
4     'expr': 'italia, francia, alemania',
5     'kind': 'subdef'
6 }
```

Fragmento de código 4.6: Salida de ***component-structure-grammar*** **Subscript**

```
{
    'real_name': 'prueba',
    'subs': [países],
    'expr': 'MAX(1 ,MAX(2, 3))',
    'kind': 'component'
}
```

Fragmento de código 4.7: Salida de ***component-structure-grammar*** **Component**

Cabe destacar que, en adelante, las ecuaciones etiquetadas como *subdef* se añadirán a un diccionario de subscripts y no se continúan analizando.

4.1.4. expression_grammar

Expression_grammar es la gramática más amplia de PySD. Solo toma como entrada aquellos elementos etiquetados como *'component'* o como *'data'*. Parsea una expresión normal. Más en concreto parsea la parte derecha de la ecuación y se encarga de generar la traducción adecuada para el código en Python.

En esta gramática se comprende la importancia de la separación de las gramáticas, debido a que se debe recurrir a variables fuera de la clase *visitor* asociadas a las reglas para poder completar la traducción. A la función que incluye la gramática *expression_grammar*, se le proporciona como parámetros: el elemento donde se encuentra la ecuación a traducir, el *namespace*, el diccionario de subscripts (*subscript_dict*), la lista de macros (*macro_list*) y el diccionario de elementos subscripts, que contiene los nombres y valores de los subscripts en Python.

Esta gramática devuelve una traducción de la parte derecha de la ecuación compatible con Python. Devuelve un diccionario con los siguientes elementos:

- La traducción de la ecuación en Python.
- Un *'kind'*, cuyo valor puede ser *component* si se trata de una función o *constant* de una constante.
- Una lista de estructuras, que servirán para la posterior ejecución del modelo traducido.

Retomando el ejemplo de código Vensim previo, para la parte derecha de la ecuación presentada en el Fragmento de código 4.7, es decir, **MAX(1, MAX(2, 3))**, la gramática *expression_grammar* la traduciría como se muestra en el Fragmento de código 4.8.

```
np.maximum(1, np.maximun(2, 3))
```

Fragmento de código 4.8: Traducción de la ecuación

Los valores que se asignan en esta gramática al elemento *'kind'* servirán para etiquetar las sentencias traducidas con un nivel diferente de caché.

4.1.5. lookup_grammar

Al igual que la gramática *expression_grammar* que solo analiza las expresiones etiquetadas anteriormente como *component* o *data*, la gramática *lookup_grammar* solo analiza las expresiones etiquetadas como *lookup*.

Al igual que la gramática anterior, ésta parsea solo el lado derecho de la ecuación creando una traducción adecuada en Python, aunque solo traduce las sentencias lookups.

La razón para separar las dos últimas gramáticas reside en la similitud de las reglas y tokens que parsean las sentencias y la dificultad que se encontraría al intentar juntar ambas gramáticas y poder lograr que funcionen correctamente.

4.1.6. include_common_grammar

Cabe destacar el uso de la función *include_common_grammar*. Cuya finalidad es presentar una serie de reglas generales que se anexan a las otras 5 gramáticas presentadas anteriormente. Define una gramática (*common_grammar*) con reglas comunes, como nombres, caracteres de escape o espacios.

Para poder adjuntar estas reglas a las demás gramáticas, se llama a la función *include_common_gramar*, pasándole como argumento las reglas definidas en cada gramática dónde se quieran anexar estas reglas comunes. En las 5 gramáticas explicadas anteriormente se pasan como argumento a esta función para así poder utilizar las reglas definidas en *common_grammar* y evitar la duplicidad de código.

4.2. Modelo de Vistas 4+1

El Modelo de Vistas 4+1 [10], diseñado por Philippe Krutchen, describe la arquitectura de sistemas software, utilizando para ello múltiples vistas concurrentes. Las diferentes vistas que se presentan en este modelo permiten entender por separado los intereses de los diferentes *stakeholders* de la arquitectura, como pueden ser: usuario final, desarrolladores, *project managers*, etc.

La arquitectura de software se ocupa de la abstracción, descomposición y composición, el estilo y la estética del sistema. Para poder describir la arquitectura de un software, se describe un modelo compuesto por múltiples vistas o perspectivas. Este modelo se constituye de cinco vistas principales: vista lógica, vista de desarrollo, vista de proceso, vista física y los escenarios o casos de uso. La relación entre las diferentes vistas se presenta en la Figura 4.1. En las secciones posteriores se explicará más en profundidad cada una de las vistas que componen este modelo.

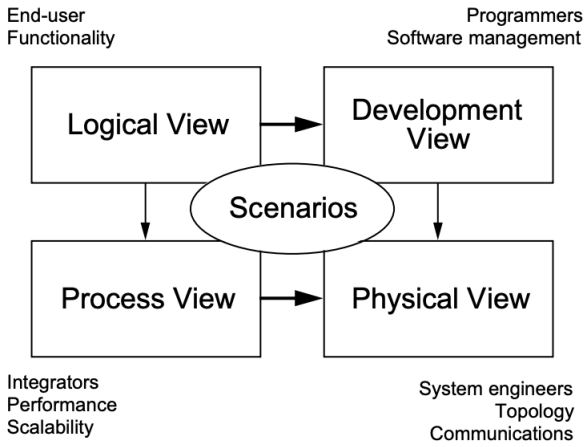

Figura 4.1: El Modelo de Vistas 4+1

4.2.1. Vista de desarrollo

La vista de desarrollo presenta una descomposición del sistema. Esta vista se centra en la organización del módulo empaquetado en pequeños fragmentos, ya bien sean bibliotecas de programas o subsistemas. Ilustra el sistema con la perspectiva del programador. La arquitectura de desarrollo tiene en cuenta los requisitos internos relacionados con la facilidad de desarrollo, gestión de software, reutilización y las posibles limitaciones que puede haber debido al lenguaje de programación o a las herramientas escogidas.

Los diagramas de componentes UML se suelen utilizar para representar la vista de desarrollo, ya que permiten describir los diferentes componentes del sistema. También, esta vista se puede representar mediante diagramas de paquetes.

Para representar la arquitectura de los fragmentos que componen *pysd* se ha utilizado un diagrama de paquetes, Figura 4.2. En él, se ha representado la estructura jerárquica de paquetes que compone PySD exceptuando las relaciones entre sus módulos, las cuales se muestran en otro diagrama aparte para una mayor claridad. En la Figura 4.3 se encuentran representadas las relaciones entre los diferentes módulos.

En primer lugar, se puede observar como **pysd** es el paquete principal que incluye a los demás y a toda la lógica del sistema. Es necesario diferenciar entre el paquete **pysd** y el módulo **pysd**. Este último permite al usuario interactuar con el sistema y posee las funciones adecuadas para que el usuario pueda traducir un modelo *Vensim* o *XMILE* y/o ejecutar la traducción generada en *Python*. Sin embargo, el proceso de traducción de modelos *XMILE* no es objetivo de estudio de este proyecto, por lo que en los posteriores diagramas no se detallará ese subsistema. Como es lógico, tras obtener la traducción de los modelos, ya sean modelos de *Vensim* o de *XMILE*, el proceso de ejecución de la simulación es el mismo para ambos ya que se parte del archivo *Python* obtenido.

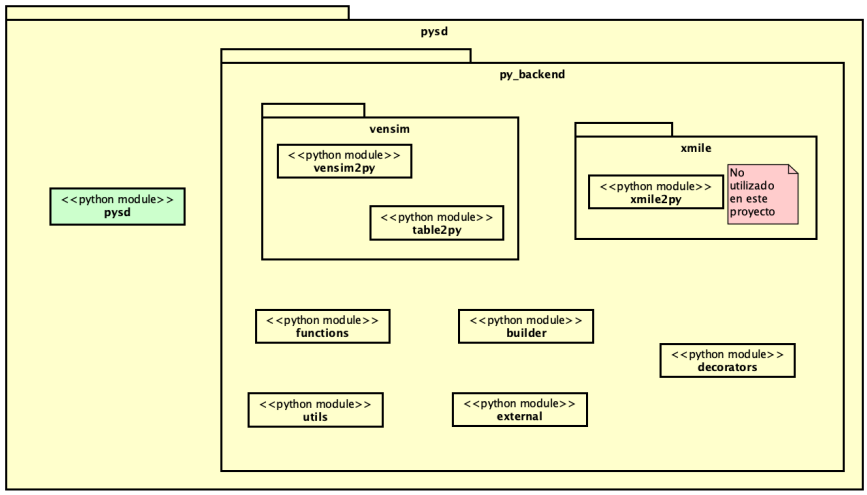

Figura 4.2: Vista de desarrollo PySD

El módulo **pysd** interactúa con los módulos que contiene el paquete **py_backend**, módulos necesarios para la traducción y ejecución de modelos Vensim que posteriormente se explicarán individualmente en detalle. A su vez, el paquete **py_backend** contiene dos subpaquetes: **vensim**, el cual engloba los módulos principales que contienen la lógica necesaria para poder traducir modelos *Vensim*, y **xmile**, para aquellos archivos *XMILE*.

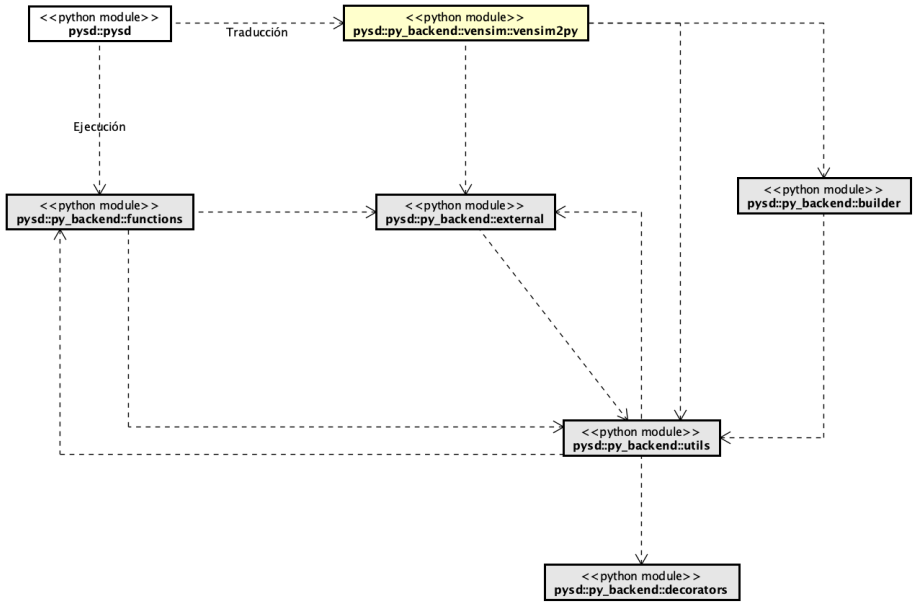

Figura 4.3: Relaciones entre módulos de PySD

En la Figura 4.3 se presentan las relaciones entre los principales módulos de PySD que se encargan de realizar la traducción de archivos Vensim. Para mayor claridad, se ha presentado cada módulo con el nombre del paquete al que pertenece, aunque se encuentra desarrollado más sencillo visualmente en el diagrama de la Figura 4.2.

Como antes se ha comentado, **pysd** es el módulo con el que interactúa el usuario. **Pysd** se encarga de traducir el modelo *Vensim* a un archivo *Python* interactuando con el módulo **vensim2py**, el cual crea la correcta traducción del modelo *Vensim*. Para este proceso de traducción, **vensim2py** interactúa y utiliza las funciones de los módulos: **external**, **utils** y **builder**. Por otra parte, para poder realizar el proceso de ejecución del modelo una vez ya traducido, el módulo **pysd** interactúa con el módulo **functions**.

4.2.2. Vista lógica

En la arquitectura lógica se definen y describen principalmente los requisitos funcionales del sistema, aquellos que el sistema debe brindar en términos de servicios a sus usuarios. En esta vista se utiliza un estilo orientado a objetos. Los diagramas UML con los que se suele representar esta vista son los diagramas de clases, diagramas de comunicación o diagramas de secuencia.

A continuación, se detalla la función de cada uno de los módulos de PySD, así como aquellas funciones más importantes y sobre las que se ha profundizado en este proyecto.

Cabe destacar, antes de comenzar la explicación, que en los diagramas en los que ha sido necesario, se han detallado aquellos parámetros de entrada con la notación *in*, de salida con la notación *out* o los que han sido modificados con *inout* en los métodos de las clases, porque el lenguaje Python permite devolver más de un parámetro en una única función y esto complica su representación en UML. Además, se ha señalado mediante anotaciones en los diagramas los diferentes tipos que pueden llegar a ser algunos parámetros, debido al tipado dinámico con el que cuenta Python. De igual manera, se ha añadido alguna anotación indicando las clases en las cuales se ha omitido el parámetro *self*, para contar con una mayor claridad en los diagramas.

En la Figura 4.4, se presentan los módulos: **pysd**, **vensim2py** y **table2py** en detalle. El módulo **pysd** se encuentra en el paquete *pysd*, mientras que los módulos **vensim2py** y **table2py**, en el paquete *vensim* (Figura 4.2).

Pysd

Pysd es el módulo *Python* de PySD que permite interactuar al usuario con el proyecto. En él se encuentran las funciones necesarias para que se pueda crear la traducción en *Python* de un archivo *Vensim*. La función **read_vensim** toma como parámetro un modelo Vensim en formato texto y lo convierte en una instancia de la clase **Model**, clase que se encuentra en el módulo **functions**. Así mismo, **pysd** cuenta con la función **load**, la cual permite generar a partir de un modelo Python una instancia de la clase **Model**, lo que permitirá poder

ejecutar y realizar la simulación de dicho modelo. La función **load** es utilizada internamente por **read_vensim**.

Table2py

El módulo **table2py** no se ha utilizado en este proyecto. Sin embargo, contiene una única función, **read_tabular**, que permite leer un modelo Vensim en forma de tabla, ya bien sea *csv*, *tab* o *xlsx*, y transformarlo a una instancia de la clase **Model**.

Vensim2py

En la Figura 4.4, además de los dos módulos comentados anteriormente, también se encuentra el módulo que contiene la lógica principal de PySD y el cual realiza el flujo principal de la traducción de un modelo Vensim, **vensim2py**. En él se encuentran definidas las cinco gramáticas y sus clases asociadas que permiten parsear y obtener la información de un modelo Vensim.

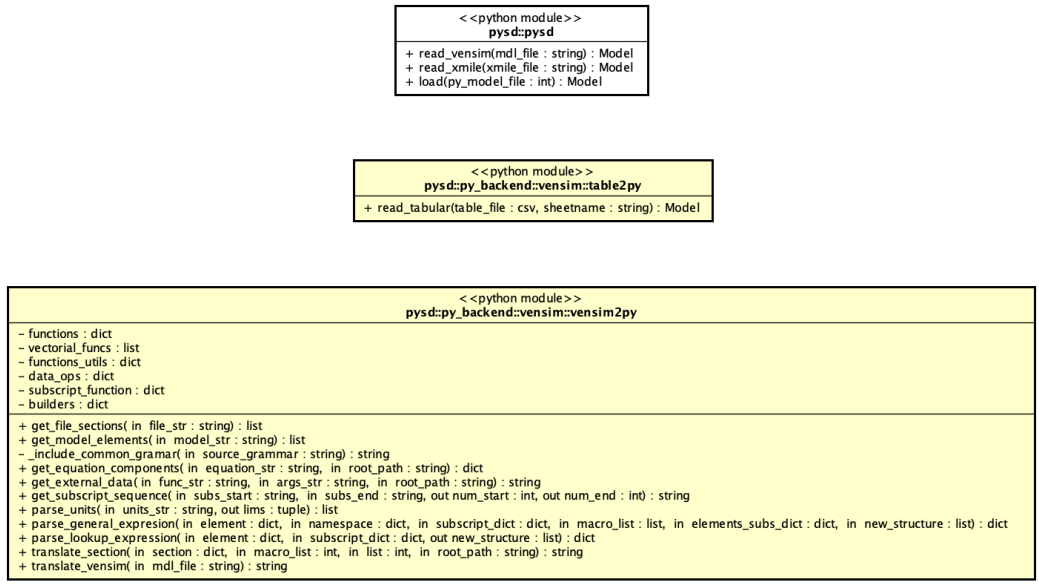

Figura 4.4: Módulos principales PySD

La primera función del módulo **vensim2py**, que además se utiliza en la función **read_vensim** del módulo **pysd**, es **translate_vensim**. Ésta se encarga de comenzar la traducción. En ella, se parsea el modelo con la primera gramática de PySD, **file_structure_grammar** a través de la función **get_file_sections**. Como se ha comentado al explicar las gramáticas que componen PySD, esta gramática permite separar el modelo en secciones: la sección principal y las *macros*. Con cada sección obtenida se llama posteriormente a la función **translate_section**.

Así mismo, las funciones **get_model_elements**, **get_equation_components**, **parse_general_expression** y **parse_lookup_expression**, contienen las 4 gramáticas restantes con las que cuenta PySD: *model_structure_grammar*, *component_structure_grammar*, *expression_grammar* y *lookup_grammar*, respectivamente. Además, justo después de cada función se definen las clases descendientes de *NodeVisitor* de cada gramática para que se pueda realizar y recorrer el árbol de análisis.

Se puede destacar la función **_include_common_grammar**, que contiene la gramática explicada en la Sección 4.1.6. Contiene las reglas gramaticales básicas comunes utilizadas por todas las demás gramáticas.

Debido a la complejidad del módulo **vensim2py**, ya que cuenta con las cinco funciones en las que se definen las gramáticas de PySD y sus clases NodeVisitors asociadas, en la Figura 4.5 se han representado, sin entrar en detalle, las clases que contiene **vensim2py**. Estas clases son: FileParser, ModelParser, ComponentParser, ExpressionParser y LookupParser. Asociadas a las 5 gramáticas de PySD: *file_structure_grammar*, *model_structure_grammar*, *component_structure_grammar*, *expression_grammar* y *lookup_grammar*, respectivamente. A su vez, estas clases heredan de la clase *NodeVisitor*, lo que les proporciona un marco de inversión de control para recorrer un árbol y devolver una nueva construcción basada en él.

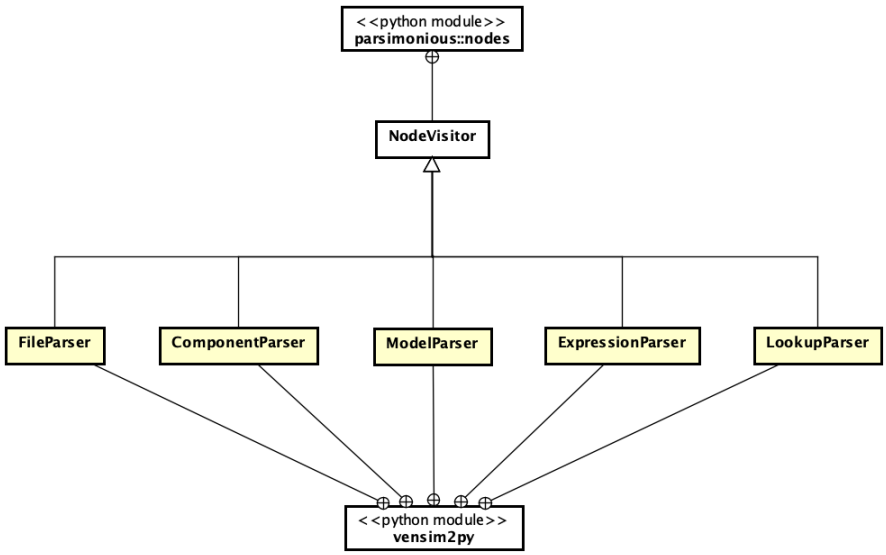

Figura 4.5: Gramáticas de **vensim2py** simplificadas

En las Figuras 4.6 y 4.7 se representan los diagramas con detalle de las clases asociadas a las gramáticas.

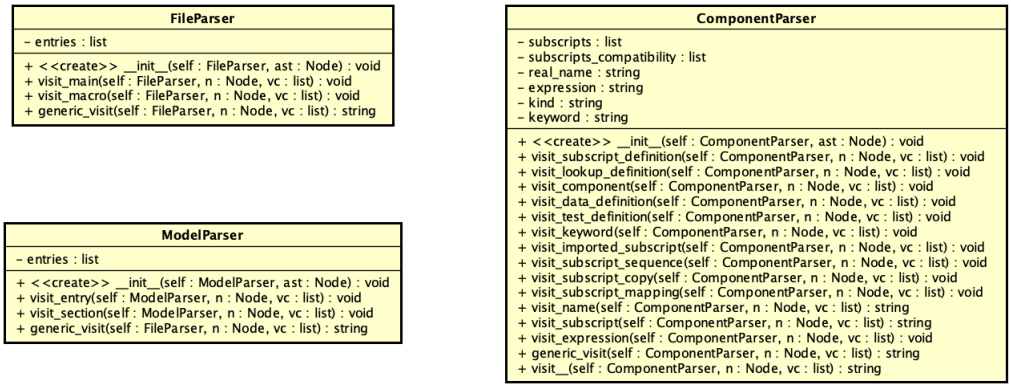

Figura 4.6: Clases asociadas a las gramáticas del módulo **vensim2py**

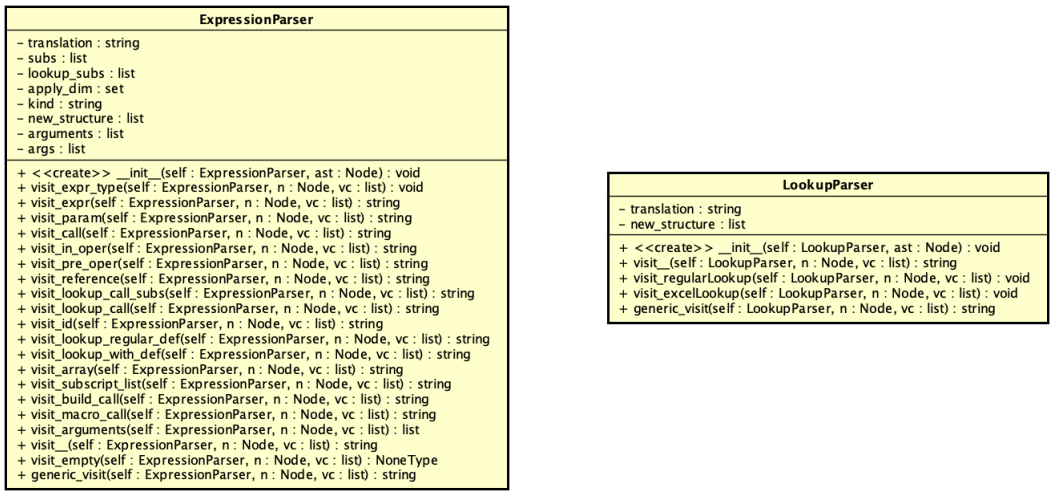

Figura 4.7: Clases asociadas a las gramáticas del módulo **vensim2py**

Los métodos presentados en cada clase se corresponden con todos los métodos *visitors* asociados a las diferentes reglas gramaticales. No hay un método *visitor* por cada regla, si no que tienen métodos *visitors* asociados solamente aquellas reglas en las que es útil almacenar cierta información del modelo que está siendo parseado. Dentro de los *visitors* se guarda esta información en los atributos de la clase, para posteriormente, cuando la gramática haya terminado de parsear los elementos del modelo, se pueda devolver el resultado a partir de esos atributos. Y a partir de esta información que se ha obtenido con ayuda de los *visitors* se creará el modelo en *Python*.

Los métodos *visitors* asociados a las gramáticas siempre tienen tres parámetros, explicados en la Sección 3.4.3. *Self*, que representa a la instancia actual de la clase; *n*, de tipo *Node*, es el nodo que se está siendo visitado del árbol de análisis y *vc* (*visit children*), que se

trata de una lista con todos los resultados de recorrer los nodos hijos que forman la sentencia que ha concordado con la regla gramatical. A partir de los valores que componen el tercer parámetro, *vc*, de los métodos *visitors*, se completa la información que se almacena en los atributos de la clase, como anteriormente se ha comentado. Posteriormente, con los atributos se rellenará el diccionario que se devuelve como resultado obtenido en cada gramática.

Una vez terminado el módulo **vensim2py**, se explican a continuación los demás módulos que conforman *PySD*.

Functions

El módulo **functions** está representado en la Figura 4.8. Es uno de los más importantes de *PySD*, ya que cuenta con las clases que se instanciarán en el modelo Pyhton traducido y con la lógica necesaria para poder ejecutar la simulación. En dicha figura se presentan las clases que se definen en él y las relaciones entre éstas. Posteriormente, se detallan aquellas clases que son más relevantes para este proyecto.

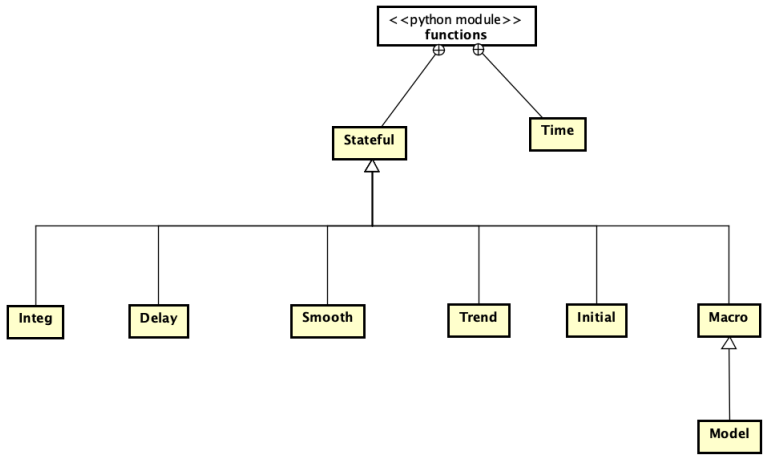

Figura 4.8: Módulo **functions** y sus clases

En la Figura 4.9, se presenta el detalle del módulo **functions** y la clase *Time* que se define en él. En este módulo encontramos muchas de las funciones que se utilizan en Vensim pero con la lógica correspondiente en Python, por ejemplo: *PULSE*, *IF THEN ELSE*, *RANDOM UNIFORM*, etc. La clase **Time** es la encargada de representar el tiempo durante la simulación. Con el atributo t representa el tiempo actual que se va modificando a medida que avanza la simulación y mediante el atributo *step* se representa el aumento de tiempo que se incrementa en cada iteración.

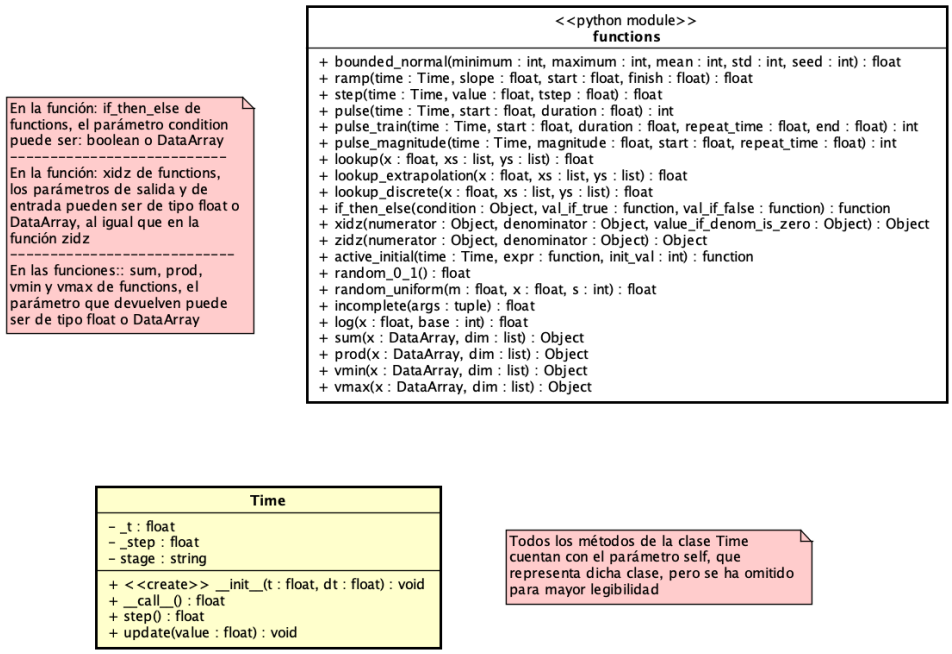

Figura 4.9: Módulo **functions** y clase **Time**

En la Figura 4.10 se presenta el diagrama con detalle de las clases **Stateful**, **Integ**, **Macro** y **Model** definidas en el módulo *functions*. Como se ha representado en la Figura 4.8, la clase **Stateful** es una de las clases más importantes de este módulo, ya que, excepto **Time**, todas las demás clases heredan de ella. Esta clase permite representar la evolución de estado de ciertos elementos del modelo, recreando el proceso de simulación en Vensim. Para ello, cuenta con un atributo *state*, que simula el estado de los elementos.

La clase **Integ** permite simular los *stocks* de Vensim. Recibe y almacena un valor inicial y la función de la que se obtiene la derivada necesaria para poder realizar la integración.

La clase **Model** almacena toda la información que respecta al código principal del modelo (ya traducido). A una instancia de esta clase se le llama modelo de pysd, ya que es la representación que se hace en el lenguaje Python del archivo Vensim. Es decir, la clase **Model** implementa una representación de objetos con estado del sistema y contiene la mayoría de métodos para poder acceder y modificar los componentes del modelo. Además, esta clase es la encargada de instanciar el tiempo dependiendo de las variables del modelo para el mismo y también se encarga de llevar a cabo la simulación utilizando la integración por Euler. De esta clase se puede destacar la función **initialize** la cual inicializa la simulación del modelo, o la función **run** que permite simular el comportamiento del modelo a medida que avanzan los instantes de tiempo o *steps*. La función **_euler_step** permite realizar la integración por Euler en un solo paso, utilizando para ello el estado que tienen los elementos *Stateful* y actualizándolo.

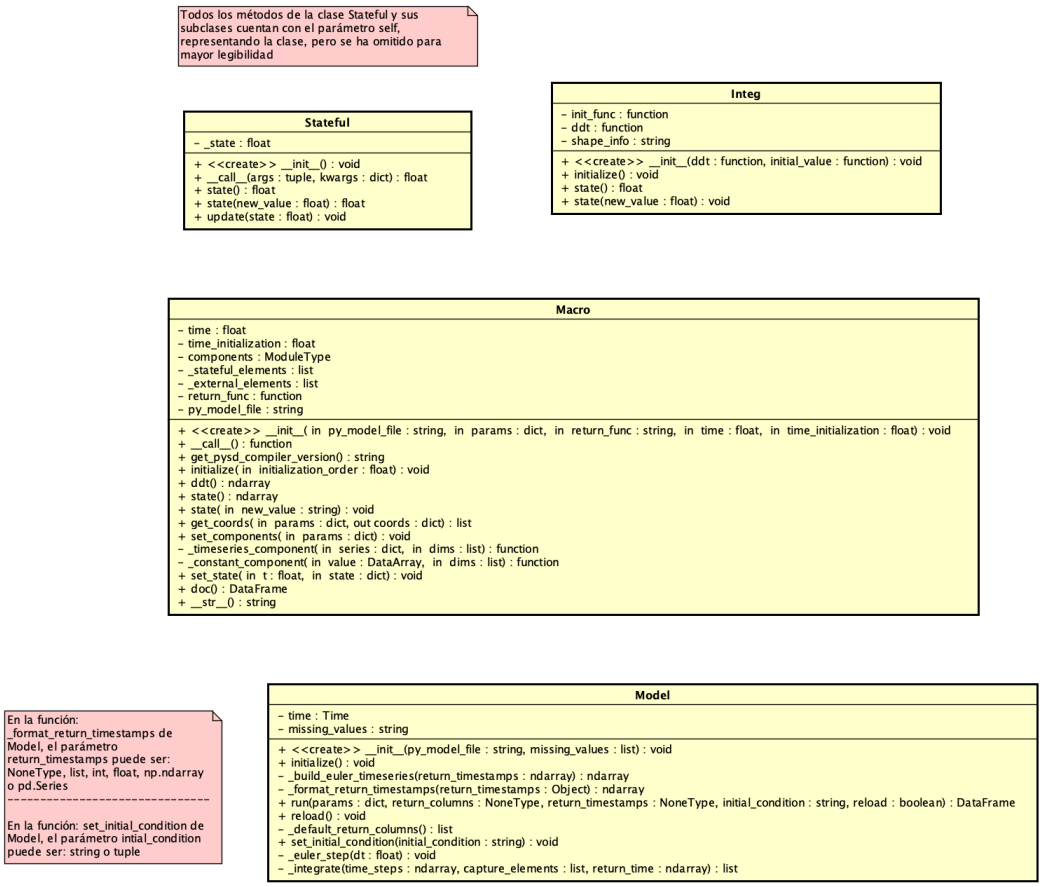

Figura 4.10: Clases **Stateful**, **Integ**, **Macro** y **Model** del módulo *functions*

Model hereda de la clase **Macro** (Figura 4.8). Ésta implementa la lógica para representar las macros de Vensim, encargándose de obtener aquellos objetos *Stateful* que hayan sido creados en la fase de traducción e inicializarlos para, posteriormente, obtener sus derivadas y los resultados de la ejecución. **Model** realiza las mismas funciones que **Macro**, pero **Model** es el objeto raíz del modelo por lo que tiene más métodos agregados que facilitan la ejecución.

Builder

Continuando con el módulo **builder**, éste se representa en detalle en la Figura 4.11. En este módulo no se define ninguna clase, pero es el encargado de realizar el texto del modelo en Python completando con los resultados obtenidos de la traducción. Tiene el código necesario para ensamblar en un modelo pysd todos los elementos que hayan sido traducidos tanto de Vensim o XMILE y formar, a partir de estos, una versión compatible con Python.

La principal función de este módulo es **build**, la encargada de construir y escribir la

representación en Python del modelo. Se llama desde el módulo **vensim2py**, después de haber terminado todo el proceso de traducción del modelo *Vensim*. Como parámetros se le pasan los diferentes elementos del modelo que han sido parseados, subscripts, namespace y el nombre del archivo donde se debe escribir el resultado de la representación en Python. Esta función contiene ciertas líneas de texto fijo que siempre se escriben en los modelos creados, como la versión de PySD, o *imports* pero luego hay ciertas líneas que se completan con la traducción generada anteriormente en el módulo **vensim2py** las cuales se le pasan a la función por parámetro.

En este módulo también se encuentra la función **build_function_call** que se menciona en el Seguimiento del proyecto (Sección 6.6). A esta función se le llama desde los visitors de la gramática *expression_grammar* y permite crear la expresión completa en Python de una función de Vensim, pasándole el nombre de la función y los argumentos.

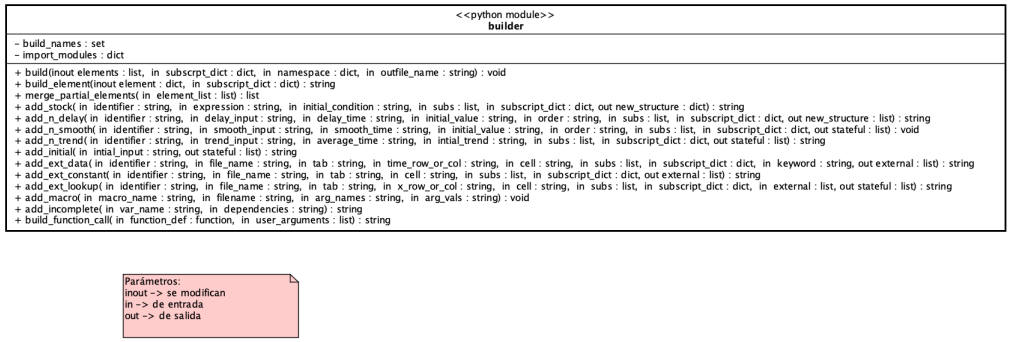

Figura 4.11: Módulo **builder**

Utils

El diagrama que presenta en detalle del módulo **utils** se encuentra en la Figura 4.12. La principal función de **utils** es fusionar en un solo módulo todas aquellas funciones de gran utilidad para el proyecto. Muchas de estas funciones se usan varias veces a lo largo del flujo de traducción. De hecho, este módulo, como se ha presentado en la Figura 4.3, es utilizado por los módulos **builder**, **functions**, **external** y **vensim2py**. A su vez, en **utils** se importan aquellos nombres accesibles de los módulos **decorators**, **external** y **functions** para poder definir una lista con aquellos nombres que ya han sido usados y que tienen un significado particular en el modelo que se esté traduciendo.

De este módulo se puede destacar la función **add_entries_undescore**, mencionada en la Sección 6.5 dedicada al seguimiento del proyecto. Dicha función se encarga de intercambiar en los nombres de las funciones los espacios por guiones bajos, ya que en Vensim se pueden escribir indistintamente.

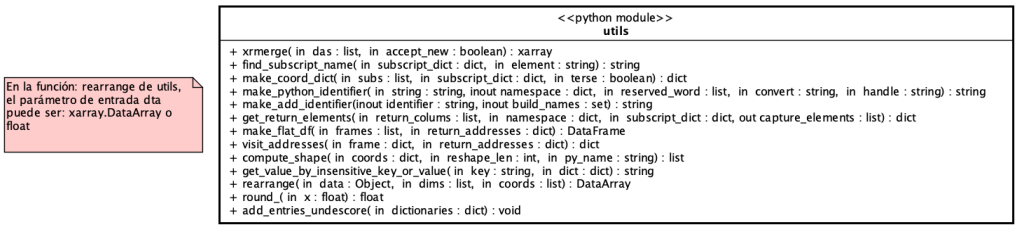

Figura 4.12: Módulo **utils**

External

En la Figura 4.13 se presenta el diagrama del módulo **external** y las clases definidas en él. Es en este módulo donde se definen varias clases cuya razón principal es leer datos externos. El principal propósito del módulo **external** se trata de unificar, en un solo archivo, todas las herramientas necesarias para leer datos de archivos externos.

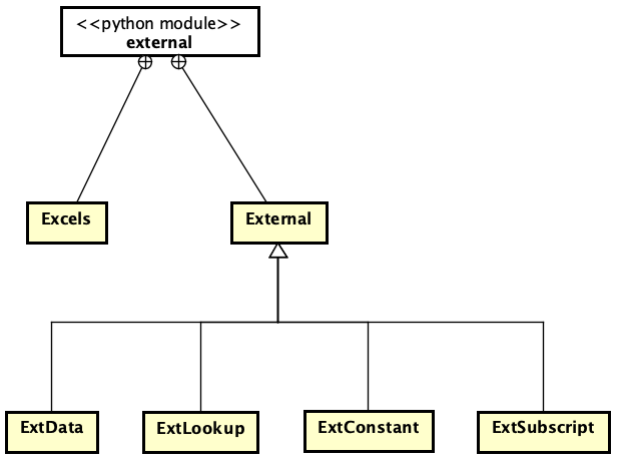

Figura 4.13: Módulo **external** y sus clases

En la Figura 4.14 se muestran los diagramas con detalle de las clases **External** y **Excels**. La clase principal de este módulo es la clase **External**, de la que heredan las demás exceptuando la clase **Excels**. La clase **External** permite almacenar cierta información como el nombre del fichero que se lee y los datos que contiene ese fichero.

La clase **Excels** se encarga de leer archivos *Excel* y almacenar información de los mismos, para evitar que dichos archivos sean leídos más de una vez, poniendo en práctica el patrón *Singleton*.

En la Figura 4.15 se presentan todas las clases pertenecientes al módulo *external* y que a su vez, heredan de la clase anteriormente comentada, **External**.

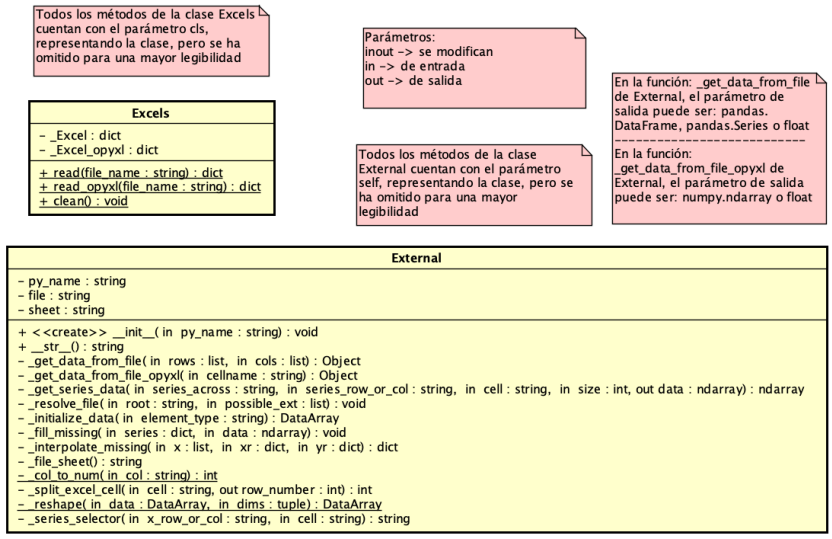

Figura 4.14: Clase **Excels** y **External** del módulo **external**

Existen diferentes sentencias en Vensim que permiten obtener datos de ficheros externos que se utilizan como variables en un modelo Vensim. El conjunto de estas funciones que tienen soporte en *PySD* son las que se presentan a continuación. Para obtener los datos de las sentencias Vensim *GET XLS DATA* y *GET DIRECT DATA*, se encuentra la clase **ExtData**. A su vez, para las sentencias *GET XLS LOOKUPS* y *GET DIRECT LOOKUPS*, la clase **ExtLookup**. Para las funciones *GET XLS CONSTANT* y *GET DIRECT CONSTANT*, la clase **ExtConstant** y, finalmente, para las sentencias *GET XLS SUBSCRIPT* y *GET DIRECT SUBSCRIPT*, se cuenta con la clase **ExtSubscript**.

Estas expresiones crean una nueva instancia de la clase **External**, donde se almacena la información necesaria para presentar las estructuras de datos necesarias. Dichas instancias de la clase **External** se inicializan antes de los objetos *Stateful*.

En la Figura 4.16 se presenta el módulo **decorators**. Para poder entender mejor el funcionamiento y la razón de este módulo se debe conocer el patrón *Decorator* y su uso en Python, explicado en la Sección 3.5.

En PySD se implementa una especie de caché de dos niveles para poder hacer que la ejecución sea lo más rápida posible. La caché se implementa mediante el uso de decoradores. En la fase de traducción se etiqueta a cada función con uno de los dos tipos de caché. Se añade el decorador *cache.run* para aquellas funciones cuyo valor es constante durante toda la ejecución independientemente del tiempo de la simulación. De esta manera, solo se calcula su valor una única vez en toda la ejecución. Se etiquetan con el decorador *cache.step* aquellas funciones que necesitan tener diferentes valores a lo largo de la ejecución y cambiar su valor dependiendo del tiempo de la simulación.

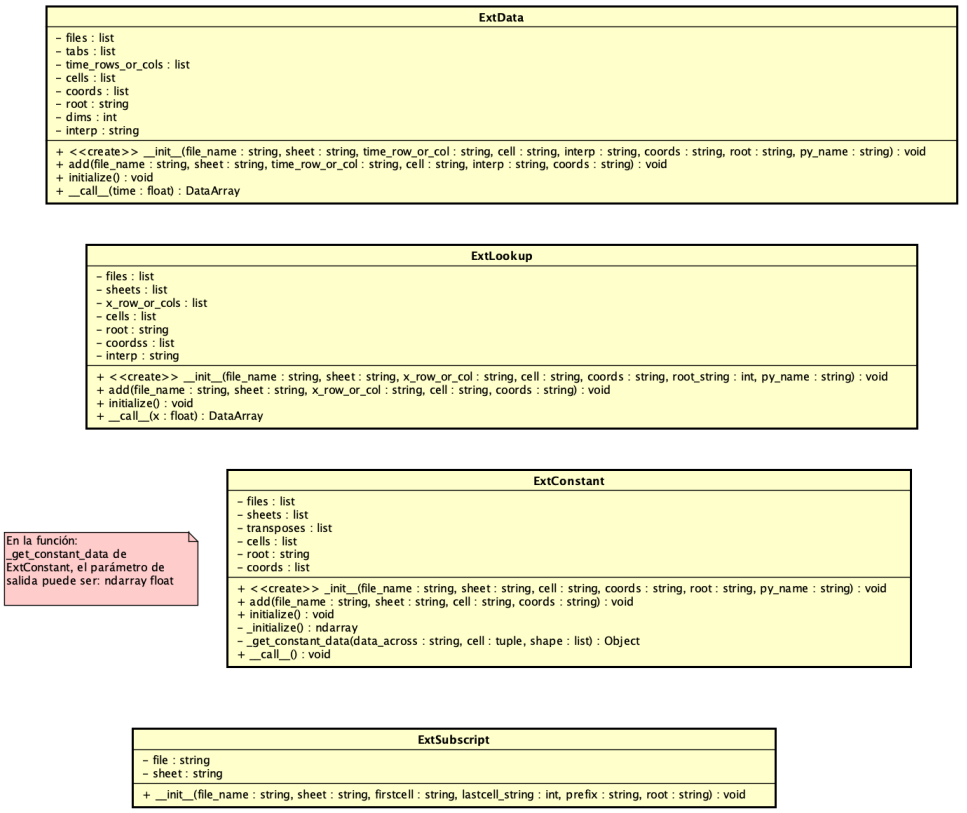

Figura 4.15: Clases del módulo **external**

Decorators

En el módulo **decorators**, detallado en la Figura 4.16, es donde se encuentran las funciones encargadas de desarrollar y decorar las funciones en la fase de traducción. La clase Cache representada es la que permite definir la funcionalidad deseada para esos decoradores. Las funciones **run** y **step** define la funcionalidad para la caché de doble nivel utilizada en PySD. La función **reset**, restablece el tiempo con el introducido por parámetro y limpia la caché de aquellos valores que se hayan etiquetado con *step*. La función **clean** limpia la caché cuyo nombre se le introduce como parámetro.

4.2.3. Vista de proceso

La arquitectura de proceso tiene en cuenta los requisitos no funcionales, como la disponibilidad y el rendimiento. Trata los aspectos dinámicos del sistema, explica los procesos del sistema y cómo interactúan entre ellos, entendiendo como proceso una agrupación de tareas que forman una unidad ejecutable. Esta vista se enfoca en el comportamiento del sistema en

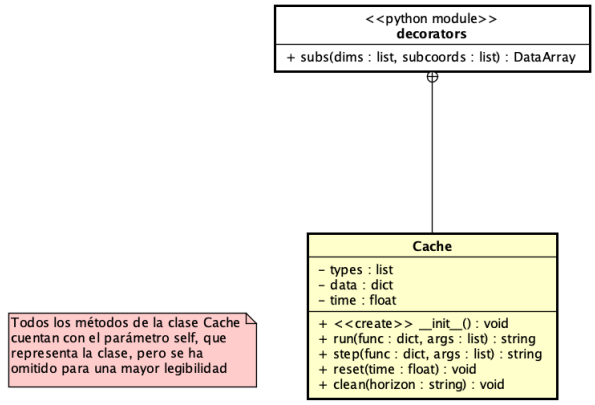

Figura 4.16: Módulo **decorators**

tiempo de ejecución. Se suele representar a través de diagramas de actividad UML.

El diagrama asociado a la secuencia principal de traducción de modelos Vensim que realiza PySD, se presenta desde la Figura 4.17 hasta la Figura 4.22. Recalcar, que la Figura 4.17 corresponde a la secuencia principal del diagrama de actividad y los demás diagramas presentados son un desglose de las principales actividades de éste.

El proceso de traducción comienza a partir del usuario, cuando indica el modelo Vensim (con extensión *mdl*) que desea traducir, utilizando para ello la función **read_vensim** del módulo *pysd* (Figura 4.4). En esta función, internamente se llama a la función **translate_vensim**, a la cual se le pasa como parámetro el modelo Vensim y se encuentra en el módulo *vensim2py* (Figura 4.4). Es entonces cuando se modifica la extensión del *path* que contiene el modelo para poder guardar la traducción con el mismo nombre y en la misma ruta que el archivo Vensim, pero con diferente extensión utilizando *.py*. Después, se separan las secciones que forman el modelo y, posteriormente, a partir de estas secciones obtenidas, se crea una lista con todas las macros existentes. Además, cada sección se organiza y se traduce dando lugar a la traducción que completará el archivo Python. A continuación, se explican los subsistemas que componen la Figura 4.17 con mayor detalle.

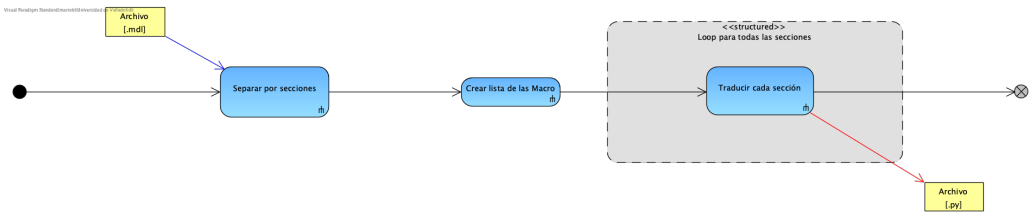

Figura 4.17: Diagrama de actividad principal

En la Figura 4.18 se presenta el subsistema **"Separar por secciones"**. Dentro de la función citada anteriormente, **translate_vensim**, se lee el modelo Vensim en modo texto y

se emplea la gramática *file_structure_grammar* (explicada en la Sección 4.1.1) encargada de separar las macros del código principal. Esta gramática se encuentra definida en la función **get_file_sections**, perteneciente al módulo *vensim2py*. A su vez, dentro de esta función se encuentra definida la clase que contiene los métodos *visitors* asociados a las reglas gramaticales, llamada **FileParser**, presentada con detalle en la Figura 4.6.

Como resultado de esta función y gramática, se consigue obtener el texto del modelo dividido en una lista con las diferentes secciones que lo forman, obteniendo una sección por cada macro del modelo y otra que almacena el código principal.

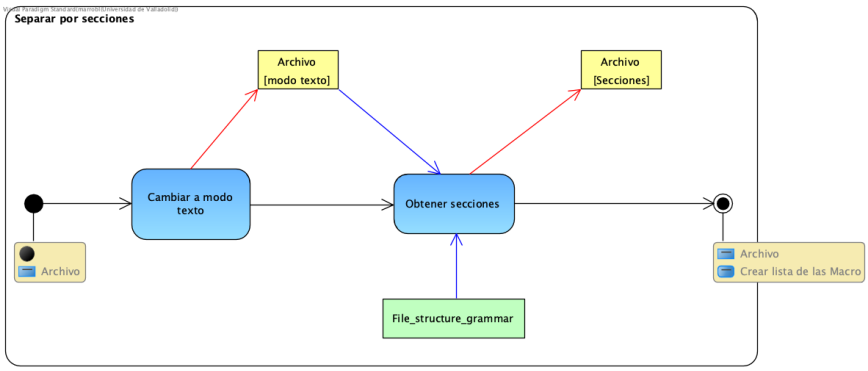

Figura 4.18: Diagrama de actividad. Separar en secciones

Una vez acabada la secuencia de **"Separar en secciones"** (Figura 4.18), se continúa creando una lista de macros, diagrama mostrado en la Figura 4.19. En esta sección de la traducción se filtran todas las secciones etiquetadas anteriormente para almacenar aquellas que sean macros y poder centralizar de esta forma todas las macros en una única lista.

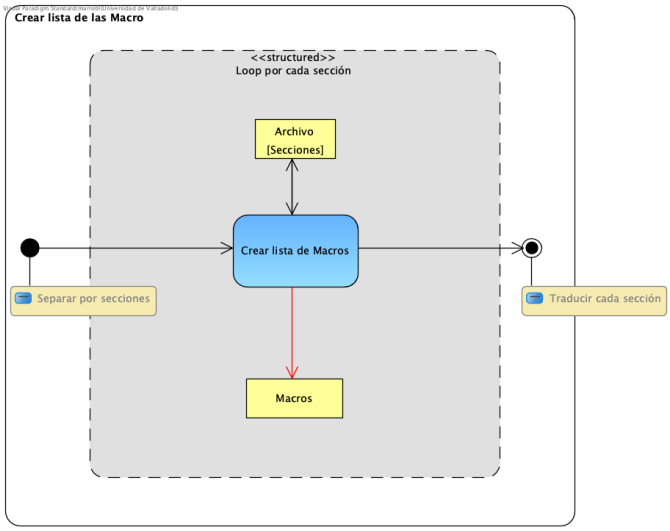

Figura 4.19: Diagrama de actividad. Crear la lista de macros

A continuación, cada una de las secciones en las que se ha dividido el modelo Vensim, se organizan y traducen a través de la función **translate_section** del módulo *vensim2py*. Esta secuencia se muestra en detalle en el diagrama de la Figura 4.20, con sus subactividades desarrolladas en las Figuras 4.21 y 4.22.

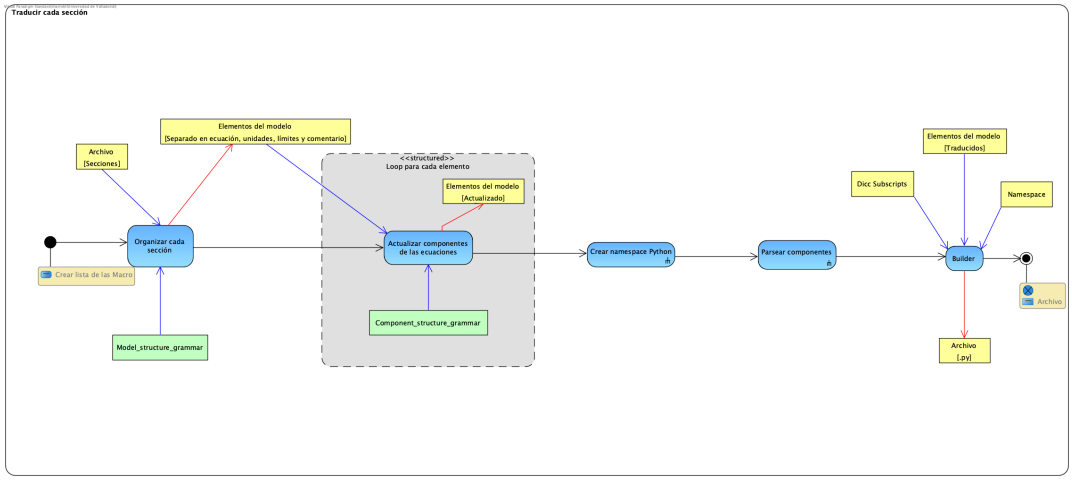

Figura 4.20: Diagrama de actividad. Organizar cada sección

En la Figura 4.20, a partir de la función **get_model_elements** (módulo *vensim2py*), se comienza a parsear cada una de las secciones con la gramática *model_structure_grammar* (Sección 4.1.2), que se encarga de organizar y actualizar las secciones a elementos del modelo dependiendo de si se trata de ecuaciones o de comentarios. En la función **get_model_elements**, además de esta gramática, se define la clase *NodeVisitor* asociada que se llama **ModelParser** (Figura 4.6). Como se ha comentado anteriormente, la gramática *model_structure_grammar* da como resultado el modelo separado en elementos que se encuentran organizados por: ecuaciones, unidades, límites, comentarios y tipo de la sentencia. Posteriormente, a medida que avanza el modelo por las diferentes gramáticas de PySD, las nuevas etiquetas en las que se dividen estos elementos del modelo se actualizan y/o se añaden a las que ya están almacenadas.

Aquellos elementos que hayan sido clasificados como comentarios no influyen en la traducción del archivo Vensim, solo son útiles para los desarrolladores de modelos. Por esta razón, se hace un filtrado de todos los elementos del modelo y aquellos que hayan sido etiquetadas como ecuaciones se actualizan a través de la gramática **component_structure_grammar** (Sección 4.1.3), reflejado en la Figura 4.20. Dicha gramática añade más información sobre el nombre y el tipo de la ecuación. En resumen, esta gramática permite actualizar y detallar la información de aquellos elementos del modelo que sean ecuaciones. La gramática **component_structure_grammar** se encuentra definida en la función **get_equation_components** del módulo *vensim2py* junto a su clase *NodeVisitor*, que contiene la lógica necesaria y se llama **ComponentParser** (Figura 4.6).

En la Figura 4.21 se representa con más detalle el subsistema **“Crear el namespace en**

Python" que es el siguiente paso en el proceso de traducción. El namespace es un diccionario compuesto de nombres de variables, de *subscripts*, de funciones y de macros que contiene el archivo Vensim, a los cuales se les asigna un nombre válido en Python. Para crear un nombre seguro en Python es necesario sustituir aquellos caracteres que estén permitidos en Vensim pero que en Python no sean válidos para nombres de variables, como espacios, palabras claves, símbolos no autorizados, etc. En este diccionario se almacenan los nombres de Vensim en las *'keys'* del diccionario y se guardan en *'value'* los nombres seguros correspondientes en Python.

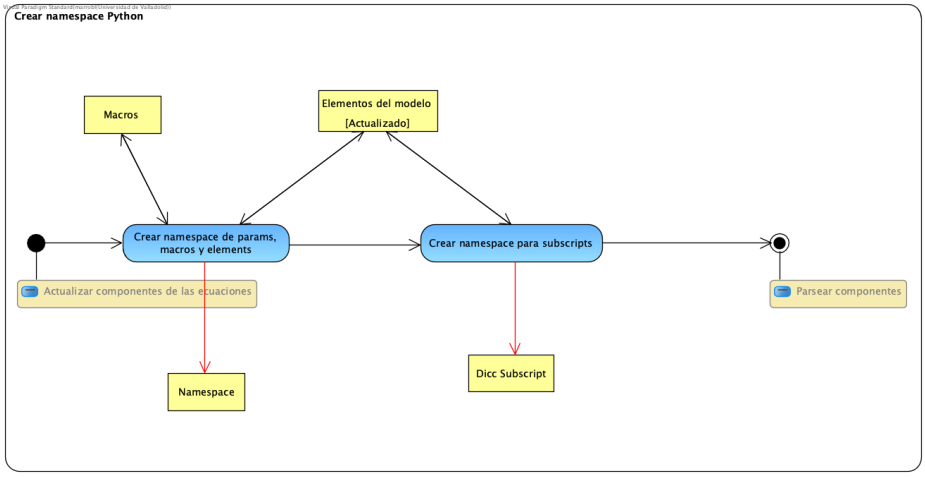

Figura 4.21: Diagrama de actividad. Crear namespace en Python

Para ello, dentro de la función **translate_section**, se accede a la lista de las macros (obtenida anteriormente en la Figura 4.19) y a las diferentes secciones que se han ido actualizando. Con cada nombre de macro, cada parámetro de las macros y demás elementos del modelo, se añade una entrada al diccionario *namespace* con el nombre que los representa en Vensim y su correspondiente válido en Python, generado a partir de la función **make_python_identifier** del módulo *utils* (Figura 4.12).

Posteriormente, se crea otro diccionario, aparte de namespace, para añadir aquellos nombres de los *subscripts* que forman el modelo, como se muestra en la Figura 4.21. Los nombres de los *subscripts* son almacenados en otro diccionario ya que no se utilizan para crear las funciones de Python, si no que solamente representan dimensiones de los DataArrays y no necesitan tener un "nombre seguro" en Python. Así que, este diccionario de *subscripts* está compuesto por todos los subscripts que forman el modelo, teniendo como valor de *key* el nombre del subscript y como *values* todos los *subscript values* asociados al mismo.

Una vez creado el namespace, se continúa parseando los diferentes componentes, como se presenta en la Figura 4.22 (subsistema de la Figura 4.20). En este punto de la secuencia de traducción se dividen los elementos del modelo dependiendo del tipo que sean, ya bien pueden ser expresiones normales o definiciones de *lookups*.

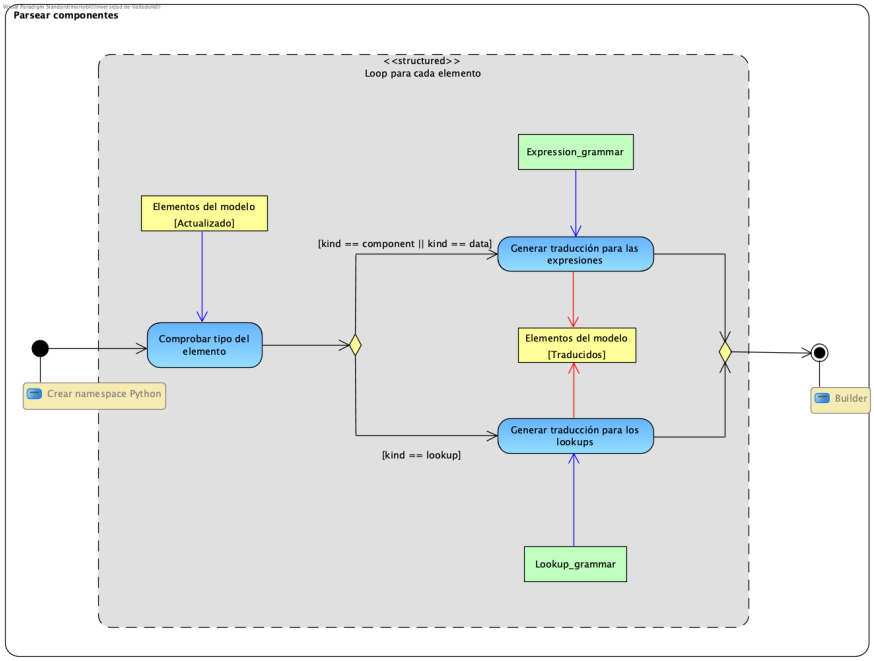

Figura 4.22: Diagrama de actividad. Parsear cada componente

En el caso de que se trate de una ecuación se parsea con la gramática *expression_grammar* (Sección 4.1.4) y si se trata de una definición de un lookup se utiliza la gramática *lookup_grammar* (Sección 4.1.5). La primera gramática comentada, *expression_grammar*, se encuentra en la función **parse_general_expression** del módulo *vensim2py*, donde también se encuentra definida la clase **ExpressionParser** (Figura 4.7) que contiene toda la lógica asociada a dicha gramática. La gramática *lookup_grammar* y su clase asociada, **LookupParser** (Figura 4.7), se encuentran definidas en la función **parse_lookup_expression** del módulo *vensim2py*. Ambas gramáticas, actualizan de nuevo los elementos almacenados, añadiendo la traducción correspondiente en el lenguaje Python en una nueva etiqueta para cada elemento.

Una vez terminada esta secuencia y volviendo a la Figura 4.20, el proceso de PySD finaliza con el *builder*, que es el encargado de crear el archivo Python que contiene la traducción del modelo Vensim, utilizando la función **build** del módulo *builder* (Figura 4.11). Para ello, parte de los namespaces creados en el proceso y los diferentes elementos del modelo previamente traducidos y etiquetados con la información relevante, que pasarán a formar parte del archivo Python.

4.2.4. Vista física

La arquitectura física se compone de aquellos requisitos no funcionales del sistema, como la disponibilidad, tolerancia a fallos, rendimiento y escalabilidad. Esta vista representa el sistema desde el punto de vista de un ingeniero de sistemas, representando los componentes

del sistema en la capa física y las conexiones físicas entre ellos. En UML se suelen utilizar los diagramas de despliegues para representar esta vista.

PySD es un sistema que se despliega en un única *workstation* y todo lo que necesita se encuentra en el mismo componente. Por lo que realizar un diagrama de despliegue para PySD no arrojaría información relevante sobre el sistema y su funcionamiento.

4.2.5. Escenarios

Los escenarios en el modelo de Vistas 4+1 se encargan relacionar las cuatro vistas presentadas previamente. Son casos de uso que describen secuencias de interacciones entre objetos y procesos del sistema que permiten identificar y validar el diseño de la arquitectura representado en las diferentes vistas.

En todo el proyecto de la biblioteca PySD se pueden diferenciar dos escenarios principales. El primer escenario con el que cuenta este proyecto es el proceso de traducción de un modelo Vensim a Python, en el cual, como se ha comentado anteriormente, se parte de un archivo Vensim y, tras la secuencia explicada en la vista de proceso, se genera un equivalente del modelo en Python. El segundo escenario corresponde a la ejecución de dicho modelo Python, lo cual permite simular el paso del tiempo con los intervalos definidos en el modelo y obtener los resultados de la simulación idénticos a los que generaría la simulación en el entorno Vensim. Estos dos escenarios se representan en la Figura 4.23 a través de un diagrama de casos de uso bastante simple que permite esquematizarlos.

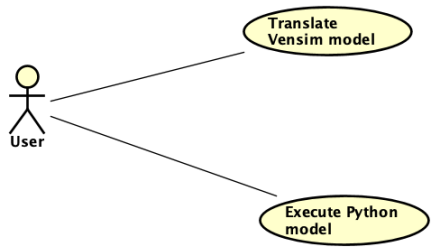

Figura 4.23: Diagrama de Casos de Uso

Como se puede comprobar en la Figura 4.23, el primer escenario con el que cuenta PySD corresponde a la **Traducción de un archivo Vensim**. Esta secuencia está desarollada en detalle en los diagramas de las Figuras 4.24, 4.25 y 4.26.

En la Figura 4.24 se muestra el diagrama en detalle de la secuencia de **Traducción**. Comienza el usuario interactuando con el módulo **pysd**, mediante la función **read_vensim**, a la cual le pasa por parámetro el modelo Vensim que desea traducir. A partir de ahí, el módulo **pysd** interactúa con el módulo **vensim2py** y es éste el encargado de parsear el modelo utilizando las diferentes gramáticas. En este diagrama, se presenta la primera gramática, *file_structure_grammar*, y la creación de la lista de macros. Además se encuentra un *interaction use* cuyo detalle se corresponde al diagrama de la Figura 4.25.

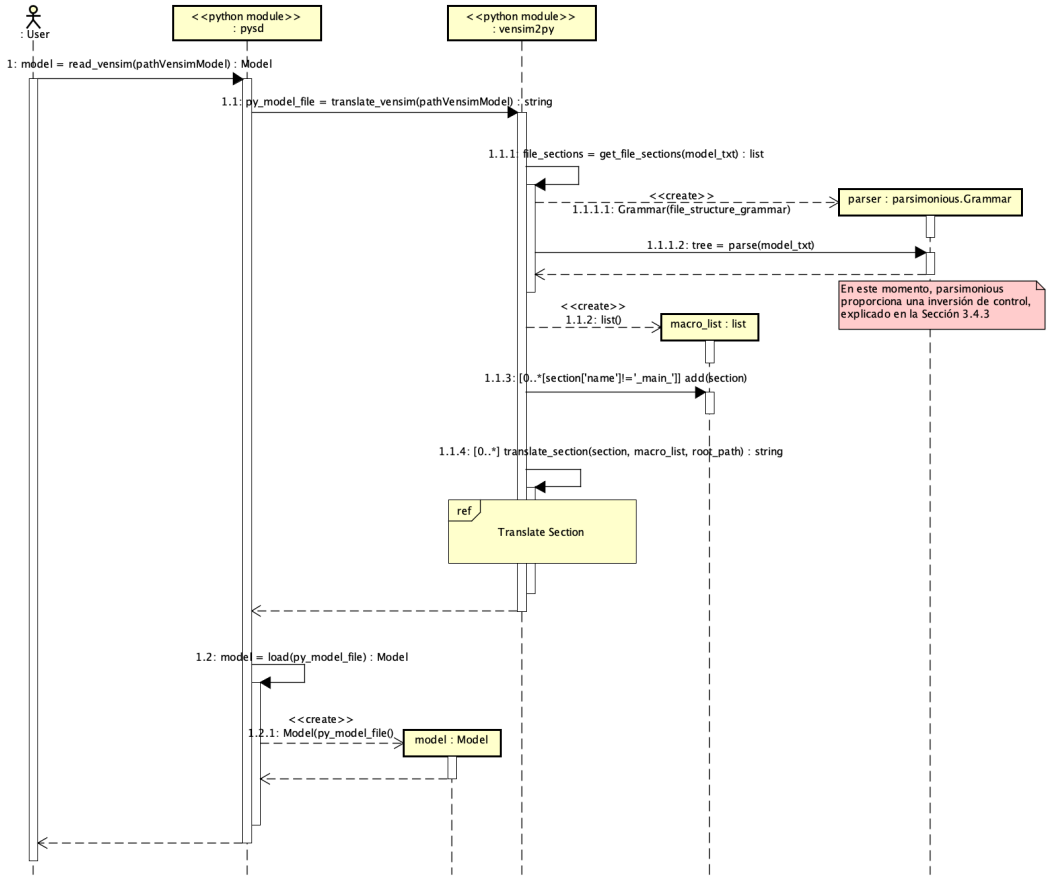

Figura 4.24: Diagrama de secuencia principal: Traducción

En la Figura 4.25, el módulo **vensim2py** parsea el modelo por la gramática *model_structure_grammar* y *component_structure_grammar*. También, se representa la creación del diccionario *namespace*, así como los elementos que se almacenan en él, para lo que el módulo **vensim2py** utiliza la función *make_python_identifier* del módulo **utils**. El diccionario *namespace* no es el único que se encuentra en este diagrama, ya que también se presenta la creación del diccionario *subscript_dict*, el cual almacena los subscripts que contiene el modelo. Posterior a esto, se presenta un *interaction use* ('Add Python translation and call builder'), cuyo detalle se presenta en el diagrama de la Figura 4.26.

En la Figura 4.26 se expone el diagrama de secuencia relativo al *interaction use* que añade a los diferentes elementos del modelo la traducción en Python y, posteriormente, llama al builder para cerrar el proceso de traducción y transcribir todos los elementos traducidos en un modelo pysd. En esta figura se presenta la condición con la que el módulo **vensim2py** interactúa con la gramática *expression_grammar* o *lookup_grammar*, dependiendo del tipo de elemento que se esté tratando en ese momento. Esta secuencia termina con la llamada a la función **build** del módulo **builder**, la cual crea un modelo Python, también llamado modelo

pysd, que contiene todas traducciones de los elementos que se han recopilado en la secuencia anteriormente representada equivalentes al modelo Vensim introducido por el usuario. Por lo que, cuando este proceso de traducción finalice se obtendrá como resultado en la misma ruta en la que se localiza el modelo Vensim, el modelo pysd equivalente.

Es en la Figura 4.27 donde se muestra el diagrama de secuencia que representa la interrelación entre los diferentes módulos en el proceso de ejecución de un modelo pysd.

El usuario comienza utilizando la función **run** de la clase **Model**, cuyos parámetros corresponden a determinadas funcionalidades que se pueden añadir a la ejecución o al resultado de la misma, como por ejemplo se permite indicar qué columnas de resultados se devuelven, en lugar de devolver todos los resultados del modelo. La instancia *model* de la clase **Model** (módulo **functions**) es la obtenida como resultado en el proceso anterior de traducción.

En esta secuencia de ejecución, se inicializan todos los objetos con los que cuente *model* y, mediante la función **_integrate**, se lleva a cabo la integración por euler de todos los elementos con los que cuente el modelo y a partir de los cuales se obtendrá el resultado.

Tras estas dos secuencias, las funciones de la librería PySD que más utilizan los usuarios son **read_vensim**, para traducir el modelo Vensim, y **run**, para ejecutarlo, como se ha explicado anteriormente y también se indica en la documentación oficial del proyecto para el uso básico enfocado a usuarios [28].

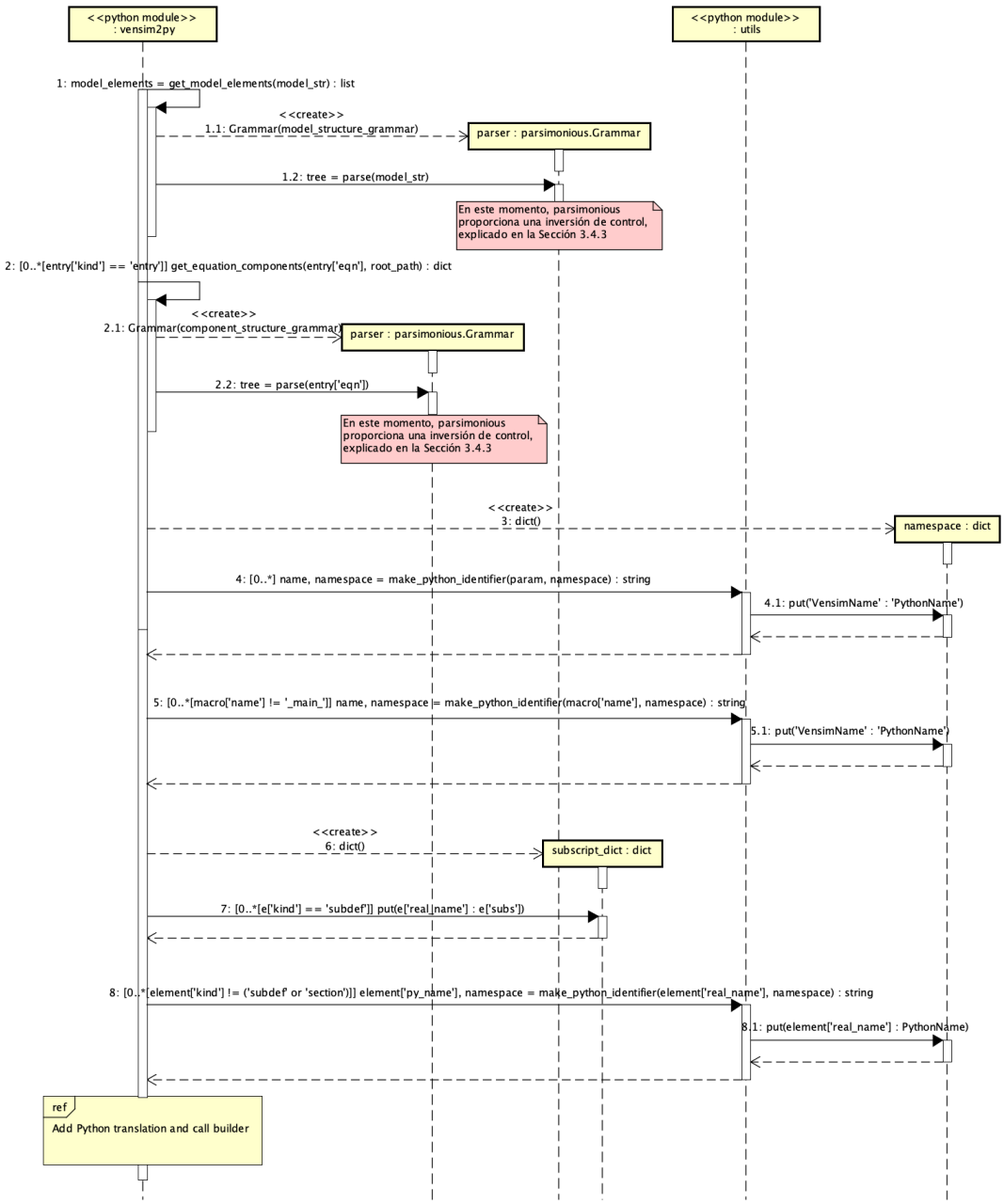

Figura 4.25: Diagrama de secuencia: Traducir sección

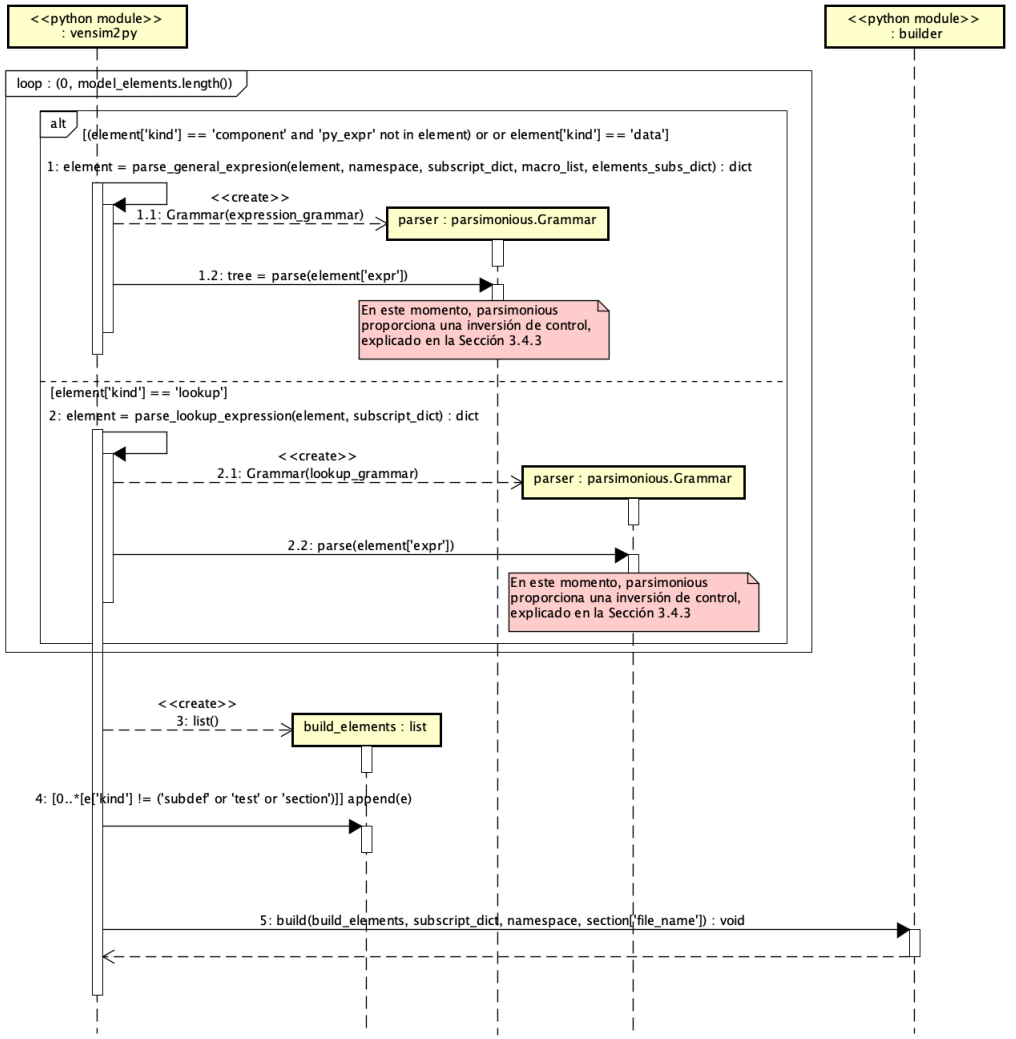

Figura 4.26: Diagrama de secuencia: Añadir traducción y llamar al builder

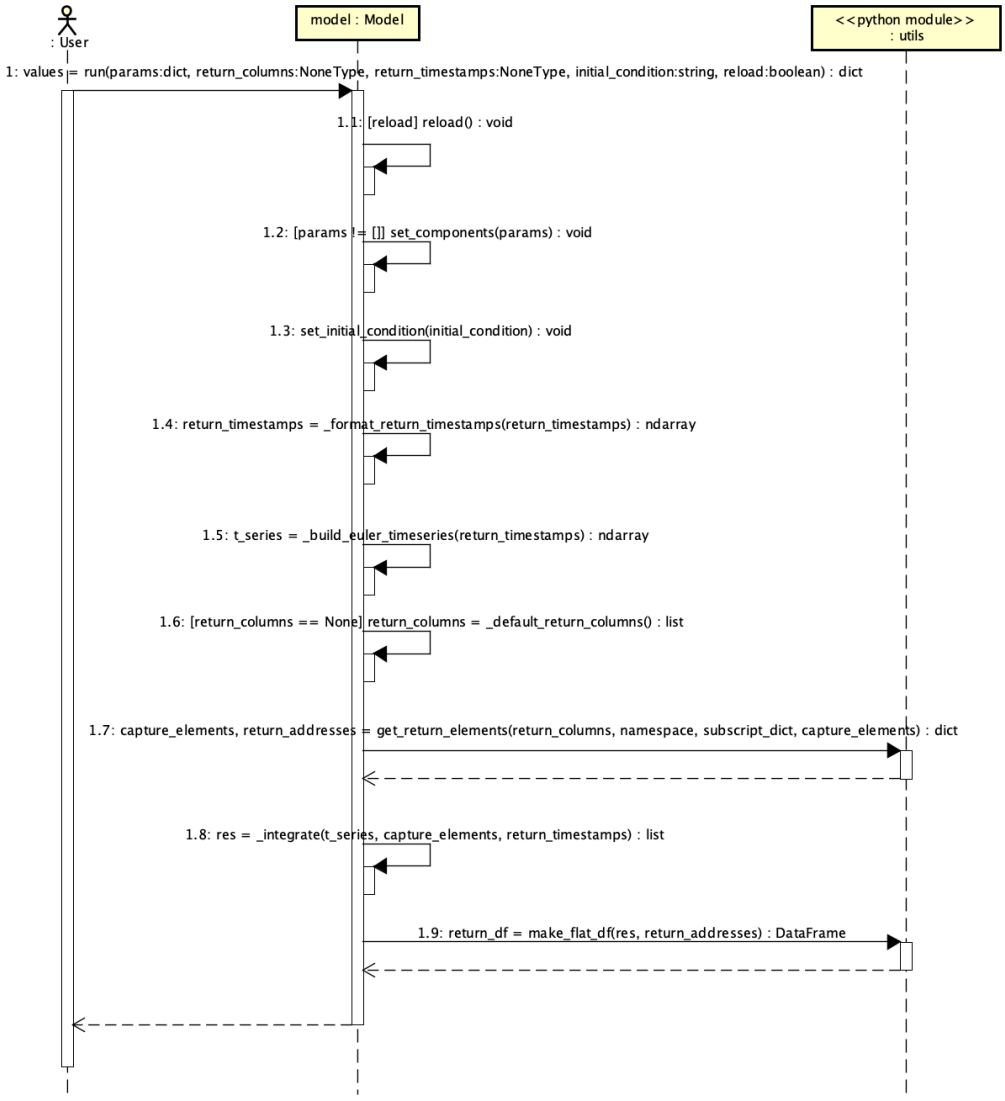

Figura 4.27: Diagrama de secuencia: Ejecución del modelo ya traducido

Capítulo 5

Tecnologías utilizadas

En este capítulo se explican las tecnologías utilizadas para la gestión y el desarrollo del proyecto.

5.1. Tecnologías para la gestión del proyecto

5.1.1. Jitsi

Se ha utilizado **Jitsi** [12], herramienta de videoconferencia proporcionada por la Escuela de Ingeniería Informática de la UVa. Mediante esta plataforma se han llevado a cabo las *weeklies* necesarias entre la tutora y la alumna para el correcto seguimiento del proyecto.

5.1.2. Rocket.Chat

Rocket.Chat [59] ha sido la plataforma de intercambio de mensajes para estar en contacto con la tutora, en el caso de que hubiera algún inconveniente o duda y no tener la necesidad de esperar a la siguiente reunión semanal. Esta plataforma la proporciona la Escuela de Ingeniería Informática.

5.1.3. GitLab Issues

GitLab Issues [13] es una herramienta que permite monitorizar y controlar el progreso del proyecto. Permite compartir y debatir mejoras o propuestas para el proyecto entre todos los integrantes del mismo. Las *issues* permiten a su vez llevar un seguimiento de las tareas y estado laboral, pudiendo incluir el tiempo estimado y el tiempo invertido en la realización

de cada tarea. Además, permiten aceptar propuestas de funciones, preguntas, solicitudes de soporte o informes de errores, función muy útil cuando se trata de un proyecto *open source* para hacer más fácil la contribución entre todos los integrantes del proyecto.

GitLab Issue Board [14] es una herramienta, dentro de **GitLab Issues**, que permite gestionar los proyectos software, planificando, organizando y visualizando un flujo de trabajo de *issues* o para la realización de una *release*. En este proyecto se ha realizado un tablero para poder visualizar y administrar las *issues* en base al marco de trabajo *Scrum*. Esto permite conocer el avance y estado de cada *issue*. El tablero se divide en listas, las cuales simulan las etapas por las que evoluciona una *issue*. En orden, estas son:

- **Icebox**: lista de todas las tareas consideradas en el proyecto.
- **Backlog**: lista de tareas ordenadas con algún tipo de prioridad.
- **Sprint Backlog**: tareas pertenecientes al sprint actual, ordenadas por prioridad.
- **Doing**: tareas que están siendo realizadas.
- **Under review**: lista de las tareas ya realizadas que se están revisando para poder darlas por finalizadas. En el caso de que se encuentre alguna deficiencia o algún aspecto a mejorar, se retrocede la *issue* a la anterior lista (*Doing*) y se corrige.
- **Done**: aquellas tareas realizadas y revisadas.

En las Figuras: 5.1 y 5.2 se presenta el tablero realizado con **GitLab Issues** y en él se encuentran las listas antes comentadas. Cada lista tiene asociadas diferentes *issues* dependiendo del nivel de completitud que lleven éstas.

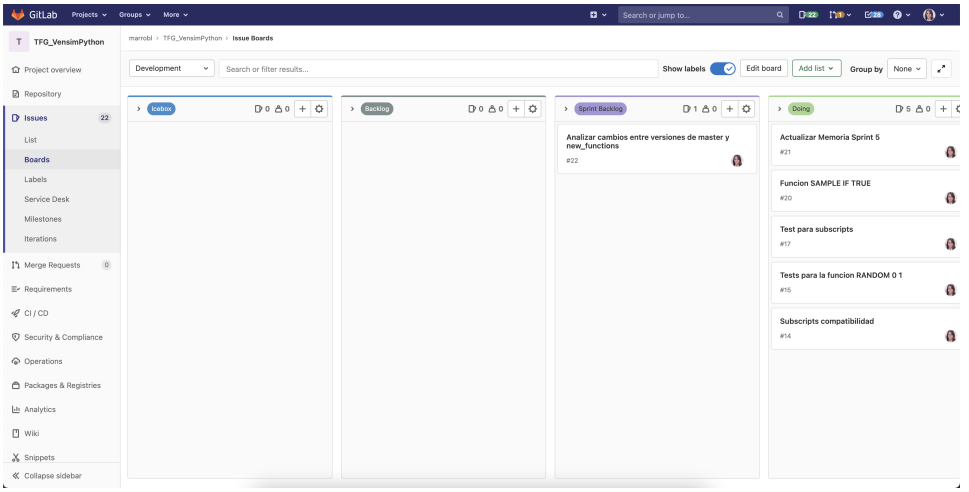

Figura 5.1: Gitlab Issues

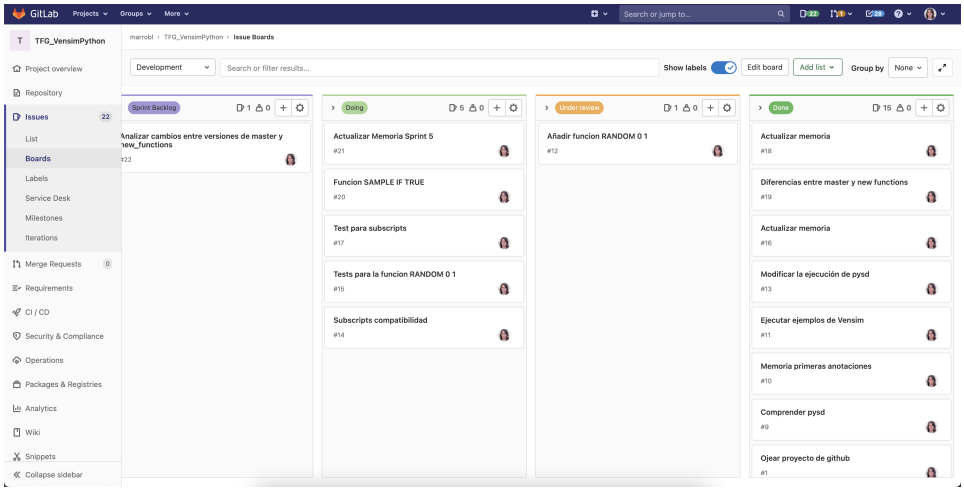

Figura 5.2: Gitlab Issues

Además, se ha añadido otro tipo de etiquetas para diferenciar la tipología de las *issues* entre sí. Estas son:

- "Memoria": utilizándose en aquellas *issues* relativas a la actualización de la memoria del proyecto.
- "Código": para las *issues* que implican desarrollo de funcionalidad.
- "Test": utilizadas en aquellas *issues* correspondientes a testear funcionalidad añadida previamente.
- "Pul request": para las *issues* que conlleven la preparación de un *pull request*.

5.1.4. Overleaf

Overleaf [8] es un editor de texto de **LaTeX** en la nube utilizado para escribir, editar y publicar documentos. Se ha utilizado esta tecnología para desarrollar la memoria del proyecto y poder plasmar el seguimiento del proyecto de manera escrita.

5.2. Tecnologías para el desarrollo del proyecto

5.2.1. Visual Studio Code

Visual Studio Code [49] es un editor de código fuente desarrollado para *Windows*, *macOS* y *Linux*, desarrollado por *Microsoft*. Incluye control integrado de *Git*, resaltado de sintaxis,

soporte para la depuración, finalización de código inteligente, etc. Se ha utilizado para realizar los cambios y mejoras del código de PySD.

5.2.2. Git

El control de versiones es un sistema que registra los cambios de un proyecto a lo largo del tiempo y permite revertir los cambios de archivos concretos a una versión anterior, revertir todo el proyecto, comparar diferentes cambios, ver quién modificó cada archivo, etc. Es muy recomendable usar herramientas de control de versiones para cualquier desarrollo, aunque sea individual [4].

Git [7], desarrollado por Linus Torvals, es un sistema de control de versiones distribuido, gratuito y de código abierto diseñado para manejar cualquier tipo de proyecto.

5.2.3. GitHub

GitHub [25] es una plataforma de desarrollo colaborativo para alojar proyectos utilizando Git. El código de los proyectos en GitHub se almacena normalmente de forma pública. GitHub es la plataforma más importante de colaboración de proyectos *open source*. El proyecto PySD se aloja en esta plataforma y cuenta con muchas contribuciones de diferentes desarrolladores.

5.2.4. GitLab

GitLab es otro servicio de control de versiones y desarrollo colaborativo basado en Git. GitLab ofrece algunas características similares para el seguimiento de problemas y la gestión de proyectos como GitHub. Pero una de las grandes diferencias entre estas dos plataformas radica en la Integración Continua que solo GitLab incorpora [54]. La CI(*continuous integration*) consiste en hacer integraciones automáticas de un proyecto lo más a menudo posible para así localizar los fallos cuanto antes, entendiendo como integrar el proceso de compilar y ejecutar las pruebas de un proyecto completo.

5.2.5. Astah

Astah [3] es una herramienta para realizar modelos de UML, creada por la empresa japonesa *Change Vision*. Permite crear fácilmente los diagramas UML que sean necesarios proporcionando todas las características precisas para ello. En este proyecto se ha utilizado esta herramienta para poder representar la estructura de paquetes y archivos de PySD mediante un diagrama de clases.

5.2.6. Visual Paradigm

Visual Paradigm [52] es una herramienta de modelado UML. Permite varios tipos de diagramas, como diagramas de clases, de secuencia, de máquinas de estado, de actividad, de componentes, de paquetes, etc. En este proyecto se ha utilizado Visual Paradigm para realizar el diagrama de actividad relativo al flujo de información y proceso de traducción de PySD. A diferencia del programa *Astah*, *Visual Paradigm* proporciona la recreación de bucles en los diagramas de actividad, material que ha sido clave para poder modelar con mayor exactitud el proyecto PySD.

5.2.7. Unittest

Unittest [21] es un módulo de Python para definir y ejecutar pruebas unitarias. Se inspiró originalmente en JUnit y es similar a los principales frameworks de prueba de otros lenguajes. Proporciona un gran conjunto de herramientas para programar y ejecutar tests. En PySD, este framework es utilizado en los tests unitarios de cada módulo.

5.2.8. Nose

Nose [55] permite escribir tests simples y proporciona una serie de funciones útiles para poder escribir test cronometrados, tests de excepciones y otros casos de uso comunes. Además, contiene plugins para permitir que se capturen los datos de salida, la cobertura del código, los tests de documentos y otras funcionalidades realmente útiles.

En el proyecto de PySD, se utiliza el framework **nose** en los tests de integración.

5.2.9. Travis CI

Travis CI [66] es un servicio de integración continua que permite ejecutar los tests, depurar e implementar código. Los proyectos se pueden sincronizar con esta herramienta y, como resultado, los tests se ejecutarán automáticamente.

En el proyecto de PySD, cada vez que se realiza una *pull request*, los tests se ejecutan automáticamente con **Travis CI**. La herramienta informa si ha habido algún error, si algún test ha fallado o si todos los tests han sido ejecutados correctamente.

5.2.10. Markdown

Markdown [43] es una herramienta de conversión de texto a HTML para escritores web. Markdown le permite escribir usando un formato de texto plano fácil de leer y escribit, y luego convertilo a XHTML (o HTML). Por lo tanto, **Markdown** proporciona una sintaxis

de formato de texto plano y una herramienta de software que convierte el texto sin formato a HTML.

En este proyecto, se ha utilizado **Markdown** para escribir un documento sobre la arquitectura del sistema PySD.

5.2.11. reStructuredText

reStructuredText [57] es un lenguaje de marcas ligero para escribir texto plano fácil de leer. Es útil para documentación de programas online, para crear rápidamente páginas web simples y para documentos independientes.

El principal objetivo de **reStructuredText** es definir e implementar una sintaxis de marcado para utilizar en cadenas de documentos de Python. En PySD, el contenido de la documentación del proyecto se encuentra escrito en **reStructuredText**.

5.2.12. Sphinx

Sphinx [6] es una herramienta que facilita la generación de documentación, creada por Georg Brandl. Fue originalmente creada para documentación Python y cuenta con facilidades para generar documentación de proyectos software es una variedad de lenguajes.

Sphinx utiliza **reStructuredText** como lenguaje de marcado. A partir de este, puede crear diferentes formatos de salida para lo documentación: HTML, LaTeX, ePub, Texinfo, páginas de manual o texto sin formato.

La documentación original de PySD está compilada con esta herramienta, **sphinx**.

Capítulo 6

Seguimiento del proyecto

En este capítulo se pretende representar el avance del proyecto real con respecto a lo establecido en un principio en la Sección 2.4.

El comienzo del proyecto fue a mediados de Septiembre del 2020. Se han realizado las correspondientes *weeklies* con la tutora los miércoles de cada semana para la supervisión y el correcto avance del proyecto.

6.1. Sprint 0

Duración: del 14/09/2020 al 21/10/2020

El Sprint 0 se ha desarrollado durante algo más de un mes, debido a que su fin es que la alumna amplíe sus conocimientos sobre Vensim y el funcionamiento de PySD. La Tabla 6.1 muestra el desglose de las tareas realizadas en este Sprint de inicio.

Se comenzó leyendo el Trabajo de Fin de Grado predecesor a este, el de **Diego Rodrigo Verdugo** [60], para conocer el alcance que había tenido su proyecto y los cambios que había realizado. Fue de gran utilidad ya que contenía partes teóricas introductorias tanto como para el software **Vensim** como para **PySD**.

Se estudió el funcionamiento de PySD hasta el momento. Entrando en detalle sobre cómo se modifica y se iba traduciendo el archivo de Vensim hasta llegar a un archivo de Python. Para lo que fue necesario profundizar en el funcionamiento sobre *parsimonious*, el parser que se utiliza en el proyecto PySD, y así poder comprender con más detalle las 5 gramáticas que posee PySD y la funcionalidad de cada una. Se ha realizado un diagrama de actividad que ilustra el flujo de información y el proceso de traducción de un fichero **.mdl** (Vensim) a un fichero Python, mostrando de una manera más conceptual el funcionamiento de las grámaticas y la utilidad de cada una.

Se descargó **Vensim** y para poder comenzar a familiarizarse con el software y se realizó algún ejemplo. Además, se estudiaron las diferentes variables que posee Vensim, como subscripts y lookups. Se consultaron ejemplos proporcionados por la tutora para aprender más sobre el software y, posteriormente, se intentaron traducir con la versión de PySD que había realizado Diego Rodrigo y así poder elicitar los requisitos necesarios para que se pudiera realizar correctamente la traducción de **WILIAM**.

También se consultó el modelo **MEDEAS** [47], archivo en Vensim que intenta simular el efecto que tendrían tanto en el sector energético como en el ecosistema diferentes decisiones. Para entender más el proyecto se consultó el MOOC de MEDEAS [48], una serie de vídeos en la que se explica el funcionamiento y el modelo planteado en MEDEAS.

Se intentó descargar *pymedeas*, versión de PySD desarrollada por *rogersamso* que permitía traducir el modelo MEDEAS. Sin embargo, se trata de una versión semimanual ya que en algunos casos la traducción no está lo automatizada. Finalmente, se encontraron muchas complicaciones con las versiones de Python necesarias para descargar esta versión y se decidió abandonar la tarea ya que no era tan relevante esa versión para el proyecto y siempre se puede acceder a dicho código a través de GitHub o la página web de MEDEAS.

Se comenzó a escribir la memoria, para plasmar los conocimientos que se iban adquiriendo sobre PySD, Vensim, MEDEAS y los proyectos *open source*. Además, se creó en GitLab un tablero para ir introduciendo las diferentes *issues*.

Nombre de la Tarea	Tiempo invertido	Estado
0.1 - Entender el proyecto MEDEAS y realizar el MOOC	5h 15m	Completada
0.2 - Leer y entender el TFG de Diego Rodrigo	7h 50m	Completada
0.3 - Aprender sobre el funcionamiento de Vensim y realizar ejemplos	3h 35m	Completada
0.4 - Seguir tutorial para descargar pymedeas	4h 50m	Completada
0.5 - Aprender sobre el parser Parsimonious de Python	2h 10m	Completada
0.6 - Aprender el funcionamiento de PySD	9h 30m	Completada
0.7 - Traducir con PySD ejemplos en Vensim y así establecer requisitos	3h 15m	Completada
0.8 - Crear en GitLab un tablero	30m	Completada
0.9 - Empezar memoria	8h 35m	Completada
Total	**45h 30m**	**8/8**

Tabla 6.1: Tareas del Sprint 0

6.2. Sprint 1

Duración: del 21/10/2020 al 4/11/2020

En la Tabla 6.2 se desglosan todas las tareas acordadas para este Sprint.

En el Sprint anterior se comprobó que la versión de PySD no tenía funcionalidad asociada a la función *RANDOM 0 1* de Vensim, explicada en la Sección 3.2.3. Función que escoge un número aleatorio entre 0 y 1. Sin embargo, esta función está obsoleta y se sustituye por *RANDOM UNIFORM (0,1,0)*. Como se indica en el manual, se mantiene por la compatibilidad con versiones anteriores [65]. Aún así, fue un avance para que la alumna comenzara a tomar contacto y realizar los primeros cambios sobre PySD.

Se realizaron algunos tests para comprobar el correcto funcionamiento de la función implementada, *RANDOM 0 1*. Sin embargo, los archivos *Vensim* sobre los que se probó solo devolvían valores constantes ya que las funciones de números aleatorios en *Vensim* necesitan una semilla para poder generar aleatoriedad. Esto se intentará mejorar más adelante, ya que para generar esa semilla se necesita la función *GET TIME VALUE*, no implementada aún en PySD. Por lo que los test únicamente probaban el correcto funcionamiento del parser con la función *RANDOM 0 1*.

Cabe destacar, la utilidad del Trabajo de Fin de Grado de Diego Rodrigo [60], que ha sido una guía con conceptos teóricos muy útiles.

También se comprobó que PySD no soportaba *subscript sequences*, *subscript copy* ni *subscript mapping*. Por lo que se añadieron estos tipos de operaciones de susbscripts a la gramática. Estas operaciones se encuentran explicadas en la Sección 3.2.2. Se modificó la gramática **component_structure_grammar** para añadir la compatibilidad de subscripts y así hacer que el parser no fallara. Hubo una estimación errónea del tiempo en un principio, ya que estas tareas serían más complejas de lo que se pensó en un primer momento, así que, por esa razón, estas tareas se retoman en sprints posteriores mejorando el trabajo implementado en este sprint. La tarea "1.5 - Añadir funcionalidad a la secuencia de subscripts", se refiere a añadir una nueva función que parseara los dos subscripts que corresponden a los límites de un rango numérico y comprobara que no hubiera errores, como que los nombres de la secuencia coincidieran o los números estuvieran en orden. Obteniendo así la parte común y los límites numéricos, para añadir posteriormente todos los elementos de esa secuencia en el subscript. Para que se entienda mejor, con el ejemplo representado en el Fragmento de código 3.5, la parte común sería *Layer* y los límites serían [1,4].

Se actualizó y mejoró el diagrama de actividades que presentaba el flujo del archivo a través del proyecto PySD comenzado en el Sprint 0. Se pulieron ciertos aspectos relacionados con la presentación.

Finalmente, durante todo este Sprint, se actualizó la memoria en relación a los diferentes cambios que habían acontecido y se amplió la información de ésta.

Nombre de la Tarea	Tiempo estimado	Tiempo invertido	Estado
1.1 - Implementar función RAMDOM 0 1	1h	1h 40m	Completada
1.2 - Añadir tests para la función RAMDOM 0 1	30m	10m	En progreso
1.3 - Ejecución y comprobación de tests Vensim	3h	1h 45m	Completada
1.4 - Añadir a la gramática la secuencia de subscripts	3h 30m	3h	En progreso
1.5 - Añadir funcionalidad a la secuencia de subscripts	2h	3h 10m	En progreso
1.6 - Añadir a la gramática subscripts copy	3h 30m	30m	En progreso
1.7 - Añadir a la gramática subscrips mapping	3h 30m	30m	En progreso
1.8 - Actualizar memoria seguimiento de Sprint 1	4h	2h	Completada
1.9 - Actualizar diagrama de actividades de PySD	30m	1h	Completada
Total		**13h 45m**	**4/9**

Tabla 6.2: Tareas del Sprint 1

6.3. Sprint 2

Duración: del 4/11/2020 al 18/11/2020

La Tabla 6.3 muestra el desglose de las tareas realizadas en este sprint.

En este segundo sprint se continuó mejorando la gramática sobre la compatibilidad de *subscripts*. Se mejoró la función empleada para la secuencia de *subscripts* (Tarea 2.1), debido a que había algunos casos de prueba para los que no funcionaba correctamente. Se añadió un diccionario a mayores en la gramática **component_structure_grammar** para poder añadir las entradas de *subscripts* que se mapeaban y poder implementar las operaciones de *subscript copy* y *subscript mapping*, por eso se añade una pequeña descripción a las Tareas 2.2 y 2.3 sobre añadir un diccionario y las entradas correspondientes cuando se tratan de las operaciones. Sin embargo, en este segundo sprint, no se acabó la implementación completa de estas dos operaciones comentadas, *subscript copy* y *subscript mapping*, por lo que se retomarán en el siguiente sprint.

Se comprobó que la versión de PySD implementara *subscript exclamation*. Símbolo "!" que dentro de un *subscript* indica todo el rango de subíndices que éste posee [36]. Se comprobó

que la versión actual de PySD implementa este símbolo.

Se actualizó y mejoró el diagrama de actividades comenzado anteriormente, ya que, a medida que se iba trabajando con PySD, se encontraban nuevas interacciones importantes que no se habían representado en el diagrama. También, se actualizó la memoria con lo correspondiente al Sprint 2.

Nombre de la Tarea	Tiempo estimado	Tiempo invertido	Estado
2.1 - Completar Tarea 1.5. Mejorar la implementación de la función de subscript sequence	2h	3h 30m	Completada
2.2 - Completar Tarea 1.6. Añadir a la gramática subscripts copy y añadir diccionario para subscripts copy	2h	2h	En progreso
2.3 - Completar Tarea 1.7. Añadir a la gramática subscripts mapping y añadir diccionario para subscripts mapping	2h	4h 30m	En progreso
2.4 - Comprobar si en ejecución se implementa Subscript exclamation	2h	30m	Completada
2.5 - Actualizar memoria seguimiento del Sprint 2	3h	4h	Completada
2.6 - Actualizar diagrama de actividades de PySD	1h	2h 15m	Completada
Total		**16h 45m**	**4/6**

Tabla 6.3: Tareas del Sprint 2

6.4. Sprint 3

Duración: del 2/12/2020 al 16/12/2020

Las tareas que se han acordado para este tercer sprint están desglosadas en la Tabla 6.4.

Se comenzó añadiendo tests para la función anteriormente implementada *RANDOM 0 1*. Dichos tests se obtuvieron a partir de los ejemplos proporcionados por *Vensim* en su guía de usuario. Sin embargo, y como ya se comentó anteriormente, estos tests solo daban como resultado valores constantes al ejecutar el modelo. Esto se debe a que se necesita utilizar la función *GET TIME VALUE* para poder hacer que la secuencia que proporcione la función *RANDOM 0 1* sea de números aleatorios. Dicha función no estaba implementada en la versión

actual de PySD. No obstante, la función *RANDOM 0 1* está implementada y no causa ningún error de parsing.

Se terminó añadiendo a la gramática las operaciones de compatibilidad de *subscripts*, solventando los errores encontrados en los sprints anteriores. Esta implementación se demoró debido a que no se había entendido con detalle el funcionamiento del parser utilizado, dando lugar al Riesgo 3, falta de formación en parsimonious (Tabla 2.5). No se encontraba la manera de acceder a los hijos del árbol de análisis para, así, poder obtener la información que se necesitaba. Finalmente, se preguntó a Diego Rodrigo sobre la posible solución y se pudo completar la gramática de *subscript mapping* y *subscripts copy*.

Para poder probar la gramática implementada, se partió de los ejemplos de la guía de usuario que proporciona *Vensim* [32]. Se crearon test unitarios para *sequence subscripts*, *subscripts copy* y *subscripts mapping*. Sin embargo, eran ejemplos incompletos y no se podían ejecutar en el entorno Vensim sin que se produjera ningún error. Debido al desconocimiento de la alumna sobre *Vensim*, se pidió ayuda a los desarrolladores expertos de modelos en *Vensim*, explicándoles los ejemplos que se necesitaban para así poder testear la nueva gramática implementada con ejemplos funcionales. Ocurrió el riesgo de la falta de conocimiento sobre *Vensim* para poder realizar tests, que corresponde al Riesgo 4 indicado en la Tabla 2.6. No obstante, cuando se terminó el Sprint 3 no se había recibido ningún ejemplo, por lo que la tarea de testear la gramática implementada está en progreso. La única consecuencia que tuvo el riesgo comentado, fue la demora de la tarea y tener que aplazarla para los sprints siguientes. Esta tarea se retomará cuando los expertos de *Vensim* nos proporcionen los ejemplos pedidos.

Como en los sprints anteriores, se actualizó la memoria en lo que respecta al sprint actual y se añadió la información precisa. En este sprint cabe destacar que se ha dedicado una mayor parte del tiempo a realizar la memoria, ya que se han añadido parte de los conocimientos adquiridos y asentados hasta el momento y esto ha permitido estructurar la memoria con una idea más clara y mejorada.

6.5. Sprint 4

Duración: del 27/1/2021 al 10/2/2021

El Sprint 4 se ha comenzado el día 27 de enero. Su fecha inicial estimada era el día 20, como se establece en la Planificación inicial, Tabla 2.2 de la Sección 2.4. Esto se debe a que la alumna estuvo confinada desde el día 7 de enero hasta el 18 de enero. Los exámenes de la convocatoria ordinaria estaban previstos para esas fechas y tuvieron que ser aplazados. Finalmente, los exámenes se realizaron el día 21 y 22 de enero, por lo que la *weekly* que se había establecido para el día 20 tuvo que verse aplazada una semana para que la alumna pudiera estudiar y centrarse únicamente en los exámenes. Así se ve materializado el riesgo por confinamiento debido a la situación sanitaria actual, expuesto en la Tabla 2.10, el cual como consecuencia deja con una semana de retraso al proyecto y ha sido necesaria una replanificación de los sprints.

Las tareas acordadas para este sprint se han enumerado en la Tabla 6.5.

Nombre de la Tarea	Tiempo estimado	Tiempo invertido	Estado
3.1 - Completar Tarea 1.1. Añadir tests para la función RANDOM 0 1	30m	30m	Completada
3.2 - Completar Tarea 1.2. Añadir a la gramática subscripts sequence	3h	15m	Completada
3.3 - Completar Tarea Tarea 2.2. Añadir a la gramática subscripts copy y añadir diccionario para subscripts copy	4h	15m	Completada
3.4 - Completar Tarea 2.3. Añadir a la gramática subscripts mapping y añadir diccionario para subscripts mapping	4h	4h 40m	Completada
3.5 - Añadir y ejecutar tests para las modificaciones de la gramática de compatibilidad de subscripts	3h	15m	En progreso
3.6 - Actualizar memoria seguimiento Sprint 3	4h	13h 25m	Completada
Total		**19h 20m**	**5/6**

Tabla 6.4: Tareas del Sprint 3

A finales del mes de diciembre, se aceptó la *pull request* que Diego Rodrigo había hecho, integrando sus cambios en PySD. Junto con las contribuciones de Eneko, se incorporaron todos los cambios a *master*. Se descargaron ambas versiones y se ejecutó *MEDEAS* con la nueva rama de *master* y fallaba en la traducción. Se han comparado las dos versiones de PySD para poder encontrar el fallo o los cambios que lo producen. Se ha invertido gran cantidad de tiempo en buscar las posibles diferencias entre ambas versiones, además de que se han ejecutado más archivos Vensim, no solo MEDEAS. Se invierte tanto tiempo en la búsqueda de las causas de fallo, debido a la inexperiencia con el proyecto PySD y la poca información que arroja *parsimonious* sobre los errores gramaticales.

Con la versión que tenía Diego Rodrigo se encontró un error con respecto a los guiones bajos. Las funciones de Vensim pueden escribirse con espacios, cuando sea necesario o, en su lugar, con guiones bajos. Por ejemplo, tendría el mismo efecto escribir la función *RANDOM 0 1* que *RANDOM_0_1*. La versión de Diego Rodrigo fallaba cuando se escribían las funciones con guiones bajos, como ocurría en el proyecto *WILIAM*. Sin embargo, en la nueva rama *master*, se ha implementado la función *add_entries_underscore* en el fichero *utils.py* que resuelve este problema.

Se ha intentado implementar la función de Vensim *SAMPLE_IF_TRUE* ya que PySD no tiene soporte para ella. En las *issues* de PySD aparece la necesidad de implementar esta función por lo que sería interesante añadir esta función para acompañar a una *pull request* y resolver dicha *issue* [41]. Se ha tenido que profundizar en su funcionamiento ya que esta función no era conocida por la alumna y era muy similar a la función *IF_THEN_ELSE*. Se ha estudiado las diferencias entre ambas funciones, ya que los parámetros que tienen son similares. Se ha implementado finalmente la función *SAMPLE_IF_TRUE*. La traducción se

realiza sin problema, sin embargo, la ejecución de dicha función ocasiona un error tras superar la máxima recursividad posible de una función. Este problema está relacionado con la caché y aún, en el Sprint 4, no se ha encontrado una posible solución.

También se ha continuado escribiendo la memoria, avanzando en el capítulo de Análisis, explicando más en profundidad las gramáticas con las que cuenta PySD y se ha comenzado a explicar PySD como un Modelo 4+1 vistas, para que fuera más fácil entender su funcionamiento. Se ha explicado en detalle el funcionamiento de Parsimonious y los Visitors asociados a cada regla. Además, se ha añadido el seguimiento del Sprint 4 correspondiente.

Nombre de la Tarea	Tiempo estimado	Tiempo invertido	Estado
4.1 - Ejecutar pruebas y mirar diferencias entre *master* y *new_functions*	7h	12h 15m	Completada
4.2 - Analizar la función *SAMPLE IF TRUE*	3h	1h 35m	Completada
4.3 - Actualizar memoria seguimiento Sprint 4	5h	17h 30m	Completada
Total		**31h 20m**	**3/3**

Tabla 6.5: Tareas del Sprint 4

6.6. Sprint 5

Duración: del 10/2/2021 al 24/2/2021

Las tareas acordadas para este sprint se presentan en la Tabla 6.6.

En este sprint, se han ejecutado nuevos ejemplos Vensim y se han intentado solventar los errores que se ocasionaban en la traducción. El archivo *WOLIM 1_5-to-share.mdl* generaba un error en la gramática *parse_general_expression*. Este error tenía lugar cuando se buscaba en el *namespace* el nombre de una variable y no se encontraba. Esto se debía a que se parseaba mal la sentencia y no se almacenaba en el *namespace*. Se ha comprobado que el error estaba causado porque el comentario en la definición de la variable contenía un número impar de comillas.

Al final, aunque el fallo se generase en la gramática *parse_general_expression*, el error se encontraba en la gramática *model_structure_grammar*, la cual parseaba mal la sentencia y no la almacenaba. En esta gramática los comentarios se definen con la regla "*element*", la cual determina que si se poseen comillas tiene que ser un número par. Puede darse el caso de que en algunas sentencias, como es este ejemplo, los comentarios no contienen un número par de comillas lo que hace que no se almacenen correctamente las variables del modelo. Para solucionar este fallo, se ha definido una nueva regla en la gramática *model_structure_grammar*

que define los comentarios de las sentencias Vensim como cualquier carácter excepto una virgulilla (“~”) o una tubería (“|”), los cuales son caracteres de control de las sentencias.

Se continuó ejecutando otro modelo *modelo vld 20201130.mdl*, el cual daba un error en el *builder* al intentar escribir el texto traducido en el archivo nuevo. El error ocurría cuando se intenta parsear e introducir saltos de línea y corregir la indentación en Python. Esto se debía a que faltan paréntesis para cerrar las funciones, es decir, estaban mal traducidas.

Más concretamente, esto estaba causado por la función *build_function_call* que se encuentra en el módulo *builder.py*, la cual se encarga de traducir las funciones Vensim. Esta función se utiliza en la gramática *parse_general_expression*. Dicho error se debe a algún cambio realizado en la nueva versión de PySD (master) ya que en la versión de Diego Rodrigo se traducían sin errores las funciones. Por esta razón, este ejemplo también se ha utilizado para poder comparar los cambios que presentan ambas versiones de PySD.

Se ha intentado implementar la función *SAMPLE IF TRUE*. Sin embargo, se continuaba con el problema de recursividad al ejecutar una sentencia como la que se muestra en el Fragmento de Código 6.1. En este ejemplo se define una variable a partir de la función *SAMPLE IF TRUE* la cual, a su vez, cuenta con la misma variable en uno de sus argumentos, exactamente en la condición de la función. Las variables y constantes en la traducción de Python se representan como funciones que devuelven un valor. Al ejecutar la traducción creada por PySD, se alcanza el máximo nivel de recursión debido a que se ha definido una función recursiva, es decir, que se define la función a partir de sí misma. Se genera un error debido a que no se cuenta con un caso base que haga terminar, en algún momento, la recursión.

Localizado este posible fallo con la función *SAMPLE IF TRUE*, la implementación de dicha función se ha dejado en progreso para poder completarla en el siguiente sprint y así poder probar si en la función *IF THEN ELSE*, que son parecidas, ocurre lo mismo cuando se trate de una sentencia recursiva.

```
1 max workforce=SAMPLE IF TRUE(
2         Workforce > max workforce, Workforce, Workforce)
```

Fragmento de código 6.1: Sentencia SAMPLE IF TRUE

Se han realizado los diagramas de clases que representan la vista lógica de PySD, para poder definir de una manera más sencilla el funcionamiento del proyecto. Se han realizado los diagramas de clases relativos a los módulos de PySD y las clases que representan las gramáticas. Sin embargo, falta pulir dichos modelos. También se ha completado y añadido el seguimiento de la memoria relativo al Sprint 5.

Nombre de la Tarea	Tiempo estimado	Tiempo invertido	Estado
5.1 - Implementar la función ***SAMPLE IF TRUE***	4h	2h 15m	En progreso
5.2 - Diagrama de clases para la vista lógica	3h	3h 30m	En progreso
5.3 - Analizar cambios entre distintas versiones de PySD	6h	4h 50m	En progreso
5.4 - Ejecutar diferentes ejemplos Vensim y corregir posibles errores	3h	12h 20m	Completada
5.5 - Actualizar memoria seguimiento Sprint 5	6h	8h 45m	Completada
Total		**31h 40m**	**2/5**

Tabla 6.6: Tareas del Sprint 5

6.7. Sprint 6

Duración: del 24/2/2021 al 10/3/2021

La Tabla 6.7 muestra el desglose de las tareas propuestas para este sprint.

En el sprint anterior no se pudo completar la implementación de la función *SAMPLE IF TRUE*, ya que se descubrió el fallo de recursividad que presentaba el modelo traducido en Python (explicado en el seguimiento del Sprint 5). En este sprint se ha comenzando estudiando si Vensim permite que exista una variable, por ejemplo: X, cuya definición esté compuesta por la función *IF THEN ELSE* y la condición de esta función esté formada por la misma variable X, para poder probar la recursividad.

Se ha buscado un ejemplo Vensim con una función *IF THEN ELSE* sencilla y se ha modificado para que se de esa situación de recursividad. Se ha abierto el modelo en el entorno Vensim y se ha realizado un *check model* que permite determinar si el modelo cuenta con algún error o *warning*. Tras modificar el modelo y abrirlo en el entorno Vensim, éste daba un error al comprobarlo, más concretamente: *ERROR: Simultaneous equations involving*. Lo que indica que Vensim no admite en la función *IF THEN ELSE* ningún tipo de recursividad. Sin embargo, con la función *SAMPLE IF TRUE* sí admite recursividad entre la variable que se define y los parámetros de la función.

En definitiva, teniendo en cuenta que variables y constantes de un modelo Vensim se traducen en PySD como funciones de Python, no es posible generar una traducción equivalente a la función *SAMPLE IF TRUE* por el problema de recursividad presentado anteriormente. Sin embargo, sí se puede realizar una traducción de la función *SAMPLE IF TRUE* para utilizarla en aquellos modelos que no tengan recursividad entre la variable y los parámetros. Se ha realizado la implementación de esta función para PySD, utilizando una variable global para poder guardar el valor de la simulación. Esta implementación genera un resultado

satisfactorio para ejemplos sin recursividad.

Con respecto al anterior sprint, se ha probado el correcto funcionamiento del cambio realizado en la gramática *model_estructure_grammar*. Se ha probado con el ejemplo *teacup.mdl*, ejemplo básico de Vensim, al cual se le han añadido 3 comillas a un comentario de una variable. PySD generaba una mala traducción en este caso, ya que parseaba mal la sentencia que contenía esas 3 comillas y, debido a esto, no almacenaba correctamente las variables del modelo. El mismo ejemplo con la modificación realizada en la gramática, generaba la traducción esperada, recalcando la necesidad de ese cambio. El ejemplo de *teacup.mdl* con el añadido de las 3 comillas puede servir entonces para ser usado como un test a la hora de añadir este cambio en el repositorio central del proyecto.

Tras probar varios ejemplos para poder solventar los problemas con *SAMPLE IF TRUE*, surgió la necesidad de que algunos modelos podían tener la extensión en mayúsculas, es decir, llamarse *archivo.MDL*, modelo que era válido para Vensim, lo abría y simulaba correctamente. En cambio, en PySD, generaba la traducción pero la escribía en el mismo fichero de partida, no generaba otro nuevo con extensión .py. Esto se debía a que no contaban con la opción de que el modelo pudiera tener la extensión en mayúsculas. Se ha implementado esta funcionalidad, además de añadir que se lance una excepción cuando se de el caso de que el modelo introducido que se quiera traducir no tenga terminación ni .mdl ni .MDL, es decir, no sea un modelo Vensim, aumentando la robustez del sistema.

En este sprint, los desarrolladores de modelos Vensim nos han mandado los tests que pedimos en el Sprint 3, relativos a los subscripts para poder comprobar la gramática implementada sobre mapping y copy de subscripts y la secuencia de subscripts.

Las reglas añadidas para poder parsear la compatibilidad de subscripts son válidas ya que dichos ejemplos no causan ningún error en el parsing. Sin embargo, faltaría añadir la funcionalidad para aquellos casos en los que haya que acceder a valores de subscripts que estén vinculados a otros, ya que a día de hoy PySD no tiene soporte para eso.

Entonces, se ha podido comprobar que la gramática añadida anteriormente para las operaciones de subscripts, es correcta.

Como en este sprint ya hemos contado con los tests de subscripts, se ha dedicado más tiempo a ello, ya que dicha tarea estaba pendiente, y no se ha podido iniciar la tarea relativa a encontrar las diferencias entre ambas versiones. Más concretamente, comprobar que la función *build_function_call* en el archivo *builder.py* tenga el funcionamiento esperado en la nueva versión de PySD. Por lo que esta tarea se pospone para el siguiente sprint.

Se ha actualizado y mejorado los diagramas de clases para poder representar las relaciones e interacciones entre clases de PySD. Falta por hacer los diagramas que representen las estructuras de datos utilizadas en PySD para realizar la traducción. Además, se ha añadido el seguimiento necesario del Sprint 6 a la memoria.

Nombre de la Tarea	Tiempo estimado	Tiempo invertido	Estado
6.1 - Completar Tarea 5.1. Implementar la función ***SAMPLE IF TRUE***	4h	3h 55m	Completada
6.2 - Completar Tarea 5.2. Diagrama de clases para la vista lógica	5h	12h 30m	En progreso
6.3 - Completar Tarea 5.3. Analizar cambios entre distintas versiones de PySD	4h	0 m	No iniciada
6.4 - Testear la modificación realizada en *model_structure_grammar*	6h	4h	Completada
6.5 - Extensión del modelo Vensim a traducir	2h	2h 10m	Completada
6.6 - Completar Tarea 3.5 - Añadir y ejecutar tests para las modificaciones de la gramática de compatibilidad de subscripts	5h	2h	En progreso
6.7 - Actualizar memoria seguimiento Sprint 6	6h	4h 25m	Completada
Total		**29h**	**4/7**

Tabla 6.7: Tareas del Sprint 6

6.8. Sprint 7

Duración: del 10/3/2021 al 24/3/2021

La Tabla 6.8 muestra el desglose de tareas acordadas para realizar en el séptimo sprint.

En el sprint anterior, al ejecutar los ejemplos de los subscripts, se descubrió que el resultado no era el mismo que el esperado. Se obtenía como resultado el nombre de la función a la que se llamaba o una sentencia de código en vez de un número (Tarea 7.4). Esto también ocurría cuando se intentó ejecutar la función *SAMPLE IF TRUE*, por lo que se dedujo que era un fallo que tenía PySD cuando guardaba los resultados de las ejecuciones. Por ejemplo, cuando se intentaba ejecutar el ejemplo de la función *SAMPLE IF TRUE* se obtenía como resultado: **<function max_workforce.<locals>.<lambda>at 0...>**.

En el caso de la función *SAMPLE IF TRUE* esto se debía a que se habían definido sus parámetros como expresiones lamba y por eso se obtenía ese resultado. Si se omitía esa definición de los parámetros, se generaba el resultado esperado. Con respecto a los subscripts, como se trata de matrices bidimensionales, para poder obtener el resultado numérico correcto hay que filtrar el resultado con las dimensiones del subscript que se quiera comprobar. Es decir, en la función *run* que ejecuta el modelo ya traducido, se debe añadir el parámetro *return_columns* e igualarlo al nombre de la variable de la cual se quiera conocer el resultado junto a las dimensiones del subscript. Un ejemplo sería: *return_columns=['Level_2[A,B]']*, siendo *Level_2* la variable y *A* y *B* los subscripts asociados.

En los tests que recibimos relativos a subscripts, se encontraba un modelo simple de subscripts, uno que realizaba un copy de susbscripts y otro sobre el mapping de subscripts. En este proyecto se ha implementado la secuencia de subscripts (Sección 3.2.2), por lo que faltaba un test que definiera un subscript con una secuencia.

Para generar este nuevo test, se ha partido de un modelo, de los tests recibidos, que define un subscript de la forma representada en el Fragmento de código 6.2. Se ha intercambiado dicha sentencia por su equivalente, representada en el Fragmento de código 6.3, en la cual se define el mismo subscript pero con un rango numérico.

```
1 DimA: da1,da2,da3,da4,da5 ~~|
```

Fragmento de código 6.2: Sentencia Subscript

```
1 DimA: (da1-da5) ~~|
```

Fragmento de código 6.3: Sentencia Sequence Subscript

Como estas dos definiciones de subscripts son equivalentes, el resultado del primer modelo contemplado debe ser igual al resultado del test creado con la secuencia. Tras ejecutarlo, el resultado ha sido el esperado, por lo que se puede concluir que la definición de un subscript a partir de un rango numérico implementada en la gramática funciona correctamente para PySD.

Sin embargo, el mapping y copy de subscripts no reflejan ningún error gramatical, pero la funcionalidad necesaria no está implementada. Por lo que al ejecutar la traducción del modelo, falla.

Para cada subscript vacío, sin subscript values asociados, que se define como *copy* de otro (Sección 3.2.2) se le han añadido los *subscripts values* del subscript copia. Con respecto a los subscripts que se mapean, se ha determinado unificar los subscripts que utilizan. Sin embargo, esta funcionalidad no se ha podido completar en este sprint.

También, en este sprint se ha intentado abordar el fallo de la función *build_function_call*, la cual es la encargada de traducir las funciones a Python y, como se ha comentado en sprints pasados, daba algún fallo en las últimas modificaciones de PySD en casos concretos. Sin embargo, no se ha podido completar por la dificultad encontrada y el tiempo empleado en la implementación de subscripts copy y mapping.

Con los cambios realizados anteriormente, como la implementación de la función *RANDOM 0 1*, la nueva definición de los comentarios en la gramática *model_structure_grammar* y los cambios realizados sobre la extensión de un modelo Vensim, se ha realizado una *pull request* al repositorio principal. La *pull request* en cuestión se encuentra en: `https://github.com/JamesPHoughton/pysd/pull/247`. El creador de PySD pidió algunos cambios con respecto a esta *pull request*, pero no dio tiempo a realizarlos en este sprint por lo que se finalizará esta tarea en el siguiente sprint.

Como en todos los demás sprints, se ha añadido la información necesaria relativa al Sprint 7 a la memoria. Además, se han terminado de pulir los diagramas de clases que se utilizan en la vista lógica de PySD.

Nombre de la Tarea	Tiempo estimado	Tiempo invertido	Estado
7.1 - Completar Tarea 5.2. Diagrama de clases para la vista lógica	3h	30 m	Completada
7.2 - Completar Tarea 5.3. Analizar cambios entre distintas versiones de PySD	2h	45m	En Progreso
7.3 - Completar Tarea 3.5 - Añadir y ejecutar tests para las modificaciones de la gramática de compatibilidad de subscripts	1h 30m	10m	Completada
7.4 - Analizar el valor que devuelve para algunas variables la nueva versión de PySD	2h	3h	Completada
7.5 - Realizar la primera contribución en PySD	3h	5h 35m	En Progreso
7.6 - Implementar funcionalidad adecuada para subscript mapping y copy	8h	15h	En Progreso
7.7 - Actualizar memoria seguimiento Sprint 7	5h	5h 25m	Completada
Total		**30h 25m**	**4/7**

Tabla 6.8: Tareas del Sprint 7

6.9. Sprint 8

Duración: del 24/3/2021 al 7/4/2021

En la Tabla 6.9 se muestra el desglose de tareas propuestas para este sprint.

Continuando con la *pull request* realizada, se han realizado los cambios que pidieron a mayores para poder mergear a la rama principal del proyecto. Los cambios que se pidieron eran que los modelos Vensim de tests añadidos, utilizados para comprobar los cambios realizados, se añadiesen al submódulo de tests (SDXorg/test-models). Se hizo un fork de este repositorio y se añadieron estos modelos de tests nuevos. Después se realizó una *pull request* al repositorio original de los tests. Además, se cambiaron los tests al archivo *integration_test_vensim_pathway.py* donde se encuentran todos los tests que comprueban el correcto funcionamiento de PySD.

En el submódulo de tests, se encontró un modelo Vensim que utilizaba la función *SAMPLE IF TRUE*, el cual no contenía ninguna recursividad entre la variable y los parámetros de

la función, problema anteriormente explicado. Por lo que se probó con ese modelo la función *SAMPLE IF TRUE* anteriormente implementada. Falló, debido a que contenía *DataArrays* y no se había añadido soporte para los mismos. Aún así, se ha dedicado bastante tiempo a modificar la función ya que la presencia de *DataArrays* y su manejo ha supuesto un aprendizaje para poder utilizarlos y modificarlos adecuadamente. Resumidamente, los parámetros que recibe la función son: una condición, un valor inicial que se utiliza en el primer step de la simulación y un valor actual. Además, se guarda un diccionario con el valor que se almacena entre los diferentes steps cuando se ejecuta el modelo. En primer lugar, se vio que el primer error cometido a la hora de implementar dicha función era que se guardaba un único valor para todas las funciones *SAMPLE IF TRUE* que pudiera haber en un modelo. Tras encontrar el ejemplo del submódulo, en el que se definían varias variables con la función *SAMPLE IF TRUE*, se vio que siempre que se tuviera que devolver el valor almacenado, se devolvía el mismo valor independientemente de la variable que se tratase. Esto se solucionó añadiendo un parámetro más a la función, el nombre de la variable a la que esa función estaba asociada, para así poder identificar los valores almacenados de cada variable. Es decir, se guarda el valor almacenado junto al nombre de la variable en la que se encuentra, para poder identificar los diferentes valores almacenados. Además, se cambió la forma de guardar los valores, y se convirtió el único valor almacenado en un diccionario con valores almacenados, en el cual se añadía una entrada cada vez que se correspondía al primer *step* de la simulación. Dicho diccionario de valores almacenados está compuesto por el nombre de la variable como *key* y el último valor almacenado como *value*.

Además, en la función implementada solamente se había tenido en cuenta el caso más simple, que se daba cuando la condición y los valores inicial, actual y almacenado, eran de tipos simples como *bool* o *int*. Sin embargo, la condición podría ser un subscript, es decir, se traduce como un *DataArray*. En otras palabras, lo que se obtiene cuando la condición es un subscript es que para una sola variable almacena varios valores diferentes, como por ejemplo una condición que almacene [False, True, False] que hacen referencia a las dimensiones 'A', 'B' y 'C'. Por consiguiente, no se puede almacenar un único valor para esa variable ya que hay que tener en cuenta cada valor de la condición y almacenar un valor u otro dependiendo de ésta. Entonces, se cambió el valor almacenado por un *DataArray* cuando la condición lo fuera, además de que el valor actual podía ser o no *DataArray*, por lo que hubo también que redimensionar el *DataArray* del valor guardado en este último caso. También, se debía comprobar la condición con las dimensiones adecuadas y actualizar el valor almacenado solo en las coordenadas necesarias.

El tercer parámetro de *SAMPLE IF TRUE* es el valor inicial, con el que se inicializa el valor almacenado. Éste siempre se ha tratado como si fuera de tipo simple y con él, en un principio, se rellenaban todas las dimensiones del *DataArray* que almacenaba los valores. Posteriormente, se encontró la posibilidad de que el valor inicial pudiese ser un *DataArray*, ya que al final las diferentes coordenadas del *DataArray* podían estar inicializadas con valores diferentes. Para esto hubo que hacer una ligera modificación donde se inicializaba el valor almacenado, teniendo en cuenta las diferentes coordenadas y dimensiones de los *DataArrays*.

Tras estos cambios y combinaciones posibles entre los parámetros de *SAMPLE IF TRUE*, el modelo de test que se encontraba en el submódulo se traducía correctamente y generaba la salida esperada.

También se intentó corregir el error anteriormente comentado, que tenía que ver con la función *build_function_call* del módulo *builder*, la cual se encargaba de generar la traducción de una función en Vensim para Python. A dicha función se le pasan dos parámetros, el nombre de la función y la lista de parámetros asociada. En la versión de PySD de Diego Rodrigo, el error no ocurría y se traducían las sentencias correctamente. Este error se daba cuando una función tenía como parámetro otra, por ejemplo como se presenta en el Fragmento de Código 6.4. No es una sentencia correcta, faltaría la variable a la cual se le iguala el resultado de *IF THEN ELSE*, pero se ha omitido para simplificar y que se entienda mejor el error.

```
IF THEN ELSE(MODULO(Time,Acctg Period)<0.5, (cash-cash with INTEG)/
    TIME STEP,0)
```

Fragmento de código 6.4: Parámetro argumentos en *build_function_call*

En esta sentencia se expone una función *IF THEN ELSE*, la cual tiene tres parámetros, una condición, un valor si la condición es cierta y un valor si la condición no es cierta, en ese orden. El primer parámetro, la condición, está compuesto a su vez por otra función, *MODULO*.

En PySD, a la función *build_function_call* como parámetros se le pasaban el nombre de la función, en este caso es *if_then_else*, y una ristra de los parámetros de ésta ya traducidos a su equivalente en Python. En la última versión de PySD, la lista de parámetros no se completaba correctamente ya que, donde parseaban la sentencia para obtener los argumentos, lo único que se hacía era separar éstos por comas. En el caso en el que se trate de una función sin mayor complejidad, esa lógica de separar los parámetros por comas tiene sentido, ya que los parámetros se dividirán por comas y se obtendrá el resultado esperado. Pero cuando se trata de una función dentro de otra función, como el ejemplo anteriormente presentado, dará fallo. Lo que ocurría era que la función *MODULO* se partía a la mitad, ya que se obtenía *np.mod(time()* como primer parámetro, es decir, donde se encontraba la primera coma. En la versión de PySD de Diego Rodrigo, la forma de separar los argumentos de una función era diferente, éstos se iban comparando con una lista de argumentos que se rellenaba en el visitor de una regla que definía los mismos. Se cambió la manera de pasarle a la función *build_function_call* los argumentos, mediante la lógica que había definido Diego Rodrigo en su versión y funcionaba correctamente. Posteriormente, se miró la *pull request* que había realizado Diego Rodrigo a PySD con sus cambios y dicho cambio sí se encontraba en su versión pero al mergear su rama a master no lo añadieron.

Se ha continuado intentando implementar la funcionalidad relativa a subscript mapping y copy pero las soluciones que se han encontrado no eran válidas. Se sigue trabajando en ello en el siguiente sprint.

También se ha continuado dedicando tiempo a la escritura de la memoria, así como a explicar en detalle los diagramas que definen *PySD*, a medida que se van realizando más cambios en el proyecto y conociendo más exhaustivamente su funcionamiento.

Nombre de la Tarea	Tiempo estimado	Tiempo invertido	Estado
8.1 - Completar Tarea 5.3. Analizar cambios entre distintas versiones de PySD	4h	1h 50m	Completada
8.2 - Completar Tarea 7.6. Implementar funcionalidad adecuada para subscript mapping y copy	8h	1h 30m	En Progreso
8.3 - Completar 7.5. Realizar la primera contribución en PySD	3h	4h 15m	En Progreso
8.4 - Corrección de la función *SAMPLE IF TRUE*	5h	10h 20m	Completada
8.5 - Actualizar memoria seguimiento Sprint 8	5h	11h	Completada
Total		**28h 55m**	**3/5**

Tabla 6.9: Tareas del Sprint 8

6.10. Sprint 9

Duración: del 7/4/2021 al 21/4/2021

Las tareas que se han previsto realizar en este sprint se exponen en la Tabla 6.10.

En la primera contribución que se realizó, James preguntó antes de unificar los cambios. Preguntó si en las sentencias Vensim no se podía añadir ni virgulillas ni tuberías aunque fuera entre comillas. Esto se probó en el entorno gráfico de Vensim, y lo remarcaba en amarillo y lo omitía. Además, sugirió añadir un *warning* a la función *RANDOM UNIFORM* de la cual se había aportado información. Se ha añadido lo pedido y se ha realizado un nuevo commit a la pull request con los cambios necesarios. Tras contestar estas preguntas, se aceptó la *pull request*, añadiendo los cambios a la rama *master* de PySD. A la vez se aceptó la *pull request* del repositorio de los tests de Vensim.

Mientras tanto, se ha creado otra rama en el repositorio para poder ir añadiendo los cambios a la segunda pull request, en la cual se quiere añadir la funcionalidad implementada de la función *SAMPLE IF TRUE*, la funcionalidad para subscript numeric range y el cambio que había hecho Diego Rodrigo en los argumentos que se pasaba por parámetro a la función *build_function_call* y que no se había añadido.

Se han encontrado ciertos problemas con el idioma de Excel, ya que en español tiene las comas como separador decimal, mientras que, en los ficheros de salida que contienen los resultados de la simulación de un modelo, el separador utilizado es el punto. Se ha demorado bastante la tarea hasta encontrar este fallo.

Este fallo se ha descubierto cuando se ha querido añadir al repositorio de los tests de Vensim un modelo para poder testar la implementación del *subscript sequence*. Para poder

añadir el modelo que contiene el *subscript sequence* es necesario añadir a mayores los resultados que genera la simulación en Vensim. Por lo que ha sido importante detectar ese error con los ficheros en Excel y poder obtener la salida en el formato correcto.

Además, cuando se ha ido a realizar una *issue* para poder añadir y explicar el *subscript sequence*, se ha encontrado un ejemplo del tutorial de Vensim (`http://www.vensim.com/documentation/22090.html`) que define a lo que habíamos llamado *subscript sequence* como *numeric range*. En este ejemplo hay una sentencia que no se había tenido en cuenta para añadirla a la gramática, ya que se pensaba que el rango de un subcript siempre iba a ser una sentencia como la presentada en el Fragmento de Código 6.5. Sin embargo, en el tutorial de Vensim se ha hallado con la sentencia representada en el Fragmento de Código 6.6.

```
DimA: (da1-da5) ~~|
```

Fragmento de código 6.5: Subscript Sequence

```
lot : (LOT1-LOT3),LOT12,(LOT14-LOT16) ~~|
```

Fragmento de código 6.6: Subscript Numeric Range

Tras ver que podía haber otra manera de definir estos rangos númericos, se ha tenido que cambiar la gramática para que fuera posible añadir más de un rango numérico y valores individuales. Además de cambiar los nombres de las funciones y reglas gramaticales necesarias, se ha modificado el nombre de *subscript sequence* por *subscript numeric range* para que coincidiese con el que se utiliza en el tutorial de Vensim.

Después de hacer los cambios gramaticales necesarios, se ha continuado añadiendo los cambios que implementan la función *SAMPLE IF TRUE*, para presentarlo en esta segunda *pull request*. Además, se han agregado los cambios en la manera de separar los argumentos que Diego Rodrigo había añadido en su TFG y que, finalmente, no se había incorporado a PySD.

Se han creado dos *issues*, una definiendo los *subscript numeric range* y otra para explicar el problema que presentaba PySD al tener una función como parámetro de otra, que se solucionaba con el cambio en la manera de separar los argumentos.

Después, se ha realizado una *pull request* al repositorio de los tests de Vensim, para añadir el test de *subscript numeric range*. Dicha *pull request* se encuentra en el siguiente enlace: `https://github.com/SDXorg/test-models/pull/70`. Como se ha comentado antes, en ella se agrega una imagen del modelo, el modelo Vensim y un archivo *output.tab* que contiene la salida esperada de la simulación.

Finalmente, se ha abierto la *pull request* al repositorio de PySD, explicando y referenciando las *issues* que se resolvían con los nuevos cambios. Esta *pull request* se encuentra en el siguiente enlace: `https://github.com/JamesPHoughton/pysd/pull/252`.

Se ha continuado la tarea de modificar el proyecto para que funcionasen y no dieran error de ejecución el mapeo de subscripts. Se encontró una forma y se terminó de implementar,

la cual era unificar todos los valores de un mapeo y cambiar las apariciones de un subscript por el subscript con el que estaba mapeado. Tras realizar los cambios necesarios en la clases que ejecutan y reproducen la simulación, los resultados obtenidos no eran los esperados que se habían obtenido con Vensim. Esta tarea se continúa en el siguiente sprint, ya que no se ha podido encontrar el fallo o si el fallo no tiene nada que ver con el mapeo de subscripts y es un problema de integración que no se había detectado antes.

Además, se ha actualizado la memoria del proyecto con lo relativo al seguimiento del Sprint 9.

Nombre de la Tarea	Tiempo estimado	Tiempo invertido	Estado
9.1 - Completar Tarea 7.6. Implementar funcionalidad adecuada para subscript mapping y copy	10h	9h 45m	En Progreso
9.2 - Completar 7.5. Realizar la primera contribución en PySD	3h	45m	Completada
9.3 - Realizar la segunda contribución en PySD	5h	10h 30m	En Progreso
9.4 - Corrección subscript sequence	3h	1h 35m	Completada
9.5 - Actualizar memoria seguimiento Sprint 9	6h	7h 05m	Completada
Total		**29h 40m**	**3/5**

Tabla 6.10: Tareas del Sprint 9

6.11. Sprint 10

Duración: del 21/4/2021 del 5/5/2021

Las tareas previstas para desarrollar en este sprint se han expuesto en la Tabla 6.11.

Comenzando por la 2ª *pull request* que se había hecho, continúa abierta y se ha dedicado tiempo a esta tarea para responder algunas dudas que surgieron sobre ella. Además, de las preguntas que se habían hecho en dicha *pull request*, se tuvo que modificar ligeramente la implementación que se había desarrollado para *numeric range* de subscripts, ya que un colaborador proporcionó un ejemplo de nombres para el rango que no funcionaban con lo implementado. Este rango era: (Y2020M1 - Y2020M12), es decir, intercalando números y letras. La solución que en un principio se implementó, solo soportaba nombres que empezasen con letras y acabasen en números, no intercalados. Este caso de prueba se tuvo que testar en la escuela, debido que los subscripts solo están disponibles en la licencia profesional de Vensim, por lo que se produjo eln riesgo de que la funcionalidad Vensim con la que contaba la alumna en su ordenador personal no era suficiente, Riesgo 9 (Tabla 2.11). Desde la licencia

de la escuela, se comprobó que dicha sentencia era válida, por lo que se tuvo que modificar la gramática implementada. A mayores, se hizo una pequeña batería de pruebas para ver que tipo de nombres Vensim permitía en el rango numérico de subscripts. Esta incidencia se representa en la Tarea 10.3. Con los resultados obtenidos de esta batería de pruebas, se implementó ese comportamiento en la función desarrollada, lanzando errores similares a los que se generaban en Vensim. Tras esto, se volvió a realizar un *commit* en la *pull request* ya abierta con los cambios realizados.

Además, tras estos cambios, un contribuidor comentó en la *pull request* sugiriendo un ligero cambio de la función *Sample if true* para que fuera más consistente con *PySD*. Este cambio se realizará en el siguiente sprint, por falta de tiempo.

En el anterior sprint no se pudo completar la funcionalidad adecuada para el mapeo de subscripts, por lo que, en este nuevo sprint, se ha continuado dicha tarea. Se había intentado implementar la funcionalidad unificando los nombres de subscripts que estaban mapeados y, simplemente, se intercambiaban todas las apariciones de un subscript por el que estaba mapeado para así unificar las dimensiones de las variables y que la ejecución no proporcionara ningún error. Cuando se hizo este cambio, se comparó el resultado obtenido de la ejecución por PySD y el resultado que proporcionaba Vensim de ese modelo. Ambos resultados, eran ligeramente diferentes.

Se materializó el Riesgo 3 (Tabla 2.5) ya que faltaba formación sobre el lenguaje Vensim, por lo que fue necesario preguntar a un experto de este lenguaje, si al eliminar el mapeo y unificar los subscripts, el resultado debía ser igual al modelo que contiene el mapeo. En un principio, se pensaba que esto era así y que los subscripts no influían en el resultado de la ejecución, que únicamente eran dimensiones para las variables. Finalmente, el resultado no era igual y el programador de modelos Vensim dijo que ese resultado variaba dependiendo de si hubiera subscripts mapeados o si éstos no estuvieran mapeados.

Tras esto, se ejecutaron, con la versión profesional de Vensim, unos modelos que contenían subscripts mapeados y otro igual pero con los subscripts que estaban mapeados "unificados". El resultado variaba ligeramente, por lo que se sospechó que igual podría deberse a que se cambiaba la manera de integración en este caso. De todas formas, se envió un correo a un experto en Vensim preguntando sobre cómo o de qué manera influían los subscripts mapeados en la ejecución del modelo o a qué se debía ese ligero cambio en los resultados. Por lo que, en este sprint, esta tarea quedó a la espera de recibir una contestación para poder continuar con la implementación adecuada.

Se ha comenzado a traducir y convertir a markdown el estudio sobre PySD que se presenta en la Sección de Análisis de PySD (Sección 4.2). Esto se ha hecho con vistas a realizar otra contribución en PySD para explicar el funcionamiento interno de PySD y que sea más fácil entenderlo y, por consiguiente, poder colaborar en él. No se ha podido terminar de traducir toda la sección por lo que se completará en el siguiente sprint.

También, se ha actualizado ciertas partes de la memoria y se ha añadido el seguimiento del Sprint 10.

Nombre de la Tarea	Tiempo estimado	Tiempo invertido	Estado
10.1 - Completar Tarea 7.6. Implementar funcionalidad adecuada para subscript mapping y copy	10h	6h	En Progreso
10.2 - Completar Tarea 9.3. Realizar la segunda contribución en PySD	3h 30m	4h 40m	En Progreso
10.3 - Corrección del nombre que pueden tener los elementos de un rango numérico	2h	1h 55m	Completada
10.4 - Traducir y convertir a markdown el Modelo 4+1 vistas	8h	7h 40m	En Progreso
10.5 - Actualizar memoria seguimiento Sprint 10	8h	10h 35m	Completada
Total		**30h 50m**	**2/5**

Tabla 6.11: Tareas del Sprint 10

6.12. Sprint 11

Duración: del 5/5/2021 al 19/5/2021

Las tareas que se han propuesto para llevar a cabo en este sprint, se encuentran esquematizadas en la Tabla 6.12.

Se ha continuado con la 2ª *pull request*. En ella, se había realizado la implementación de la función *SAMPLE IF TRUE*, pero uno de los colaboradores propuso realizar esta implementación de una manera más eficiente y consistente con *PySD*. Se trataba de realizar una clase que heredase de *Stateful* para que así se pudiera actualizar el estado de las variables que utilizan *Sample if true*, o lo que es lo mismo, instanciar objetos de la clase *SampleIfTrue* con estado para poder simular el comportamiento de la función. Además de cambiar la manera de implementar esta funcionalidad, se mejoró la información de los mensajes de error que se lanzaban en el rango numérico de subscripts (añadido en esta *pull request*). Se volvió a hacer un *push* con los nuevos cambios a esa *pull request* abierta para que pudieran opinar, realizar más sugerencias o añadirlos finalmente al repositorio central.

Tras estas modificaciones de la segunda *pull request*, ésta fue añadida al repositorio central mergeando la rama con los cambios realizados.

Se continúo traduciendo el análisis que se ha realizado sobre todo el proyecto PySD y escribiéndolo en markdown, con vistas de presentarlo en una *pull request* nueva. Esto ayudaría a los posibles futuros contribuidores que quieran formar parte y mejorar el proyecto. Se ha comenzado a realizar un diagrama de secuencia para poder añadir a la vista de escenarios, que no se había realizado antes.

Retomando la tarea relativa a implementar la funcionalidad correspondiente a el mapeo

de subscripts y el copy, se recibió respuesta por parte de los programadores de Vensim. Como se había pensado en un principio, los subscripts mapeados no variaban el resultado de la simulación. Finalmente, en el modelo de prueba que habían enviado los programadores de Vensim faltaba un mapeo entre dos subscripts, es decir, estaba una sentencia de la forma: DimB -> DimC, pero faltaba la sentencia: DimC -> DimB. Al añadir esta sentencia el resultado proporcionado por Vensim era el mismo que devolvía PySD. Al realizar este mapeo en ambas direcciones causa el mismo efecto que si fuera un copy entre ambos subscripts.

Se ha decidido tener cuanto antes implementada en PySD la funcionalidad de copy de subscripts y la funcionalidad 'simple' de mapping de subscripts para, junto a la documentación del análisis en markdown, poder realizar otra *pull request* al repositorio. La funcionalidad de mapear varios subscripts es mucho más amplia de lo que se ha presentado en este proyecto, ya que se pueden mapear únicamente *subranges* (subrangos) de un subscript con otro. Esto requiere mucho más estudio sobre cómo afecta al resultado de la simulación y cómo sería la mejor manera de implementarlo. Debido al tiempo, se ha decido implementar únicamente aquella funcionalidad de mapeo de subscripts que tiene, a fin de cuentas, el mismo resultado que el copy de subscripts. Se ha continuado intentando desarrollar dicha funcionalidad, pero se continuará en el siguiente sprint.

Además, se ha añadido el seguimiento relativo al Sprint 11 a la memoria del proyecto.

Nombre de la Tarea	Tiempo estimado	Tiempo invertido	Estado
11.1 - Completar Tarea 7.6. Implementar funcionalidad adecuada para subscript mapping y copy	6h	9h 10m	En Progreso
11.2 - Completar Tarea 9.3. Realizar la segunda contribución en PySD	5h	2h 40m	Completada
11.3 - Completar Tarea 10.4. Traducir y convertir a markdown el Modelo 4+1 vistas	8h	11h	Completada
11.4 - Redefinir la función *SAMPLE IF TRUE*	6h	5h	Completada
11.5 - Completar Vista de Escenarios con Diagrama de Secuencia	4h	2h 10m	En Progreso
11.6 - Actualizar memoria seguimiento Sprint 11	3h	1h	Completada
Total		**31 h**	**4/6**

Tabla 6.12: Tareas del Sprint 11

6.13. Sprint 12

Duración: del 19/5/2021 al 2/6/2021

En la Tabla 6.13 se presentan las tareas propuestas para realizar en el Sprint 12.

Se ha comenzado este sprint continuando las tareas pendientes del anterior sprint, completando el diagrama de secuencia para añadirlo a la vista de escenarios e implementar la funcionalidad de subscript copy así como de subscript mapping simple. Se ha priorizado la realización de éstas ya que urge crear una nueva *pull request* en PySD para poder añadir cuanto antes los nuevos cambios.

Se ha realizado la *pull request*, que se encuentra en el siguiente enlace: `https://github.com/JamesPHoughton/pysd/pull/260`. Se sugirió convertir el documento sobre el análisis para desarrolladores de PySD a rst (reStructuredText) [57], un lenguaje de marcas ligero, similar a *markdown*, ya que es el que se utiliza para generar la documentación asociada a PySD. Además, cabe destacar que tuvo buena acogida la documentación añadida, comprobando realmente su necesidad en el proyecto para los futuros contribuidores.

Finalmente, se integraron los cambios de esa *pull request* en el repositorio central y se actualizó la documentación de PySD con el trabajo de análisis que se había adjuntado, pudiendo acceder a ella a través del siguiente enlace: `https://pysd.readthedocs.io/en/master/development/pysd_architecture_views/4+1view_model.html`.

Tras tener los nuevos cambios se probó a traducir el modelo WILIAM. Surgió un error debido a que faltaba la implementación de name box a la hora de escoger las casillas en la función *GET SUBSCRIPT DIRECT*. Un *name box* en Excel determina un rango de celdas y en la Wiliam se utiliza name box para determinar en la función *GET SUBSCRIPT DIRECT* cuáles son las celdas del Excel con las que se tiene que trabajar en dicha función. Tras ver este error, se abrió una nueva *issue* en la que se indicaba el problema y se adjuntaba el fichero Excel y el modelo Vensim que se había probado.

En este sprint, se ha dedicado una gran parte del tiempo a redactar algunos capítulos de la memoria pendientes.

6.14. Resumen y calendarización final

En la Tabla 6.14, se presenta un resumen en el cual se contabiliza algunos de los hechos más relevantes en este proyecto sobre PySD, como son las *pull requests* realizadas, *issues* añadidas, etc.

En la Tabla 6.15, se encuentra la planificación final que realmente se ha llevado a cabo, indicando brevemente aquellas causas por las que la planificación inicial del proyecto ha tenido que verse pospuesta.

Nombre de la Tarea	Tiempo estimado	Tiempo invertido	Estado
12.1 - Completar Tarea 7.6. Implementar funcionalidad adecuada para subscript mapping y copy	6h	10h	Completada
12.2 - Completar Tarea 11.5. Completar Vista de Escenarios con Diagrama de Secuencia	6h	5h 30m	Completada
12.3 - Realizar tercera contribución en PySD	5h	10h 45m	Completada
12.4 - Añadir issue a PySD con el error visto	30m	45m	Completada
12.5 - Actualizar memoria seguimiento Sprint 12	8h	24h 10m	Completada
Total		**51h 10m**	**5/5**

Tabla 6.13: Tareas del Sprint 12

Nombre	**Número de veces**
Issues añadidas a PySD	10
Issues cerradas de PySD	5
Pull Requests abiertas en PySD ... de las cuales cerradas	3 3
Pull Requests abiertas en test-models ... de las cuales cerradas	3 3
Modelos Vensim añadidos a test-models	5

Tabla 6.14: Tabla resumen final

6.15. Costes simulados finales

La duración del proyecto finalmente fue de 8 meses y medio, por lo que los costes tanto de personal, como de depreciación del ordenador y de licencias software calculados en la Sección 2.6 deben ser actualizados.

La duración final en horas del proyecto ha sido de 389 horas y 20 minutos, incrementando el tiempo inicial estimado. Siguiendo con la misma métrica calculada en la Sección anteriormente citada, el coste de plantilla por horas son 11,32 €. Total: **4.403,48 €**.

Los gastos generales finales teniendo en cuenta el valor previo, calculado en la Sección 2.6, del precio medio al día de la luz es de 0,11059 €/kWh y contemplando que la duración del proyecto finalmente ha sido de 389 horas, los gastos generales ascenderían a la cantidad de **43,02 €**.

La depreciación del ordenador calculada era de 519,8 €/año. Debido a que finalmente

Sprint	Fecha de comienzo	Fecha de finalización	Observaciones
Sprint 0	14/09/2020	21/10/2020	
Sprint 1	21/10/2020	4/11/2020	
Sprint 2	4/11/2020	18/11/2020	
Exámenes parciales			
Sprint 3	2/12/2020	16/12/2020	
Entregas finales, Navidad y convocatoria ordinaria			
Cuarentena por COVID-19, convocatoria ordinaria aplazada			
Sprint 4	27/1/2021	10/2/2021	
Sprint 5	10/2/2021	24/2/2021	
Sprint 6	24/2/2021	10/3/2021	
Sprint 7	10/3/2021	24/3/2021	
Sprint 8	24/3/2021	7/4/2021	
Sprint 9	7/4/2021	21/4/2021	
Sprint 10	21/4/2021	5/5/2021	
Sprint 11	5/5/2021	19/5/2021	Sprint de refuerzo
Sprint 12	19/5/2021	2/6/2021	Sprint de refuerzo

Tabla 6.15: Calendarización final del proyecto

el proyecto se alargó y duró 8 meses y medio, la **depreciación del ordenador** para este periodo de tiempo es de **368,19 €**.

El coste relativo a las licencias se verá aumentado por la duración del proyecto. La licencia de *Vensim DSS* tiene un coste de 1636,73 € por usuario al año. Teniendo en cuenta la prolongación en el tiempo del trabajo, para 8 meses y medio la licencia de ***Vensim DSS*** tiene una coste final de **1159,35 €**.

Continuando con la licencia de *Astah Professional*, según lo calculado en la Sección 2.6 antes citada, se calculó un costo de 33,35 € al año. El coste final de la licencia asociada a *Astah Professional* ascendería a **23,62 €** en base al incremento en la duración del proyecto.

Para *Visual Paradigm* se utiliza una licencia que conlleva un costo de 15,85 €/mes, de acuerdo con el aumento del tiempo equivale a **134,73 €** para el proyecto.

Las tecnologías *GitLab*, *GitHub*, *Visual Studio Code* y *Overleaf*, utilizadas en el desarrollo de este trabajo, tienen licencias de uso abiertas por lo que no aumentan el presupuesto del proyecto.

Por lo tanto, las licencias de *Astah Professional*, *Vensim DSS* y *Visual Paradigm* acumulan un presupuesto total de **1317,7 €** siendo éste el coste asociado a las licencias software.

En la Tabla 6.16 se recogen todos estos costos anteriormente razonados y la suma de los mismos, obtieniendo un total de **6.132,39 €** como coste simulado final de este Trabajo Fin de Grado.

Costes de Plantilla	4.403,48 €
Gastos Generales	43,02 €
Material	368,19 €
Licencias software	1.317,7 €
Total	**6.132,39 €**

Tabla 6.16: Costes simulados finales

6.16. Costes reales finales

Finalmente, aún habiendo un aumento en la duración del proyecto, los costes de plantilla reales asociados a éste son los mismos que los calculados en la Sección 2.7, **1.800 €** como resultado de la beca por 6 meses.

Los gastos generales son idénticos a los calculados en la Sección 6.15, es decir, **43,02 €**.

Los gastos relativos al material con el que se ha trabajado en el desarrollo del proyecto son análogos a los calculados en la Sección 6.15, siendo **368,19 €**.

Los costes asociados a las licencias de software las ha proporcionado la Universidad de Valladolid, por lo que su coste no aumenta el presupuesto de este proyecto.

En la Tabla 6.17 se presenta un resumen con todas los costes anteriormente justificados, acompañados del total que suman, ascendiendo esta cantidad a **2.211,92 €** como coste real final de este Trabajo Fin de Grado.

Costes de Plantilla	1.800 €
Gastos Generales	43,02 €
Material	368,9 €
Licencias software	0 €
Total	**2.211,92 €**

Tabla 6.17: Costes reales finales

Capítulo 7

Diseño de las modificaciones

En esta sección se exponen y se explican todas las mejoras y contribuciones realizadas al código de PySD.

Se han corregido bugs, se han implementado funciones de Vensim para las cuales PySD aún no tenía soporte y se han añadido algunas operaciones sobre subscripts, además de mejorar la documentación para desarrolladores.

Todas las modificaciones y mejoras realizadas sobre PySD se han hecho en base a la regla del ***boy scout***: "*leave the campground cleaner than you found it*" [46]. En ella, se hace hincapié en limpiar el código o dejarlo mejor de lo que te has encontrado. En esta regla se destaca que no es necesaria una limpieza muy extensa sobre el código y que, por pequeñas que sean las mejoras, siempre se ayudará a que el código no empeore con el paso del tiempo y los futuros contribuidores se verán beneficiados con esta "limpieza".

7.1. Bugs

7.1.1. Extensión de los modelos Vensim

En el entorno gráfico Vensim están permitidos aquellos modelos en los que el nombre del fichero se encuentre con la extensión en mayúsculas, como ejemplo el nombre del archivo: *teacup.MDL*.

Sin embargo, en PySD solo se había contemplado el caso en el cual los nombres de los modelos Vensim terminan con la extensión en minúscula. Cuando se hizo una prueba con un ejemplo Vensim cuyo nombre estaba en mayúsculas, incluida la extensión, se encontró el fallo de que la traducción se guardaba en el mismo archivo del que se partía, es decir, se sobrescribía el modelo Vensim con el resultado Python. Esto se debía a que PySD no era capaz de encontrar la extensión *.mdl*, ya que estaba en mayúsculas, de tal forma que no

llegaba a intercambiar dicha extensión por *.py* y almacenaba el resultado en la misma ruta en la que se encontraba el modelo Vensim, sobrescribiéndolo.

Se añadieron las líneas de código pertinentes para que PySD permitiese indistintamente modelos con la extensión en mayúsculas o minúsculas. A mayores, se creyó conveniente añadir un ligero control para comprobar la extensión de los archivos que se querían traducir. Se incorporó entonces el lanzamiento de un error cuando el parámetro de la función **translate_vensim** no correspondiese a un archivo Vensim, o lo que es lo mismo, solo se admitían archcivos cuya extensión fuera: *mdl* o *MDL*.

7.1.2. Comentarios de sentecias en Vensim

Las sentencias Vensim suelen constar de 3 partes separadas por virgulillas (símbolo ~), como se muestra en el Fragmento de Código 7.1. La gramática *model_structure_grammar* es la segunda gramática de PySD y la encargada de separar ecuación, unidades y comentario que contenga cada sentencia Vensim.

```
nombre: valor
    ~ unidades
    ~ comentario |
```

Fragmento de código 7.1: Sentencia Vensim

Intentando traducir y ejecutar un determinado ejemplo Vensim, PySD fallaba y no almacenaba correctamente en el diccionario *namespace* todos los nombres de las variables definidas en el modelo. Esto se debía a que en la gramática *model_structure_grammar* se había definido mal los comentarios de las sentencias Vensim. El problema era cuando se tenía una sentencia Vensim con un número de comillas impares en el comentario, ya que solamente estaba definida la posibilidad de comentarios con un número par de comillas.

Se modificó y añadió una regla en esta gramática, indicando que los comentarios podían estar formados por cualquier carácter que no fuera ni una tubería (|) ni una virgulilla(~), ya que estos caracteres eran caracteres especiales de Vensim porque se utilizan como delimitadores en las sentencias para indicar la separación de elementos o el fin de una sentencia. El uso de estos delimitadores se puede comprobar en el fragmento de código anteriormente presentado, 7.1.

7.2. Funciones

7.2.1. *Random 0 1*

Se añadió la implementación para la función RANDOM 0 1 de Vensim, cuyo funcionamiento está explicado en la Sección 3.2.3. Para ello, se definió en el módulo functions la

función *random_0_1*, la cual utilizaba la función *random.uniform* de la librería *numpy* de Python que permitía devolver un valor aleatorio entre 0 y 1.

En la función creada se añadió la documentación de la misma. Además, para que en PySD se pudiera mapear la función *Random 0 1* de Vensim con la creada en Python, *random_0_1*, se añadió una entrada que mapeaba ambos nombres al diccionario de funciones, denominado *functions* en el módulo principal de la traducción, *vensim2py*.

Cabe destacar que la documentación de Vensim califica a esta función como obsoleta. Aún así, se decidió añadirla debido a que se encontró en un modelo Vensim con el que se trabajó al principio. Además, de esta manera se añadía soporte para aquellos modelos Vensim de versiones anteriores y permitía que PySD guardase la compatibilidad hacia atrás. Fue un cambio relativamente sencillo y, por ello, un buen punto de comienzo para contribuir y realizar cambios en el proyecto de PySD.

7.2.2. *Sample If True*

La función *Sample If True* de Vensim se ha explicado en la Sección 3.2.3. Se trata de una función cuyo resultado varía en tiempo de ejecución, ya que evalúa la condición pasada por parámetro en cada *step*. El valor que esta función devuelve, depende del resultado de la condición en cada paso de tiempo del modelo. En el caso de que dicha condición sea cierta, se devolverá y almacenará el valor del segundo parámetro. En caso contrario, se retorna el valor que estaba previamente almacenado.

Debido a que consta de un valor almacenado, el cual depende de cada iteración del modelo, y también cuenta con un valor inicial que permite inicializarlo, dicha función se ha implementado como una clase derivada de la clase *Stateful*. De esta manera se permite que cada función *Sample If True* que aparezca en un modelo Vensim tenga un estado que se corresponda al valor guardado y se actualice siempre que sea necesario.

Para implementar esta funcionalidad, se añadió la clase *SampleIfTrue* al módulo **functions**. Cuenta con tres atributos que corresponden a los parámetros de la función: condición, valor actual y valor inicial. Además, al heredar de la clase *Stateful* cuenta con un atributo *state* que permite simular el comportamiento del valor guardado. Se creó una función *initialize* que inicializa el atributo *state* con el valor inicial, teniendo en cuenta si éste es de tipo *DataArray* o no. Si el valor inicial es de tipo *DataArray*, se almacena en una única variable, en este caso *initial_value*, una matriz con varios valores, por lo que será necesario que el valor almacenado o *state* también sea un DataArray y pueda almacenar varios valores en una variable. La función *__call__* contiene el comportamiento principal, actualiza el estado utilizando para ello la función *if_then_else*. En la Sección 3.2.3 se comentó el parecido entre el comportamiento de la función *If Then Else* y *Sample If True*, llegando a la conclusión de que la diferencia entre ambas se encontraba en que esta última almacenaba el estado y lo devolvía como resultado cuando fuera oportuno. Por ese motivo, se puede utilizar la funcionalidad de *If Then Else*, añadiéndole la característica de almacenar el valor devuelto por *If Then Else* en el caso de que la condición sea cierta.

A la clase *SampleIfTrue* creada también se le añadió un método *ddt* el cual devolvía un

valor constante, 0. Este método se utiliza en la integración por Euler de aquellas variables cuyo valor varíe en cada iteración, correspondiendo a la derivada. En el caso de esta función, al ser 0 se consigue que este valor no influya en el resultado que se almacena en la variable *state*.

Para crear instancias de esta clase en base a las apariciones de la función *Sample If True* en los modelos de Vensim, se creó la función *add_sample_if_true* en el módulo **builder**. Esta función es la encargada de instanciar objetos con estado de la clase *SampleIfTrue*.

7.3. Subscripts

7.3.1. Subscript numeric range

La explicación sobre la funcionalidad de la operación *subscript numeric range* se encuentra en la Sección 3.2.2. Esta operación de atajo para definir subscripts que proporciona Vensim, no tenía su traducción en PySD. Para ello, se ha modificado la gramática *component_structure_grammar* y se ha añadido una regla correspondiente con la definición de un subscript a partir de un rango numérico o un valor único o la combinación de ambos separados por comas. Los nombres de los subscript que forman parte de un rango numérico deben tener la misma secuencia de caracteres y terminar con un número. Es decir, en el ejemplo presentado en el Fragmento de Código 7.2, es necesario que la "cadena" del primer nombre del subscript sea igual a la "cadena" del segundo nombre del subscript. Los números 1 y 4 se corresponden a los límites del rango que están definiendo.

```
1       numericRange :
2           (cadena1 - cadena4)
```

Fragmento de código 7.2: Subscript Numeric Range

Para comprobar que ambas cadenas de caracteres son iguales, fue necesario hacerlo desde el visitor de la regla, a partir de una expresión regular utilizando *regex* de Python. Se añadió funcionalidad asociada al visitor de "numeric_range", que separa la cadena de ambos elementos del número y se realiza un bucle desde el límite inferior (determinado por el primer número) hasta el límite superior (determinado por el segundo número) en el cual se añade en cada iteración una entrada formada por la cadena común y el número de la iteración al diccionario de subscripts.

La definición de un subscript utilizando un rango numérico simplemente es un atajo que Vensim proporciona a los programadores de modelos y no interfiere en la ejecución del modelo, por lo que no se tuvo que hacer ningún cambio relativo a la ejecución.

7.3.2. Subscript copy

La operación de *subscript copy* también se explica en la Sección 3.2.2. Se trata de que dos subscripts tengan los mismos valores pero diferentes nombres.

Se ha modificado la gramática *component_structure_grammar*, añadiendo una regla que utilizase el símbolo "<->" para definir un subscript copia de otro. En la clase correspondiente a esa gramática, *ComponentParser*, se ha creado un atributo nuevo al que se ha llamado *subscript_compatibility*, un diccionario en el que se almacenan aquellos subscript que están copiados o mapeados. En el método visitor de la regla de *subscript_copy*, se añade una entrada al diccionario *subscript_compatibility* con los nombres de los subscripts, indicando que son una copia.

Cada elemento del modelo tiene asociado un diccionario de compatibilidad. En la función *translate_section* de *vensim2py*, después de haber llamado a la función que contiene la gramática *component_structure_grammar*, se unifica en un solo diccionario, llamado *subs_compatibility_dict*, todas las entradas que cada elemento contiene en su diccionario *subscript_compatibility*.

En el Fragmento de código 7.3, se presenta una operación de copia de subscripts. El subscript *'DimB'* no tiene ningún *subscript value* asociado. Por esa razón y porque se encuentra enlazado con el subscript *'DimA'*, se toman los *subscript values* de *'DimA'* (da1, da2, da3) y se duplican para formar parte, a su vez, de *'DimB'*.

```
1       DimA : da1, da2, da3
2       DimB <-> DimA
```

Fragmento de código 7.3: Subscript Copy

7.3.3. Subscript mapping

La implementación de subscript copy y de subscript mapping son bastante similares, debido a que la funcionalidad de subscript mapping que se ha implementado es equivalente a utilizar un subscript copy. En el Fragmento de código 7.3 se ha presentado una sentencia de una copia de los subscripts DimA y DimB. En el Fragmento de código 7.4 se encuentra una sentencia equivalente a la copia de subscripts presentada, pero utilizando sólo el mapeo (símbolo ->) entre los dos subscripts.

```
1       DimA : da1, da2, da3 -> DimB
2       DimB : da1, da2, da3 -> DimA
```

Fragmento de código 7.4: Subscript Mapping

Se modificó la gramática *component_structure_grammar*, añadiendo las reglas necesarias para que se pudiera parsear el mapping de subscripts. En el método visitor asociado a esta

regla se añaden al diccionario *subs_compatibility* aquellos nombres de subscripts que están siendo mapeandos. Dicho diccionario es el mismo que se ha mencionado antes en la explicación de subscript copy y se encarga de almacenar una entrada por cada par de subscripts que estén mapeados, importando la dirección del mapeo. Es decir, si partimos de una sentencia como la de la línea 1 del Fragmento de código 7.4, donde se mapea los valores del subscript DimB a los valores del subscript DimA, en el diccionario *subs_compatibility* se almacenará: 'DimA': ['DimB']. El subscript DimB se almacena en una lista porque puede darse el caso de que haya más de un subscript mapeado con DimA, de esta forma se almacenan todos las entradas necesarias sin sobrescribir ninguna ni perder información.

Al igual que se ha comentado en la explicación de subscript copy, las entradas de *subs_compatibility* de todos los elementos se unifican en un diccionario único en la función *translate_section* del módulo vensim2py. Posteriormente, ese diccionario se pasa como parámetro a la función que contiene la cuarta gramática de PySD, *expression_grammar*, para poder controlar aquellas expresiones o sentencias en las que se utilicen subscript mapeados y no se genere un error en tiempo de ejecución.

7.4. Documentación

Cabe destacar, que tanto las funciones como los errores encontrados y solventados anteriormente se acompañaron de los comentarios y la documentación pertinente.

7.4.1. *Random Uniform*

Al realizar la implementación de la función *Random 0 1*, se localizó en la función *Random Uniform* un *TODO* que indicaba que faltaba la documentación relativa a ésta y se añadió.

En la función *Random uniform* de Vensim existe un tercer parámetro *"s"* o *Stream ID* [72], que corresponde a un flujo de números aleatorios. Esta funcionalidad no estaba implementada en la función equivalente de Python debido a que *random.uniform* de la librería *numpy* no tenía esta característica. Por ese motivo, se añadió un *warning* para cuando ese parámetro *s* tuviera un valor diferente a 0, ya que es en ese momento cuando el resultado de la función *Random uniform* sería diferente. El *warning* que se lanzaba alertaba al usuario que el resultado podía verse ligeramente modificado.

7.4.2. *The 4+1 view model*

Tras realizar el análisis de PySD que ayudó a la alumna a comprender el funcionamiento de todo el proyecto y, sobretodo, qué papel tomaba cada módulo en PySD, se consideró que sería de gran utilidad traducir a inglés este análisis y añadirlo al proyecto.

PySD cuenta con una extensa documentación sobre el proyecto, pero únicamente está enfocada a usuarios finales que deseen traducir y ejecutar modelos. Existe cierta documenta-

ción para desarrolladores en PySD donde se explica brevemente los pasos para contribuir en el proyecto y las tecnologías utilizadas en él. No existe una documentación extensa acerca del funcionamiento interno del proyecto, que sería de gran ayuda para futuros contribuidores.

El proceso de ingeniería inversa realizado sobre PySD se documentó y acompañó de diagramas sobre el sistema y su funcionamiento. Fue una tarea indispensable para poder realizar mejoras y añadir funcionalidades. Por este motivo, se creyó que sería bastante útil y ayudaría a los futuros contribuidores a reducir la curva de aprendizaje sobre el sistema. Se tradujo a inglés y se convirtió el documento a *markdown* para que se pudiera agregar a la documentación de PySD enfocada a desarrolladores.

Se realizó la *pull request* con este documento utilizando *markdown*, pero se pidió convertir el documento a *rst. reStructured Text* es en lenguaje de marcas en el que está escrita toda la documentación de PySD. Estos archivos luego se compilan con *sphinx*, generador de documentación para Python que convierte archivos *reStructured Text* en sitios web HTML.

Después de convertir la documentación al formato pedido, se añadió a la documentación general de PySD a la que se puede acceder a través de: `https://pysd.readthedocs.io/en/master/` y se encuentra en la ruta: *Developer Documentation > The 4+1 Model View of Software Architecture.*

7.5. Modificaciones desde el punto de vista de diseño

En las siguientes figuras se presentan todas las modificaciones realizadas desde el punto de vista de diseño. Se muestran unos diagramas similares a los de la Vista Lógica (Sección 4.2.2) en los que se ha señalizado con colores aquellos métodos o atributos que han sido modificados para poder implementar los cambios anteriormente citados. He de destacar que no se presentan todos los diagramas que se han mostrado en la Vista Lógica, aquellos que no han sido modificados, no se han incluido en esta sección.

En la Figura 7.1 se muestran los cambios realizados en el módulo **builder**, al cual se ha añadido la función *add_sample_if_true*, encargada de instanciar objetos de la clase *SampleIfTrue*.

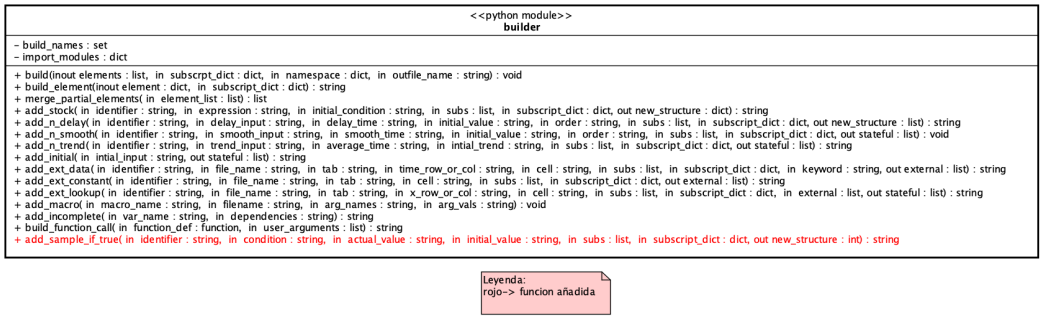

Figura 7.1: Diagrama modificaciones módulo builder

Las modificaciones sobre las clases que contiene el módulo *functions*, se encuentran en la Figura 7.2. En ella se presentan de forma esquemática las clases que se encuadran en el módulo *functions*, con la nueva clase añadida *SampleIfTrue*

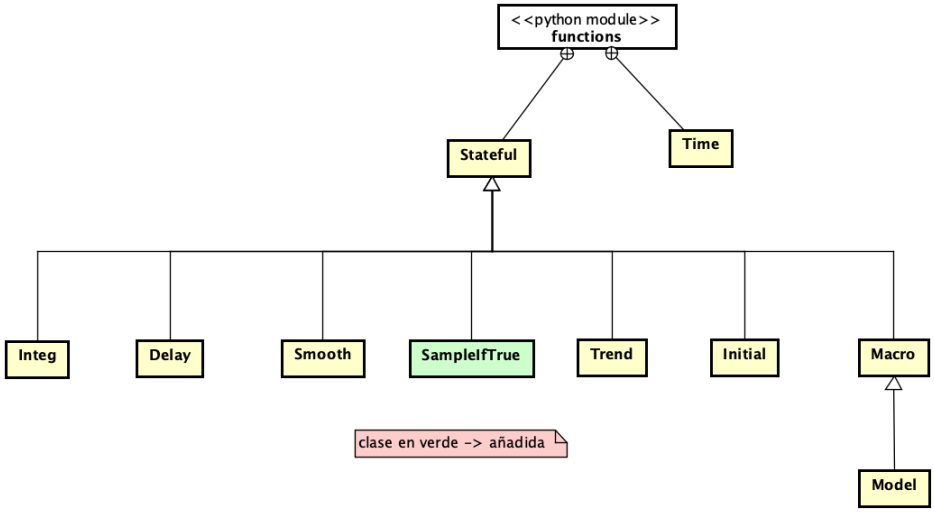

Figura 7.2: Diagrama módulo **functions** con clase SampleIfTrue añadida

En la Figura 7.3 se muestra en detalle la clase *SampleIfTrue* añadida, con los métodos y operaciones.

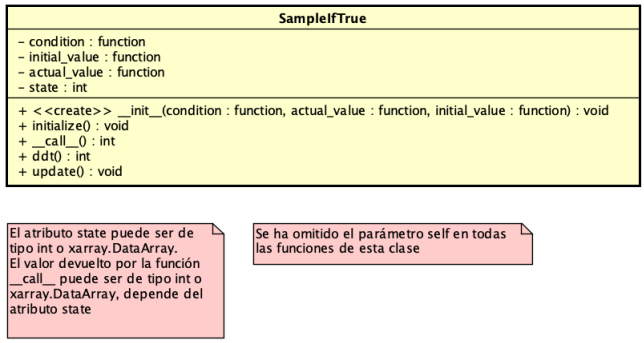

Figura 7.3: Diagrama clase SampleIfTrue añadida

Las modificaciones realizadas sobre las operaciones del módulo **functions** se presentan en la Figura 7.4, cubriendo los cambios relativos a la función *Random 0 1* y *Random Uniform*.

Las funciones que se han visto modificadas pertenecientes al módulo **vensim2py** se presentan en la Figura 7.5 en color azul. En la función *translate_vensim* se ha corregido el error de la extensión de un modelo Vensim. Las funciones *parse_general_expression* y *translate_section*

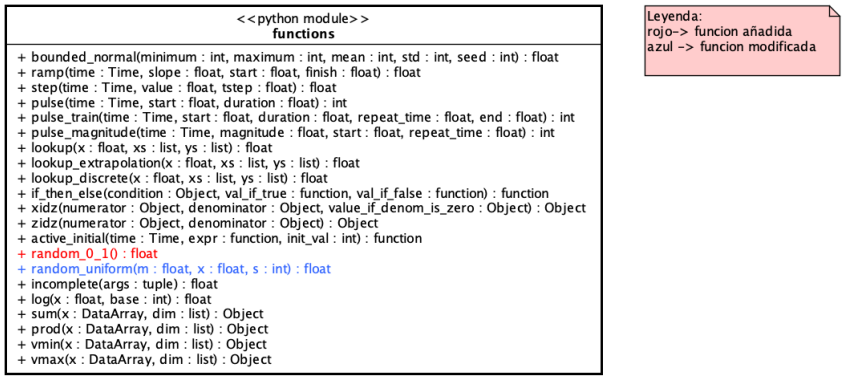

Figura 7.4: Diagrama modificaciones módulo functions

se han modificado incluyendo el diccionario de compatibilidad y la lógica necesaria para poder implementar las operaciones de *subscript copy*, *subscript mapping* y *subscript numeric range*.

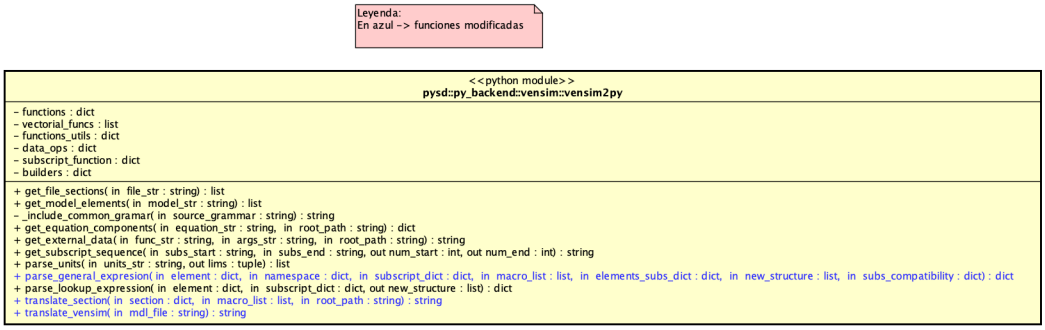

Figura 7.5: Diagrama modificaciones módulo vensim2py

Los cambios realizados sobre las gramáticas se pueden comprobar con los cambios realizados en sus clases asociadas, presentados en la Figura 7.6. Corresponden a la implementación de las operaciones relacionadas con subscripts. Cabe destacar que en la gramática *model_structure_grammar* se añadió una regla para poder solventar el error de las comillas en los comentarios de sentencias Vensim, pero se trata de un cambio que no se puede mostrar con un diagrama.

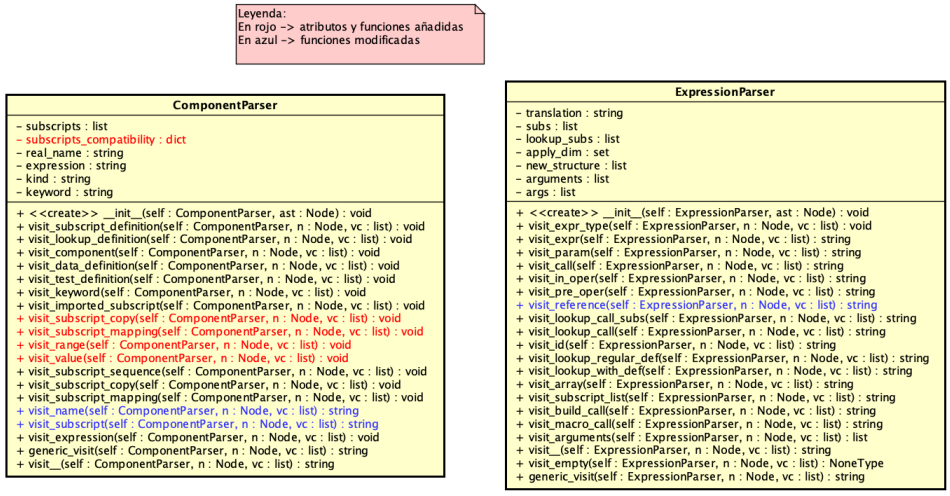

Figura 7.6: Diagrama gramáticas modificadas

Capítulo 8

Implementación y pruebas de las modificaciones

8.1. Contribuir en PySD

Los proyectos *open source* cuentan con licencias abiertas para poder permitir a todos los desarrolladores y usuarios acceder al código fuente del proyecto sin limitaciones.

PySD es un proyecto *open source*, en el cual cada usuario tiene acceso al código del mismo, así como los colaboradores, pero esto no quiere decir que cualquiera pueda agregar modificaciones sin que estas sean revisadas y aprobadas. Para realizar una contribución en PySD se debe seguir una serie de pasos concretos [29], similares a los que se deben realizar en la mayoría de proyectos *open source*.

Para comenzar es necesario crear un ***fork*** de la rama, consiguiendo así tener una copia del código del repositorio central en nuestro repositorio, donde se puede ir añadiendo los cambios y las mejoras del código. Cabe destacar que no es estrictamente necesario que el *fork* se realice desde la rama principal o *master* del proyecto, más bien depende de las conveniencias que tenga cada contribuidor.

Cuando se haya finalizado el mantenimiento y la mejora del proyecto en la rama local del repositorio anteriormente creado, los nuevos cambios podrán ser añadidos al repositorio principal mediante una ***pull request***, en la cual se añade una descripción del trabajo realizado. Se pueden añadir referencias a determinadas *issues* que se resuelvan, referencias a documentación externa que facilite entender las modificaciones realizadas o adjuntar archivos. Una vez creada ***pull request***, todos los *contributors* y colaboradores del repositorio podrán opinar sobre los cambios añadidos o proponer ciertas mejoras. En algunos casos, es entonces cuando se abre un interesante debate en el que se puede mejorar la calidad del código añadido, pueden surgir nuevos casos de prueba que no se habían tenido en cuenta o también se pueden poner de manifiesto ciertas funcionalidades cuyo funcionamiento en Ven-

sim se podría haber malentendido. Cabe destacar que en este punto, cuando una *pull request* está abierta, los *commits* que contiene ésta no se añadirán al repositorio central hasta que el dueño del repositorio o alguna otra persona autorizada para ello, cierre la *pull request*.

Mientras la *pull request* esté abierta, se pueden seguir añadiendo *commits* a ella, los cuales pueden contener recomendaciones hechas por otros contribuidores del proyecto que mejoren los cambios realizados.

Cuando se hayan resuelto las dudas o sugerencias, se cerrará la *pull request* en el caso de que el dueño del proyecto lo crea conveniente. En PySD se tienen en cuenta algunos factores por cada *pull request* que se realiza. Al abrir la *pull request* se comienza automáticamente a compilar la documentación del proyecto y posteriormente, la ejecución de los tests utilizando para ello Travis-CI (Sección 5.2.9). Estos dos requisitos tienen que contar con un resultado satisfactorio para que se puedan añadir los cambios. Al mismo tiempo que se están ejecutando los tests, se analiza la cobertura del código, calculando un porcentaje de cobertura de todo el código del proyecto teniendo en cuenta los cambios realizados. En el caso de que el porcentaje de cobertura disminuya un 1 % o más en comparación con el porcentaje previo a la *pull request*, seguramente, sea necesario añadir tests que prueben las líneas de código agregadas para aumentar dicho porcentaje y que la *pull request* pueda ser mergeada al repositorio.

En PySD, para poder añadir nuevos tests se deben añadir a los archivos que se encuentran en la carpeta "pysd/tests/". Dichos tests pueden ser de integración o unitarios y, en ocasiones, quizás sea necesario que hagan referencia a modelos Vensim para probar el correcto funcionamiento de traducción y ejecución de PySD. Estos modelos referenciados se deberán adjuntar al súbmodulo que contiene los modelos Vensim utilizados para los test, llamado **test-models** [62].

Para poder añadir nuevos modelos Vensim al repositorio **test-models** es necesario realizar de nuevo el proceso para contribuir en un proyecto *open source*, comenzando por realizar un ***fork*** del proyecto para tener una copia del repositorio y desde ahí comenzar a añadir los nuevos modelos Vensim. Se deberá adjuntar el modelo Vensim (extensión *mdl*) y los resultados que se obtengan con Vensim en un archivo que se denomine "output", para poder automatizar la comprobación de éstos contra los resultados que genera PySD. A mayores, se deberá adjuntar una captura de pantalla del modelo en el entorno gráfico Vensim y un *README* que explique brevemente lo que se intenta comprobar con ese modelo, en el que se haga referencia a la captura de pantalla y una tabla con datos sobre el contribuidor que ha añadido ese modelo.

Cuando se hayan realizado todos los pasos anteriormente descritos, se puede abrir una *pull request* en el repositorio **test-models**. Si ésta es aceptada se podrá añadir entonces al repositorio PySD aquellos tests en los que se referencien los nuevos modelos Vensim añadidos.

Una vez que se haya cerrado la *pull request*, se puede continuar realizando más cambios en el proyecto de nuestro repositorio y creando *pull requests* cada vez que lo creamos conveniente en base a los cambios que vayamos realizando. De la misma forma, será necesario actualizar nuestro repositorio en base a los cambios que se vayan haciendo en el repositorio central para que no haya conflictos futuros a la hora de realizar futuras modificaciones y/o mejoras.

8.2. Tests en PySD

Test Driven Development (TDD) [18] es una práctica iterativa de diseño de software orientado a objetos, presentada por *Kent Beck* y *Ward Cunningham* como parte de la metodología *Extreme Programming*. Principalmente esta técnica se centra en la realización de las pruebas antes de escribir el código a probar. Al realizar los tests primero, se fuerza a pensar acerca de cómo se desea que el código se comporte e interaccione con sus colaboradores. Una vez que el test se ha realizado, se escribe el código asociado.

En PySD se destaca el uso de **TDD**, debido a que se conoce de antemano el comportamiento que se espera del sistema, ya que se cuenta previamente con los ejemplos de Vensim y los resultados que éstos generan al ser ejecutados. Por esa razón, se parte sabiendo qué resultados se quiere que genere el sistema PySD, que son los mismos que se han obtenido al ejecutar los ejemplos con Vensim, dando cabida al desarrollo de software bajo **TDD**.

En el repositorio de PySD se encuentra un directorio dedicado a los tests, en él se encuentran los tests unitarios y los tests de integración.

8.2.1. Test unitarios

Los **test unitarios** de PySD son aquellos que se encargan de probar una funcionalidad concreta del código. **TDD** implica desarrollar las pruebas unitarias a las que se va a someter el software antes de escribirlo, es decir, crear primero aquellos tests que prueben aisladamente la funcionalidad de cada módulo que compone el sistema.

En el repositorio de PySD, los tests unitarios se encuentran en el directorio de **"tests"**. Este directorio cuenta con tantos archivos de tests unitarios como módulos hay en PySD.

En Python, los tests unitarios se pueden realizar utilizando la librería *unittest* (Sección 5.2.7). En PySD los ficheros cuyo nombre comienza por "unit_test" seguido de el nombre de un módulo, son aquellos que contienen los tests unitarios que prueban ese módulo. Para poder ejecutar estos tests se debe utilizar el comando representado en el Fragmento de Código 8.1. Tras ejecutar el comando, se prueba uno de los módulos que componen PySD. Posteriormente, cuando el test haya finalizado, se informará por pantalla si el resultado esperado es igual al obtenido o si ha ocurrido algún error y dónde se ha producido este.

```
1 python -m unittest <nombre-test-unitario.py>
```

Fragmento de código 8.1: Ejecutar test unitario

8.2.2. Test de integración

Una vez que se han realizado los tests unitarios necesarios, se deben realizar las **pruebas de integración**. Estas pruebas se encargan de probar en conjunto el correcto funcionamiento

de los elementos unitarios que componen el sistema y de que la interacción entre ellos funcione adecuadamente.

Los test de integración de PySD se encuentran, al igual que los tests unitarios, en el directorio **"test"** de la carpeta *"pysd"*. Estos tests se identifican fácilmente ya que su nombre comienza por "integration_test", seguido del nombre del subsistema que prueba, ya bien sea Vensim o XMILE. Más concretamente, los test de integración de Vensim se encuentran en el archivo llamado *integration_test_vensim_pathway.py*, el cual contiene aquellos tests que se encargan de probar la traducción y ejecución de modelos Vensim.

Los modelos Vensim que se referencian en los tests de integración de PySD, se encuentran en un submódulo Git, llamado **test-models**, explicado en detalle en la subsección 8.2.3.

En el archivo *integration_test_vensim_pathwat.py* se hacen referencias a los modelos Vensim de test-models, los cuales se traducen, se ejecutan y se compara el resultado obtenido con el esperado. De esta forma, se permite comprobar si la interacción entre todos los módulos que componen el proyecto y el funcionamiento en conjunto generan el resultado esperado. Con el comando del Fragmento de Código 8.2 se ejecutan los tests de integración de PySD, utilizando para ello la librería **nosetests** (Sección 5.2.8).

```
python -m nose <nombre-test-integracion.py>
```

Fragmento de código 8.2: Ejecutar test de integración

Además, en PySD se ha añadido un *Makefile* que permite ejecutar todos los tests del proyecto automáticamente y generar un informe html sobre la cobertura del código.

8.2.3. Submódulo SDXorg/test-models

Los submódulos [24] permiten mantener un repositorio de Git como subdirectorio de otro repositorio. De esta manera se consigue clonar otro repositorio en un proyecto manteniendo los *commits* separados pero siempre actualizados. En PySD, este submódulo se encuentra en el directorio de *tests*, se llama *test-models* y está redirigido al proyecto **test-models** [62].

Test-models contiene modelos Vensim con la salida que se espera de estos archivos. Está compuesto por dos directorios, **test** y **samples**. El directorio de **test** contiene modelos funcionales en Vensim que cuentan con mínima funcionalidad, para poder probar funciones concretas. En el directorio **samples** encontramos modelos Vensim completos.

Además, cada ejemplo de modelo que forma este proyecto cuenta con una imagen que representa el modelo en el entorno gráfico de Vensim, un archivo con extensión *.csv* o *.tab* que recoge los resultados que genera el modelo y, en algunos casos, cuenta con un modelo en Xmile idéntico al modelo presentado en Vensim.

Aunque PySD utilice el proyecto **test-models** para probar la funcionalidad implementada, este proyecto se puede utilizar para cualquier otro que requiera modelos Vensim.

8.3. Organización de la implementación de las aportaciones

Es posible acceder a todo el código relativo a los cambios comentados en la Sección 7. En los siguientes enlaces se pueden encontrar las diferentes *pull request* realizadas al repositorio central de PySD. Asimismo, se muestran las *issues* creadas y solventadas en cada *pull request*, la *pull request* realizada al repositorio de tests Vensim y se especifican los test creados o descomentados de los archivos de tests de pysd que permitían comprobar los cambios realizados.

Primera pull request

- Enlace: `https://github.com/JamesPHoughton/pysd/pull/247`
- Cambios realizados:
  - Extensión de los modelos Vensim
  - Función Random 0 1
  - Documentación Random uniform
  - Comentarios de sentencias en Vensim
- Issues abiertas para realizar esta *pull request*:
  - Issue 244, extensión de los modelos Vensim.
    Enlace: `https://github.com/JamesPHoughton/pysd/issues/244`
  - Issue 245, soporte para la función Random 0 1.
    Enlace: `https://github.com/JamesPHoughton/pysd/issues/245`
  - Issue 246, fallo en comentarios de sentencias Vensim.
    Enlace: `https://github.com/JamesPHoughton/pysd/issues/246`
- Issues solventadas:
  - Issue 244, extensión de los modelos Vensim.
  - Issue 245, soporte para la función Random 0 1.
  - Issue 246, fallo en comentarios de sentencias Vensim.
- *Pull request* asociada en el repositorio **test-models**: `https://github.com/SDXorg/test-models/pull/69`

En esta pull request se añaden a los tests de integración dos tests: *test_run_uppercase* y *test_odd_number_quotes*, que testean el correcto funcionamiento de pysd cuando se introduce un modelo Vensim cuya extensión está en mayúsculas y un modelo Vensim que en un comentario de una sentencia Vensim tiene un número impar de comillas, respectivamente. Para estos tests se utilizan los modelos Vensim *teacup-upper.MDL* y *teacup_3quotes.mdl*, que se

pueden encontrar en la pull request hecha en el repositorio de los tests. También, se añade un test que no es un archivo Vensim al repositorio pysd, llamado *Not-Vensim.txt*. Este archivo se utiliza en un test añadido a los test unitarios del módulo pysd, para comprobar que el programa lanza una excepción cuando se introduce un fichero que no es un modelo Vensim.

Segunda pull request

- Enlace: `https://github.com/JamesPHoughton/pysd/pull/252`
- Cambios realizados:
  - Función Sample If True
  - Soporte Subscript numeric range
- Issues abiertas para realizar esta *pull request*:
  - Issue 251, soporte para rango numérico de subscripts.
    Enlace: `https://github.com/JamesPHoughton/pysd/issues/251`
- Issues solventadas:
  - Issue 217, soporte para la función Sample If True.
    Enlace: `https://github.com/JamesPHoughton/pysd/issues/217`
  - Issue 251, soporte para rango numérico de subscripts.
- *Pull request* asociada en el repositorio **test-models**: `https://github.com/SDXorg/test-models/pull/70`

En esta segunda pull request se descomenta el test que comprueba el funcionamiento de *Sample If True* del archivo que contiene los tests de integración de Vensim. Se añade a los tests de integración el test: *test_subscript_numeric_range*, que testea el correcto funcionamiento de pysd cuando un modelo Vensim contiene una definición de un subscript a partir de un rango numérico. Para este test se utiliza el modelo Vensim llamado *test_subscript_numeric_range.mdl* que se ha añadido a la pull request del repositorio de test_models.

Tercera pull request

- Enlace: `https://github.com/JamesPHoughton/pysd/pull/260`
- Cambios realizados:
  - Soporte subscript copy
  - Soporte subscript mapping
  - Documentación para desarrolladores
- *Pull request* asociada en el repositorio **test-models**: `https://github.com/SDXorg/test-models/pull/73`

- La documentación para desarrolladores añadida a la documentación principal de PySD se encuentra en el siguiente enlace:

  `https://pysd.readthedocs.io/en/master/development/pysd_architecture_views/4+1view_model.html`

  Esta documentación parte del modelo de vistas 4+1 sobre PySD desarrollado en la Sección 4.2. Este modelo tuvo que traducirse al inglés y convertirlo al formato *rst* (explicado en la Sección 5.2.11), ya que este formato es el que se utiliza para generar la documentación asociada al proyecto PySD.

  Al realizar esta *pull request*, el colaborador Eneko Martin y el dueño del proyecto, James Houghton, me dieron su enhorabuena por el trabajo realizado sobre la documentación añadida. Se muestra en las Figuras 8.1 y 8.2 las felicitaciones que ambos dejaron en los comentarios de esta *pull request*. A ellos se puede acceder a través del enlace de la *pull request*.

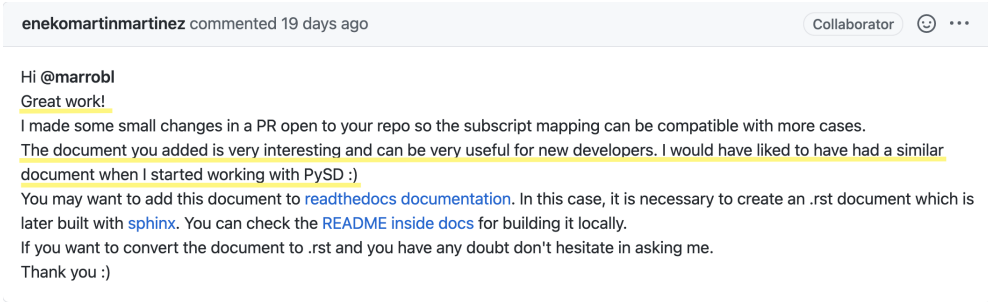

Figura 8.1: Respuesta Eneko Martin

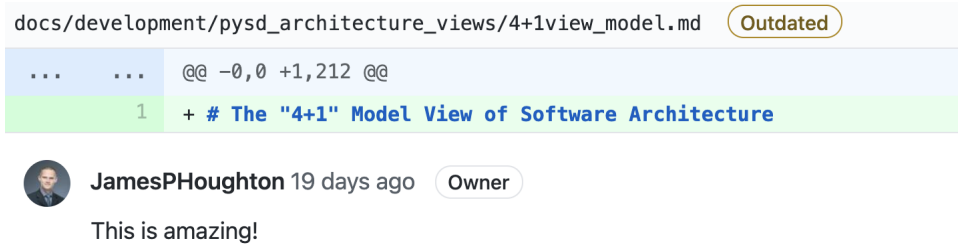

Figura 8.2: Respuesta James Houghton

Para comprobar el correcto funcionamiento de las modificaciones añadidas en esta tercera pull request se añade a los tests de integración el test *test_subscript_copy* que testea el correcto funcionamiento de pysd cuando en un modelo Vensim se utiliza la operación de copia de subscripts, y el test *test_subscript_mapping_simple* que comprueba el correcto funcionamiento de la lógica desarrollada para la operación de subscript mapping. Para estos tests se utilizan los modelos Vensim llamados *test_subscript_copy.mdl*, que contiene una copia de subscripts, y *test_subscript_mapping_simple.mdl*, que contiene un mapping de subscripts. Se pueden encontrar en la pull request asociada del repositorio de test_models.

Además, se han actualizado los tests unitarios del módulo vensim2py debido a la creación del nuevo diccionario *subs_compatibility*.

Al finalizar la 3ª pull request, la cobertura del proyecto PySD obtenida con los modelos y tests añadidos fue de 94.902 %.

8.4. Comprobación del objetivo fundamental

Tras todos los cambios realizados y añadidos al repositorio central de PySD a través de las 3 *pull requests* anteriormente comentadas, se ha comprobado la traducción del modelo WILIAM. La traducción de este modelo, como se ha comentado en la Sección 1.2, es el objetivo fundamental de este Trabajo Fin de Grado. Así que, una vez finalizadas todos las contribuciones que se han realizado al repositorio, se ha probado a traducir el modelo WILIAM con la versión actual de PySD.

Al intentar traducir WILIAM se han encontrado varios errores de parseo debido a que aún falta la implementación de algunas funciones de Vensim en el proyecto PySD. Para poder probar el correcto funcionamiento de los cambios implementados, se han extraído las funciones que daban error del modelo, intercambiándolas por constantes para que no generasen ningún fallo. Además, se han ido anotando todas las funciones que no tienen soporte en PySD y se ha abierto una *issue* en el repositorio por cada función que falta. Éstas se han listado a continuación, en la Sección 9.1, donde se describen las posibles líneas de trabajo futuro.

Después de extraer las funciones que generaban error de parsing del modelo WILIAM, éste se ha podido parsear perfectamente y se ha generado la traducción del mismo sin ningún problema.

Capítulo 9

Conclusiones

Tras completar este Trabajo de Fin de Grado se ha obtenido un resultado muy satisfactorio y realmente enriquecedor. Se han cumplido los objetivos presentados inicialmente. Se ha logrado aportar funcionalidad nueva al repositorio de pysd, dotando al proyecto de características que Vensim posee y que PySD no tenía. Además, se ha llegado a traducir el modelo WILIAM, utilizando para ello las nuevas funcionalidades desarrolladas y con la salvedad de las funciones enumeradas en la Sección 9.1.

Se ha conseguido realizar un estudio exhaustivo de PySD y, a partir de él, se ha logrado evolucionar y mejorar la funcionalidad y documentación con la que contaba previamente el proyecto de PySD.

Este trabajo ha supuesto un gran aprendizaje para la alumna, pues ha significado:

- Conocer los sistemas dinámicos que existen, cómo funcionan y cómo pueden ser utilizados con fines realmente provechosos. En concreto, el lenguaje Vensim y algunas de sus funciones en las que se ha incidido en este proyecto.
- Mejorar los conocimientos sobre Python, obteniendo más soltura y destreza con el lenguaje, así como con algunas de sus librerías.
- Conocer la librería *parsimonious* en profundidad y su funcionamiento, junto al uso del patrón *Visitor*.
- Comprobar la gran utilidad de la documentación en proyectos software y la gran ayuda que ésta puede aportar para futuros desarrolladores.
- Comprender en profundidad la librería PySD y su funcionamiento.
- Trabajar y aportar a un proyecto *open source*, lo que conlleva aprender cómo hacerlo y bajo qué requisitos. Cabe destacar que ha sido de gran utilidad cuando colaboradores del proyecto daban su opinión o sugerían ciertas mejoras en las pull request, aprendiendo de algunos conocimientos de los colaboradores. De la misma manera, era

muy gratificante cuando resaltaban los cambios realizados y los añadían al repositorio principal.

Finalmente, se ha logrado colaborar en un proyecto *open source*, junto a otros *contributors*. Se ha conseguido aportar nueva funcionalidad al proyecto y mejorar la gramática del mismo. Un trabajo fundamental ha sido la ingeniería inversa realizada sobre PySD, que ha permitido obtener un buen nivel de conocimientos sobre el proyecto para poder realizar modificaciones y mejoras en él. Este proceso de ingeniería inversa ha sido traducido y añadido al repositorio principal, recibiendo felicitaciones por su aportación y el trabajo realizado.

9.1. Líneas de trabajo futuras

Si bien los objetivos de este trabajo están completos, pero se pueden hacer mejoras para continuar el proyecto de PySD y el de traducción y ejecución de WILIAM. Sería un buen punto de partida mejorar o implementar algunas de las funcionalidades de Vensim en PySD con las que cuenta WILIAM, para poder llevar a cabo una perfecta traducción y ejecución automatizada del citado modelo. Las funciones Vensim o palabras reservadas que se han encontrado en WILIAM y para las que, a día de hoy, PySD no tiene soporte, son:

- La función *GET TIME VALUE* de Vensim.
- La función *INVERT MATRIX* de Vensim.
- La función *ALLOCATE BY PRIORITY* de Vensim.
- La función *VECTOR SELECT* de Vensim.
- La función *FORECAST* de Vensim.
- La función *SHIFT IF TRUE* de Vensim.
- El uso de la palabra reservada *:NA:* en Vensim.

Se ha abierto una lista de issues para indicar la necesidad sobre la implementación de estas funciones en el *Issue Tracker* del proyecto PySD. Algunas de esas funciones no se encuentran en la lista de issues del proyecto, debido a que están indicadas como no implementadas en el código del mismo junto a un *TODO*.

Además, puede ser realemente interesante estudiar una posible mejora sobre la eficiencia del proyecto PySD.

Apéndice A

Manuales

A.1. Manual de despliegue e instalación

En esta sección se detallan los pasos necesarios para instalar PySD.

Se puede descargar el código fuente clonando el repositorio de GitHub, Fragmento de código A.1.

```
1 git clone https://github.com/marrobl/pysd.git
```

Fragmento de código A.1: Descargar PySD

Utilizando el instalador de paquetes **pip** se podrá descargar la versión más reciente del paquete PySD, como se muestra en el Fragmento de código A.2.

```
1 pip pysd
```

Fragmento de código A.2: Descargar PySD

Una vez descargado el paquete PySD, será necesario actualizar e instalar las dependencias. Para ello, dentro del directorio **pysd** se ejecutará el comando de la Fragmento de código A.3.

```
1 cd pysd
2 python setup.py install
```

Fragmento de código A.3: Instalar dependencias y PySD

Si se ha realizado la descarga del paquete PySD con pip, las dependencias se deberían haber instalado automáticamente, por lo que no sería necesario utilizar el último comando presentado.

A.2. Manual de usuario

Para realizar la traducción y ejecución de un modelo Vensim será necesario crear un archivo Python que utilice la librería pysd y sus funciones.

El módulo pysd cuenta con la función **read_vensim** que se encarga de leer un modelo Vensim y traducirlo a Python.

A partir del modelo traducido, se puede utilizar la función **run** que permitirá ejecutarlo y obtener los resultados.

Utilizando estas dos funciones se puede crear un archivo como el que se muestra en el Fragmento de código A.4 para poder traducir y obtener los resultados de un modelo Vensim.

```
import pysd

model = pysd.read_vensim("Ruta al modelo Vensim")

results = model.run()

print(results)
```

Fragmento de código A.4: Traducir y ejecutar modelo Vensim

En el caso de que se parta de un modelo Python que ha sido traducido anteriormente, se puede utilizar la función **load** del módulo pysd. Esta función permite cargar un modelo Python con el propósito de ejecutarlo posteriormente y así, evitar repetir el proceso de traducción del modelo Vensim.

```
import pysd

model = pysd.load("Ruta al modelo Python")

results = model.run()

print(results)
```

Fragmento de código A.5: Cargar y ejecutar modelo Python

Apéndice B

Resumen de enlaces adicionales

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

- Repositorio del proyecto en el gitlab de la Escuela: `https://gitlab.inf.uva.es/marrobl/tfg_vensimpython`.
- Repositorio GitHub del fork de PySD: `https://github.com/marrobl/pysd`.
- Repositorio GitHub del fork de test-models: `https://github.com/marrobl/test-models`.
- Pull requests realizadas y mergeadas a master en el repositorio principal de PySD:
  - `https://github.com/JamesPHoughton/pysd/pull/247`
  - `https://github.com/JamesPHoughton/pysd/pull/252`
  - `https://github.com/JamesPHoughton/pysd/pull/260`
- Pull requests realizadas y mergeadas a master en el repositorio de test-models:
  - `https://github.com/SDXorg/test-models/pull/69`
  - `https://github.com/SDXorg/test-models/pull/70`
  - `https://github.com/SDXorg/test-models/pull/73`
- Issues abiertas en el repositorio principal de PySD:
  - `https://github.com/JamesPHoughton/pysd/issues/244`
  - `https://github.com/JamesPHoughton/pysd/issues/245`
  - `https://github.com/JamesPHoughton/pysd/issues/246`
  - `https://github.com/JamesPHoughton/pysd/issues/251`
  - `https://github.com/JamesPHoughton/pysd/issues/250`

- https://github.com/JamesPHoughton/pysd/issues/261
- https://github.com/JamesPHoughton/pysd/issues/263
- https://github.com/JamesPHoughton/pysd/issues/264
- https://github.com/JamesPHoughton/pysd/issues/265
- https://github.com/JamesPHoughton/pysd/issues/266

Bibliografía

[1] Engineering 360. Reverse engineering. https://insights.globalspec.com/article/7367/how-does-reverse-engineering-work. Accessed: 2020-10-14.

[2] Astah. Academic university discount licenses for astah software. https://astah.net/pricing/academic/, Feb 2020. Accessed: 2021-02-04.

[3] Astah. Premier diagramming, modeling software & tools. https://astah.net/, Nov 2020. Accessed: 2021-02-03.

[4] Atlassian. Qué es el control de versiones: Atlassian git tutorial. https://www.atlassian.com/es/git/tutorials/what-is-version-control. Accessed: 2020-10-11.

[5] Oscar Blancarte. Patrón visitor. https://reactiveprogramming.io/blog/es/patrones-de-diseno/visitor. Accessed: 2021-02-04.

[6] Georg Brandl and the Sphinx team. Sphinx. https://www.sphinx-doc.org/en/master/. Accessed: 2021-06-03.

[7] Antonio García Candil. Git - como gestionar y cuidar nuestro código. https://enmilocalfunciona.io/git-como-gestionar-y-cuidar-nuestro-codigo/, Jun 2018. Accessed: 2020-10-10.

[8] Overleaf community. Overleaf. https://www.overleaf.com/about, Dec 2020. Accessed: 2020-09-14.

[9] Python community. parsimonious. https://pypi.org/project/parsimonious/. Accessed: 2021-02-02.

[10] Philippe Kruchten Rational Software Corp. Architectural blueprints—the "4+1" view model of software architecture. https://www.cs.ubc.ca/~gregor/teaching/papers/4+1view-architecture.pdf. Accessed: 2021-02-08.

[11] Ministerio de empleo y seguridad social. Disposición 3156 del boe núm. 57 de 2018. https://www.boe.es/boe/dias/2018/03/06/pdfs/BOE-A-2018-3156.pdf. Accessed: 2021-01-29.

[12] Universidad de Valladolid. Jitsi meet. https://videoconferencia.inf.uva.es/. Accessed: 2020-09-14.

[13] GitLab Docs. Gitlab issues. https://docs.gitlab.com/ee/user/project/issues/. Accessed: 2020-09-14.

[14] GitLab Docs. Issue boards. https://docs.gitlab.com/ee/user/project/issue_board.html. Accessed: 2020-09-14.

[15] Sebastian Dormido and Fernando Morilla. http://www.dia.uned.es/~fmorilla/Web_FMorilla_Julio_2013/MaterialDidactico/Vensim.pdf, Mar 2005. Accessed: 2020-10-20.

[16] Ealde. En qué consiste el product backlog y el sprint backlog en scrum. https://www.ealde.es/product-backlog-sprint-backlog/, Sep 2020. Accessed: 2021-01-26.

[17] Mariana Experta en desarrollo de carrera eClass. ¿cuáles son los eventos de scrum? conócelos aquí. https://blog.eclass.com/cuales-son-los-eventos-de-scrum-conocelos-aqui-0. Accessed: 2021-01-26.

[18] Moisés Carlos Fontela. Estado del arte y tendencias en test-driven development. http://sedici.unlp.edu.ar/bitstream/handle/10915/4216/Documento_completo.pdf?sequence=1. Accessed: 2021-02-16.

[19] EASME Executive Agency for SMEs. Horizon 2020 environment and resources data hub. https://sc5.easme-web.eu/?p=821105. Accessed: 2021-06-05.

[20] Python Software Foundation. Pep 8 – style guide for python code. https://www.python.org/dev/peps/pep-0008/#function-and-method-arguments. Accessed: 2021-04-20.

[21] Python Software Foundation. unittest - unit testing framework¶. https://docs.python.org/3/library/unittest.html. Accessed: 2021-05-25.

[22] Erich Gamma, Richard Helm, Ralph Johnson, and Jonh Vlissides. Design patterns : elements of reusable object-oriented software : Gamma, erich : Free download, borrow, and streaming. https://archive.org/details/designpatternsel00gamm/page/174/mode/2up, Jan 1995. Accessed: 2021-04-04.

[23] GanaEnergía. ¿cuánto consumo el ordenador? sal de dudas. https://ganaenergia.com/blog/que-consumo-nos-supone-utilizar-el-ordenador/, Dec 2020. Accessed: 2021-02-11.

[24] Git. 7.11 herramientas de git - submódulos. https://git-scm.com/book/es/v2/Herramientas-de-Git-Subm\unhbox\voidb@x\bgroup\let\unhbox\voidb@x\setbox\@tempboxa\hbox{o\global\mathchardef\accent@spacefactor\spacefactor}\accent19o\egroup\spacefactor\accent@spacefactor\futurelet\@let@token\protect\penalty\@M\hskip\z@skipdulos. Accessed: 2021-02-16.

[25] GitHub. Where the world builds software. https://github.com/. Accessed: 2020-10-11.

[26] IONOS Digital Guide. La ingeniería inversa de software. https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/ingenieria-inversa-de-software/. Accessed: 2020-10-14.

[27] LOCOMOTION H2020. Locomotion. https://www.locomotion-h2020.eu/. Accessed: 2021-06-05.

[28] James Houghton. Basic usage¶. https://pysd.readthedocs.io/en/master/basic_usage.html. Accessed: 2021-05-29.

[29] James Houghton. Contributing to pysd¶. https://pysd.readthedocs.io/en/master/development/contributing.html. Accessed: 2021-05-26.

[30] Joel Francia Huambachano. ¿qué es scrum? https://www.scrum.org/resources/blog/que-es-scrum. Accessed: 2021-01-26.

[31] Bob Hughes and Mike Cotterell. Software project management, 2009. Fifth edition, ed. Mc Graw Hill Education.

[32] Ventana Systems Inc. Mapping of subscript ranges. https://www.vensim.com/documentation/ref_subscript_mapping.htm. Accessed: 2020-12-4.

[33] Ventana Systems Inc. Mapping subscripts. https://www.vensim.com/documentation/ref_subscript_mapping.htm. Accessed: 2021-01-28.

[34] Ventana Systems Inc. Subscripts. https://www.vensim.com/documentation/ref_subscripts.htm. Accessed: 2021-01-27.

[35] Ventana Systems Inc. Variable types. https://www.vensim.com/documentation/ref_variable_types.htm. Accessed: 2021-01-28.

[36] Ventana Systems Inc. Vector functions. https://www.vensim.com/documentation/21230.htm. Accessed: 2020-11-10.

[37] Ventana Systems Inc. Vensim. https://vensim.com/purchase/. Accessed: 2021-02-12.

[38] Ventana Systems Inc. Vensim help. https://www.vensim.com/documentation/macros.html. Accessed: 2021-02-04.

[39] Ventana Systems Inc. Ventana software: Vensim; ventity. https://www.ventanasystems.com/software/, Mar 2019. Accessed: 2020-10-20.

[40] JamesPHoughton. Jamesphoughton/pysd. https://github.com/JamesPHoughton/pysd. Accessed: 2020-10-21.

[41] JamesPHoughton. not implemented function sample if true · issue #217 · jamesphoughton/pysd. https://github.com/JamesPHoughton/pysd/issues/217. Accessed: 2021-02-09.

[42] Lainformacion.com. ¿cuál es el coste real de tener un trabajador para una empresa? https://www.lainformacion.com/practicopedia/cual-es-el-coste-real-de-un-trabajador-para-una-empresa/6491464/, Feb 2019. Accessed: 2021-01-30.

[43] The Daring Fireball Company LLC. Markdown. https://daringfireball.net/projects/markdown/. Accessed: 2021-06-03.

[44] José I. Santos Luis R. Izquierdo, José M. Galán and Ricardo del Olmo. Modelado de sistemas complejos mediante simulación basada en agentes y mediante dinámica de sistemas. http://www.luis.izqui.org/papers/Izquierdo_Galan_Santos_Olmo_2008.pdf. Accessed: 2020-10-20.

[45] Estefanía Mac. Fórmulas de contabilidad básica: ¿cómo calcular la depreciación de una computadora? https://www.cuidatudinero.com/13074026/como-calcular-la-amortizacion-de-una-computadora-portatil, Sep 2019. Accessed: 2021-02-03.

[46] Robert C. Matin. Clean code, 2009. ed. Pearson Education, Inc.

[47] MEDEAS. Modeling the renewable energy transition in europe. https://www.medeas.eu/. Accessed: 2020-09-20.

[48] MEDEAS. Mooc course. https://www.medeas.eu/model/mooc-course. Accessed: 2020-09-20.

[49] Microsoft. Visual studio code - code editing. redefined. https://code.visualstudio.com/, Apr 2016. Accessed: 2020-09-28.

[50] Javier Olmo. ¿qué es la ingeniería inversa y cómo funciona? - blog de tecnicoo. https://acentocoop.es/blog/ingenieria-inversa/, Sep 2020. Accessed: 2020-10-14.

[51] Opensource.org. Open source news. https://opensource.org/. Accessed: 2020-10-10.

[52] Visual Paradigm. The #1 development tool suite. https://www.visual-paradigm.com/. Accessed: 2021-02-03.

[53] Visual Paradigm. Visual paradigm pricing. https://www.visual-paradigm.com/shop/vp.jsp. Accessed: 2021-02-04.

[54] Thomas Peham. Gitlab vs github: What are the key differences? the ultimate guide. https://usersnap.com/blog/gitlab-github/, Sep 2020. Accessed: 2020-10-11.

[55] Jason Pellerin. Testing with nose¶. https://nose.readthedocs.io/en/latest/testing.html. Accessed: 2021-05-26.

[56] Refactoring.Guru. Decorator. https://refactoring.guru/es/design-patterns/decorator. Accessed: 2021-04-17.

[57] reStructuredText Docutils. https://docutils.sourceforge.io/rst.html, May 2016. Accessed: 2021-05-27.

[58] EOS Costa Rica. ¿por qué involucrarse en proyectos open source? https://medium.com/@eoscostarica/https-medium-com-eoscostarica-por-que-involucrarse-en-proyectos-open-source-80533e34408c, Jan 2020. Accessed: 2020-10-10.

[59] rocket.chat. The ultimate communication platform. https://rocket.chat/es/. Accessed: 2020-09-14.

[60] Diego Rodrigo Verdugo. Desarrollo de nuevas funcionalidades del software pysd para la traducción del modelo medeas de lenguaje vensim a python. http://uvadoc.uva.es/handle/10324/44423, Jan 1970. Accessed: 2020-09-20.

[61] Ken Schwaber and Jeff Sutherland. La guía de scrum. https://scrumguides.org/docs/scrumguide/v2016/2016-Scrum-Guide-Spanish.pdf. Accessed: 2021-06-12.

[62] SDXorg. Sdxorg/test-models. https://github.com/SDXorg/test-models/tree/8fe779fb9b84eb8426ddf3e4e7e1e796d180d636. Accessed: 2021-02-09.

[63] Selectra. Precio del kwh de luz por horas. https://tarifaluzhora.es/. Accessed: 2021-02-11.

[64] Fabio Mascarenhas Sergio Medeiros and Roberto Ierusalimschy. The packrat parsing and parsing expression grammars page. https://bford.info/packrat/. Accessed: 2021-02-02.

[65] Ventana Systems. Random number functions. https://www.vensim.com/documentation/fn_random.htm. Accessed: 2020-10-24.

[66] GMBH TRAVIS CI. Travis ci. https://travis-ci.org/. Accessed: 2021-06-03.

[67] Informática UV. Mantenimiento. http://informatica.uv.es/iiguia/2000/IPI/material/tema7.pdf. Accessed: 2020-10-17.

[68] UVAgeeds. ¿qué es la dinámica de sistemas? https://geeds.es/que-es-la-dinamica-de-sistemas/, Mar 2020. Accessed: 2020-10-20.

[69] Ventana Systems Inc. Vensim. Vensim. https://vensim.com/. Accessed: 2020-10-24.

[70] Inc. Ventana Systems. Vensim help. https://www.vensim.com/documentation/22090.html. Accessed: 2021-04-19.

[71] Inc. Ventana Systems. Vensim help. https://www.vensim.com/documentation/fn_sample_if_true.html. Accessed: 2021-04-05.

[72] Inc. Ventana Systems. Vensim help. https://www.vensim.com/documentation/stream_id.html. Accessed: 2021-05-30.

[73] w3sDesign. The gof design patterns memory - learning object-oriented design & programming. http://w3sdesign.com/?gr=s04&ugr=proble#gf, Oct 2017. Accessed: 2021-04-17.

[74] XE.com. 19 usd to eur: Convert dólares estadounidenses to euros. https://www.xe.com/es/currencyconverter/convert/?Amount=19&From=USD&To=EUR. Accessed: 2021-02-04.

[75] XE.com. 40 usd to eur: Convert dólares estadounidenses to euros. https://www.xe.com/es/currencyconverter/convert/?Amount=40&From=USD&To=EUR. Accessed: 2021-02-04.

[76] XE.com. The worlds trusted currency authority: Money transfers & free exchange rate tools. https://www.xe.com/es/currencyconverter/convert/?Amount=1995&From=USD&To=EUR. Accessed: 2021-05-30.