UCLV
Universidad Central
"Marta Abreu" de Las Villas

MFC
Facultad de Matemática
Física y Computación

Licenciatura en Ciencia de la Computación

TRABAJO DE DIPLOMA

YADSL: Lenguaje de programación para el Análisis Visual de datos

Autor: Alejandro Javier González Fonte

Tutores: Dr. C. Carlos Pérez Risquet

Lic. David Rodríguez Mollineda

Santa Clara, junio, 2018

Hago constar que el presente trabajo fue realizado en la Universidad Central Marta Abreu de Las Villas como parte de la culminación de los estudios de la especialidad de Ciencia de la Computación, autorizando a que el mismo sea utilizado por la institución, para los fines que estime conveniente, tanto de forma parcial como total y que además no podrá ser presentado en eventos ni publicado sin la autorización de la Universidad.

_____

Firma del autor

Los abajo firmantes, certificamos que el presente trabajo ha sido realizado según acuerdos de la dirección de nuestro centro y el mismo cumple con los requisitos que debe tener un trabajo de esta envergadura referido a la temática señalada.

_____ _____

Firma del tutor Firma del jefe del Laboratorio

La mejor forma de predecir el futuro es inventarlo.

Alan Kay

A la memoria de mi mamá ...

A la memoria de mi abuelo Alido ...

A mi abuela Judith ...

A mi hermano Adri ...

A mi papá Melvin ...

A mi familia, tío, tía, mi primo ...

A mi novia Lianet ...

A todos mis amigos ...

A todos aquellos que siempre me ayudaron y creyeron en la realización de este trabajo

RESUMEN

Existen muchos Lenguajes de Dominio Específico (DSL, por las siglas en inglés Domain-Specific Language) que se emplean en distintas ramas de la investigación. Estos son muy usados debido a su sencillez y expresividad en términos del dominio; sin embargo, su implementación es compleja y muchas veces es difícil enmarcar un dominio de aplicación concreto para desarrollar uno, y no existe un lenguaje específico para el análisis visual de datos usando el modelo del campo de Visual Analytics planteado en (Mollineda, 2017) *el cual permite manejar un gran volumen de datos heterogéneos de manera intuitiva y sencilla. El objetivo general de este trabajo de diploma consiste en desarrollar un DSL que permita el manejo de estos datos usando el modelo mencionado. Los principales resultados son: (1) la definición de un modelo extendido para el análisis visual de datos, (2) la definición de la gramática de un lenguaje propio para el modelo, así como la implementación de un conjunto de instrucciones para probar su aplicabilidad y (3) la incorporación de este a la herramienta previamente desarrollada.*

Palabras Clave: *DSL, Visual Analytics, datos heterogéneos, modelo extendido, gramática*

ABSTRACT

There are many Domain-Specific Languages (DSL) that are used in different branches of scientific research, wich are plenty used due it's simplicity and expansibility in domain terms, nevertheless, it's implementation is very complex and some times is hard to frame it in a concrete domain to it's develop, but there isn't any ment to data visual analysis using the model of the field Visual Analytics developed by (Mollineda, 2017) *wich allows handle a large amount of heterogeneous data in a simple and intuitive way. The main goal of this research is develop a DSL in wich data can be handled using the previously mencioned model. The main results are: (1) the definition of an extended model for data visual analysis, (2) the definition of a grammar for a specific programming language for the model, as well as the implementation a set of the instructions to prove its relevance and (3) the incorporation of these to the previously developed tool.*

Keywords: *DSL, Visual Analytics, heterogeneous data, extended model, grammar*

Tabla de Contenidos

INTRODUCCIÓN **7**

CAPÍTULO 1. MODELO PARA ANÁLISIS VISUAL DE DATOS **13**

1.1 CARACTERÍSTICAS DEL MODELO 13

Estructura general *13*

Ejecución del flujo de análisis *14*

Formas de interacción *17*

1.2 ANÁLISIS DE LOS COMPONENTES DEL MODELO 19

Consideraciones generales *20*

Componentes de entrada *20*

Componentes de análisis *21*

Componentes de visualización *22*

1.3 MODELO EXTENDIDO 22

Estructura general *22*

Componentes del modelo *23*

Consideraciones sobre los componentes ya desarrollados *23*

1.4 ALTERNATIVAS A CONSIDERAR 24

1.5 TRABAJOS SIMILARES 26

WOOL *26*

Kepler *28*

VisTrails *28*

Dotty *29*

WebWorkFlow *30*

YAWL *30*

1.6 ALTERNATIVAS PARA DESARROLLAR INTÉRPRETES 31

Lex y Yacc *31*

GNU Bison *32*

JavaCC *33*

ANTLR *33*

1.7 HERRAMIENTAS SELECCIONADAS 35

Lenguaje de programación *35*

Analizador Sintáctico *36*

Interfaz gráfica *36*

1.8 CONCLUSIONES PARCIALES 37
CAPÍTULO 2. PROPUESTA DEL LENGUAJE YADSL 38
2.1 RESUMEN DE YADSL 38
2.2 CARACTERÍSTICAS DEL LENGUAJE 38
2.3 ESTRUCTURA DE UN PROGRAMA YADSL 39
2.4 TIPOS DE DATOS 40
Datos básicos 40
Datos Compuestos 44
2.5 OPERADORES 50
Operaciones aritméticas 50
Operadores específicos del dominio 51
2.6 ESQUEMAS 54
2.7 SENTENCIAS BÁSICAS 55
Sentencias de asignación 55
Sentencia if-elif-else 57
Sentencia if-else 58
Sentencias iterativas 59
Break, continue, pass 60
Sentencia setup 60
Sentencia del 61
Sentencia start 61
Sentencia publish 62
2.8 SENTENCIAS DE VISUALIZACIÓN 63
Show 63
2.9 SENTENCIA *IMPORT Y FROM-IMPORT* 64
2.10 WORKFLOWS 66
2.11 FUNCIONES BUILT-IN 67
Funciones de visualización 68
Persistencia de las vistas de datos 69
2.12 DEFINICIÓN DE FUNCIONES 69
Sentencia return 70
2.13 MECANISMO DE COMUNICACIÓN CON EL MODELO 71
CONCLUSIONES PARCIALES 72

CAPÍTULO 3. IMPLEMENTACIÓN DE YADSL EN BCSMAP **73**
3.1. ANÁLISIS DE ACTORES Y CASOS DE USO 73
3.2. DISEÑO Y COMPORTAMIENTO DE LA INTERFAZ DE USUARIO 74
Consola Interactiva *75*
Pestañas de Scripts *75*
3.3. DISEÑO DE LA IMPLEMENTACIÓN 76
Paquete `gramar` *77*
Paquete `visitor` *78*
Módulo `controller` *81*
3.2 CASO DE ESTUDIO PLANTEADO POR (MOLLINEDA, 2017): PROPIEDADES QUE INFLUYEN SOBRE LA PERMEABILIDAD 82
3.3 COMPARACIÓN DE RESULTADOS 87
3.4 CONCLUSIONES PARCIALES 88
CONCLUSIONES **89**
RECOMENDACIONES **90**
BIBLIOGRAFÍA **91**
ANEXOS **93**
GRAMÁTICA DE YADSL 93

LISTA DE FIGURAS

Figura 1.1 Comportamiento de la ejecución de un componente 16
Figura 1.2 Núcleo Monolítico 24
Figura 1.3 Núcleo distribuido 25
Figura 1.4 Núcleo centrado en forma de estrella 26
Figura 1.5 Arquitectura de VisTrails 29
Figura 1.6 Estructura general de un parser generado por JavaCC 33
Figura 2.1 Ejemplo de datos usados por el modelo 54
Figura 2.2 Interfaz de usuario del componente csvinput 64
Figura 2.3 Mecanismo de interacción general 71
Figura 3.3 Interfaz estilo obscuro 74
Figura 3.4 Interfaz estilo claro 75
Figura 3.5 Estructura de paquetes y módulos de la aplicación 77
Figura 3.6 Diagrama de clases: ts 79
Figura 3.7 Diagrama de clases: controller 81
Figura 3.8 Esquema y componentes 83
Figura 3.9 Configuración de los componentes 84
Figura 3.10 Conexiones entre componentes 85
Figura 3.11 Representación visual del resultado de las instrucciones 85
Figura 3.12 Comparación de la distribución de clases para el nuevo agrupamiento 87
Figura 3.13 Primera vista 88
Figura 3.14 Tercera vista 88

INTRODUCCIÓN

Según (Hudak, 1997) cuando la mayoría de las personas piensan en un lenguaje de programación piensan en un **lenguaje de programación de propósito general** (GPL[1], por las siglas del inglés *General Purpose Language*)**:** uno capaz de programar cualquier aplicación con relativamente el mismo grado de expresividad y eficiencia. Para la mayoría de aplicaciones, sin embargo, hay una vía más natural de expresar la solución a un problema que aquellos atacados por un lenguaje de propósito general. Como resultado, los investigadores en la actualidad han desarrollado muchos lenguajes de programación específicos del dominio (DSL[2]).

Un DSL es un conjunto reducido de construcciones y operaciones que brindan una mayor expresividad y optimización para un dominio particular (Luzza, et al., 2012). Según (Hudak, 1996) un DSL es "la última abstracción" que captura la semántica de un dominio de aplicación. Algunos DSL bien conocidos son SQL (Kreines, 2000), expresiones regulares, Mathematica, Matlab, Csound, GraphViz (Bank, 2015), Tex y LaTex, (Borbón & Mora, 2013) por mencionar algunos.

En (Hudak, 1997) se sintetizan las ventajas de un DSL:

1. Son más concisos.
2. Se pueden escribir más rápidos los programas en ellos.
3. Son fáciles de mantener.
4. Son fáciles de comprender.
5. **Usualmente pueden ser usados por personas sin grandes conocimientos de programación.**

[1] General Purpose Language

[2] Domain-Specific Language

Esta última ventaja es la mejor justificación del porqué desarrollar un DSL, ya que usualmente están destinados a investigadores de cualquier rama de la ciencia, que no necesariamente poseen vastos conocimientos de programación.

Desarrollar DSL permite a los investigadores expresar su conocimiento de una materia en una herramienta de cálculo muy poderosa como son las computadoras, algo muy bien recibido por los investigadores, ya que muchas veces se enfrentan con un increíble volumen de datos para analizar. Para poder analizar exitosamente y validar varias hipótesis, es necesario generar varias consultas, correlacionar datos disparejos y crear complejas visualizaciones tanto del proceso simulado como del fenómeno en estudio. La exploración de los datos mediante visualizaciones requiere que los científicos realicen varios pasos de análisis, necesitan construir complejos flujos de trabajos para la selección de los datos, la especificación de una serie de operaciones a realizar sobre los datos para poder generar apropiadamente las visualizaciones, antes de que finalmente puedan analizar los resultados. Usualmente para poder arribar a conclusiones necesitan comparar los resultados de varios procesos distintos o realizar el mismo proceso con diferentes juegos de datos o bien variando los parámetros de la simulación. Todo este proceso de exploración está lejos de la interactividad, es propenso a errores, y muy consumidor de tiempo (Callahan, et al., 2006).

Si los analistas e investigadores tuvieran vastos conocimientos en un lenguaje de programación tan potente como C o C++ (Lippman, 2002), estos pudieran realizar sus investigaciones sobre algunos de estos lenguajes. Esto generaría una buena eficiencia al momento de ejecutar algoritmos, pero el tiempo de desarrollo sería muy elevado debido a la complejidad de programar en un lenguaje tan complejo. Wesley J. Chung planteó en la conferencia de *Google IO* del año 2011 que: “es más fácil encontrar un científico en Ciencia de la Computación interesado en ramas como Biología o Física, que encontrar a un físico o un biólogo interesado en las ramas de la programación”, por lo que enseñar a estos analistas como programar sobre estos lenguajes sería algo complejo. Sin embargo, si se les brinda un lenguaje de más alto nivel, que fuera sencillo de aprender, que en el código quede planteado la experticia del investigador sobre el tema de una manera clara y expresiva, y que no requiera vastos conocimientos de programación, los investigadores pudieran disminuir el tiempo que demoran sus investigaciones, ya que no deben procesar todos estos datos manualmente y el

desarrollo de algoritmos que los procesen tampoco sería muy extenso en tiempo, ya que en un DSL solo se describe cómo resolver el problema en términos del dominio y no hay construcciones especiales de código para poder completar un algoritmo como ocurría con los GPL (Luzza, et al., 2012).

Aunque estos DSL presentan muchas ventajas, también es necesario considerar las desventajas propuestas por (van Deursen, et al., 1998):

1. El costo de diseño e implementación.
2. La dificultad de encontrar un alcance apropiado para el DSL.
3. La dificultad del balance entre lenguaje de dominio específico y de propósito general.
4. La potencial pérdida de eficiencia comparados con otros lenguajes.

Un tipo de DSL ampliamente usados son los *workflow languages*[3], que han sido de reciente interés como herramienta para la computación científica. Muchos problemas científicos de dominio específico se pueden expresar muy fácilmente como *workflows* [4], porque se acercan al nivel conceptual de abstracción de los investigadores (Hulette, 2008).

Los *workflows* son un paradigma interesante, porque reflejan un enfoque intuitivo a la construcción de programas similar al sencillo estilo de "cajas y flechas" cuando se está realizando un boceto de un programa durante la fase conceptual de su creación (Weske & Vossen, 1998). Además, permiten la reproducibilidad, compartir los resultados, y el reúso del conocimiento en la comunidad científica (Davidson & Freire, 2008).

Según (Weske & Vossen, 1998) usar lenguajes de programación de bajo nivel puede oscurecer la abstracción natural de los *workflows,* porque los desarrolladores son forzados a trabajar con primitivas del lenguaje y detalles que no están relacionados con los asuntos específicos del dominio de aplicación.

En las aplicaciones actuales los datos se producen a un ritmo sin precedentes. Mientras la capacidad de colectar nuevos datos y almacenarlos se incrementa rápidamente, la habilidad

[3] Lenguajes basados en flujos de trabajos

[4] Flujo de trabajo

de analizar ese volumen de datos crece muy lentamente. Esta situación crea nuevos desafíos en el proceso de análisis, ya que los analistas, ingenieros y tomadores de decisiones dependen de información escondida en los datos. El campo emergente de *Visual Analytics* se enfoca en manejar ese masivo, heterogéneo, y dinámico volumen de información mediante la integración del juicio humano usando representaciones visuales y técnicas de interacción en el proceso de análisis(Kerren *et al.*, 2008). Dada la complejidad de los datos, es posible que el proceso de extracción de conocimientos usando esta técnica se convierta en un proceso largo y complejo. Para evitar esta extensión de tiempo y automatizar un poco el proceso, pero aun dejando en mano de los especialistas las decisiones, muchas veces es necesario un lenguaje de programación que brinde facilidades de control y visualización. Para ello sería útil el desarrollo de un DSL, ya que un GPL dificultaría mucho este proceso.

Una de las aplicaciones prácticas del campo de *Visual Analytics* es el proceso de fabricación de fármacos, que es costoso económicamente y muchas veces las decisiones sobre qué fármacos desarrollar es errónea, por la cantidad de variables en juego en el proceso de análisis (Mollineda, 2017). Destinado a este proceso se realizó en la Universidad Central "Marta Abreu" de las Villas una tesis de grado, donde se plantea un modelo para el uso de técnicas de *Visual Analytics* aplicadas a este campo, usando la clasificación BCS (Absorption System, 2013). Este modelo está basado en *workflows*, y no está limitado solo a aplicaciones biofarmacéuticas, ya que puede ser aplicable a cualquier otro campo donde los datos tengan características semejantes al usado (Mollineda, 2017).

Como parte de la tesis previamente mencionada se desarrolló un software dedicado a este aspecto, que implementa el modelo planteado por (Mollineda, 2017), pero que carece de un lenguaje de programación propio que provea más facilidades de uso y la capacidad de expresar el conocimiento humano de una forma expresiva y concisa, ya que solo se puede usar mediante su interfaz gráfica. Además, posee dos grandes limitantes, no existe una separación entre la definición de sus flujos de trabajos y sus instancias, o sea, para poder ejecutar un flujo de trabajo con diferentes parámetros el usuario necesita interactuar con el GUI[5] para manualmente introducirlos, claramente este proceso no escala a más de unas pocas

[5] Graphical user interface, interfaz gráfica de usuario

visualizaciones y esta modificación de parámetros es destructiva, no existe un historial de cambios.

Por tanto, el **objetivo general** de la investigación es:

Diseñar un lenguaje de programación propio para el modelo propuesto por (Mollineda, 2017) y realizar una implementación de un subconjunto de instrucciones del lenguaje para aumentar su flexibilidad y facilitar su uso en aplicaciones reales.

En base a ello se definen los siguientes **objetivos específicos**:

1. Identificar las características relevantes que presenta el modelo para su extensión mediante un lenguaje de programación.
2. Determinar las características que debe presentar un lenguaje de programación para aumentar las potencialidades del modelo.
3. Proponer el diseño de un lenguaje de programación.
4. Extender la herramienta informática para que implemente un intérprete para el lenguaje propuesto.
5. Demostrar la aplicabilidad del lenguaje con un caso de prueba.

Las **preguntas de investigación** planteadas en base a estos objetivos son:

1. ¿Cuáles propiedades y elementos del modelo pueden extenderse?
2. ¿Cuáles serían las características de los lenguajes de programación actuales más útiles para aumentar las funcionalidades del modelo?
3. ¿Cómo mejora el lenguaje propuesto la usabilidad y flexibilidad del modelo?
4. ¿Qué mecanismos resultan adecuados para el desarrollo y adaptación del lenguaje propuesto al modelo respetando su marco de trabajo y funcionamiento?
5. ¿Cuál caso de prueba sería apropiado demostrar su aplicabilidad?

El presente trabajo presenta una gran importancia **teórica y práctica**, ya que se desarrollará el diseño de un nuevo lenguaje válido para cualquier aplicación que cumpla con las características popuestas en el modelo planteado, mejorando la capacidad de los expertos de

expresar su conocimiento en términos del dominio, permitiendo mayor flexibilidad en el uso del modelo y así reducir el tiempo empleado para realizar las investigaciones científicas.

Este documento está divido en tres capítulos. El CAPÍTULO 1 hace un análisis del modelo propuesto por (Molineda, 2017) para extraer sus principales características, si es necesario extender este modelo, los posibles patrones de diseño de lenguajes de programación, así como paradigmas de programación existentes en varios lenguajes y potencialidades que pudieran brindar a modo de introducción. En el CAPÍTULO 2 se discute la propuesta del diseño del lenguaje YADSL (***Y**et **A**nother **D**omain-**S**pecific **L**anguage*) para realizar análisis visual de datos usando el modelo propuesto por (Mollineda, 2017), haciendo énfasis en las consideraciones referentes a utilidad y extensibilidad. En el CAPÍTULO 3 se presentan los resultados obtenidos a partir de la implementación de la propuesta y su validación. Finalmente se exponen las conclusiones obtenidas, recomendaciones para la continuación de este trabajo en el futuro y las referencias bibliográficas que fueron empleadas.

CAPÍTULO 1. MODELO PARA ANÁLISIS VISUAL DE DATOS

Debido a que el lenguaje a desarrollar es específico del dominio es necesario tener un buen conocimiento del ámbito sobre el cual se desarrolla, por lo que a continuación se analizará el modelo propuesto en (Mollineda, 2017) para determinar sus principales partes, su principio de funcionamiento básico, sus formas de interacción y los componentes que presenta. También se propone un modelo extendido para facilitar la integración de YADSL con el modelo.

1.1 Características del modelo

En (Mollineda, 2017) se plantea un modelo para el análisis visual de datos que presenta muchas características que se deben analizar y tener en consideración. En este epígrafe se discute la propuesta del modelo. Primeramente, se aborda su estructura general y luego se analizan las particularidades de la ejecución e interacción con este.

Estructura general

La forma que adopta el modelo es la de un conjunto de componentes interconectados, que se puede imaginar como un grafo dirigido, por lo que la forma general del modelo está dada por un grafo dirigido G_M compuesto por un conjunto de componentes analíticos C_A, haciendo como vértices, y una serie de conexiones entre estos E_C, que determinan la forma en que fluyen los datos a través de los componentes interconectados (Mollineda, 2017).

Los componentes pueden ser divididos en conjuntos de acuerdo a su función, pudiendo determinarse tres grandes grupos: componentes de entrada, de visualización y de análisis.

De los elementos de entrada $C_I \subset C_A, C_i \neq \emptyset$ su función principal es la entrada y cierto nivel de limpieza y filtrado de los datos que entran al sistema. Aunque todos los tipos de componentes son capaces de recibir todos los tipos de interacción, estos en la mayoría de los casos no deberían recibir entradas de otros componentes, aunque está permitido (Mollineda, 2017).

Los componentes de análisis representan procesos de análisis automatizado, aplicación de técnicas estadísticas o filtrado y transformación de los datos de forma general, contenidos en el conjunto $C_T \subset C_A$. Generalmente, los datos entrados serán transformados y procesados en estos componentes (Mollineda, 2017).

Los componentes de visualización $C_V \subset C_A$ reciben la mayor parte de la interacción del usuario, permitiendo la manipulación de los datos y focalizar los elementos que le resulten interesantes a este, así como ser los principales responsables de la presentación de información y la generación de hipótesis (Mollineda, 2017).

De las conexiones entre los componentes es necesario recalcar que no se admiten conexiones múltiples entre el mismo par de componentes, o sea, dos componentes no pueden tener dos conexiones en la misma dirección, aunque si en direcciones contrarias, ya que esto para el modelo carece de sentido y además no se pueden realizar conexiones del componente consigo mismo, aunque este comportamiento en el modelo extendido sería deseable aplicar para ganar generalidad, por lo que (Mollineda, 2017) define las conexiones como: $e_j \in E_c = (c_p, c_q)$ con $c_p, c_q \in C_A, p \neq q$.

Ejecución del flujo de análisis

Cada componente $c_i \in C_A$ tiene asociada información que conforma su estado actual s_i, el cual determina su comportamiento frente a las entradas que reciba. Este estado incluye, además de características propias al componente, su estado de activación actual, o sea, si reacciona o no ante las entradas recibidas (Mollineda, 2017).

Según (Mollineda, 2017) el comportamiento de un componente está definido como una función en ramas B_i de la entrada recibida por el componente (I_i), su estado actual (s_i) y la interacción del usuario (U) de la siguiente manera:

$$B_i(I_i, s_i, U) = \begin{cases} B_{Si}(s_i), I_i = \emptyset \wedge U = U_{exe} \\ B_{Ii}(I_i, s_i), I \neq \emptyset \wedge U = \emptyset \\ B_{Ui}(I_i, s_i, U), U \neq \emptyset \end{cases}$$

Sería conveniente extender este comportamiento de manera tal que soporte la interacción con el lenguaje (K) de una manera sencilla, de manera que el comportamiento de un componente

estaría definido por: $B_i(I_i, s_i, U, K) = \begin{cases} B_{Si}(s_i), I_i = \emptyset \wedge U = U_{exe} \\ B_{Ii}(I_i, s_i), I \neq \emptyset \wedge U = \emptyset \\ B_{Ui}(I_i, s_i, U), U \neq \emptyset \\ B_{Ki}(I_i, s_i, K), U = \emptyset, K = \ Instrucción \end{cases}$ donde el resultado de la activación del componente en la última rama estaría determinado por la instrucción dada al mismo por el lenguaje.

El resultado de la evaluación de esta función, lo cual es equivalente a la ejecución del componente, es siempre una lista de vistas de una fuente de datos (Mollineda, 2017). Este comportamiento se debe mantener en el diseño del lenguaje, ya que esta contiene los resultados de la activación de un componente que en la mayoría de los casos es de interés. El lenguaje debe ser capaz de obtener estos resultados y manipularlos de una manera eficiente y útil para los especialistas, ya que estos representan los resultados de los análisis.

Según (Mollineda, 2017) las vistas de datos son la estructura en la que se considera que se mueven los datos a través del flujo de análisis y la naturaleza de las entradas recibidas por los componentes, siendo susceptibles de ser creadas o modificadas de acuerdo con la interacción o función de estos.

Aunque todos los componentes generan salidas, o vistas de datos como lo define (Mollineda, 2017) se contempla la posibilidad de que la lista generada sea vacía ($L_k = \emptyset$) esto permite un comportamiento estándar, necesario para definir correctamente el lenguaje.

De igual manera, la ejecución del componente puede, como efecto secundario, modificar su estado interno, de tal manera que ante la misma entrada se pueda obtener un resultado diferente dependiendo de la naturaleza del componente y las ejecuciones anteriores. Esto incluye la posibilidad de ignorar entradas ya procesadas anteriormente, o refinar resultados anteriores en base a conocimiento obtenido con posterioridad (Mollineda, 2017).

La Figura 1.1**¡Error! No se encuentra el origen de la referencia.** ilustra el comportamiento general de la ejecución de un componente frente a los diferentes tipos de entrada e interacción, incluyendo su interacción con el lenguaje.

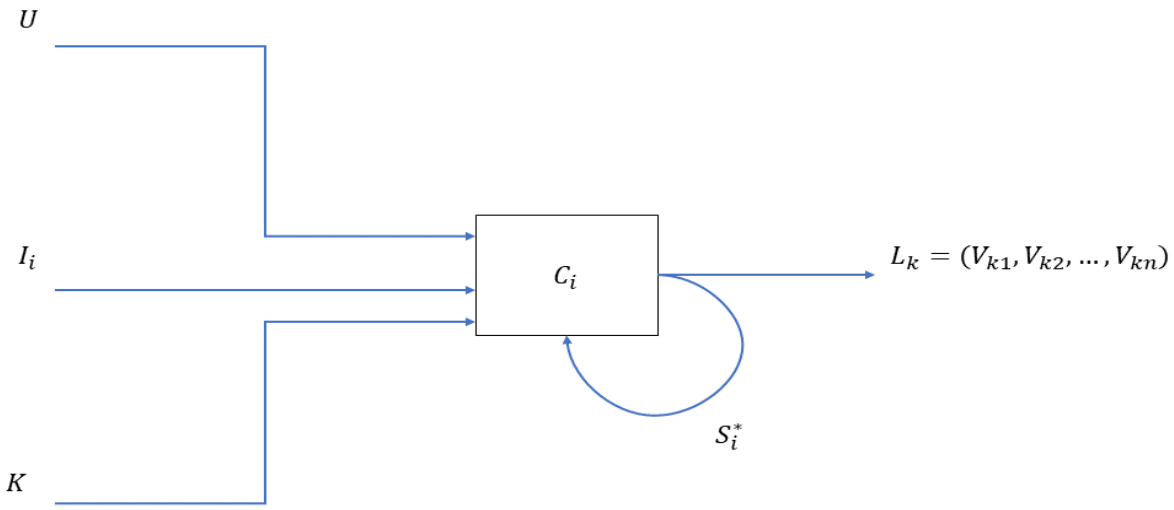


Figura 1.1 Comportamiento de la ejecución de un componente

Según (Mollineda, 2017) el modelo contempla dos escenarios para iniciar su ejecución, global o detonada localmente a partir de un componente específico. En el primero de los casos, se produce la ejecución simultánea de todos los componentes de entrada existentes en el sistema, transfiriendo su salida a cada uno de los componentes que se encuentren directamente conectados a estos. Esta primera salida debe consistir normalmente de una única vista correspondiente a la totalidad de los datos existentes en la fuente manejada por el componente de entrada que la generó, teniendo en cuenta los filtros y condiciones que este haya impuesto (representados por su estado interno) pero no está limitado a esto, depende en gran medida de los componentes.

En el segundo caso, la ejecución puede empezar a partir de una localización específica en vez de detonando todas las entradas al mismo tiempo, mediante la interacción del usuario. Para que la ejecución no se limite únicamente al componente inicial, este debería pertenecer al conjunto C_I (componentes de entrada). En esta forma de ejecución es menos probable que la ejecución se propague por todo el modelo, siendo por tanto ideal para trabajar separadamente con subprocesos de análisis (Mollineda, 2017).

Sería interesante proponer un tercer caso: ejecución controlada por el lenguaje, que pudiera ser cualquiera de las dos anteriores, pero no está limitado a estas, ya que pudieran aparecer instrucciones más complejas que permitan un comportamiento del flujo de análisis más

dinámico como cambios en la topología de la red, cambios en el estado de un componente, etc.

También es importante analizar que los componentes y conexiones pueden estar desactivados. En cualquiera de estos casos, cuando llega un dato a procesar, el componente o conexión omite esa entrada y no la propaga por la red, se pudiera decir que muere en él. Este comportamiento debe ser manipulable mediante alguna instrucción de YADSL, este debe poseer algún mecanismo de manipulación del estado de activación de los componentes y conexiones, así como poder preguntar su estado actual.

Formas de interacción

En el trabajo de (Mollineda, 2017) se definen las formas en las que el modelo contempla la interacción del usuario. Su forma más básica es la alteración de los parámetros representados en el estado interno de un componente, lo que en otros términos se refiere a aplicar un cambio en su configuración. Se había mencionado además que, en el caso de una acción que responda puramente a un cambio de configuración, la salida correspondiente por parte del componente es vacía, esta interacción también es necesaria cuando se esté definiendo el lenguaje, ya que este debe poseer alguna instrucción que permita estos cambios de configuración de una manera sencilla y rápida.

La interacción directa con las visualizaciones contenidas en los componentes que pertenecen al conjunto de los componentes de visualización es una de las más intuitivas. Su estado interno contiene los parámetros de la configuración del componente, y las vistas de datos que han transitado por ellos o información extraída a partir de estas. Esta información adicional es la que se emplea para crear la visualización correspondiente al componente. Como estas visualizaciones son una parte muy importante en el análisis, el lenguaje debe poseer instrucciones que permitan realizar estas visualizaciones de una manera sencilla, permitiendo incluso generar una persistencia de estas visualizaciones.

Según (Mollineda, 2017) la otra forma de interacción más poderosa disponible para el especialista es la posibilidad de ajustar y rediseñar el flujo de análisis de acuerdo a los resultados obtenidos con cada ejecución. La distinción entre ajuste y rediseño se realiza en base a si la interacción modifica o no la estructura del flujo de análisis existente. En el caso

de un ajuste, se refiere por lo general a un cambio en el estado de activación de algún conjunto de componentes o conexiones. Como consecuencia de este cambio, la ejecución sigue una ruta diferente, posiblemente cambiando las entradas recibidas por los componentes en una etapa posterior del flujo. Sin embargo, la estructura subyacente se mantiene intacta y el cambio puede ser revertido fácilmente, puesto que los componentes y la información contenida en su estado todavía se encuentra disponible. Esta forma de interacción debe ser la más tratada por el lenguaje, ya que este debe permitir activar o desactivar las conexiones, realizar cambios más agresivos de una manera simple, permitiendo al especialista modificar el flujo de análisis sin necesidad de reconfigurar la red cada vez lo que redunda en una ganancia en el tiempo que los analistas dedicarían al trabajo con el modelo.

Según (Mollineda, 2017) lo anterior equivale a que si se considera este tipo de interacción como una función U_{adj} se tiene que: $U_{adj}(G_M) = G_M^*$ con $G_M^* = (C_A^*, E_c^*)$ tal que existe algún componente c_x o conexión e_y para el cual su estado de activación difiere entre G_M y G_M^*. En este caso solamente cambia el estado de los componentes involucrados.

En (Mollineda, 2017) se plantea que cuando se trata de una operación de rediseño, se considera que la estructura del flujo de análisis actual es alterada, ya sea mediante la inserción o eliminación de componentes o conexiones. Particularmente en el caso de la eliminación, a diferencia de cuando solamente se cambia el estado de activación del componente, el estado asociado al componente se pierde irreversiblemente cuando este es eliminado. Esto implica que, incluso si se vuelve a insertar otro componente del mismo tipo que el que fue eliminado, los resultados obtenidos a partir de la ejecución del flujo pueden ser distintos de los que se habrían obtenido originalmente a menos que el estado interno del componente pueda ser reproducido perfectamente.

Este comportamiento es deseable mantenerlo, con la excepción de que la operación de rediseño no sea irreversible, o sea, que el rediseño pueda ser temporal, de tal manera que dadas ciertas condiciones se pueda eliminar un componente o conexión y luego restablecerlo, mientras aún se está realizando la ejecución del flujo de análisis. Incluso sería deseable tener un mecanismo capaz de salvar el estado actual de un componente para después poder reproducirlo. Estos mecanismos facilitarían aún más el trabajo de los analistas cuando realizan sus investigaciones, ya que reduciría el tiempo que se pierde al hacer estos cambios

manualmente y eliminaría la necesidad de estar anotando el estado de los componentes para luego recrearlos.

Finalmente, (Mollineda, 2017) plantea que la forma de interacción restante se relaciona directamente con la preservación y externalización del conocimiento obtenido a partir del proceso de análisis. Esta se realizaría a través de los componentes, pero actuaría sobre las vistas de datos que circulan por el sistema. La idea es que los conjuntos de datos con características interesantes determinados a través de la interacción puedan ser recuperados y reutilizados. En una implementación real del modelo, esto podría verse como guardar a un soporte persistente (fichero, base de datos, etc.) las características que definen a esas secciones interesantes dentro de los datos.

Esta característica es imprescindible extender, dada la necesidad de la retroalimentación en base al conocimiento extraído, la necesidad de almacenar ciclos de análisis para su después uso, permitiendo que un mismo diseño de flujo de análisis sea aplicado sucesivamente para entradas diferentes, incorporando en cada ocasión nuevas entradas correspondientes al conocimiento extraído de iteraciones anteriores. Teniendo en cuenta lo anterior sería interesante una propuesta de un meta-modelo, un modelo relativamente similar al propuesto por (Mollineda, 2017), pero donde cada componente sería todo un flujo de análisis y las conexiones transportarían los resultados de unos a otros, quedando más general y flexible.

1.2 Análisis de los componentes del modelo

Habiendo extraído las características esenciales del modelo en su conjunto, en esta sección se analizan las particularidades correspondientes a tipos de componentes específicos. Intentando extraer generalidades para incluir en el modelo extendido, o meta-modelo, que sean útiles en el proceso de análisis y puedan ser aprovechadas por el lenguaje.

Como los componentes tanto de visualización, análisis o entrada, son componentes en sí, todos tienen características comunes, capaces de recibir entradas, emitir salidas y realizar algún tipo de procesamiento sobre los datos. Como estas características las presentan todos los componentes, es posible extraer rasgos comunes que pueden ser aprovechados por el lenguaje, como poder guardar las salidas de cada componente, provocar que se realice algún procesamiento sobre los datos, etc.

Consideraciones generales

Todos los componentes, en el modelo extendido, serán capaces de reportar al modelo las variables o procedimientos que el desarrollador del componente considere necesario para su uso por el analista. Al mismo tiempo si son atributos, se debe poder especificar su tipo de acceso, ya sea de solo lectura o lectura y escritura. También deben poseer un mecanismo que permita tanto suministrarle datos a procesar como obtener los ya procesados. Finalmente, a cualquier componente en cualquier momento, al no ser que esté realizando una operación, se le podrá cambiar sus parámetros de configuración de la manera que especifique el analista, especificando que parámetro se alterará.

Componentes de entrada

En (Mollineda, 2017) se plantea que los componentes de entrada son el medio por el que dichos datos entran a formar parte del flujo de análisis, por tanto, son los más involucrados en la creación de las vistas de datos.

Según (Mollineda, 2017) el estado interno s_w de un componente de entrada $c_w \in C_I$ solamente necesita considerar su propio estado interno y la fuente de datos a partir de la que crea las vistas que incorpora al proceso de análisis.

Un caso especial es cuando la función b_{v_w} siempre devuelve verdadero para cualquier entrada, equivalente a decir que se incorporan al análisis todos los datos disponibles en la fuente de datos asociada al componente. La principal utilidad de esto radica en la incorporación de los resultados de análisis. El comportamiento de los componentes de entrada puede ser generalizado además para tratar con múltiples fuentes de datos: basta asumir que F_d es una lista de estas, con lo que v_w puede tratarse como la concatenación de las salidas correspondientes a la aplicación del componente para cada una de las fuentes disponibles o producir una salida independiente para cada fuente de datos (Mollineda, 2017).

La observación más importante sobre este tipo de componente, es que el lenguaje debe ser capaz de suministrarle los datos en su estado puro, ya sea de una base de datos, un archivo local u otra fuente de datos, de manera dinámica, o sea, que pudiera cambiar durante la ejecución o incluso suministrarle varias fuentes de datos.

Componentes de análisis

En (Mollineda, 2017) se plantea que, a diferencia de los componentes de entrada o los de visualización, para los componentes de análisis la forma más usual de su función de activación depende de entradas al componente, dado que su principal función es la de procesar y transformar las salidas generadas por otros componentes. Los componentes correspondientes a este conjunto pueden ser distinguidos de acuerdo al impacto que tiene cada ejecución en el estado interno o parámetros que el componente emplea para su funcionamiento, y a cómo afecta su ejecución a las entradas recibidas.

Los tres tipos de componentes principales de análisis que (Molineda, 2017) menciona son: filtros, observadores y generadores. Un componente filtro sería aquel que no opera a nivel de los datos que componen la vista que recibe como entrada, sino que considera a la vista en su conjunto, permitiendo o previniendo su propagación hacia otros componentes del flujo de análisis.

Según (Molineda, 2017) un componente observador estaría cercano en su naturaleza a un componente de visualización, en el sentido de que su ejecución produciría normalmente la misma vista de datos que le fue dada como entrada. Este tipo de componentes simplemente extraerían información a partir de los datos que los atraviesen sin modificar activamente el flujo de análisis, distinguiéndose fundamentalmente de un componente de visualización en que la información o conocimiento recabado no tiene por qué ser expresado mediante una visualización.

Los generadores son aquellos que, como consecuencia de la acción del componente, la vista de datos recibida como entrada produce una salida diferente a la entrada en al menos un valor. Estos componentes están destinados a realizar transformaciones sobre los datos que componen las vistas que recibe como entrada.

A diferencia de las determinadas por el impacto de la ejecución del componente en su estado interno, estas categorías no son excluyentes entre sí, pudiendo ser combinadas de acuerdo a las necesidades particulares a las que un componente haya de responder (Mollineda, 2017).

Componentes de visualización

Según (Mollineda, 2017), desde el punto de vista de la interacción con el usuario, son los componentes de visualización los que tienen el mayor grado de responsabilidad.

Los componentes de visualización tienen como función principal mostrar resultados mediante tablas, imágenes o cualquier otro medio de visualización. Sobre estos componentes el lenguaje debe poseer mecanismos que permitan generar estas imágenes, almacenarlas, e incluso hacer comparaciones con ellas. También sería deseable que estas imágenes se puedan guardar como una imagen que se pueda interpretar por cualquier visor de imágenes y además como un formato propio.

Al construir un componente de visualización para el modelo, se debería evitar dentro de lo posible, para actuar en concordancia con el principio de flexibilidad y no imponer restricciones innecesarias al analista, pensar en escenarios muy específicos con propiedades fijas. El diseño de representaciones aplicables a cualquier grupo de propiedades con características similares y que puedan ser intercambiadas dinámicamente sin afectar su funcionalidad está más de acuerdo con la propuesta de modelo que ha sido discutida (Mollineda, 2017).

1.3 Modelo extendido

Previamente se ha descrito implícitamente el meta-modelo propuesto, pero ahora se formalizará en su totalidad y se especificarán las nuevas características a tener en cuenta. Este modelo es necesario para escalar las potencialidades del modelo propuesto por (Mollineda, 2017), así como generalizar su comportamiento.

Estructura general

El nuevo modelo, o modelo extendido, según lo visto anteriormente quedaría definido como un grafo dirigido, MtG_M compuesto por un conjunto de componentes MtC como vértices, y una serie de conexiones entre estos, que determinan la forma en que fluyen los datos a través de los componentes interconectados. Es necesario aclarar que un componente $c \in MtG_M$ está compuesto por uno o varios componentes del modelo propuesto por (Mollineda, 2017), o por uno o varios $c \in MtG_M$ además de otras características que se abordaran más adelante.

Componentes del modelo

Cada componente $C_m \in MtC$ es en sí todo un flujo de análisis donde tiene un componente interno, o varios, de entrada que son los responsables de obtener la información de otro componente. Esto se podría definir como $C_e = \left(C_{I_1}, C_{I_2}, \dots, C_{I_n}\right), C_{I_i} \in (C_I \vee MtC)$, donde C_I son los componentes mencionados por (Mollineda, 2017) como entrada, y además tiene uno o varios de salida que quedan definido como $C_s = (C_1, C_2, \dots, C_n), C_i \in (C_A \vee MtC)$, donde C_A es el conjunto de cualquier componente del modelo presentado por (Mollineda, 2017), o sea, cualquier componente es capaz de publicar sus resultados hacia su meta-componente superior. Burdamente se podría imaginar el meta-modelo como un grafo dirigido, donde la información de cada nodo es otro grafo dirigido del mismo tipo.

La única función de las conexiones entre los meta-componentes que conforman el meta-modelo es transportar el resultado de un flujo de análisis hacia otro meta-componente $Mte_j \in MtE_c = \left(MtC_p, MtC_q\right)$ con $MtC_p, MtC_q \in MtC, p = q \vee p \neq q$.

Consideraciones sobre los componentes ya desarrollados

Debido a que ya existen componentes desarrollados, es necesario tenerlos en cuenta y tomar una decisión sobre qué hacer con estos para que sean compatibles con el nuevo modelo y el lenguaje. Una de las opciones sería reescribirlos completos de tal manera que sean totalmente compatibles, pero esto generaría una gran pérdida de tiempo innecesaria. Otra variante sería mantenerlos como están y añadirles características nuevas, la cual sería la mejor opción, ya que se basa en el trabajo previo y permite la escalabilidad, ya que todos los componentes heredan de una clase común y previamente se analizaron sus características generales. La última opción sería modificarlos independientemente, de acuerdo con las necesidades de cada uno, lo que proveería una gran integración de los componentes ya escritos, pero provocaría una gran falta de generalización, que en el futuro implicaría muchas restricciones si se desea añadir componentes nuevos.

En el nuevo modelo, cada componente posee una característica importante, la cual es la capacidad de informar al lenguaje las variables y métodos que desea exportar hacia el lenguaje de manera que pueda ser usado por los investigadores. En el caso de que sean variables, estas tienen que tener su análogo en el lenguaje, lo que limita estas a los tipos

básicos de Python excluyendo tuplas. Además, deben especificar el tipo de acceso que se podrá tener sobre ellas, los cuales son fundamentalmente lectura o lectura y escritura.

1.4 Alternativas a considerar

El diseño de un nuevo lenguaje de programación, sin importar su utilidad final, siempre está sujeto a controversia, a nuevas ideas, a retomar ideas ya usadas siempre y cuando tenga sentido. Siempre es necesario pensar en qué tipos de datos son necesarios, qué estructuras de control son más prácticas. Incluso hay que considerar él o los paradigmas de programación que serían útiles en el nuevo lenguaje.

Es necesario considerar la forma en que YADSL se acoplará con el meta modelo. Las tres formas básicas que saltan inmediatamente a la mente son: primeramente, que el núcleo del lenguaje envuelva al meta modelo enviando sus instrucciones de forma descendente hacia cada uno de los componentes. La Figura 1.2 muestra lo anteriormente planteado. Este tipo de diseño generalmente es rápido y preciso, pero dificulta mucho su escalabilidad.

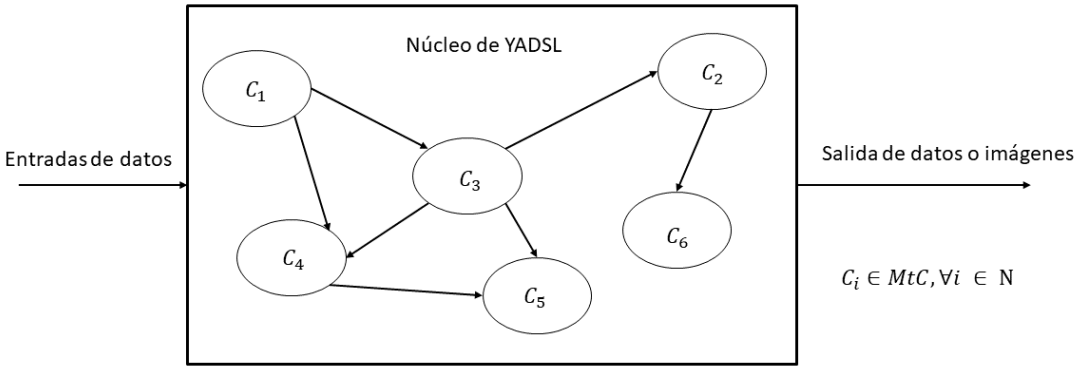


Figura 1.2 Núcleo Monolítico

Una segunda alternativa sería que el lenguaje estuviera embebido dentro de cada componente como lo muestra **¡Error! No se encuentra el origen de la referencia.**. Esto trae consigo muchos retos de implementación como la sincronización, la dificultad para la escalabilidad, ya que es necesario modificar todos los componentes. Además, presenta una gran dependencia el modelo de la funcionalidad del lenguaje, cuestión que se desea evitar, ya que

si algún usuario no desea realizar ninguna acción de programación y solo usar las características básicas que propone (Mollineda, 2017) se le dificulta su uso.

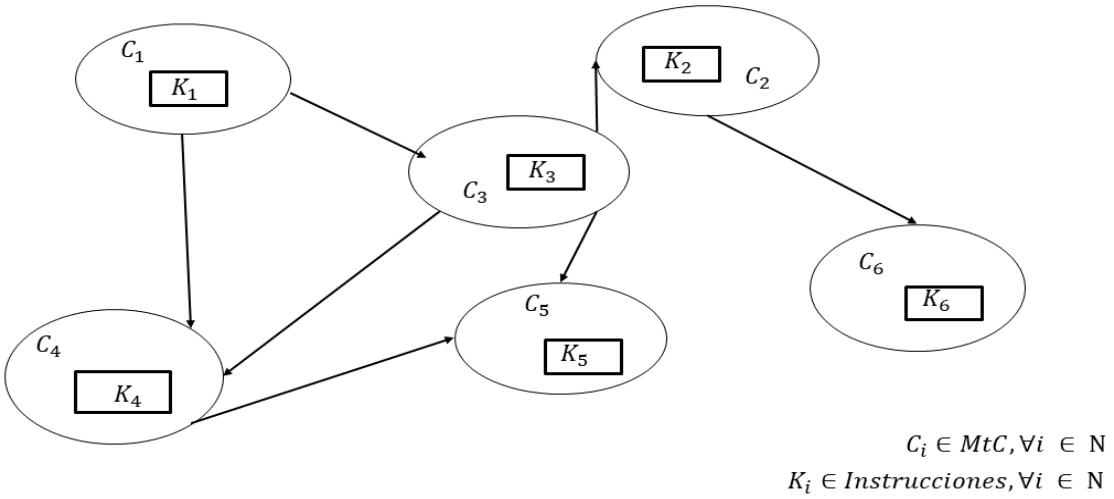
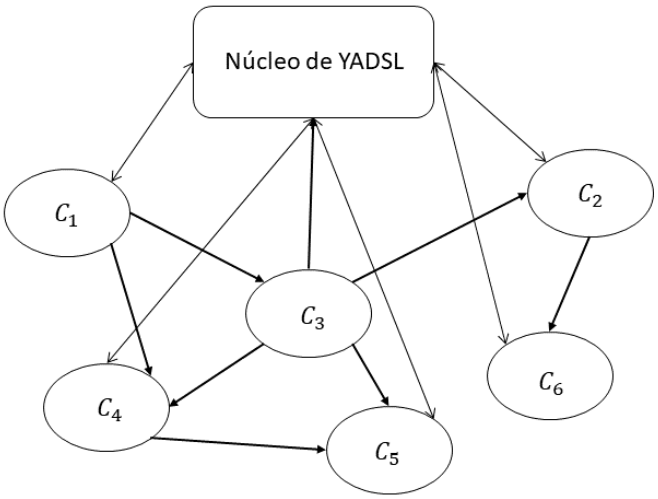


Figura 1.3 Núcleo distribuido

La tercera alternativa se muestra en la Figura 1.4, donde el núcleo se encuentra aislado del modelo. Los componentes mantienen su conexión, el modelo funciona tal cual, solo que el lenguaje adquiere la capacidad de enviar instrucciones a cada componente por separado o al modelo en general. Esta alternativa presenta varias ventajas, la primera y más evidente es la facilidad de realizar cambios en el núcleo de YADSL o en algún componente sin afectar el funcionamiento general. Esto también permite un mayor control de los resultados ya que todos son accesibles por el lenguaje, permitiéndole hacer alguna transformación si es

necesario. Permite una fácil escalabilidad ya que los componentes se pueden acoplar sin mayor esfuerzo al igual que removerlos.



$C_i \in MtC, \forall i \in \mathrm{N}$

Figura 1.4 Núcleo centrado en forma de estrella

1.5 Trabajos similares

En este epígrafe se abordarán trabajos relacionados, ya sea con la creación de lenguajes específicos del dominio del tipo *workflow* o lenguajes para la gestión de grafos.

WOOL

WOOL es un lenguaje de definición de tipos de *workflows*. Permite al programador definir flujos de trabajos desde actividades básicas, combinar flujos de trabajos para crear jerarquías y asegurar que los datos que fluyen entre actividades son consistentes. Según (Hulette, 2008) los WOOL *workflows* son basados en relaciones entre actividades basadas en flujos de datos. Una actividad representaría un conjunto de entradas sobre las que se realizarían ciertos tipos de procesamientos y luego produce ciertos resultados, donde las conexiones entre las actividades describirían el flujo de trabajo.

En este modelo se manejan términos como tipos de datos primitivos, los cuales serían, por ejemplo, números y cadenas de texto. En este modelo las instancias particulares de un tipo de dato no son importantes, solo el tipo en sí. Además, se plantean tipos de actividades que serían una abstracción de un procesamiento en particular. Estas actividades presentan puertos de entrada y salida, además de propiedades que ayudan al compilador a optimizar el código.

Estas actividades pueden ser a su vez complejas, dejando de ser abstractas y pasando a ser todo un flujo de trabajo en sí (Hulette, 2008).

Otro componente importante son las conexiones entre los flujos de trabajo, conectando un puerto de salida con uno de entrada. Estas conexiones son creadas cuando se define el flujo de trabajo y no pueden ser cambiadas en tiempo de ejecución, estas conexiones deben ser entre puertos del mismo tipo.

WOOL está designado para describir flujos de trabajos abstractos, en oposición a los flujos dependientes de la arquitectura, por tanto en (Hulette, 2008) se plantea que dependen poco del sistema en el cual esté corriendo, sin embargo, algunas características sí dependen del sistema.

Las características generales de este lenguaje se resumen según (Hulette, 2008) como:

- En tiempo de ejecución existe un mecanismo para proveer información o datos hacia los flujos de trabajos usando sus puertos de entrada, y obtener los resultados de sus puertos de salida.
- Los datos que son entregados a un flujo y no pueden ser inmediatamente procesados pasan a una cola en el orden en que llegan.
- Las actividades ejecutarán las tareas de la cola mencionada anteriormente.
- Las actividades cuando procesan un *ítem* [6] de datos lo eliminan de la cola.
- Un puerto de salida puede estar conectado a varios de entrada, así como un puerto de entrada puede estar conectado a varios de salida.

De manera general, el compilador de WOOL es un compilador de dos pasadas. En la primera pasada se transforma un conjunto de archivos de texto conteniendo una sintaxis válida a un grafo intermedio de objetos del tipo de Java. Esta representación es almacenada en memoria para su posterior ejecución o guardada al disco usando el protocolo de serialización de objetos de Java. En esta etapa se realizan todos los chequeos de tipos y las conexiones entre las actividades. La segunda pasada sería entonces para transformar esta representación de grafo hacia una estructura que permita su ejecución como *workflow*. La gramática de WOOL

[6] Elemento

es tokenizada y reconocida por un analizador sintáctico generado por ANTLR usando como target [7] código Java.

Kepler

Kepler es un software que permite el trabajo con flujos de trabajo del corte científico de una manera sencilla, los cuales pueden ser intercambiados, archivados y ejecutados. Este consta de un GUI intuitivo y un sistema de modelado orientado al actor, que permiten el trabajo con los flujos de trabajo. Los actores de Kepler corren como *threads*[8] locales de Java por defecto, pero están preparados para ser puestos de manera distribuida vía web (Altintas, et al., 2004). Actualmente Kepler soporta el diseño de *workflows*, ejecución distribuida (web y servicios Grid), el acceso a bases de datos y la ejecución de consultas. El sistema de *workflows* para trabajo en el área científica de Kepler ha sido usado para diseñar y ejecutar varios flujos de trabajos en ramas como la biología, ecología, geología, astrofísica y química (Altintas, et al., 2004).

VisTrails

El objetivo principal de VisTrails es proveer a los investigadores un proceso de análisis visualización de los datos simplificado. Este maneja los datos y los metadatos asociados con las visualizaciones.

[7] Lenguaje al cual se le generará el *parser*.

[8] Proceso ligero, conocido como hilo

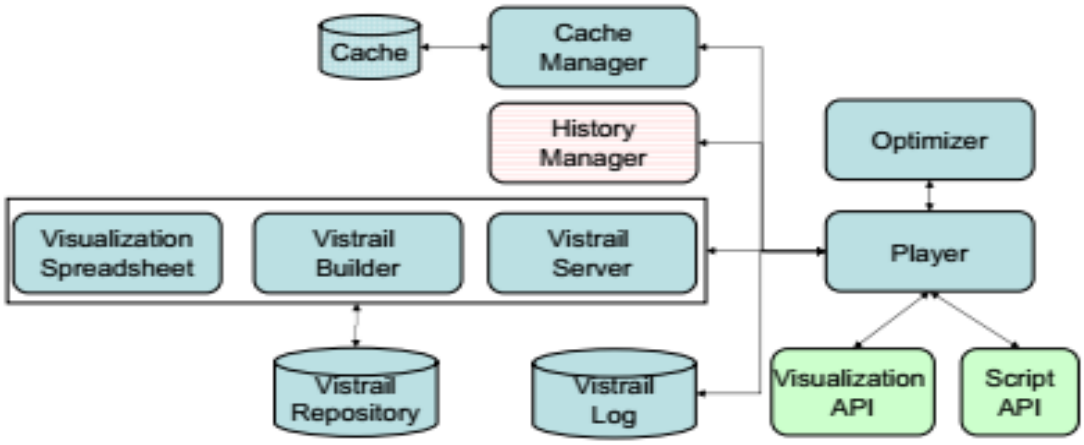


Figura 1.5 Arquitectura de VisTrails

En la Figura 1.5 se muestra la arquitectura de alto nivel presente en este sistema. Los usuarios crean y editan los flujos de trabajo con la interfaz gráfica del módulo *Vistrail Builder*. Las especificaciones son guardadas en *Vistrail Repository*. Estas especificaciones puden ser invocadas por el usuario mediante *Vistrail Server* o importarlas mediante *Visualization Spreadsheet.* Lo interesante de *VisTrail* es que introduce el concepto de rastro de visualización, que captura la evolución de los datos. Un *vistrail* consiste en una colección de varias versiones de los datos y sus instancias. Esto permite a los analistas explorar los datos mediante el regreso a versiones previas y comparar las diferencias (Callahan, et al., 2006). *VisTrail* cuenta con un DSL para realizar acciones en situaciones específicas durante el proceso de exploración, incluyendo o eliminando módulos, conexiones entre los módulos o añadiendo datos a ciertos módulos.

Dotty

Dotty es un editor gráfico para el sistema Windows X. Puede ser ejecutado como un editor solamente o como un front-end[9] para aplicaciones que usen grafos como representación gráfica. Dotty está escrita encima de *dot* y *lefty*. *Lefty* es un editor programable de propósito general para imágenes (Koutsofios & North, 1996). Presenta un lenguage interpretado similar

[9] Front-end y back-end son términos que se refieren a la separación de intereses entre una capa de presentación y una capa de datos.

a AWK [10] y C. La disposición visual de los grafos está realizada por *dot*, que se ejecuta en un proceso separado y se comunica con *lefty* mediante *pipes*[11]. *Dotty* puede ser customizada para manejar grafos para aplicaciones específicas, o sea, grafos que presenten ciertos tipos de restricciones como por ejemplo que no pudean tener ciclos (Koutsofios & North, 1996). El núcleo de *dotty* contiene varios *scripts*[12] y estructuras de datos que el usuario puede usar. Algunas de las funciones que brinda al usuario son *dotty.init function()*, *gt.creategraph(protogt), gt.copygraph(ogt)* y *dotty.monitorfile(data).*

WebWorkFlow

Es un lenguaje de modelado de *workflow* orientado a objetos para la descripción a alto nivel de *workflows* en aplicaciones web. Es un lenguaje embebido (Bravenboer & Visser, 2004) que extiende de WebDSL (Visser, 2008), el cual es un lenguje específico del dominio para el desarrollo de aplicaiones web con la abstracción de que permiten los *workflows* (Hemel, et al., 2008). Desde la definición de procedimientos que operan sobre objetos y la descripción del control del flujo para conectar esos procedimientos, aplicaciones web completas se pueden desarrollar (Hemel, et al., 2008). En vez de desarrollar un lenguaje exclusivo, WebWorkFlow soporta la interacción con WebDSL. Este enfoque permite al usuario la posibilidad de usar los flujos de trabajo cuando le sea posible y usar las facilidades del modelado web cuando lo necesite, esta práctica es llamada integración del lenguaje y separación de interés (Hemel, et al., 2008).

YAWL

Basado en un riguroso análsis de los existentes sistemas de manejo de *workflows* y los *workflows languages*, YAWL[13] fue desarrollado. Este lenguage está basado en las redes de Petri, la bien establecida teoría de la concurrencia con representación mediante grafos y del bien conocido *Workflow Patterns*. YAWL extiende la definición de la red de Petri con

[10] Lenguaje para procesar datos basados en texto

[11] Tuberías, sistema de paso de información entre procesos

[12] Archivo de órdenes o procesamiento por lotes

[13] Yet Another Workflow Language

constructos dedicados al control del flujo, la cancelación del flujo y el *OR-join* generalizado. Este ofrece soporte comprensivo para los patrones del control de flujo, los datos son capturados mediante esquemas XML[14] y soporte para los *workflows* dinámicos a través de los *Worklet.* Ha sido desarrollado independiente de cualquier interés comercial y aspira a ser una de las herramientas más poderosas para la especificación de procesos (The YAWL Foundation, 2010).

1.6 Alternativas para desarrollar intérpretes

Existen disímiles herramientas a utilizar que permiten el desarrollo de un intérprete, aunque todas son muy poderosas, algunas presentan ventajas por encima de las otras que pueden ser un factor decisivo al momento de escoger entre una de ellas. En este epígrafe se presenta una recopilación de las más usadas, mostrando sus puntos fuertes así como sus desventajas.

Lex y Yacc

Las herramientas Lex y Yacc son las más difundidas y usadas, ya que están escritas originalmente en C, lo que provee un desempeño veloz y eficiente. Estas suelen usarse juntas, aunque son totalmente funcionales cada una por separado. Lex es un programa que genera analizadores léxicos o scanners[15]. Este fue desarrollado por Eric Schmidt y Mike Lesk, y todavía es el analizador léxico estándar en los sistemas Unix[16], y se incluye en el estándar de POSIX[17]. A Lex se le especifican reglas mediante expresiones regulares, para las cuales él genera el analizador léxico. Lex es código propietario, pero se han hecho variantes basadas en él como software libre, una de ellas y la más conocida es Flex . El código generado, originalmente, era en el lenguage C, pero se ha reescrito variantes para otros lenguajes como Java y se le han realizado bindings para Python. Junto con Lex como se mencionaba con anterioridad, es necesario usar Yacc para realizar el análisis sintáctico, o propiamente dicho generar código que realice el análisis sintáctico. Sus siglas significan Yet Another Compiler-

[14] Extensive Markup Language.

[15] Es un programa que recibe una secuencia de caracteres y retorna como salida una secuencia de tokens.

[16] Es un sistema portable, multitarea y multiusuario.

[17] POSIX es el acrónimo de Portable Operating System Interface, y X viene de UNIX como seña de identidad.

Compiler18 y fue desarrollado por Stephen C. Johnson en AT&T para el sistema operativo Unix. Yacc genera un analizador sintáctico del tipo LALR(K) basado en las reglas que toma por entrada, basadas en una gramática analítica escrita en una notación similar a la BNF[19], es la parte de un compilador que comprueba que la estructura del código fuente se ajusta a la especificación sintáctica del lenguaje. Yacc originalmente tambíen al igual que Lex generaba código C, pero también se ha reescrito para otros lenguajes como Ada, Java, Limbo y Python. Este conjunto de herramientas permiten la creación de eficientes reconocedores de lenguajes, en su variante en C pero generan un código oscuro difícil de revisar y manipular gracias a la obscura sintaxis de C y los complejos procesos que se generan de *shit-reduce* y *reduce-reduce* que forman un mecanismo difícil de revisar incluso para los más expertos cuando el lenguaje es complejo. Sus otras implementaciones en otros lenguajes pierden la gran eficiencia del código escrito en C, ya que lo generan en lenguajes de más alto nivel, aunque es más manipulable y fácil de mantener siguen estando los complejos procesos de reducción y cambios. La mayor ventaja que presenta es que como es del tipo LALR(K) es capaz de reconocer un gran número de gramáticas libres del contexto.

GNU Bison

GNU Bison es un programa generador de analizadores sintácticos de propósito general, al igual que Yacc, perteneciente al proyecto GNU disponible para prácticamente todos los sistemas operativos, se usa normalmente acompañado de Flex como analizador léxico. Bison convierte la descripción formal de un lenguaje, escrita como una gramática libre de contexto LALR, en un programa en C, C++, o Java que realiza análisis sintáctico. Es utilizado para crear analizadores para muchos lenguajes, desde simples calculadoras hasta lenguajes complejos. Para utilizar Bison, es necesaria experiencia con la sintaxis usada para describir gramáticas. GNU bison tiene compatibilidad con Yacc: todas las gramáticas bien escritas para Yacc, funcionan en Bison sin necesidad de ser modificadas. Cualquier persona que esté familiarizada con Yacc podría utilizar Bison sin problemas. Bison fue escrito en un principio por Robert Corbett; Richard Stallman lo hizo compatible con Yacc.

[18] Otro generador de compiladores más

[19] Notación de Backus-Naur

JavaCC

JavaCC [20] es un generador de analizadores sintácticos de código abierto para el lenguaje de programación Java. JavaCC es similar a Yacc en que genera un parser para una gramática presentada en notación BNF, con la diferencia de que la salida es en código Java. A diferencia de Yacc, JavaCC genera analizadores descendentes, lo que lo limita a la clase de gramáticas LL(K) , en particular la recursión desde izquierda no se puede usar. El constructor de árboles que lo acompaña, JJTree, construye árboles de abajo hacia arriba. JavaCC está licenciado bajo una licencia BSD. En la Figura 1.6 se evidencia la capacidad de JavaCC de generar tanto un analizador sintáctico como semántico y su proceso que es bastante simple.

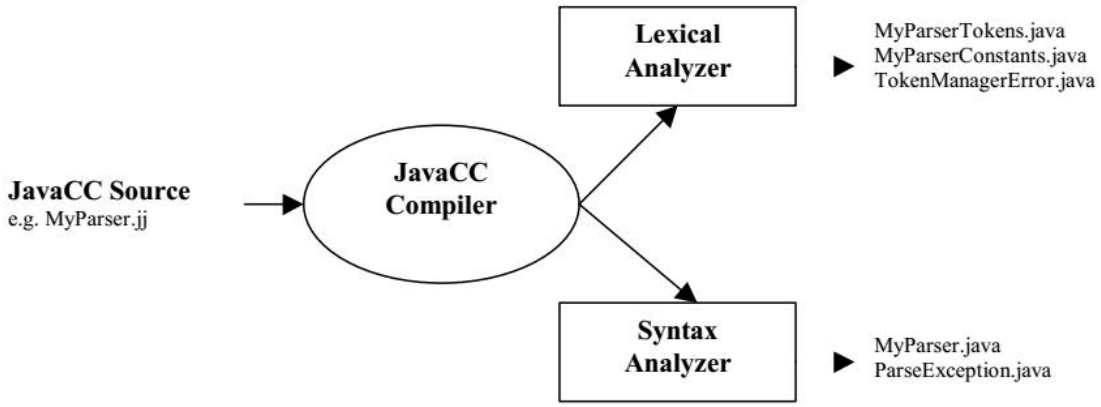


Figura 1.6 Estructura general de un parser generado por JavaCC

ANTLR

Según (Parr, 2012) ANTLR es una herramienta excepcionalmente potente y flexible para reconocer lenguajes formales. Fue creado por Terence Parr durante su tesis de Máster en la Universidad de Purdue. Sus siglas significan *Another Tool for Language Recognition*[21]. La gramática que requiere es clara y concisa y el código generado es eficiente y estable. ANTLR es un analizador del tipo descendente recursivo y utiliza algoritmos del tipo LL(*) para reconocer los lenguajes. Una de las ventajas que posee ANTRL es que proporciona facilidades para la creación de estructuras intermedias de análisis como los *AST*[22] y provee

[20] Java Compiler Compiler

[21] Otra herramienta para el reconocimiento de lenguajes

[22] Abstract Syntax Tree, o árbol abstracto de sintaxis

mecanismos para recuperarse automaticamente de los errores y realizar reportes de los mismos. ANTLR es un proyecto bajo licencia BSD, viniendo con todo el código fuente disponible, y preparado para su instalación bajo plataformas Linux, Windows y Mac OS X lo que garantiza su portabilidad. Está escrito totalmente en Java y es capaz de generar código para Java, Python, Delphi, C, C++, C#, Perl, etc. A diferencia de JavaCC, ANTLR si es capaz de resolver la recursión a la izquierda inmediata, aunque la indirecta todavía no se ha resuelto, siendo este su principal problema. Según (Parr, 2012) es ampliamente usado en el nivel académico y el industrial para construir todo tipo de lenguajes. En la compañía Twitter se usó exclusivamente para reconocer las consultas en el motor de búsqueda con más de dos billones de consultas al día. Los lenguajes como Hive an Pig y los almacenes de datos y sistemas de análisis de Hadoop usan ANTLR. Lex Machina lo usa para la extracción de información en textos legales. También es usado por Oracle en su SQL Developer IDE y sus herramientas de migración. También el conocido IDE [23] para Java NetBeans utiliza ANTLR para reconocer el lenguaje C++. Estos no son solo los únicos casos donde ANTLR es aplicable, ya que el lenguage HQL del *framework*[24] de Hibernate para el mapeo objeto-relacional, lo usa a gran escala. Esto demuestra las potencialidades de ANTLR para su aplicación en proyectos de gran embergadura.

La otra ventaja significativa que presenta ANTLR es que genera también los conocidos *tree walkers*[25] que permiten visitar nodos que necesitan ejecutarse y no aquellos que son irrelevantes. Los dos principales *tree walkers* que genera siguen el patron “listener” y “visitor”, el cual el primero visita todos los nodos generados y realiza acciones antes y después de haber pasado por estos, mientras que el visitor, “visita” cada nodo según el programador lo especifique, es posible que algunos nodos no sean visitados. Este mecanismo, junto con el *AST* generado lo hacen una excelente herramienta para el desarrollo de un eficiente intérprete, por lo que es necesario tenerlo en cuenta.

[23] Integrated Developement Enviroment

[24] Estructura básica conceptual

[25] Recorredores de árboles

1.7 Herramientas seleccionadas

En este epígrafe se realizará un resumen de las principales alternativas para la creación de intérpretes, ya sea el lenguaje de programación usado, el sistema de compilación usado, y los principales aspectos a tener en cuenta al momento de escoger los módulos para la interfaz gráfica.

Lenguaje de programación

Debido a las facilidades para un desarrollo rápido, la amplia disponibilidad de herramientas y la gran capacidad de introspección que posee se decidió emplear Python 3 como lenguaje para desarrollar la implementación, específicamente la versión 3.6.2. Debido a que el trabajo previo está desarrollado en el mismo, sería de gran utilidad usar ese poder de introspección para permitir la escalabilidad del software, ya que no esta limitado a los componentes por defecto, sino que se le pueden incrementar funcionalidades sin mayor esfuerzo que copiarlas para el directorio correspondiente.

Debido a que es un lenguaje interpretado, esto podría suponer una disminución de la velocidad de ejecución con respecto a alternativas como C/C++ o incluso Java, pero la simplicidad y elegancia del código Python permite un fácil mantenimiento y lectura del código. Además de que provee una gran cantidad de estructuras de datos de forma nativa como los diccionarios y tuplas que permiten la escritura de algoritmos complejos de una manera sencilla.

Además, la pérdida de velocidad de ejecución no constituye algo significativo, sobre todo dado que las versiones más recientes del intérprete de Python cachean el código compilado a *bytecode* después de que este se ejecuta por primera vez, aumentado considerablemente la velocidad de ejecuciones subsecuentes (Mollineda, 2017).

Otra consideración de mucho peso en la elección del lenguaje es la portabilidad, ya que Python, al ser interpretado puede correr un cualquier sistema operativo que tenga un intérprete, y además, la mayoría de la familia Linux, incluso los Mac OS X de Apple, vienen con el intérprete integrado. Para el sistema operativo Windows existen múltiples distribuciones que cuentan con todos los paquetes necesarios para tareas científicas.

Analizador Sintáctico

El analizador sintáctico seleccionado fue ANTLR en su versión 4, el cual presenta todas las características necesarias para desarrollar un *parser* [26] eficiente. Incluso evita el trabajo de diseñar una forma interna, ya que se puede usar su *AST* como forma interna. Este analizador fue escogido ya que es un proyecto maduro, ampliamente utilizado, con una gran comunidad trabajando sobre él. También porque presenta un patrón de diseño muy eficiente y fácil de usar, como el patrón *visitor* que brinda grandes facilidades, por ejemplo, permite la addición de sentencias de una manera clara y sencilla y brinda un contexto para cada sentencia para realizar chequeos.

Interfaz gráfica

Al momento de diseñar interfaces gráficas en Python surgen varias alternativas, a diferencia de Java o C++. Para Python se encuentran varios módulos como PyQt5, Tkinter, PyGTK o wxPython. Una decisión sobre ellos sería totalmente estética ya que todos son ampliamente usados y desarrollados. Todas presentan soporte para todos los sistemas operativos. Tkinter se considera la biblioteca estándar de Python para el trabajo con interfaces gráficas, es un *binding* de la biblioteca Tcl/Tk creada por John Ousterhout. Esta viene instalada por defecto en la plataforma Windows. PyGTK es un *binding* de la biblioteca GTK[27] que se usa para desarrollar el entorno gráfico GNOME, así como sus aplicaciones. Por último, wxPython es también un *binding* sobre la biblioteca wxWidgets que se caracteriza por ser multiplataforma

La decisión final es usar PyQt en su versión 5, por varios motivos. Primeramente, el trabajo previo está realizado sobre este mismo módulo lo que facilitaría la integración y la homogeneidad en el GUI[28]. Este módulo es un *binding* sobre la biblioteca de Qt, permitiendo un ambiente único sin importar la plataforma y un gran rendimiento ya que está implementado en C++.

[26] Otra denominación para los analizadores sintácticos

[27] GIMP Tool Kit, Conjunto de Herramientas de GIMP

[28] Graphical User Interface, o interfaz gráfica de usuario

1.8 Conclusiones parciales

El modelo planteado por (Mollineda, 2017) presenta muchas características útiles para desarrollarle un lenguaje de programación, además de ser un modelo planteado de manera muy general lo que facilita la integración del lenguaje. Para el correcto funcionamiento del lenguaje este debe respetar el marco de trabajo propuesto por (Mollineda, 2017). El modelo extendido representa una mayor generalización, pero sigue manteniendo el principio básico del modelo original, por lo que es compatible. El lenguaje debe permitir todas las funcionalidades del modelo e incluso flexibilizarlas más, manteniendo las complejidades y construcciones especiales de lenguajes como C lejos de los analistas para permitir un rápido desarrollo de las investigaciones y acercar más el lenguaje a los conocimientos del dominio que poseen los analistas.

CAPÍTULO 2. PROPUESTA DEL LENGUAJE YADSL

El diseño de cualquier lenguaje de programación no es una tarea sencilla. Es necesario tener muchos aspectos en cuenta como los paradigmas de programación que usará, los tipos de datos que dará soporte, etc. Esto también se aplica al diseño de un DSL, y además es necesario tener en cuenta el dominio de aplicación, ya que mientras más fácil sea expresar el conocimiento a nivel de programación más sencillo será su uso para los analistas.

2.1 Resumen de YADSL

YADSL es un lenguaje de programación específico del dominio, creado especialmente para ser usado con el modelo propuesto por (Mollineda, 2017). Es un lenguaje fácil de aprender ya que basa su sintaxis en el conocido lenguaje Python (Rossum, 2017). Es un lenguaje interpretado, basado fuertemente en el estilo de programación *scripting* y toma características de lenguajes basados en flujos de trabajos, programación orientada a grafos y orientados a objetos. Presenta tipado dinámico por lo que no es necesario declarar el tipo de las variables, permite declaración de funciones y la reusabilidad de código usando el concepto de encapsulación y debido a su naturaleza interpretada, permite un desarrollo rápido de aplicaciones, con un corto período del proceso de escritura de código, compilación, prueba y rectificación que generalmente es extenso en lenguajes compilados. Como este lenguaje está destinado al ámbito científico, intenta solucionar la necesidad de realizar varias ejecuciones seguidas de manera relativamente rápidas, sin la pérdida de tiempo en el proceso de compilación que generalmente viene acompañado de los lenguajes compilados. Este capítulo introduce de manera formal los conceptos y características básicas del lenguaje.

2.2 Características del lenguaje

Primeramente, es necesario definir conceptos necesarios y la terminología asociada al ámbito en que se desenvuelve este lenguaje. El modelo presentado por (Mollineda, 2017) tiene el funcionamiento básico de una red de Petri, y como es de esperar el modelo extendido se comporta de manera similar, por lo que podemos definir como una representación

matemática o gráfica de un sistema de eventos discretos en el cual se puede describir la topología de un sistema distribuido, paralelo o concurrente. Este modelo incluye características extras de los sistemas basados en flujos de trabajo como el mencionado anteriormente llamado WOOL. En estos los entes principales son los datos que fluyen y los componentes que los procesan. En este caso pudiéramos definir los datos como las vistas de datos mencionadas por (Mollineda, 2017) en su trabajo y los componentes serían los meta componentes definidos en el capítulo anterior.

Este lenguaje, aunque posee algunas características de lenguajes de propósito general su principal objetivo es controlar todo el sistema de eventos que definen una red de Petri, siendo capaz de obtener los resultados generados y procesarlos según sea necesario.

2.3 Estructura de un programa YADSL

De una manera concreta los programas YADSL están compuestos por módulos, sentencias, expresiones y objetos, los cuales serían a nivel del modelo planteado componentes del tipo $C_m \in MtC$. Los programas en YADSL se pueden describir de la siguiente manera:

- Los programas están compuestos por sentencias.
- Las sentencias están compuestas por expresiones.
- Las expresiones crean y procesan objetos.

De una manera más precisa, un módulo comienza por la definición de todos los esquemas que va a usar con la palabra reservada *schemas* (2.5 más adelante), luego se pueden definir *workflows* (2.10 más adelante), funciones (2.12 más adelante) o realizar sentencias básicas (2.7 más adelante) según se desee.

Como YADSL es un lenguaje de programación, en él se pueden declarar variables y funciones, estas poseen un nombre, este nombre es un **identificador**. Los identificadores en YADSL pueden comenzar por una letra o un guion bajo (_) seguido por letras o números indistintamente.

El lenguaje permite la escalabilidad de tres maneras, la primera variante sería añadir nuevos componentes al modelo propuesto en (Mollineda, 2017), que se convertirían en nuevos tipos de datos (más adelante), otra variante sería usar los mismos *scripts* que desarrolle el usuario,

donde se declaren funciones (más adelante) o flujos de trabajos (más adelante) para importarlos en otro *script* y poder usarlo, lo que permite el intercambio fácil de resultados entre analistas, ya que pueden compartir código como bibliotecas. La última vía sería mediante la adición de funciones nativas del lenguaje (más adelante) que permiten realizar acciones más complejas ya que tienen acceso a toda la infraestructura del núcleo del lenguaje. Esta última está destinada no a los analistas, sino a desarrolladores que continúen con el diseño, refinamiento y desarrollo del lenguaje. La primera variante, estaría destinada a otro tipo de desarrollador, más cercano al proceso de investigación, pero con suficientes habilidades de programación como para desarrollar todo un módulo en Python pero que tenga conocimiento del área o al menos una tutoría de algún analista de la rama de la investigación, mientras que la segunda variante si está totalmente enfocada en los analistas, está diseñada para que los investigadores plasmen sus resultados como código de tal manera que reproducir sus resultados sería tan sencillo como ejecutar el *script* apropiado.

2.4 Tipos de datos

Los tipos de datos básicos que soporta YADSL son los clásicos tipos de datos que soporta casi cualquier lenguaje y resultan familiares si alguna vez se ha programado en otro lenguaje. Los más comunes se pudieran considerar que son los números y las cadenas, en este epígrafe se analizarán estos tipos de datos en YADSL, además de otros más complejos que brindan soporte a tareas complejas durante la programación, como diccionarios, listas y tipos de datos particulares del lenguaje.

Datos básicos

Dentro de los datos básicos, YADSL cuenta con los tipos numéricos, ya sea los enteros o con punto flotante, y además las cadenas de texto. YADSL cuenta con un tipo de dato especial ‘None’ que es usado para asignar el valor nulo a cualquier variable.

Numéricos

Los tipos de datos básicos con los que cuenta YADSL son los números enteros y las cadenas de texto. Dentro de los números podemos contar con enteros, ya sea en su tipo binario,

decimal, hexadecimal u octal y en punto flotante. Por ejemplo, YADSL reconoce los enteros que se puedan escribir como -1234, 0, +456855, etc. Además YADSL brinda soporte para números enteros en formato octal, binario y hexadecimal, por lo que es capaz de reconocer números escritos, de forma general, usando un prefijo *0x* donde la *x* puede ser bien 'o' o 'O' para los números en formato octal, 'b' o 'B' para los números escrito en formato binario y 'x' o 'X' para aquellos números representados en su notación hexadecimal, seguido de una secuencia de dígitos o caracteres (caso hexadecimal) respetando el formato indicado. Por ejemplo, los siguientes números son válidos para el lenguaje: 0o444777, -0O11123, 0b111100, 0B10111, 0xABCD o incluso -0XAB4C23DF. Sin embargo, no son válidos los ejemplos siguientes:

1. 0xA5ART porque los dígitos 'R' y 'T' no son dígitos válidos en el formato hexadecimal.
2. 0o77745 porque el dígito 7 no es válido en la notación octal.
3. 0b11102 porque el dígito 2 no es válido en la notación binaria.

Los números en punto flotante o como se conocen comúnmente en otros lenguajes, los números de tipo *float*, se pueden escribir como 5.6, 4.0 o incluso en notación científica como 2.1e10 o 3.4e-2. Estos números son muy usados en el ámbito científico, de ahí la necesidad de incluirlos en el lenguaje. Todos estos números son de precisión arbitraria, o sea, no se desbordan, ya que están sobre la base de las clases numéricas de Python que están preparadas para soportar cualquier valor, sin importar su tamaño, siempre y cuando se posea suficiente memoria en el dispositivo (Ascher & Lutz, 2003).

Además, YADSL soporta números complejos, que pueden escribirse de la siguiente manera, 4+5j, 8-6J, e incluso 8e-10+4E5j. Este tipo de dato se introdujo por la falta de declaración de nuevos tipos de datos en el lenguaje, y la usabilidad de estos números en aplicaciones del corte científico.

Lógicos

YADSL soporta los datos lógicos 'true' y 'false'. Internamente estos datos están representados por los valores '1' y '0' del tipo entero respectivamente, por lo que pueden ser

tratados como tal en operaciones de adición, substracción, multiplicación, etc. Estos tipos de datos son necesarios en lenguajes como Java para las condiciones, pero en YADSL no, ya que el resto de datos se pueden usar como lógicos en ciertas condiciones.

Los valores “”, [], {}, None, 0, 0.0 se consideran como falso y cualquier otro valor se considera verdadero, por lo que es válido escribir en YADSL:

```
a = [1,2,3,4]
if a:
      print('arreglo true')
      if {}:
            print('diccionario true')
      else:
            print('diccionario  false')
else:
      print('arreglo false')
```

Strings

Casi todos los lenguajes de programación reconocen las cadenas de texto ya que este tipo de datos es muy usado y cómodo para realizar ciertas tareas, por tanto, en YADSL no podían faltar. Las cadenas de texto comúnmente se conocen como *string*[29]. Este tipo de datos almacena una secuencia de caracteres que puede ser vacía, (ej.: “”) o contener varios caracteres. Desde el punto de vista funcional se pueden usar para representar casi cualquier cosa que pueda ser codificada como texto: símbolos, palabras, contenidos de archivos de texto cargados en memoria, direcciones de Internet, códigos fuente de cualquier lenguaje, etc. Algunos ejemplos de *strings* pueden ser “The quick brown fox jumps over the fence” o ‘The quick brown fox jumps over the fence’, o incluso con las triple comillas “””The quick brown fox jumps over the fence”””. Todos estos son válidos para YADSL.

YADSL no hace una distinción especial en el tipo de dato *char* usado para almacenar un solo carácter en varios lenguajes como Java (Schildt, 2005), ya que este es perfectamente reemplazable por el tipo *string* permitiendo realizar los mismos programas con resultados semejantes en código. Los *strings* de YADSL están desarrollados sobre la base de los *string* del lenguaje Python sobre el cual se desarrolla YADSL. Lo que permite el uso de la amplia

[29] Tipo de dato que almacena una secuencia de caracteres

gama de funciones que este tipo de dato posee en Python (Ascher & Lutz, 2003). Este tipo de dato, al igual que en Python es **inmutable**, lo que significa que no se puede modificar su valor.

Sobre este tipo de dato se puden realizar algunas operaciones, por ejemplo pedir un subsegmento de él mediante subíndices. Un ejemplo de este caso en otros lenguajes como Java sería de la manera siguiente:

```
String test = "The quick brown fox jumps over the fence";
test.substring(0, 20);
```

donde el resultado sería *"The quick brown fox"*, mientras que en YADSL sería un poco más sencillo, sería como sigue:

```
In[0]: test = "The quick brown fox jumps over the fence"
Out[0]: The quick brown fox jumps over the fence
In[1]: test[0:20]
Out[1]: The quick brown fox␣
```

En el ejemplo anterior se muestra la forma de instanciar los *strings* en YADSL, evitando más complejidades se pueden poner las comillas dobles, sencillas o incluso triples, con la información adentro, si existe, y automáticamente se retorna un tipo de dato *string*, limitando las construcciones sintácticas de otros lenguajes. Además YADSL permite usar a los *strings* como arreglos de caracteres sencillos. El operador ':', es el operador *slice*[30] que realiza un corte y devuelve la porción indicada del *string* (también válido para los arreglos, en la página 44), los límites tienen que estar entre el máximo y el mínimo valor del tamaño del *string*. También es posible indexar los *string* y arreglos con números negativos, y esto empezaría a contar desde atrás hacia adelante, ej *test[20:-1]* generaría como resultado "*jumps over the fence"* o para *test[20:-5]* el resultado sería "*jumps over the "*. Además se pueden indexar similar a como sucede con los arreglos en la mayoría de los lenguajes, esta operación sería similar lo que sin usar el operador *slice*, sería *test[0]*, y el resultado obtenido es *"T"*. Estas operaciones facilitan mucho el trabajo con los *strings*, por lo que se decidió incluirlas en YADSL.

[30] Significa tajada en español, operador para delimitar un rango, ej 2:4 sería desde elemento 2 al elemento 4

Otra operación a realizar sobre los *string* es la concatenación, por ejemplo *"The quick brown fox jumps" + " over the fence"*, esta operación se puede hacer incluso sin el operador, de tal manera que dos *strings* continuos automáticamente son concatenados, por *ejemplo "The quick fox jumps"" over the fence"*, ambos devuelven como resultado *"The quick fox jumps over the fence"*. Además los *strings* soportan el operador '*' que "multiplica", por ponerlo de cierta forma, el *string* por un número *n*, repitiendo el valor del *string n* cantidad de veces, por ejemplo "string" * 3 retorna "stringstringstring". Estas operaciones son muy importantes para la el trabajo con este tipo de dato.

Es necesario remarcar que en YADSL **no se pueden definir nuevos tipos de datos**, por lo menos no de tipos básicos o como en la programación orientada a objetos, clases. Se pueden incluir nuevos módulos pertenecientes al modelo propuesto en (Mollineda, 2017) o los *workflows* (66) que se describen en el modelo extendido, pero ningún otro. De este tema se discute más adelante cuando se analicen los datos compuestos.

Datos Compuestos

Dentro de los datos compuestos YADSL cuenta con arreglos, diccionarios y los más importantes son las vistas de datos, los flujos de trabajos y los componentes básicos del modelo de (Mollineda, 2017).

Los arreglos y diccionarios son tipos de datos muy usados y útiles ya que pueden almacenar varios elementos, en algunos casos del mismo tipo, como en Java o C++, o varios tipos indistintamente como en Python.

Arreglos

En YADSL los arreglos son colecciones **mutables** de elementos. Estos elementos a su vez pueden ser de cualquier otro tipo de dato de YADSL, *strings*, números, *workflows*, diccionarios, e incluso otros arreglos (arreglos anidados). En YADSL, las listas o arreglos (*array*[31]) , desde un punto de vista funcional, son un lugar para agrupar datos, así que se pueden tratar como grupo. Los *arrays* también definen una política de ordenamiento de

[31] Nombre en Inglés para el tipo de dato arreglo.

izquierda a derecha según la posición del dato en la lista. Los accesos a los datos se realizan mediante un *offset*[32], similar a como se accede a un elemento en los *string*, usando los corchetes. Un ejemplo sería

```
In[0]:a = [1,2,3,4,5]
Out[0]: [1,2,3,4,5]
In[1]: a[1]
Out[1]:2
```

lo que daría como resultado el elemento *2* del arreglo *a*. Además se muestra una forma sencilla de instanciar un arreglo, usando los corchetes y poniendo dentro de ellos los elementos deseados separados por coma. Es importante recordar que estos elementos **no necesariamente tienen que ser del mismo tipo**. Por ejemplo en YADSL es válido escribir *[1,2,* "a", *[4,5, "test"], otra_variable]*.

A diferencia de el tipo de dato *string* los arreglos son mutables y pueden variar su tamaño varias veces en el mismo programa. Estos soportan añadirles y eliminarles elementos. Además en cualquier momento se puede conocer el tamaño real de un arreglo haciendo una llamada a una función *built-in*[33] llamada *lenght*[34] (*más adelante*), la cual retorna un número entero con la cantidad de elementos del arreglo.

Muchas operaciones lucen semejantes a como se haría con los *strings*, ya que ambos son secuencias de valores. Por ejemplo los *arrays* también soportan el operador '+', el cual concatena dos o más arreglos, por ejemplo *[1,2,3] + [4,5,6] + ['a', 'fox',[7,8,9]]* retorna el arreglo *[1,2,3, 4,5,6, 'a', 'fox',[7, 8, 9]]*, operación que resulta útil en aplicaciones reales muchas veces, por lo que se decidió darle soporte en YADSL.

Otra operación que muchas veces resulta útil es la operación '*', que al igual que con los *strings* multiplica el arreglo y lo repite las veces deseadas. A continuación se muestran ejemplos del trabajo con arreglos en YADSL:

```
In[0]: example = [1,2,3,4]
Out[0]: [1,2,3,4]
In[1]: length(example)
```

[32] Desplazamiento, comúnmente usado para moverse sobre direcciones de memoria.

[33] Embebida en el núcleo, o creada junto con el lenguaje.

[34] En español significa largo o tamaño.

```
Out[1]: 4
In[2]: example = example + [5,6,7]
Out[2]: [1,2,3,4,5,6,7]
In[3]: example * length(example)
Out[3]: [1,2,3,4,5,6,7,1,2,3,4,5,6,7,1,2,3,4,5,6,7,1,2,3,4,5,6,7]
```

En la práctica uno de los mayores usos de los arreglos es para representar matrices multidimensionales, en YADSL, se puede realizar anidando arreglos sencillos, por ejemplo:

```
In[0]: example = [[1,2],[3,4]]
Out[1]: [[1,2],[3,4]]
In[1]: example[0]
Out[1]: [1,2]
In[2]: example[1][1]
Out[2]: 4
```

En YADSL indexando el arreglo anidado, con un índice se obtiene el arreglo que se encuentra en esa posisición, y luego este se puede indexar para obtener un valor específico. En YADSL los arreglos se pueden **anidar indefinidamente**, o sea, no hay límite de cuantos arreglos puede estar dentro de otros.

Como los arreglos son **mutables** se puede variar la información contenida en una celda específica usando la sentencia de asignación '='(en la página 55) de la siguiente manera

```
In[0]: example = [[1,2],[3,4]]
Out[1]: [[1,2],[3,4]]
In[1]: example[0] = 'test'
Out[1]: test
In[2]: example
Out[2]: ['test',[3,4]]
```

Además de asignar en una posición específica, también YADSL soporta la **asignación a porciones del arreglos** (Ascher & Lutz, 2003) :

```
In[0]: example = ['cat', 'dog', 'fish']
Out[0]: ['cat', 'dog', 'fish']
In[1]: example[0:2] = ['horse', 'chicken']
Out[1]: ['horse', 'chicken']
In[2]: example
Out[2] = ['horse', 'chicken', 'fish']
```

Ambas operaciones, el indexado o asignado con porciones modifican físicamente el valor del arreglo en lugar de generar una nueva como resultado.

Diccionarios

Otra estructura o tipo de dato presente, es el tipo **diccionario**, este es quizás el tipo de dato más flexible en YADSL. Estos son colecciones sin orden de elementos, donde su principal características es que para recuperar un dato, o para insertarlo, no es necesario un índice, porque no existe el orden, sino que es necesario una llave.

Ya que es un tipo de dato nativo, los diccionarios pueden reemplazar muchos de los algoritmos de búsqueda y estructuras de datos que quizás sean necesarios implementar manualmente en otros lenguajes de bajo nivel (Ascher & Lutz, 2003). También pueden representar estructuras de datos *sparse*[35] y usualmente son usado como tabla de símbolos en varios lenguajes.

La creación de diccionarios en YADSL se realiza poniendo pares de elementos (llave, valor) separados por coma entre llaves ("{}", en el caso de un diccionario vacío) como se muestra a continuación:

```
In[0]: dict = {'integer':1, 'array': [1,2,3], 45: "dictionary"}
Out[0]: {'integer':1, 'array': [1,2,3], 45: "dictionary"}
```

Las llaves en los diccionarios pueden ser cualquier tipo de dato básico, numéricos, *strings* y lógicos. No se puede usar como llave en ningún contexto los tipos de datos diccionario y arreglo.

Los diccionarios, como los arreglos, es un tipo de dato **mutable**, por lo que permiten la asignación con el operador '=' previamente visto, la diferencia es que es necesario especificar sobre que llave se va a asignar, si esta llave existe, se sobreescribe su valor, de lo contrario se crea y se le asigna el valor.

```
In[0]: dict = {'integer':1, 'array': [1,2,3], 45: "dictionary"}
Out[0]: {'integer':1, 'array': [1,2,3], 45: "dictionary"}
In[1]:dict['array'] = [4,5,6]
Out[1]: [4,5,6]
In[2]: dict
Out[2]: {'integer':1, 'array': [4,5,6], 45: "dictionary"}
In[3]: dict['string'] = 'cadena de texto'
Out[3]: cadena de texto
In[2]: dict
```

[35] Dispersos

```
Out[2]: {'integer':1, 'array': [4,5,6], 45: "dictionary", 'string': 'cadena de texto'}
```

La característica más importante a remarcar del tipo de dato diccionario es que es **heterogéneo**, o sea, las llaves pueden ser de varios tipos (sin incluir otros diccionarios y arreglos) y los elementos que almacenan pueden ser de cuaquier otro tipo, semejante a los arreglos y además se pueden anidar al igual que los arreglos.

Componentes del modelo propuesto por (Mollineda, 2017)

Como se había mencionado previamente, otro tipo de dato compuesto son los componentes desarrollados por (Mollineda, 2017). Estos son los más importantes, se pudiera decir, ya que realizan todo el trabajo de análisis y visualización.

Los componentes con que cuenta YADSL por el momento son:

Entrada:

- `csvinput`
- `gate`

Análisis:

- `filters`
- `kmeans`
- `workflow`

Visualización:

- `radviz`
- `scatter`

De los componentes de entrada el más importante es `csvinput`, el cual es el encargado de cargar los archivos de datos y generar las primeras vistas de datos a procesar.

El componente `gate`, aunque no es un componente de análisis porque no modifica los datos es importante porque actúa de cierta manera como un filtro debido a que deja pasar solo las vistas que cumplan con la cantidad de datos especificada al componente.

Los componentes de análisis realizan procesamiento sobre los datos, el componente `filters`, como su nombre indica, actúa como un filtro, dejando pasar solo las vistas de datos

que cumplan con sus condiciones, mientras que `kmeans` realiza agrupamiento o operaciones de clusterización.

Los componentes de visualización `radviz` y `scatter` realizan visualizaciones siguiendo ciertos algoritmos (Mollineda, 2017).

Cada uno de estos exporta las variables y métodos que el diseñador de cada uno considere útil al investigador. Por el momento, ningún componente posee exportada ninguna función pero sí todas las variables relativas a su configuración.

Estas variables son (primero se muestran las generales a todos los componentes y luego por cada componente las suyas específicas):

Generales:

- Lectura:
  - `vkey`, identificador del componente (se genera automático)
  - `flag_enabled`, variable booleana para conocer si el componente está activo
- Lectura y escritura:
  - `vtasks`, donde se encuentran las vistas de datos a procesar
  - `vlabel`, la etiqueta del componente para su visualización
  - `vconfig`, la configuración del componente

`csvinput`:

- Lectura y escritura:
  - `vschema`, el esquema que usa para generar las entradas

`workflow`:

- Lectura y escritura:
  - `schemas`, el esquema que usa para generar las entradas
  - `auto_start`, dice si el componente puede iniciar incluso sin entradas de datos

`radviz`:

- Lectura y escritura:
    - `vviews`, las vistas que han llegado al componente

`scatter`:

- Lectura y escritura:
    - `vviews`, las vistas que han llegado al componente

Es importante decir que como el modelo sobre el cual se trabaja es extensible (Mollineda, 2017), el lenguaje hereda esta característica, ya que cualquier extensión desarrollada sobre el modelo se convierte en un tipo de dato compuesto en el lenguaje.

2.5 Operadores

En el siguiente epígrafe se discuten los operadores disponibles en YADSL, ya sean los aritméticos o los específicos del dominio.

Operaciones aritméticas

YADSL soporta la mayoría de las operaciones aritméticas presentes en muchos lenguajes como suma, resta, multiplicación, etc. Más detalladamente las operaciones soportadas son según su precedencia (de mayor a menor, las que están al mismo nivel se realizan en el orden en que aparezcan):

1. Paréntesis “()” (para agrupación, ej.: (4 + 5) * 2).
2. Potencia “**”.
3. Multiplicación “*”, división “/”, división entera “//” y módulo “%”.
4. Suma “+”, resta “-”.
5. *Shift* [36] a la derecha “>>”, *shift* a la izquierda “<<”.
6. *And* binario “&”.
7. *Xor* binario “^”.
8. *Or* binario “|”.
9. Comparaciones (>, <, <=, >=, ==, !=, in , not in, is, is not).

[36] En la rama de la computación comúnmente se usa para indicar corrimiento de bits

10. Negación lógica “not”.
11. *And* lógico ‘and’.
12. *Or* lógico ‘or’.
13. Asignación ‘=’.

Estas operaciones no la soportan todos los tipos de datos. En Tabla 1 se muestra una tabla con los tipos de datos y los operadores que soportan

Tabla 1 Operaciones soportadas por cada tipo de dato datos

Datos\Operaciones	+	-	*	/	//	%	<< >>	**	&	^	|
enteros	Si	Si	Si	Si	Si	Si	Si	Si	Si	Si	Si
float	Si	Si	Si	Si	Si	Si	no	Si	No	No	No
boolean	Si	Si	Si	Si	Si	Si	Si	Si	Si	Si	Si
strings	Si	No	Si	No	No	No	No	No	No	No	No
arreglos	Si	No	Si	No	No	No	No	No	No	No	No
diccionarios	No	No	No	No	No	No	No	No	No	No	No
vistas de datos	Si	No	No	No	No	No	No	No	No	No	No
componentes del modelo	No	No	No	No	No	No	No	No	No	No	No
esquemas	No	No	No	No	No	No	No	No	No	No	No
workflows	No	No	No	No	No	No	No	No	No	No	No

Operadores específicos del dominio

Además de los operadores mencionados, YADSL posee otros operadores que son más cercanos al dominio donde se desenvuelve. El más usado probablemente sea el operador flecha (“`->, <-, ~>, <~`”). Este operador solo es aplicable a componentes del modelo extendido. Las flechas (`->, <-`) son para realizar conexiones entre los componentes de la siguiente manera:

```
In[0]: a -> b
Out[0]: True
```

O de una manera más general permite conexiones entre varios juegos de componentes como sigue:

```
a, b, c <- d,e,f,g
```

donde en este caso los components $x\epsilon\ X, X\{a, b, c\}$ son conectados cada uno con cada uno de los componentes $y\epsilon\ Y, Y\{d, e, f, g\}$, o sea, $\forall x\epsilon\ X, \forall y\epsilon\ Y\ \ x \longleftarrow y$, donde esta conexión cumple con los requisitos planteados en la página 13, en el caso que los elementos de X, Y sean componentes del tipo básico definido en el modelo original. Si existieran componentes del tipo *workflow* las conexiones son diferentes (véase Workflows).

Es importante remarcar que este operador **crea** la conexión en caso de no existir ninguna, y **si existe no la modifica**, además siempre devuelve si se pudo realizar la conexión o no.

A la par del operador de creación de conexiones, existe un operador para desactivar las conexiones (**~>, <~**), este tiene el mismo comportamiento que su homólogo, o sea, si existe la conexión la desactiva, sino no tiene acción ninguna.

También es posible preguntar si alguna conexión existe añadiendo el operador '?' al final de la sentencia de la siguiente manera:

```
In[0]: a, b, c <- d,e,f,g?
Out[0]: True
```

Este operador, si se le pregunta por varias conexiones al mismo tiempo retorna verdadero si alguna de las conexiones existe, falso en cualquier otro caso. Este comportamiento se puede definir como una función por ramas:

$$C(X\{x_1, x_2, \dots, x_n\}, arrow(\longrightarrow \mid \longleftarrow), Y\{y_1, y_2, \dots, y_n\}) = \begin{cases} True, & \exists x\epsilon\ X, \exists y\epsilon\ Y \mid\ x\ arrow\ y \\ False, en\ otro\ caso \end{cases}$$

Estas conexiones también se pueden eliminar con una sentencia parecida a la de crearlas, excepto que con la palabra reservada '*del*' al principio. Esta instrucción no se puede realizar con listas de componentes, esta decisión está basada en seguridad para el programador, de tal manera que el mismo especifique las conexiones que desea eliminar, ya que estas se

borran completamente, devuelve *True* si se pudo completar satisfactoriamente, *False* en caso contrario.

La instrucción sería de la siguiente manera:

```
In[0]: del a -> b
Out[0]: True
```

Los operadores de conexión más avanzados son los basados en la teoría de las redes de Petri, estos son los de *join* y *split*, que a su vez cada uno puede ser del tipo *and*, *or*, o x*or*. Estos operadores definen el comportamiento de las salidas y entradas a los componentes de una manera más dinámica. El operador *split* define el comportamiento de las salidas que generan los componentes, por ejemplo la instrucción: '*component split and'* programa las salidas que genere *component* de manera simultanea, este es el comportamiento por defecto del operador '`->`', pero se puede variar este comportamiento modificando el tipo, por ejemplo '*component split xor component1 if a> b else: component2 '* lo que prepara el modelo para cada vez que un componente realice una salida de datos, se ejecuta el código asociado decidiendo a cuál de los componentes se le pasará la información. A su vez la instrucción '*component split or'* le envía la tarea a procesar al componente que no esté ocupado en el momento, pudiendo ser varios.

La instrucción *join* presenta características similares, lo que define como el componente va a recibir las entradas. La instrucción '*component join or'* también coincide con la instrucción básica de conexión, esta admite todas las entradas sobre *component*, si esta procesando en caso de llegar alguna, esta pasa a una cola para ser procesada después. Las más interesantes del tipo *join* son las *and* y *xor*. La instrucción *join and* espera a tener al menos una entrada de cada componente conectado y la *xor* decide si existen varias entradas simultáneas cual ejecutar mientras la otra es descartada.

Otro par de operadores presentes en YADSL son los operadores de activación y desactivación de componentes, estos son '@' y '~' respectivamente. Estos junto con el operador que devuelve el estado de activación '?' de cada componente, son muy útiles para realizar activación o desactivación de componentes entre flujos de ejecución. Un ejemplo práctico de su uso es el siguiente:

```
inp = csvinput('input')
```

```
if inp?:
      process(inp, task)
      print('componente activo', inp, 'se desactivará')
      ~inp
if inp?:
      print('activo', inp)
else:
      @inp
      print('No activo, se activará el componente', inp)
```

2.6 Esquemas

Los esquemas son una característica que pudiera variar de un dominio a otro según los requerimientos. En este caso particular los esquemas son necesarios ya que el modelo está diseñado para trabajar con datos de forma tabular, o sea columnas nombradas y filas con valores como lo muestra la Figura 2.1.

Compound	Dmax	Smin	Do	LogP	CLogP	PAMPA	P-Rats	P-RatsEV	MDCK	P-Human
ABACAVIR SULFATE	300	77	0.02	0.22	0.58					
ACETAZOLAMIDE	250	0.8	1.25	0.14	-1.13					
ACETYLSALICYLIC ACID	500	3.33	0.6	1.18	1.02	3.8				
ACYCLOVIR	4	2.5	0.32	-1.59	-2.42	0	-0.42	1.08	0.26	
ALBENDAZOLE	400	0.01	160	2.8	3.461					
ALLOPURINOL	300	0.57	2.1	0.32	0.63					
AMILORIDE HYDROCHLORIDE	5	50	0.0004	-0.71	-2.22			15.3	3.8	1.6
AMIODARONE	400	0.7	2.3	7.12	8.94					
AMITRIPTYLINE	75	0.9	0.33	4.42	4.85				0.474	
AMLODIPINE	5	1.97	0.01	0.29	3.43					0.33
AMODIAQUINE	200	3.2	0.25	3.99	4.94					
AMOXICILLIN	500	4	0.5	-0.58	-1.87	1.5	12	6.4	0.24	0.3
CLAVULANIC ACID	125	300	0.002	-1.98	-1.2					
ANASTROZOLE	1	0.5	0.008	4.1	1.48					
ANTYPIRINE			0.2	1.01	0.204	20.1	96	39.74	31.1	5.6
ARTEMETHER	20	1	0.08	3.51	3.05					
ARTESUNATE	50	0.1	2	3.04	2.93					
ATAZANAVIR	300	1	1.2	4.54	5.92		47			
ATENOLOL	100	24.8	0.02	0.5	-0.11	0.1	6	10.7	1.5	0.2

Figura 2.1 Ejemplo de datos usados por el modelo

Los esquemas definen dada una entrada, o sea una vista de datos, que columnas van a ser usadas para el procesamiento y como se van a tratar, como dato continuo, nominal, etc.

La sintaxis para definir esquemas es sencilla, por ejemplo:

```
In[0]:schemas (test:name='PAMPA' type='Continuous' treated='Permeability',
name='Smin' type='Continuous' treated='Solubility')
```

La semántica de esta instrucción es, seguido de la palabra reservada *schemas* viene el nombre del esquema en general, que sería la variable a la que se le asignaría, esta variable puede ser referenciada más adelante. Luego viene la palabra reservada '*name*' indicando el nombre de

la columna de interés, en el caso anterior sería ‘PAMPA’ y ‘Smin’, seguido por ‘type’ que indica el tipo de la columna, o sea si se va a procesar esos datos como ‘Continuous’ o ‘Nominal’ (para el caso en análisis son los únicos disponibles), y finalmente la clasificación a usar (‘Permeability’, ‘Solubility’, ‘Bioavailability’, ‘Tag’,‘BCS Class’, ‘Descriptor’). Pueden haber varios esquemas seguidos separados por ‘;’ de la siguiente manera:

```
In[0]:schemas (test:name='PAMPA' type='Continuous' treated='Permeability'
; test1:name=’Smin’ type=’Continuous’ treated=’Solubility’)
```

Los esquemas mínimo deben tener la definición de una columna con la cual trabajar (la mayoría de los componentes del modelo original necesitan un esquema para procesar los datos).

2.7 Sentencias básicas

En su núcleo, la sintaxis de YADSL, está compuesta por sentencias y expresiones. Las expresiones procesan datos y están embebidas en sentencias. El código de las sentencias es la mayor porción de la operación de un programa y usan directamente expresiones para realizar las acciones para las que son destinadas. Estas usualmente son de asignación, llamadas a funciones, condicionales, sentencias de ciclos, etc. A continuación, se explican las sentencias presentes en YADSL.

Sentencias de asignación

Esta sentencia ya ha sido mencionada antes, pero es necesario definirla propiamente ya que es una de las más usada por los programadores, por el hecho que permite la modificación de las variables de un programa, siendo esta su función principal.

En YADSL no es necesario declarar el tipo de las variables por su tipado dinámico. Si la variable no existe, se crea, si existe se actualiza su valor, por ejemplo:

```
In[0]: a = 5
Out[0]: 5
In[1]: a
Out[1]: 5
In[2]: a = “testing”
Out[2]: testing
In[3] : a
Out[3]: testing
```

La asignación, YADSL la realiza de dos maneras, cuando el valor a asignar es un elemento inmutable, como los números y cadenas, se realiza por valor, o sea, se realiza una copia del valor y se le asigna a la variable. En el caso de que el elemento sea mutable como arreglos, diccionarios, componentes del modelo, etc., esta operación se realiza por referencia, o sea, a la variable se le asigna un puntero hacia donde está el valor real del elemento. Esto es importante a tener en cuenta ya que puede llevar a situaciones indeseadas como, por ejemplo:

```
In[0]: a = [1,2,3,4]
Out[0]: [1,2,3,4]
In[1]: b = a
Out[1]: [1,2,3,4]
In[2]: append(a, 5)
Out[2]: [1,2,3,4,5]
In[3]: b
Out[3]: [1,2,3,4,5]
```

Como se puede observar el valor de ‘b’ se modificó como resultado de la llamada a la función del tipo *built-in* que realiza una concatenación del arreglo con el valor.

Como la asignación es la última operación que se hace, es posible realizar operaciones más complejas como:

```
In[0]: a = ((2+5)*3  <<  2)  // 5
Out[0]: 16
In[1]: a
Out[1]: 16
```

YADSL soporta varios tipos de asignación, a continuación se muestra una lista con todas las opciones:

1. Asignación clásica ‘=’.
2. Aumentada ‘+=’.
3. Restada ‘-=’.
4. Multiplicada ‘*=’
5. Dividida ‘/=’.
6. Resto de la división ‘%=’.
7. *And* lógico y luego asignación‘&=’.
8. *Or* lógico y luego asignación ‘|=’.
9. *Xor* lógico y luego asignación ’^=’.
10. Corrimiento de bits a la izquierda ‘<<=’.

11. Corrimiento de bits a la derecha '>>='.
12. Asignación elevada a la potencia '**='.
13. Asignación de la división entera '//='.

Estas asignaciones realizan primero la operación que presentan entre el valor del elemento a la izquierda con el valor del elemento a la derecha y luego asignan el resultado al elemento de la izquierda sobrescribiendo su valor, lo que significa que:

```
In[0]: a = ((2+5)*3  <<  2)  // 5
Out[0]: 16
In[1] : a += 4
Out[1]: 20
In[2]: a
Out[2]: 20
```

Sentencia if-elif-else

La principal y única sentencia condicional de YADSL es la sentencia *if-elif-else*. La forma de ejecución de esta sentencia es la siguiente:

1. Se evalúa la condición de la parte *if.*
2. Si es verdadero se ejecuta el código dentro del bloque *if* y se termina la instrucción.
3. Si es falso, se va al primer *elif*, se evalúa la condición y si resulta verdadera se ejecuta el código asociado a este bloque y se termina la instrucción, de lo contrario se continúa al siguiente *elif.*
4. Si no existe *elif*, o ninguno resultó verdadero se ejecuta el código asociado al bloque del *else* si existe. Luego termina la instrucción.

El caso básico de esta sentencia es el siguiente:

```
if 4 < 2:
      print('true')
```

Como se puede observar las instrucciones *elif* o *else* son opcionales. La parte *elif* se puede repetir cuantas veces sea necesario, pero solo puede existir una cláusula *else.* Esta sentencia en general se puede anidar, o sea, declarar un bloque *if-elif-else* dentro de un *if*, o un *elif*, o un *else* indistintamente.

A diferencia de otros lenguajes, como Java y C++, que presentan una sentencia '*switch*', YADSL no cuenta con esta sentencia, ya que este se puede expresar como una sentencia *if-elif-else* con varios *elif* seguidos. Este caso se puede ver como sigue:

```
a = 4
if  a == 0:
      print('case 0')
elif a == 1:
      print('case 1')
elif a == 2:
      print('case 2')
elif a == 3:
      print('case 3')
else:
      print('case 4')
```

Es necesario aclarar que YADSL no cuenta con ámbitos en la sentencia *if-elif-else,* lo que significa que las variables que se declaren dentro de uno de estos bloques, son visibles fuera, por ejemplo el siguiente bloque de instrucciones no genera ningún error:

```
a = 4
if  a == 0:
      print('a is 0')
      b = 5
print(b)
```

Sentencia if-else

La sentencia *if-else* de YADSL presenta un comportamiento similar al operador ternario de Java ('?'). Esta sentencia también es del tipo condicional. Su estructura es la siguiente:

expresión *if* condición *else* expresión. Un ejemplo práctico de esta sentencia es el siguiente:

```
In[0]: a = 4 if 4 < 5 else [1,2,3,4]
Out[0]: 4
```

Aunque es una sentencia sencilla es muy usada por programadores, por lo cual se decidió incluir en YADSL.

Sentencias iterativas

Las sentencias iterativas son básicas en casi cualquier lenguaje de programación, ya que permiten la repetición de determinadas acciones de acuerdo a ciertas condiciones. YADSL cuenta con dos construcciones repetitivas, *while* y *for.*

Sentencia while

La instrucción *while* es la más general dentro de las instrucciones de ciclos. En términos simples, esta instrucción repite un bloque de sentencias con el nivel de indentación correspondiente, mientras la condición sea verdadera. Cuando esta condición se convierta en falso el control de la ejecución sigue con la siguiente sentencia que no pertenezca al bloque del *while.* Si en la primera iteración la condición es falsa nunca se ejecuta el código asociado al bloque del *while,* en estos casos es posible añadir un cláusula *else,* que se ejecuta en estos casos, por ejemplo:

```
a = 6
while a != 0:
      if a % 2 == 0:
            print(a, 'Even')
      a = a - 1
else:
      print('False')
```

Este segment de código genera como salida:

```
6, Even
4, Even
2, Even
```

Sin embargo si el valor de la variable *a* fuera de 0 el resultado sería *False.*

Sentencia For

El ciclo *for,* semejante al *for* de Python (Guttag, 2013) itera sobre secuencias de elementos, ya sean arreglos o *strings*. Esta instrucción empieza con una sección de variables que se usarán para asignar valores extraídos del elemento sobre el cual se iterará. Luego es necesario definir cuál será el elemento a iterar. El ciclo se realiza mientras queden elementos por iterar en el iterador. Por ejemplo, el resultado de este ciclo sería:

```
for i in range(3):
```

```
      print(i)
0
1
2
```

YADSL cuando ejecuta este *for*, asigna cada valor que se encuentre en el resultado de *range(3)* que es un arreglo como el siguiente *[0,1,2]*, uno a la vez y realiza la ejecución de ese ciclo en particular. Durante la iteración **el tamaño del elemento sobre el que se itera no puede variar**.

Además es posible, al igual que la sentencia *while*, añadir una cláusula *else* para realizar cierta acción si nunca se entra al *for*.

Break, continue, pass

Las sentencias *break*, *continue* y *pass* son sentencias de control del flujo de ejecución en general. Las dos primeras tienen sentido solo cuando están dentro de algún ciclo debido a que están destinadas a alterar el mecanismo natural de la ejecución de algún ciclo, si se utiliza fuera de estos, no genera error sintáctico solo que no realiza acción alguna. La sentencia *break* rompe con el ciclo actual más interno mientras que la sentencia *continue* salta al inicio del ciclo más interno, en el caso de ciclos anidados. La instrucción *pass* no tiene consecuencia ninguna, solo existe con el objetivo de poder definir un bloque de sentencias cuando no se quiere que tenga ninguna, por ejemplo en Java, este *if* es válido *if (5 < 2){}*, debido a que las {} definen un bloque de acciones, mientras que en YADSL esto no es así, el homólogo de la instrucción anterior en YADSL sería:

```
if 5 < 2:
      pass
```

Sentencia setup

La instrucción ***setup*** está diseñada para poder cambiar la configuración de cualquier componente de una manera sencilla. Usualmente esta acción se realiza mediante la interfaz visual, por lo que es necesario que el investigador vaya componente a componente abriendo su interfaz visual y cambiando los parámetros manualmente. La ventaja que supone esta instrucción es que elimina este trabajo y se puede automatizar esta tarea de una manera

sencilla entre cada flujo de trabajo. Esta sentencia permite cambiar la configuración de cada componente según se desee, un ejemplo es el siguiente:

```
In[0]:inp = csvinput(label='input')
Out[0]:input(14250546)
In[1]:setup inp: vconfig={'source':"D:\Documentos\bcsmap-proj\input-
files\test-input-no-smiles.csv", 'vschema': test }
Out[1]: True
```

Su uso está muy ligado con el uso de los tipos de datos básicos del lenguaje. Como se puede observar en el ejemplo, se usan principalmente diccionarios, aunque esto no es un requerimiento. Como cada componente exporta variables para su uso por los analistas, la instrucción ***setup***, se puede usar para cambiar el valor de estas variables, por ejemplo

```
In[0]:setup inp: vlabel='csvinputitem'
Out[0]: True
```

Sentencia del

La sentencia ***del*** sencillamente elimina una variable del intérprete. Aunque es opcional y en la mayoría de los casos no es muy práctica se añadió al lenguaje ya que al ser interpretado generalmente no se hace un uso eficiente de la memoria, por lo que se brinda como una alternativa para equipos con pocos recursos. El único caso donde es útil es cuando se desea eliminar por completo un componente del modelo ya que estos si están presentes en el modelo y no son necesarios para el analista, añaden un procesamiento inecesario al flujo. Su uso es muy sencillo como se muestra en el siguiente ejemplo:

```
In[0]:del inp
Out[0]: True
```

Sentencia start

La sentencia ***start*** es la más importante de todas ya que hecha a andar todo el modelo de análisis planteado hasta el momento en que se va a ejecutar esta sentencia. Esta sentencia tiene dos usos fundamentales, el primero es el arranque global, y el otro es el detonado local.

Para realizar un detonado global, sencillamente se escribe la sentencia sola y nada más, mientras que para ejecutar el modelo partiendo por un componente en específico se escribe la sentencia seguida por los componentes a iniciar separados por coma.

Un ejemplo del detonado local sería:

```
inp = csvinput(label='input')
rad = radviz(label='viz')
fil = filters(label='filters')
setup inp: vconfig={'source':"D:\Documentos\bcsmap-proj\input-files\test-
input-no-smiles.csv", 'vschema': test }
setup fil: vconfig = {'filters':{'filter1':{'schema': test,
'Property':'PAMPA', 'Operation':'lt', 'Value':2}}}
inp -> fil -> rad
show inp in x=0 y=0, fil in x=20 y=100, rad in x =150 y = 50
start inp, fil
```

Lo más importante a tener en cuenta al usar esta sentencia es que es bloqueante debido a la naturaleza asíncrona del modelo, mismo principio de la red de Petri. Cuando se ejecuta esta instrucción el intérprete frena su ejecución hasta que el modelo complete su ejecución y luego continúa las instrucciones subsiguientes.

Otro aspecto a remarcar es que inmediatamente después del uso de esta instrucción, queda disponible en una variable llamada '_ _views_ _' que contiene todas las vistas de datos generadas durante la ejecución del modelo. Esta variable es un diccionario donde por cada una de sus llaves(componentes del modelo) contiene un arreglo con cada una de las vistas generadas por esa llave, un uso de esta variable sería:

```
#some code before with start instruction
for i in __views__[fil]:
    bcs_class(i, 'BCS-Kasim', 'imgs/BCS-Kasim_for_view_' + string(i) +
'.jpg')
    viz_radviz(i, ['PAMPA', 'CLogP', 'Dmax', 'Smin'], 'BCS-
Lindenberg','imgs/radviz_for_view_' + string(i) + '.jpg')
```

Este ejemplo recorre cada una de las vistas de datos generadas por el componente `fil` y guarda sus resultados usando dos funciones *built-in*.

Sentencia publish

Aunque el uso de los componentes se recomienda que sea mediante el modelo, existe la posibilidad de usar un componente para que procese una vista de datos específica que no ha llegado a él o nunca va a llegar. En este caso los resultados nunca pasan por el modelo y para que el modelo los obtenga se usa esta instrucción. Su uso es sencillo, después de la palabra reservada ***publish*** se pone seguido todas las vistas de datos deseadas cada una de ellas separadas por comas.

El uso correcto de esta instrucción es dentro de un componente del tipo ***workflow*** aunque no es obligatorio. Como dentro de un componente del tipo *workflow* puede existir todo un flujo complejo, esta instrucción permite a una vista de datos que se genere dentro del componente, pasar al siguiente componente que esté conectado con él.

2.8 Sentencias de visualización

Las sentencias de visualización están enfocadas principalmente en el manejo de la interfaz, o sea en la manipulación visual de la posición de los componentes y la disposición que presentan en el área de trabajo.

Show

La principal sentencia es *show* la cual muestra los componentes especificados en la posición deseada, por ejemplo:

```
show inp in x=0 y=0, fil in x=20 y=100, rad in x =150 y = 50
```

Esta instrucción muestra el componente *inp* en la ubicación *x = 0* y *y =0*, además de los otros dos componentes en sus respectivas ubicaciones. En caso de existir conexiones entre los componentes, se muestran las conexiones entre los componentes visibles.

Esta sentencia enlaza la parte lógica de un componente con su representación visual, es importante decir que aunque un componente no posee su contraparte visual si es capaz de realizar los procesamientos para el cual está diseñado y se le puede añadir en cualquier instante. Este comportamiento flexibiliza el modelo y separa por completo la lógica de ejecución de la interfaz gráfica.

Showui

La instrucción ***showui*** notifica al modelo de la solicitud del usuario de usar la interfaz de usuario de un componente en específico, por ende el modelo notifica al componente para que la muestre. Esta instrucción no es muy práctica y está destinada a los usuarios que deseen usar el teclado como método de entrada alternativo. Su uso es muy sencillo, luego de poner la palabra reservada ***showui*** viene seguido el identificador del componente al cual se le solicita que muestre su interfaz gráfica, como se muestra en Figura 2.2:

```
In[0]: show inp
```

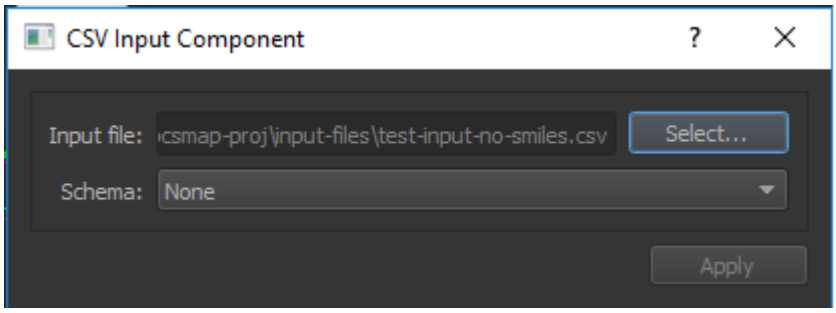


Figura 2.2 Interfaz de usuario del componente csvinput

Su uso es recomendado solo si el componente ya posee una representación gráfica en el modelo, ya que sino se lanza una excepción.

Hide

La instrucción hide realiza la acción contraria a la instrucción ***show***, esta sencillamente oculta la representación visual de un componente. Su uso es igual a la instrucción ***showui***.

Visualize

La sentencia ***visualize***, se diseño para que tenga un uso similar a ***showui***, con la diferencia que está destinada a componentes de visualización. La utilidad de esta sentencia es que permite realizar una comunicación con la interfaz gráfica de un componente, pero con la ventaja que permite el pase de parámetros, los cuales son usados por el componente para realizar la representación visual de los datos deseados.

Layout

Los *layouts* en YADSL no son más que una forma de reorganización de la prepresentación gráfica de los componentes. Esta instrucción realiza un algoritmo de reordenamiento, el algoritmo deseado se pasa por parámetro junto con los parámetros del algoritmo. Algunos de estos algoritmos pueden ser ‘circular’, ‘organic’, entre otros.

2.9 Sentencia *import y from-import*

Una gran potencialidad que posee YADSL es su capacidad de importar código de varios *scripts*. Este mecanismo es muy similar al mecanismo de importe que posee Python. En YADSL se pueden importar *workflows*, funciones y esquemas que estén en otro *script*.

YADSL posee varias formas de importar código. La más común es el importe nombrado, o sea, se especifica el código que se desea importar escribiendo el nombre del *script* al que pertenece y luego renombrándolo con la palabra reservada *'as'.*

Por ejemplo:

```
from schemas.schema_gamma import schema_gamma as sg
```

Esta instrucción carga el archivo *schema_gamma* que se encuentra en la carpeta *schemas* e import el esquema *schema_gamma* usando el nombre *sg,* de tal manera que para referirse a este esquema, luego *sg* sería el nombre a usar. Esta vía evita conflictos en nombres iguales, ya que puede pasar que varios *scripts* usen nombres de variables de la misma manera. La instrucción '*as*' es opcional, o sea, se puede o no poner, solamente es útil cuando pueda existir conflictos entre los identificadores o por comodidad para acortar un identificador muy extenso. Esta variante solo es aplicable cuando el código deseado se encuentra en la dirección destinada para esto, la cual es `'app/control/scripts'`. Por supuesto esto es algo molesto, por lo que YADSL posee otra forma, este puede importar desde una dirección física real, o sea es posible escribir:

```
from "C:\mycode.yl" import testcode as tc
```

Esta variante es muy cómoda ya que permite importar código desde cualquier lugar, pero es necesario tener cuidado al usarlo y principalmente al compartir archivos de código ya que las direcciones pueden no coincidir. Esta variante se recomienda para su uso local, o sea, siempre sobre la misma computadora.

Además de estas, se puede importar de la siguiente manera:

```
import mycode
import mycode.examples.example1
from mycode.examples import example1
from mycode.examples import *
from mycode import **
```

El operador '*****' y '******' cuando se usa en las instrucciones de importe cambian de semántica, el operador '*****' importa todo lo que se encuentre en la dirección especificada que sea archivos con la extensión '*.yl*', mientras que el operador '******' importa todo recursivamente, importa los archivos en el directorio especificado así como todos los archivos en cualquier subcarpeta existente.

2.10 Workflows

Los *workflows* se pueden considerar como uno de los tipos de datos más importante de YADSL. Este tipo de dato sería la representación de un componente en el meta-modelo. Este tipo de dato puede contener dentro todo un flujo de ejecución. Puede recibir entradas como los componentes básicos y producir salidas de la misma manera. Una instancia de este tipo de dato se define escribiendo la palabra reservada '*workflow*' seguido del nombre de la instancia, luego dos puntos, un cambio de línea y un bloque indentado. Por ejemplo:

```
workflow example:
      schemas (test:name='PAMPA' type='Continuous' treated='Permeability')
      inputs inp = csvinput(label='input')
      outputs fil = filters(label='filters')
      setup inp: vconfig = {'source':"D:\Documentos\bcsmap-proj\input-
files\test-input-no-smiles.csv", 'vschema': test}
      setup fil: vconfig = {'filters': {'filter1': {'schema': test,
'Property':'PAMPA', 'Operation':'lt', 'Value':2}}}
      inp -> fil
      start
```

Dentro de este tipo de dato se puede usar cualquier componente del modelo de (Mollineda, 2017) así como definir funciones o cualquier otra sentencia incluyendo la definición de otros *workflows*, o sea se pueden anidar *workflows*.

Este tipo de dato se puede tratar como cualquier otro componente del modelo original, por ejemplo, este puede recibir entradas de otros componentes usando la sentencia vista anteriormente:

```
another_component1 -> example -> another_component2
```

Como se muestra en el ejemplo este tipo de dato soporta todo un flujo de trabajo pero esto no es necesario. Puede contener sentencias básicas como *if* o *for*, e incluso llamar a funciones *built-in*.

La definición de un flujo de trabajo puede comenzar con la definición de los esquemas que se van a usar internamente, además pueden especificarse los componentes que van a recibir entradas o generar salidas de datos con la palabra reservada *inputs* y *outputs.* Si existen componentes de entrada, cualquier vista de dato que se reciba se le proporcionará como entrada a todos los componentes declarados, en caso de no existir *inputs*, cualquier valor recivido será propagado a cualquier componente conectado al otro extremo.

Cualquier *output* que produzcan los componentes de salida, en caso de existir, también será propagado hacia adelante.

Dentro de un flujo de trabajo no tiene mucho sentido declarar componentes de visualización, aunque no está prohibido, ya que estos no pueden ser vistos por el investigador. Para estos tipos de componentes, es más común y práctico que sean declarados al mismo nivel del *script* que se está programando.

2.11 Funciones built-in

YADLS cuenta con un cierto número de funciones nativas del lenguaje. En epígrafes anteriores se ha visto las funciones *print, range, length* y *append,* pero estas no son las únicas, además de ellas se encuentran *remove, pop, extend, string, float* e *int.* Aunque no son muchas, esto no representa un gran problema ya que YADSL es fácilmente escalable como se mencionó en la página 39. La forma más sencilla es añadir *scripts* de Python a la dirección `"/app/control/libs"`, y el mismo lenguaje se encarga de cargarlos y usarlos como sea necesario.

Las funciones desarrolladas por el momento son:

- ***print(*args)*** Imprime en la consola el resultado de una operación.
- ***range(init, stop, step)*** Genera un arreglo de enteros empezando por el valor init hasta el valor stop aumentando cada vez por step. Útil para recorrer arreglos.
- ***length(arg)*** Retorna la cantidad de elementos que posee un arreglo, o la cantidad de caracteres de un string, o la cantidad de llaves de un diccionario según el tipo de args.
- ***append(args, val)*** Añade val al arreglo args.
- ***remove(args, index)*** Elimina el elemento que se encuentra en la posición index del arreglo args.
- ***pop(args)*** Elimina el último elemento insertado en el arreglo args. Útil para tratar una lista como pila.
- ***extend(args, value)*** Concatena el arreglo args con el arreglo value.

- ***string(val), float(val), int(val)*** Realiza un *cast* [37] de val a string, float e int respectivamente, siempre que sean compatibles los datos.
- ***process(component, dataview)*** Realiza la ejecución de un componente sobre la vista deseada.
- Otras funciones de visualización (más adelante).

Funciones de visualización

Las funciones más importantes dentro de las incluidas en YADSL se pudiera considerar que son las de visualización. Estas también son funciones *built-in* y son las que ayudan al investigador a aplicar su conocimiento para realizar descubrimientos sobre los resultados obtenidos.

Como estaba concebido en el modelo de (Mollineda, 2017) los componentes de visualización están diseñados para interactuar con el investigador y mostrarle los resultados, pero este proceso es lento, ya que para poder comparar resultados, el investigador debe generar todo un flujo de trabajo, ejecutarlo, analizar los resultados, luego repetir todo este proceso usando varias alternativas anotando los resultados y luego comparar todos estos.

Con estas funciones de visualización y la capacidad de realizar varios flujos de trabajo de YADSL en un *script*, el investigador puede diseñar los flujos de trabajos consecutivos en el *script* y cuando termina cada uno guardar las imágenes resultantes así como sus resultados, de manera que luego puede comparar los resultados en su conjunto. Estas funciones pueden suplantar por completo a los componentes de visualización si se usan correctamente.

Las funciones desarrolladas son por el momento las que están desarrolladas por el modelo de (Mollineda, 2017) pero se le pueden adicionar más. Estas son:

- bcs_class, muestra la descomposición de los fármacos contenidos en ella de acuerdo a su clasificación BCS, en caso de hallarse disponible.
- bcs_agree, una descripción estadística de los valores de las propiedades que se encuentran contenidas en la vista pasada por parámetro.

[37] Conversión de tipo

- viz_scatter, realiza un ploteo esparcido de los datos.
- viz_radviz, aplica un algoritmo de visualización de datos multivariados.

Persistencia de las vistas de datos

La mayoría de los lenguajes de propósito general presentan mecanismos de persistencia de datos y manejo de archivos, pero debido al corte específico de YADSL, sólo soporta con ciertas acciones de esta índole. YADSL soporta la persistencia de las vistas de datos sobre una base de datos local, y por supuesto permite realizar cargas sobre esta. Puede estar ubicada en cualquier dirección física de la computadora siempre que sea válida y cuente con los permisos de escritura apropiados. Estas acciones se puden realizar usando las funciones ***save_data_view*** y ***load_data_view***. Las cuales se encargan de la persistencia de las vistas de datos, guardándolas y cargándolas según desee el usuario.

2.12 Definición de funciones

La definición de funciones en YADSL es muy similar a como se realiza esto en Python. Cualquier definición empieza con la palabra reservada *'def'* seguido del nombre de la función que debe ser un identificador válido, luego, entre paréntesis, la declaración de los parámetros con su valor por defecto si tienen, estos valores por defecto o lo poseen todos los parámetros o ninguno, seguido por un bloque indentado con el código asociado a la función.

En ejemplo práctico es el siguiente:

```
In[0]: def a(b=4, c=2):
            print('El valor de b es:', b, ‘, El valor de c es:’, c)
Out[0]:
In[1]: a()
Out[1]: El valor de b es: 4, El valor de c es: 2
In[2]:a(5,7)
Out[2]: El valor de b es: 5, El valor de c es: 7
```

La declaración puede contener opcionalmente una anotación sobre el tipo de dato que se desea devolver aunque no es obligado devolver ese tipo de dato o puede no devolver ninguno. Se pueden declarar funciones recursivas incluso como cualquier otro lenguaje, el factorial por ejemplo quedaría de una manera sencilla usando la sentencia return (en la página 70) de la siguiente manera:

```
def factorial(n):
```

```
      if not n:
            return 1
      return n*factorial(n-1)
In[0]:factorial(5)
Out[0]:120
```

Sentencia return

Las funciones en YADSL pueden devolver valores, para esto existe la sentencia ***return***. Esta sentencia es ampliamente conocida ya que muchos lenguajes la soportan. Puede existir dentro de una función nada más y cuando se alcanza se termina la ejecución de la función. En YADSL cualquier función puede devolver cualquier tipo de dato, incluído los componentes del modelo, lo que flexibiliza el proceso de construcción del flujo de trabajo.

A continuación se muestran ejemplos de su uso:

```
In[0]:def prom(arr):
            sum = 0
            for i in arr:
                  sum += i
            return sum/length(arr)
            print(a([1,2,3,4,5]))
Out[0]: 3
```

En el ejemplo anterior se muestra el uso de la sentencia ***return*** en su manera más simple, devolviendo un valor numérico sencillo, calculado mediante la función prom, la cual calcula el promedio de los valores almacenados en un arreglo.

```
In[1]: def createarr():
            arr = []
            for i in range(10):
                  append(arr, i)
            return arr
      print(b)
Out[1]:[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

El caso anterior muestra el uso de la instrucción ***return*** pero ahora devolviendo un dato compuesto, en este caso un arreglo. Además se puede observar que la sentencia `print` no se llega a ejecutar.

```
In[2]: def component(d):
            if d:
```

```
                return csvinput('Because d was true')
            else:
                return filter('Because d was false')
        component(true)
Out[2]: csvinput(123456789)
```

Por último, se muestra la capacidad de devolver componentes del modelo, lo que permite cambiar la estructura del mismo flujo de trabajo según ciertas condiciones en corridas diferentes.

2.13 Mecanismo de comunicación con el modelo

Para poder hacer un uso correcto de YADSL como lenguaje es necesario comprender como este interactúa con el modelo. Como el modelo es un grafo que cumple con todas las características de una red de Petri, su ejecución es totalmente asincrónica, por lo que no tiene sentido intentar secuenciar la ejecución de los componentes. Por este motivo cuando el modelo se encuentra en ejecución es imposible realizarle ningún cambio, y cualquier instrucción subsiguiente espera por la finalización del modelo.

Se puede imaginar la interacción entre YADSL, el modelo y su interfaz gráfica como se muestra en la Figura 2.3**¡Error! No se encuentra el origen de la referencia.**.

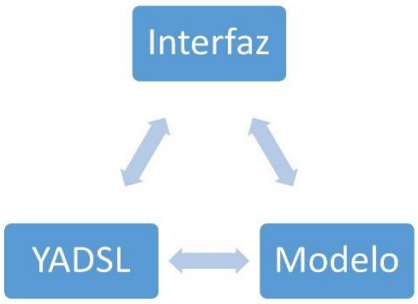


Figura 2.3 Mecanismo de interacción general

Este mecanismo permite la interacción directa desde la interfaz gráfica con el modelo y el lenguaje permitiendo además un canal de comunicación hacia atrás por parte de estos dos elementos. La comunicación entre la interfaz y el modelo se mantiene igual a la planteada por (Mollineda, 2017), debido a que no es necesario alterarla para insertar el lenguaje,

mientras que la comunicación del modelo y la interfaz con el lenguaje si es un aspecto nuevo y necesario de destacar.

Como YADSL es capaz de comunicarse con ambos, tanto el modelo, como la interfaz gráfica, puede actualizarlos mientras se ejecuta algún código en específico en el interior del lenguaje lo que permite al usuario realizar un seguimiento de la ejecución de las acciones que haya programado y chequear su correcta ejecución. Además, es comprensible que el lenguaje necesite un mecanismo de comunicación con el modelo ya que estos dos deben realizar tareas en conjunto y poseer cierta sincronización uno con el otro.

Como cada uno de estos elementos es independiente esta comunicación se realiza mediante el paso de mensajes, para no interferir en el ciclo de vida propio de cada uno, y que de no existir alguno por un determinado motivo, no se daña la ejecución, sino que los mensajes no llegan a nadie y por tanto no se realizaría esa acción. Este mecanismo permite una fácil inserción, modificación, eliminación o cambio de cualquiera de estos sin afectar al resto.

Conclusiones parciales

Debido a los paradigmas de programación incluidos en el lenguaje, así como toda su gama de instrucciones y el mecanismo de interacción del lenguaje con el modelo, YADSL parece ser suficientemente flexible y cómodo para un desarrollo rápido de aplicaciones específicas del dominio, además de aumentar las potencialidades del modelo permitiendo por ejemplo el cambio de configuración de los componente de una manera rápida, presentando la posibilidad de almacenar configuraciones antiguas para su reutilización y permitiendo cambios morfológicos de una manera sencilla.

CAPÍTULO 3. IMPLEMENTACIÓN DE YADSL EN BCSMAP

En este capítulo se discuten las características de la implementación desarrollada a partir de la definición del lenguaje propuesto en el CAPÍTULO 2 para el análisis de datos, como medio para probar la aplicabilidad de este sobre el meta-modelo.

En primer lugar, se analizan los actores involucrados y los casos de uso principales del lenguaje. Posteriormente se abordan las características y funcionalidades de las principales clases y estructuras de datos involucradas en el funcionamiento del intérprete desarrollado. Finalmente, se presenta un ejemplo de un caso de uso real de la aplicación.

3.1. Análisis de actores y casos de uso

Como la implementación de YADSL ocurre sobre una aplicación previamente elaborada, el lenguaje añade ciertos casos de usos además de los que ya existían en la aplicación (Mollineda, 2017). Debido al propósito específico de la aplicación, se puede considerar un único actor para los casos de uso de esta: el analista o especialista biofarmacéutico. Se asume que este tiene un conocimiento extenso del dominio del problema, es capaz de interpretar la información representada en los valores de las propiedades de los fármacos y está familiarizado con el uso de herramientas informáticas para el análisis (Mollineda, 2017). Este puede poseer conocimientos de programación o no, pero si los posee podrá ser capaz de usar el lenguaje y realizar acciones más complejas y avanzadas. Los casos de uso (**¡Error! No se encuentra el origen de la referencia.**) para la aplicación se mantienen igual a las interacciones que fueron descritas en (Mollineda, 2017), con la adición de los elementos destinados al manejo del lenguaje y la interacción con el usuario de este. En este último caso se encuentran contenidos los casos correspondientes a **crear** un nuevo *script* de análisis, **salvar** un *script* y **cargar** un *script* de análisis previamente guardado. Evidentemente, estas interacciones básicas de manejo de archivos son necesarias para persistir el código desarrollado. En el caso relacionado con la ejecución, se puede subdividir en dos casos más

específicos: iniciar la **ejecución global** del *script* y **realizar un subconjunto de instrucciones** sobre el modelo cargado.

3.2. Diseño y comportamiento de la interfaz de usuario

En la primera versión de BCSMap, el diseño de la interfaz de usuario consiste de una ventana principal cuyos elementos principales son una paleta de componentes separados en tipos por pestañas, una consola de eventos y un área principal correspondiente a la escena de análisis (Mollineda, 2017). A este diseño, para su segunda versión incluyendo las características del lenguaje, se le añadió una consola interactiva que funciona como entrada de comandos y, además, el área principal se embebió dentro de un componente con pestañas donde la primera es el área de diseño seguido por pestañas de *scripts*. Además, se le añadió la posibilidad de cambiar entre dos temas, uno obscuro (Figura 3.1) y otro claro (Figura 3.2), destinados a ambientes con distintas intensidades de luz.

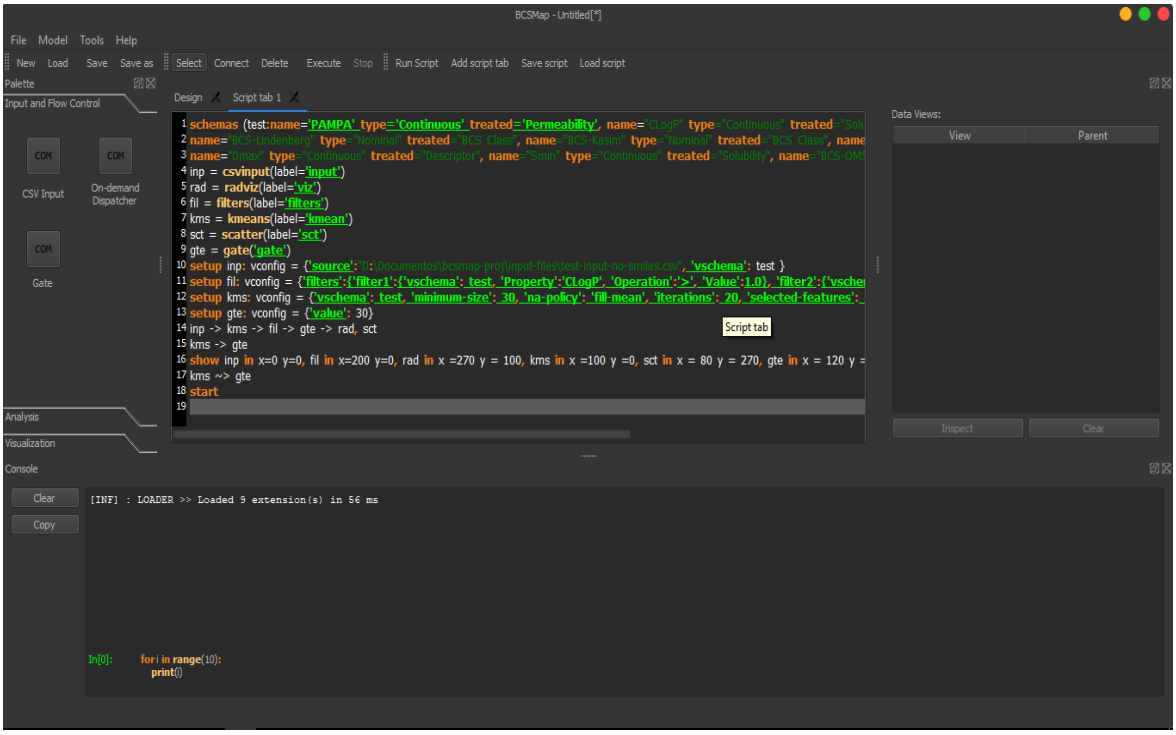


Figura 3.1 Interfaz estilo obscuro

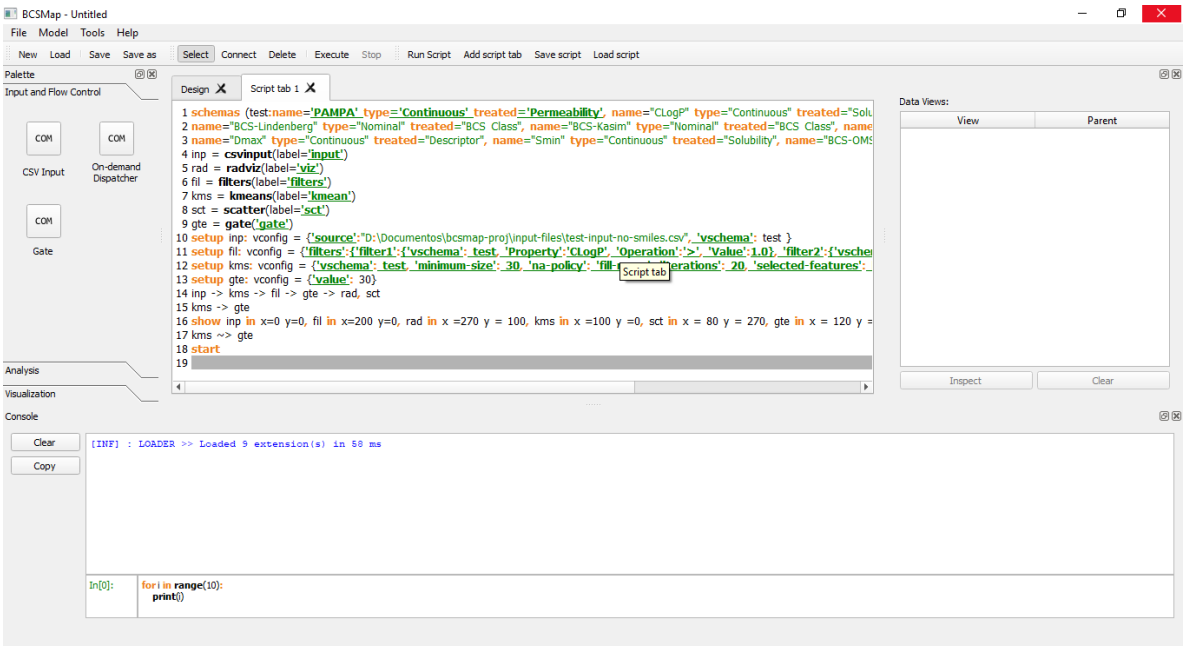


Figura 3.2 Interfaz estilo claro

Consola Interactiva

La consola interactiva es una de las más potentes herramientas que posee la aplicación. Está ubicada en la sección inferior de la **consola de registros** (Mollineda, 2017), esta, al igual que casi todos los componentes, puede ser ocultada o usada como una ventana flotante. La función primordial de este componente es realizar la interacción de YADSL con el usuario de una manera más sencilla y rápida que la otra vía que son lo *scripts*. Como las instrucciones que se escriban ahí pueden generar resultados los cuales el usuario puede ver, estos resultados se muestran junto con los mensajes generados por la aplicación y los componentes en respuesta a las interacciones (con el usuario y entre componentes), así como cualquier mensaje de error resultado de alguna excepción en el comportamiento de alguno de los elementos en la consola de registros.

Pestañas de Scripts

Las pestañas de *scripts* son la otra vía de interacción con YADSL además de la **consola interactiva**. Se encuentran al lado del área de diseño de los flujos de trabajo y pueden ser creados cualquier cantidad de ellas. Al cargar un *script* automáticamente se crea una nueva pestaña con el código que esté dentro del *script* lista ya para ser ejecutada. Al mismo tiempo

si se realiza una acción de salvar un *script* el código que se guardará es el de la pestaña que se esté editando en ese momento.

3.3. Diseño de la implementación

El diseño adoptado para la implementación de un intérprete del lenguaje YADSL en base al modelo analizado puede ser dividido en dos partes fundamentales: un núcleo, que contiene todas las funciones y extensiones nativas o que están disponibles para el usuario al arrancar la aplicación, así como todas las variables y funciones que se creen por alguna instrucción; un modelo de ejecución, responsable de organizar la lógica representada en la escena y permitir la ejecución del flujo construido el cual se mantiene el desarrollado por (Mollineda, 2017).

El núcleo contiene en sí todas las funcionalidades necesarias como intérprete para poder analizar el código y ejecutarlo, este núcleo contiene la implementación de las acciones en sí. Este núcleo contiene una referencia al modelo de ejecución permitiéndole el paso de mensajes con este, realizando así las acciones asociadas.

Tanto los elementos de la escena como los del modelo de ejecución poseen un identificador que es compartido entre las dos partes que componen una unidad conceptual del modelo. Mediante este identificador, la aplicación mantiene sincronizado el estado de un componente frente a las interacciones del usuario y a la ejecución del modelo (Mollineda, 2017), pero como a nivel de programación, cada componente se traduce en una variable, es muy engorroso tratar a las variables como números, por lo que YADSL mantiene ambas referencias, el identificador para poderse comunicar con el modelo y la interfaz y el nombre de la variable para poder ser referenciada en el código.

En (Mollineda, 2017) se garantiza el extensibilidad y mantenibilidad del sistema dentro de la implementación mediante el concepto de extensiones. Una extensión se considera que es la suma de la parte lógica y la visual de un componente, o sea, para que una extensión desarrollada sea válida debe poseer estos dos aspectos, mientras que YADSL añade una forma de extensión más que es el mecanismo de importe de código, ya que segmentos de código desarrollados por otra persona pueden ser reutilizados según se necesite.

La estructura que se muestra en Figura 3.3, es en principio los módulos que contienen las clases que permiten realizar todas las funcionalidades de YADSL. Toda la estructura de clases que se encuentra debajo del paquete **`control`** son las relacionadas directamente con esta función.

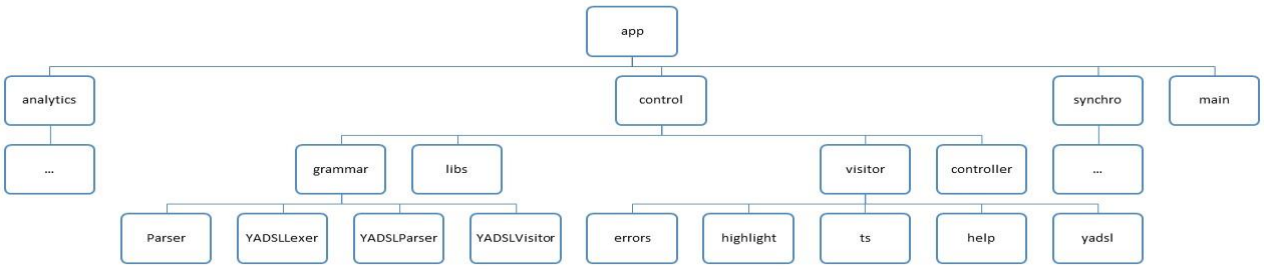


Figura 3.3 Estructura de paquetes y módulos de la aplicación

Paquete `gramar`

Este módulo concentra las clases y funciones relativas al reconocimiento del lenguaje desde el punto de vista sintáctico. Estas clases son generadas automáticamente por el generador del *parser* usado (ANTLR4). Luego redefiniéndose sus métodos para realizar las acciones apropiadas. Dentro de este módulo se encuentra la clase **`YADSLLexer`** que es la clase encargada de realizar la tokenización del flujo de texto y luego le provee este flujo al *parser*. En esta clase están definidas todas las palabras reservadas, y cualquier tóken válido del lenguaje. Además, se encuentra **`YADSLParser`** que es la clase encargada principalmente del chequeo de la sintaxis del flujo de texto a procesar mediante su mecanismo recursivo (CDR), si es inválido el flujo intenta recuperarse del error, pero en caso de ser imposible esta recuperación sencillamente detiene la ejecución. Por último, se encuentra la clase **`YADSLVisitor`** que es la que provee el mecanismo para "visitar" cada nodo en la forma

interna asociada y ejecutar el código asociado a este siguiendo el patrón *visitor*[38], de esta clase hereda **`YADSL`**.

Paquete `visitor`

En el módulo *visitor* se encuentran la mayoría de las funcionalidades principales del intérprete incluyendo otros módulos que ayudan a una implementación clara de las funcionalidades.

Módulo `errors`

El módulo **`errors`** es una colección de todos los errores que pudiera lanzar el intérprete durante su ejecución para un mejor tratamiento de estos. La clase **`YADSLException`** es la principal clase dedicada a este aspecto, la cual es la más general y se usa cuando ocurre un error inesperado, además de ella heredan todas las demás clases de error asociadas con la ejecución del intérprete. Un ejemplo de estas clases es **`NameNotFound`**, la cual se lanza cuando se hace una referencia a una variable que no está previamente inicializada, otro ejemplo sería **`UnsuportedOperation`** la cual está destinada a cuando se pretende realizar una operación entre dos tipos de datos incompatibles como por ejemplo la suma de un componente con cualquier otra variable.

Módulo `ts`

El módulo **`ts`** (Figura 3.4) contiene una colección de clases destinadas al control y gestión de la tabla de símbolos. Dentro de este se encuentran cuatro clases, la primera es la clase **`Type`**, la cual contiene un identificador único para cada tipo de dato que soporta YADSL como lenguaje, esta clase se usa principalmente estática, ya que estos identificadores son globales y no se usan de ninguna otra manera.

[38] Técnica de implementación sobre compiladores descendentes recursivos

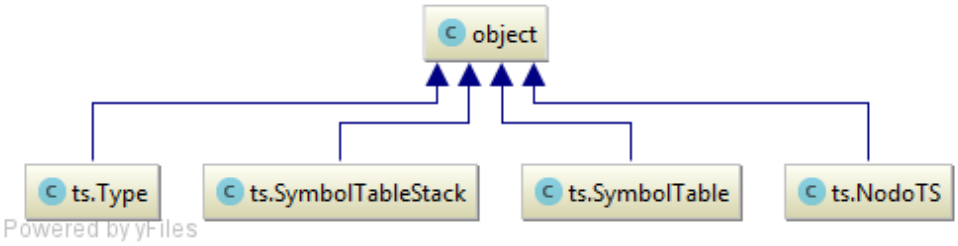


Figura 3.4 Diagrama de clases: ts

Otra clase muy importante para la correcta manipulación y mantenimiento de la tabla de símbolos es la clase **`NodoTS`**. Esta clase representa una instancia de cualquier tipo de dato en YADSL, así como funciones nativas o declaradas por el usuario. Cada instancia de esta clase contiene una referencia a un objeto de Python que sería el dato en sí y una variable que almacena el tipo de dato que representa en YADSL, esta variable no es más que una copia del identificador correspondiente de la clase **`Type`**. En el caso de ser componentes del modelo esta clase además almacena una referencia a su contraparte en la interfaz gráfica. Cada vez que se crea una variable nueva o se declara una función nueva, estas terminan como una instancia de esta clase para poder ser después fácilmente manipuladas.

La clase **`SymbolTable`** es la encargada de gestionar la tabla de símbolos ya que ella es en sí la tabla de símbolos. Almacena todas las variables, funciones, métodos nativos y todos los componentes que se encuentren instalados. Cuando se crea una instancia nueva del intérprete se crea al menos una instancia de esta clase donde se almacenan como **NodoTS** los componentes que tiene añadido el modelo, además de cargar todas las funciones nativas del lenguaje, enlazando su nombre o identificador con la referencia apropiada para su posterior uso. En esta clase además se guardan todas las conexiones entre componentes para su rápido acceso y modificación.

Debido a que YADSL presenta ámbitos en las funciones, o sea, cualquier variable declarada dentro de una función no es visible fuera de ella, es necesario controlar esta funcionalidad y para esto se desarrolló la clase **`SymbolTableStack`**. Esta clase internamente controla una estructura de datos del tipo pila, con la filosofía *LIFO*[39], donde sus elementos son instancias

[39]Técnica de manipulación de listas de elementos: *'Last in-first out'*, ultimo en entrar es el primero que sale

de la clase **`SymbolTable`**. Además, contiene dos funciones principales representadas por los dos métodos: **`new_function_context`** y **`pop`**. La primera función crea una nueva tabla de símbolos copiando los elementos nativos, o sea, las funciones nativas del lenguaje, las vistas de datos generadas hasta el momento y elimina todas las variables declaradas anteriormente para que no sean accesibles dentro de la función, mientras que la segunda solamente elimina la última tabla de símbolos que se haya creado. Este mecanismo permite gestionar el ámbito en las funciones de una manera sencilla y cómoda.

Módulo highlight

Este módulo se encarga principalmente del resaltado de la sintaxis y todo lo relacionado con este aspecto. Este no es un módulo con muchas clases y su funcionalidad pudiera haber estado repartida en otras clases, pero para su posterior optimización o cambios se decidió realizar esta acción concentrada en un solo lugar.

Este módulo cuenta con una sola clase la cual es la única y principal encargada de realizar el resaltado de la sintaxis en el editor del código. Esta clase, cuyo nombre es **`Highlighter`**, hereda de la clase del *framework* de Qt, **`QSyntaxHighlighter`** la cual brinda facilidades para realizar el resaltado de una manera sencilla y eficiente. Dentro de la clase principal se definieron las palabras reservadas y las reglas a resaltar, por lo que si se desea añadir nuevas palabras reservadas o sencillamente es añadir su definición como una expresión regular y como se desea realizar el resaltado usando la clase **`QTextCharFormat`** del *framework* de Qt.

Módulo `help`

El módulo **`help`** contiene una colección de funciones prácticas pero que semánticamente no pertenecen a otra clase. Además, se tomó la decisión de colocar estas funciones en este módulo para aliviar la cantidad de código que toman otras clases.

Las principales funciones en este módulo son:

- **`load_exported_vars_and_methods`**: Es el encargado de buscar las variables y funciones que el diseñador de un componente considere útiles para el analista respetando siempre los permisos.

- **`deatach_node_ts`**: Es uno de los métodos más usados ya que permite, junto con `attach_node_ts`, una comunicación sencilla entre el código Python de los componentes y el código desarrollado por los analistas. Este realiza un mapeo de los tipos de datos YADSL a los de Python.
- **`attach_node_ts:`** Realiza el proceso contrario de `deatach_node_ts`, lleva datos de Python a YADSL.
- **`find_type:`** Encuentra el tipo de dato de una variable desconocida.

Módulo `controller`

El módulo **controller** contiene la clase encargada de realizar la comunicación entre la interfaz y el lenguaje junto con otras clases necesarias para el correcto funcionamiento del intérprete. Estas clases se muestran en la Figura 3.5.

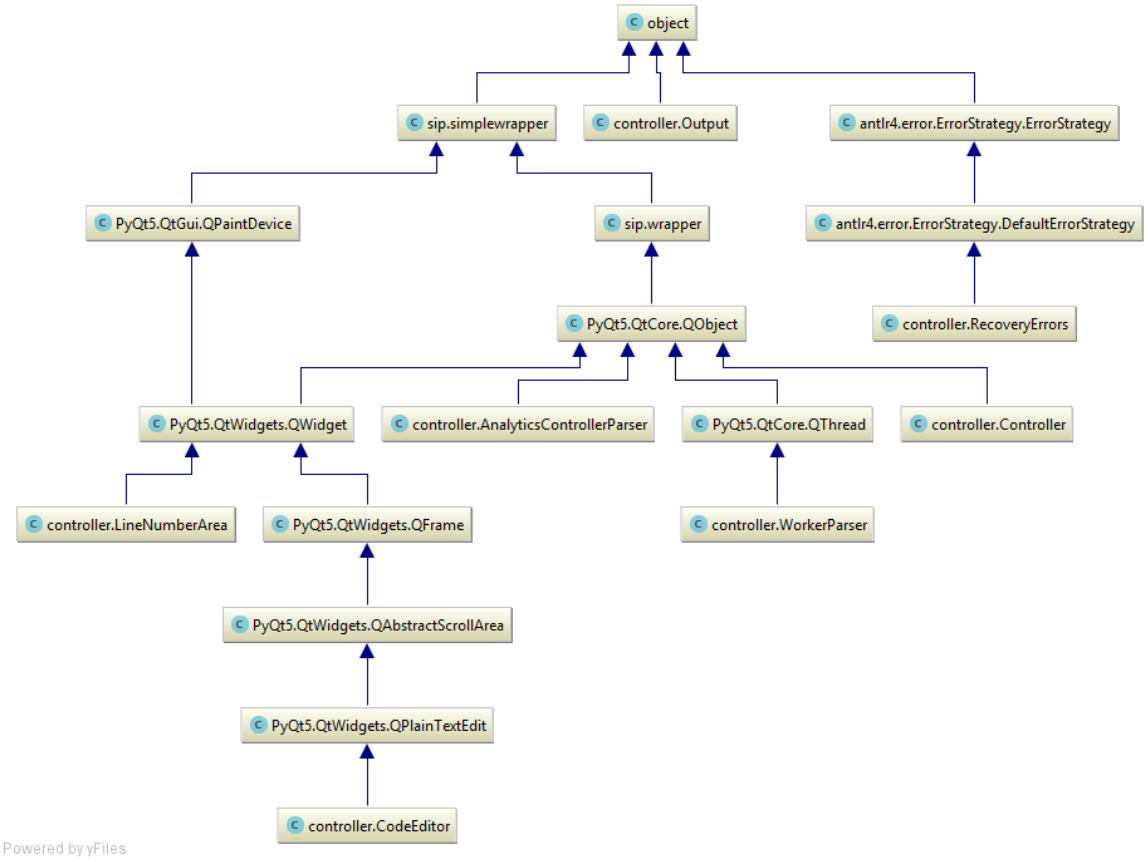


Figura 3.5 Diagrama de clases: controller

En este módulo se encuentran dos clases directamente relacionadas con la interfaz gráfica destinadas a la manipulación del editor de código, ellas son **`CodeEditor`** y **`LineNumberArea`**. La primera gestiona la parte de la interfaz donde se escribe le código directamente la cual soporta varios *shortcuts*[40] como **`ctrl-c, ctrl-v, ctrl-z,ctrl-y`** y **`ctrl-a`** destinados a copiar el texto seleccionado, pegar el contenido textual que se encuentre en el *clipboard*[41] del sistema operativo, volver al estado anterior del texto, ir al estado siguiente del texto y seleccionar todo el texto respectivamente. Sobre esta clase es que está la representación visual del texto escrito como código por los analistas. La segunda se encuentra muy ligada con esta clase ya que realiza la única acción de llevar actualizado en la interfaz la cantidad de líneas de códigos que se han escrito.

Otras dos clases muy unidas en este módulo son **`AnalyticsControllerParser`** y **`WorkerParser`** ya que la primera prepara al intérprete para su primera ejecución y ejecuciones subsiguientes y permite que el intérprete notifique a la interfaz de cualquier cambio mientras que la segunda es la encargada de realizar la ejecución del intérprete en un hilo aparte, por lo que esta hereda de la clase **`QThread`** (Johnson, 2014) la cual permite crear hilos ligeros de una manera fácil y eficiente y de esta manera no interferir con la interfaz gráfica evitando problemas de actualización en esta.

La clase que une todas las funcionalidades es la clase **`Controller`**. Está diseñada para detectar una entrada por la consola o la acción de ejecutar un script y notificar al intérprete de esta acción pasándole la información necesaria. Además, es la que se encarga de generar las pestañas de *scripts,* así como guardar y cargar estos sobre archivos físicos en el disco.

3.2 Caso de estudio planteado por (Mollineda, 2017): propiedades que influyen sobre la permeabilidad

Para probar la capacidad del lenguaje en la solución de problemas, su aplicabilidad sobre el modelo y la implementación de este desarrollada en la aplicación BCSMap, se sometió a un uso práctico mediante un caso de estudio diseñado en (Mollineda, 2017) por su autor y un

[40] Acceso rápido, generalmente es una combinación de teclas

[41] Portapapeles, lugar destinado a almacenar temporalmente información textual, imágenes, audio o video

grupo de especialistas de la rama biofarmacéutica. Este consiste en constatar, mediante el uso del lenguaje, la relación existente entre un conjunto de propiedades y la permeabilidad de los compuestos sometidos a estudio.

Como fuente para los datos empleados en la demostración se empleó un extracto de una base de datos biofarmacéutica, en formato CSV[42] y conteniendo 322 compuestos y 184 propiedades para cada uno de estos. Para adaptar la entrada al subconjunto de propiedades interesantes para este caso de análisis en particular, se creó un esquema de entrada conteniendo 17 propiedades que influencian o son influenciadas por los valores de permeabilidad de un fármaco como se muestra en la Figura 3.6.

```
schemas (test:
        name='MW'          type='Continuous'  treated='Descriptor',      #Molecular weight
        name="nHDon"       type="Continuous"  treated="Descriptor",      #Number of Hydrogen donated
        name="nHAcc"       type="Continuous"  treated="Descriptor",
        name="Ui"          type="Continuous"  treated="Descriptor",
        name="BCS-Final"   type="Nominal"     treated="BCS Class",
        name="Hy"          type="Continuous"  treated="Descriptor",
        name="AMR"         type="Continuous"  treated="Descriptor",
        name="TPSA(NO)"    type="Continuous"  treated="Descriptor",
        name="TPSA(Tot)"   type="Continuous"  treated="Descriptor",
        name="MLOGP"       type="Continuous"  treated="Descriptor",      #Partition coefficient
        name="MLOGP2"      type="Continuous"  treated="Descriptor",
        name="ALOGP"       type="Continuous"  treated="Descriptor",
        name="ALOGP2"      type="Continuous"  treated="Descriptor",
        name="Do"          type="Continuous"  treated="Descriptor",      #Maximum dosis
        name="Foral"       type="Continuous"  treated="Bioavailability"  #Oral Fraction absorbed
    )

inp = csvinput(label='input')
rad = radviz(label='viz')
fil = filters(label='Rule of Five')
kms = kmeans(label='First-Clustering')
kms2 = kmeans(label='Second-Clustering')
```

Figura 3.6 Esquema y componentes

El diseño del flujo de análisis contempla un componente de entrada del tipo `csvinput` a partir del fichero CSV, además, se configuró un componente de filtrado en base a los valores de propiedades, configurado como se muestra en la Figura 3.7 para aplicar la regla de cinco de Lipinski (Lipinski, 2004), un criterio biofarmacéutico empleado para discriminar fármacos que pueden presentar problemas potenciales en su absorción (Mollineda, 2017).

[42] Comma separated value, elementos separados por coma

Seguidamente se usaron dos componentes de análisis que aplican una implementación de k-medias como se muestra en la Figura 3.6 y Figura 3.7.

Para permitir el paso selectivo de vistas de datos en base a las demandas del analista se colocó un componente del tipo `dispatcher`. Finalmente, para poder realizar la exploración visual, un componente de visualización equipado con una técnica destinada a la identificación de estructura en base a un conjunto de propiedades y etiquetas (Mollineda, 2017).

Posteriormente es necesario definir las conexiones entre los componentes como se muestra en la Figura 3.7.

```
23 kms2 = kmeans(label='Second-Clustering')
24 dis = dispatcher('On-Demand dispatcher')
25
26
27 setup inp: vconfig = {'source':"D:\Documentos\bcsmap-proj\input-files\test-input-no-smiles.csv", 'vschema': test }
28
29 setup fil: vconfig = {'filters':{
30                         'filter1':{'vschema': test, 'Property':'MW',    'Operation':'<=', 'Value':500.0},
31                         'filter2':{'vschema': test, 'Property':'MLOGP', 'Operation':'<=', 'Value':5.0},
32                         'filter3':{'vschema': test, 'Property':'nHAcc', 'Operation':'<=', 'Value':10.0},
33                         'filter4':{'vschema': test, 'Property':'nHDon',  Script tab n':'<=', 'Value':5.0}
34                       }
35               }
36 setup kms: vconfig = {
37                   'vschema': test,
38                   'minimum-size': 30,
39                   'na-policy': 'drop',
40                   'iterations': 20,
41                   'selected-features': ['Foral', 'Do', 'MW', 'MLOGP'],
42                   'threshold': 0.0001,
43                   'k': 4
44               }
```

Figura 3.7 Configuración de los componentes

La conexión entre `dis` y `kms2` esta desactivada porque no todas las vistas van a pasar inicialmente a `kms2` como se muestra en la Figura 3.8, luego esta conexión se activa para permitir el paso.

```
41                'selected-features': ['Foral', 'Do', 'MW', 'MLOGP'],
42                'threshold': 0.0001,
43                'k': 4
44              }
45 setup kms2: vconfig = {
46                'vschema': test,
47                'minimum-size': 30,
48                'na-policy': 'drop',
49                'iterations': 20,
50                'selected-features': ['TPSA(Tot)', 'MLOGP', 'MW', 'nHDon', 'Do'],
51                'threshold': 0.0001,
52                'k': 2
53               }
54
55 inp -> fil, rad
56 fil ->kms, dis
57 kms, kms2 -> dis -> rad, kms2
58
59 show inp in x=0 y=0, fil in x=0 y=200, rad in x =300 y = 0, kms in x =100 y =100, kms2 in x = 320 y = 170
60 dis ~> kms2
61
62 start
63
```

Figura 3.8 Conexiones entre componentes

Luego de esta serie de instrucciones se genera una representación visual en la aplicación (Figura 3.9), la cual hasta ese momento no ha realizado ninguna acción de procesamiento de datos, solo se han declarado los componentes de análisis , creado un esquema sobre el cual trabajar y realizado todas las conexiones pertinentes entre los componentes y el modelo está listo para su ejecución.

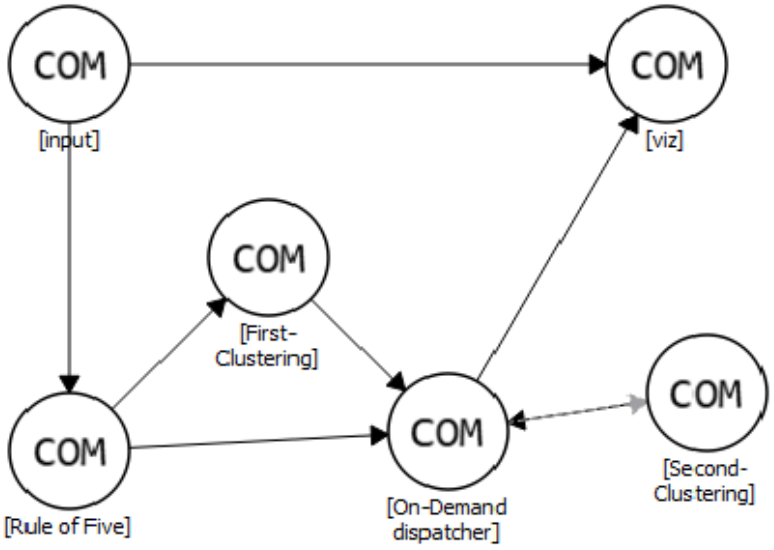


Figura 3.9 Representación visual del resultado de las instrucciones

Para iniciar la ejecución global se escribe la sentencia `start` e inicia el proceso de análisis. Inicialmente el componente `inp` genera una vista de datos con toda la información contenida en el CSV utilizado. Luego después de la aplicación del filtro configurado con la regla de cinco, la ejecución global del flujo produce en primera instancia un conjunto de datos filtrados donde se reduce la cantidad de compuestos a considerar a 252, concentrados principalmente en fármacos de la clase I, II y III, con elevada permeabilidad y solubilidad. La mayor parte de los fármacos pertenecientes a la clase IV fueron eliminados en este paso, dado que en su mayor parte violan algún punto de la regla de Lipinski (Mollineda, 2017).

Como salida del procesamiento del primer componente de k-medias, se obtienen cinco vistas de datos, de las cuales una se corresponde a la vista de entrada filtrada, pero con la información de agrupamiento agregada a ella. Las restantes cuatro se corresponden con los grupos identificados. Para el caso de estudio aplicado resulta de particular interés la vista correspondiente a la clasificación III/I, donde se experimenta la mayor variación en la permeabilidad (Mollineda, 2017).

Cuando pasa esa vista hacia el componente de visualización `rad` mediante la activación del componente de control de flujo, es posible estudiar el comportamiento de la estructura visible en las clases BCS con respecto a las propiedades seleccionadas lo que permite una generación de hipótesis por parte del especialista que se encuentra conduciendo el análisis. Mediante esta interacción, se reduce el número de propiedades seleccionadas hasta un subconjunto donde es perceptible la división del conjunto de los datos en dos con valores de permeabilidad en extremos opuestos.

Al aplicar esta misma selección de propiedades sobre los datos originales sin filtrar, es posible observar la misma regularidad, lo que aumenta el número de elementos que apoyan que este conjunto de propiedades en particular es una buena elección para discriminar entre compuestos con elevada y baja permeabilidad (Mollineda, 2017)

Luego se aplica un nuevo paso de agrupamiento en el componente de k-medias `kms2` y mediante el empleo del componente de visualización `rad`, se puede constatar que los datos filtrados han sido divididos en dos grupos correspondientes a elementos de baja y alta permeabilidad respectivamente (Figura 3.10). La visualización del nuevo agrupamiento y su

comparación con la distribución de las clases propiamente dichas apoya fuertemente la conclusión obtenida.

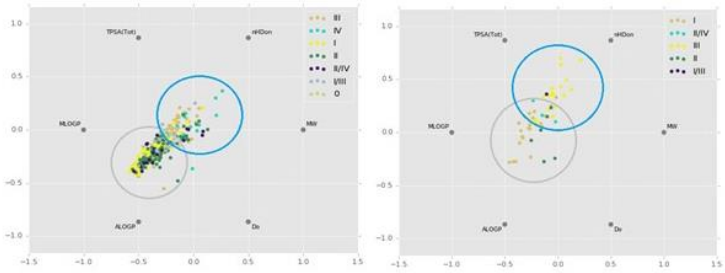


Figura 3.10 Comparación de la distribución de clases para el nuevo agrupamiento

3.3 Comparación de resultados

Usando el modelo original, para varios *datasets*[43]: uno pequeño de 100 entradas, uno mediano de 200 y el anteriormente mencionado de 322, se definió un flujo de trabajo de prueba donde las imágenes resultantes se almacenaron en formato *jpeg*[44] (Acharya & Ray, 2005) para realizar una comparación entre estas y las resultantes usando el lenguaje. El modelo mencionado consta de cuatro componentes: `CSVInput`, `KMeans`, `Filter` y `RadViz` donde estos están conectados de la siguiente manera, `CSVInput` con `KMeans`, este con `Filter` y `Filter` con `RadViz`. Luego de realizar la ejecución del modelo se obtienen cuatro vistas de datos, las cuales son almacenadas. Luego se realiza el mismo flujo de ejecución, pero esta vez usando YADSL e igualmente se obtienen los mismos cuatro resultados los cuales son almacenados. Luego estas vistas(se muestran solo dos de ellas) son comparadas obteniéndose efectivamente el mismo resultado como se muestra en las figuras: Figura 3.11 y Figura 3.12.

[43] Conjunto de datos

[44] Método de compresión de imágenes

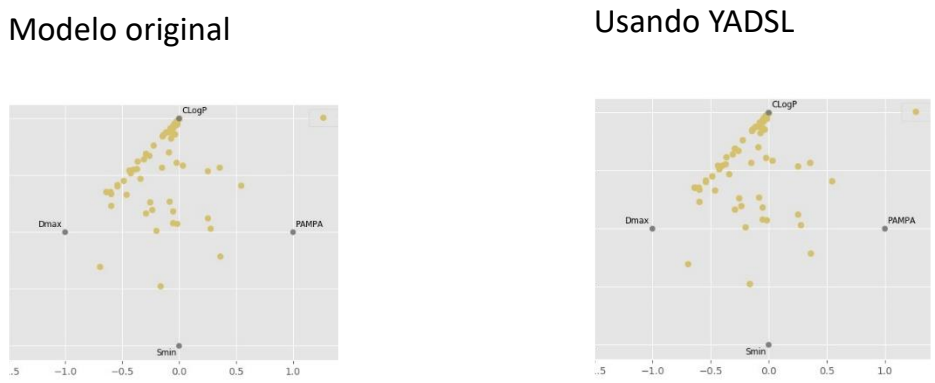


Figura 3.11 Primera vista

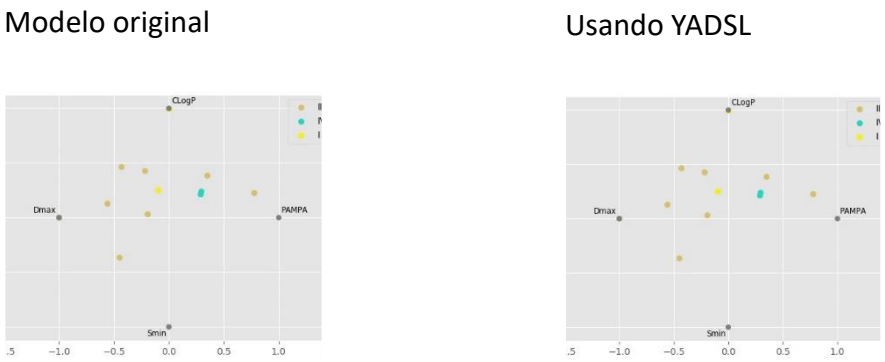


Figura 3.12 Tercera vista

3.4 Conclusiones parciales

Como resultado del proceso de análisis planteado por (Mollineda, 2017) se obtuvo un nuevo conocimiento en forma de un conjunto de propiedades potencialmente útil para discriminar nuevos fármacos (Mollineda, 2017) de los cuales no se posea su clasificación BCS, lo cual se pudo alcanzar usando como herramienta el lenguaje diseñado.

Conclusiones

Como resultado de este trabajo se diseñó un lenguaje de programación propio para el modelo propuesto por (Mollineda, 2017) y se realizó una implementación de un subconjunto de instrucciones del lenguaje para aumentar su flexibilidad y facilitar su uso en aplicaciones reales, cumpliéndose de esta forma su objetivo general. Además se demostró que el empleo del lenguaje como herramienta para enfrentar el problema de la búsqueda de conocimiento mediante el análisis visual de datos usando como base el modelo propuesto por (Mollineda, 2017) es aplicable aumentando la flexibilidad del modelo.

Se pudieron identificar las características relevantes que presenta el modelo para su extensión mediante un lenguaje de programación, se determinaron las características que debe presentar un lenguaje de programación para aumentar las potencialidades del modelo y se extendió la herramienta informática para que implemente un intérprete.

Recomendaciones

Derivado del estudio realizado, así como las conclusiones emanadas del mismo, se recomienda para el posterior estudio y utilización:

1. Considerar la modularización de la herramienta, que permita una mejor integración del modelo con el lenguaje.
2. Añadir la posibilidad de definición de nuevos tipos de datos, semejante a la definición de clases en lenguajes orientados a objetos.
3. Brindar un mejor soporte y más instrucciones de manipulación de las vistas de datos, para que estos sean posibles de tratar como diccionarios nativos.
4. Incluir técnicas de programación orientada a aspectos al comportamiento de los componentes para poder programar tareas antes de la ejecución de cada componente y después, por ejemplo.

Bibliografía

Absorption System, 2013. BCS Biowiavers: Fundamental Question.

Acharya, T. & Ray, A. K., 2005. Image Processing Principles and Applications. *Wiley-Interscience*, pp. 351-387.

Altintas, I. et al., 2004. Kepler: An Extensible System for Design and Execution of Scientific Workflows.

Ascher, D. & Lutz, M., 2003. Learning Python. *O'Reilly*, p. 620.

Bank, S., 2015. Graphviz Documentation.

Borbón, A. A. & Mora, W. F., 2013. Edición de textos científicos Latex. *Revista digital Matemática, Educación e Internet.*

Bravenboer, M. & Visser, E., 2004. Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions.

Callahan, S. P. et al., 2006. Managing the Evolution of Dataflows with VisTrails.

Davidson, S. . B. & Freire, J., 2008. Provenance and Scientific Workflows: Challenges and Opportunities.

Guttag, J. V., 2013. Introduction to Computation and Programming Using Python. *Spring*, pp. 23-25.

Hemel, Z., Verhaaf, R. & Visser, E., 2008. WebWorkFlow: An Object-Oriented Workflow Modeling Language for Web Applications.

Hudak, P., 1996. *Building domain-specific embedded languages.* s.l.:s.n.

Hudak, P., 1997. *Domain Specific Languages.* s.l.:s.n.

Hulette, G. C., 2008. *The WOOL Workflow Programming Language.* s.l.:s.n.

Johnson, D., 2014. QThread: Are you doing it wrong?.

Koutsofios, E. & North, S. C., 1996. Editing graphs with dotty.

Kreines, D. C., 2000. Oracle SQL The Essential Reference. *O'Reilly.*

Lipinski, C. A., 2004. ‘Lead-and drug-like compounds: the rule-of-five revolution Drug Discovery Today: Technologies.. *Elsevier*, p. 337–341.

Lippman, S. B., 2002. Essential C++. *Addison Wesley*, p. 416.

Luzza, M. et al., 2012. Diseño y Construcción de Lenguajes Específicos del Dominio.

Mollineda, D., 2017. *Analisis Visual de Datos Biofarmaceuticos.* Santa Clara: s.n.

Parr, T., 2012. *The Definitive ANTLR4 Reference.* s.l.:The Pragmatic Programmers.

Rossum, G. V., 2017. The Python Tutorial.

Schildt, H., 2005. Java A Beginner's Guide. *McGraw-Hill/Osborne.*

The YAWL Foundation, 2010. YAWL - Technical Manual.

van Deursen, A., Klint, P. & Visser, J., 1998. *Domain-Specific Languages.* s.l.:s.n.

Visser, E., 2008. WebDSL: A case study in domain-specific language engineering. *Lecture Notes in Computer Science, Springer.*

Weske, M. & Vossen, G., 1998. *Workflow Lenguages,* Berlin: Springer.

ANEXOS

Gramática de YADSL

```
tokens { INDENT, DEDENT }

console_input : single_input
                    | dsl_stmt
                    ;

file_input
    :  NEWLINE* (workflow_def | stmt | funcdef_stmt)+  EOF
    ;

schemas : SCHEMAS '('(NAME ':' schema_vars )(';' NAME ':' schema_vars )*';'?')';

schema_vars : ('name''=' strr 'type''='strr 'treated''='strr) (','name''=' strr 'type''='strr 'treated''='strr)*;

workflow_def: WORKFLOW NAME ':' NEWLINE
                    INDENT
                            (schemas NEWLINE)?
                            inputs?
                            outputs?
                            (stmt | funcdef_stmt | NEWLINE)*
                    DEDENT
   ;


inputs:   NODEIN NAME '=' power (',' NAME '=' power)* NEWLINE;

outputs:  (NODEOUT NAME | node_out_def)(',' NODEOUT NAME | node_out_def)* NEWLINE;

node_out_def: NAME '=' power (',' NAME '=' power)*
   ;
single_input
    : NEWLINE
    | simple_stmt
    | compound_stmt NEWLINE
    ;
stmt
    : simple_stmt
    | compound_stmt
```

```
    | dsl_stmt
   ;
simple_stmt
    : small_stmt ( ';' small_stmt )* ';'? NEWLINE
    ;
compound_stmt
     :  if_stmt
      | while_stmt
      | for_stmt
    ;
dsl_stmt: (schemas | conecctions_stmt | control_stmt | visualize_stmt |
setup_stmt | publish)';'? NEWLINE ;
publish: 'publish' exprlist;
connect_arrows: '->' | '~>' | '<-' | '<~' ;
setup_stmt: SETUP NAME (':' varargslist)? ;
control_stmt: start_stmt;
start_stmt :  START testlist?;
visualize_stmt: visual_adjust | node_visual_info;
visual_adjust:    LAYOUT NAME
                  |  SHOW  NAME  IN  'x''='test  'y''='test  (','  NAME  IN
   'x''='test 'y''='test)*
                  | HIDE NAME (',' NAME)*
                  | SHOW_UI NAME
                  ;
node_visual_info : VISUALIZE NAME varargslist?;
small_stmt :
        expr_stmt
      | del_stmt
      | pass_stmt
      | flow_stmt
      | import_stmt
   ;
import_stmt
    : import_name
    | import_from
    | import_file_as_name
    ;
del_stmt
```

```
    : DEL NAME (',' NAME)*
   ;
pass_stmt
    : PASS
 ;
flow_stmt
    : break_stmt
    | continue_stmt
    | return_stmt
    ;
break_stmt
    : BREAK
 ;
continue_stmt
    : CONTINUE
    ;
return_stmt
    : RETURN expr?
   ;
if_stmt
    : IF test ':' suite elifs? ( ELSE ':' suite )?
    ;
elifs:( ELIF test ':' suite )+;
while_stmt
    : WHILE test ':' suite ( ELSE ':' suite )?
    ;
funcdef_stmt
    : DEF NAME parameters ('->' (NAME| WORKFLOW))? ':' suite
    ;
for_stmt
    : FOR NAME (',' NAME)* IN testlist ':' suite ( ELSE ':' suite )?
    ;

expr_stmt
    : assig                                 #ASSIGNATION
    | test                                  #TEST_EXPR
    ;
```

```
power_name : NAME trailer*;
parameters
    : '(' typedargslist? ')'
    ;
typedargslist
    : tfpdef default_value? ( ',' tfpdef default_value? )*
    ;
varargslist
    : (NAME '=')? test (',' (NAME '=')? test )*
    ;
default_value : '=' test;

tfpdef
    : NAME ( ':' NAME )?
    ;
suite
    : simple_stmt
    | NEWLINE INDENT stmt+ DEDENT
    ;
import_name
    : IMPORT dotted_as_names
 ;
import_from
    : FROM  dotted_name IMPORT ( '*' | '**' | import_as_names )
    ;
import_file_as_name  :  import_file  (AS  NAME)?  |  FROM  strr  IMPORT
import_as_names
   ;
import_file: IMPORT strr
   ;
import_as_name : NAME ( AS NAME )?
    ;
import_as_names
    : import_as_name ( ',' import_as_name )* ','?
    ;

dotted_as_names
```

```
    : dotted_as_name ( ',' dotted_as_name )*
    ;
dotted_as_name
    : dotted_name ( AS NAME )?
    ;
dotted_name
    : NAME ( '.' NAME )*
    ;
test
    : or_test ( IF or_test ELSE test )?
    ;
testlist
    : test ( ',' test )*
    ;
or_test
    : and_test ( OR and_test )*
    ;
and_test
    : not_test ( AND not_test )*
    ;
not_test
    : NOT not_test
    | comparison
    ;
   comparison
    : expr (comp_op expr)*  ;
comp_op
    : '<'
    | '>'
    | '=='
    | '>='
    | '<='
    | '!='
    | IN
    | NOT IN
    | IS
    | IS NOT
```

```
    | TYPE
    ;
augassign
    : '+='
    | '-='
    | '*='
    | '/='
    | '%='
    | '&='
    | '|='
    | '^='
    | '<<='
    | '>>='
    | '**='
    | '//='
    ;
expr
    : xor_expr ( '|' xor_expr )*
    ;
xor_expr
    : and_expr ( '^' and_expr )*
   ;
and_expr
    : shift_expr ( '&' shift_expr )*
    ;
shift_expr
    : arith_expr ( operatorLEFT_SHIFT_RIGHT_SHIFT arith_expr)*
    ;
operatorLEFT_SHIFT_RIGHT_SHIFT : LEFT_SHIFT | RIGHT_SHIFT ;
arith_expr
    : term (operatorADD_MINUS term)*
    ;
operatorADD_MINUS : ADD | MINUS ;

term
    : factor ( operatorSTAR_DIV_MOD_IDIV factor)*
    ;
```

```
operatorSTAR_DIV_MOD_IDIV : STAR | DIV | MOD | IDIV;

conecctions_stmt:
               DEL test op=('->'|'<-') test    #DEL_CON
             | testlist (connect_arrows testlist)+    #CONNECT
             | NAME ('split'|'join') ('xor' | 'and' | 'or')
   expr?#PETRI_SPLIT_JOIN
   ;
factor
    : factor ('->'|'<-') factor '?'    #TEST_CONECCT
    | factor '?'                       #TEST_NODE
    |'+' factor                        #PLUS_FACTOR
    | '-' factor                       #MINUS_FACTOR
    | '@' factor                       #ACTIVATE_NODE
    | '~' factor                       #DEACTIVATE_ATOM
    | power                            #POWER
    ;
assig :   power_name augassign test      #AUGMENT_ASSIG
          | (power_name '=')+ test          #ASSIG
    ;
power
    : atom trailer* (POWER factor)?
    ;
trailer
    : '(' varargslist? ')'             #TRAILER_PAR
    | '[' subscriptlist ']'            #TRAILER_SQUARE
    | '.' NAME                         #TRAILER_DOT_NAME
 ;
arglist : argument ( ',' argument)* ;
argument
    : test comp_for?
    | NAME '=' test
    ;
atom
    :'('expr')'                        #ATOM_PARENTESIS
    |'[' testlist_comp? ']'            #ATOM_SQUARE_BRACES
    | '{' dictorsetmaker? '}'          #ATOM_CURLY_BRACES
```

```
    | NAME                                #ATOM_NAME
    | number                              #ATOM_NUMBER
    | strr+                               #ATOM_STRING
    | NONE                                #ATOM_NONE
    | TRUE                                #ATOM_TRUE
    | FALSE                               #ATOM_FALSE
    ;
dictorsetmaker :  expr ':' test (',' expr ':' test)*
    ;
exprlist
    : expr ( ',' expr )*
    ;
comp_iter
    : comp_for
    | comp_if
    ;
comp_if
    : IF or_test comp_iter?
    ;
testlist_comp
    : test ( comp_for | ( ',' test )* ','? )
    ;
comp_for
    : FOR exprlist IN or_test comp_iter?
    ;
subscriptlist
    : subscript ( ',' subscript )*
    ;
subscript
    : expr
    | expr? ':' subscript_last
    ;
subscript_last: expr? sliceop?;
sliceop
    : ':' expr?
    ;
number
```

```
    : integer
    | FLOAT_NUMBER
    | IMAG_NUMBER
    ;
strr
    : STRING_LITERAL
    | BYTES_LITERAL
    ;
integer
    : DECIMAL_INTEGER
    | OCT_INTEGER
    | HEX_INTEGER
    | BIN_INTEGER
    ;
DEF : 'def';
RETURN : 'return';
FROM : 'from';
IMPORT : 'import';
AS : 'as';
IF : 'if';
ELIF : 'elif';
ELSE : 'else';
WHILE : 'while';
FOR : 'for';
IN : 'in';
OR : 'or';
AND : 'and';
NOT : 'not';
IS : 'is';
NONE : 'None';
TRUE : 'true';
FALSE : 'false';
DEL : 'del';
PASS : 'pass';
CONTINUE : 'continue';
BREAK : 'break';
TYPE : 'istype';
```

```
PAUSE:'pause';
STOP : 'stop';
START: 'start';
LAYOUT: 'layout';
NODEOUT:'outputs';
NODEIN: 'inputs';
WORKFLOW: 'workflow';
SETUP: 'setup';
CLEAN:'clean';
SHOW : 'show';
HIDE : 'hide';
SCHEMAS : 'schemas';
VISUALIZE : 'visualize';
SHOW_UI: 'showui';
NEWLINE
    : (SPACES |( '\r'? '\n' | '\r') SPACES?)
   ;
NAME: ID_START (ID_CONTINUE)*
    ;
STRING_LITERAL
    :  SHORT_STRING | LONG_STRING
    ;
BYTES_LITERAL
    : ( [bB] | ( [bB] [rR] ) | ( [rR] [bB] ) ) ( SHORT_BYTES | LONG_BYTES
   )
    ;
DECIMAL_INTEGER
    : NON_ZERO_DIGIT DIGIT*
    | '0'+
    ;
OCT_INTEGER
    : '0' [oO] OCT_DIGIT+
    ;
HEX_INTEGER
    : '0' [xX] HEX_DIGIT+
    ;
BIN_INTEGER
```

```
    : '0' [bB] BIN_DIGIT+
    ;
FLOAT_NUMBER
    : POINT_FLOAT
    | EXPONENT_FLOAT
    ;
IMAG_NUMBER
    : ( FLOAT_NUMBER | INT_PART ) [jJ]
    ;
DOT : '.';
STAR : '*';
OPEN_PAREN : '(';
CLOSE_PAREN : ')';
COMMA : ',';
COLON : ':';
SEMI_COLON : ';';
POWER : '**';
ASSIGN : '=';
OPEN_BRACK : '[';
CLOSE_BRACK : ']' ;
OR_OP : '|';
XOR : '^';
AND_OP : '&';
LEFT_SHIFT : '<<';
RIGHT_SHIFT : '>>';
ADD : '+';
MINUS : '-';
DIV : '/';
MOD : '%';
IDIV : '//';
NOT_OP : '~';
OPEN_BRACE : '{' {self.opened += 1};
CLOSE_BRACE : '}' {self.opened -= 1};
LESS_THAN : '<';
GREATER_THAN : '>';
EQUALS : '==';
GT_EQ : '>=';
```

```
LT_EQ : '<=';
NOT_EQ_2 : '!=';
ARROW : '->';
ADD_ASSIGN : '+=';
SUB_ASSIGN : '-=';
MULT_ASSIGN : '*=';
DIV_ASSIGN : '/=';
MOD_ASSIGN : '%=';
AND_ASSIGN : '&=';
OR_ASSIGN : '|=';
XOR_ASSIGN : '^=';
LEFT_SHIFT_ASSIGN : '<<=';
RIGHT_SHIFT_ASSIGN : '>>=';
POWER_ASSIGN : '**=';
IDIV_ASSIGN : '//=';
SKIP_
    : ( SPACES | COMMENT ) -> skip
    ;
UNKNOWN_CHAR
    : .
    ;
fragment SHORT_STRING
    : '\'' ( STRING_ESCAPE_SEQ | ~[\\\r\n\f'] )* '\''
    | '"' ( STRING_ESCAPE_SEQ | ~[\\\r\n\f"] )* '"'
    ;
fragment LONG_STRING
    : '\'\'\'' LONG_STRING_ITEM*? '\'\'\'' 
    | '"""' LONG_STRING_ITEM*? '"""'
    ;
fragment LONG_STRING_ITEM
    : LONG_STRING_CHAR
    | STRING_ESCAPE_SEQ
    ;
fragment LONG_STRING_CHAR
    : ~'\\'
    ;
fragment STRING_ESCAPE_SEQ
```

```
    : '\\' .
    | '\\' NEWLINE
    ;
fragment NON_ZERO_DIGIT
    : [1-9]
    ;
fragment DIGIT
    : [0-9]
    ;
fragment OCT_DIGIT
    : [0-7]
    ;
fragment HEX_DIGIT
    : [0-9a-fA-F]
    ;

fragment BIN_DIGIT
    : [01]
    ;
fragment POINT_FLOAT
    : INT_PART? FRACTION
    | INT_PART '.'
    ;
fragment EXPONENT_FLOAT
    : ( INT_PART | POINT_FLOAT ) EXPONENT
    ;
fragment INT_PART
    : DIGIT+
    ;
fragment FRACTION
    : '.' DIGIT+
    ;
fragment EXPONENT
    : [eE] [+-]? DIGIT+
    ;
fragment SHORT_BYTES
    : '\'' ( SHORT_BYTES_CHAR_NO_SINGLE_QUOTE | BYTES_ESCAPE_SEQ )* '\''
```

```
    | '"' ( SHORT_BYTES_CHAR_NO_DOUBLE_QUOTE | BYTES_ESCAPE_SEQ )* '"'
    ;
fragment LONG_BYTES
    : '\'\'\'' LONG_BYTES_ITEM*? '\'\'\''
    | '"""' LONG_BYTES_ITEM*? '"""'
    ;
fragment LONG_BYTES_ITEM
    : LONG_BYTES_CHAR
    | BYTES_ESCAPE_SEQ
    ;
fragment SHORT_BYTES_CHAR_NO_SINGLE_QUOTE
    : [\u0000-\u0009]
    | [\u000B-\u000C]
    | [\u000E-\u0026]
    | [\u0028-\u005B]
    | [\u005D-\u007F]
    ;
fragment SHORT_BYTES_CHAR_NO_DOUBLE_QUOTE
    : [\u0000-\u0009]
    | [\u000B-\u000C]
    | [\u000E-\u0021]
    | [\u0023-\u005B]
    | [\u005D-\u007F]
    ;
fragment LONG_BYTES_CHAR
    : [\u0000-\u005B]
    | [\u005D-\u007F]
    ;
fragment BYTES_ESCAPE_SEQ
    : '\\' [\u0000-\u007F]
    ;
fragment SPACES
    : [ \t]+
    ;

   fragment COMMENT
    : '#' ~[\r\n\f]*
```

```
 ;
fragment ID_START
 : '_'
 | [A-Z]
 | [a-z]
 | '\u00AA'
 | '\u00B5'
 | '\u00BA'
 | [\u00C0-\u00D6]
 | [\u00D8-\u00F6]
 | [\u00F8-\u01BA]
 | '\u01BB'
 | [\u01BC-\u01BF]
 | [\u01C0-\u01C3]
 | [\u01C4-\u0241]
 | [\u0250-\u02AF]
 | [\u02B0-\u02C1]
 | [\u02C6-\u02D1]
 | [\u02E0-\u02E4]
 | '\u02EE'
 | '\u037A'
 | '\u0386'
 | [\u0388-\u038A]
 | '\u038C'
 | [\u038E-\u03A1]
 | [\u03A3-\u03CE]
 | [\u03D0-\u03F5]
 | [\u03F7-\u0481]
 | [\u048A-\u04CE]
 | [\u04D0-\u04F9]
 | [\u0500-\u050F]
 | [\u0531-\u0556]
 | '\u0559'
 | [\u0561-\u0587]
 | [\u05D0-\u05EA]
 | [\u05F0-\u05F2]
 | [\u0621-\u063A]
```

```
| '\u0640'
| [\u0641-\u064A]
| [\u066E-\u066F]
| [\u0671-\u06D3]
| '\u06D5'
| [\u06E5-\u06E6]
| [\u06EE-\u06EF]
| [\u06FA-\u06FC]
| '\u06FF'
| '\u0710'
| [\u0712-\u072F]
| [\u074D-\u076D]
| [\u0780-\u07A5]
| '\u07B1'
| [\u0904-\u0939]
| '\u093D'
| '\u0950'
| [\u0958-\u0961]
| '\u097D'
| [\u0985-\u098C]
| [\u098F-\u0990]
| [\u0993-\u09A8]
| [\u09AA-\u09B0]
| '\u09B2'
| [\u09B6-\u09B9]
| '\u09BD'
| '\u09CE'
| [\u09DC-\u09DD]
| [\u09DF-\u09E1]
| [\u09F0-\u09F1]
| [\u0A05-\u0A0A]
| [\u0A0F-\u0A10]
| [\u0A13-\u0A28]
| [\u0A2A-\u0A30]
| [\u0A32-\u0A33]
| [\u0A35-\u0A36]
| [\u0A38-\u0A39]
```

```
| [\u0A59-\u0A5C]
| '\u0A5E'
| [\u0A72-\u0A74]
| [\u0A85-\u0A8D]
| [\u0A8F-\u0A91]
| [\u0A93-\u0AA8]
| [\u0AAA-\u0AB0]
| [\u0AB2-\u0AB3]
| [\u0AB5-\u0AB9]
| '\u0ABD'
| '\u0AD0'
| [\u0AE0-\u0AE1]
| [\u0B05-\u0B0C]
| [\u0B0F-\u0B10]
| [\u0B13-\u0B28]
| [\u0B2A-\u0B30]
| [\u0B32-\u0B33]
| [\u0B35-\u0B39]
| '\u0B3D'
| [\u0B5C-\u0B5D]
| [\u0B5F-\u0B61]
| '\u0B71'
| '\u0B83'
| [\u0B85-\u0B8A]
| [\u0B8E-\u0B90]
| [\u0B92-\u0B95]
| [\u0B99-\u0B9A]
| '\u0B9C'
| [\u0B9E-\u0B9F]
| [\u0BA3-\u0BA4]
| [\u0BA8-\u0BAA]
| [\u0BAE-\u0BB9]
| [\u0C05-\u0C0C]
| [\u0C0E-\u0C10]
| [\u0C12-\u0C28]
| [\u0C2A-\u0C33]
| [\u0C35-\u0C39]
```

```
| [\u0C60-\u0C61]
| [\u0C85-\u0C8C]
| [\u0C8E-\u0C90]
| [\u0C92-\u0CA8]
| [\u0CAA-\u0CB3]
| [\u0CB5-\u0CB9]
| '\u0CBD'
| '\u0CDE'
| [\u0CE0-\u0CE1]
| [\u0D05-\u0D0C]
| [\u0D0E-\u0D10]
| [\u0D12-\u0D28]
| [\u0D2A-\u0D39]
| [\u0D60-\u0D61]
| [\u0D85-\u0D96]
| [\u0D9A-\u0DB1]
| [\u0DB3-\u0DBB]
| '\u0DBD'
| [\u0DC0-\u0DC6]
| [\u0E01-\u0E30]
| [\u0E32-\u0E33]
| [\u0E40-\u0E45]
| '\u0E46'
| [\u0E81-\u0E82]
| '\u0E84'
| [\u0E87-\u0E88]
| '\u0E8A'
| '\u0E8D'
| [\u0E94-\u0E97]
| [\u0E99-\u0E9F]
| [\u0EA1-\u0EA3]
| '\u0EA5'
| '\u0EA7'
| [\u0EAA-\u0EAB]
| [\u0EAD-\u0EB0]
| [\u0EB2-\u0EB3]
| '\u0EBD'
```

```
| [\u0EC0-\u0EC4]
| '\u0EC6'
| [\u0EDC-\u0EDD]
| '\u0F00'
| [\u0F40-\u0F47]
| [\u0F49-\u0F6A]
| [\u0F88-\u0F8B]
| [\u1000-\u1021]
| [\u1023-\u1027]
| [\u1029-\u102A]
| [\u1050-\u1055]
| [\u10A0-\u10C5]
| [\u10D0-\u10FA]
| '\u10FC'
| [\u1100-\u1159]
| [\u115F-\u11A2]
| [\u11A8-\u11F9]
| [\u1200-\u1248]
| [\u124A-\u124D]
| [\u1250-\u1256]
| '\u1258'
| [\u125A-\u125D]
| [\u1260-\u1288]
| [\u128A-\u128D]
| [\u1290-\u12B0]
| [\u12B2-\u12B5]
| [\u12B8-\u12BE]
| '\u12C0'
| [\u12C2-\u12C5]
| [\u12C8-\u12D6]
| [\u12D8-\u1310]
| [\u1312-\u1315]
| [\u1318-\u135A]
| [\u1380-\u138F]
| [\u13A0-\u13F4]
| [\u1401-\u166C]
| [\u166F-\u1676]
```

```
| [\u1681-\u169A]
| [\u16A0-\u16EA]
| [\u16EE-\u16F0]
| [\u1700-\u170C]
| [\u170E-\u1711]
| [\u1720-\u1731]
| [\u1740-\u1751]
| [\u1760-\u176C]
| [\u176E-\u1770]
| [\u1780-\u17B3]
| '\u17D7'
| '\u17DC'
| [\u1820-\u1842]
| '\u1843'
| [\u1844-\u1877]
| [\u1880-\u18A8]
| [\u1900-\u191C]
| [\u1950-\u196D]
| [\u1970-\u1974]
| [\u1980-\u19A9]
| [\u19C1-\u19C7]
| [\u1A00-\u1A16]
| [\u1D00-\u1D2B]
| [\u1D2C-\u1D61]
| [\u1D62-\u1D77]
| '\u1D78'
| [\u1D79-\u1D9A]
| [\u1D9B-\u1DBF]
| [\u1E00-\u1E9B]
| [\u1EA0-\u1EF9]
| [\u1F00-\u1F15]
| [\u1F18-\u1F1D]
| [\u1F20-\u1F45]
| [\u1F48-\u1F4D]
| [\u1F50-\u1F57]
| '\u1F59'
| '\u1F5B'
```

```
| '\u1F5D'
| [\u1F5F-\u1F7D]
| [\u1F80-\u1FB4]
| [\u1FB6-\u1FBC]
| '\u1FBE'
| [\u1FC2-\u1FC4]
| [\u1FC6-\u1FCC]
| [\u1FD0-\u1FD3]
| [\u1FD6-\u1FDB]
| [\u1FE0-\u1FEC]
| [\u1FF2-\u1FF4]
| [\u1FF6-\u1FFC]
| '\u2071'
| '\u207F'
| [\u2090-\u2094]
| '\u2102'
| '\u2107'
| [\u210A-\u2113]
| '\u2115'
| '\u2118'
| [\u2119-\u211D]
| '\u2124'
| '\u2126'
| '\u2128'
| [\u212A-\u212D]
| '\u212E'
| [\u212F-\u2131]
| [\u2133-\u2134]
| [\u2135-\u2138]
| '\u2139'
| [\u213C-\u213F]
| [\u2145-\u2149]
| [\u2160-\u2183]
| [\u2C00-\u2C2E]
| [\u2C30-\u2C5E]
| [\u2C80-\u2CE4]
| [\u2D00-\u2D25]
```

```
| [\u2D30-\u2D65]
| '\u2D6F'
| [\u2D80-\u2D96]
| [\u2DA0-\u2DA6]
| [\u2DA8-\u2DAE]
| [\u2DB0-\u2DB6]
| [\u2DB8-\u2DBE]
| [\u2DC0-\u2DC6]
| [\u2DC8-\u2DCE]
| [\u2DD0-\u2DD6]
| [\u2DD8-\u2DDE]
| '\u3005'
| '\u3006'
| '\u3007'
| [\u3021-\u3029]
| [\u3031-\u3035]
| [\u3038-\u303A]
| '\u303B'
| '\u303C'
| [\u3041-\u3096]
| [\u309B-\u309C]
| [\u309D-\u309E]
| '\u309F'
| [\u30A1-\u30FA]
| [\u30FC-\u30FE]
| '\u30FF'
| [\u3105-\u312C]
| [\u3131-\u318E]
| [\u31A0-\u31B7]
| [\u31F0-\u31FF]
| [\u3400-\u4DB5]
| [\u4E00-\u9FBB]
| [\uA000-\uA014]
| '\uA015'
| [\uA016-\uA48C]
| [\uA800-\uA801]
| [\uA803-\uA805]
```

```
 | [\uA807-\uA80A]
 | [\uA80C-\uA822]
 | [\uAC00-\uD7A3]
 | [\uF900-\uFA2D]
 | [\uFA30-\uFA6A]
 | [\uFA70-\uFAD9]
 | [\uFB00-\uFB06]
 | [\uFB13-\uFB17]
 | '\uFB1D'
 | [\uFB1F-\uFB28]
 | [\uFB2A-\uFB36]
 | [\uFB38-\uFB3C]
 | '\uFB3E'
 | [\uFB40-\uFB41]
 | [\uFB43-\uFB44]
 | [\uFB46-\uFBB1]
 | [\uFBD3-\uFD3D]
 | [\uFD50-\uFD8F]
 | [\uFD92-\uFDC7]
 | [\uFDF0-\uFDFB]
 | [\uFE70-\uFE74]
 | [\uFE76-\uFEFC]
 | [\uFF21-\uFF3A]
 | [\uFF41-\uFF5A]
 | [\uFF66-\uFF6F]
 | '\uFF70'
 | [\uFF71-\uFF9D]
 | [\uFF9E-\uFF9F]
 | [\uFFA0-\uFFBE]
 | [\uFFC2-\uFFC7]
 | [\uFFCA-\uFFCF]
 | [\uFFD2-\uFFD7]
 | [\uFFDA-\uFFDC]
 ;

fragment ID_CONTINUE
 : ID_START
```

```
| [0-9]
| [\u0300-\u036F]
| [\u0483-\u0486]
| [\u0591-\u05B9]
| [\u05BB-\u05BD]
| '\u05BF'
| [\u05C1-\u05C2]
| [\u05C4-\u05C5]
| '\u05C7'
| [\u0610-\u0615]
| [\u064B-\u065E]
| [\u0660-\u0669]
| '\u0670'
| [\u06D6-\u06DC]
| [\u06DF-\u06E4]
| [\u06E7-\u06E8]
| [\u06EA-\u06ED]
| [\u06F0-\u06F9]
| '\u0711'
| [\u0730-\u074A]
| [\u07A6-\u07B0]
| [\u0901-\u0902]
| '\u0903'
| '\u093C'
| [\u093E-\u0940]
| [\u0941-\u0948]
| [\u0949-\u094C]
| '\u094D'
| [\u0951-\u0954]
| [\u0962-\u0963]
| [\u0966-\u096F]
| '\u0981'
| [\u0982-\u0983]
| '\u09BC'
| [\u09BE-\u09C0]
| [\u09C1-\u09C4]
| [\u09C7-\u09C8]
```

```
| [\u09CB-\u09CC]
| '\u09CD'
| '\u09D7'
| [\u09E2-\u09E3]
| [\u09E6-\u09EF]
| [\u0A01-\u0A02]
| '\u0A03'
| '\u0A3C'
| [\u0A3E-\u0A40]
| [\u0A41-\u0A42]
| [\u0A47-\u0A48]
| [\u0A4B-\u0A4D]
| [\u0A66-\u0A6F]
| [\u0A70-\u0A71]
| [\u0A81-\u0A82]
| '\u0A83'
| '\u0ABC'
| [\u0ABE-\u0AC0]
| [\u0AC1-\u0AC5]
| [\u0AC7-\u0AC8]
| '\u0AC9'
| [\u0ACB-\u0ACC]
| '\u0ACD'
| [\u0AE2-\u0AE3]
| [\u0AE6-\u0AEF]
| '\u0B01'
| [\u0B02-\u0B03]
| '\u0B3C'
| '\u0B3E'
| '\u0B3F'
| '\u0B40'
| [\u0B41-\u0B43]
| [\u0B47-\u0B48]
| [\u0B4B-\u0B4C]
| '\u0B4D'
| '\u0B56'
| '\u0B57'
```

```
| [\u0B66-\u0B6F]
| '\u0B82'
| [\u0BBE-\u0BBF]
| '\u0BC0'
| [\u0BC1-\u0BC2]
| [\u0BC6-\u0BC8]
| [\u0BCA-\u0BCC]
| '\u0BCD'
| '\u0BD7'
| [\u0BE6-\u0BEF]
| [\u0C01-\u0C03]
| [\u0C3E-\u0C40]
| [\u0C41-\u0C44]
| [\u0C46-\u0C48]
| [\u0C4A-\u0C4D]
| [\u0C55-\u0C56]
| [\u0C66-\u0C6F]
| [\u0C82-\u0C83]
| '\u0CBC'
| '\u0CBE'
| '\u0CBF'
| [\u0CC0-\u0CC4]
| '\u0CC6'
| [\u0CC7-\u0CC8]
| [\u0CCA-\u0CCB]
| [\u0CCC-\u0CCD]
| [\u0CD5-\u0CD6]
| [\u0CE6-\u0CEF]
| [\u0D02-\u0D03]
| [\u0D3E-\u0D40]
| [\u0D41-\u0D43]
| [\u0D46-\u0D48]
| [\u0D4A-\u0D4C]
| '\u0D4D'
| '\u0D57'
| [\u0D66-\u0D6F]
| [\u0D82-\u0D83]
```

```
| '\u0DCA'
| [\u0DCF-\u0DD1]
| [\u0DD2-\u0DD4]
| '\u0DD6'
| [\u0DD8-\u0DDF]
| [\u0DF2-\u0DF3]
| '\u0E31'
| [\u0E34-\u0E3A]
| [\u0E47-\u0E4E]
| [\u0E50-\u0E59]
| '\u0EB1'
| [\u0EB4-\u0EB9]
| [\u0EBB-\u0EBC]
| [\u0EC8-\u0ECD]
| [\u0ED0-\u0ED9]
| [\u0F18-\u0F19]
| [\u0F20-\u0F29]
| '\u0F35'
| '\u0F37'
| '\u0F39'
| [\u0F3E-\u0F3F]
| [\u0F71-\u0F7E]
| '\u0F7F'
| [\u0F80-\u0F84]
| [\u0F86-\u0F87]
| [\u0F90-\u0F97]
| [\u0F99-\u0FBC]
| '\u0FC6'
| '\u102C'
| [\u102D-\u1030]
| '\u1031'
| '\u1032'
| [\u1036-\u1037]
| '\u1038'
| '\u1039'
| [\u1040-\u1049]
| [\u1056-\u1057]
```

```
| [\u1058-\u1059]
| '\u135F'
| [\u1369-\u1371]
| [\u1712-\u1714]
| [\u1732-\u1734]
| [\u1752-\u1753]
| [\u1772-\u1773]
| '\u17B6'
| [\u17B7-\u17BD]
| [\u17BE-\u17C5]
| '\u17C6'
| [\u17C7-\u17C8]
| [\u17C9-\u17D3]
| '\u17DD'
| [\u17E0-\u17E9]
| [\u180B-\u180D]
| [\u1810-\u1819]
| '\u18A9'
| [\u1920-\u1922]
| [\u1923-\u1926]
| [\u1927-\u1928]
| [\u1929-\u192B]
| [\u1930-\u1931]
| '\u1932'
| [\u1933-\u1938]
| [\u1939-\u193B]
| [\u1946-\u194F]
| [\u19B0-\u19C0]
| [\u19C8-\u19C9]
| [\u19D0-\u19D9]
| [\u1A17-\u1A18]
| [\u1A19-\u1A1B]
| [\u1DC0-\u1DC3]
| [\u203F-\u2040]
| '\u2054'
| [\u20D0-\u20DC]
| '\u20E1'
```

```
  | [\u20E5-\u20EB]
  | [\u302A-\u302F]
  | [\u3099-\u309A]
  | '\uA802'
  | '\uA806'
  | '\uA80B'
  | [\uA823-\uA824]
  | [\uA825-\uA826]
  | '\uA827'
  | '\uFB1E'
  | [\uFE00-\uFE0F]
  | [\uFE20-\uFE23]
  | [\uFE33-\uFE34]
  | [\uFE4D-\uFE4F]
  | [\uFF10-\uFF19]
  | '\uFF3F'
;
```