Universidad de Valladolid

ESCUELA DE INGENIERÍA INFORMÁTICA

GRADO EN INGENIERÍA INFORMÁTICA
MENCIÓN EN INGENIERÍA DE SOFTWARE

Desarrollo de nuevas funcionalidades del software PySD para la traducción del modelo MEDEAS de lenguaje Vensim a Python

Alumno/a:
Diego Rodrigo Verdugo

Tutor/es/as:
Yania Crespo González-Carvajal
David Escudero Mancebo

Resumen

Este trabajo de fin de grado se engloba dentro del proyecto de investigación Locomotion, el cual tiene como objetivo mejorar las herramientas de simulación que se utilizan como apoyo en la toma de decisiones políticas relacionadas con el cambio climático. [47]

En concreto, este trabajo de fin de grado tiene como objetivo la traducción del modelo MEDEAS, un modelo capaz de simular computacionalmente el efecto que tendrían en el ecosistema y en el sector energético las diferentes decisiones políticas, de lenguaje Vensim a lenguaje Python mediante el desarrollo de mejoras para el software de traducción y simulación PySD.

Para la realización de este proyecto se ha seguido la metodología ágil Scrum, permitiendo una mayor tolerancia a los cambios en el proyecto.

Las mejoras realizadas al traductor PySD desarrolladas a lo largo de este proyecto permiten traducir el modelo MEDEAS además de servir como aporte a la comunidad mediante su publicación en la plataforma Github.

Abstract

This final degree project is included within the Locomotion research project [47], which aims to improve the simulation tools used to support political decision-making related to climate change.

To be more specific, this project principal goal is to translate the MEDEAS model from Vensim language to Python language through the development of improvements for PySD translation and simulation software.This model is capable of computationally simulating the effect that different political decisions would have on the ecosystem and energy sector.

The agile Scrum methodology has been followed throughout the development of the project, allowing greater tolerance for changes.

The improvements that have been developed and made to the PySD translator due to this project allow the MEDEAS model to be translated, alongside serving as a contribution to the community by its publication on the Github platform.

Índice general

Resumen **III**

Abstract **V**

Lista de figuras **1**

1. Introducción **3**

1.1. Motivación 3

1.2. Objetivos 4

1.3. Estructura del documento 4

2. Plan de proyecto **7**

2.1. Scrum 7

2.1.1. Características 7

2.1.2. Artefactos 8

2.1.3. Participantes 8

2.1.4. Aplicación de Scrum 8

2.2. Planificación inicial 9

2.2.1. Plan de Riesgos 9

2.2.2. Riesgos producidos 11

2.3. Evolución del proyecto 11

2.3.1. Sprint 1 11
2.3.2. Sprint 2 12
2.3.3. Sprint 3 12
2.3.4. Sprint 4 12
2.3.5. Sprint 5 12
2.3.6. Sprint 6 13
2.4. Coste del proyecto 13

3. Sistemas Dinámicos 15

3.1. Qué es un sistema dinámico 15
3.2. Vensim 16
3.2.1. Estructuras específicas de Vensim 19
3.3. MEDEAS 20
3.4. Método de Euler 21

4. Control de versiones 25

4.1. Git 25
4.2. Github 25

5. Parsimonious 27

5.1. Qué es Parsimonious 27
5.2. Estructura de un Parser 27
5.3. Como funciona un parser PEG 28
5.4. Gramática en Parsimonious 29
5.5. Patrón Visitor 30
5.5.1. Cuando usar el Patrón Visitor 30
5.5.2. Patrón Visitor en Parsimonious 32

6. PySD **37**

6.1. Qué es PySD 37

6.2. Instalación de PySD 37

6.3. Entornos Virtuales 39

6.4. Estructura de PySD 39

6.5. Traducción de Vensim 40

6.5.1. file_structure_grammar 41

6.5.2. model_structure_grammar 42

6.5.3. component_structure_grammar 43

6.5.4. expression_grammar 44

6.5.5. lookup_grammar 44

6.5.6. Funcionamiento 45

6.6. Ejecución 48

6.6.1. Load 49

6.6.2. Initialization 50

6.6.3. Run 51

6.6.4. Caché 52

7. Aportación a PySD **55**

7.1. Variables 55

7.2. Fichero Excel grande 56

7.3. Lectura de Excel compleja 57

7.4. Subscripts 58

7.5. Funciones 60

7.6. Espaciados 61

7.7. Definición de lookups 61

7.8. GET DIRECT CONSTANTS 62

7.9. En detalle 62

8. Tests **63**

8.1. Tests 63

8.2. Tests Unitarios 63

8.2.1. Tests unitarios en PySD 65

8.3. Tests de Integración 66

8.3.1. Tests de Integración en PySD 66

8.4. Integración Continua 67

9. Conclusión y trabajo futuro **69**

9.1. Conclusión 69

9.2. Trabajo futuro 69

Bibliografía **71**

A. Manual de instalación y uso **75**

A.1. Instalación 75

A.2. Ejecución 76

Índice de figuras

3.1. Modelo Vensim 17
3.2. Representación de un lookup 20
3.3. Error acumulado en el método de Euler 22
3.4. Comparación de distintos pasos 24

5.1. Flujo de un parser 28
5.2. Problema para el Patrón Visitor 31
5.3. Solución con el Patrón Visitor 32

6.1. Estructura de PySD 40
6.2. Diagrama de clases de ejecución 49
6.3. Diagrama de clases simplificado 50

Capítulo 1

Introducción

1.1. Motivación

El modelo desarrollado por el proyecto MEDEAS (Ver sección 3.3), el cual finalizó en 2018, simula computacionalmente la evolución del medio ambiente en función de las decisiones que se lleven a cabo.

El proyecto Locomotion [47] en el que se enmarca este trabajo de fin de grado tiene como objetivo mejorar el modelo desarrollado en el proyecto MEDEAS y prepararlo para utilizarlo en proyectos futuros.

El modelo MEDEAS es un modelo complejo con el que se han simulado diferentes escenarios para obtener unas conclusiones. Pero más allá de esto, se desea poder permitir al público simular la evolución del medio ambiente tomando sus propias decisiones para concienciar de la importancia que tienen. Sin embargo, el modelo actualmente se desarrolla en el lenguaje Vensim, el cual no permite utilizarlo en otros proyectos, como una página web, ni permite a la mayoría de usuarios ejecutar sus propias simulaciones, ya que es necesario obtener una licencia de pago de Vensim para ello.

La solución tomada para solucionar esto ha sido traducir el modelo a un lenguaje más polivalente y gratuito como es Python. Al traducir el modelo a Python se permite utilizarlo otros proyectos, y al publicar el código del mismo se permite que cualquier persona con conocimientos de informática ejecute sus propias simulaciones de forma gratuita e incluso lo utilice en sus propios proyectos.

Se ha realizado anteriormente una traducción a código Python de manera semi automática, pero es un proceso costoso que se debería repetir en cada actualización del modelo.

Con el ánimo de traducir automáticamente este modelo tiene lugar el siguiente trabajo de fin de grado.

1.2. Objetivos

El objetivo de este TFG consiste en lograr una traducción automática del código perteneciente al proyecto MEDEAS (Ver sección 3.3). Para lograrlo se han propuesto los siguientes objetivos:

- **Comprensión del lenguaje Vensim:** El aprendizaje del funcionamiento de este lenguaje con el que nunca he trabajado. (Ver sección 3.2)
- **Comprensión del traductor PySD:** Estudiar el traductor en desarrollo que existe actualmente para poder realizar cambios, así como su documentación para facilitar la curva de aprendizaje a nuevos desarrolladores. (Ver sección 6.1)
- **Contribución al repositorio de PySD:** Mediante los intentos de traducción de MEDEAS, agregar mejoras al traductor PySD para lograr la traducción, y compartir el código con la comunidad.

Estos no fueron los primeros objetivos del TFG, sino que fueron modificados durante el mismo.

Al inicio del proyecto ya contábamos con una verisón traducida de MEDEAS: `pymedeas_w`. Cuyo problema era la velocidad de ejecución. (Ver sección 3.3)

El objetivo inicial fue la mejora del tiempo de ejecución del archivo `pymedeas_w`, para lo que se propusieron los siguientes objetivos:

- Comprender el funcionamiento de `pymedeas_w` y la parte de ejecución/simulación de PySD
- Identificar los cambios desarrollados en el fork del usuario rogersamso (Ver sección 4.2) y compararlos con el repositorio original
- Encontrar el motivo de la lentitud de la versión traducida
- Acelerar la ejecución de las simulaciones

Sin embargo, al comprobar que el archivo `pymedeas_w` no había sido traducido de manera completamente automática por PySD, se decidió abandonar estos objetivos.

1.3. Estructura del documento

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

- El capítulo 2 expone la metodología utilizada para el desarrollo de este proyecto, así como el plan de riesgos y el presupuesto del mismo.

- El capítulo 3 introduce los conceptos necesarios para la comprensión de los proyectos y tecnologías en los que se basa este trabajo.
- El capítulo 4 explica la relación entre la plataforma GitHub y este proyecto, así como la manera en la que se ha contribuido y los repositorios que se han utilizado.
- El capítulo 5 introduce el funcionamiento de los parsers y explica el funcionamiento específico del parser Parsimonious.
- El capítulo 6 Explica el funcionamiento del traductor PySD sobre el cual nos hemos basado para traducir MEDEAS.
- El capítulo 7 enumera y explica las aportaciones realizadas al código del software PySD para lograr la traducción de MEDEAS.
- El capítulo 8 introduce los tests así como su uso en PySD y el resultado de los mismos tras las aportaciones.
- El capítulo 9 expone finalmente las conclusiones de este trabajo y posibles las líneas de trabajo futuras.

Capítulo 2

Plan de proyecto

2.1. Scrum

Scrum es un marco de trabajo usado para gestionar el desarrollo de productos complejos. Scrum no pretende ser un proceso para construir productos sino un marco de trabajo dentro del cual se pueden emplear diferentes procesos y prácticas de desarrollo. [39]

2.1.1. Características

Su principales características son:

- Desarrollo incremental en lugar de planificación y ejecución completa del proyecto. Se producen constantemente iteraciones (en Scrum se denominan sprints) hasta que se considera el proyecto terminado.
- La calidad del resultado está condicionada por la experiencia de las personas en equipos auto organizados.
- Se solapan las fases de desarrollo.
- Seguir la filosofía de desarrollo ágil. [37]

Scrum se basa, como hemos mencionado, en el desarrollo incremental gracias a los sprints. Otro concepto importante al que Scrum otorga mucha importancia son las revisiones, para lograr transparencia y comunicación, por ello se planifican diferentes tipos de reuniones:

- Reunión de planificación de sprint: Al principio de cada sprint se decide en que se va a trabajar durante el próximo sprint.

- Reunión diaria: Se trata de una breve reunión en la que discutir los problemas que se han encontrado a lo largo de una jornada.
- Reunión de revisión de sprint: Al final de cada sprint se exponen los avances.
- Retrospectiva del sprint: Los implicados dan sus impresiones sobre los avances. [37]

2.1.2. Artefactos

Los artefactos de Scrum son:

- Pila de producto: Lista de requisitos creada a partir de una versión inicial del producto, pero cambiará durante el desarrollo.
- Pila de sprint: Lista de trabajos que debe realizar el equipo durante el sprint.
- Sprint: Nombre que recibe cada iteración.
- Incremento: Resultado de cada sprint. [40]

2.1.3. Participantes

El equipo en Scrum se compone de 3 roles:

- Dueño del producto: Es el encargado de optimizar y maximizar el valor del producto ya que es el intermediario entre el equipo y los interesados. También es el encargado de estructurar y priorizar las tareas.
- Facilitador: Tiene 2 funciones principales. Por una parte es el encargado de gestionar y asegurar que el proceso Scrum se cumpla correctamente. Su otra tarea es eliminar los impedimentos que se encuentren en la organización que afecten al valor o integridad del proceso Scrum.
- Desarrolladores: El equipo de desarrolladores suele estar formado por entre 3 y 9 personas auto organizadas, los cuales se encargan de desarrollar el producto. [38]

La última categoría de participante son los interesados y/o inversores, los cuales guiarán en cierta medida el rumbo del desarrollo mediante las retrospectivas de sprint.

2.1.4. Aplicación de Scrum

Se ha decidido usar Scrum como metodología para la realización de este TFG. Sin embargo, no se ha seguido la metodología pura, sino que se ha modificado para adaptarlo a la situación de este proyecto.

El equipo de desarrolladores está formado únicamente por el autor de este TFG, mientras que el resto de roles los cumplirán los tutores del mismo. Cumpliendo Yania con el rol de facilitadora y David el rol de dueño del producto.

Los interesados en este proyecto son:

- Dueños del proyecto europeo Locomotion
- Interesados del proyecto PySD

La duración del sprint se ha definido al principio del proyecto en 2 semanas debido a las limitaciones de personal en el equipo de desarrollo y las horas de disponibilidad. No se han realizado las reuniones diarias, sino que se han realizado reuniones semanales.

2.2. Planificación inicial

Tomando como referencia el número de créditos de un TFG en el grado de ingeniería informática de Valladolid y el número de horas de trabajo estimadas por crédito obtenemos un tiempo estimado de trabajo de 300 horas.

Tomando como inicio del proyecto el día 10/02/2020 se han planeado 10 sprints, tomando una carga de trabajo de 15 horas semanales, resultando en 300 horas de trabajo.

1	10/02/2020	24/02/2020
2	24/02/2020	09/03/2020
3	09/03/2020	23/03/2020
4	23/03/2020	06/04/2020
5	06/04/2020	20/04/2020
6	20/04/2020	04/05/2020
7	04/05/2020	18/05/2020
8	18/05/2020	01/06/2020
9	01/06/2020	15/06/2020
10	15/06/2020	29/06/2020

2.2.1. Plan de Riesgos

Hay que intentar prever los problemas que puedan surgir durante el desarrollo de un proyecto, para ello se ha elaborado el siguiente plan de riesgos en el cual estimamos la probabilidad de que sucedan ciertos escenarios, el grado de impacto que tendría en la planificación y la forma de proceder ante ellos para poder continuar con el proyecto.

R01 Enfermedad

El desarrollador de este TFG puede enfermar y no estar en condiciones de trabajar.

Probabilidad: Baja

Impacto: Alto

Plan de mitigación: Ninguno

Plan de contingencia: Aplazar los objetivos al siguiente sprint

R02 Falta de formación

No se puede completar algún objetivo debido a que el desarrollador no tiene el conocimiento necesario para llevarla a cabo.

Probabilidad: Media

Impacto: Medio

Plan de mitigación: Dedicar parte de las horas de trabajo a la formación

Plan de contingencia: Consultar profesionales en ese área

R03 Fallo de planificación

La planificación no puede ser llevada a cabo.

Probabilidad: Baja

Impacto: Alto

Plan de mitigación: Utilizar Scrum, al ser una metodología ágil se reduce el impacto.

Plan de contingencia: Ninguno

R04 Fallo en equipos de hardware

El equipo sobre el que se trabaja se estropea.

Probabilidad: Muy baja

Impacto: Medio

Plan de mitigación: Ninguno

Plan de contingencia: Utilizar ordenador personal

R05 Pérdida de datos

Se pierden los cambios realizados durante el proyecto.

Probabilidad: Baja

Impacto: Alto

Plan de mitigación: Utilizar software de control de versiones.

Plan de contingencia: Ninguno

2.2.2. Riesgos producidos

Durante el desarrollo del proyecto se han producido situaciones de riesgo planificadas y no planificadas.

El riesgo 01 Enfermedad, tuvo lugar durante el Sprint 5, por lo que ese Sprint se aplazó.

El riesgo 03 Fallo de planificación, tuvo lugar al tener que cambiar los objetivos generales de trabajo, produciendo un retraso en la finalización del mismo.

El riesgo 02 Falta de formación, tuvo lugar al centrarse en el ámbito de la traducción de ficheros Vensim, de los cuales el alumno no es experto, por lo que se recurrió en repetidas ocasiones a personal cualificado en esta materia para resolver dudas sobre el funcionamiento de este lenguaje.

Además de estos riesgos que tuvimos en cuenta durante la planificación, también han sucedido otros riesgos:

Situación de confinamiento debido al COVID-19, impidiendo trabajar en el centro y trasladar el entorno de trabajo a otro ordenador.

Cambio de objetivo general del TFG, debido a que el objetivo inicial no era viable.

2.3. Evolución del proyecto

2.3.1. Sprint 1

Durante el primer sprint se ha definido el objetivo de preparar el entorno de trabajo para poder ejecutar diferentes versiones de PySD en la máquina del trabajo (Ver seccion 6.1). Traducir un pequeño ejemplo y entender el resultado.

2.3.2. Sprint 2

Durante el segundo sprint se ha definido el objetivo de traducir pequeños ejemplos Vensim con 2 versiones diferentes de PySD para entender su funcionamiento y ver las diferencias en los archivos resultantes y en los pasos de ejecución. (Ver seccion 3.2)

Las versiones de PySD utilizadas son:

- PySD oficial
- PySD fork del usuario rogersamso

(Ver seccion 4.2)

2.3.3. Sprint 3

Durante el tercer sprint, al no apreciar diferencias entre las traducciones generadas por ambas versiones de PySD, se ha definido el objetivo de buscar sus diferencias en el código y traducir MEDEAS, ya que es un archivo mucho más complejo para poder observar diferencias. (Ver sección 3.3)

En este sprint observamos que no podemos traducir MEDEAS con ninguna de las dos versiones de PySD, por lo que a mediados de sprint, durante la reunión semanal, se modifican los objetivos de este sprint. El nuevo objetivo es descargar la versión ya traducida y estudiar su funcionamiento.

Durante el transcurso de este sprint el equipo dejó de reunirse presencialmente y el alumno dejó de acudir al centro para trabajar desde casa debido a la situación excepcional del COVID-19

2.3.4. Sprint 4

Durante el cuarto sprint se establecen los objetivos:

- Instalar el entorno de trabajo en el ordenador personal del alumno
- Continuar el estudio de la ejecución de PySD

2.3.5. Sprint 5

Este sprint fue aplazado debido a enfermedad del alumno.

Durante este sprint, ante la imposibilidad de traducir MEDEAS, se toma la decisión de cambiar el objetivo del TFG para hacer una aportación al repositorio github de PySD que agregase la funcionalidad necesaria para traducir MEDEAS.

Como el aporte sobre el que se va a trabajar se desarrolla en la parte de traducción, los objetivos de este sprint se establecen como:

- Documentar la etapa de ejecución estudiada
- Comenzar el estudio de la etapa de traducción
- Búsqueda de la mejor versión sobre la que trabajar

2.3.6. Sprint 6

Durante el sexto sprint, ya se ha encontrado una versión que implementa algunas funcionalidades que necesitaremos, aunque de manera básica.

Debido a que no conocemos que funcionalidades vamos a necesitar implementar, sino que tenemos que implementar las funcionalidades según avancemos en la traducción de MEDEAS, no es posible establecer los objetivos del sprint con anterioridad, por lo tanto, se realizará un seguimiento por parte de los tutores con una frecuencia más amplia de 2 días a la semana para prestar suporte sobre el funcionamiento de Vensim y se establecen objetivos aproximados del porcentaje de traducción obtenido.

2.4. Coste del proyecto

Se sabe que el salario medio de un ingeniero informático es de 25600€/año a lo que hay que añadir los costes sociales que ascienden a 8450€/año. Este salario corresponde a un trabajador en jornada completa, es decir, 40 horas semanales. Puesto que este trabajo corresponde a 300 horas de trabajo podemos calcular un presupuesto estimado de 4256€.

A este presupuesto hay que añadir los costes materiales, que en este proyecto es un ordenador de un precio estimado de 700€.

Para calcular el coste que aporta este dispositivo tenemos que calcular su depreciación a lo largo del tiempo. Necesitamos los siguientes valores:

- **El valor inicial** o precio de adquisición.
- **Su vida útil** o duración estimada en años.
- **Su valor residual** o valor estimado cuando su vida útil ya ha terminado

Suponiendo una vida útil de 5 años y un valor residual de 50€ podemos calcular su depreciación como:

$$Cuota\ de\ depreciación\ anual = \frac{valor\ inicial - valor\ residual}{vida\ útil\ en\ años} \quad [41]$$

Obteniendo una depreciación de 130€ al año. Sin embargo, este proyecto tiene una duración estimada de 5 meses, por lo que la depreciación en este tiempo será de 54€.

De esta forma obtenemos un presupuesto de **4310€**

Capítulo 3

Sistemas Dinámicos

3.1. Qué es un sistema dinámico

Se trata de una metodología orientada al modelado y simulación por ordenador de sistemas complejos de diferentes áreas. [14]

Los sistemas dinámicos son sistemas de ecuaciones cuyas variables cambian con respecto a alguna variable, generalmente el tiempo.

Más concretamente vamos a trabajar con sistemas dinámicos continuos. En estos sistemas el tiempo varía de forma continua y el sistema se expresa con ecuaciones diferenciales. [16]

Para representar un sistema se construye un modelo, el cual es una representación simplificada de un sistema.

El estado de un modelo es un conjunto de variables dependientes cuyo conocimiento en un instante de tiempo, asumiendo conocidos los valores de las entradas, permite calcular el valor de cualquier otra variable dependiente en dicho instante. Y si el modelo es determinista, conocido el estado del modelo en un instante de tiempo y las entradas en todo momento, es posible predecir la evolución de todas las variables. [15]

Estas variables a las que nos referimos pueden ser de diferentes tipos, los más importantes para la compresión de este trabajo son:

- **Existencia o Stock:** Representa una cantidad de algo. Por ejemplo, en una cuenta bancaria el Stock sería la cantidad de dinero de la cuenta.
- **Flujo o Flow:** Representa una cantidad con respecto al tiempo. Por ejemplo, en el ejemplo anterior, un flujo sería el sueldo, el cual es una cantidad de dinero por cada mes. [6]

Este tipo de modelos y simulaciones sirven para ayudarnos a comprender la estructura y comportamiento de sistemas complejos y posteriormente a analizar los posibles escenarios y tomar decisiones sobre los mismos.

3.2. Vensim

Vensim [3] es un software propietario desarrollado por Ventana Systems [4] desde 1990, el cual permite desarrollar, analizar y simular modelos de dinámica de sistemas mediante una interfaz gráfica dibujando diagramas causales o en versión texto.

En caso de construir un modelo con la interfaz gráfica se genera un archivo con la extensión **mdl** con su correspondiente sintaxis en texto como si se hubiera programado en modo texto.

En la Figura 3.1 se puede observar un modelo programado en Vensim que define el comportamiento de la temperatura de una tetera [7]

En esta figura tan solo podemos ver la representación del modelo, no podemos ver las ecuaciones que definen el comportamiento y se han introducido durante el desarrollo del mismo.

En este ejemplo hay tan solo un Stock, correspondiente a la temperatura de la tetera. Esta temperatura se verá afectada por la temperatura ambiente de la sala en la que se encuentra, la cual se supone constante.

Dichas ecuaciones simulan la pérdida de temperatura de la tetera mediante la ley de enfriamiento de Newton [8]

$$\frac{dT}{dt} = -k(T - T_A)$$

Donde k es una constante de enfriamiento y T_A la Temperatura Ambiente.

Este modelo lo podemos abrir en formato texto y veremos las ecuaciones que se han definido.

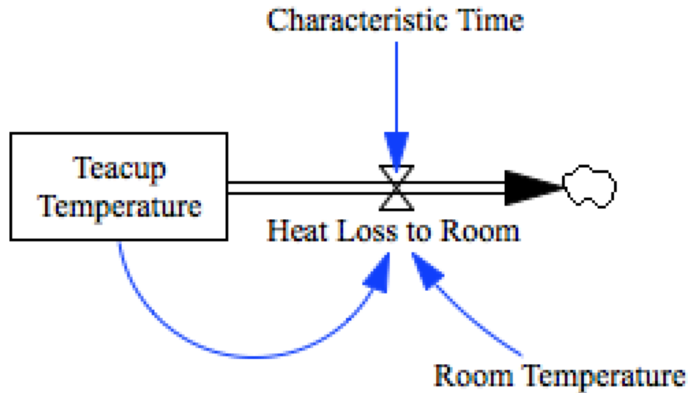

Figura 3.1: Modelo Vensim

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

Heat Loss to Room =
         (Teacup Temperature - Room Temperature) /
            ↪ Characteristic Time
        ~ Degrees Fahrenheit/Minute [-inf, inf]
        ~ This is the rate at which heat flows from the cup
           ↪ into the room.
        |

Room Temperature =
         70
        ~  Degrees Fahrenheit [-459.67, inf]
        ~ The room temperature cannot go below absolute zero.
        |

Teacup Temperature =
         INTEG (-Heat Loss to Room, 180)
        ~ Degrees Fahrenheit [32.0, 212.0]
        ~ The model is only valid for the liquid phase of tea.
        |
```

Como se puede ver en este ejemplo, se han definido 4 ecuaciones para representar el sistema.

$$\begin{cases} Characteristic\ Time & = & 10 \\ Room\ Temperature & = & 70 \\ Heat\ Loss\ to\ Room & = & \dfrac{Teacup\ Temperature - Room\ Temperature}{Characteristic\ Time} \\ Teacup\ Temperature & = & 180 + \int -Heat\ Loss\ to\ Room \end{cases}$$

Podemos ver el uso de la fórmula de la ley de Newton anteriormente mencionada:

- Characteristic Time equivale a $\frac{1}{k}$,
- Room Temperature equivale a la temperatura ambiente T_A
- Heat Loss to Room equivale a la derivada que se ha mencionado anteriormente $k(T - T_A)$
- Teacup Temperature es el resultado de la temperatura de la tetera suponiendo la temperatura inicial de 180º

En este pequeño ejemplo podemos ver la estructura del código que genera Vensim. Las variables que veíamos en la Figura 3.1 se definen escribiendo su nombre seguido del símbolo '=' y su definición, que se divide en 3 elementos separados por el carácter '∼'

- **Ecuación:** Un número en caso de ser una constante o una ecuación que dependa de otras variables o funciones predefinidas en Vensim, como por ejemplo la integración *INTEG* que podemos observar en el ejemplo.
- **Unidades:** Un texto que ayude a comprender la naturaleza de la variable.
- **Comentario:** Un texto que describa el comportamiento de la variable.

Las unidades y el comentario son campos opcionales.

Al finalizar la definición se encuentra el carácter '|'

Para poder ejecutar una simulación tenemos que especificar siempre unos parámetros que se verán reflejados en el archivo mdl como definiciones de constantes. Dichos parámetros son:

- **FINAL TIME:** El tiempo hasta el que se tiene que simular.
- **INITIAL TIME:** El tiempo en el que empieza la simulación. Suele utilizarse 0 si las unidades son minutos, o un año en específico al simular modelos más grandes.

- **SAVEPER:** La frecuencia de tiempo con la que se almacenarán los valores de los stock para mostrar como resultado.
- **TIME STEP:** El paso de tiempo entre cada iteración de la simulación.

Con estos parámetros definidos el modelo estaría completo, sin embargo, si se crea el modelo con la interfaz gráfica, después de estos parámetros podemos encontrar información que Vensim guarda para mostrar en su interfaz, no intervienen en la simulación por lo que no hace falta tenerlos en cuenta.

3.2.1. Estructuras específicas de Vensim

Subscripts

Los subscripts son un recurso que ofrece Vensim para permitir a una variable representar más de un valor. Para utilizarlos hay que definirlos como uno nombre seguido del carácter ':' y seguido de una secuencia de elementos separados por comas denominados **subscript elements**. [9]

Por ejemplo, de esta manera definimos un subscript de paises y uno de deportes:

```
paises: italia, francia, alemania ~~|
deportes: futbol, baloncesto, tenis ~~|
```

Para utilizarlos, tenemos que utilizar antes de una variable los caracteres `[]` entre los cuales podemos incluir hasta 8 subscripts separados por comas.

Por ejemplo, podemos definir la cantidad de nacimientos en los paises como:

```
nacimientos[paises] =
    poblacion[paises] * factor de natalidad[paises] ~~|
```

O utilizando los 2 subscripts podemos definir los aficionados al baloncesto de cada pais como:

```
aficionados[paises, baloncesto] =
    poblacion[paises] * factor de aficionados[baloncesto] ~~|
```

Como se puede apreciar, se puede consultar el valor de una variable con subscripts mediante el nombre del propio subscript o mediante uno de sus subscript elements.

Lookups

Los lookups se pueden definir de diferentes formas, pero todas ellas son un conjunto de pares de valores relacionados entre sí de una forma no lineal. La utilidad de esta herra-

mienta es la posibilidad de representar estas relaciones no lineales así como obtener valores interpolados de estos. [10]

En la Figura 3.2 se representa en lookup con los valores (0,0),(1,2),(2,3) y podemos ver de qué manera se realiza la interpolación de los datos que no están explícitamente indicados en la definición del lookup.

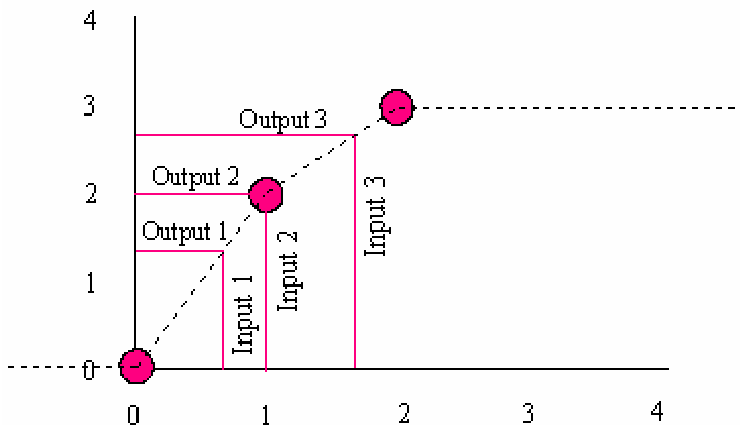

Figura 3.2: Representación de un lookup
[10]

3.3. MEDEAS

Modelling the Energy Development under Environmental And Socioeconomic constraints, en adelante MEDEAS [11], es un proyecto europeo que trata de construir un modelo capaz de simular el efecto que tendrían en el ecosistema y en el sector energético las diferentes decisiones políticas.

En este proyecto se han creado varios modelos desarrollados en Vensim:

- **Modelo global:** Contiene más de 4000 variables y datos desde el año 1995, capaz de simular la evolución del mundo hasta el año 2050 gracias a sus más de 37000 líneas de código.
- **Modelo europeo:** Utilizando el modelo global como marco es el núcleo del proyecto, ya que se pretende utilizar para la toma de decisiones a nivel europeo.
- **Modelo a nivel de País:** Se han creado 2 modelos a nivel de país, Bulgaria y Austria. Con el fin de observar el problema desde diferentes perspectivas. [12]

Al estar desarrollado en Vensim y ser un modelo complejo no es posible utilizarlo sin licencia de pago. Por ello, se generó una traducción del modelo global a código Python con ayuda del traductor PySD. De esta manera cualquiera puede utilizar estos modelos.

En este proyecto se simularon 3 escenarios diferentes para tratar de llevar a cabo una transición a una economía baja en carbono:

- **BAU:** Seguir aplicando las prácticas actuales.
- **OLT:** Intensificar ligeramente el aumento en las fuentes de energía renovables.
- **TRANS:** Intensificar al máximo posible las fuentes de energía renovables. [13]

Se pretende que cualquiera pueda crear su propio escenario, por lo tanto las variables de los escenarios se configuran desde un archivo excel, por lo tanto es posible crear un escenario totalmente nuevo o modificar los existentes.

Actualmente existen varias versiones del código de MEDEAS disponibles en su web oficial [12].

Para el desarrollo de este TFG se utilizaron los siguientes archivos:

- `MEDEAS_W_v1_3`: Corresponde con la versión 1.3 global de MEDEAS.
- `pymedeas_w`: Traducción de MEDEAS a código Python de manera semi automática.
- `inputs_W`: Fichero excel con las variables configurables de MEDEAS.

3.4. Método de Euler

El método de integración por Euler es un método numérico para calcular de manera aproximada la integral de una función diferencial acotada conociendo su punto de inicial. [24] [27]

Este método de integración se basa en la definición de la recta tangente, ya que una recta tangente a una función f(x) en un punto a toma el valor f(a) y la pendiente de la función en ese punto. [28]

Recordemos que sabemos el punto de inicio de la función que queremos integrar, ya que está acotada, podemos obtener la recta tangente mediante la derivada de la función y avanzar nuestro punto inicial un paso.

El paso, o intervalo de integración, representa cuantas unidades avanzamos en cada recta tangente.

Ahora que hemos calculado el segundo punto de la función podemos seguir repitiendo estos cálculos para calcular el siguiente punto. Es decir, calcularemos la recta tangente en

el nuevo punto para corregir la dirección de la curva y avanzaremos otro paso. Iterando de esta manera calcularemos la curva de la función de manera aproximada como se muestra en la Figura 3.3.

La fórmula matemática que equivale a este método numérico es:

$$y_i = f(x_i) = f_(x_{i-1}) + h * f'(x_{i-1})$$

Donde h es el paso.

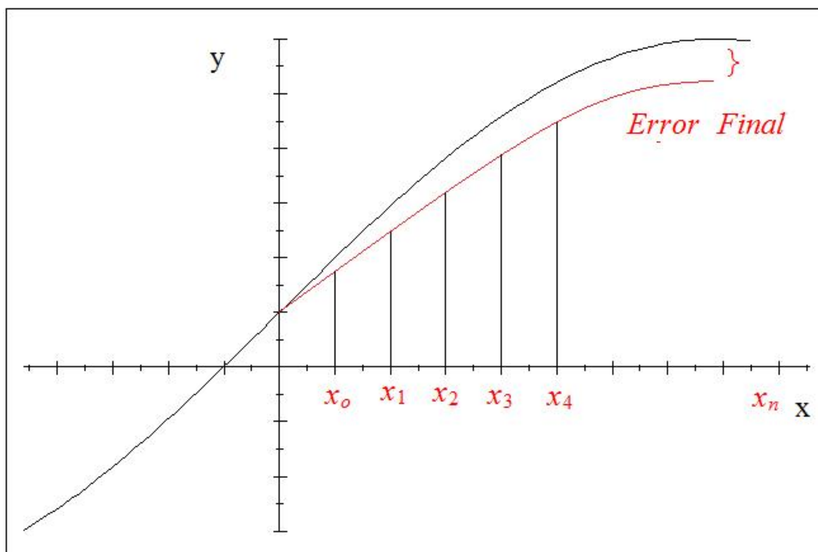

Figura 3.3: Error acumulado en el método de Euler
[24]

Como podemos ver en la Figura 3.3, no se representa de manera exacta la función, si no que se comenten errores, los cuales son:

- **Error local:** En cada iteración cometemos un pequeño error al avanzar por la recta tangente en lugar de por la curva real de la función.
- **Error de truncamiento o redondeo:** El número de decimales que podemos utilizar para calcular es finito, mientras que el número de decimales de los números representados puede ser infinito. Al redondear o truncar estos para realizar el cálculo acarreamos un error.
- **Error global:** El único punto libre de errores es el primero, a partir de dicho punto se avanza desde un punto que realmente no pertenece a la recta, por lo cual se acumulan los errores locales y de truncamiento en cada paso haciendo que cada punto sea más inexacto. [25]

Podemos reducir el error cometido reduciendo el paso, ya que reduciremos el error local, a costa de incrementar el error de truncamiento, pero este último es mínimo en comparación.

De esta manera podemos comparar una función y sus representaciones calculadas con el método de Euler con 2 pasos distintos, en la Figura 3.4 se ve esta situación

En la figura 3.4 podemos ver como la recta verde se aproxima más que la azul, esto es debido a que se ha calculado con un paso más pequeño. Sin embargo, reducir el paso en exceso elevará también el tiempo de cómputo, ya que si queremos representar la función desde el punto A hasta el punto B con un paso h, es necesario calcular un número de pasos igual a

$$\frac{(B-A)}{h}$$

Éste método es el utilizado para calcular la evolución de la temperatura de la tetera que tratamos en la sección 3.2.

En dicho ejemplo, necesitábamos calcular la integral

$$f(x) = \int -k(T - T_A)$$

Conociendo el punto inicial $f(0) = 180$.

Para calcular aproximadamente la integral $f(x)$ se utiliza el método de Euler, por lo que solo necesitamos el valor inicial $f(0)$ y la derivada $f'(x)$.

Por ejemplo, para calcular el valor de la integral en el siguiente minuto se puede calcular como:

$$f(1) = f(0) + 1 * f'(0)$$

Tomando como intervalo de integración 1. La derivada la conocemos, por lo tanto:

$$f(1) = f(0) + 1 * (-k(T - T_A))$$

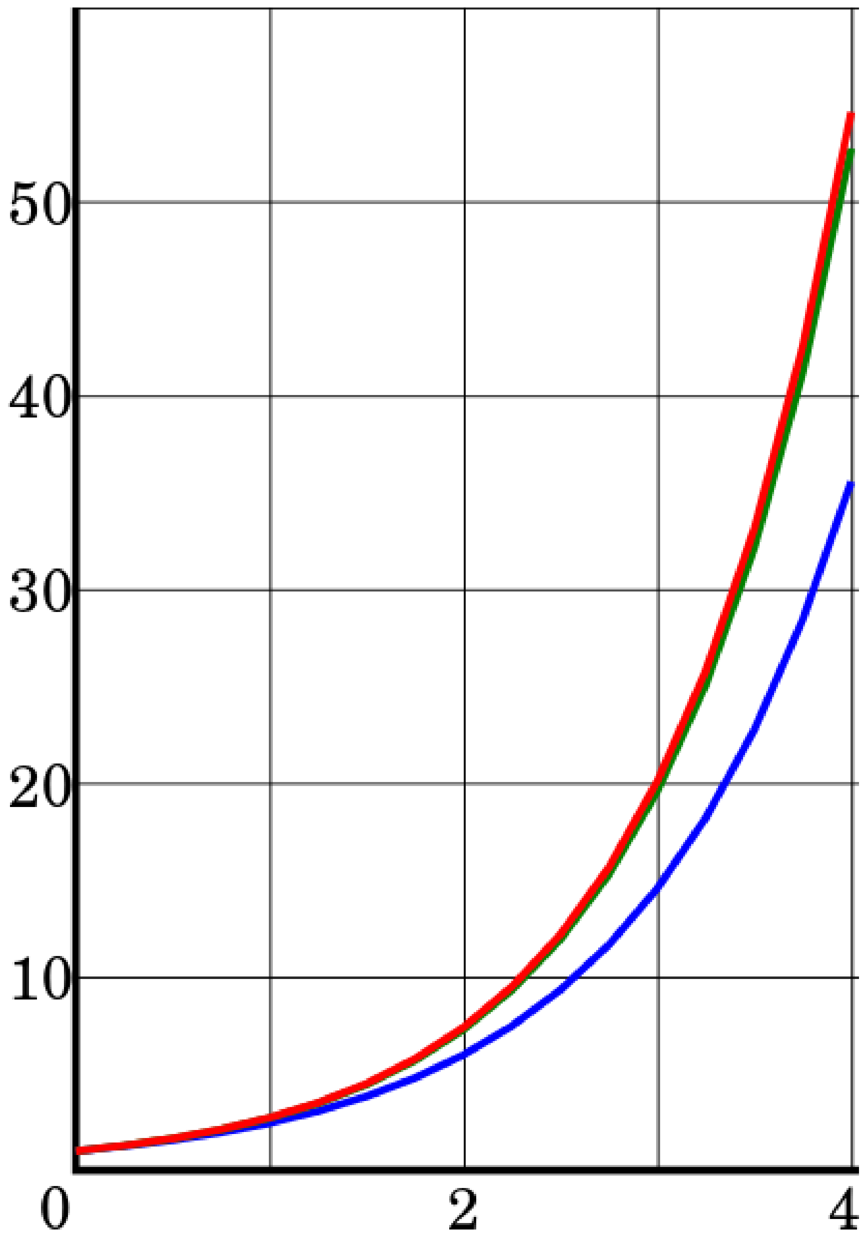

Figura 3.4: Comparación de distintos pasos
[25]

Capítulo 4

Control de versiones

4.1. Git

Git es un sistema de control específico de versión de fuente abierta creada por Linus Torvalds en el 2005.

Específicamente, Git es un sistema de control de versión distribuida, lo que quiere decir que la base del código entero y su historial se encuentran disponibles en la computadora de todo desarrollador, lo cual permite un fácil acceso a las bifurcaciones y fusiones.

Git es un sistema de control de versiones open source creado por Linus Torvalds en 2005.

Esta herramienta permite a los desarrolladores de un proyecto compartir el código del mismo, gestionar los cambios al código ordenadamente y con permisos, y mantener un historial de los cambios producidos con la finalidad de poder revisar que desarrollador introdujo cada cambio e incluso deshacer los cambios. [42]

4.2. Github

Github es una compañía sin ánimo de lucro que ofrece un servicio gratuito de hosting de repositorios git con una interfaz web intuitiva que agrega funcionalidades sociales para que los desarrolladores descubran los proyectos de otras personas y colaboren entre sí.

Durante este TFG se ha utilizado esta plataforma para colaborar con los desarrolladores del repositorio PySD para incluir mejoras y nuevas funcionalidades a este software.

Para explicar el funcionamiento de Github hay que aclarar algunos términos:

- **Fork:** consiste en crear una réplica del repositorio de otra persona en la que tu eres el propietario. Esta herramienta no consiste en plagiar, si no en facilitar un entorno con control de versiones configurado como el repositorio original para continuar el desarrollo.
- **Pull request:** es una petición en la que un desarrollador solicita al dueño de un repositorio la inclusión de algunos cambios que previamente ha desarrollado.
- **Merge:** es el proceso en el que 2 versiones diferentes de un mimo proyecto se unen en una sola versión que contiene el código de ambas. [42]

La manera en la que se puede colaborar con un proyecto es la siguiente:

- Crear un fork del repositorio con el que quieres colaborar
- Desarrollar los cambios normalmente con la ayuda de git como control de versiones
- Hacer una pull request para solicitar que se incluyan los cambios en el repositorio original
- El desarrollador del repositorio original puede aceptar los cambios o rechazarlos, github ofrece un pequeño servicio de mensajeria para intercambiar opiniones sobre los cambios que quieres introducir con el propietario del repositorio.

Versiones de PySD

Durante este TFG se ha trabajado con diferentes versiones del repositorio PySD:

- Repositorio original [43]
- Fork del usuario rogersamso [45]. Con esta versión se tradujo `MEDEAS MEDEAS_W_v1_3` a pymedeas_w
- Fork del usuario julienmalard [44]. Versión elegida como punto de partida de trabajo.

Como se acaba de mencionar, para contribuir en el proyecto PySD nos hemos basado en un fork existente sin terminar de desarrollar, perteneciente al usuario julienmalard.

Por lo tanto, para contribuir en el desarrollo hemos creado un fork de este repositorio, el cual es a su vez fork del repositorio original.

Al finalizar el TFG se ha creado una pull request para que el usuario julienmarald incorpore los cambios a su repositorio, y será él quien cree otra pull request al repositorio oficial para que se incluyan nuestros cambios en el software original.

Capítulo 5

Parsimonious

5.1. Qué es Parsimonious

Parsimonious es la librería que se utiliza en PySD para parsear los ficheros Vensim. [20]

Parsimonious es un PEG (Parser Expresion Grammar), este tipo de parsers nos ofrecen una serie de ventajas con respecto a otros como los populares parsers LL(1)

- No se separa el análisis léxico y el análisis sintáctico
- Son más fáciles de escribir
- No tienen límite de lookahead, podemos crear gramáticas equivalentes a gramáticas LL(k)
- No se crean gramáticas ambiguas [22]

Además, Parsimonious nos ofrece estas ventajas con un tiempo de ejecución lineal O(longitud de la gramática) y un gasto de memoria asequible O(longitud de la gramática * longitud del texto a parsear)

Para utilizar este parser primeramente se crea una gramática, mediante la cual se definen los elementos y estructura del código Vensim. Y crear una clase que herede de NodeVisitor, la cual implementa la lógica a seguir durante el parseo.

5.2. Estructura de un Parser

Un parser se compone normalmente de 2 partes:

En primer lugar necesitamos que un **analizador léxico** reciba el texto que queremos analizar y lo divida en tokens.

En segundo lugar un **analizador sintáctico** recibirá esos tokens en orden y los analiza de acuerdo a una gramática dada para construir una estructura de datos representativa del texto de entrada, generalmente un árbol sintáctico abstracto.

Supongamos un parser programado para analizar expresiones matemáticas, trabajaría como vemos en la Figura 5.1.

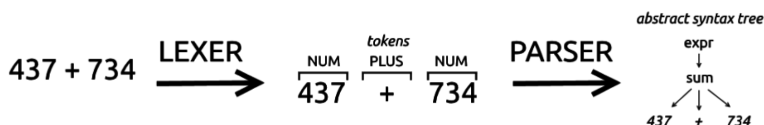

Figura 5.1: Flujo de un parser
[23]

Se recibe el texto correspondiente a una suma, el analizador léxico lo divide en 3 tokens: número, signo y número, el analizador sintáctico interpreta esos tokens según una gramática y deduce que es una expresión de tipo suma con los 2 operandos.

La definición de las gramáticas se componen de reglas, las cuales se definen según las posibles combinaciones de tokens.

Existen parsers que no necesitan un analizador sintáctico, este es el caso de los parser PEG como Parsimonious. Estos parsers se denominan 'scanerless parsers'

5.3. Como funciona un parser PEG

Los parsers PEG no deducen la regla de la gramática que se va a usar, simplemente comprueban secuencialmente todas las posibilidades hasta que una sea correcta. Por lo que el orden en el que se definan las reglas en la gramática es importante.

Esta manera de elegir las reglas perjudica la rapidez y la memoria, ya que se necesita implementar con técnicas de backtracking, pero permite generar gramáticas más fácilmente y no tener que preocuparse en exceso de la ambigüedad.

Realmente no erradicamos la ambigüedad solo por usar este tipo de parsers, sin embargo la evitamos. Por ejemplo, supongamos que definimos la gramática [22]:

```
saludo = A | B
A = "hola"
B = "hola"
```

Realmente esta gramática no se utilizaría, pero es un ejemplo sencillo para ilustrar el comportamiento del parser.

Supongamos que llega el token "hola":

Con una gramática libre de contexto como las que usaríamos con un parser LL(*) como ANTLR o LALR(1) como Yacc, al llegar al token 'hola' no sabríamos elegir entre la regla A o la regla B, por lo que tendríamos un problema de ambigüedad.

Con un parser PEG, iría probando cada regla secuencialmente en el orden que se han definido las alternativas de la regla saludo, entraría en A, emparejaría 'hola' y escogería esa regla. De esta manera en realidad nunca se utilizará B, pero no tenemos errores de ambigüedad.

Con este ejemplo también podemos apreciar la pérdida de eficiencia, si llega el token 'adios':

- un parser LL(1) no tiene ese token en el set de primeros tokens, terminará inmediatamente ya que no hay ninguna regla que lo cumpla.
- un parser PEG, sin embargo, probará la regla A, y después la regla B y seguiría probando cada una de las reglas definidas hasta que todas fallen.

Para evitar que el tiempo de ejecución sea exponencial se utiliza una caché, la cual recuerda las decisiones tomadas de cada token, por lo que la próxima vez que llegue ese token no tiene que recorrer todas las reglas hasta encontrar la correcta, iría directamente a la regla que funcionó la primera vez. De esta manera intercambiamos memoria por velocidad y conseguimos que el parser tenga una complejidad lineal en lugar de exponencial.

5.4. Gramática en Parsimonious

Para crear una gramática en Parsimonious tenemos que crear un objeto Grammar, el cual recibe como único parámetro un string que represente una gramática válida.

Una gramática válida es una secuencia de reglas separadas por un salto de línea, cada regla tiene un nombre seguido del carácter '=' y una definición de la misma.

Para definir una regla podemos utilizar las siguientes herramientas:

- `r"literal"`:El texto entrecomillado se considera literalmente, opcionalmente podemos usar el carácter 'r' antes de las comillas para tratar el texto como crudo y no tener que escapar caracteres
- espacio: Las secuencias de tokens separados por espacios significa que se esperan esos tokens en ese orden (sin los espacios entre medias) para esperar un espacio tendremos que usar la regla anterior

- / : el carácter '/' significa or, las reglas separadas por este carácter se evalúan en orden hasta que una de ellas tenga éxito
- token?: el carácter '?' significa opcionalidad, se consumirá el token si existe, pero en caso contrario la regla continua
- &token: el caracter '&' significa lookahead, se asegura de que el siguiente token sea ese sin consumirlo.
- !token: el carácter '!' significa negative lookahead, se asegura de que el siguiente token no sea ese, y tampoco lo consume. Es el contrario que la regla anterior.
- token+: el carácter '+' significa repetición del token 1 o más veces
- token*: el carácter '*' significa repetición del token 0 o más veces
- ∼ `r"regex"ilmsux`:el carácter '∼' sirve para utilizar expresiones regulares, el texto entrecomillado que siga a este carácter se considerará expresión regular, al igual que con el texto literal podemos utilizar el carácter 'r' para tratar el texto de la expresión regular como crudo sin tener que escapar caracteres. Además, después de la expresión regular podemos utilizar algunos caracteres para facilitarnos el trabajo.
  - i: ingore_case, trata igual a mayúsculas y minúsculas
  - l: locale, hace los caracteres { `\w,\W,\b,\B`} dependientes de la configuración local
  - m: multiline, los caracteres `^` y `$` corresponden al principio y final de cada linea en lugar del string.
  - s: el carácter '.' también sustituye al carácter de nueva línea.
  - u: unicode, hace los caracteres { `\w,\W,\b,\B`} dependientes de la codificación de caracteres
  - x: verbose, permite insertar comentarios dentro de la expresión regular seguidos del carácter `#` [18]

5.5. Patrón Visitor

El patrón Visitor permite añadir operaciones a una estructura de objetos sin modificar el código de dichos objetos, separando los datos de sus métodos.

No parece buena idea separar los datos de sus métodos en un objeto, ya que estamos perdiendo cohesión y agregamos dependencias. Sin embargo existen situaciones en las que mantener ambos en el mismo objeto nos ocasiona problemas. [17]

5.5.1. Cuando usar el Patrón Visitor

En el ejemplo de la figura 5.2 podemos ver una estructura de objetos, tenemos una clase Figura y varias figuras concretas que heredan de la clase Figura.

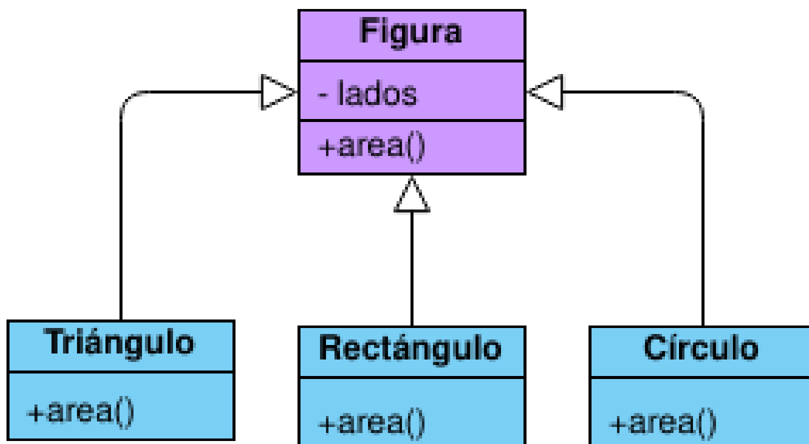

Figura 5.2: Problema para el Patrón Visitor

En este ejemplo podemos suponer que el número de figuras no va a cambiar a lo largo del tiempo, no vamos a necesitar más figuras. Sin embargo es probable que agreguemos operaciones nuevas a las figuras.

Si pensamos en la evolución del código, cada vez que queramos añadir una operación nueva, tendremos que modificar cada una de las clases de nuestra estructura de objetos, además, las líneas de código de cada objeto seguirá creciendo, ocasionando clases gigantes y haciendo el código difícil de mantener.

Por estos motivos vamos a separar las operaciones de las figuras de sus atributos con el patrón Visitor como se muestra en la Figura 5.3

Como se puede apreciar en la Figura 5.3, ahora cada vez que queramos agregar una nueva operación sobre las figuras solo tenemos que crear una nueva clase heredando de Visitor e implementar la visita a cada figura, de esta manera, solo creamos una clase en lugar de modificar cada figura, y todas las clases tendrán un tamaño aceptable.

Sin embargo, si vamos a querer añadir nuevas figuras más adelante este patrón no es aconsejable, ya que para añadir una nueva figura tendríamos que modificar cada una de las clases herederas de Visitor. Y si no vamos a agregar nuevas operaciones a las figuras es posible que tampoco queramos aplicar este patrón, ya que agregamos complejidad al código, reducimos la cohesión y aumentamos la dependencia.

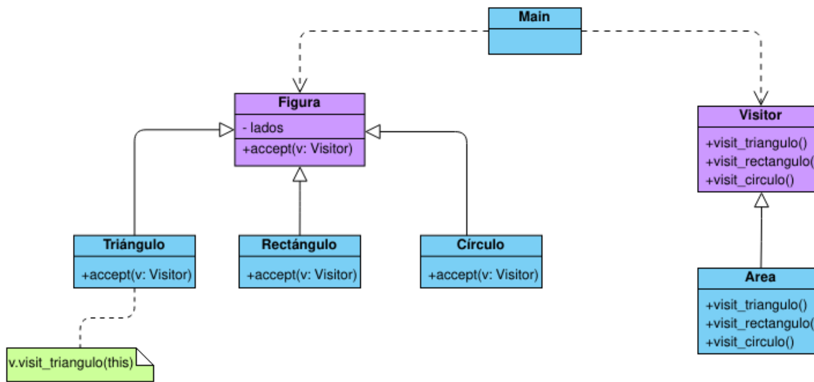

Figura 5.3: Solución con el Patrón Visitor

5.5.2. Patrón Visitor en Parsimonious

Como hemos podido ver anteriormente, mediante este patrón es posible añadir funcionalidades a Parsimonious sin modificar el código del mismo, simplemente hay que crear una clase heredera de NodeVisitor y crear en ella los métodos que Parsimonious visitará.

En el ejemplo anterior, para añadir la operación del Área, creamos el Visitor correspondiente, y definimos una implementación de visit por cada tipo de objeto. Sin embargo, en Parsimonious queremos agregar un visit por cada regla de la gramática, es decir, no queremos distinguir entre diferentes tipos de objetos, si no entre diferentes instancias de un mismo objeto.

Para que Parsimonious visite un método de la clase heredera hay que nombrarlo como ‘visit_¡nombre¿‘ donde nombre es una regla definida en nuestra gramática.

Por ejemplo, si en la gramática definimos que es nombre y que es DNI:

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

Podemos crear la lógica creando una clase

```
class miLogica(NodeVisitor):
    def visit_nombre(self, nodo, hijos):
        # Logica de nombre
    def visit_DNI(self, nodo, hijos):
        # Logica de DNI
    def generic_visit(self, nodo, hijos):
        # Logica de cualquier otro elemento
```

Cuando parseemos un texto Parsimonious visitará la función visit_nombre y visit_DNI cuando encuentre el texto que satasfaga las definiciones. En caso de que definamos más cosas que no necesitemos tratar de una manera especial podemos usar generic_visit.

Estos métodos reciben como parámetro el nodo en el que se encuentran y sus hijos, en el caso del DNI, hijos sería una lista de 2 elementos, por una parte los números y por otra parte la letra.

Sin embargo si difinimos el DNI como:

```
DNI = ~"[0-9]+[A-Z]"IU
```

DNI ya no está definido con 2 tokens sino con uno solo. Al ser uno solo no tiene hijos, el dni lo encontramos directamente en el nodo, podemos recuperarlo consultando el atributo text del nodo.

Para este ejemplo no usamos la potencia de un parser, podríamos implementarlo directamente con expresiones regulares. Para aprovechar mejor las capacidades de un parser podemos definir un elemento en base a otros elementos, como por ejemplo:

```
string_grammar = """
    expresiones = expresion+
    expresion = nombre "(" numeros ")" "\n"?
    nombre = "sumar" / "restar"
    numeros = numero ("," numero)*
    numero = ~"[0-9]+"
"""
```

Cuando visitemos una expresión tendremos que visitar el nombre y los números antes de terminar la visita de la expresión, así pues, podemos cambiar los tokens 'al vuelo'

Por ejemplo, utilizando esta gramática podemos implementar el Visitor de la siguiente manera

```
    class ExpresionVisitor(NodeVisitor):
    def visit_expresiones(self, nodo, hijos):
        return "\n".join(hijos)
    def visit_expresion(self, nodo, hijos):
        nombre = hijos[0]
        numeros = hijos[2]
        res = 0
        if nombre == 'sumar': res = sum(numeros)
        elif nombre == 'contar': res = len(numeros)
        return f"La expresion es {nombre} y el resultado es {res}"
    def visit_numeros(self, nodo, hijos):
        lista = [int(hijos[0])]
        if len(hijos[1]) > 1:
            lista += [int(i) for i in hijos[1][1:].split(',')]
        return lista
    def generic_visit(self, nodo, hijos):
        return nodo.text
```

Con este pequeño ejemplo podemos ver la potencia que tiene Parsimonious, si lo vemos detalladamente:

- numero y nombre están definidas con un solo token y no necesitamos cambiar nada en ellos, por lo que no definimos un método para ellos y visitarán generic_visit, el cual devuelve directamente el token por el que están formados.
- números será una secuencia de números separados por comas, sin embargo para procesarlo más cómodamente en visit_numeros los podemos convertir en una lista de enteros.
- expresión se define como combinación de tokens que ya hemos definido, nombre, paréntesis, números y un salto de línea. El hijo numero 2 corresponde a los números, que ya han sido parseados y convertidos a una lista de enteros, por lo que podemos tratar directamente con esta lista y lo convertimos a un string personalizado.
- expresiones simplemente es una secuencia de expresiones, en visit_expresiones simplemente unimos los strings personalizados que creamos en expresión separados por un salto de linea.

Podemos probarlo con las siguientes líneas:

```
gramatica = parsimonious.Grammar(string_grammar)
tree = gramatica.parse("""sumar(1,2,3,4)
contar(1,2,3,4)
contar(1,2)
sumar(22)""")
v = ExpresionVisitor()
print(v.visit(tree))
```

Las cuales dan como resultado un objeto del tipo que devuelva la regla más general, en este caso es la regla expresiones, la cual devuelve el siguiente obejeto de tipo string

```
La expresion es sumar y el resultado es 10
La expresion es contar y el resultado es 4
La expresion es contar y el resultado es 2
La expresion es sumar y el resultado es 22
```

En PySD se necesita más información que la que podemos obtener de esta manera, así que se define primeramente un __init__ que inicializa datos relevantes del parseo.

Por ejemplo

```
def __init__(self, ast):
            self.translation = ""
            self.kind = 'constant'
            self.new_structure = []
            self.arguments = None
            self.in_oper = None
            self.args = []
            self.visit(ast)
```

de esta manera se puede, por ejemplo, añadir contenido a las listas args y new_structure desde cualquier función. Y al terminar la ejecución de visit obtendremos como resultado un objeto con los atributos que hemos definido en el `__init__` en lugar de el objeto que retorna la regla más general

Capítulo 6

PySD

6.1. Qué es PySD

Python System Dynamics (PySD) es un proyecto open source alojado en github iniciado por James Houghton y Michael Siegel.

Este proyecto se creó en 2013 con el ánimo de crear una librería en Python capaz de traducir a código Python el resultado de construir un modelo de dinámica de sistemas y ser capaz de ejecutar el mismo.

La filosofía de este proyecto reside en enfocar el esfuerzo en llevar los modelos que se realizan en los entornos de modelado de dinámica de sistemas a un entorno en el que sea posible y cómodo analizar los resultados y llevar a cabo estudios de los mismos en cualquier ámbito del data science.

La motivación de traducir estos modelos a Python (o a R con PySD2R) es poder usar todas las herramientas que se crean en estos lenguajes para el tratamiento de datos en lugar de que cada herramienta de modelado replique estas herramientas.

PySD da soporte a los lenguajes Vensim y XMILE, sin embargo la librería aún no es capaz de traducir correctamente modelos complejos.

6.2. Instalación de PySD

Es posible instalar PySD directamente desde pip [1], el instalador de paquetes de python.

PySD se desarrolló para la versión 2 de python, sin embargo se adaptó para funcionar también en la actual versión 3.

para instalar PySD desde pip solamente hace falta ejecutar

```
pip install pysd
```

También se puede instalar PySD desde el código fuente, que suele estar más actualizado.

Primeramente hay que escoger la versión de PySD que se quiere instalar de entre las releases disponibles en https://github.com/JamesPHoughton/pysd/releases

O clonar el repositorio para instalar la última versión

```
git clone https://github.com/JamesPHoughton/pysd.git
```

Para instalar cualquiera de estas versiones hay que instalar las dependencias del proyecto, las cuales son:

- pandas
- parsimonious
- xarray
- yapf
- lxml
- funcsigs
- pydoe
- xlrd
- xlwt
- regex
- chardet

Podemos instalar PySD y sus dependencias ejecutando

```
python setup.py
```

Para poder trabajar con varias versiones de pysd he trabajado con la herramienta virtualenv [2], creando 3 entornos virtuales en los que instalar cada una de las versiones mencionadas en la sección 4.2

6.3. Entornos Virtuales

Para no tener conflictos entre versiones de PySD ni con las bibliotecas de mi sistema operativo decidí trabajar con entornos virtuales.

Se crearon los siguientes entornos virtuales con la herramienta virtualenv [2]:

- PySD_original entorno virtual con Python3.6 y las dependencias instaladas con el setup.py oficial de PySD
- PySD_fork entorno virtual con Python3.6 y las dependencias instaladas con el setup.py del fork de Roger de PySD

La finalidad de tener estas dos versiones de pysd era comparar sus resultados.

Tras el cambio de objetivo del TFG se creó otro entorno virtual: `PySD_julien` con la versión de julenmalard de la que se ha basado este trabajo.

El comando para crear entorno virtual es:

```
virtualenv -p /usr/bin/python3 <nombre>
```

Para activar los entornos virtuales he utilizado VSCode, el cual permite escoger el intérprete de python de manera gráfica.

Para usar los entornos virtuales sin VSCode hay que activarlo al momento de usarlo con

```
source venv/<entorno_vritual>/bin/activate
```

donde entorno_virtual es el nombre que se ha puesto al entorno que queremos actiar, por ejemplo PySD_original

Al finalizar ejecutamos

```
deactivate
```

Para dejar de utilizar ese entorno virtual.

6.4. Estructura de PySD

La estructura de PySD se representa en la Figura 6.1. El cliente interactúa con la fachada pysd, la cual se encarga de interactuar con el subsistema py_backend.

El subsistema py_backend se puede dividir conceptualmente en 2 partes representadas en la figura 6.1:

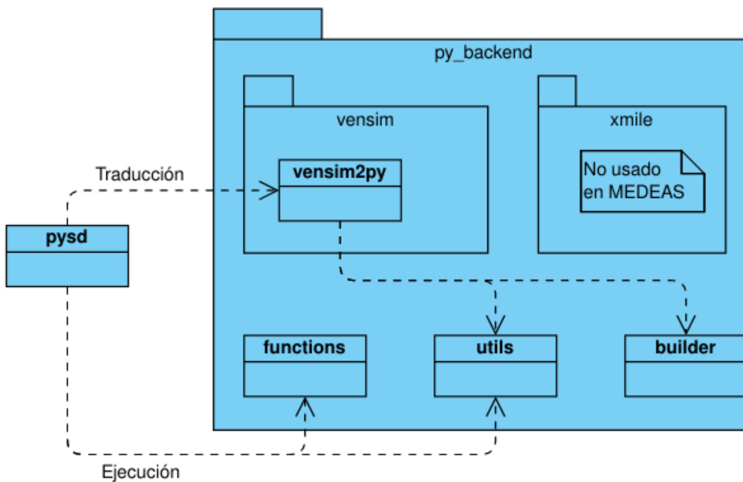

Figura 6.1: Estructura de PySD

- Traducción
- Ejecución

Dentro de la división de traducción se puede dividir a su vez en tras 2 partes:

- Vensim
- XMILE

Sin embargo, en este TFG solo abordamos la traducción de Vensim. Una vez completada la traducción se genera un archivo Python independiente del lenguaje original, por lo que la división de ejecución no se subdivide en más partes.

6.5. Traducción de Vensim

La traducción de Vensim es compleja porque no se ha implementado un parser de manera tradicional.

Anteriormente, se ha expuesto el funcionamiento de un parser y cómo se utiliza en concreto el parser PEG Parsimonious, sin embargo, en PySD no se ha creado una gramática con todas las reglas pertenecientes a Vensim con su clase Visitor correspondiente, sino que se han creado un total de 5 gramáticas diferentes, cada una con un nivel de abstracción distinto.

Esta implementación es algo confusa y causa problemas al tener la información repartida por diferentes partes del código, se tiende a escribir código duplicado como por ejemplo, la definición de caracteres separadores como el espacio y el tabulador que se encuentran duplicadas:

```
_ = ~r"[\s\\]*"
```

Para intentar evitar el código duplicado se ha creado una función **include_common_grammar** que recibe un parámetro como argumento. La finalidad de esta función es ofrecer una serie de reglas generales que se anexarán a las reglas que reciba como argumento. Esto arregla en parte el código duplicado pero tan solo evita duplicar el código en la gramática.

Por ejemplo, la regla para los caracteres separadores que tomamos en el ejemplo anterior tiene 3 implementaciones en distintas clases Visitor.

```
def visit__(self, n, vc):
    return ' '

def visit__(self, n, vc):
    return ''

def visit__(self, n, vc):
    return ''
```

Incluso estas implementaciones no son iguales, en 2 de ellas se retorna una cadena vacía y en otra se retorna un espacio.

Esto hace más difícil trabajar con la gramática y obliga a tener que estudiarla detenidamente para añadir reglas o modificar las existentes.

Veamos para que se utiliza cada una de estas gramáticas:

6.5.1. file_structure_grammar

En Vensim existen las MACROS, una MACRO está delimitada, comienza por :MACRO: y termina con :END OF MACRO:, recibe unos argumentos y produce unas salidas para ellos.

Esta gramática simplemente separa cada MACRO y el código principal. Devuelve una lista de diccionarios, en la cual cada diccionario representa una MACRO o el código principal, tiene la siguiente estructura:

```
    {
        'name': _main_ o nombre de macro
        'params': [lista con parametros]
        'returns': [lista con valores a devolver]
        'string': codigo vensim
    }
```

Como se puede ver, el código principal se etiqueta con el nombre _main_, este además tendrá la lista de params y returs vacía y el código se encuentra en strings.

6.5.2. model_structure_grammar

Esta gramática parsea el código principal de un fichero Vensim, su finalidad es encontrar todas las sentencias del código y organizar su contenido de la siguiente manera:

```
{
    'doc': documentacion
    'unit': unidades
    'eqn': ecuacion
    'lims': limites de las unidades
    'kind': entry | section
}
```

Como se ha explicado anteriormente, las sentencias de Vensim tienen 3 partes:

- Ecuación
- Unidades
- Documentación

Esta gramática separa cada una de esas partes y de la sección de unidades saca los límites de las unidades, ya que opcionalmente se pueden agregar intervalos en los que esa variable puede tomar valor. Por ejemplo, la temperatura del agua líquida en grados centígrados varia entre 0 y 100.

Además agrega el kind entry si es una ecuación o section en caso de que sea una sección de documentación. Estas últimas no afectan en la ejecución, simplemente son parte de la documentación para los programadores de ese modelo, por lo que una section solo tiene el campo doc, el resto permanecen vacíos.

Por ejemplo, puede parsear el código:

```
un ejemplo[sub1, sub2] =
    1
    ~ aqui unidades [0, 100]
    ~ aqui documentacion
    |
```

y obtendríamos como resultado:

```
[
    {
        'doc': 'aqui documentacion'
        'unit': 'aqui unidades'
        'eqn': 'un ejemplo[sub1, sub2] = 1'
        'lims': '[0, 100]'
        'kind': 'entry'
    }
]
```

Notese que el resultado es el diccionario dentro de una lista, ya que en un ejemplo real se parsea todo el documento a la vez y el resultado es una lista con el diccionario de cada sentencia.

6.5.3. component_structure_grammar

Esta gramática parsea una ecuación, es decir, una sentencia sin las unidades ni la documentación. Su objetivo es dividirla en su parte izquierda:'real_name' y 'subs', y parte derecha: 'expr' para generar un diccionario con la siguiente estructura:

```
{
    'real_name': nombre,
    'subs': subscripts,
    'expr': parte derecha,
    'kind': component | lookup | subdef
}
```

Por ejemplo, continuando con el ejemplo anterior, si esta gramática recibe:

```
un ejemplo[sub1, sub2] = 1
```

generará el resultado:

```
{
    'real_name': 'un ejemplo',
    'subs': ['sub1', 'sub2'],
    'expr': '1',
    'kind': 'component'
}
```

Nótese que ahora se otrorga un 'kind' de component, mientras que anteriormente su 'kind' fue 'entry'. Esto es debido a que 'kind' sirve para conducir cada sentencia por diferentes caminos durante la ejecución.

Esta gramática puede catalogar a las ecuaciones con 3 valores de 'kind':

- **component:** A una expresión o constante
- **lookup:** A un lookup de cualquier tipo
- **subdef:** A una definición de subscript

En el fork del usuario julienmalard se añadió otro elemento al diccionario: 'keyword' y kind puede tener otro valor más: 'data'

Este valor de 'kind' añadido se utiliza para identificar definiciones de datos que no utilizaremos. Sin embargo, la variable 'keyword' sí es importante para la lectura de ficheros excel, ya que en ocasiones se especifica que se desean interpolar los datos con la palabra clave INTERPOLATE.

6.5.4. expression_grammar

Esta gramática es la más grande, parsea la parte derecha de la ecuación y se encarga de generar una traducción equivalente en código Python.

En esta gramática se hace notable el uso de gramáticas separadas ya que hay que recurrir a variables fuera de la clase Visitor para completar la información, como por ejemplo la parte izquierda o si tiene subscripts.

Devuelve un diccionario con:

- la traducción
- el kind que puede ser component o constant
- una lista de estructuras que explicaremos más adelante

De nuevo se vuelve a etiquetar con diferentes valores de 'kind', en esta ocasión servirán para etiquetar a la traducción con un nivel de caché diferente durante la ejecución del modelo. La caché se explicará más adelante en la sección (6.6.4)

6.5.5. lookup_grammar

Esta gramática, al igual que la anterior, también parsea la parte derecha de la ecuación. La diferencia radica en que esta gramática solo se usa con las sentencias de tipo lookup.

El motivo de separar expression_grammar y lookup_grammar si ambas se dedican a parsear la parte derecha de la ecuación es que tienen muchos tokens parecidos para definir las reglas y habrá que diseñar con mucho cuidado la gramática para que funcionase correctamente.

Al igual que expression_grammar la finalidad de esta gramática es generar una traducción a Python y crear las nuevas estructuras.

6.5.6. Funcionamiento

Sabiendo para que sirve cada gramática es posible comprender el funcionamiento del traductor.

Para poder traducir un fichero Vensim el primer paso es leer dicho fichero, seguidamente creará un archivo Python por cada MACRO y finalmente otro archivo para el código principal.

Para ello primeramente utiliza **file_structure_grammar** para descomponer el código Vensim en las distintas MACROS y el código principal.

Para traducir cada porción de código utiliza **model_structure_grammar**, la cual extrae todas las sentencias y extrae de cada una de ellas la ecuación. Cada ecuación pasará por la gramática **component_structure_grammar**, la cual ya separa el nombre, los subscripts y la parte derecha.

En este momento, ya sabemos el tipo de sentencia que estamos parseando. Si es una definición de subscript se guardará en la variable subscript_dict.

También en este momento el nombre se transforma en un nombre seguro, es decir, se eliminan todos los caracteres que no puede contener una variable en Python. Sin embargo, seguimos necesitando guardar el nombre original, por lo que se almacena en el diccionario namespace.

A continuación se comprueba el kind de la sentencia y ello decidirá si se utiliza **expresion_grammar** o **lookup_grammar**, los cuales generarán su traducción correspondiente y pueden devolver nuevas estructuras. Estas nuevas estructuras son elementos que se añadirán a la traducción, ya que una sentencia Vensim puede equivaler a más de una sentencia Python.

Para finalizar, builder colocará todas las traducciones que se han generado en el parser ordenadamente en una plantilla y lo escribirá en el archivo de salida.

Ejemplo

Para aclarar el funcionamiento de la fase de traducción se puede estudiar el comportamiento del traductor con el ejemplo de la tetera expuesto en la sección 3.2.

Este ejemplo no utiliza MACRO, por lo que **file_structure_grammar** devuelve una lista con un solo diccionario correspondiente al código principal, por lo que 'name' tendrá el valor `'_main_'`

Se pasa el código principal, que en este caso es todo el código, a **model_structure_grammar**, la cual obtiene 8 sentencias del mismo, las cuales corresponden a:

- Los 4 parámetros de simulación en los que se especifica el tiempo de inicio y final así como el intervalo de integración y la frecuencia con la que se quieren mostrar resultados.

- Las 4 ecuaciones que definen la ley de enfriamiento de Newton

Como muestra, uno de los diccionarios que representan las sentencias es:

```
{
    'eqn': 'Teacup Temperature= INTEG (\t-Heat Loss to Room
        ↪ ,\t\t180)',
    'unit': 'Degrees Fahrenheit',
    'lims': '(32.0, 212.0)',
    'doc': ...,
    'kind': 'entry'
}
```

La ecuación (eqn) de cada una de estas sentencias será procesada por **component_structure_gramma** separando sus partes. Por ejemplo, continuando con el ejemplo de la sentencia *Teacup Temperature*, el resultado es:

```
{
    'real_name': 'Teacup Temperature',
    'subs': [],
    'expr': 'INTEG ( -Heat Loss to Room, 180)',
    'kind': 'component',
    'keyword': None
}
```

Ahora que se ha catalogado como 'component' su parte derecha (expr) se procesa con **expression_grammar** dando como resultado su traducción y, en caso de ser necesario, nuevas estructuras.

En el ejemplo se genera la traducción:

```
_integ_teacup_temperature()
```

Esta traducción se basa en llamar a una nueva estructura, ya que todos los Stocks generan un objeto de tipo functions.Integ

La nueva estructura creada es:

```
{
    'py_name': '_integ_teacup_temperature',
    'py_expr': 'functions.Integ(lambda: -heat_loss_to_room(),
        ↪ lambda: 180)',
    'kind': 'stateful',
    'arguments': ''
}
```

esto se construirá en el builder como:

```
@cache('step')
def teacup_temperature():
    return _integ_teacup_temperature()

_integ_teacup_temperature = functions.Integ(lambda: -
   ↪ heat_loss_to_room(), lambda: 180)
```

Nótese la diferencia entre este Stock, que se representa como el emento con estado functions.Integ, y cualquiera de las otras 7 sentencias, que no mantienen un estado y se traducen simplemente como:

```
@cache('run')
def characteristic_time():
    return 10

@cache('step')
def heat_loss_to_room():
    return (teacup_temperature() - room_temperature()) /
       ↪ characteristic_time()
```

Al finalizar la traducción, se habrán completado 2 diccionarios:

- **namespace:** Este diccionario tiene la equivalencia entre los nombres de las variables en Vensim y su equivalencia en Python. En el ejemplo de la tetera, el namespace generado es:

```
        _namespace = {
        'TIME': 'time',
        'Time': 'time',
        'Characteristic Time': 'characteristic_time',
        'Heat Loss to Room': 'heat_loss_to_room',
        'Room Temperature': 'room_temperature',
        'Teacup Temperature': 'teacup_temperature',
        'FINAL TIME': 'final_time',
        'INITIAL TIME': 'initial_time',
        'SAVEPER': 'saveper',
        'TIME STEP': 'time_step'
    }
```

- **subscript_dict:** Es un diccionario con cada subscript como clave y sus subscript values como valores. Por ejemplo, si se define el subscript:

```
        paises: italia, francia, alemania ~~|
```

  Al procesar **component_structure_grammar** esta sentencia la etiquetara con el kind 'subdef', por lo que en lugar de seguir siendo procesada por otra gramática se agrega a subscrit_dict, dando como resultado:

```
_subsript_dict = {"paises": ['italia', 'francia', 'alemania
    ↪ ']}
```

6.6. Ejecución

En la ejecución hay 2 archivos fundamentales:

- El archivo que se genera durante la traducción, donde se encuentran todas las variables, relaciones y ecuaciones.
- El archivo functions.py, en el cual están implementadas las clases que se instanciarán en el archivo anteriormente mencionado y la lógica de simulación.

El diagrama de clases de functions.py se muestra en la Figura 6.2

En la Figura 6.3 se muestra un diagrama de clases muy simplificado con los aspectos más importantes que vamos a tratar.

La clase **Time** representa el tiempo durante la simulación.

La clase **Stateful** tiene un atributo state y se implementa un getter y un setter para el mismo, el resto de clases, a excepción de Time, serán herederos de esta clase para tener un estado, ya que en dinámica de sistemas nos interesa la evolución de estado de ciertos elementos. A las instancias de los objetos herederos de esta clase los denominaremos **Sateful elements**, ya que son elementos con estado.

La clase **Integ** representa los Stocks, por lo que recibe como parámetros en su constructor su valor inicial y la función que obtiene la derivada necesaria para llevar a cabo la integración.

La clase **Macro** implementa la lógica para representar las MACROS, lo cual implica obtener los objetos de tipo Stateful de un fichero previamente generado en la etapa de traducción, inicializarlos a sus valores de inicio y obtener sus derivadas para poder llevar a cabo la simulación.

La clase **Model** representa al código principal del modelo. Realiza las mismas laboras que Macro, por lo tanto hereda de esa clase. Además al ser la clase principal es la encargada de instanciar el tiempo y llevar a cabo la simulación mediante la integración por Euler.

El resto de clases, como **Delay**, representan diversas herramientas que se pueden utilizar en Vensim para crear el modelo, tienen diferentes parámetros para inicializar su estado y la fórmula para calcular su derivada se ha obtenido mediante demostraciones matemáticas y se calcula sin necesidad de que el propio PySD derive.

Por lo tanto, la clase **Model** es la principal y la que llevará a cabo la simulación en 3 etapas bien definidas.

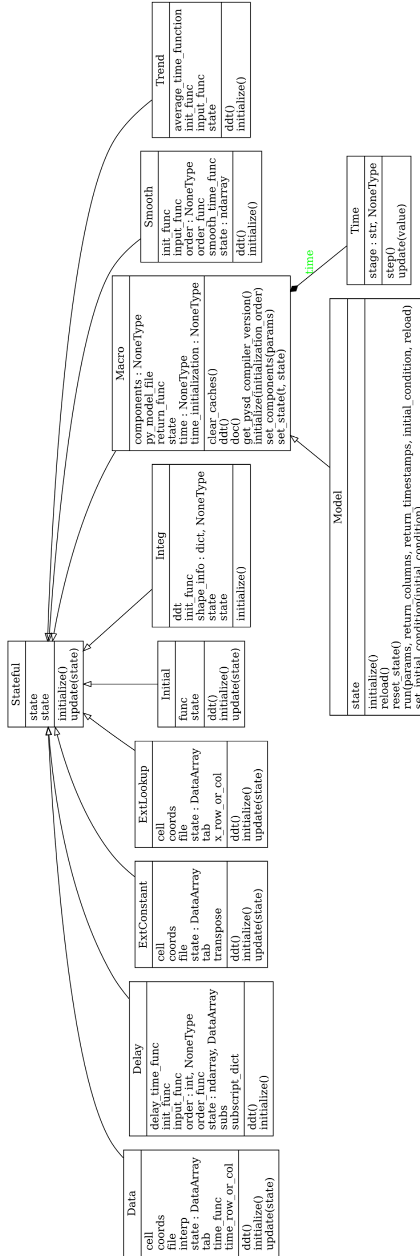

Figura 6.2: Diagrama de clases de ejecución

6.6.1. Load

Durante la etapa Load se lee el fichero Python previamente traducido para importar todas sus funciones mediante la libreria imp, la cual carga todo el fichero en la variable components.

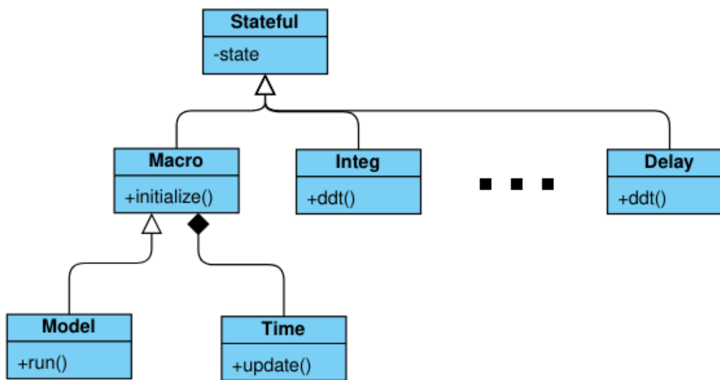

Figura 6.3: Diagrama de clases simplificado

Seguidamente se filtran los objetos de tipo Stateful y se guardan en stateful_elements. Con esto damos por finalizada la etapa Load y tenemos una lista de objetos herederos de Stateful, es decir, los stocks como objetos Integ, los objetos de lectura excel y demás herramientas.

6.6.2. Initialization

Esta etapa se lleva a cabo en dos tiempos diferentes:

Primeramente, al instanciar el objeto Model, se asigna el primer valor del tiempo de la simulación y se intentan asignar los valores por defecto a todos los objetos Stateful teniendo en cuenta las restricciones de dependencias, ya que algunos elementos dependen de otros, por lo que se instancian en orden topológico [48], de forma que todos los Stateful elements residen en una cola para ser inicializados, en caso de no poder ser inicializado un elemento se vuelve a insertar en la cola para evitar errores al inicializar sus valores. La implementación de este proceso es iterativa debido a que el algoritmo de ordenación topológica recursivo podría dar errores de memoria al traducir un fichero muy grande.

Esta inicialización deja el modelo en el estado determinado en la programación Vensim del Modelo, es decir, tanto el estado inicial de cada variable como el periodo en el que se va a desarrollar la simulación están predefinidos por el modelado Vensim.

Sin embargo se puede desear cambiar el tiempo de inicio o fin de la simulación, los valores iniciales para las variables, especificar de qué variables en concreto se quiere obtener su evolución o en que timestamps se quiere comprobar su estado. PySD permite pasar este tipo de argumentos antes de comenzar la simulación, si estas opciones no existieran sería necesario cambiar el modelo Vensim y volver a traducirlo entero incluso para simular el mismo modelo en un intervalo de tiempo diferente.

En este punto, cuando ya está el modelo inicializado, se comprueba si se ha llamado la

ejecución con algún parámetro. En dicho caso se cambian los valores a los Stateful elements correspondientes.

Como PySD realiza la simulación mediante la integración por Euler, el siguiente paso es calcular los puntos del tiempo en los que se va a parar para calcular cada estado, para calcular estos tiempos se tienen en cuenta 2 cosas:

- El espacio de tiempo dt que se quiere utilizar para integrar por el método de Euler.
- Los tiempos en los que el usuario desea que se le notifique el estado del modelo.

Para llevar a cabo la simulación sería suficiente con generar una lista de tiempos desde el tiempo de inicio hasta el tiempo de fin separados por el intervalo de tiempo especificado para integrar por Euler, sin embargo agregar más puntos no perjudica la simulación, al contrario, la hace más precisa, por lo que se agregan los puntos en los que el usuario quiere parar para ver el estado.

Por lo tanto hay que tener en cuenta que agregar puntos de tiempo en los que comprobar el estado del modelo puede afectar al resultado de los estados.

6.6.3. Run

En esta etapa tiene lugar la simulación, la cual se lleva a cabo mediante la integración por Euler.

Este método de integración, como hemos explicado anteriormente, es iterativo y calcula el siguiente estado del modelo partiendo del estado actual, y para que sea preciso debe calcularse el estado en intervalos lo más pequeños posibles de tiempo.

En PySD la lista de tiempos en los que se calcula Euler se ha calculado durante la etapa de inicialización, por lo que en esta etapa simplemente hay que ejecutar la integración mientras se recorre esta lista de timestamps.

Para llevar a cabo la integración el modelo ejecuta:

```
self.state = self.state + self.ddt() * dt
```

Lo cual equivale al método de integración por Euler explicado en la sección 3.4:

$$f(x_i) = f_(x_{i-1}) + h * f'(x_{i-1})$$

Donde state es el estado actual, almacenado en un ndarray de la librería numpy, dt es el intervalo de tiempo para la integración y ddt es una función encargada de llamar al método ddt de cada Stateful element y almacenarlo en otro ndarray.

Cada Stateful element tiene definido su propio ddt, esta función devuelve la derivada de la función que queremos integrar. La ventaja de utilizar la estructura de datos ndarray es que podemos sumar dos ndarrays y se sumarán elemento a elemento, es decir el elemento iésimo del primer array se sumará con el el elemento iésimo del segundo, y también podemos multiplicar un ndarray por un solo número y se multiplicará cada elemento del ndarray por ese número.

Como esta lista ha sido creada en base al intervalo de tiempo definido por Euler y los timestamps en los que se desea consultar el estado del modelo, en cada iteración se comprueba si en ese timestamp se quiere obtener el estado del modelo, para que en caso de ser así, se almacenen todos los Stateful elements seleccionados en una lista.

En el ejemplo de la tetera expuesto en la sección 3.2, al ejecutar el modelo traducido con un intervalo de integración de 0.125 se calcula el estado de la temperatura de la tetera de esta manera:

En el instante inicial, la tetera tiene una temperatura de 180, por lo que self.state en el tiempo 0 equivale al ndarray [180]. self.ddt() calcula el valor de la derivada en este instante, el cual devuelve el ndarry [-11], lo que indica que bajarían 11 grados en un minuto, pero como nuestro intervalo de integración es de tan solo 0.125 minutos se multiplica el resultado de la derivada por este valor, y nos indica que en este tiempo la tetera baja 1,375º de temperatura.

Para calcular la temperatura en el instante 0.3 se utilizará como base el estado de la tetera en el instante 0.125 que se acaba de calcular.

Al finalizar la simulación se crea un pandas.DataFrame con los valores de estado de los Stateful elements que se han capturado durante la simulación.

Al convertirlo a pandas.DataFrame se obtienen funcionalidades de la librería Pandas, como la representación gráfica de la evolución del modelo.

6.6.4. Caché

PySD implementa una caché a 2 niveles para ahorrar el máximo tiempo posible durante la ejecución, esta caché está implementada mediante el uso de decoradores.

Los decoradores son un patrón de diseño que permite alterar una función de manera dinámica sin necesidad de cambiar el código de esta ni crear subclases.

En Python los decoradores son etiquetas que se escriben encima de la definición de una función y empiezan con el carácter @ y permiten añadir funcionalidad al principio y/o al final de la ejecución de la función que se decora.

En PySD la funcionalidad de caché se implementa con un decorador que recibe un parámetro y devuelve una función decoradora dependiendo de este.

En fase de traducción se etiqueta a cada función traducida con um decorador llamado 'cache', el cual recibe como parámetro el string 'step' o 'run'.

Todas las funciones etiquetadas con la cache ’run’ son constantes, y por lo tanto se calcula su valor una sola vez durante la ejecución.

Las funciones etiquetadas con la caché ’step’ son funciones variables que pueden tener diferentes valores a lo largo del tiempo de la simulación, sin embargo, para calcular el siguiente estado del modelo es posible que se necesite consultar el valor de una misma función en repetidas ocasiones, por lo tanto, se guarda su valor en la primera ejecución de cada estado para no volverlo a calcular hasta el siguiente estado.

Estas etiquetas se añaden en el periodo de traducción a todas las funciones que se generan.

Capítulo 7

Aportación a PySD

La versión oficial de PySD, o la versión utilizada para traducir las anteriores versiones de MEDEAS están lejos de poder traducir MEDEAS automáticamente.

La versión activa más cercana a este objetivo es el fork de Julien Malard, concretamente en su rama new_funcions. En esta rama se ha añadido soporte para subscripts básicos y para llamadas básicas a Excel, estas aportaciones no están perfectamente terminadas pero se han añadido reglas a las gramáticas para soportarlas, funciones en los correspondientes Visitor para manejarlas, y clases Stateful para que puedan ser manejadas durante la ejecución.

Aun así, MEDEAS necesita más cambios para poder ser traducido.

Los principales problemas para traducir MEDEAS que no cubre ninguna versión actual son:

- Nombre de variables con combinaciones de caracteres problemáticas
- Lectura de un fichero Excel muy grande
- Disposición de datos en el fichero Excel problemáticas
- Uso complejo de subscripts
- Múltiples espaciados y saltos de línea en definiciones
- Definición de lookups de varias maneras
- Soperte para la sentencia GET DIRECT CONSTANTS

7.1. Variables

Los identificadores de variables entrecomillados pueden contener multitud de caracteres que pueden ser malinterpretadas durante la traducción.

Es el caso de variables como:

- **" Electronic/Components ":** El carácter / se malinterpreta por una división
- **"deoinized, water":** El carácter ',' se malinterpreta en un subscript como si fueran 2 subscripts diferentes, por una parte deoinized y por otra parte water.
- **"Total CO2e [GWP=100 years]":** Se malinterpretan los corchetes como si fuera un subscript.

Este problema se origina principalmente en expression_grammar, ya que al tener las gramáticas separadas hay que inyectar información obtenida en gramáticas anteriores y las comillas entraban en conflicto.

Para inyectar información, como por ejemplo los subscript values que se han definido, se utilizan strings formateados a la hora de crear la gramática de la siguiente forma:

```
datos_a_inyectar = [lista con elementos]
gramatica = parsimonious.Grammar("""
datos_inyectados = ~r"(%(datos)s)"
""" % {'datos': '|'.join(datos_a_inyectar)}))
```

Es decir, se sustituye la variable datos en el string por todos los datos de la lista datos_a_inyectar casteados a string y separados cada uno de ellos por el carácter `'|'`,que al ser tratado en la gramática como una expresión regular ese carácter significa 'or'.

El problema es que se inyectaban los subscript elements tal cual se leían en la gramática anterior, es decir, con las comillas, y al utilizarlo en esta gramática entraban en conflicto.

Se soluciona simplemente quitando las comillas antes de inyectarlos expression_grammar, que si bien es un fallo fácil de arreglar es bastante complejo de detectar.

El caso de "Total CO2e [GWP=100 years]" origina problemas incluso después de solucionar la inyección, pero no en traducción si no en la etapa de ejecución. El problema se origina al seleccionar esta variable para estudiar su evolución durante la simulación.

El código encargado de formatear las variables detecta si tiene subscripts tan solo comprobando si contiene el carácter `']'`.Por lo que se solucionó excluyendo el caso en el que la variable estuviese entrecomillada.

7.2. Fichero Excel grande

La implementación de lectura de ficheros Excel se realiza con la librería Pandas. En la etapa de Initialization de ejecución, cada vez que se encuentra una lectura a un archivo Excel, se carga el fichero en una variable gracias a la librería pandas, se leen las celdas

correspondientes del fichero, y se cierra el fichero. Esta implementación se hizo pensando en archivos pequeños con pocas lecturas a ficheros Excel pequeños, sin embargo, MEDEAS realiza miles de consultas a un fichero Excel grande, por lo que abrir y cerrar múltiples veces el fichero causa problemas de rendimiento.

Para soluciona este problema se ha implementado el patrón Singleton. Se ha creado la clase Excels, la cual, al implementar el patrón Singleton solo tendrá una instancia, y esta clase se utiliza para leer los ficheros Excels. Esta clase sigue dependiendo de la librería Pandas para la lectura de los ficheros Excel, pero guarda el archivo abierto en lugar de cerrarlo, ahorrando una gran cantidad de tiempo.

7.3. Lectura de Excel compleja

La implementación de lectura de datos de Excel en PySD es básica, por lo que al tratar con ficheros complejos como el Excel de MEDEAS surgen conflictos que se deben solucionar.

La sentencia GET XLS DATA en Vensim se utiliza en MEDEAS para leer series o matrices de datos con respecto al tiempo. Por ejemplo:

	A	B	C	D
1	2010	2011	2012	2013
2	100	120	100	110

Supongamos que esta tabla se representan en la fila 1 los años y en la fila 2 los datos asociados a cada año, de manera que en 2010 ese dato equivale a 100, en 2011 a 120, etc.

La sentencia para leer los datos en Vensim sería:

```
GET XLS DATA('hoja1', 1, 'A2')
```

En el primer argumento le indicamos la hoja en la que tiene que buscar dentro del Excel, en el segundo le indicamos la fila o columna en la que se encuentra la escala temporal, ya que la tabla se puede encontrar en horizontal o en vertical, y en el tercero le indicamos en que celda concreta se encuentra el primer dato.

Con estos argumentos tenemos que ser capaces de detectar hasta que punto tenemos que leer, lo cual, en un fichero sencillo como el que se expone de ejemplo, no es complicado, tan solo hay que leer hasta que se encuentre una celda vacía.

Los problemas surgen cuando:

- Una de las 2 filas es más corta
- Seguido a una de ambas filas se encuentran otros datos que no corresponden, por ejemplo, si al ejemplo anterior le añadimos la casilla E1 con valor ⹂years”
- Las columnas llegan más allá de la letra Z

Para solucionar estos problemas se utiliza la ayuda de la propia librería Pandas, la cual nos permite eliminar las celdas con valor nulo. Sin embargo, también se ha tenido que implementar lógica para descartar casillas no numéricas.

Además, al eliminar las casillas sobrantes a la izquierda debemos tener cuidado, ya que los índices de la estructura de datos pandas.Series se verán afectados.

También se ha añadido el soporte para las celdas con la columna de doble letra.

7.4. Subscripts

Solo esta implementado el un uso muy básico de subscripts en PySD, por ejemplo, dado el subscript:

```
paises:
    italia, francia, alemania
```

podríamos utilizar correctamente una variable con subscript de la siguiente forma:

```
variable[paises]
```

Pero no podríamos usar:

```
variable[italia]
```

Esto es debido a que se obtienen los subscripts entre corchetes en la gramática y más adelante, para construir la estructura de datos en Python, se obtiene el subscript que se llame de tal manera. Sin embargo, no existe ningún subscript que se llame italia, si no que italia es un subscript value del subscript paises.

Para solucionarlo, se busca también en los subscript values de cada subscript hasta encontrarlo. Con este problema podemos utilizar mejor los subscripts, pero seguimos teniendo problemas como el siguiente.

En Vensim no existe el concepto de matriz, por lo que en MEDEAS se definen subscripts duplicados. Por ejemplo, sectors y sectors1. Estos subscripts tienen los mismos subscript values, se utilizan para poder utilizarlos en la misma variable ya que no es sintácticamente correcto utilizar:

```
variable[paises, paises]
```

se crea el duplicado paises1 para poder hacer:

```
variable[paises, paises1]
```

El problema radica en que, si se utiliza:

```
variable[paises, italia]
```

No tenemos forma de distinguir si italia es un subscript value de sectors o sectors1.

Este problema no se ha solucionado de manera general, ya que para saber exactamente a que subscript nos referimos se requiere un análisis sintáctico demasiado complejo para PySD al haber dividido el análisis sintáctico en partes.

Para solucionarlo en MEDEAS todos los subscripts se definen en orden, primero el original, luego la copia. Esta solución implica que siempre que se encuentre un subscript value se va a asignar al subscript original, y todo funcionará correctamente debido a que la copia, en MEDEAS, solo se utiliza con el fin anteriormente mencionado de la matriz.

Otro problema con los subscripts son los subscripts definidos por partes.

Es una práctica muy utilizada en MEDEAS, consiste en definir una variable con subscript definiendo sus subscript values por separado. Por ejemplo, continuando con el ejemplo de paises:

```
variable[italia] = 1
variable[francia] = 2
variable[alemania] = 1
```

De esta forma se está definiendo 3 veces la variable, cada una con un subscript value diferente, con el comportamiento de PySD se iría sobre escribiendo, por lo tanto tan solo se traduciría la última de las 3 sentencias.

Para evitar esto, se crean 3 variables auxiliares empezando desde el número 1 hasta el 3, y se traduce como la fusión de estas variables.

Esto se puede llevar a cabo ya que los subscripts se traducen a Python como objetos de tipo xarray.DataArray.

Esta estructura de datos que permite almacenar objetos del mismo tipo, generalmente int64 o float64 en PySD, y los almacena mediante coordenadas.

Las coordenadas son diccionarios, que en PySD nos sirven para representar los subscripts, usando el nombre del subscript como clave del diccionario, y los subscript values se almacenan como una lista.

Esta estructura de datos es muy eficiente y muy útil para manejar las variables con varias subscripts. Por ejemplo, si creamos una variable jugadores, la cual tiene 2 subscripts, paises y deportes, esta variable representaría la cantidad de jugadores que hay de cada deporte en cada país, y se convertiría en un xarray.DataArray como el que vemos a continuación:

```
coords = {"paises": ['italia', 'francia', 'alemania'],
          "deportes": ['futbol', 'baloncesto', 'tenis']}

data = np.array([
    [1,2,3],
    [4,5,6],
    [7,8,9]
])

da = xr.DataArray(data=data, coords=coords, dims=('paises', '
    ↪ deportes'))
print(da.loc[{'deportes':'tenis', 'paises':['italia', 'alemania
    ↪ ']}])
```

En esta sección de código podemos ver cómo se almacenan los subscripts en un diccionario en la variable coords, con el nombre del subscript como clave y como valor una lista de subscript values.

A continuación se crea un ndarray con datos de ejemplo para crear el DataArray, y finalmente se crea el DataArray, el cual almacena la información con las coordenadas que le hemos facilitado y nos ofrece la propiedad loc para consultar las coordenadas que qureamos específicamente.

De esta forma podemos seleccionar el subscript entero, solo una variable, o un subconjunto. En el ejemplo se han seleccionado los jugadores de tenis de italia y alemania.

Como en MEDEAS hay un extenso uso de subscripts, la traducción produce una gran cantidad de DataArrays. Tantos, que se encuentran DataArrays en situaciones que no estaban previstas en PySD.

Por este motivo se ha ido refinando el código para permitir el uso de ellos, siendo problemáticas las operaciones entre ellos, ya que deben tener el mismo número de values en cada coordenada para que las operaciones se puedan llevar a cabo.

Al finalizar este TFG, aun no se ha compatibilizado totalmente el uso de DataArrays en PySD, lo que impide la correcta ejecución de MEDEAS.

7.5. Funciones

La gramática expression_grammar confundía el constructor del DataArray con argumentos en las llamadas a otras funciones.

Esto es debido a que la implementación de la regla 'call' identificaba los argumentos de la llamada separando la cadena que le llegaba en .ªrgumentos"por comas. El problema es que los propios argumentos, como los DataArray, pueden incluir comas.

Para arreglar esto, se sustituyó la regla en la que estaban los argumentos de la llamada por otra regla nueva que no se utilizase en el resto de la gramática, de esta manera, en la clase Visitor se podían almacenar las traducciones en una cola, la cual se vacía cuando llegue la siguiente llamada 'call'.

De esta manera podemos pasar argumentos con comas a las funciones.

Este fallo salió a la luz por el uso de DataArrays, sin embargo fallaba con cualquier argumento con comas que se pasase a una función. Por ejemplo:

```
MAX(MAX(1,2),3)
```

Esta pequeña operación, para calcular el máximo entre 3 números, se habría parseado como una única función MAX, la cual recibe como argumentos "MAX(1", "2)z "3".

7.6. Espaciados

En varios puntos de la gramática no se consideraban saltos de linea, tabulaciones o espacios que en realidad, son sintácticamente correctos en el lenguaje Vensim.

Estos errores son fáciles de corregir, ya que estos caracteres están definidos en una regla de la gramática con el nombre `'_'`,por lo que, una vez localizados los lugares en los que falta tener esto en cuenta, tan solo hay que añadir la posibilidad de los mismos.

La complejidad de este problema viene entonces de la localización de estos lugares, ya que una cuando la gramática falla al parsear tan solo da información de que regla ha fallado y como empieza el texto que ha generado el conflicto. Para localizar exactamente que sintaxis está fallando hay que filtrar el código de Vensim con el comienzo de texto que facilita el error de Parsimonious, deducir que línea ha fallado teniendo en cuenta la regla que ha dado el error para, después, aislar esta linea en un fichero recreando las variables que esta utilice para detectar el error concreto y poder solucionarlo.

7.7. Definición de lookups

PySD soporta la operación de lookup con la sintaxis WITH LOOKUP. MEDEAS utiliza otros 2 tipos de lookup:

- Con paréntesis
- Con Excel

Para dar soporte a estos se ha modificado la gramática expression_grammar para detectarlos y etiquetarlos como lookup. También se ha tenido que modificar la lookup_grammar y se ha generado su traducción en el Visitor de esta última.

7.8. GET DIRECT CONSTANTS

MEDEAS utiliza la sentencia GET DIRECT CONSTATS, entre otras sentencias, para leer datos del fichero Excel. Sin embargo esta sentencia no está definida en la gramática ni tiene una traducción en PySD.

Esta sentencia es equivalente a GET XLS CONSTANTS, la cual sí que está implementada. Por lo que la solución fue agregar esta sentencia a la gramática y realizar la misma traducción que la sentencia anteriormente mencionada.

7.9. En detalle

Es posible ver los cambios realizados desde el comparador de ramas de GitHub disponible en la pull request abierta para integrar los cambios
(Ver en https://github.com/julienmalard/pysd/pull/1/files)

Capítulo 8

Tests

8.1. Tests

Los tests son piezas de software diseñadas para probar el correcto funcionamiento del código que estamos escribiendo.

Los errores al escribir código son imposibles de evitar, y la probabilidad de que estos se produzcan se incrementa en proyectos con varios desarrolladores en los que continuamente se insertan cambios.

Por esto, los tests son importantes en el desarrollo de software, ya que sirven para asegurarse de que el código sigue ofreciendo los mismos resultados tras realizar cambios, o, en su defecto, detectar las funcionalidades que han dejado de funcionar antes de que los cambios se lleven a producción. [31]

El repositorio de PySD contiene un directorio tests, en el cual se encuentran los tests unitarios y los tests de integración del repositorio.

8.2. Tests Unitarios

Los tests unitarios son piezas de software diseñadas para probar una función o un objeto en concreto. Cada test debe probar una única unidad de código de forma independiente al resto de tests.

Asimismo, los tests unitarios deben ser automatizables y cubrir la mayor cantidad e código posible, probando los casos límite de cada unidad de código. [30]

Para realizar tests unitarios en Python podemos utilizar la librería unittest. Por ejemplo, si queremos implementar tests unitarios para la siguiente función

```
1    def division(a,b):
2        if b == 0: return None
3        try:
4            return a / b
5        except:
7            raise ValueError(f"Object {a} of type {type(a)} is
   ↪   not divisible by object {b} of type {type(b)}")
```

En esta función definimos la división de dos variables. Como no es posible dividir entre 0, esta función devuelve None si se intenta hacer esa división. Si la división provoca un error, lanzamos una excepción de tipo ValueError.

Supongamos que en nuestro proyecto queremos utilizar esta función con números enteros, tanto nativos de Python como de numpy, podemos crear los siguientes tests unitarios con unittest

```
import unittest
import numpy as np

class divisionTests(unittest.TestCase):
    def testPositive(self):
        a = b = 2
        self.assertEqual(division(a,b), 1)

    def testAZero(self):
        a = 0
        b = 2
        self.assertEqual(division(a,b), 0)

    def testBZero(self):
        a = 2
        b = 0
        self.assertEqual(division(a,b), None)

    def testString(self):
        a = "1"
        b = 2
        with self.assertRaises(ValueError): division(a,b)

    def testNumpyZero(self):
        a = 1
        b = np.int64(0)
        self.assertEqual(division(a,b), None)
```

En estos tests comprobamos una división correcta y diversos casos límite para asegurarnos de que la función se comporta tal cual la hemos diseñado.

Podemos ejecutar estos tests con

```
python -m unittest <fichero.py>
```

Este comando simplemente nos mostrará como resultado las siguientes lineas

```
Ran 5 tests in 0.000s

OK
```

Sin embargo, supongamos que realizamos cambios en el código, y modificamos la línea 2 de nuestra función, cambiando la condición que comprueba si el divisor es 0 por la siguiente comprobación:

```
if b is 0: return None
```

Al ejecuar de nuevo los tests obtenemos un Error:

```
FAIL: testNumpyZero (ut.divisionTests)
----------------------------------------
Traceback (most recent call last):
  File "/home/diego/ut.py", line 35, in testNumpyZero
    self.assertEqual(division(a,b), None)
AssertionError: inf != None

----------------------------------------
Ran 5 tests in 0.001s

FAILED (failures=1)
```

De esta forma, nos daríamos cuenta de que hemos eliminado la compatibilidad de la función con numpy.

En caso de que no hubiéramos sido lo suficientemente previsores como para implementar el test 'testNumpyZero' no nos daríamos cuenta de este error. Es por ello que se tiene en cuenta la cobertura que ofrecen nuestros tests.

La cobertura de nuestros tests se calcula como las líneas del código que estamos probando ejecutan nuestros tests con respecto al total de líneas de código que existen en el fichero. [32]

En nuestro ejemplo la cobertura es del `100%` ya que se ejecutan todas las líneas del código.

8.2.1. Tests unitarios en PySD

PySD tiene un fichero de tests unitarios por cada archivo Python. Dentro de cada uno de estos ficheros se crea un objeto de Test por cada función, y dentro de cada objeto los tests unitarios de dicha función.

Pese a que en PySD se utiliza también el módulo unittest para crear los tests, no se ejecutan de la forma anteriormente descrita, sino con la utilidad nosetests ya que permite comprobar la cobertura de nuestros tests y exportar los resultados en otros formatos, como xml o html. [29]

El comando para ejecutar los tests de PySD con nosetests es

```
nosetests --with-coverage --cover-package=pysd
```

La metodología Test Driven Development (TDD) se basa en el principio de crear los tests del proyecto en la primera etapa del desarrollo, antes de programar el código mismo del proyecto. Y, después, crear el código necesario para implementar las funcionalidades que permitan pasar los tests. [35]

PySD se planificó según la metodología Test Driven Development, por lo que existen multitud de tests creados que no se utilizan, están etiquetados con el decorador @unittest.skip para no ser tenidos en cuenta al ejecutar los tests.

La cobertura del código nos la ofrece la misma utilidad nosetests.

Al finalizar la aportación al proyecto comprendida en este TFG los tests pasan satisfactoriamente con una cobertura del 87 %.

Los tests unitarios que se pasan gracias a las aportaciones de este trabajo se pueden consultar en los cambios del fichero `tests/unit_test_vensim2py.py` en la comparación de ramas expuesta en la sección 7.9.

8.3. Tests de Integración

Como hemos visto anteriormente, los tests unitarios tienen un alcance limitado a una sola función o clase. Sin embargo, los tests de integración prueban la interacción entre varios de estos elementos una vez que han sido probados con tests unitarios para asegurar el correcto funcionamiento de un sistema o subsistema, o la interacción entre ellos. [34]

8.3.1. Tests de Integración en PySD

Para los tests de integración, en PySD se utiliza a su vez otro repositorio de GitHub llamado SDXorg. [33]

El repositorio SDXorg contiene 2 directorios:

- **Test:** Este directorio contiene modelos con mínima funcionalidad, es utilizado para probar funcionalidades concretas.
- **Samples:** Este directorio contiene modelos completos.

Como acabamos de mencionar, ambos directorios contienen **modelos**. Más concretamente, cada ejemplo contenido en estos directorios contiene:

- **Modelo Vensim** : Un fichero con extensión **mdl** de Vensim.
- **Imagen** : Una imagen que represente el modelo. Es suficiente con una captura de pantalla del editor Vensim.
- **Salida** : Un fichero **output.tab** o **output.csv** con la salida obtenida al ejecutar dicho modelo con las variables programadas por defecto.
- **Modelo XMILE** : Opcionalmente se puede aportar un fichero con extensión **xmile** de XMILE con el mismo modelo que el fichero Vensim.

Para usar estos tests se han creado herramientas en el fichero tests/test_utils.py entre las que destacan

- **runner:** Esta función ejecuta el modelo con el nombre que se pase por parámetro y obtiene la salida esperada del fichero output.tab o output.csv y devuelve ambos resultados
- **assert_frames_close:** Compara los resultados obtenidos con runner y devuelve un error si el alguno de los resultados esperados difiere más de 0.00001 del resultado obtenido.

Por lo tanto, los tests de integración usan estas herramientas para traducir y simular los modelos que se encuentran en el repositorio SDXorg y comparar las soluciones.

El repositorio SDXorg no está actualmente activo, por lo que, al intentar agregar nuevos ejemplos al repositorio, fue necesario hablar con el propietario del repositorio de PySD, James Houghton, quien nos dio acceso de edición en el repositorio SDXorg.

Los tests de integración que se pasan gracias a las aportaciones de este trabajo se pueden consultar en los cambios del fichero `tests/integration_test_vensim_pathway.py` en la comparación de ramas expuesta en la sección 7.9.

Además se ha añadido un test de integración con 3 casos de lecturas de Excel, el cual también está disponible en esta comparativa.

8.4. Integración Continua

En el repositorio de PySD se utiliza Travis para implementar Integración Continua (CI).

Travis ofrece servicios un servicio de prueba integración continua gratuita, con Travis se automatiza la ejecución de los tests cada vez que se añade código al repositorio en GitHub,

de esta manera se puede ver sin esfuerzo si el trabajo que se ha desarrollado puede integrarse al repositorio. [36]

Para dar de alta un repositorio de Github en Travis hay que seguir los siguientes pasos:

- Crearse una cuenta en travis-ci.com
- Dar permisos a Travis sobre la cuenta de Github
- Seleccionar el repositorio desde la web de Travis
- Añadir un fichero .travis.yml al repositorio con las instrucciones

El fichero .travis.yml contiene los comandos que se deben seguir para ejecutar los tests, el archivo de configuración del repositorio contiene los siguientes comandos

```
language: python
python:
- "2.7"
- "3.5"
- "3.6"
# command to install dependencies
cache: pip
install:
- pip install cython
- pip install --upgrade pip setuptools wheel
- pip install --only-binary=numpy,scipy numpy scipy
- pip install .
- pip install -r requirements.txt
- pip install coveralls
# command to run tests
script:
    - cd tests
    - nosetests --with-coverage --cover-package=pysd
    - coveralls
```

Con esta configuración se prueba, cada vez que se modifica el código del repositorio, si todos los tests pasan con diferentes versiones de Python

Capítulo 9

Conclusión y trabajo futuro

9.1. Conclusión

Tras finalizar este TFG no se han cumplido los objetivos iniciales del mismo, ya que no hemos obtenido una versión de PySD en lenguaje Python cuya ejecución sea más rápida que la versión existente al inicio del proyecto.

Sí que se han cumplido los objetivos que se replantearon durante el transcurso del proyecto logrando traducir el fichero MEDEAS satisfactoriamente.

A nivel de aprendizaje este proyecto ha superado mis expectativas, pues he mejorado diferentes aspectos:

- Compensión de las librerías matemáticas de Python numpy, pandas, xarray
- Mayor comprensión sobre los parsers, descubrimiento de la librería parsimonious así como su comprensión.
- Aprendizaje de sistemas dinámicos y el lenguaje Vensim
- Aprendizaje de algunas herramientas disponibles en VSCode
- Apreciación de la utilidad de los tests
- Descubrimiento de la librería unittest

9.2. Trabajo futuro

Si bien se ha cumplido el objetivo de traducir MEDEAS a código Python, no se ha logrado terminar la fase de ejecución para incorporar todos los cambios presentes, por lo que se puede terminar la fase de ejecución para ello.

Tampoco se han introducido mejoras en la eficiencia de PySD, por lo que otra línea de trabajo para este proyecto es el estudio de la eficiencia de el código traducido por este traductor y su posible mejora.

Bibliografía

[1] Package Installer for Python, *Web oficial de PIP*, Consultado el 02/09/2020, Disponible en https://pypi.org/project/pip/

[2] Herramienta de entornos virtuales de Python, *Web oficial de Virtualenv*, Consultado el 05/03/2020, Disponible en https://virtualenv.pypa.io/en/stable/

[3] Descripción de Vensim, *Web oficial de Vensim*, Consultado el 12/02/2020, Disponible en http://vensim.com/vensim-software/

[4] Ventana Systems, *Ventana Systems*, Consultado el 03/09/2020, Disponible en https://www.ventanasystems.com/

[5] Introducción a la dinámica de sistemas, *Web de la Universidad de Salamanca*, Consultado el 18/02/2020, Disponible en https://cidta.usal.es/cursos/simulacion/software/VENSIM/Vensim.PDF

[6] Stocks and flows, *Wikipedia*, Consultado el 04/03/2020, Disponible en https://en.wikipedia.org/wiki/Stock_and_flow

[7] The Teacup Model, *PySD Cookbook*, Consultado el 10/02/2020, Disponible en https://pysd-cookbook.readthedocs.io/en/latest/analyses/getting_started/Hello_World_Teacup.html

[8] Ley de enfriamiento de Newton, *Sites Google*, Consultado el 16/04/2020, Disponible en https://sites.google.com/site/ecuacionesdiferentes/home/ley-de-enfriamiento-de-newton

[9] Subscripts, *Documentación de Vensim*, Consultado el 21/04/2020, Disponible en https://vensim.com/documentation/ref_subscripts.htm

[10] Lookups, *Documentación de Vensim*, Consultado el 11/05/2020, Disponible en https://vensim.com/documentation/22820.htm

[11] MEDEAS overview, *Web oficial de MEDEAS*, Consultado el 09/03/2020, Disponible en https://medeas.eu/project/overview

[12] Entregables de MEDEAS, *Web oficial de MEDEAS*, Consultado el 13/02/2020, Disponible en https://medeas.eu/deliverables

[13] RESULTADOS Y CONCLUSIONES,*Web oficial de MEDEAS*, Consultado el 16/03/2020, Disponible en https://www.medeas.eu/system/files/documentation/files/Medeas%20Brochure%20Spanish.pdf

[14] DINÁMICA DE SISTEMAS COMPLEJOS. INTRODUCCIÓN.,*UVa_online*, Consultado el 23/04/2020, Disponible en https://www.youtube.com/watch?v=7EsFf_0EUQk

[15] MODELADO Y SIMULACIÓN DE SISTEMAS DINÁMICOS,*Universidad Nacional de Rosario*, Consultado el 24/04/2020, Disponible en https://www.fceia.unr.edu.ar/~kofman/files/eci_MyS_1.pdf

[16] MANUEL LOAIZA RAMÍREZ,*Diseño y simulación de un criptosistema caótico para comunicaciones seguras [Tesis]*, Cholula, Puebla, México 2006. , Disponible en http://catarina.udlap.mx/u_dl_a/tales/documentos/lem/loaiza_r_m/capitulo3.pdf

[17] PATRONES DE DISEÑO. 1.6. VISITOR,*Canal de YouTube de la Universidad Politécnica de Madrid*, Consultado el 18/06/2020, Disponible en https://www.youtube.com/watch?v=RKR6tbjHnZY

[18] PARSIMONIOUS 0.8.1,*Web PyPI*, Consultado el 09/06/2020, Disponible en https://pypi.org/project/parsimonious/

[19] PYTHON REGEX FLAGS,*Xah Lee*, Consultado el 06/08/2020, Disponible en http://xahlee.info/python/python_regex_flags.html

[20] PARSIMONIOUS,*Github*, Consultado el 15/08/2020, Disponible en https://github.com/erikrose/parsimonious

[21] VISUAL PARADIGM ONLINE,*Visual Paradigm*, Consultado el 27/08/2020, Disponible en https://online.visual-paradigm.com

[22] UN NUEVO PEG PARSER PARA PYTHON,*paradigmadigital*, Consultado el 01/09/2020, Disponible en https://www.paradigmadigital.com/eventos/un-nuevo-peg-parser-para-python/

[23] USEFUL THINGS TO KNOW ABOUT PARSERS,*Gabriele Tomassetti*, Consultado el 26/08/2020, Disponible en https://tomassetti.me/parsing-in-python/#structure

[24] MÉTODO DE EULER,*Wikipedia*, Consultado el 11/03/2020, Disponible en https://es.wikipedia.org/wiki/M%C3%A9todo_de_Euler

[25] MÉTODO DE EULER,*QWE wiki*, Consultado el 06/03/2020, Disponible en https://es.qwe.wiki/wiki/Euler_method#Local_truncation_error

[26] DECORATOR (PATRÓN DE DISEÑO),*Wikipedia*, Consultado el 01/04/2020, Disponible en https://es.wikipedia.org/wiki/Decorator_(patr%C3%B3n_de_dise%C3%B1o)

[27] MÉTODOS NUMÉRICOS,*MateWiki Universidad Politécnica de Madrid*, Consultado el 24/08/2020, Disponible en https://mat.caminos.upm.es/wiki/M%C3%A9todos_num%C3%A9ricos

[28] RECTA TANGENTE,*superprof*, Consultado el 03/09/2020, Disponible en https://www.superprof.es/apuntes/escolar/matematicas/calculo/derivadas/recta-tangente.html

[29] NOSETESTS,*readthedocs*, Consultado el 12/08/2020, Disponible en https://nose.readthedocs.io/en/latest/man.html

[30] PRUEBA UNITARIA,*Wikipedia*, Consultado el 07/08/2020, Disponible en https://es.wikipedia.org/wiki/Prueba_unitaria

[31] UNA INVERSIÓN DE FUTURO,*paradigmadigital*, Consultado el 07/08/2020, Disponible en https://www.paradigmadigital.com/dev/test-unitarios-perdida-de-dinero-o-una-inversion-de-futuro/

[32] COVERAGE.PY,*Wiki del paquete Coverage*, Consultado el 08/08/2020, Disponible en https://coverage.readthedocs.io/en/latest/

[33] SDXORG,*Github*, Consultado el 06/04/2020, Disponible en https://github.com/SDXorg/test-models/tree/d7762091600d6b40786521207cb564c86ac456f0

[34] TEST DE INTEGRACIÓN VS TEST UNITARIOS,*Blackpentsoft*, Consultado el 10/08/2020, Disponible en https://blackpentsoft.wordpress.com/2013/02/04/test-de-integracion-vs-test-unitarios/

[35] TDD,*agilealliance*, Consultado el 10/08/2020, Disponible en https://www.agilealliance.org/glossary/tdd/#q=~(infinite~false~filters~(postType~(~'page~'post~'aa_book~'aa_event_session~'aa_experience_report~'aa_glossary~'aa_research_paper~'aa_video)~tags~(~'tdd))~searchTerm~'~sort~false~sortDirection~'asc~page~1)

[36] TO GET STARTED WITH TRAVIS CI USING GITHUB,*Travis CI*, Consultado el 04/06/2020, Disponible en https://docs.travis-ci.com/user/tutorial/#to-get-started-with-travis-ci-using-github

[37] METODOLOGIAS ÁGILES,*Conectart*, Consultado el 02/09/2020, Disponible en https://blog.conectart.com/metodologias-agiles/#1_metodologia_agileScrum

[38] LA GUÍA DE SCRUM,*scrumguides*, Consultado el 02/09/2020, Disponible en https://www.scrumguides.org/docs/scrumguide/v2016/2016-Scrum-Guide-Spanish.pdf#zoom=100

[39] QUE ES SCRUM,*Joel Francia*, Consultado el 02/09/2020, Disponible en https://www.scrum.org/resources/blog/que-es-scrum

[40] ARTEFACTOS,*scrummanager*, Consultado el 02/09/2020, Disponible en https://www.scrummanager.net/bok/index.php?title=Artefactos

[41] CÓMO CALCULAR LA DEPRECIACIÓN DE UN ORDENADOR,*Ángel Juárez*, Consultado el 05/09/2020, Disponible en https://www.rentoox.com/blog/como-calcular-depreciacion-ordenador/

[42] QUE ES GITHUB,*Hostinger*, Consultado el 07/09/2020, Disponible en https://www.hostinger.es/tutoriales/que-es-github/

[43] Repositorio de PySD oficial, *Github*, Consultado el 10/02/2020, Disponible en `https://github.com/JamesPHoughton/pysd`

[44] Fork de PySD de julienmalard, *Github*, Consultado el 07/04/2020, Disponible en `https://github.com/julienmalard/pysd`

[45] Fork de PySD de Roger, *Github*, Consultado el 10/02/2020, Disponible en `https://github.com/rogersamso/pysd`

[46] Fork de PySD de este trabajo, *Github*, Consultado el 15/09/2020, Disponible en `https://github.com/DVRodri8/pysd`

[47] Locomotion, *Web de la Universidad de Valladolid*, Consultado el 02/09/2020, Disponible en `http://escueladoctorado.uva.es/export/sites/comunicacion/e2d64d2d-980e-11e9-a779-d59857eb090a/`

[48] Topological sorting, *Wikipedia*, Consultado el 18/09/2020, Disponible en `https://en.wikipedia.org/wiki/Topological_sorting#Kahn's_algorithm`

[49] Pull request del TFG, *GitHub*, Consultado el 24/09/2020, Disponible en `https://github.com/julienmalard/pysd/pull/1/files`

Apéndice A

Manual de instalación y uso

A continuación se encuentran las instrucciones precisas para la instalación y ejecución del software desarrollado.

A.1. Instalación

Es posible realizar la instalación con o sin entornos virtuales, en caso de que se desee utilizarlos se instalará un entorno virtual de python 3, con la herramienta virtualenv lo podemos instalar y activar ejecuando:

```
virtualenv -p /usr/bin/python3 <nombre>
source venv/<nombre>/bin/activate
```

Se descarga el código fuente del software clonando el repositorio de github, y se selecciona la rama new_functions:

```
git clone https://github.com/DVRodri8/pysd.git
cd pysd
git checkout new_functions
```

A continuación se instalan las dependencias y PySD mediante:

```
python setup.py
```

Con estos pasos queda instalada la versión de PySD desarrollada en este trabajo.

A.2. Ejecución

Para realizar una traducción y llevar a cabo una simulación hay que crear un pequeño archivo que importe la librería pysd.

Por ejemplo:

```
import pysd

model = pysd.read_vensim("ruta a fichero .mdl")
model = pysd.load("ruta a fichero .py")

resultados = model.run()
print(resultados)
```

En el ejemplo anterior se ilustra como instanciar el modelo mediante 2 opciones:

- traducir un fichero Vensim utilizando **read_vensim**
- cargar un archivo previamente traducido con **load**

Al obtener el modelo mediante cualquera de estas 2 opciones podemos ejecutarlo para llevar a cabo la simulación mediante **model.run()**, lo cual deveuelve un pandas.DataFrame con los resultados.