CAMPUS
DE EXCELENCIA
INTERNACIONAL

Graduado en Ingeniería Informática

Universidad Politécnica de Madrid

Escuela Técnica Superior de
Ingenieros Informáticos

TRABAJO FIN DE GRADO

Nuevo lenguaje de especificación de experimentos para el simulador GRO

Autor: Marco Catalán Valbuena

Director: Alfonso Rodríguez Patón

MADRID, JUNIO DE 2017

ÍNDICE

1. Introducción y objetivos 1

1.1. Introducción 1

1.2. Objetivos 2

2. Estado del arte 3

2.1. GRO 3

2.2. Otros lenguajes de especificación de circuitos genéticos 3

2.3. Lenguajes de representación de datos 4

3. Planteamiento 7

3.1. Entrada de datos 7

4. Solución Propuesta 10

4.1. Arquitectura 10

4.1.1. Clases 10

4.1.2. Transformación de listas a mapas 11

4.1.3. Herencia de experimentos 12

4.1.4. Sobrescritura de listas a mapas 13

4.1.5. Errores 14

4.2. Implementación 16

4.2.1. Lectura de datos 16

4.2.2. Referencias 17

4.2.3. Errores 19

5. Resultados y conclusiones 21

5.1. Pruebas 21

5.1.1. Prueba1 21

5.1.2. Prueba2 24

5.1.3. Prueba3 25

5.2. Conclusiones 28

5.2.1. Representación en GRO (ProSpec) 28

5.2.2. Representación en YAML 31

6. Referencias 35

7. Anexos **37**

7.1. Anexo A: Definición del lenguaje de definición de experimentos 37

7.2. Anexo B: Código de ejemplo de uso del módulo 42

ÍNDICE DE FIGURAS

1. Repressilator 1
2. Representación como árbol 8
3. Primer nivel configuración 9
4. UML de la clase principal 11
5. UML de la clase Genetics 12
6. Ejemplo de funcionamiento de la herencia 13
7. UML de la validación 16
8. Ejemplo de una referencia válida 18
9. Categorización de errores 20

ÍNDICE DE TABLAS

1. Comparación entre JSON y YAML 6
2. StatusCodes 15
3. Datos aceptados por el módulo 41

Agradecimientos

Quisiera darle las gracias a mi tutor, Alfonso Rodríguez y a los compañeros del LIA, especialmente a Martín Gutiérrez y a Luis Enrique Muñoz Martín, por ayudarme a realizar y corregir esta memoria.

Resumen

La biología sintética es un campo de investigación que está en auge actualmente. Hay numerosos grupos de investigación por todo el mundo, y estos grupos necesitan unas herramientas eficaces para poder trabajar adecuadamente. Para ello el Laboratorio de Inteligencia Artificial (LIA) de la UPM está desarrollando un simulador genético multicelular. Este simulador necesita que los usuarios introduzcan los experimentos genéticos que quieren realizar. Para ello, se ha diseñado un lenguaje de entrada basado en el lenguaje YAML, teniendo como objetivo la simplicidad de lectura por personas sin grandes conocimientos en informática. Se ha creado un módulo para este simulador, cuyo objetivo es, dado un fichero de entrada de configuración, determinar si es correcto, informando al usuario de posibles errores de tipo léxico, semántico o sintáctico, y exponer la información contenida en la configuración al resto de módulos del simulador. Se pretende que este lenguaje de especificación de experimentos no sea únicamente usado en el simulador que se está desarrollando en el LIA, sino que se pueda usar en otros simuladores existentes, como el simulador GRO. La motivación detrás de crear este lenguaje es la simplificación de la especificación de experimentos genéticos. La mayoría de lenguajes de especificación de experimentos existentes, como el lenguaje usado por el simulador GRO, resultan muy complicados a usuarios sin conocimientos de informática, como biólogos, ya que no es completamente declarativo, si no que dentro de los archivos de configuración se necesita especificar código imperativo. Este nuevo lenguaje está basado en ser completamente declarativo, de tal manera que el usuario del simulador no necesite programar. En concreto, se ha elegido el lenguaje base YAML por su simplicidad y legibilidad por humanos. **Palabras clave:** Biología sintética, Bioingeniería, Genética. . .

Abstract

Synthetic biology is a research field that is gaining a lot of traction. There are many research groups around the world, and these groups need proper tools in order to work properly. The LIA (Laboratorio de Inteligencia Artificial) of the UPM is developing a multicellular genetic simulator. This simulator needs users to provide the experiments they want to perform as inputs. An input language based on the YAML markup language has been designed. Its main objective is to make reading it easy for users without a computer science background. A module for the simulator has been developed, which given an input file, checks its validity at multiple levels (lexical, semantical, syntactical) and passes the parsed information on to other modules of the simulator. The objective of this experiment specification language is not only to be used with the mentioned simulator, but also to be compatible with other well-known simulators, like the GRO simulator. The main motivation behind this language is simplifying the specification of genetic experiments. Most genetic specification languages, like the one used by the GRO simulator, are very hard to understand to users without computer science knowledge, like biologists, which are one of the main user groups of the simulator. This is because these languages are not purely declarative, and in the specification files users need to write imperative code. This new language has been designed with the main objective of being purely declarative, so the end user doesn't need to write any code. YAML language has been chosen because of its simplicity and legibility by both machines and humans. **Keywords:** Synthetic biology, Bioengineering, Genetics…

1. INTRODUCCIÓN Y OBJETIVOS

1.1. Introducción

La bioingeniería, también conocida como biología sintética, consiste en la aplicación de métodos y técnicas de la ingeniería al campo de la biología. Se puede expresar que la biología es el hardware con el que construir sistemas artificiales o sintéticos. La biología sintética tiene multitud de aplicaciones, y estas se extienden entre muchos campos, desde biosensores, producción de materiales, exploración espacial, almacenamiento de información, etc.

Los genes son secuencias de ADN con instrucciones para fabricar una determinada proteína. La activación y desactivación de un gen se puede controlar mediante otras proteínas. Los factores que controlan la activación o desactivación de un gen son conocidos como factores de transcripción. Esto nos da la capacidad de diseñar circuitos lógicos como lo haríamos en la electrónica, pero usando genes y proteínas como base. Se puede observar un ejemplo de un circuito genético en la figura 1. Este circuito, conocido como Repressilator[1] es un oscilador que controlará la aparición de una proteína que, en este ejemplo concreto, hace que la célula que la contiene se ilumine con un color verde.

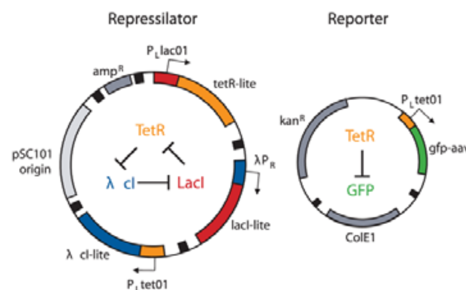


Figura 1: Representación gráfica del circuito genético oscilador conocido como Repressilator

Codificando células con estos genes podemos determinar su comportamiento y programarlas para hacer determinado trabajo. Es aquí donde ayuda el simulador GRO[2], un simulador biológico multicelular que simula el comportamiento de colonias de células con unos determinados genes. Este simulador presenta algunas limitaciones, y es por ello por lo que el LIA está trabajando en un simulador multicelular[3] basado en GRO, pero más adaptado a las necesidades del grupo.

Para definir un circuito genético existen en la actualidad numerosos lenguajes y sis-

temas de entrada (SBOL, Bioblocks, ...), y dado que es un campo muy novedoso, hay mucha heterogeneidad entre estos lenguajes y herramientas, lo cual resulta muy confuso para los usuarios de estas herramientas, que no siempre van a ser informáticos, y hace que el coste asociado a aprender a usar una herramienta nueva sea muy alto.

1.2. Objetivos

Los objetivos que se han buscado en este trabajo son, en orden de realización:

- Aprender cuales son las herramientas y técnicas usadas en el campo de la bioingeniería.
- Diseñar la arquitectura de clases que contenga los datos leídos.
- Implementar un módulo capaz de leer archivos básicos.
- Implementar una batería de pruebas para asegurar el correcto funcionamiento del módulo.
- Implementar la especificación de experimentos al completo.
- Diseñar la arquitectura y el flujo del manejo de errores y avisos.
- Implementar el manejo de errores y avisos.

2. ESTADO DEL ARTE

2.1. GRO

Al igual que los transistores son la base de los circuitos electrónicos, los genes son el elemento base de los circuitos genéticos. Un gen es una secuencia de ADN que genera una proteína cuando está activo. La activación y desactivación de este gen se puede controlar mediante otras proteínas o factores de transcripción, ya sea como un activador, tal que el gen se activa cuando existe una cierta concentración de una proteína, o como un represor, de manera que el gen se activa cuando no está presente una proteína. De esta manera podemos programar el ADN de una célula para que actúe de acuerdo a una función, que tiene como entrada las proteínas presentes y como salida, generará o no una proteína. Esto se asemeja mucho a los circuitos lógicos tradicionales, que en función a unas entradas, representadas como señales eléctricas, generan una salida, representada también como una señal eléctrica.

GRO[2] es un lenguaje que permite programar, modelar y simular el comportamiento de colonias de células. El lenguaje permite especificar todos los atributos de la simulación, desde los comportamientos básicos de las células, hasta eventos externos. Esto hace que GRO sea una herramienta muy potente para especificar experimentos genéticos. Mediante GRO se pueden simular circuitos genéticos como los expuestos en la figura 1 de la introducción.

La desventaja de GRO reside en que al ser un lenguaje tan potente, requiere de una programación, cosa que no siempre es sencilla, especialmente para usuarios sin conocimientos de informática.

2.2. Otros lenguajes de especificación de circuitos genéticos

Existen numerosos lenguajes de especificación de circuitos genéticos. Algunos de los más destacables son:

- Eugene[4]: Eugene es un conjunto de lenguajes que actúan con varios niveles de abstracción con el objetivo de definir circuitos genéticos legibles por humanos y máquinas. Estos niveles son: “System”, “Device”, “Part” y “DNA”. En el ejemplo del Repressilator estaría especificado como un dispositivo, y los genes que conforman el Repressilator, como el GFP serían partes.

- GEC[5]: Genetic Engineering of Cells es un lenguaje de programación orientado a trabajar con células, que permite definir interacciones entre genes y proteínas de una manera modular. Mediante un compilador traduce este lenguaje a bajo nivel, seleccionando secuencias que cumplen las restricciones impuestas y aportan la funcionalidad requerida.
- SBOL[6][7]: SBOL es un estándar abierto para la biología sintética, que intenta unificar como se representa la información sobre diseños en biología sintética. Busca agilizar el movimiento de datos entre usuarios. También esta orientado a ser fácilmente interpretado y generado por máquinas ya que se basa en en el estándar RDF. Como parte de la estandarización, incluye repositorios con pequeñas piezas sobre las que construir.
- CELLO[8]: Cello es un framweork de alto nivel que permite especificar circuitos genéticos, que posteriormente compilará a secuencias de ADN. Permite generar secuencias a partir de especificación de hardware HDL, como Verilog.

2.3. Lenguajes de representación de datos

A lo largo de la historia de la informática han existido numerosos sistemas para representar datos. El uso original de estos lenguajes era poder transferir estructuras de datos complejas en forma de cadenas de texto, operación conocida como serialización, pero su uso se ha extendido mucho, y hoy en día se utilizan en muchos ámbitos, desde la definición de configuraciones hasta incluso en las bases de datos.

El sistema más tradicional es el lenguaje XML[9], estándar abierto usado ampliamente en la web y desarrollado por el World Wide Web Consortium. Define etiquetas que dentro pueden contener a otras etiquetas, o a valores escalares, como cadenas de texto y números enteros. Otro sistema es el lenguaje JSON[10], que nació como una manera de representar los objetos del lenguaje JavaScript. JSON aplica la filosofía de que todo conjunto de datos se puede representar en forma de tres estructuras: listas, mapas y valores escalares. JSON es un lenguaje que está en auge debido a su simplicidad y a su gran flexibilidad.

El lenguaje que se quiere usar para gestionar la definición de experimentos es el lenguaje YAML[11]. El lenguaje YAML comparte muchas similitudes con el lenguaje JSON, dado que ambos categorizan los datos de la siguiente manera, como listas, mapas clave-valor, y datos escalares. La ventaja principal que aporta YAML es que está diseñado teniendo en cuenta la legibilidad tanto por humanos como por máquinas, y a diferencia

de JSON, resulta más sencillo de leer, dado que las indentaciones de las líneas aportan información, en vez de caracteres especiales como llaves y corchetes. En la tabla 1 se puede ver un ejemplo de la misma estructura de datos representada en JSON y YAML. Es importante recordar que todos estos lenguajes aportan la gran ventaja de ser declarativos, puesto que únicamente definen datos y sus relaciones. Esto y la facilidad de lectura del lenguaje YAML hace que para alguien sin conocimientos de informática resulte muy simple leer, modificar y escribir experimentos.

JSON	YAML

{
  "output": {
    "pictures": [
      {
        "center_position":
          {
          "y": 0.6,
          "x": 0.4
        },
        "pattern": "img_",
        "zoom": 0.5,
        "refresh": 5.0,
        "center_cell_id":
          4234,
        "time": [
          100.0,
          300.0
        ],
        "path": "~/User/
          Video",
        "resolution": {
          "width": 1024,
          "height": 768
        }
      }
    ]
  }
}
	output:
	  pictures:
	  - path: ~/User/Video
	    pattern: img_
	    resolution: {width:
	        1024, height:
	      768}
	    refresh: 5.0
	    time: [100.0,
	      300.0]
	    zoom: variable
	    zoom: 0.5
	    center_position: {x
	      : 0.4, y: 0.6}
	    center_cell_id:
	      4234

Tabla 1: Misma estructura de datos en el lenguaje JSON(izquierda) y en el lenguaje YAML(derecha)

3. PLANTEAMIENTO

El problema a solucionar consiste en diseñar e implementar un módulo para el simulador que está siendo desarrollado por el LIA. Tiene que cumplir con los siguientes requisitos:

- Ser capaz de leer correctamente un archivo válido y exponer de manera clara los datos a otros módulos del programa.
- Ser arquitecturado de una manera modular, que permita realizar modificaciones y adiciones de una manera sencilla.
- Ser capaz de detectar cuando un archivo de entrada no es válido, y informar correctamente del error concreto que contiene.
- Ser extensible, de manera que se pueda usar también en otros simuladores, como, por ejemplo GRO.

3.1. Entrada de datos

El módulo tomará como entrada archivos en formato YAML, y expondrá como salida la información que estos contienen. El lenguaje YAML permite definir datos estructurados. Estos datos se pueden representar en forma de árbol. Cada nodo del árbol tiene asignado un tipo, y este puede ser desde un tipo básico (string, integer, boolean) hasta otro nodo o un array. En la figura 2 se puede observar el árbol equivalente a un simple archivo de configuración.

Sobre este lenguaje se definió un estándar de especificación de experimentos, que puede ser consultado en el Anexo A. Este lenguaje contiene la información suficiente para llevar a cabo un experimento. Contiene la información agrupada en 6 grandes categorías. Estas configuraciones deberán ser extensibles, siendo necesario crear un sistema que permita la herencia de configuraciones. En el nodo raíz de cada configuración deberá existir la clave “include”, cuyo valor será una cadena de texto que referenciará al archivo padre. Si no hay un archivo padre será necesario asignar el valor “base_experiment” a esta clave. Tendrán prioridad los datos más específicos, es decir, los hijos sobrescribirán los datos del padre. Cada configuración solo podrá especificar un archivo padre.

También se puede especificar un tema mediante la clave “theme”, y este especificará únicamente aspectos visuales del simulador, como el tamaño de la ventana o el color de

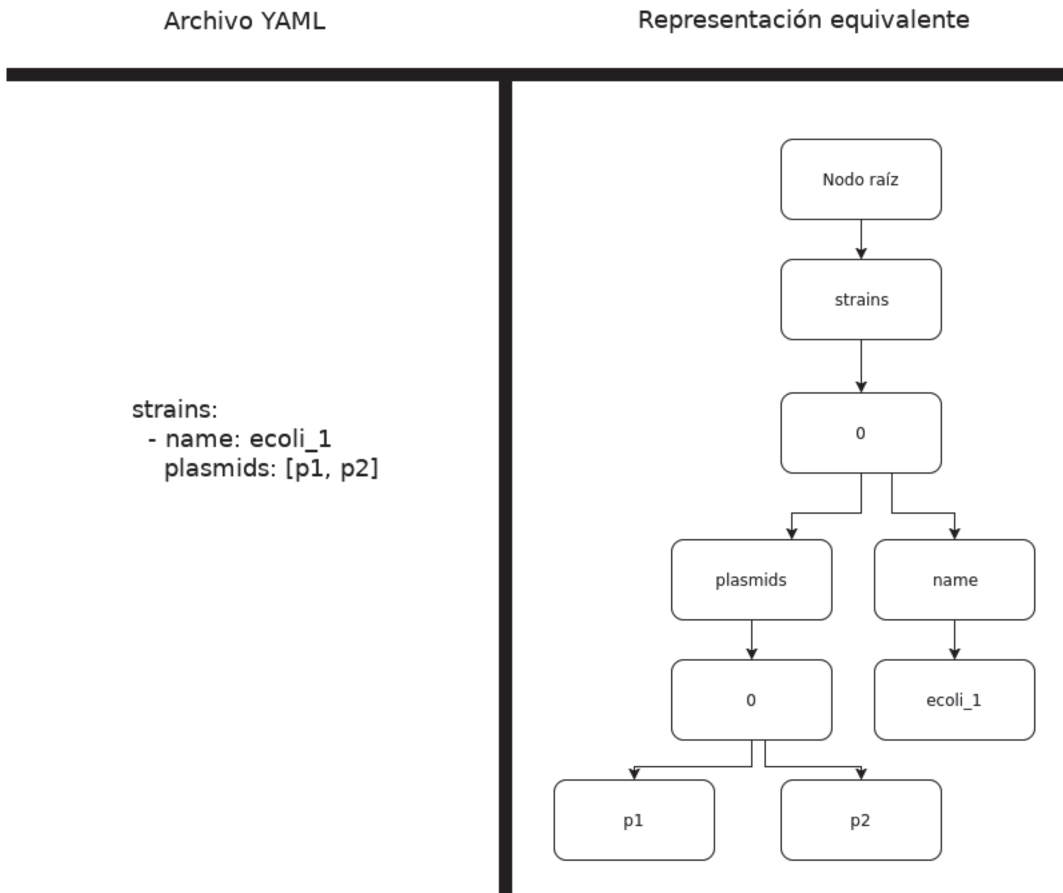


Figura 2: Equivalencia entre un pequeño archivo YAML y un árbol. Los "0" son el resultado de transformar las listas en mapas cuyas claves son los índices de la lista

fondo de la simulación. En la figura 3 se puede ver el primer nivel del árbol que seguirán todos los archivos de configuración.

- Simulation: Define elementos globales de la simulación, como por ejemplo la población máxima de la colonia o la semilla inicial. La semilla es el número origen a partir del cual se generarán todos los números aleatorios. Esto garantiza que siempre se obtendrán los mismos resultados si las semillas y el resto del experimento son iguales. Si no se especifica, se escogerá un número al azar.

- Signals: Permite definir señales. Las semillas son que se usan de manera alternativa para transmitir información. Pueden ser elementos naturales del entorno, o estar operadas manualmente por individuos, que puede emitir señales al entorno, o leerlas del mismo.

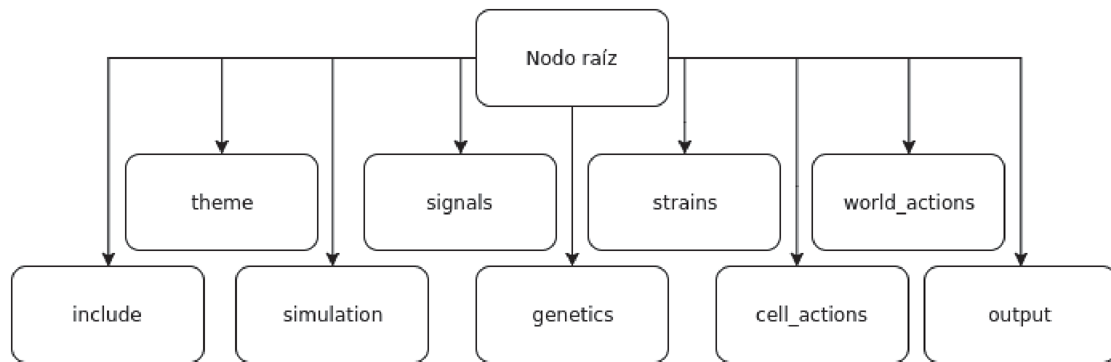


Figura 3: Árbol representativo de las configuraciones con un solo nivel de profundidad

- Genetics: En esta sección se declaran atributos como los elementos genéticos (proteínas o arns), se definen los operones, que actúan a modo de puertas lógicas del sistema, y los plásmidos, que son agrupaciones de operones

- Strains: Los strains, o cepas, en español, son grupos de operones. Corresponde a un grupo de células de una bacteria que comparten unas características comunes.

- Cell Actions/ World Actions: Se define cómo reacciona el simulador, o las células ante determinadas señales o proteínas.

- Output: Almacena la información de salida de información, como por ejemplo donde se van a realizar los volcados de datos del simulador o donde se van a guardar las capturas de pantalla.

4. SOLUCIÓN PROPUESTA

Esta sección se dividirá en las siguientes subsecciones:

- Arquitectura: Arquitectura interna del módulo, esquema de clases, funcionamiento de la herencia y validación
- Implementación: Detalles concretos de implementación de la arquitectura concretos (C++, yaml-cpp)

4.1. Arquitectura

4.1.1. Clases

Para llevar a cabo el diseño del módulo se ha separado el problema en diferentes clases. La clase principal se llama "Configuration" y se expone un método público Parse, que inicializa en la memoria del objeto los valores leídos. Por cada nodo del árbol de entrada que almacenase pares de claves y valores se ha creado una clase. Todas las clases tienen la misma interfaz pública, con un método "Parse" para inicializar el proceso de lectura, un método "validate" para comprobar si la información leída es correcta semánticamente (Los errores léxico/sintácticos se comprueban y devuelven en las llamadas a Parse()) y un atributo de carácter público por cada dato leído. En caso de que una de estas claves fuese a su vez un nodo contenedor de más pares clave/valor, la llamada a Parse del objeto contenedor llamará al método "Parse" del objeto contenido, con el subárbol correspondiente. El diagrama UML de la figura 4 describe las interfaces de estas clases de manera reducida. Se puede observar el diagrama UML de otra clase, en este caso la clase Genetics en la figura 5

Mediante este proceso se puede unificar el proceso de lectura y validado en únicamente dos llamadas a los procedimientos "Parse" y "Validate" de la clase principal, "configuration". Estos irán llamando sucesivamente a los métodos análogos de las clases contenidas hasta completar el árbol de configuración. Esta arquitectura está diseñada teniendo como principal objetivo la modularidad, y hace que añadir un nuevo nodo al árbol sea un proceso sencillo y con un bajo coste. Dado que la interfaz de todas las clases es homogénea, el programa que utilice el módulo también puede elegir no leer toda la configuración y sólo trabajar con una pequeña parte de la misma, con unos cambios mínimos.

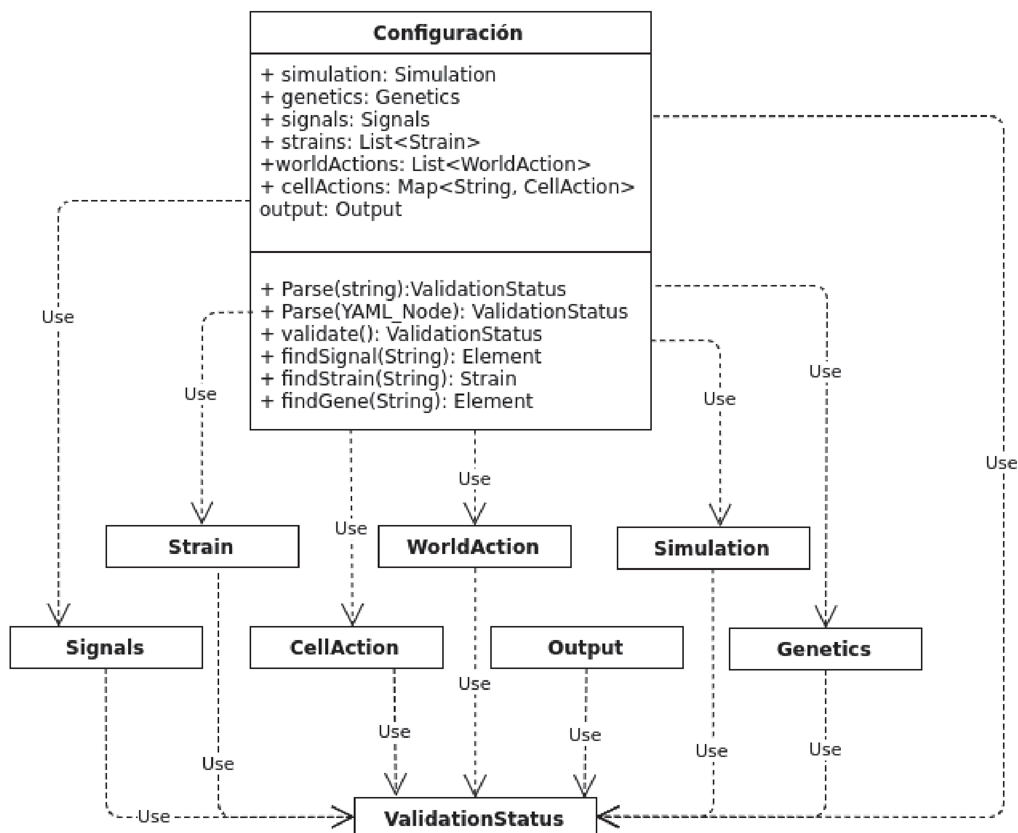


Figura 4: Diagrama UML resumido de la clase Configuration

4.1.2. Transformación de listas a mapas

Para facilitar el uso del módulo, algunos datos que están representados como listas de mapas en el archivo de entrada, se transforman internamente a mapas. Para realizar esto es necesario que todos los elementos de la lista tengan algún atributo en común, y se usará dicho atributo de cada elemento de la lista como clave del nuevo mapa. Por ejemplo a la hora de definir los elementos genéticos, esto podría ser un archivo de entrada:

```
element:
  - name: gfp
    type: protein
    degradation_time: {mean: 30.5, deviation: 0.45}
  - name: yfp
    type: protein
    degradation_time: {mean: 30.5, deviation: 0.45}
```

Internamente no se almacenará la lista Element con dos elementos genéticos, sino que

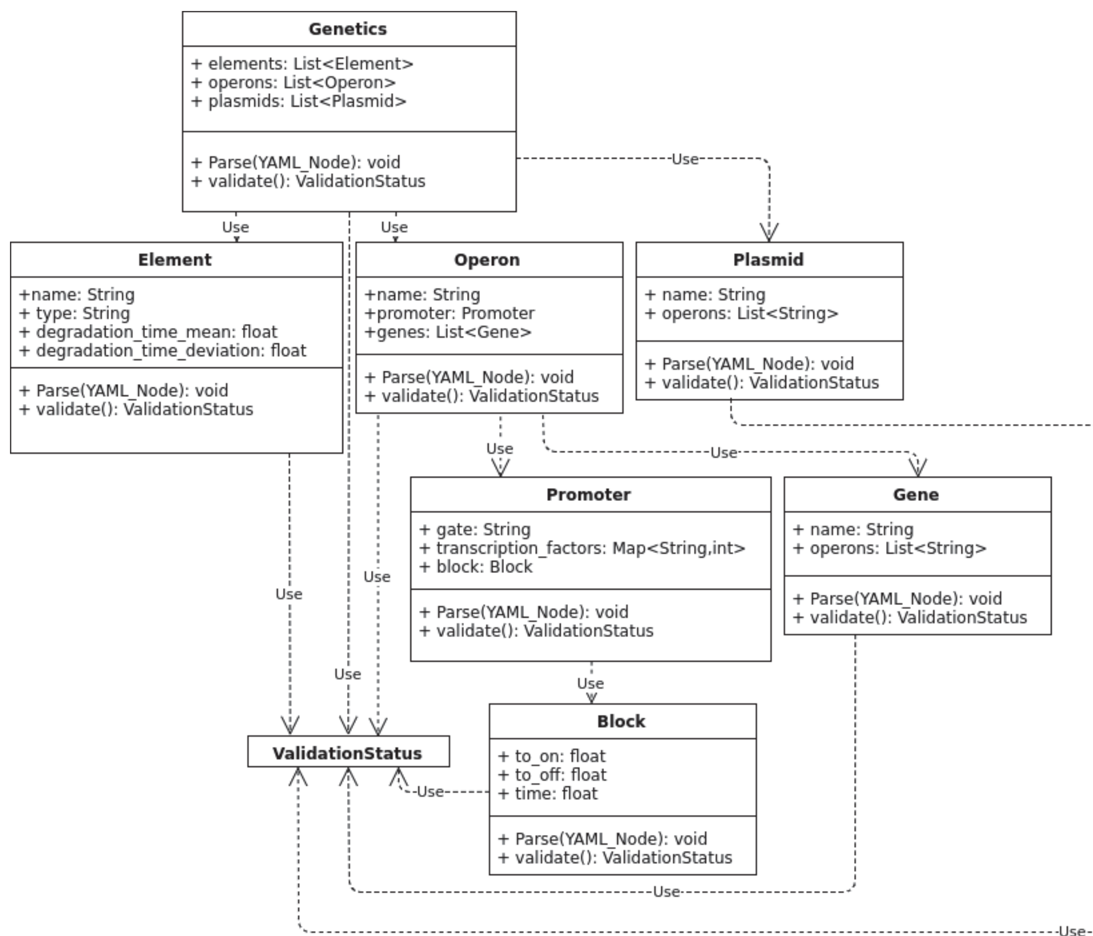


Figura 5: Diagrama UML completo de la clase genetics

se extraerán los nombres de cada uno y element será un mapa de elementos clave valor, teniendo dos claves: “gfp” e “yfp”. Esto se plantea así para no complicar los archivos de entrada, dado que una lista es mas simple de entender y de representar que un mapa.

4.1.3. Herencia de experimentos

La gestión de la herencia de experimentos se realiza en la clase principal, “Configuration”, ya que es a este nivel del archivo YAML donde se almacena la información sobre el experimento padre. El proceso para gestionar la herencia de experimentos es sencillo: En la primera llamada a “Parse” de la clase “Configuration”, el primer valor que se buscará es el valor “include”. Una vez se encuentre este valor, se buscará el experimento referenciado, a no ser que sea “base_experiment”, en cuyo caso se considerará como el experimento base. Si no se puede encontrar este valor se dará un error y se informará al usuario ade-

cuadamente. Si se ha encontrado un experimento padre, se leerá el archivo que contenga el experimento padre, y se llamará al método "Parse" de la instancia actual del objeto con dicho archivo, lo cual rellenará la información que contenta dicho experimento. Una vez haya acabado esta llamada, y si no ha habido ningún error, se llamará al método "Parse" con el experimento actual, y se sobrescribirán o juntarán los valores encontrados, teniendo siempre prioridad el experimento hijo sobre el padre. Este proceso recursivo hace posible que haya varios experimentos anidados, no solo dos. Se puede observar un ejemplo de este proceso con tres experimentos en la figura 6.

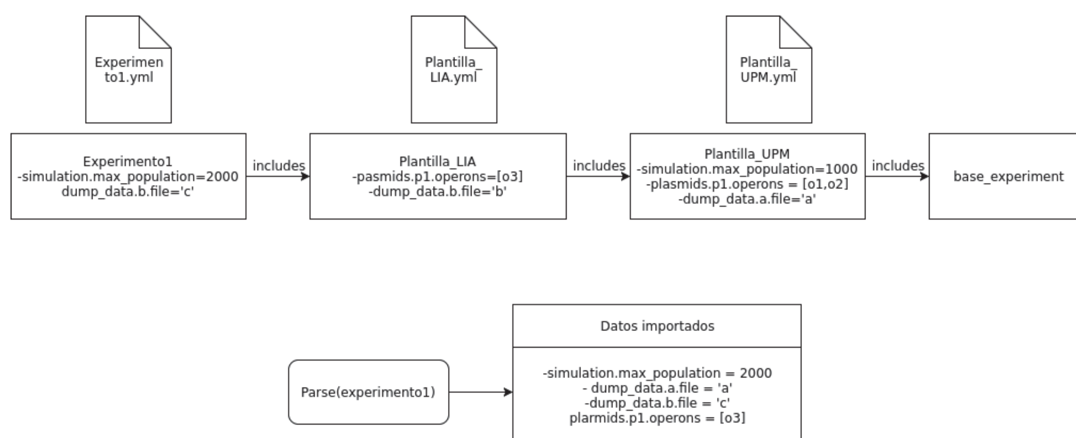


Figura 6: En este ejemplo se plantean 3 archivos con los datos que han especificado y se puede observar el resultado final de cargar un archivo

4.1.4. Sobrescritura de listas a mapas

Cuando una lista o un mapa se encuentran en un archivo de configuración padre, y se lee esa misma clave en una configuración hija, las políticas de sobrescritura que se siguen son las siguientes:

- Lista: Se crea una nueva lista que contiene los elementos de ambas listas, sin importar los duplicados.
- Mapa: Por cada clave del mapa hijo, se comprueba si está en el mapa padre, y en caso de estar, se unen los valores del padre y del hijo, teniendo siempre prioridad los valores del hijo. Si la clave no esta en el padre, se añaden la clave y el valor directamente. Se puede ver un ejemplo de este comportamiento en el apartado 5.1.2.

4.1.5. Errores

Para la detección de errores se han creado dos clases: “Status” y “validationStatus”. La clase status define un error o aviso. Contiene los siguientes atributos:

- StatusCode: Este elemento representa el valor de una enumeración de posibles códigos de error. Aporta información determinada y programable sobre un error o aviso. Está pensado para que ser accedido programáticamente por otros módulos del simulador. Sus posibles valores se especifican en la tabla 2.

- StatusType: Determina si se trata de un error o un aviso. Para ello escoge uno de los dos valores de la enumeración StatusType: “ERROR” o “WARNING”.

- message: Cadena de texto que aporta una descripción legible por humanos del error. Es más concreta que el StatusCode, y aporta información adicional.

La clase validationStatus se utiliza para simplificar el manejo de varios Status. Internamente almacena una lista de instancias de Status, y aporta métodos para ayudar a manejar los estados:

- hasErrors(): Devuelve un valor booleano si en la lista de estados hay alguno que representa un error.

- getErrors(): Devuelve un subconjunto de los estados con únicamente los estados que representan un error.

- getWarnings(): Devuelve un subconjunto de los estados con únicamente los estados que representan un aviso.

- merge(ValidationStatus): Junta las listas de estados del validationStatus dado con la lista de estados de la instancia sobre la que se llama.

- error(StatusCode, std::string): Crea un nuevo error con los valores dados y lo añade a la lista de estados.

- warning(Status::StatusCode, std::string): Crea un nuevo aviso con los valores dados y lo añade a la lista de estados.

Esta clase no es estrictamente necesaria para el manejo de errores, pero sus métodos aportan mucha simplicidad tanto a la hora de implementar las validaciones, como a la hora de comprobar, por un usuario del módulo si hay errores o no. La figura 7 representa el diagrama UML de las clases encargadas de realizar la validación.

StatusCode	Descripción
OK	No ha ocurrido ningún error. Se puede usar para dar mensajes de información
IO_FILE_ERROR	Ha habido un error de entrada/salida al leer un archivo de configuración
FILE_PARSE_ERROR	Ha habido un error realizando el análisis léxico del fichero
TYPE_MISMATCH	Se esperaba un tipo concreto para el valor de una clave y se ha recibido otro
REF_NOT_FOUND	Un valor referenciado no se ha encontrado. Más información sobre las referencias en el apartado de Referencias
REQUIRED_ATTRIBUTE_MISSING	Una clave requerida para el funcionamiento del simulador no tiene valor definido
INVALID_VALUE	El valor de una clave esta restringido a un conjunto de valores, y el valor leído no esta dentro de esos valores
UNKNOWN_ERROR	Error desconocido
WARNING	Aviso, más información en el mensaje del error

Tabla 2: Posibles códigos de un Status

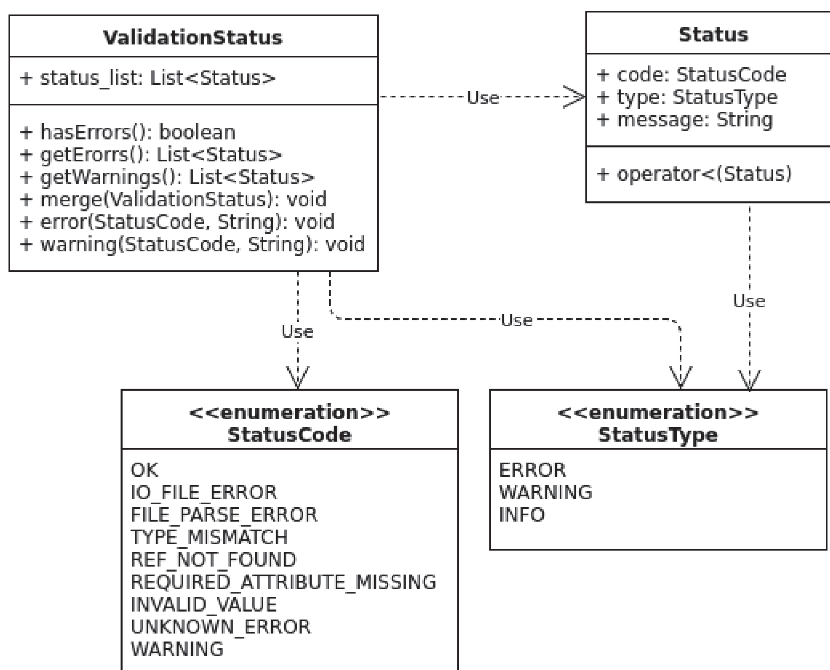


Figura 7: Diagrama UML de las clases que intervienen en la validación

4.2. Implementación

Para la implementación del módulo se ha usado el lenguaje de programación C++, siendo éste el lenguaje que se usa en el resto del simulador. Este lenguaje aporta todas las principales ventajas de C, como son su alto rendimiento y su bajo nivel, añadiendo el paradigma de la programación orientada a objetos, lo que hace que el código sea más limpio y mantenible. Para trabajar con YAML en C++ la librería más usada se llama yaml-cpp. Esta librería, de código abierto y bajo una licencia MIT, es muy adecuada para la realización del módulo, ya que realiza el análisis léxico de un fichero y expone llamadas de API para consultar los datos. También comprueba que el fichero sea válido léxica y sintácticamente.

4.2.1. Lectura de datos

Todas las clases definidas en el apartado de Arquitectura anterior se han implementado en el lenguaje C++ de la siguiente manera: Las clases que corresponden a los nodos con menor profundidad del árbol, que se pueden ver en la figura 3 y la clase Configuration, se han definido mediante dos archivos, un archivo de cabeceras, con formato .h, y un archivo de implementación, con formato .cpp. Para mantener el directorio de trabajo y el

tamaño del módulo pequeño, el resto de clases se han implementado como subclases del nodo padre que las contiene. Para automatizar la compilación de todos estos archivos y los ficheros de pruebas he creado un fichero Makefile, que compila todas las clases, las vincula y genera un ficheros de salida .o para las clase y ejecutables para los ficheros de pruebas.

La llamada al método "Parse" de cualquier clase excepto la principal sigue el siguiente flujo:

```
-Por cada valor que hay que rellenar:
  -Comprueba si en el nodo actual contiene una clave con
     ese valor
  -Si lo contiene:
    -Si el valor es un escalar, lee el valor y lo asigna
       a la variable correspondiente
    -Si el valor es una lista de escalares, itera la
       lista e inserta los valores a la lista
       correspondiente
    -Si el valor es un mapa, busca la subclase adecuada y
        llama al método Parse con el valor como nodo, y
       guarda en memoria una referencia a dicho objeto.
    -Si el valor es una lista de mapas, busca la subclase
        adecuada, e iterando la lista llama al método
       Parse con cada uno de los nodos, guardando todas
       las referencias en la lista
  -Si no lo contiene no se generará ningún error,
     simplemente no se rellenará el valor.
```

La llamada al método "Parse" de la clase principal, Configuration, es muy similar, con la única diferencia en que implementa el mecanismo de herencia descrito en el apartado anterior.

4.2.2. Referencias

En el archivo de configuración pueden aparecer relaciones entre dos valores en las que el valor de una clave referencia a una clave o valor que existe en otro punto del árbol. Se puede observar un ejemplo de una referencia en la figura 8. La etapa de validación

comprueba que estas referencias existan, y en caso de que no lo hagan genera un error. En la clase principal se han implementado funciones auxiliares que dada la string de una referencia obtienen una instancia de la clase asignada a esa referencia. Si la referencia es inválida, el comportamiento no esta definido, dado que debería haberse validado previamente y habría fallado. Un caso de ejemplo concreto ocurre en la definición de las condiciones que activa la acción de una célula. Para acceder a estas condiciones podríamos hacer:

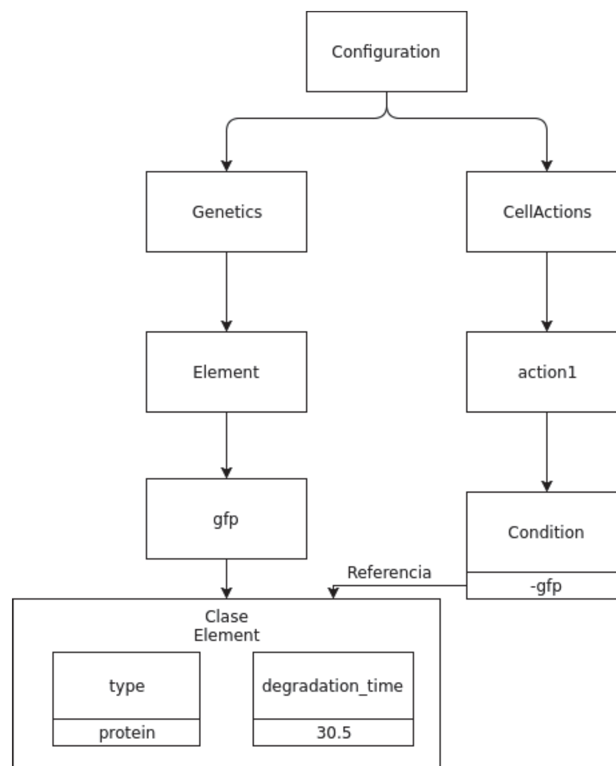


Figura 8: En la figura se muestra un ejemplo de una referencia válida con una posible configuración

```
std::list<std::string> elements = configuration.
    cell_actions["myaccion"].condition;
```

Si se quisiese obtener más información sobre estos elementos genéticos se podría hacer una llamada a “findElement” con dicha string:

```
Genetics::Element element = configuration.findElement(
    elements[0]);
```

4.2.3. Errores

La validación es una parte muy importante del módulo, ya que es la encargada de informar al resto del programa de los errores que haya podido cometer el usuario, y el simulador nunca debería intentar trabajar con un archivo que contiene errores. Para implementar la validación, cada clase tiene un método validate que devuelve un elemento del tipo validationStatus. En el inicio de cada validate se genera una nueva instancia del validationStatus, y si ocurre un error se añade a dicha instancia mediante los métodos ".error" ó ".warning" de la clase ValidationStatus. Si el nodo tiene nodos hijos que necesiten validación llamará al método validate de dicho nodo y cuando obtenga el resultado usará el método ".merge" para juntar los errores de su hijo con sus errores. De esta manera llamar al método validate de la clase principal desencadenará una serie de llamadas por todos los nodos hijos y los resultados se acumularán en la instancia de la clase principal que será devuelta al acabar la función.

Algunos datos no pueden ser validados en el contexto de la clase que les contiene, porque necesitan tener información de otras ramas del árbol. Este es el caso de las referencias, que no se pueden validar dentro de la clase que define la referencia. En este caso la validación se realizará en el validate del ancestro común más cercano, que con el árbol de configuración actual, siempre es la clase principal, pero no tendría porque ser así con otro árbol de configuración. Para implementar estas validaciones referenciadas, se han creado funciones privadas auxiliares para cada clase que contenga referencias, que primero llamarán al validate de la clase, y luego, teniendo toda la información del árbol completo, podrá validar que las referencias se estén haciendo correctamente. En caso de que un valor referencie a otro no definido previamente, se añadirá un error a la lista de errores y se continuará validando.

Como consecuencia de la arquitectura de la aplicación, la validación no parará hasta haber pasado por todo el árbol, incluso si hay errores. Esto es muy ventajoso, ya que permite generar una lista de errores completa, aunque, al igual que pasa en otros sistemas de validación, como en el de los compiladores, solo el primer error es fiable, ya que el resto pueden estar derivados de él.

También se realiza una validación en el método parse, que detectará errores léxicos y sintácticos. Estos errores son generados automáticamente por la librería yaml-cpp y el método parse se encarga de darse cuenta de cuando ha ocurrido un error, y encapsularlo en la clase correspondiente.

Con esto tenemos las tres validaciones necesarias: léxico, sintáctico y semántico, las

dos primeras realizadas al hacer la lectura de datos, y la tercera se debe llamar manualmente después de leer los datos con el método "validate". La figura 9 representa esta categorización y detalla algún error concreto de cada fase.

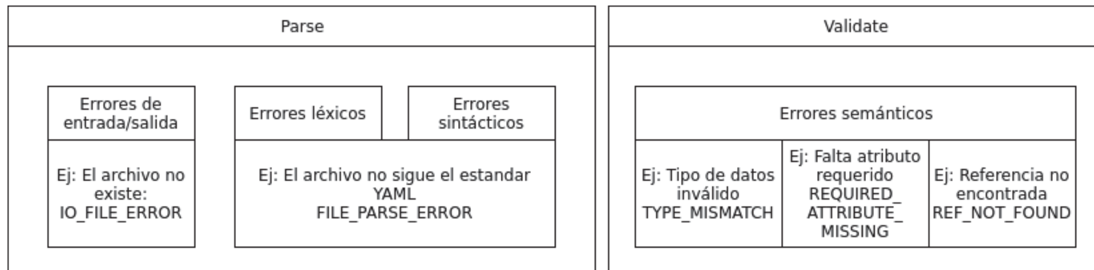


Figura 9: Agrupación de los diferentes tipos de error posibles por método que los puede generar, con ejemplos

5. RESULTADOS Y CONCLUSIONES

5.1. Pruebas

Se ha realizado una serie de pruebas para comprobar el correcto funcionamiento del simulador. La primera prueba consiste en leer un experimento e imprimir alguno de sus datos por pantalla. La segunda consiste en crear un experimento que tenga como experimento padre el experimente de la prueba 1, para comprobar el correcto funcionamiento de la herencia de configuraciones. La tercera prueba se basa en crear un experimento con errores, y determinar si el módulo es capaz de detectarlos y avisar de ellos. Se puede ver un código similar al que se ha utilizado para realizar las pruebas en el apartado 7.2.

5.1.1. Prueba1

Prueba 1: Lectura correcta de un experimento simple

Archivo de entrada

```
include: base_experiment
theme: dark_theme

simulation:
  dt: 0.1
  seed: 123
  max_population: 100000

signals:
  grid:
    type: continuous
    diffusion_method: gro_original
    neighbors: 8

  elements:
    - name: iptg
      diffusion: 0.3
      degradation: 0.4
```

```yaml
      init_value: 0.0


genetics:
  element:
    - name: gfp
      type: protein
      degradation_time: {mean: 30.5, deviation: 0.45}

  operons:
    - name: o1
      promoter:
        gate: AND
        trancription_factors: [gfp, -rfp]
        block: {to_on: 0.5, to_off: 0.1, time: 200.0}
      genes:
        - expresses: gfp
          time: {mean: 30.5, deviation: 0.45}

  plasmids:
    - name: p1
      operons: [o1, o2]


strains:
  - name: ecoli_1
    width: 1.0
    default_growth_rate: 0.1
    division_length: {x: 3.5, y: 4.0}
    division_proportion: {x: 0.4, y: 0.6}
    plasmids: [p1, p2]

cell_actions:
  - emit_area:
```

```
      condition: [-gfp]
      signal: iptg
      concentration: 0.98

world_actions:
  - strain: ecoli_1
    population: 100
    time: [100.4, 90.6]
    circle:
      center: {x: 0.4, y: 0.6}
      radius: 13.8

output:
  dump_data:
  - condition: [gfp]
    file: ~/User/NoWindows.txt

  pictures:
  - path: ~/User/Video
    pattern: img_
    resolution: {width: 1024, height: 768}
    refresh: 5.0
    time: [100.0, 300.0]
    zoom: 0.5
    center_cell_id: 4234
```

Salida

```
Configuration file is: test_experiment1.yml
File parsed. Validating...
Max population = 100000
Strain
        name ecoli_1
        width = 1
CellAction
```

```
        name emit_area
        concentration = 0.98
        signal = iptg
```

5.1.2. Prueba2

Prueba 2: Lectura de un experimento con herencia

Archivo de entrada

```
include: test_experiment1

simulation:
  max_population: 200000

strains:
  - name: ecoli_2
    width: 2.0

cell_actions:
  - emit_area:
      concentration: 0.8
```

Salida

```
Configuration file is: test_experiment2.yml
File parsed. Validating...
Max population = 200000
Strain
        name ecoli_1
        width = 1
Strain
        name ecoli_2
        width = 2
CellAction
        name emit_area
```

```
    concentration = 0.8
    signal = iptg
```

5.1.3. Prueba3

Prueba 3: Lectura de un experimento con errores

Archivo de entrada

```
include: base_experiment

simulation:
  dt: 0.1
  seed: 123
  max_population: 100000

signals:
  grid:
    type: discontinuous # ERROR
    diffusion_method: gro_original
    neighbors: 8

  elements:
    - name: iptg
      diffusion: 0.3
      degradation: 0.4
      init_value: 0.0


genetics:
  element:
    - name: gfp
      type: proteina # ERROR
      degradation_time: {mean: 30.5, deviation: 0.45}

```

```
  operons:
    - name: o1
      promoter:
        gate: AAAND # ERROR
        trancription_factors: [gfp, -rfp]
        block: {to_on: 0.5, to_off: 0.1, time: 200.0}
      genes:
        - expresses: gfp
          time: {mean: 30.5, deviation: 0.45}

  plasmids:
    - name: p1
      operons: [o1, o2]


strains:
  - name: ecoli_1
    width: 1.0
    default_growth_rate: 0.1
    division_length: {x: 3.5, y: 4.0}
    division_proportion: {x: 0.4, y: 0.6}
    plasmids: [p1, p2]

cell_actions:
  - emit_area:
      condition: [-gfpa, -rfpa] # 2 ERRORES
      signal: iptg
      concentration: 0.98

world_actions:
  - strain: ecoli_12 # ERROR
    population: 100
    time: [100.4, 90.6]
    circle:
```

```
      center: {x: 0.4, y: 0.6}
      radius: 13.8
  - signal: mysignal # ERROR
    position: {x: 0.4, y: 0.6}
    concentration: 15.9
    refresh: 5.0
    time: [100.0, 300.0]


output:
  dump_data:
  - condition: [aaagfp] # ERROR
    file: ~/User/NoWindows.txt

  pictures:
  - path: ~/User/Video
    pattern: img_
    resolution: {width: 1024, height: 768}
    refresh: 5.0
    time: [100.0, 300.0]
    zoom: 0.5
    center_cell_id: 4234
```

Salida

```
Configuration file is: test_experiment3.yml
File parsed. Validating...
--------Errors--------
Error: Gene aaagfp is not declared.
Error: Strain ecoli_12 is not declared.
Error: Signal mysignal is not declared.
Error: Gene -gfpa is not declared.
Error: Gene -rfpa is not declared.
Error: Invalid element type value. Valid types are only
   protein and arn.
```

```
Error: Invalid gate value. Only posible gates are: 'TRUE','
    FALSE','YES','NOT','AND','OR','NAND','XOR'
Error: Invalid grid type. Valid values are 'continuous' y '
    discrete'
————Errors————
```

5.2. Conclusiones

El módulo creado simplificará la labor de los usuarios del simulador enormemente, que únicamente necesitarán conocer el lenguaje YAML y las restricciones concretas de la especificación de experimentos. Las configuraciones finales serán más sencillas de leer y escribir, y también serán más sencillas de generar programáticamente. A continuación se van a comparar una especificación del experimento 'repressilator' en GRO, y una especificación del mismo experimento en YAML. El circuito repressilator simula un oscilador mediante el uso de 3 genes diferentes, representado en la figura 1.

5.2.1. Representación en GRO (ProSpec)

```
include gro

set ( "dt", 0.1 );
set ( "population_max", 2000000 );

t := 0;

genes([ name := "TetOperon",
                proteins := {"TetR"},
                promoter := [function := "NOT",
                             transcription_factors := {"
                                LacI"},
                             noise := [toOff := 0.001, toOn
                                 := 0.001, noise_time:=
                                450.0]],
                prot_act_times := [times := {30.0},
                   variabilities := {6.0}],
```

```
                  prot_deg_times := [times := {30.0},
                     variabilities := {2.0}]
         ]);

genes([ name := "LacOperon",
                  proteins := {"LacI"},
                  promoter := [function := "NOT",
                                 transcription_factors := {"cI
                                    "},
                                 noise := [toOff := 0.001, toOn
                                     := 0.001, noise_time:=
                                    450.0]],
                  prot_act_times := [times := {30.0},
                     variabilities := {6.0}],
                  prot_deg_times := [times := {30.0},
                     variabilities := {2.0}]
         ]);

genes([ name := "cIOperon",
                  proteins := {"cI"},
                  promoter := [function := "NOT",
                                 transcription_factors := {"
                                    TetR"},
                                 noise := [toOff := 0.001, toOn
                                     := 0.001, noise_time:=
                                    450.0]],
                  prot_act_times := [times := {30.0},
                     variabilities := {6.0}],
                  prot_deg_times := [times := {30.0},
                     variabilities := {2.0}]
         ]);

genes([ name := "GFPOperon",
                  proteins := {"GFP"},
```

```
                promoter := [function := "NOT",
                             transcription_factors := {"
                               TetR"}],
                prot_act_times := [times := {30.0},
                   variabilities := {6.0}],
                prot_deg_times := [times := {30.0},
                   variabilities := {2.0}]
        ]);

plasmids_genes([ p1 := {"TetOperon", "LacOperon", "cIOperon
   "},
                 p2 := {"GFPOperon"}]);

action({"GFP"},"d_paint",{"1","0","0","0"});
action({"-GFP"},"d_paint",{"-1","0","0","0"});

route1 := "~/Repressilator/";

filename1 := "RepressilatorSingle.csv";
filename2 := "RepressilatorMultiple.csv";

fp1 := fopen ( route1 <> filename1, "w" );
fp2 := fopen ( route1 <> filename2, "w" );

program p() :=
{
        set ("ecoli_growth_rate", 0.011);
        selected:
        {
                dump_single(fp1);
        }
};

program main() :=
```

```
{
        t = 0:
        {
                fprint(fp1, "Time, No Protein, TetR, LacI,
                   cI, GFP, Total\n");
                fprint(fp2, "Time, No Protein, TetR, LacI,
                   cI, GFP, Total\n");
        }

        true:
        {
                t := t + dt;
                dump_multiple(fp2,{"-TetR","-LacI","-cI","-
                   GFP"},{"TetR"},{"LacI"},{"cI"},{"GFP
                   "},{});
        }

        t > 600:
        {
                stop();
        }

        c_ecolis(30, 0, 0, 200, {"p1","p2"}, program p());
        select_random_cell();
};
```

5.2.2. Representación en YAML

```
include: base_experiment
theme: dark_theme

simulation:
  dt: 0.1
  seed: 123
```

```yaml
  max_population: 2000000

signals:
  grid:
    type: continuous
    diffusion_method: gro_original
    neighbors: 8

genetics:
  element:
    - name: TetR
      type: protein
      degradation_time: {mean: 30.0, deviation: 2.0}
    - name: GFP
      type: protein
      degradation_time: {mean: 30.0, deviation: 2.0}
    - name: LacI
      type: protein
      degradation_time: {mean: 30.0, deviation: 2.0}
    - name: cI
      type: protein
      degradation_time: {mean: 30.0, deviation: 2.0}

  operons:
    - name: TetOperon
      promoter:
        gate: NOT
        trancription_factors: [LacI]
        block: {to_on: 0.001, to_off: 0.001, time: 450.0}
      genes:
        - expresses: TetR
          time: {mean: 30.0, deviation: 6.0}
    - name: LacOperon
      promoter:
```

```
      gate: NOT
      trancription_factors: [cI]
      block: {to_on: 0.001, to_off: 0.001, time: 450.0}
    genes:
      - expresses: LacI
        time: {mean: 30.0, deviation: 6.0}
  - name: cIOperon
    promoter:
      gate: NOT
      trancription_factors: [TetR]
      block: {to_on: 0.001, to_off: 0.001, time: 450.0}
    genes:
      - expresses: cI
        time: {mean: 30.0, deviation: 6.0}
  - name: GFPOperon
    promoter:
      gate: NOT
      trancription_factors: [TetR]
    genes:
      - expresses: GFP
        time: {mean: 30.0, deviation: 6.0}

plasmids:
  - name: p1
    operons: [TetOperon, LacOperon, cIOperon]
  - name: p2
    operons: [GFPOperon]


strains:
  - name: ecoli_1
    width: 1.0
    default_growth_rate: 0.1
    division_length: {x: 3.5, y: 4.0}
```

```
    division_proportion: {x: 0.4, y: 0.6}
    plasmids: [p1, p2]

cell_actions:
  - paint_on:
      condition: [GFP]
      concentration: 1
      color: 0x00FF00
  - paint_off:
      condition: [-GFP]
      concentration: 1
      color: 0xFF00FF

output:
  dump_data:
  - confition: []
    file: ~/Repressilator/RepressilatorSingle.csv
  - condition: []
    file: ~/Repressilator/RepressilatorMultiple.csv
```

Como se puede observar, el tamaño del experimento especificado en YAML es ligeramente mayor en este caso, cosa que no tiene que ocurrir siempre, ya que el lenguaje YAML se puede escribir de una manera más compacta. A cambio es mucho más sencillo de leer y entender, ya que únicamente se definen datos. Esto se aprecia especialmente en algunos puntos como la definición de las cepas, siendo en GRO: "c_ecolis(30, 0, 0, 200, "p1","p2", program p())". Esto es más difícil de entender que la definición en YAML, ya que el usuario a primera vista no puede determinar que significa cada uno de los números. Lo mismo ocurre con otras secciones, como las acciones y la gestión de la salida.

6. REFERENCIAS

[1] M. B. Elowitz y S. Leibler. (2000). A synthetic oscillatory network of transcriptional regulators, dirección: http://www.elowitz.caltech.edu/publications/Repressilator.pdf (visitado 06-06-2017).

[2] S. Jang, K. Oishi, R. Egbert y E. Klavins. (2012). Specification and Simulation of Synthetic Multicelled Behaviors, dirección: http://pubs.acs.org/doi/abs/10.1021/sb300034m (visitado 19-05-2017).

[3] M. E. Gutiérrez, P. Gregorio-Godoy, G. P. del Pulgar, L. E. Muñoz, S. Sáez y A. Rodríguez-Patón. (2017). A new improved and extended version of the multicell bacterial simulator gro, dirección: http://pubs.acs.org/doi/abs/10.1021/acssynbio.7b00003 (visitado 19-05-2017).

[4] L. Bilitchenko, A. Liu, S. Cheung, E. Weeding, B. Xia, M. Leguia, J. C. Anderson y D. Densmore. (2011). Eugene – A Domain Specific Language for Specifying and Constraining Synthetic Biological Parts, Devices, and Systems, dirección: http://journals.plos.org/plosone/article?id=10.1371/journal.pone.0018882 (visitado 05-06-2017).

[5] M. Pedersen, M. Lakin, F. Polo, R. Petersen, C. Gravill, N. Dalchau y A. Phillips. (2009). Genetic Engineering of Living Cells, dirección: https://www.microsoft.com/en-us/research/project/genetic-engineering-of-living-cells/ (visitado 05-06-2017).

[6] J. Beal, R. Sidney, R. Grünberg, J. McLaughlin y T. Nguyen. (2016). Synthetic Biology Open Language (SBOL), dirección: http://sbolstandard.org/wp-content/uploads/2016/10/BBF-RFC112-SBOL2.1.0.pdf (visitado 05-06-2017).

[7] N. Roehner, J. Beal, K. Clancy, B. Bartley, G. Misirli, R. Grünberg, E. Oberortner, M. Pocock, M. Bissell, C. Madsen, T. Nguyen, M. Zhang, Z. Zhang, Z. Zundel, D. Densmore, J. H, Gennari, A. Wipat, H. M. Sauro y C. J. Myers. (2016). Sharing Structure and Function in Biological Design with SBOL 2.0, dirección: http://pubs.acs.org/doi/abs/10.1021/acssynbio.5b00215 (visitado 06-06-2017).

[8] A. A. K. Nielsen, B. S. Der, J. Shin, P. Vaidyanathan, V. Paralanov, E. A. Strychalski, D. Ross, D. Densmore y C. A. Voigt. (2016). Genetic circuit design automation, dirección: http://science.sciencemag.org/content/352/6281/aac7341 (visitado 05-06-2017).

[9] T. Bray y J. Paoli. (2008). Extensible Markup Language (XML) 1.0, dirección: https://www.w3.org/TR/REC-xml/ (visitado 16-05-2017).

[10] T. Bray y Ed. (2014). The JavaScript Object Notation (JSON) Data Interchange Format, dirección: https://tools.ietf.org/html/rfc7159 (visitado 16-05-2017).

[11] O. Ben-Kiki, C. Evans e I. döt Net. (2009). YAML Ain't Markup Language (YAML™) Version 1.2, dirección: http://yaml.org/spec/1.2/spec.html (visitado 16-05-2017).

7. ANEXOS

7.1. Anexo A: Definición del lenguaje de definición de experimentos

Como el lenguaje de definición de experimentos esta basado en YAML, su gramática esta definida en la especificación oficial de YAML[11]. Aquí se van a definir los valores que busca el módulo y cual es su significado.

Ruta	Descripción
include	Define cual es la configuración padre. Si no se quiere especificar configuración padre include debe de tener el valor "base_experiment". **Obligatorio.**
theme	Apunta al fichero YML que define el tema. Si no se especifica se cargará uno por defecto.
simulation.dt	Duración del paso de la simulación, en minutos.
simulation.seed	Número a partir del cual se generarán los números aleatorios que vaya a usar el simulador.
simulation.max_population	Número máximo de células que podrán vivir en una simulación.
signals.grid.type	Modo de representación de la señal. Posibles valores continuous(la señal puede tomar cualquier valor entre un rango) o discrete(la señal está presente o no). **Obligatorio**
signals.grid.diffusion_method	Modo de difusión de la señal. Determinará la matriz de difusión a usar.
signals.grid.neighbours	Numero de vecinos que se utilizarán para calcular la malla de difusión.
signals.elements	Lista de señales definidas.
signals.elements[i].name	Nombre de la señal.
signals.elements[i].diffusion	Índice de difusión de la señal.
signals.elements[i].degradation	Índice de degradación de la señal.

signals.elements[i].init_value	Valor de concentración inicial dado a la señal.
genetics.element	Define los elementos genéticos existentes. Esta lista se transforma en un mapa internamente, usando el atributo "name" como claves.
genetics.element[i].name	Nombre del elemento.
genetics.element[i].type	Tipo del elemento. Puede ser "arn" o "protein" únicamente.
genetics.element[i]. degradation_time.mean|deviation	Media y desviación del tiempo de degradación del elemento.
genetics.operons	Lista de operones.
genetics.operons[i].name	Nombre del operón.
genetics.operons[i].promoter.gate	Puerta (promotor) del operón. Posibles valores: TRUE, FALSE, YES, NOT, AND, OR, NAND o XOR.
genetics.operons[i]. promoter.transcription_factor	Elementos genéticos que afectan al operón. Internamente se trata como un mapa que tiene como claves los elementos genéticos y como valores un 1 o un -1, dependiendo de si el elemento tiene "-" como prefijo.
genetics.operons[i].promoter .block{to_on|to_off|time}	Probabilidad de fallo que haga que el gen se quede activo (to_on) o inactivo (to_off) después de un determinado time. Usado para simular ruido.
genetics.operons[i].genes	Lista de proteínas que emite un operón cuando su condición se cumple.
genetics.operons[i].genes[j].expresses	Elemento genético que expresa. Tiene que estar definido en el "genetics.element".
genetics.operons[i].genes[j].time.mean	Tiempo medio que tarda en expresarse dicho gen.
genetics.operons[i].genes[j].time.deviation	Desviación del tiempo que tarda en expresarse dicho gen.

genetics.plasmids	Lista de plásmidos.
genetics.plasmids[i].name	Nombre del plásmido.
genetics.plasmids[i].operons	Lista de operones que forman parte de ese plásmido.
strains	Lista de cepas.
strains[i].name	Nombre de la cepa.
strains[i].width	Anchura de las bacterias.
strains[i].default_growth_rate	Ritmo de crecimiento de la cepa por unidad de tiempo(dt).
strains[i].division_lengthxly	Rango de tiempos en los que una célula se puede dividir.
strains[i].division_proportionxly	Rango de proporciones del tamaño de una célula hija respecto a la célula madre.
strains[i].plasmids	Plásmidos de una cepa.
cell_actions	Acciones de célula
cell_actions[i].'action_name'.condition	Proteínas que activan una acción
cell_actions[i].'action_name'.color	Color. Su funcionamiento es dependiente del tipo de acción.
cell_actions[i].'action_name'.plasmid	Plásmido relacionado con la acción. Su efecto depende del tipo de acción.
cell_actions[i].'action_name'.rate	Ratio de crecimiento. Su funcionamiento es dependiente del tipo de acción.
cell_actions[i].'action_name'.expresses	Activa inmediatamente un gen concreto al cumplirse la condición.
cell_actions[i].'action_name'.benefit	Interacción sobre como una señal afecta al metabolismo. Puede ser positive, negative o neutra.
cell_actions[i].'action_name'.concentration	Concentración de la señal necesaria para que se de la acción.
cell_actions[i].'action_name'.upper_threshold	Límite máximo de la señal para que se de la acción.
cell_actions[i].'action_name'.lower_threshold	Límite mínimo de la señal para que se de la acción.

cell_actions[i].'action_name'.signal	Señal relacionada a la acción. Su efecto depende del tipo de acción.
world_actions	Acciones globales.
world_actions[i].strain	Cepa que se va a generar.
world_actions[i].signal	Señal que se va a generar.
world_actions[i].population	Cantidad de células que se van a generar.
world_actions[i].time	Tiempo, o intervalo en el caso de señales, de actuación del evento. Define cuando, o cada cuanto se van a crear células o señales
world_actions[i].concentration	Concentración de la señal creada.
world_actions[i].refresh	Cada cuanto se va a re-emitir la señal dentro del intervalo de tiempo.
world_actions[i].circlellinear	Determina donde se van a posicionar las bacterias.
world_actions[i].circle.center.xly	Centro del círculo donde se posicionarán las bacterias.
world_actions[i].circle.radius	Radio del círculo donde se posicionarán las bacterias.
world_actions[i].linear.start.xly	Inicio de la linea donde se posicionarán las bacterias.
world_actions[i].linear.end.xly	Fin de la linea donde se posicionarán las bacterias.
world_actions[i].linear.width	Ancho de la linea donde se posicionarán las bacterias.
output.dump_data	Generación de ficheros cuando se cumple cierta condición
output.dump_data[i].condition	Condiciones para la generación del fichero. Tienen que ser elementos genéticos declarados.
output.dump_data[i].file	Fichero de salida.
output.pictures	Generación de imágenes.
output.pictures[i].path	Ruta de las imágenes generadas.
output.pictures[i].pattern	Patrón de las imágenes generadas.

<table>
<tr><td>output.pictures[i].resolution.width|height</td><td>Resolución de las imágenes generadas.</td></tr>
<tr><td>output.pictures[i].refresh</td><td>Cada cuanto tiempo se hará una nueva imagen.</td></tr>
<tr><td>output.pictures[i].time</td><td>Intervalo de tiempo en el cual se capturarán imágenes.</td></tr>
<tr><td>output.pictures[i].zoom</td><td>Zoom de las imágenes. Puede ser 'variable' para ajustar automáticamente.</td></tr>
<tr><td>output.pictures[i].center_position</td><td>Posición central de la imagen.</td></tr>
<tr><td>output.pictures[i].center_cell_id</td><td>Id de la célula central de la imagen. Únicamente puede estar este atributo o "center_position". Si se encuentran ambos se dará un aviso.</td></tr>
</table>

Tabla 3: Descripción de todos los datos que puede leer el módulo del archivo de un experimento

7.2. Anexo B: Código de ejemplo de uso del módulo

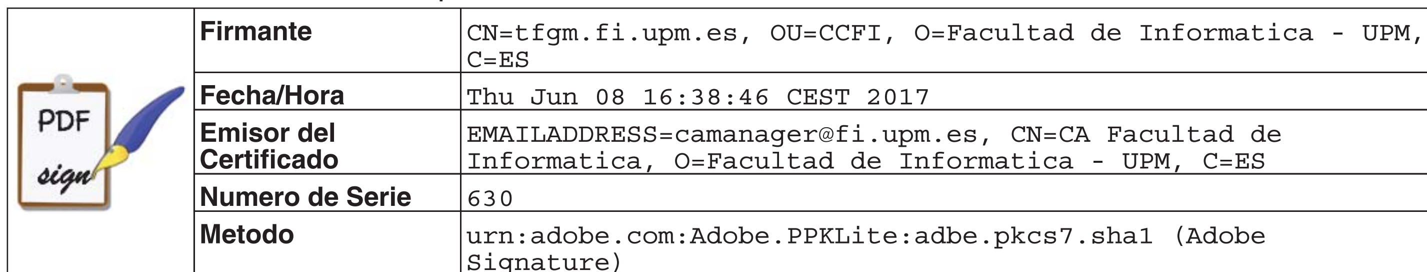
Este es un ejemplo de código usado para probar el módulo, en concreto, se ha usado este código para realizar la mayoría de pruebas. Su labor es cargar un archivo de configuración que lee como argumento y llamar al módulo con el. Si hay errores, imprime los errores y acaba. Si no hay errores, imprime los avisos si los hubiese e imprime algunos de los valores leídos del experimento, para comprobar que se han leído correctamente.

```
#include "configuration.h"
#include <iostream>
#include "status.h"
#include "validationStatus.h"

int main(int argc, char *argv[]){
  if(argc > 1){
    Configuration configuration;
    std::cout << "Configuration file is: " << argv[1] <<
       std::endl;
    ValidationStatus s = configuration.Parse(argv[1]);
    if(!s.hasErrors()){
      std::cout << "File parsed. Validating..." << std::
         endl;
      ValidationStatus s = configuration.validate();
      if(!s.hasErrors()){
        std::list<Status> warnings = s.getWarnings();
        if(warnings.size()>0){
          std::cout << "--------Warnings--------" << std::
             endl;
          for(Status s:warnings){
            std::cout << "Warning: " << s.message << std::
               endl;
          }
          std::cout << "--------Warnings--------" << std::
             endl;
        }
        //Extraemos datos
```

```
        int max_population = configuration.simulation.
            max_population;
        std::cout << "Max population = " << max_population
            << std::endl;
        std::list<Strain> strains = configuration.strains;
        for(Strain s:strains){
          std::cout << "Strain" <<std::endl;
          std::cout << "\tname " << s.name << std::endl;
          std::cout << "\twidth = " << s.width << std::endl
              ;
        }
        std::map<std::string, CellAction> cellActions =
            configuration.cellActions;
        for(std::pair<std::string, CellAction> p:
            cellActions){
          CellAction ca = p.second;
          std::cout << "CellAction" <<std::endl;
          std::cout << "\tname " << p.first << std::endl;
          std::cout << "\tconcentration = " << ca.
              concentration << std::endl;
          std::cout << "\tsignal = " << ca.signal << std::
              endl;
        }

      } else {
        std::list<Status> errors = s.getErrors();
        std::cout << "--------Errors--------" << std::endl;
        for(Status s:errors){
          std::cout << "Error: " << s.message << std::endl;
        }
        std::cout << "--------Errors--------" << std::endl;
      }
    } else {
      std::cout << "Parse error:" << " | " << s.getErrors()
```

```
            .front().message << std::endl;
    }
  } else {
    std::cout << "Usage: "<< argv[0] << " <filename>" <<
        std::endl;
  }
}
```

Este documento esta firmado por



Firmante	CN=tfgm.fi.upm.es, OU=CCFI, O=Facultad de Informatica - UPM, C=ES
Fecha/Hora	Thu Jun 08 16:38:46 CEST 2017
Emisor del Certificado	EMAILADDRESS=camanager@fi.upm.es, CN=CA Facultad de Informatica, O=Facultad de Informatica - UPM, C=ES
Numero de Serie	630
Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)