TRABAJO FIN DE GRADO

TRADUCTOR DE ESPECIFICACIÓN DE EXPERIMENTOS EN YAML PARA EL SIMULADOR MULTICELULAR GRO

GRADO EN MATEMÁTICAS E INFORMÁTICA

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INFORMÁTICOS

UNIVERSIDAD POLITÉCNICA DE MADRID

Autor: Enrique Martínez García
Director: Alfonso Rodríguez-Patón

ÍNDICE

1. Introducción

1.1. Introducción de conceptos biológicos 1

1.2. Objetivos del trabajo 2

1.3. Organización de la memoria 3

2. Estado del arte

2.1. Gro 4

2.2. Otros simuladores celulares 5

3. Planteamiento

3.1. Entrada de datos 7

4. Solución: Traductor de YAML a GRO

4.1. Definición del lenguaje YAML para experimentos biológicos 9

4.2. La interfaz del traductor 11

4.3. El código del traductor 12

4.4. Cómo realizar pruebas 22

4.5. Prueba sin errores 25

4.6. Prueba con errores 31

5. Referencias 39

6. Anexos 41

ÍNDICE DE FIGURAS

Figura1. Screenshots from simulations in GRO. 5

Figura 2. Figura explicativa del lenguaje YAML 8

Figura 3. Captura del inicio de la interfaz 11

Figura 4. Captura del apartado de ayuda de la interfaz 12

Figura 5. Código con el nombre de los botones con acciones 13

Figura 6. Código en el cual se accede a cada sección del fichero YAML 14

Figura 7. Ejemplo de traducción del apartado “simulation”. 14

Figura 8. Ejemplo de traducción del apartado “signals”. 15

Figura 9. Ejemplo de traducción del apartado “genetics→operons”. 16

Figura 10. Ejemplo de traducción del apartado “genetics→plasmids”. 17

Figura 11. Ejemplo de traducción del apartado “cell_actions”. 19

Figura 12. Ejemplo de traducción del apartado “output”. 20

Figura 13. Ejemplo de traducción de los apartados “strains” y “world_actions”. 21

Figura 14. Captura de la interfaz antes de abrir un archivo. 22

Figura 15. Captura de la interfaz después de abrir un archivo 23

Figura 16. Captura de la interfaz al comprobar que es correcto el formato yaml 23

Figura 17. Captura de la interfaz una vez traducido a *GRO* 24

Figura 18. Captura de la interfaz una vez traducido a *GRO* señalando el

botón guardar archivo gro 24

Agradecimientos

Quisiera dar las gracias en primer lugar a mi familia, en especial a mis padres que siempre me han apoyado en esta trayectoria, a mis amigos con los que he compartido estos años de estudios, a mi tutor Alfonso y a mi cotutor Martín por mostrarme este campo nuevo e interesante que es la biología sintética.

Resumen

La biología sintética es la ingeniería de los sistemas biológicos. Su objetivo es poder fabricar circuitos y dispositivos biológicos sintéticos. Esto lo consigue usando la maquinaria biológica natural como software y hardware con el que poder construir circuitos biológicos artificiales. Es un campo que está muy en auge y en el que están trabajando conjuntamente biólogos, matemáticos, físicos e informáticos.

Existe un simulador genético multicelular, llamado *GRO*, desarrollado en una universidad estadounidense y mejorado en el Laboratorio de inteligencia artificial de la UPM. Para poder hacer simulaciones hay que introducir los datos en formato *GRO*. Este lenguaje es bastante complejo para personas que no tienen muchos conocimientos informáticos, ya que, es un leguaje que no es completamente declarativo, tiene también partes del código que son imperativas y por tanto su dificultad es mayor sobre todo si no están familiarizados con lenguajes de programación.

YAML, en cambio, es un lenguaje mucho más sencillo y entendible, ya que es un lenguaje que es completamente declarativo y por tanto no es necesario tener conocimientos de informática previos, es por ello, que se ha creado una estructura rígida de manera que los datos que se introducen en *YAML* tienen que corresponder con *GRO*. La desventaja de *YAML* con respecto a *GRO* es que no se pueden realizar todos los experimentos que *GRO* soporta, pero sí un gran número de ellos.

Primero se creó un corrector que comprueba si un experimento introducido en *YAML*, para hacer pruebas en el simulador *GRO*, es correcto tanto léxica, sintática como semánticamente. Una vez comprobado que es correcto habría que traducirlo al lenguaje que el simulador comprende. Para ello hemos creado un traductor, objetivo de este trabajo fin de grado, así biólogos o bioingenieros pueden introducir datos y hacer simulaciones sin complicaciones.

Este traductor, primero comprueba que el texto introducido en *YAML* es válido, gracias a un corrector hecho por Marco, exalumno de la ETSI Informáticos de la UPM en su TFG. Una vez que está correcto lo traduce a *GRO*. Después, el traductor guarda el

fichero .gro en la carpeta deseada para poder ser introducido al simulador y poder hacer todas las pruebas necesarias y oportunas. Esto supone un gran avance en la facilidad de uso por parte de biólogos y otros usuarios no necesariamente informáticos.

Abstract

Synthetic biology is the engineer of the biology system. Its goal is to be able to manufacture circuits and synthetic biological devices. This is achieved by using the natural biological machinery as software and hardware with which to build artificial biological circuits. It is a field that is very booming and in which they are working together biologist, mathematicians, physicists and computer scientists.

There is a multicell genetic simulator, called GRO, developed in a US university and improved in the Artificial Intelligence Laboratory of the UPM. To be able to do simulations you have to enter the data in GRO format. This language is quite complex for people who do not have much computer knowledge, since, it is a language that is not completely declarative, it also has parts of the code that are imperative and therefore its difficulty is greater especially if they are not familiar with programming languages.

YAML, on the other hand, is a much simpler and more understandable language, since it is a language that is completely declarative and therefore it is not necessary to have previous computer knowledge, that is why a rigid structure has been created in such a way that the data that are entered in YAML have to correspond with GRO. The disadvantage of YAML with respect to GRO is that you can not perform all experiments that GRO supports, but a large number of them. First, a corrector was created that checks whether an experiment introduced in YAML, to make tests in the GRO simulator, is correct both lexically, syntactically and semantically. Once verified that it is correct, it should be translated into the language that the simulator understands. For this we have created a translator, objective of this final degree project, so biologist o bioengineers can enter data and make simulations without complications.

This translator, first check that the text entered in YAML is valid, thanks to a proofreader made by Marco, alumnus of the ETSI Informáticos of the UPM in his TFG. Once it is correct, translate it to GRO. Afterwards, the translator saves .gro file in the desired folder in order in order to be able to do all necessary and opportune tests. This represents a great advance in the ease of use biologist and other users not necessarily computerized.

1. INTRODUCCIÓN

1.1. INTRODUCCIÓN

Un circuito biológico sintético es un circuito de células en el cual, las células son diseñadas para realizar funciones lógicas. La meta es generar células que se puedan mejorar, sustituir o curar cambiando su implementación. Para ello hay que hacer un estudio del circuito génico natural. El primer circuito génico natural estudiado en profundidad fue el operón lac. Un operón es una unidad genética funcional. Está formada por conjuntos de genes que pueden ejercer una regulación de su propia expresión por medio de los sustratos con los que interactúan las proteínas codificadas por sus genes. El operón lac es un operón requerido para el transporte y metabolismo de la lactosa en la bacteria E.Coli. Presenta tres genes estructurales adyacentes, un promotor, un regulador y un operador.

La expresión génica es un proceso, por el cual, los microorganismos procariotas y las células eucariotas transforman la información codificada por los ácidos nucleicos en las proteínas necesarias para su desarrollo, funcionamiento y reproducción con otros organismos. En todos los organismos el contenido del ADN de todas sus células es esencialmente idéntico (salvo en los gametos). No todos los genes se expresan al mismo tiempo ni en todas las células, esto depende de la función de cada célula.

La transcripción significa, pasar la información que hay en el núcleo ADN a otro "lenguaje" llamado ARN, que es el lenguaje que los ribosomas pueden leer para sintetizar aminoácidos. Este proceso es distinto en eucariotas y procariotas.

El código genético es un código que asocia a cada posible triplete de bases uno de los 20 diferentes aminoácidos.

Los ácidos nucleicos son polímeros formados por nucleótidos. Los nucleótidos están formados por un grupo fosfato, un azúcar y una base nitrogenada. El azúcar puede ser desoxirribosa (ADN) o ribosa (ARN).

La traducción es el proceso en el que el ARN mensajero se decodifica para generar una cadena específica de aminoácidos llamada polipéptido.

Un promotor es una región de ADN que controla la iniciación de la transcripción de una determinada porción de ADN a ARN, es decir, promueve la transcripción de un gen.

El factor de transcripción es una proteína que se une a secuencias específicas de ADN (en un promotor), controlando así la transformación de la información genética de ADN a ARN mensajero. Los factores de transcripción hacen esta función individualmente o en conjunto con otros complejos proteicos promoviendo (como activador) o silenciando (como represor) el reclutamiento de la ARN polimerasa a genes específicos.

Los plásmidos son pequeños fragmentos circulares de ADN que contienen de dos a treinta genes y algunos tienen la capacidad para incorporarse o salir del cromosoma bacteriano.

Comprendiendo estos conceptos nos será más fácil entender los experimentos hechos en *YAML* para poder luego traducirlos a *GRO* con la aplicación y poder hacer las pruebas y simulaciones deseadas.

1.2. OBJETIVOS

Este apartado muestra la meta que se persigue en este proyecto. Se han buscado varios objetivos, pero el principal y más importante es hacer un traductor de *YAML* a *GRO* para poder hacer simulaciones de manera más sencilla puesto que *YAM*L es un lenguaje completamente declarativo y, sin embargo, *GRO*, tiene partes declarativas, pero también tiene partes de código imperativo.

Los objetivos secundarios de este trabajo son:

- Informarme y aprender acerca de la biología sintética y lo relativo a ella como son las técnicas que se están usando en la actualidad.
- Implementar un módulo capaz de escoger un archivo *YAML*
- Anexionar el programa que comprueba si un archivo *YAML* está correcto o incorrecto.

1.3 ORGANIZACIÓN

Este trabajo lo he dividido en cinco partes principales y a continuación voy a explicar detalladamente cada una de ellas para su correcto entendimiento.

La primera parte describo una breve introducción. En ella cuento los conceptos biológicos que he tenido que aprender previamente para poder entender el simulador multicelular y comprender mejor *YAML* y *GRO* para poder desarrollar el traductor.

En la segunda hablaré sobre el estado del arte, es decir, contaré como funciona *GRO* y en qué consiste. También explicaré mediante un resumen otros simuladores que se están usando actualmente para poder aportar nuevas soluciones a los problemas biológicos planteados.

En la tercera parte hablaré acerca del planteamiento. El problema a resolver consiste en diseñar una aplicación que transforme un texto en formato *YAML* a su equivalente en formato *GRO*.

En la cuarta parte hablaré acerca del desarrollo que he llevado a cabo para construir el traductor y mostraré partes del código y de la interfaz de la aplicación explicándolas correctamente para, si fuese necesario, en un futuro alguien pueda mejorarlo, ampliarlo o cambiarlo según los requerimientos que necesite. También mostraré ejemplos de programas traducidos para que se vea la gran diferencia de complejidad de *YAML* a *GRO*, motivo que ha hecho necesario la propuesta de este trabajo.

En la última parte mostraré las referencias y a un anexo con la definición del lenguaje de definición de experimentos que realizó un alumno de la ETSI Informáticos de la UPM ya que es necesario para poder entender el lenguaje de entrada del traductor.

2. ESTADO DEL ARTE

En la biología sintética el uso de simuladores es fundamental, ya que permite hacer muchos experimentos y pruebas ahorrando tiempo y dinero. Con estos simuladores se consigue representar el comportamiento real de manera precisa de una bacteria, de colonias de bacterias, de células etc.

Hay un inconveniente en estas simulaciones y es que en la naturaleza las bacterias siempre tienen alguna distorsión u otros inconvenientes que impide representar el comportamiento de manera totalmente real.

2.1 GRO

GRO[1] es un framework creado por Eric Klavins e implementado en C++. Para la visualización de las simulaciones se usa Qt (es el IDE en el que he construido el traductor).

GRO es un lenguaje que permite programar, modelar y simular el comportamiento de colonias de bacterias que están, habitualmente, centradas en un solo individuo con interacciones físicas poco ajustadas a la realidad. Con el lenguaje de *GRO* puedes especificar todos los atributos necesarios para hacer la simulación deseada. Es, por tanto, una potente herramienta para la simulación de comportamientos de bacterias.

Entre las funcionalidades que tiene *GRO*[2] están la simulación del crecimiento bacteriano, la difusión de señales moleculares que permiten transmitir mensajes a lo largo de una colonia e interacciones físicas robustas.

El framework se ha usado además en clases universitarias para explicar biología sintética a ingenieros.

La principal desventaja del lenguaje Gro es que es muy complejo para personas

que no tienen suficientes conocimientos informáticos. Puesto que es, un lenguaje muy potente.

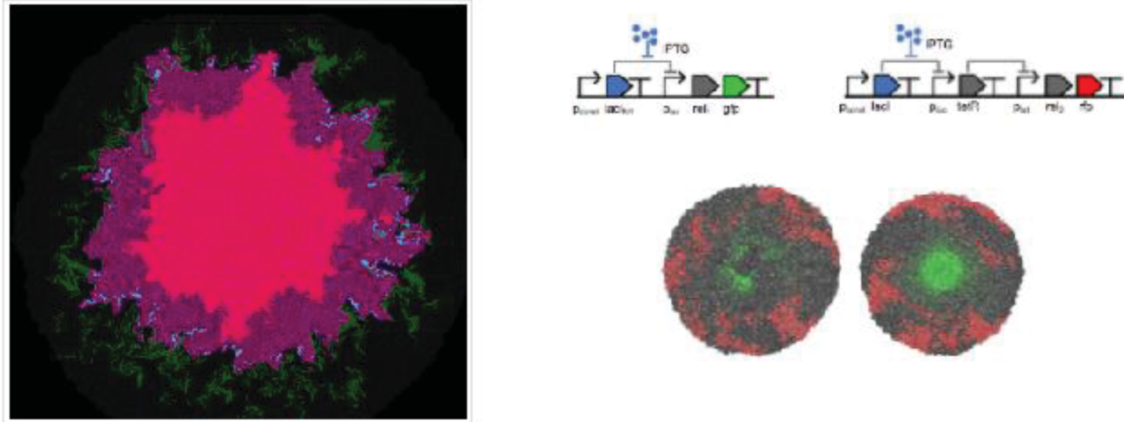


Figura1. Screenshots from simulations in GRO.

2.2 OTROS SIMULADORES

CellModeller[3][4] es un módulo construido en Python para el análisis multicelular 2D. Fue diseñado por la universidad de Cambridge. Este programa fue diseñado para simular el comportamiento de células en general. Ofrece una gran variedad de herramientas para facilitar la configuración del usuario.

BactoSim[5] es un simulador espacial implementado en Java. Fue creado por Antonio García y Alfonso Rodríguez-Patón para simular el comportamiento de bacterias esféricas en una colonia en el que se tienen en cuenta las relaciones espaciales. Este espacio se define como una malla o rejilla discreta en la que se colocan todos los individuos e interaccionan entre ellos.

DiSCUS[6] es un simulador desarrollado en Python. Ofrece en sus simulaciones redes genéticas y comunicación intercelular mediante conjugación. En él, las células se representan en forma de bacilo con una longitud y radio dados. El proceso de conjugación se representa mediante un muelle elástico que conecta a un donante con una receptora. Se puede establecer, además, si las células

transconjugadas tienen la capacidad o no de donar plásmidos.

iDynoMICS[7] es un simulador de código abierto, desarrollado en java que permite representar modelos computacionales de bacterias, en 2D y en 3D. Además, permite configurar una gran cantidad de parámetros respecto al entorno y reacciones químicas que te permiten obtener resultados más realistas. En este simulador se agregó un módulo capaz de trabajar con bacterias con forma capsular, ya que al principio solo se podía con forma esférica. Más tarde se añadió otro módulo que permitió la simulación de puertas AND mediante conjugación. iDynoMics cuenta con una característica que lo hace único, puede simular la presencia de sustancias poliméricas extracelulares. Esta sustancia consigue la formación de colonias bacterianas manteniéndolas unidas entre sí.

3. PLANTEAMIENTO

El problema a solucionar consiste en diseñar una aplicación capaz de recibir un archivo en formato *YAML* (entrada de datos) y transformarlo a su equivalente en formato *GRO* (salida de datos).

3.1 ENTRADA DE DATOS

El traductor toma como entrada un fichero *YAML* en el que se introducen los datos del experimento o prueba que se desea simular. El lenguaje *YAML* es completamente declarativo estructurado, permite definir los datos en forma de árbol. Como se usa frecuentemente *XML* para serialización de datos y es un lenguaje de marcado de documentos, con lo cual, es razonable considerar *YAML* como un lenguaje de marcado ligero.

YAML fue creado pensando que todos los datos pueden ser representados como listas, mapas y datos escalares. Fue creado pensando que fuese muy legible y mapeable a los tipos de datos más comunes en la mayoría de los lenguajes de alto nivel.

Sus características son las siguientes:

- La estructura del documento se denota indentando con espacios en blanco, no se permite usar tabulaciones.
- Los miembros de las listas se denotan empezando por un guion (-) con un miembro por cada línea, o bien entre corchetes ([ ]) y separados por coma espacio (,).
- Los arrays se representan por "clave: valor", bien uno por línea o entre llaves ({ }).
- Un valor de un array asociativo está precedido por (?), con lo cual, podemos construir claves complejas sin ambigüedad.
- Los valores escalares suelen ir sin entrecomillar
- En las comillas dobles los valores especiales van precedidos por (\ )

- Los nodos repetidos se pueden denotar por (&) y ser referidos posteriormente usando (*).
- Los comentarios vienen encabezados por la almohadilla (#) y continúan hasta el final de la línea.

A continuación, voy a mostrar un ejemplo explicativo de código *YAML*.

```
###################
# TIPOS ESCALARES #
###################

# Nuestro objeto raíz (el cual es el mismo a lo largo de todo el
# documento) será un mapa, equivalente a un diccionario, hash,
# u objeto en otros lenguajes.

llave: valor
otra_llave: Otro valor
un_valor_numerico: 100
notacion_cientifica: 1e+12
booleano: true
valor_nulo: null
llave con espacios: valor
# Nótese que los strings no deben estar entre comillas, aunqué también es válido.
llave: "Un string, entre comillas."
"Las llaves tambien pueden estar entre comillas.": "valor entre comillas"

# Los strings de líneas múltiples pueden ser escritos
# como un 'bloque literal' (usando pipes |)
# o como un 'bloque doblado' (usando >)

bloque_literal: |
    Este bloque completo de texto será preservado como el valor de la llave
    'bloque_literal', incluyendo los saltos de línea.

    Se continúa guardando la literal hasta que se cese la indentación.
        Cualquier línea que tenga más indentación, mantendrá los espacios dados
        (por ejemplo, estas líneas se guardarán con cuatro espacios)

bloque_doblado: >
    De la misma forma que el valor de 'bloque_literal', todas estas
    líneas se guardarán como una sola literal, pero en esta ocasión todos los
    saltos de línea serán reemplazados por espacio.
```

Figura 2. Figura explicativa del lenguaje YAML

Imagen obtenida de https://learnxinyminutes.com/docs/es-es/yaml-es/

4. SOLUCIÓN: TRADUCTOR DE YAML A GRO

En esta sección se va a desarrollar la explicación del traductor, sus motivos y algunas pruebas.

4.1 DEFINICION YAML PARA EXPERIMENTOS BIOLOGICOS

Este lenguaje tiene un estándar de especificación de experimentos definido adjuntado en el anexo 1. Con este lenguaje puedes realizar casi cualquier simulación que estimes oportuna.

La primera línea de este lenguaje siempre empieza por "include: nombreArchivoYaml". Si todavía no se ha creado el archivo *YAML* es necesario asignar el nombre "base_experiment".

La segunda línea es para especificar el tema. Esto se consigue mediante la palabra "theme". Esta sentencia es únicamente para especificar los aspectos visuales del simulador.

A continuación, tienen que aparecer los siete grandes grupos que son:

→ Simulation: En este apartado se indicarán en forma de variables globales:

- La duración del paso de la simulación en minutos.
- La semilla inicial que es el número origen a partir del cual se generarán todos los números aleatorios.
- El número máximo de células que podrán vivir en una simulación.

→ Signals: En esta sección se definirán las señales

- Su modo de representación continuo o discreto.
- Su modo de difusión
- El número de vecinos
- La lista de señales definidas y sus atributos

→ Genetics: En este grupo se definen los elementos, los operones y los plásmidos con sus atributos.

→ Strains: en español significa cepas. Son grupos de operones, es decir, grupos de células de una bacteria que tienen características comunes.

→ Cell actions: es el equivalente de actions de gro. Sirve para saber cómo reaccionan las células ante determinadas circunstancias o proteínas.

→ World actions: es muy similar a cell actions pero en lugar de saber cómo reaccionan las células, sirve para saber cómo funciona el simulador ante determinadas circunstancias o proteínas.

→ Output: guarda la información de salida, donde se va a realizar el volcado de datos del simulador o donde se van a guardar las capturas de pantalla.

4.2 LA INTERFAZ DEL TRADUCTOR

Para realizar esta aplicación, lo primero fue hacer el diseño de la interfaz. Un diseño sencillo, intuitivo y en inglés ya que es el idioma universal y amplía así el uso a muchas más personas.

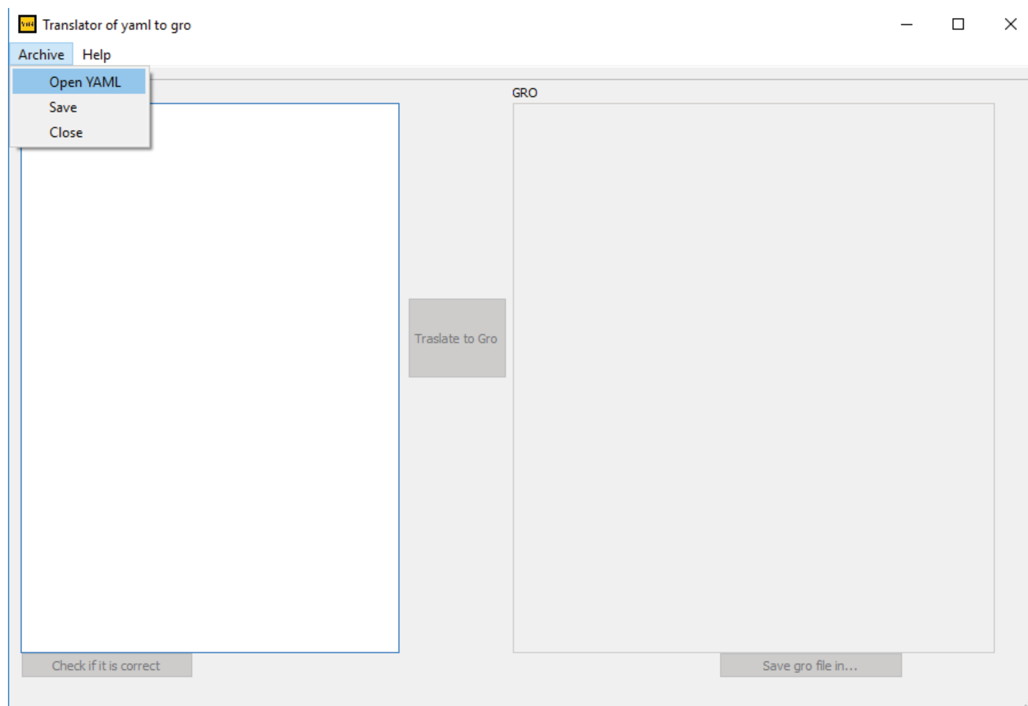


Figura 3. Captura del inicio de la interfaz

Para empezar, tiene una barra de herramientas muy sencilla con dos submenús. El primero es Archive y dentro de él encontramos:

- ➔ Open YAML: sirve para poder importar el documento *YAML*.
- ➔ Save: si quieres hacer cambios en el archivo *YAML* abierto desde la aplicación es necesario, después pulsar en save para poder traducirlo, ya que, la aplicación hace la traducción desde el archivo *YAML*.
- ➔ Close: sirve para cerrar este submenú de archive

Además de archive, cuenta con un menú de ayuda, donde explica el funcionamiento de la aplicación.

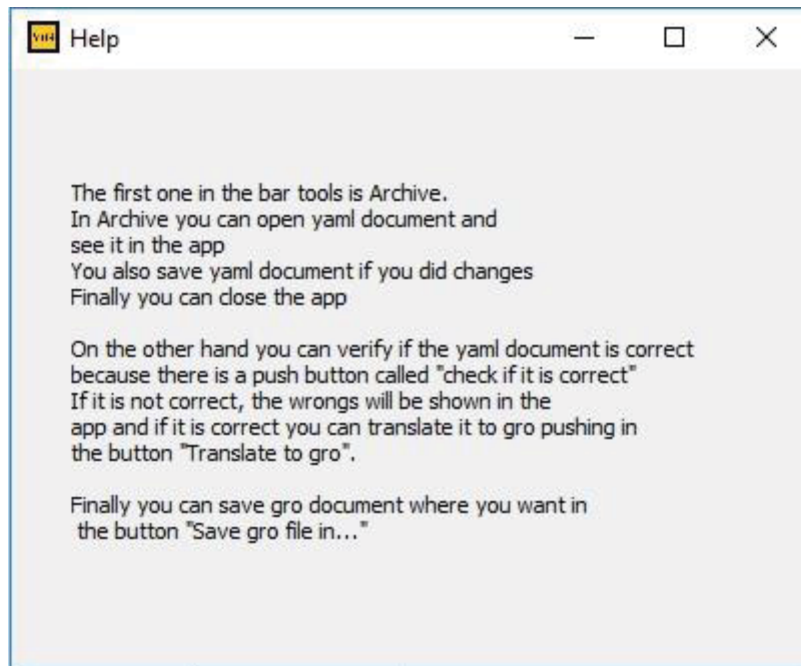


Figura 4. Captura del apartado de ayuda de la interfaz

A parte de esta barra de herramientas cuenta con tres botones que se habilitarán cuando sea cumplan los requisitos para ello, por ejemplo, “Check if it is correct” se habilitará cuando ya se haya abierto un fichero, o “translate to gro” cuando ya se haya verificado que el documento *YAML* es correcto.

4.3 EL CÓDIGO DEL TRADUCTOR

En el entorno de desarrollo Qt, la interfaz se hace gráficamente por lo que no hay que programar para realizar esta parte. Sabemos que, a la hora de programar en C++, hay que programar headers, es decir ficheros *.h* y sources, ficheros *.cpp*.

Entonces, cree dos ficheros headers, uno de la pantalla que se muestra cuando pulsas en “help” y el otro que contiene la ventana principal con los slots necesarios y las variables y métodos globales.

A continuación, voy a mostrar los slots necesarios para esta aplicación

```
private slots:
    void on_actionAbrir_YAML_triggered();

    void on_actionGuardar_triggered();

    void on_actionCerrar_triggered();

    void on_botonComprobacion_clicked();

    void on_exportarGro_clicked();

    void on_botonTraducir_clicked();

    void on_actionHelp_triggered();
```

Figura 5. Código con el nombre de los botones con acciones

Los nombres son muy evidentes e intuitivos para poder reusarlos fácilmente cuando sea necesario.

En el archivo *cpp* de la ventana principal he implementado métodos de una manera que si, se añadiesen más funcionalidades a *GRO* y se ampliasen también a *YAML* puedan implementarse en el traductor de forma sencilla. Esto es debido, a la existencia de un método que lee el archivo *YAML* y lo almacena cada línea en una posición de un *QStringList*, eliminando las tabulaciones, espacios en blanco, comentarios y saltos de línea.

A continuación, cuando se pulsa el botón “translate to gro” recorre dicho *QStringList* y si, en alguna posición, encuentra una línea que contenga el nombre de una de las siete grandes categorías entra en el método de dicha categoría, por lo que, si se añaden posteriormente más categorías sería muy fácilmente reusable esta aplicación. Lo podemos comprobar en este fragmento de código

```
for(int i = 0; i < linesYaml.size(); i++)
{
    if(linesYaml[i].contains("simulation"))
    {
        i = metodoSimulation(i);
    }
    if(linesYaml[i].contains("signals"))
    {
        i = metodoSignals(i);
    }
    if(linesYaml[i].contains("genetics"))
    {
        i = metodoGenetics(i);
    }
    if(linesYaml[i].contains("strains"))
    {
        i = metodoStrain(i);
    }
    if(linesYaml[i].contains("cell_action"))
    {
        i = metodoCellActions(i);
    }
    if(linesYaml[i].contains("world"))
    {
        i = metodoWorldActions(i);
    }
    if(linesYaml[i].contains("output"))
    {
        i = metodoOutput(i);
    }
}
```

Figura 6. Código en el cual se accede a cada sección del fichero *YAML.*

Una vez que entra en “simulation” la traducción tiene que ser de la siguiente manera:

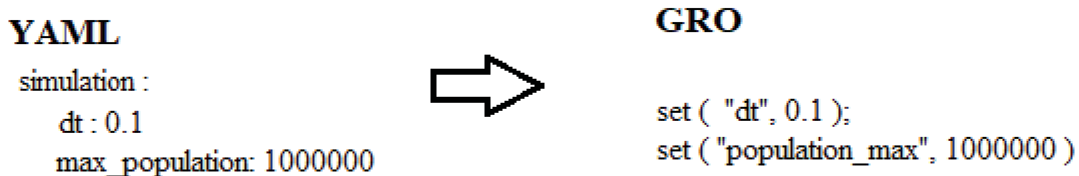


Figura 7. Ejemplo de traducción del apartado “simulation”.

Podemos comprobar que la traducción de “simulation” es bastante sencilla e intuitiva, pero es más amigable en *YAML* que en *GRO*.

La sección “signals” también tiene una traducción muy directa. La primera parte es “grid” y cuenta con los siguientes apartados: “type”, “diffusion_method” y “neighbors” que se traducen a *GRO* en una sola línea. La segunda parte, “element” consta de: “name”, “degradation” y “diffusion” que se traducen a *GRO* también en una sola línea como podemos ver en el ejemplo de a continuación:

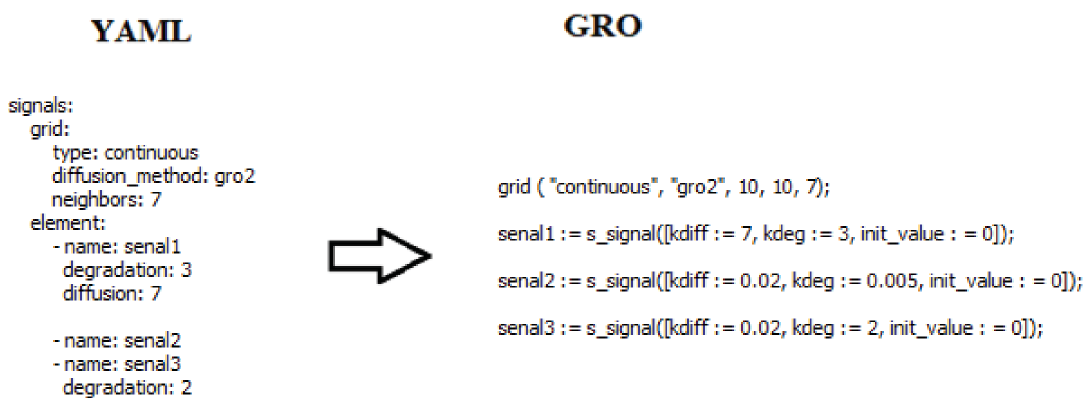


Figura 8. Ejemplo de traducción del apartado “signals”.

La siguiente parte, “genetics” ya es mucho más compleja y la traducción no es directa. En el primer apartado, “element”, aparecen los factores de transcripción y sus propiedades para luego, en el segundo apartado, “operons”, relacionar cada operón y sus propiedades con su factor de transcripción, ya que, en el lenguaje *GRO* aparece, en una misma sentencia, cada operón con su factor de transcripción y las propiedades, tanto del operón como del factor de transcripción. En esta sección, también había que comprobar que los factores de transcripción de los operones estuvieran en “element”, de lo contrario, se producirían errores. Cada operón tiene sus propiedades y estas, a su vez, tienen subapartados para poder explicarlas correctamente. En esta parte, se nota la diferencia de trabajar en *YAML* y hacerlo en *GRO*, ya que, en *YAML* aparece cada propiedad y elemento por su nombre y su valor, mientras que, en *GRO*, en una sola sentencia aparecen todas las propiedades del operón y de forma poco evidente y más confusa. A continuación, voy a mostrar un ejemplo de la descripción de un operón en *YAML* y su equivalente en *GRO*:

YAML

```
genetics:
  element:
    - name: LacI
      type: protein
      degradation_time: {mean: 30.0, deviation: 2.0}
  operons:
    - name: TetOperon
      promoter:
        gate: NOT
        transcription_factors: [LacI]
        block: {to_on: 0.001, to_off: 0.001, time: 450.0}
      genes:
        - expresses: TetR
          time: {mean: 30.0, deviation: 6.0}
```

GRO

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

Figura 9. Ejemplo de traducción del apartado "genetics→operons".

Podemos apreciar la diferencia entre programar los experimentos en *YAML* y en *GRO*.

También, dentro de "genetics", está el subapartado "plamids". Este subapartado, tiene una traducción directa e intuitiva de *YAML* a *GRO* y no hay apenas dificultad entre un lenguaje y el otro, en este caso.

YAML

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

GRO

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

Figura 10. Ejemplo de traducción del apartado "genetics→plasmids".

La siguiente parte, "strains", no tiene traducción directa, de hecho, su traducción no va a continuación de "genetics", así que he tenido que almacenar los datos en arrays globales para su posterior utilización, que explicaré más adelante.

A continuación, puede aparecer "cell_actions" y las acciones son las siguientes:

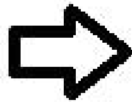
- ➔ "paint(green, red, yellow, cyan)": pinta la bacteria del color especificado. El color es dado por una mezcla de verde, rojo, amarillo y azul. Esta instrucción es absoluta e impone el color sin tener en cuenta el color que tenía anteriormente la bacteria
- ➔ "d_paint(green, red, yellow, cyan)": similar a la acción anterior pero esta tiene en cuenta el color anterior y suma o resta el color especificado en esta acción.
- ➔ "die": mata a la bacteria y la hace desaparecer de la simulación. Esta acción no requiere parámetros
- ➔ "conjugation(plasmid, rate)": esta acción copia un plásmido de la bacteria origen a una bacteria vecina aleatoria. Esta copia se realiza por el promedio "rate", dado por el número medio de conjugaciones que ocurren durante el tiempo de vida de la bacteria.
- ➔ "directed_conjugation(plasmid, rate)": es similar a la acción anterior, salvo que esta considera un mecanismo en la bacteria destino que puede no permitir la entrada del plásmido. Cuando la entrada está cerrada al plásmido en la bacteria de destino, esta bacteria de destino no es

considerada como un vecino viable y, por lo tanto, la selección del destino está dirigida a todos los demás vecinos viables. El parámetro de velocidad en esta acción tiene el mismo significado que en la acción conjugada.

- ➔ "lose_plasmid(plasmid)": acción que borra el plásmido de la lista de plásmidos que residen en la bacteria.
- ➔ "set_entry_exclusion(plasmid)": establece una restricción al plásmido que entra en la bacteria actual.
- ➔ "remove_entry_exclusion(plasmid)": borra la restricción establecida al plásmido que olvida su entrada a la bacteria actual.
- ➔ "set_growth_rate(rate)": establece el promedio que la bacteria crecerá.
- ➔ "emit_cross_feeding(signal_id, concentration)": esta acción hace que la bacteria actual emita la cantidad de "concentration" de la señal "signal_id".
- ➔ "read_quorum_sense(signal_id, comparison,threshold, protein)": siente una cierta concentración de la señal del entorno y si satisface "comparison" y un "threshold" entonces "protein" es activada.
- ➔ "absorb_quorum_sense(signal_id, comparison, threshold, protein)": muy parecida a la anterior solo que esta además de sentir también absorbe cierta concentración de la señal.

YAML

```
cell_actions:
  - paint:
      condition: [GFP]
      concentration: 1
      color: [1,5,8,58]
  - d_paint:
      condition: [-GFP]
      concentration: 1
      color: [5, 5 , 7, 9]
  - conjugation:
      condition: [GFP, LacI]
      plasmid: p1
      rate: 2.3
```



GRO

```
action(GFP, "paint" , {"1","5","8","58"});

action({-GFP}, "d_paint" , {"5","5","7","9"});

action({"GFP","LacI"}, "conjugation" , {"p1","2.3"});
```

Figura 11. Ejemplo de traducción del apartado “cell_actions”.

El siguiente apartado importante es “world_actions”. Las acciones globales pueden ser:

- ➔ Strain: es la cepa que se va a generar.
- ➔ Signal: es la señal que se va a generar.
- ➔ Population: es la cantidad de células que se van a generar.
- ➔ Time: es el tiempo o intervalo de actuación del evento. Define cuando o cada cuanto se van a crear células o señales.
- ➔ Concentration: es la concentración de la señal dada.
- ➔ Refresh: determina cada cuanto se va a re-emitir la señal dentro del intervalo intervalo de tiempo.
- ➔ Circle: determina donde se van a posicionar las bacterias en forma de circulo
    - o Center: centro del circulo donde se posicionarán las bacterias
    - o Radius: radio del círculo donde se posicionarán las bacterias

➔ Linear: determina donde se van a posicionar las bacterias en forma de lineal

  - Start: determina el inicio de la línea donde se posicionarán las bacterias.
  - End: determina el fin de la línea donde se posicionarán las bacterias.
  - Width: determina el ancho de la línea donde se posicionarán las bacterias.

Estas son las posibles acciones globales, su traducción es directa y se sitúa al final del documento *GRO*.

La última sección es “output”, contiene el nombre de los ficheros de salida cuando se cumple cierta condición. Esta condición tiene que ser de elementos genéticos declarados.

En este apartado también se pueden generar imágenes, además puedes elegir ciertos atributos de ellas según sean o no, necesarios.

La traducción en el caso de este apartado es también muy directa como podemos ver a continuación:

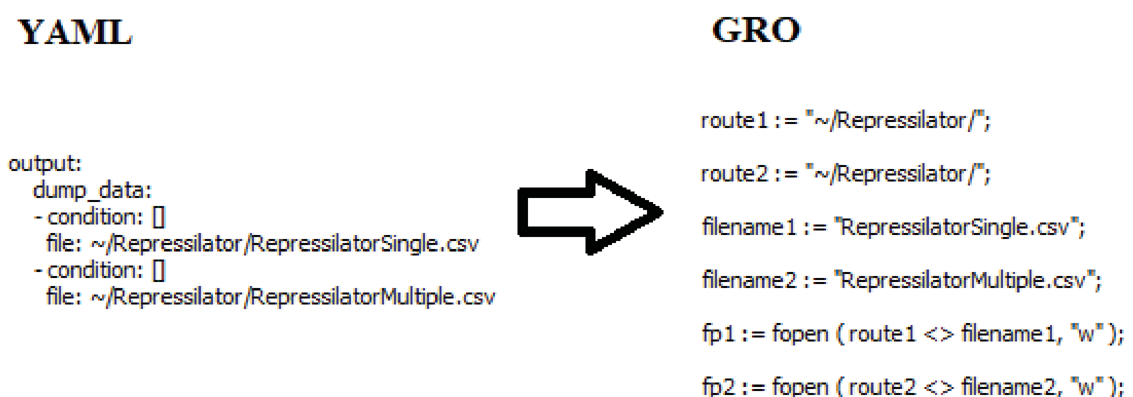


Figura 12. Ejemplo de traducción del apartado “output”.

Ahora solo falta explicar las traducciones de “strains” y “world_actions” que no hemos hecho anteriormente ya que, aunque en *YAML* se especifican en ese orden, dicha parte en *GRO*, se sitúa al final del fichero.

YAML

```
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]
  - name: ecoli_2
    width: 1.0
    default_growth_rate: 0.2
    division_length: {x: 3.5, y: 4.0}
    division_proportion: {x: 0.4, y: 0.6}
    plasmids: [p1, p3]


world_actions:
  - strain: ecoli_2
    population: 300
    time: [100.4, 90.6]
    circle :
      center: {x: 0.5, y: 0.7}
      radius: 15.8
  - strain: ecoli_1
    population: 100
    time: [100.4, 90.6]
    circle :
      center: {x: 0.4, y: 0.6}
      radius: 13.8
```

GRO

```
program p1() :=
{
        set ("ecoli_2_growth_rate", 0.2);
        selected:
        {
                dump_single(fp1);
        }
};

program p2() :=
{
        set ("ecoli_1_growth_rate", 0.1);
        selected:
        {
                dump_single(fp2);
        }
};


c_ecolis(300, 0.5, 0.7, 15.8, {"p1,p3"}, program p1())

c_ecolis(100, 0.4, 0.6, 13.8, {"p1,p2"}, program p2())
```

Figura 13. Ejemplo de traducción de los apartados “strains” y “world_actions”.

Podemos comprobar que la traducción no es directa y a priori no se puede comprender a simple vista. Ahora nos fijamos en el framento de código *YAML*, se puede observar que el nombre de las cepas del apartado “strains” coincide con el nombre de las cepas del apartado “world_actions”, aunque no se encuentren en el mismo orden. Esto es necesario para el correcto funcionamiento de la traducción. Se puede observar también, que *YAML* al ser completamente declarativo no se pueden realizar tantas variantes como en *GRO*.

4.4 COMO REALIZAR PRUEBAS

A continuación, vamos a explicar mediante ilustraciones los pasos que hay que seguir para poder realizar esta traducción:

1º Abre el fichero que desea traducir

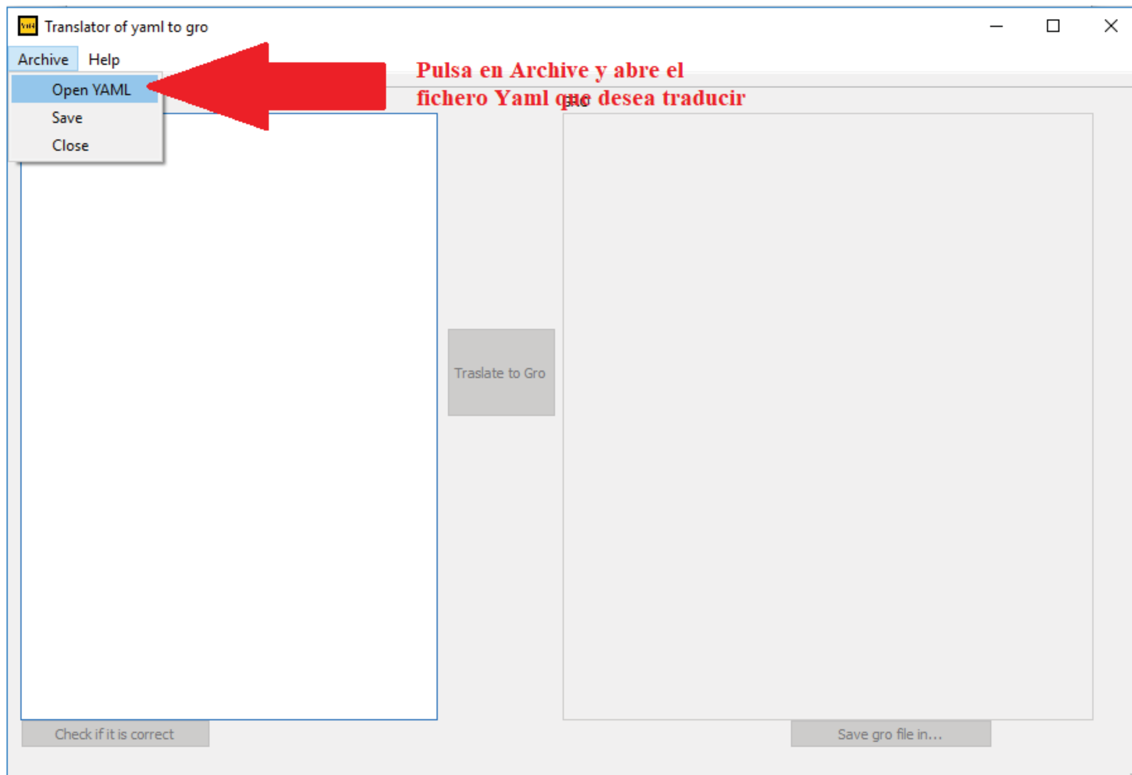


Figura 14. Captura de la interfaz antes de abrir un archivo

2º (OPCIONAL)Una vez que está abierto y se muestra, si desea cambiar partes del código en la aplicación, antes de traducir, debe guardar los cambios

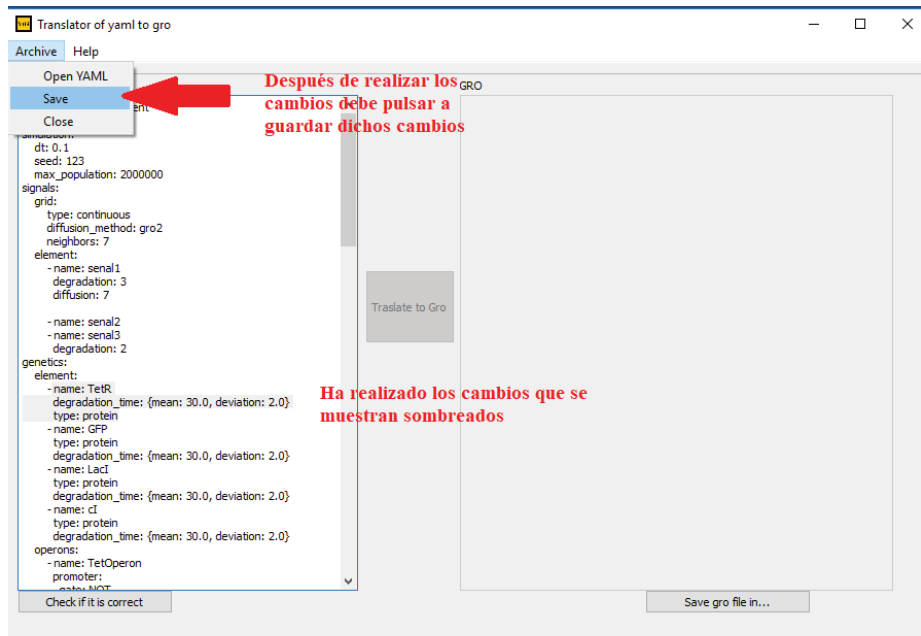


Figura 15. Captura de la interfaz después de abrir un archivo

3º Hay que realizar una verificación que compruebe si el fichero *YAML* es correcto

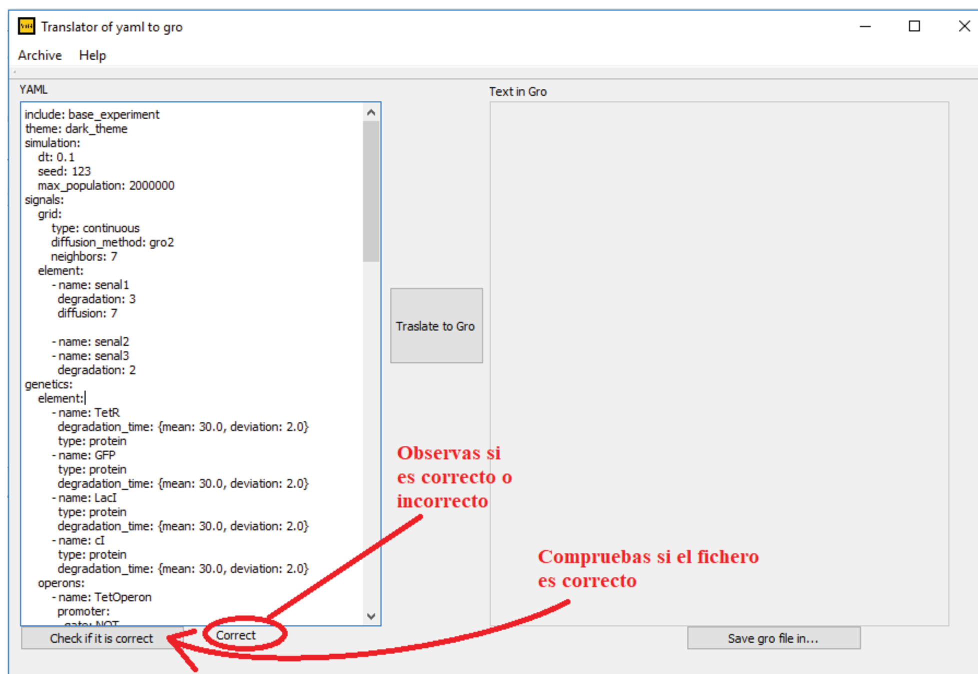


Figura 16. Captura de la interfaz al comprobar que es correcto el formato yaml 4º En el

caso de que sea correcto podemos proceder a la traducción

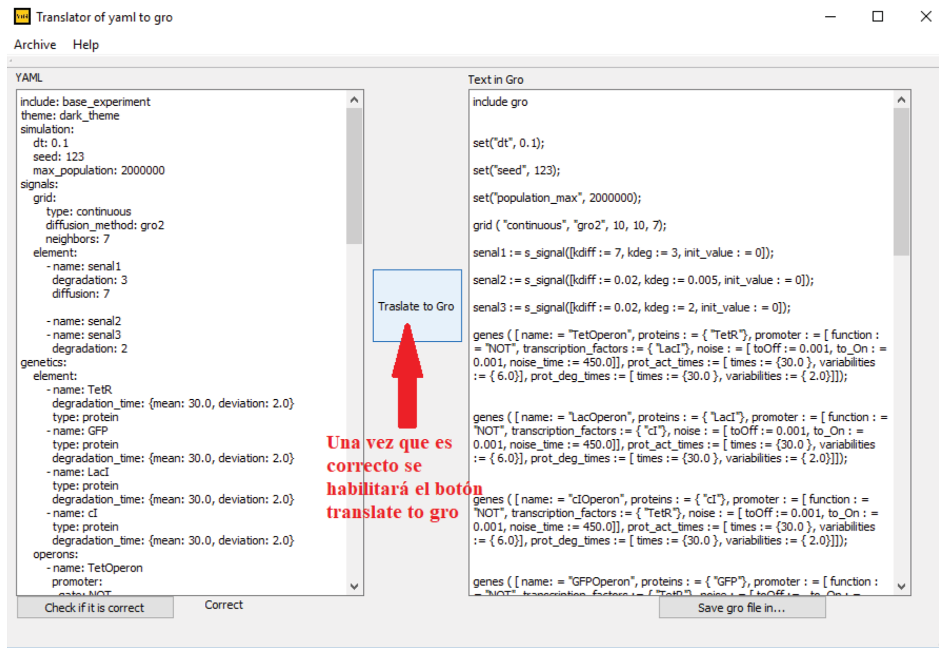


Figura 17. Captura de la interfaz una vez traducido a *GRO*

5ª Una vez traducido podemos finalmente guardar el archivo *GRO* en la carpeta deseada

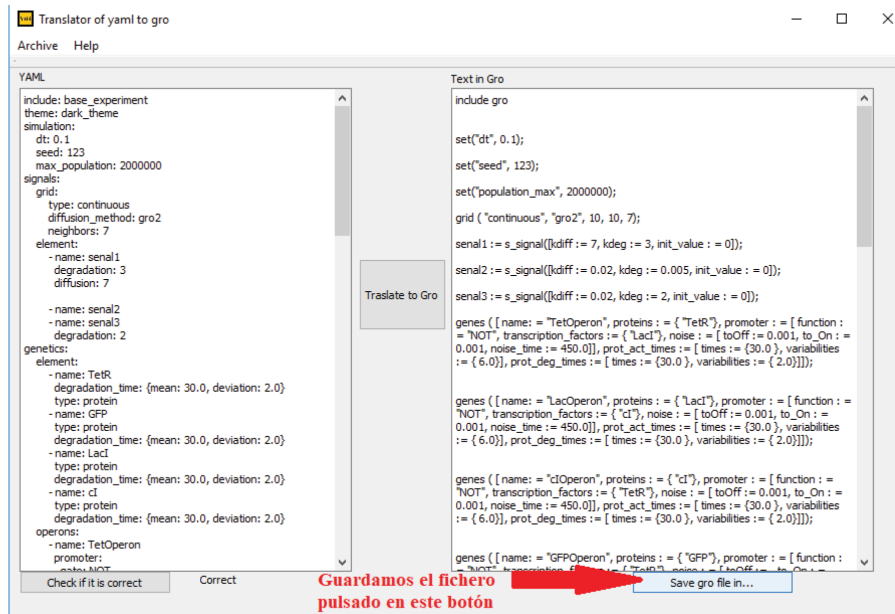


Figura 18. Captura de la interfaz una vez traducido a *GRO* señalando el botón guardar archivo gro

4.5 PRUEBA SIN ERRORES

Para poder comprobar el correcto funcionamiento del simulador se han realizado una serie de pruebas intentando abarcar todos los casos posibles, siempre y cuando, el fichero *YAML* haya pasado la verificación

```
include: base_experiment #se pueden poner comentarios
theme: dark_theme

simulation:
  dt: 0.3
  seed: 100
  max_population: 2000000

signals:
  grid:
    type: continuous
    diffusion_method: gro2
    neighbors: 7
  element:
    - name: senal1
      degradation: 4
      diffusion: 7
    - name: senal2
    - name: senal3
      degradation: 6

genetics:
  element:
    - name: TetR
      degradation_time: {mean: 15.0, deviation: 3.0}
      type: protein
    - name: GFP
      type: protein
      degradation_time: {mean: 20.0, deviation: 5.0}
    - name: LacI
      type: protein
      degradation_time: {mean: 10.0, deviation: 2.0}
    - name: cI
      type: protein
      degradation_time: {mean: 30.0, deviation: 4.0}
```

```
operons:
  - name: TetOperon
    promoter:
      gate: NOT
      transcription_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
      transcription_factors: [cI]
      block: {to_on: 0.1, to_off: 0.01, time: 450.0}
    genes:
      - expresses: LacI
        time: {mean: 30.0, deviation: 6.0}
  - name: cIOperon
    promoter:
      gate: NOT
      transcription_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
      transcription_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]
```

```
 - name: ecoli_2
   width: 1.0
   default_growth_rate: 0.2
   division_length: {x: 3.5, y: 4.0}
   division_proportion: {x: 0.4, y: 0.6}
   plasmids: [p1, p3]

cell_actions:
  - paint:
    condition: [GFP]
    concentration: 1
    color: [1,5,8,58]
  - d_paint:
    condition: [-GFP]
    concentration: 1
     color: [5, 5, 7, 9]
   - die:
     condition: [GFP]
  - conjugation:
    condition: [GFP, LacI]
    plasmid: p1
    rate: 2.3
  - lose_plasmid:
    condition: [TetOperon, LacI,GFP]
    plasmid: p3
  - emit_cross_feeding:
    condition: [GFP]
    signal: s2
    concentration: 23

world_actions:
  - strain: ecoli_2
    population: 300
    time: [100.4, 90.6]
    circle:
      center: {x: 0.5, y: 0.7}
      radius: 15.8
  - strain: ecoli_1
     population: 100
     time: [100.4, 90.6]
     circle:
       center: {x: 0.4, y: 0.7}
       radius: 13.8
```

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

Se puede observar que este ejemplo en *YAML* es muy amplio y se pueden observar también, cómo se pueden introducir los datos en todas sus partes para su posterior traducción.

TEXTO TRADUCIDO A GRO

```
include gro

set("dt", 0.1);

set("seed", 123);

set("population_max", 2000000);

grid ( "continuous", "gro2", 10, 10, 7);

senal1 := s_signal([kdiff := 7, kdeg := 3, init_value : = 0]);

senal2 := s_signal([kdiff := 0.02, kdeg := 0.005, init_value : = 0]);

senal3 := s_signal([kdiff := 0.02, kdeg := 2, init_value : = 0]);

genes ( [ name: = "TetOperon", proteins : = { "TetR"}, promoter : = [ function : =
"NOT", transcription_factors := { "LacI"}, noise : = [ toOff := 0.001, to_On : = 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, to_On : = 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, to_On : = 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"}, noise : = [ toOff := , to_On : = , noise_time := ]], 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, "paint" , {"1","5","8","58"});

action({-GFP}, "d_paint" , {"5","5","7","9"});

action({GFP}, "die" , {"0"});

action({"GFP","LacI"}, "conjugation" , {"p1","2.3"});

action({"TetOperon","LacI","GFP"}, "lose_plasmid" , {"p3"});

action({GFP}, "emit_cf" , {tostring(s2),"20"});

route1 := "~/Repressilator/";

route2 := "~/Repressilator/";

filename1 := "RepressilatorSingle.csv";

filename2 := "RepressilatorMultiple.csv";

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

fp2 := fopen ( route2 <> filename2, "w" );
```

```
program p1() :=
{
set ("ecoli_2_growth_rate", 0.2);
selected:
{
dump_single(fp1);
}
};



program p2() :=
{
set ("ecoli_1_growth_rate", 0.1);
selected:
{
dump_single(fp2);
}
};

program main() :=
{ t = 0:
{
fprint(fp1, "Time, No protein, TetR, GFP, LacI, cI, Total\n");

fprint(fp2, "Time, No protein, TetR, GFP, LacI, cI, Total\n");
}
true:
{
t := t + dt;
dump_multiple(fp1,{"-TetR","-GFP","-LacI","-
cI",},{"TetR"},{"GFP"},{"LacI"},{"cI"},{});

dump_multiple(fp2,{"-TetR","-GFP","-LacI","-
cI",},{"TetR"},{"GFP"},{"LacI"},{"cI"},{});
}

c_ecolis(300, 0.5, 0.7, 15.8, {"p1,p3"}, program p1())

c_ecolis(100, 0.4, 0.6, 13.8, {"p1,p2"}, program p2())

select_random_cell();
};
```

4.6 PRUEBA CON ERRORES

TEXTO EN YAML

```
include: base_experiment

theme: dark_theme

simulation:

  dt: 0.1

  seed: asdg                        #ERROR

  max_population: 2000000.3             #ERROR

signals:

  grid:

    type: discontinuous                #ERROR

    diffusion_method: gro2

    neighbors: 7.32                   #ERROR

  element:

    - name: senal1

      degradation: dsfgd              #ERROR

      diffusion: 7



    - name: senal2

    - name: senal3

      degradation: 2
```

```
genetics:
  element:
    - name: TetR
      degradation_time: {mean: 30.0, deviation: 2.0}
      type: protein
    - name: Hka
      type: proteina                    #ERROR
      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: sdf, deviation: 2.0} #ERROR
  operons:
    - name: TetOperon
      promoter:
        gate: NOT
        transcription_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
    transcription_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
    transcription_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
```

```
        transcription_factors: [TetR]
      genes:
        - expresses: GFP                    #ERROR
        time: {mean: 30.0, deviation: 6.0}
  plasmids:
    - name: p1
      operons: [TetOperon, LacOperon, cIOperon]
    - name: p2
      operons: [GFPOperon, OpNoDeclarado]  #ERROR no está definido
strains:
  - name: ecoli_1
    width: sd                              #ERROR
    default_growth_rate: 0.1
    division_length: {x: 3.5, y: 4.0}
    division_proportion: {x: 0.4, y: 0.6}
    plasmids: [p1, p2]
  - name: ecoli_2
    width: 1.0
    default_growth_rate: 0.2
    division_length: {x: 3.5, y: 4.0}
    division_proportion: {x: 0.4, y: 0.6}
```

```
      plasmids: [p1, p3]
cell_actions:
  - paint:
      condition: [GFP]
      concentration: 1
      color: [1,5,8,58]
  - d_paint:
      condition: [-GFP]
      concentration: 1
      color: [5, 5 , 7, 9]
  - die:
      condition: [GFP]
  - conjugation:
      condition: [GFP, LacI]
      plasmid: p1
      rate: 2.3
  - lose_plasmid:
      condition: [TetOperon, LacI,GFP]
      plasmid: p3
  - emit_cross_feeding:
      condition: [GFP]
```

```
      signal: s2
      concentration: 20
world_actions:
  - strain: ecoli_7                              #ERROR no declarada
    population: 300
    time: [100.4, 90.6]
    circle :
      center: {x: 0.5, y: 0.7}
      radius: 15.8
  - 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: []
    file: ~/Repressilator/RepressilatorSingle.csv
  - condition: []
    file: ~/Repressilator/RepressilatorMultiple.csv
```

ERRORES DETECTADOS POR LA APLICACIÓN

```
ERROR: the seed's value has to be a number(int)

ERROR: the max_population's value has to be a number(int)

ERROR: Invalid element type value. Valid types are only "continuous" and "discrete"
and the value of neighbors has to be a number(int)

ERROR: in signals, in elements, diffusion and degradation have to be a number (float)

ERROR: Invalid element type value. Valid types are only protein and arn.

ERROR: Invalid element type value. Valid types are only protein and arn.

ERROR: Invalid element type value. Valid types are only protein and arn.

ERROR: in genetics, in element, in degradation_time, mean and deviation have to be a
numers(float)
ERROR: Invalid element type value. Valid types are only protein and arn.

ERROR: in genetics, in element, in degradation_time, mean and deviation have to be a
numers(float)
ERROR: in genetics, in element, in degradation_time, mean and deviation have to be a
numers(float)
ERROR: in genetics, in element, in degradation_time, mean and deviation have to be a
numers(float)
ERROR: There is a genes->expresses that it doesn't exist in genetics->element

ERROR: There are operons in plasmid that they aren't defined.

ERROR: the width of strain has to be a number (float)

ERROR: There are strains in world actions that they weren't declared
```

A lo largo de los apartados tres y cuatro de esta memoria hemos podido observar de manera teórica, práctica y gráfica la diferencia entre introducir los datos para realizar experimentos en *GRO* y en *YAML*. Por tanto, este traductor amplía el personal que puede realizar experimentos y con esto el número de ellos consiguiendo así aumentar las probabilidades de conseguir algún descubrimiento y seguir mejorando en un campo tan importante para el presente y futuro como es la biología sintética.

5 REFERENCIAS

[1] S. Jang, K. Oishi, R. Egbert y E. Klavins. Specification and Simulation of Synthetic Multicelled Behaviors. *ACS Synth. Biol.*, **2012**, *1*, pp 365–374 http://pubs.acs.org/doi/abs/10.1021/sb300034m

[2] M. E. Gutiérrez, P. Gregorio-Godoy, G.P. del Pulgar, L. E. Muñoz, S. Sáez y A. Rodríguez-Patón. A new improved and extended versión of the multicell bacterial simulator gro. *ACS Synth. Biol.*, **2017**, *6*, pp 1496–1508

http://pubs.acs.org/doi/abs/10.1021/acssynbio.7b00003

[3] TJ Rudge, PJ Steiner, A Phillips, J Haseloff. Computational Modeling of Synthetic Microbial Biofilms. *ACS Synth. Biol.*, **2012**, *1(6)* pp 345–352 http://haselofflab.github.io/CellModeller/pdfs/acs2012.pdf

[4] Timothy J. Rudge, Fernan Federici, Paul J. Steiner, Anton Kan, and Jim Haseloff. Cell Polarity-Driven Instability Generates Self-Organized, Fractal Patterning of Cell Layers. *ACS Synth. Biol.*, **2013**, *2,* pp 705–714

http://haselofflab.github.io/CellModeller/pdfs/acs2013.pdf

[5] Antonio Prestes García and Alfonso Rodríguez-Patón. BactoSim – An Individual-Based Simulation Environment for Bacterial Conjugation. *PAAMS* **2015**, LNAI 9086, pp. 275–279.

https://link.springer.com/chapter/10.1007/978-3-319-18944-4_26

[6] Martyn Amos and Angel Goñi Moreno. Computational Synthetic Biology. *BioSystems* **2011,** 105:3, pp 286-294. https://www.elec.york.ac.uk/events/ncfrontiers/documents/ncfrontiers_amos.pdf

[7] Laurent A. Lardon, Brian V. Merkey, Sónia Martins, Andreas Dötsch, Cristian Picioreanu, Jan-Ulrich Kreft, Barth F. Smets. iDynoMiCS: next-generation individual-based modelling of biofilms. *Environmental Microbiology*, **2011**, 13, pp 2416-2434

http://onlinelibrary.wiley.com/doi/10.1111/j.1462-2920.2011.02414.x/abstract;jsessionid=843A86AE7D59BA465D65DA8BD78BC567.f04t04

6 Anexos

A: Definición del lenguaje de definición de experimentos

Como el lenguaje de definición de experimentos está 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 cuál es su significado. Dicho anexo que aquí adjunto, fue publicado por el exalumno anteriormente citado Marco en su trabajo fin de grado.

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	Número 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.

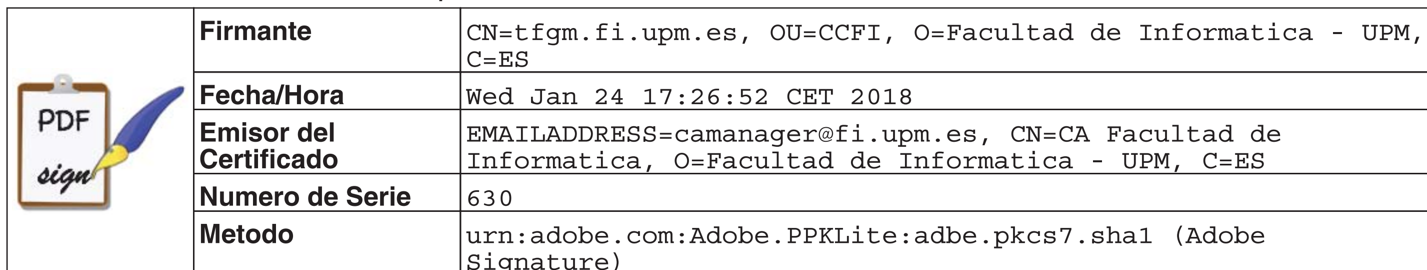
genetics.plasmids	Lista de plásmidos.
genetics.plasmids[i].name	Nombre del plásmido.
genetics.plasmids[i].operons	Lista de operones que forman parte de ese plásmido.
strains	Lista de cepas.
strains[i].name	Nombre de la cepa.
strains[i].width	Anchura de las bacterias.
strains[i].default_growth_rate	Ritmo de crecimiento de la cepa por unidad de tiempo(dt).
strains[i].division_lengthx|y	Rango de tiempos en los que una célula se puede dividir.
strains[i].division_proportionx|y	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 dé la acción.
cell_actions[i]. ’action_name’.lower_threshold	Límite mínimo de la señal para que se dé la acción.

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

output.pictures[i].refresh	Cada cuanto tiempo se hará una nueva imagen.
output.pictures[i].time	Intervalo de tiempo en el cual se capturarán imágenes.
output.pictures[i].zoom	Zoom de las imágenes. Puede ser 'variable' para ajustar automáticamente.
output.pictures[i].center_position	Posición central de la imagen.
output.pictures[i].center_cell_id	Id de la célula central de la imagen. Únicamente puede estar este atributo o "center_position". Si se encuentran ambos se da-rá un aviso.

Tabla que describe los datos que puede leer el módulo del archivo de un experimento en *YAML*

Este documento esta firmado por



	Firmante	CN=tfgm.fi.upm.es, OU=CCFI, O=Facultad de Informatica - UPM, C=ES
	Fecha/Hora	Wed Jan 24 17:26:52 CET 2018
	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)