POLITÉCNICA
"Ingeniamos el futuro"

CAMPUS
DE EXCELENCIA
INTERNACIONAL

Graduado en Matemáticas e Informática

Universidad Politécnica de Madrid

Escuela Técnica Superior de
Ingenieros Informáticos

TRABAJO FIN DE GRADO

Traductor de especificación de experimentos para el simulador multicelular Gro

Autor: Fernando Moreno Bendala

Director: Alfonso Rodríguez-Patón

MADRID, JUNIO 2018

ÍNDICE

ÍNDICE **1**

TABLA DE FIGURAS **2**

RESUMEN **3**

ABSTRACT **4**

INTRODUCCIÓN **5**
- Objetivos / Planteamiento del problema 5
- Tareas 6
- Estructura del documento 7

ESTADO DE LA CUESTIÓN **8**
- Gro 8
- Bioblocks 13
- Blockly y Grockly 14

SOLUCIÓN PROPUESTA **16**

DESARROLLO **18**
- Flujo de ejecución 19
- Bloques y traducciones 20
- A. Bloques principales 20
- Acciones 27
- Otros bloques 33

PRUEBAS DE VALIDACIÓN **38**
- 1. Prueba básica 38
- 2. Prueba con bloque de nutrientes 39
- 3. Prueba con acción 40
- 4. Prueba con señal y dump single 41
- 5. Prueba con todos los bloques 43
- 6. Prueba con múltiples instancias de todos los bloques posibles 45

CONCLUSIONES **48**

LÍNEAS FUTURAS DE TRABAJO **49**

REFERENCIAS **50**

TABLA DE FIGURAS

Figura 1. Diagrama de Gantt del proyecto 6
Figura 2. Tabla de planificación de proyecto 7
Figura 3. Una bacteria E. coli vista al microscopio 10
Figura 4. Logo de BioBlocks 14
Figura 5. Logo de Blockly 14
Figura 6. Prototipo de aplicación inicial 17
Figura 7. Programa de ejemplo en Blockly + traductor 18
Figura 8. Bloque de experimento 21
Figura 9. Bloque de bacteria 22
Figura 10. Bloque de plásmido 22
Figura 11. Bloque de operón 23
Figura 12. Bloque de promotor con y sin proteína, respectivamente 25
Figura 13. Bloque de proteína 26
Figura 14. Bloque de proteína auxiliar (ProteinL) 27
Figura 15. Bloque de acción principal 27
Figura 16. Bloques para pintar bacterias 28
Figura 17. Bloques para modificar color de bacteria 29
Figura 18. Bloques de conjugación y conjugación directa 30
Figura 19. Bloques de perder plásmido, establecer eex y eliminar eex 31
Figura 20. Bloques de emitir señal y emitir señal CF 32
Figura 21. Bloques de absorber y leer QS 33
Figura 22. Bloque de módulo de nutrientes 34
Figura 23. Bloque de módulo de señales 35
Figura 24. Bloque de creación de señal 36
Figura 25. Bloque de dump single 37
Figura 26. Resultado de traducir el código de prueba nº1 38
Figura 27. Resultado de traducir el código de prueba nº2 39
Figura 28. Resultado de traducir el código de prueba nº3 40
Figura 29. Resultado de traducir el código de prueba nº4 42
Figura 30. Resultado de traducir el código de prueba nº5 44
Figura 31. Resultado de traducir el código de prueba nº6 47

RESUMEN

En este proyecto se ha construído un software de traducción de Gro, un lenguaje de especificación de experimentos biológicos, a Grockly, un lenguaje de programación gráfico construído sobre la librería Blockly de Google, con el uso de un parser construído con tecnologías web (HTML y Javascript).

ABSTRACT

This project consists on a translation software, that interprets Gro, a specification language for biological experimentation, and converts it into Grockly, a graphic programming language built from Google's Blockly library, using a parsing mechanism programmed in web technologies (HTML and Javascript).

INTRODUCCIÓN

El campo de la Biología Sintética es un campo interdisciplinario relativamente nuevo (surgido en el año 2000 en el laboratorio de Inteligencia Artificial del MIT en Estados Unidos)[1]. Nace de la aplicación de los principios de la ingeniería a la biología para poder construir sistemas biológicos. En España, el Laboratorio de Inteligencia Artificial de la Universidad Politécnica de Madrid (LIA UPM) se dedica a este campo, entre otros.

Objetivos / Planteamiento del problema

Este Trabajo de Fin de Grado nace de la necesidad de los investigadores que se dedican a la biología sintética de poder realizar simulaciones y tener acceso a experimentos ya programados sin tener conocimientos de programación. En este caso concreto, se busca dar acceso a los experimentos y simulaciones ya existentes especificados en el lenguaje de especificación GRO (diseñado en el laboratorio de biología sintética del profesor Eric Klavins, en Universidad de Washington[2]).

Otra problemática que plantea este lenguaje en concreto es la dificultad que plantea su instalación, sobre todo para un usuario básico. GRO y su entorno gráfico tienen un gran número de paquetes de los que dependen, y a su vez estos paquetes tienen más dependencias. Además, algunos necesitan de versiones específicas y otros tienen incompatibilidades.

Con este proyecto se pretende eliminar todas estas barreras para que los usuarios que necesiten usar este lenguaje puedan hacerlo de forma sencilla e intuitiva.

Tareas

La planificación del proyecto sigue la siguiente estructura:

1. Aprender la especificación de experimentos en gro y el funcionamiento general del simulador. Aprender el lenguaje gráfico basado en bloques denominado Grockly.
2. Diseñar la arquitectura del traductor de .gro a Grockly.
3. Implementar el traductor. Desarrollo guiado por pruebas.
4. Validación del funcionamiento del traductor.
5. Redacción de la memoria y preparación de la presentación oral.

En la siguiente figura podemos apreciar el diagrama de Gantt que se construyó para realizar a cabo la estimación de la duración del proyecto y la cantidad de esfuerzo que se previó para cada una de las tareas.

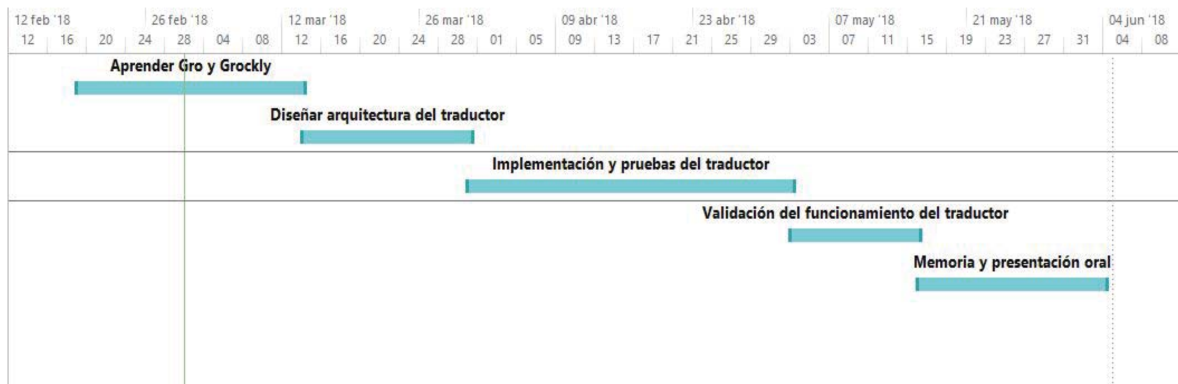


Figura 1. Diagrama de Gantt del proyecto

A continuación, figura una tabla de detalle complementaria al diagrama, con datos de duración de cada tarea, y fecha estimada de comienzo y fin.

Nombre de tarea	Duración	Comienzo	Fin
Aprender Gro y Grockly	70 horas	lun 19/02/18	mié 14/03/18
Diseñar arquitectura del traductor	54 horas	mié 14/03/18	sáb 31/03/18
Implementación y pruebas del traductor	100 horas	sáb 31/03/18	jue 03/05/18
Validación del funcionamiento del traductor	40 horas	jue 03/05/18	mié 16/05/18
Memoria y presentación oral	60 horas	mié 16/05/18	lun 04/06/18

Figura 2. Tabla de planificación de proyecto

Estructura del documento

La memoria de este proyecto está pensada con varios objetivos en mente. El primero, como es lógico, es para que pueda ser evaluado, pero también tiene el fin de servir de documentación para el uso, no solo del traductor de especificación, sino también el de Gro, y sobretodo, el de Grockly.

Teniendo esto en cuenta, la estructura del documento es la siguiente:

- Primero introduciré, en la sección de Estado de la cuestión, todo lo necesario para poder comprender el funcionamiento del software y el lenguaje con el que trabaja. Esto incluye una introducción a Gro, Blockly, y Grockly.
- Después, se hará un planteamiento del problema y las posibles soluciones, en el apartado de Solución propuesta.
- Más adelante, se describe el funcionamiento del software y las posibilidades de uso del mismo, en el apartado Desarrollo.
- Por último, se expondrán las conclusiones de proyecto y se discutirá sobre las posibles líneas de trabajo que se pueden tomar después del final de este proyecto.

ESTADO DE LA CUESTIÓN

Para comprender mejor este trabajo, debemos considerar otros proyectos en los que se basa, realizados en el mismo campo.

1. Una versión del lenguaje GRO desarrollado conjuntamente por Martín Gutiérrez, de la Escuela de Informática y Telecomunicaciones, de la Universidad Diego Portales, en Santiago de Chile, por Paula Gregorio-Godoy, Guillermo Pérez, Luis Muñoz, Sandra Sáez, y Alfonso Rodríguez-Patón, del LIA-UPM (Laboratorio de Inteligencia Artificial de la ETSIINF, en la Universidad Politécnica de Madrid)[3].
2. También podemos considerar BioBlocks, basado en la librería Blockly de Google, y también en la librería Scratch del MIT, para especificación de experimentos en una interfaz gráfica, realizado por Vishal Gupta, Jesús Irimia, Iván Pau
   y Alfonso Rodríguez-Patón, del LIA-UPM[4].
3. Por último, el Trabajo de Fin de Grado de Aitor Moya, “Desarrollo de una plataforma web para la especificación de protocolos biológicos basado en BIOBLOCKS”, también para el LIA, que consiste en una representación gráfica de Gro, llamada Grockly, igualmente desarrollado con la librería Blockly, que constituye la base de este trabajo[5].

Desarrollaremos los aspectos relevantes de dichos proyectos a continuación.

Gro

Como ya se ha expuesto en la introducción, Gro es un lenguaje de especificación de experimentos biológicos. Nos centraremos en la versión desarrollada por el LIA, aunque la mayoría de las cosas no cambian respecto a la versión inicial.

Los detalles del lenguaje se verán en el apartado de funcionamiento del programa (El traductor de Gro), ya que así podremos ver sus equivalencias en Grockly.

La estructura de un programa en gro es la siguiente[6]:

1. Inclusión de librerías, configuración de parámetros y definición de variables globales. Respecto a las librerías, nuestro programa solo trabajará con la librería estándar.
2. Entidades genéticas y relaciones existentes entre ellas.
3. Acciones asociadas a bacterias (una de las entidades genéticas).
4. Programas de alto nivel y control global.

En la primera parte del programa, se incluye siempre la librería estándar de gro (usando la instrucción **`include gro`**, y después la configuración de los parámetros se lleva a cabo mediante instrucciones **`set`**.
Estas configuraciones pueden ser:

1. Duración del paso de tiempo: Un time step (o paso de tiempo) es la duración de un ciclo de ejecución de la simulación.
2. Población máxima: El número máximo de bacterias que puede haber en el entorno de simulación en un momento dado.
3. Configuraciones de nutrientes: Estas sentencias de configuración tienen todas que ver con el módulo de nutrientes, que se encarga de manejar todo lo relacionado con la alimentación de las bacterias. Con ellas se puede:
    a. Encender o apagar el módulo de nutrientes.
    b. Establecer la cantidad de nutrientes disponible.
    c. Configurar la tasa de consumo de nutrientes.
    d. Modificar el número de células en el espacio de nutrientes.
    e. Modificar el tamaño del espacio de nutrientes.
    f. Establecer el modo de consumo de nutrientes.
4. Creación y manejo de señales. Estas sentencias de configuración tienen que ver con el módulo de señales, que se utilizan para distintos propósitos en la simulación. Con ellas se puede:
    a. Encender o apagar el módulo de señales.
    b. Establecer si las señales se mostrarán gráficamente o no.

c. Crear señales (aunque la sintaxis es distinta).

En la siguiente parte del programa, se incluyen las entidades genéticas, es decir, los agentes que intervienen en el experimento. Estos son: bacterias, plásmidos, operones, promotores y proteínas.

1. Bacteria: Las bacterias son microorganismos procariotas que presentan un tamaño de unos pocos micrómetros. Son la primera entidad que se define en el experimento. En gro solo se trabaja con una bacteria, la *Escherichia coli* (E. Coli).

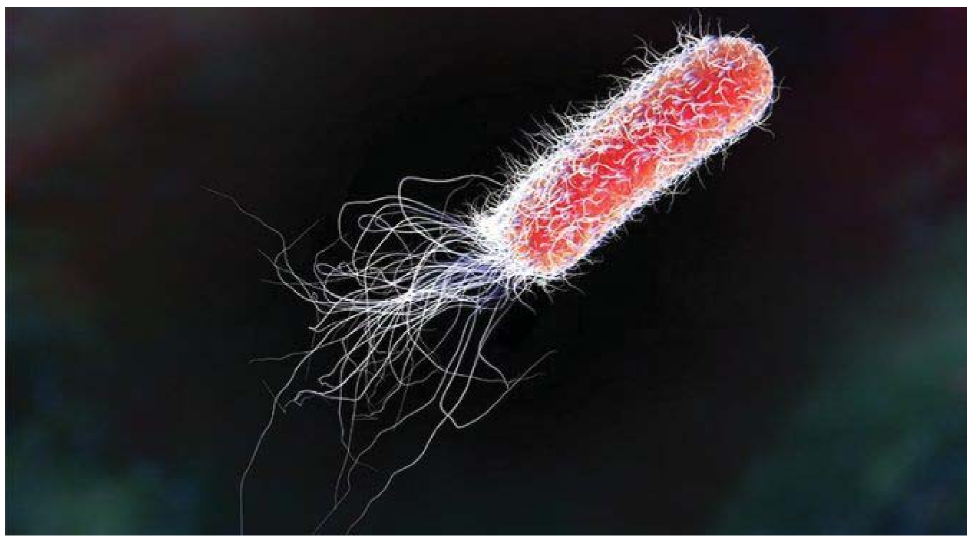


Figura 3. Una bacteria E. coli vista al microscopio

Se define con:

`ecoli([x, y, theta, volumen], lista_plasmidos, programa)`

para una sola celula, o con:

`e_colis(n_celulas, x, y, radio, lista_plasmidos, programa)`

para una colección de células en un círculo.

2. Plásmido: Los plásmidos son moléculas de ADN extracromosómico generalmente circular que se replican y transmiten independientes del ADN cromosómicos. Están presentes en bacterias, y así se refleja en la jerarquía, ya que es la siguiente entidad en la lista. Se define con:

`plasmids_genes([nombre_plasmido_1:={lista_operones}, …])`

3. Operón: Es un conjunto de genes que regulan la expresión proteica asociada a dichos genes, gracias a las proteínas con las que interaccionan los promotores, por medio de su factor de transcripción. Se define como:
   genes(nombre, lista_proteinas, promotor, parametros_proteinas)

   Dentro de la estructura de los operones se encuentran las siguientes entidades genéticas, que se explicarán en los siguientes apartados.

4. Promotor: Los promotores son regiones de ADN que se encargan de iniciar la transcripción de un fragmento de ADN a ARN, es decir, un gen.
   Dentro de la declaración de los operones, la estructura del promotor es:
   promoter:=[function:="funcion_logica",transcription_factors:= {"nombre_proteina"},noise:=[toOff:=n,toOn:=n,noise_time:=n]]

   Los promotores actúan sobre la proteína especificada aplicando la función que se les proporciona, que tiene forma de operador lógico.

5. Proteínas: Las proteínas son macromoléculas formadas por cadenas lineales de aminoácidos. Se sintetizan dependiendo de cómo se encuentren regulados los genes que las codifican (promotores).
   Está definido dentro de los operones en los que se usa:
   proteins:={lista_proteinas}
   Los parámetros de las proteínas también están dentro de la estructura:

```
prot_act_times:=
      [times:={lista_tiempos_activacion},
      letiabilities:={lista_variaciones_activacion}],
      prot_deg_times:=[times:={lista_tiempos_desactivacion},
      letiabilities:={lista_variaciones_desactivacion}]
```

La siguiente parte del programa son las acciones que se aplican sobre todas las bacterias en las que se cumple la condición especificada por la lista de proteínas (por ejemplo, existe x proteína o no existe x proteína) cada ciclo de simulación. Las acciones tienen la siguiente forma:

```
action({lista_proteinas}, nombre_accion, {parametros_accion})
```

A continuación, enumero la lista de acciones que se utilizan en Grockly, que son un subconjunto de todas las acciones posibles en Gro:

1. Pintar bacteria (paint): rellena la bacteria con la combinación de colores seleccionada.
2. Modificar color de bacteria (d_paint): modifica el color de la bacteria añadiendo o sustrayendo la cantidad de cada color indicada.
3. Conjugar (conjugate): copia un plásmido de la bacteria a otra bacteria aleatoria que se encuentre cerca de la bacteria que realiza la conjugación.
4. Conjugación direccionada (conjugate_directed): similar a la anterior, pero considera un mecanismo de restricción presente en las bacterias adyacentes (llamado eex), que dictamina si dicha bacteria es elegible para la conjugación. Este mecanismo se manipula con las siguientes acciones.
5. Establecer eex (set_eex): Aplica una restricción para la conjugación a la bacteria.
6. Eliminar eex (remove_eex): Elimina la restricción para la conjugación.
7. Perder plásmido (lose_plasmid): El plásmido elegido es eliminado de la lista de plásmidos de la bacteria que realiza la acción.
8. Emitir señal (s_emit_signal): La bacteria emite una determinada concentración de señal elegida al entorno.
9. Emitir señal CF (emit_cf): La bacteria emite una determinada concentración de señal de cross-feeding (CF). Esta señal, cuando es leída por por otra bacteria, afecta a su ratio de crecimiento.
10. Absorber Quorum Sensing (s_absorb_QS): La bacteria absorbe una cantidad determinada de una señal, y si cumple una condición, activa la proteína elegida. Si no, la desactiva. El Quorum Sensing (QS) se refiere a un mecanismo de

regulación de la expresión genética en respuesta a la densidad de población celular.

11. Leer QS (s_get_QS): Igual que la acción anterior, pero sin absorber la señal, solo la lee.
12. Dump Single: Esta acción no está relacionada con ningún agente. Su propósito es guardar los datos de la simulación que se vaya a realizar en el fichero especificado.

La última parte son los programas de alto nivel y el control global del programa.
Los programas suelen asignarse a las bacterias para dictaminar su comportamiento, y si no queremos incluir ningún comportamiento especial, usaremos la función `skip()`.
Por otra parte, debemos mencionar el programa principal o `main`, desde el que se inicia la ejecución y se lleva a cabo el control global del experimento.
Nuestro software no tendrá en cuenta programas de alto nivel, excepto para el programa principal (main), que es indispensable para la ejecución del experimento, y el programa dump_single, explicado más adelante, pero no para el resto de programas, ya que Grockly no tiene soporte para ellos.

Bioblocks

Bioblocks es un proyecto que surgió con una motivación similar a este, la de permitir a gente sin conocimientos de programación poder realizar programar experimentos biológicos con facilidad. Está basado en Blockly, que será introducido a continuación, y en el lenguaje de especificación de experimentos Autoprotocol, y presenta una interfaz gráfica basada en bloques que se traducen tanto a código fuente de Autoprotocol (escrito en Python o en JSON), como a texto descriptivo del experimento en inglés, como a Protocol Workflow, un diagrama de flujo específico del lenguaje.

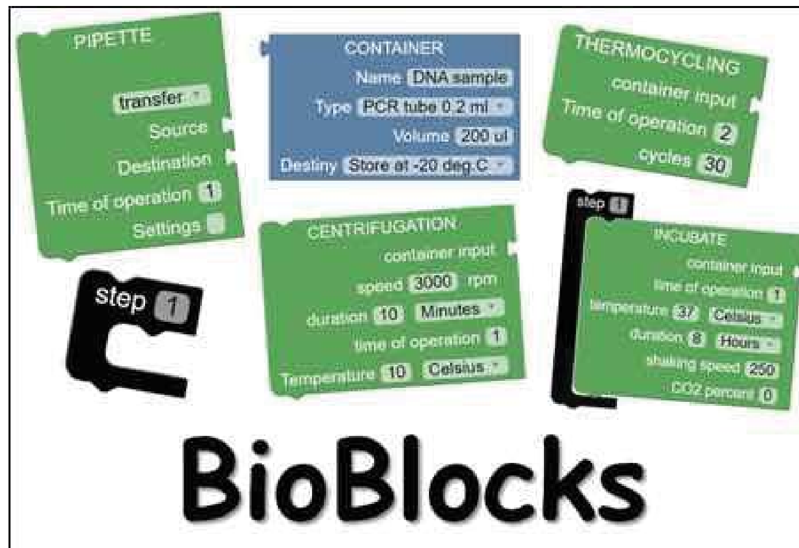


Figura 4. Logo de BioBlocks

Blockly y Grockly

Blocky[7] es una librería de Google que proporciona una interfaz gráfica que consiste en bloques completamente personalizables que representan conceptos de programación, cuyas relaciones pueden ser determinadas por el usuario, y que se juntan para crear código funcional y sintácticamente correcto.

Además de la interfaz gráfica para crear bloques, y la posibilidad de definir tus propios bloques, Blockly nos ofrece una interfaz completa para crear y manejar bloques, sus conexiones, y el propio workspace, en definitiva, todo lo necesario para poder generar bloques a partir de código y código a partir de bloques.

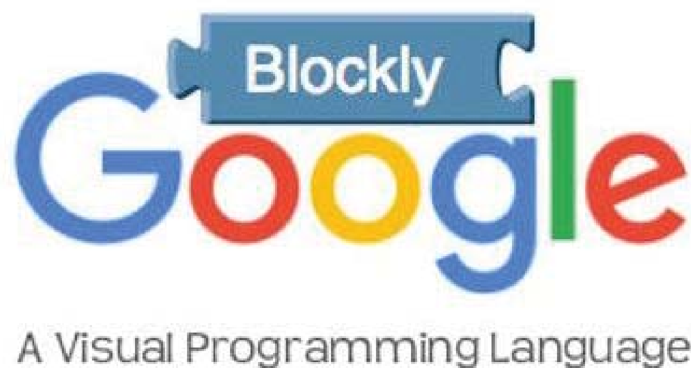


Figura 5. Logo de Blockly

Grockly es un proyecto similar a Bioblocks, también basado en Blockly, para Gro. En Grockly, los conceptos de Gro introducidos anteriormente se traducen en bloques que contienen todo lo necesario para implementar el concepto que representan, los cuales se unen con otros bloques para especificar el experimento que se quiere describir.
Grockly se realizó teniendo en mente el mismo objetivo final que este proyecto, que es acercar la programación de experimentos a gente sin conocimientos de programación, y lo consigue gracias a la facilidad que ofrece la programación por bloques, que limita la posibilidad de escribir código erróneo y pone todas las herramientas necesarias para escribir un programa de forma gráfica.
Además, Grockly también es capaz de generar código escrito en Gro a partir de la estructura de bloques resultante de construir código gráficamente, y este código, que será sintácticamente correcto y contendrá toda la información contenida en los bloques y sus relaciones, podrá ser ejecutado o simulado.

SOLUCIÓN PROPUESTA

Para lograr el objetivo propuesto en el apartado de introducción, consideré varias posibilidades. Primero, era necesario escoger las tecnologías, las herramientas y los lenguajes que se usarían para construir la solución. Barajé el uso de lenguajes convencionales como Java, dado mi conocimiento de ellos, pero finalmente opté por usar tecnologías web, es decir, tanto JavaScript como HTML. Tomé esta decisión por varias razones.
La primera fue porque he trabajado bastante con estos lenguajes, por lo que me sería más sencillo desarrollar con ellos, y no tendría que dedicar mucho tiempo a formarme.
La segunda tiene que ver con que, junto a la tecnología Node.js[8], son los lenguajes con los que está construída la aplicación web Grow, que es la plataforma en la que se integraría en el futuro mi software.
La tercera y última es la facilidad que ofrecen las tecnologías web para desplegar software, lo cual encaja con el objetivo de ofrecer una herramienta que sea fácil de usar y no requiera de una instalación complicada, para que los usuarios básicos puedan utilizarla sin problemas (ya que se podría acceder a ella o bien desde una página web, o bien desplegando la aplicación en la máquina local, lo cual es bastante sencillo).

Una vez decidido esto, lo siguiente fue decidir la tecnología en concreto que usaría. Respecto a esto, primero consideré la tecnología Ionic, ya que la conozco y he desarrollado aplicaciones web con ella, pero no la escogí porque me parecía más adecuada para desarrollo de aplicaciones móviles y para páginas web con una carga de diseño mayor, y este proyecto no se enfoca en el diseño sino en la funcionalidad.
Después pensé en hacerlo en Node.js, y decidí en un principio que sería buena idea, ya que aporta muchas funcionalidades y es muy sencillo desarrollar una aplicación web y desplegarla.

De aquí surgió el primer prototipo de la aplicación, que consistía en un servidor con un sistema de subida de ficheros que comprobaba si el archivo subido tenía la extensión .gro, ya que nuestra aplicación solo trabaja con ese tipo de ficheros.

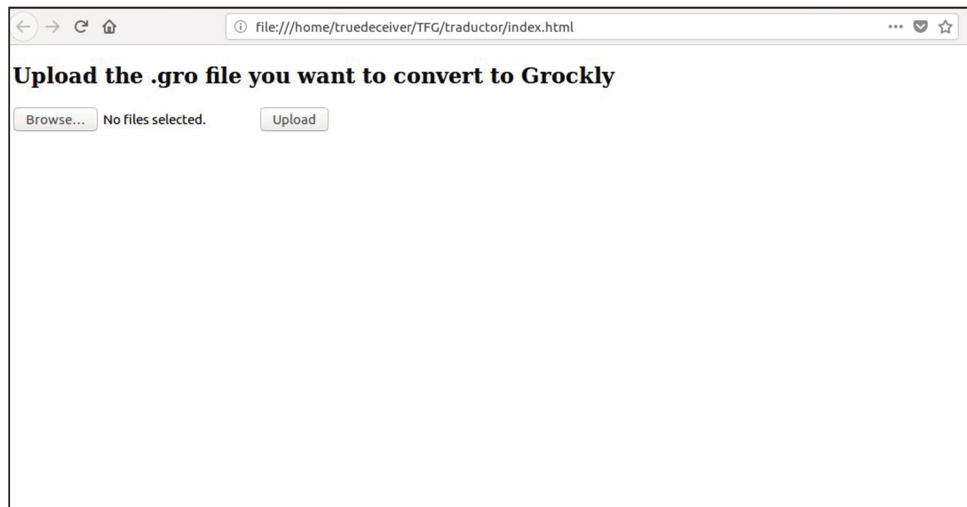


Figura 6. Prototipo de aplicación inicial

El problema de esta aproximación era que necesitaba tener la funcionalidad de Grockly para mostrar los bloques, y más importante todavía, para poder trabajar sobre ellos después de generarlos. Para esto, es necesario integrar la librería Blockly, y la aplicación Grockly. Después de estudiarlo e intentarlo en varias ocasiones, decidí que era mejor en términos de consumo de tiempo y de dificultad desarrollar mi aplicación directamente sobre Grockly, y así aprovechar las funcionalidades que tiene Grockly ya implementadas. Como ya he señalado anteriormente, Grockly consiste únicamente de una página web construida sobre un archivo HTML con scripts externos e internos, por lo tanto, descarté el prototipo, y empecé a trabajar con HTML y JS sin ninguna tecnología adicional.
Al final, la solución escogida fue esta, y el proyecto se desarrolló hasta el final con esta metodología.

DESARROLLO

El traductor de especificación de experimentos para el simulador multicelular gro consiste en un analizador o parser de código, junto con un constructor de bloques, escrito en Javascript, integrado directamente en la aplicación de Grockly. Este analizador cuenta con un cargador de archivos, el cual únicamente acepta archivos en formato .gro.

El parser está programado de forma modular, y cada módulo lleva a cabo una tarea definida. La ejecución se lleva a cabo de forma secuencial, y cada módulo llama al siguiente hasta terminar el proceso de traducción de cada fragmento de código.

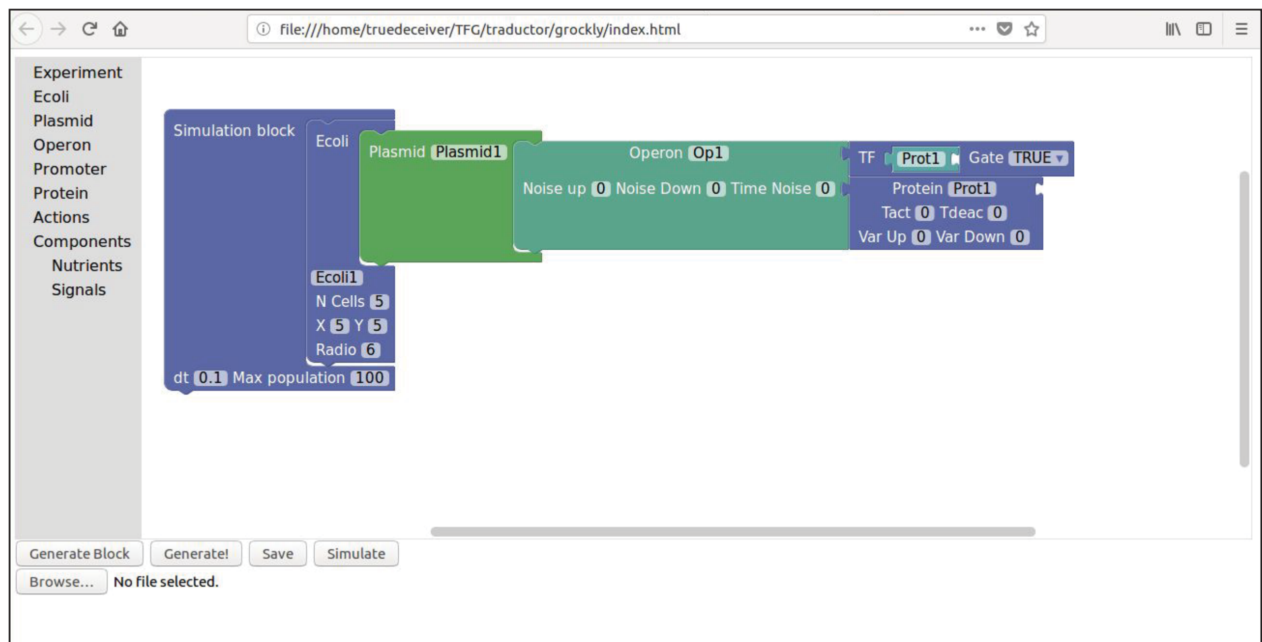


Figura 7. Programa de ejemplo en Blockly + traductor

Flujo de ejecución

A continuación describiré el flujo normal del programa:
Primero, el usuario sube un archivo, si es del formato incorrecto, el sistema pedirá un archivo del formato correcto y descartará el archivo. Si tiene el formato correcto, continuará al siguiente paso.
Después de subir un archivo correcto, el programa alertará al usuario de la pérdida de su espacio de trabajo actual, ya que la traducción debe llevarse a cabo en un workspace vacío.
Si el usuario acepta, se llevará a cabo en proceso de parseo del código. Si no, el programa descartará el archivo y no realizará ninguna acción más, pudiendo el usuario continuar con su trabajo anterior.

El primer módulo, al que llamaremos primer parser, separa el código en segmentos más pequeños, cada uno conteniendo uno o varios fragmentos significativos de código.

El segundo módulo, llamado analizador, examina y categoriza cada segmento según la palabra o palabras clave que contengan, y en función de la palabra encontrada, llamará al módulo que corresponda a dicha palabra, y éste se encargará de convertir el fragmento de código a una estructura de datos que contiene toda la información contenida en dicho fragmento organizada de forma que sea fácilmente tratable por el siguiente módulo. Por ejemplo, si un fragmento contiene la palabra “action”, el módulo de análisis de fragmentos llamará a parseAction, que es el módulo que se encarga de transformar la sentencia action y sus parámetros en una estructura de datos que estará preparada para ser convertida en un bloque de Grockly.

El tercer módulo, el parser principal, se encarga de convertir el fragmento de código que recibe en datos que puedan ser fácilmente manipulables y transformables.
Esto se consigue aplicando sobre el bloque del código funciones de tratamiento de cadenas de caracteres (strings) para separar los datos del resto del código, y luego almacenándolos en estructuras de datos o bien globales (en el caso de que se necesiten

más tarde para construir otros bloques), o bien locales, que se le pasan al siguiente módulo.

El último módulo, el módulo constructor, como su nombre indica, construye los bloques y las conexiones entre ellos usando los datos refinados que le llegan del parser principal. La forma de hacerlo es usando las llamadas que nos ofrece Blockly para la creación y manejo de bloques, y para la creación y manejo de sus conexiones.

Bloques y traducciones

En esta sección seguiremos una estructura concreta para que sea más fácil ver los datos de cada bloque, sus propiedades y su código. La estructura es la siguiente:
-Descripción del bloque.
-Conexiones del bloque, divididas en externas (A la izquierda, arriba o abajo del bloque), o internas (a la derecha o dentro del bloque).
-Parámetros del bloque. También incluye el tipo de parámetro, entre los siguientes: Float (número en coma flotante), Entero, String (cadena de caracteres), String[] (array de strings), Desplegable (opciones a elegir en un desplegable en el propio bloque).
-Figura.
-Código correspondiente.
Esta sección está diseñada para documentar el uso del traductor (facilitando los fragmentos de código que corresponden a cada bloque) y el uso de Grockly.

A. Bloques principales

Estos bloques se consideran principales porque es necesario que haya un bloque de cada tipo para que se pueda producir código y comenzar la simulación.

1. **Bloque de experimento:**
   Este bloque es el primer bloque que se coloca, y a partir de él se conectan los demás bloques.

Conexiones:

-Externas: Bloque de nutrientes, bloque de acción principal, bloque de señales.

-Internas: Bloque de bacterias.

Parámetros:

-Paso de tiempo (Float): Duración de cada ciclo de simulación.

-Población máxima (Float): Nº máximo de bacterias en el entorno.

Figura 8. Bloque de experimento

Código:

```
set("dt",0); set("population_max",0);
```

2. Bacteria-E. Coli:

El bloque de bacteria añade un grupo de bacterias contenidas en un círculo al entorno.

Conexiones:

-Externas: Otros bloques de bacterias.

-Internas: Bloque de plásmido.

Parámetros:

-Nombre (String): Para distinguirla del resto de agrupaciones.

-N_Cells (Entero): Nº de bacterias que componen el grupo.

-X (Float): Coordenada X del centro del grupo de bacterias.

-Y (Float): Coordenada Y del centro del grupo de bacterias.

-Radio (Float): Longitud del radio en píxeles del círculo grupo de bacterias.

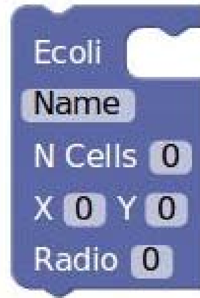


Figura 9. Bloque de bacteria

Código:

```
e_colis(n_celulas, x, y, radio, lista_plasmidos, programa)
```

Parámetros de código:

-lista_plasmidos (String[]): Lista de plásmidos asociados al grupo de bacterias.

-programa (Bloque de código): Bloque de código que indica el comportamiento de las bacterias. Solo puede ser skip() o dump_single(...), que se explica más adelante.

3. **Plásmido:**

Añade un plásmido asociado a la bacteria a la que se conecta.

Conexiones:

-Externas: Otros bloques de plásmido.

-Internas: Bloque de operón.

Parámetros:

-Name (String): Nombre que identifica al plásmido.

Figura 10. Bloque de plásmido

Código:

```
plasmids_genes([nombre_plasmido_1:={lista_operones_1},
    nombre_plasmido_2:={lista_operones_2], ...)
```

Parámetros de código:

-nombre_plasmido_n: Equivalente a Name.

-lista_operones_n (String[]): Lista de operones asociados al plásmido nombre_plasmido_n.

4. Operón:

Añade un operón asociado al plásmido al que se conecta.

Conexiones:

-Externa: Otros bloques de operón.

-Interna: Un bloque de promotor y otro de proteína.

Parámetros:

-Name (String): El nombre que identifica al operón.

-Noise_up (Float): Probabilidad de que la proteína asociada al operón se active a lo largo de un periodo de tiempo igual a Time_Noise. Valores de 0.0 a 1.0.

-Noise_Down (Float): Probabilidad de que la proteína asociada al operón se desactive a lo largo de un periodo de tiempo igual a Time_Noise. Valores de 0.0 a 1.0.

-Time_Noise (Float): Periodo de tiempo considerado para la actuación del ruido (en segundos).

Figura 11. Bloque de operón

Código:

```
genes(nombre, lista_proteinas, promotor, parametros_proteinas)
```

Hay que tener en cuenta que dentro de la estructura de los operones se encuentran las proteínas (en **`lista_proteinas`** y **`parametros_proteinas`**) y el promotor (en **`promotor`**) asociados a dicho operón.
Aquí tenemos un ejemplo del código completo del operón:

```
genes([name:="Operon1",
      proteins:={"Protein1","Protein2"},
      promoter:=[
            function:="TRUE",
            transcription_factors:={"Promoter1"},
            noise:=[toOff:=0,toOn:=0,noise_time:=0]
      ],
      prot_act_times:=[times:={0,0},letiabilities:={0,0}],
      prot_deg_times:=[times:={0,0},letiabilities:={0,0}]]);
```

Además los parámetros de ruido se encuentran dentro de promotor, y tienen la siguiente forma:

```
noise:=[toOff:=0,toOn:=0,noise_time:=0]
```

Donde **`toOff`** es Noise_Down, **`toOn`** es Noise_Up y **`noise_time`** es Time_Noise.

5. Promotor:

Añade un promotor asociado al operón al que se conecta.

Conexiones:

-Externas: Bloque de operón al que está asociado.

-Internas: Bloques de proteína auxiliar (uno o varios).

Parámetros:

-Gate (Desplegable): Función lógica que regula la síntesis de proteína en el promotor. Puede ser: YES, NOT, TRUE, FALSE, OR, XOR, AND, NAND.

Nota: el/los bloque/s de proteína auxiliar interno es obligatorio a menos que Gate sea TRUE o FALSE.

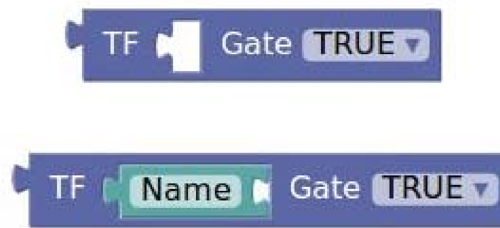


Figura 12. Bloque de promotor con y sin proteína, respectivamente

Código:

```
promoter:=[function:="funcion_logica",transcription_factors:=
{"nombre_proteina"},noise:=[toOff:=n,toOn:=n,noise_time:=n]]
```

Parámetros de código:

-function (String): Equivale a Gate.

-transcription_factors (String[]): Lista de proteínas que funcionan como entrada del promotor para la síntesis de proteínas. Equivalen a los bloques de proteínas auxiliares internos del promotor.

6. Proteína estándar:

Añade una proteína que será sintetizada por el promotor.

Conexiones:

-Externas: El bloque de operón al que está asociado.

-Internas: Otros bloques de proteínas estándar.

Parámetros:

-Name (String): El nombre por el que se identifica la proteína.

-Tact (Float): Tiempo medio de activación de la proteína (en minutos).

-Tact (Float): Tiempo medio de desactivación de la proteína (en minutos).

-Var_Up (Float): Desviación estándar del tiempo de activación de la proteína (en minutos).

-Var_Down (Float): Desviación estándar del tiempo de desactivación de la proteína (en minutos).

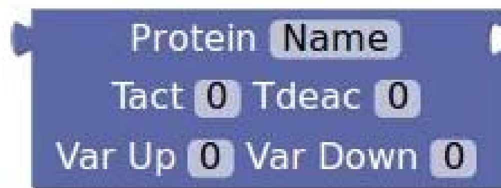


Figura 13. Bloque de proteína

Código: Está definido dentro de los operones en los que se usa:

```
proteins:={lista_proteinas}
```

Los parámetros de las proteínas también están dentro del operón:

```
prot_act_times:=
      [times:={lista_tiempos_activacion},
      letiabilities:={lista_variaciones_activacion}],
prot_deg_times:=
      [times:={lista_tiempos_desactivacion},
      letiabilities:={lista_variaciones_desactivacion}]
```

Donde **times** equivale a Tact o a Tdeac, y **letiabilities** equivale a Var_Up o Var_Down (en prot_act_times y prot_deg_times, respectivamente).

7. Proteína auxiliar o lista de proteínas:

Añade una proteína que sirve de parámetro de entrada de un promotor o de una acción. Se pueden encadenar varias entre sí.

Conexiones:

-Externas: Bloque de promotor, bloque de acción principal o bloque de proteína auxiliar.

-Internas: Bloque de proteína auxiliar.

Parámetros:

-Name (String): Nombre de la proteína.

Figura 14. Bloque de proteína auxiliar (ProteinL)

El código de una proteína auxiliar o una lista es simplemente un string o un array de strings con el nombre o los nombres de las proteínas.

Acciones

Los bloques de acciones representan acciones que se llevan a cabo en las bacterias en las que se cumple una condición específica.

1. **Bloque de acción principal:**

   Este bloque es un contenedor para construir acciones.

   -Conexiones:

   -Externas: Bloque de experimento, bloque de nutrientes, bloque de crear señales.

   -Internas (Por orden de izquierda a derecha):

   a. Bloques de proteínas auxiliares y bloque de acción de conjugar.

   b. Bloques de acciones.

Figura 15. Bloque de acción principal

Código:

```
action({lista_proteinas}, nombre_accion, {parametros_accion})
```

Parámetros del código:

-lista_proteinas (String[]): lista de proteínas que tienen o no que estar presentes en la bacteria para que se dispare la acción. Si una proteína no se encuentra en la lista quiere decir que no importa que esté o no. Ejemplo: {"-prot1", "prot2"} indica que para dispararse la acción es necesario que la proteína prot1 no esté presente y que la proteína prot2 esté presente.

-nombre_accion (String): El código de la acción.

-parametros_accion (String): La lista de parámetros que recibe la acción.

Los dos últimos parámetros son específicos de cada acción, y se verán uno por uno a continuación.

2. **Pintar bacteria (paint):** Rellena la bacteria con la combinación de colores indicada o con el color seleccionado.

Código:

```
action({...}, "paint", {"green","red","yellow","cyan"});
```

Parámetros:

-Bloque 1: Concentración de color de verde, rojo, amarillo y cian (Entero), con valores entre -3200 y 3200.

-Bloque 2. Color elegido entre rojo, azul, verde, amarillo, naranja, cian y morado.

Figura 16. Bloques para pintar bacterias

Nota: El bloque 2 de paint tiene el mismo código que el 1, ya que los colores a elegir están codificados como verde, rojo, amarillo y cian

3. **Modificar color (d_paint):** Modifica el color de la bacteria añadiendo o sustrayendo la cantidad de cada color o el color indicado.

   Código:

```
action({...}, “d_paint", {"green","red","yellow","cyan"});
```

   Parámetros:

   -Bloque 1: Concentración de color de verde, rojo, amarillo y cian (Entero), con valores entre -3200 y 3200.

   -Bloque 2. Color elegido entre rojo, azul, verde, amarillo, naranja, cian y morado.

Figura 17. Bloques para modificar color de bacteria

4. **Conjugar (conjugate):** Copia un plásmido de la bacteria a otra bacteria aleatoria que se encuentre cerca de la bacteria que realiza la conjugación.

   Código:

```
action({...}, “conjugate", {"Plasmid", “Rate”});
```

5. **Conjugación direccionada (conjugate_directed):** similar a la anterior, pero considera un mecanismo de restricción presente en las bacterias adyacentes (llamado eex), que dictamina si dicha bacteria es elegible para la conjugación. Este mecanismo se manipula con las siguientes acciones.

   Código:

```
action({...}, “conjugate_directed", {"Plasmid", “Rate”});
```

   Las dos acciones usan el mismo bloque y los mismos parámetros.

Parámetros:

-Plasmid (String): El nombre del plásmido a conjugar.

-Rate (Float): La tasa media a la que se llevan a cabo las conjugaciones.

Figura 18. Bloques de conjugación y conjugación directa

6. **Perder plásmido (lose_plasmid):** El plásmido elegido es eliminado de la lista de plásmidos de la bacteria que realiza la acción.

   Código:

   ```
   action({...}, “lose_plasmid", {"Plasmid"});
   ```

7. **Establecer eex (set_eex):** Aplica una restricción para la conjugación del plásmido pasado como parámetro a la bacteria.

   Código:

   ```
   action({...}, “set_eex", {"Plasmid"});
   ```

8. **Eliminar eex (remove_eex):** Elimina la restricción para la conjugación.

   Código:

   ```
   action({...}, “remove_eex", {"Plasmid"});
   ```

Las tres acciones usan el mismo bloque y los mismos parámetros.

Parámetros:

-Plasmid (String): El nombre del plásmido afectado.

Figura 19. Bloques de perder plásmido, establecer eex y eliminar eex

9. **Emitir señal (emit_signal):** La bacteria emite una determinada concentración de una señal específica.

   Código:

```
action({"...},
"s_emit_signal",{“Signal_ID”,"Concentration","Emission_Type"})
;
```

10. **Emitir señal CF (emit_cf):** La bacteria emite una determinada concentración de señal de cross-feeding(CF). Esta señal, cuando es leída por por otra bacteria, afecta a su ratio de crecimiento.

    Código:

```
action({"...},
"s_emit_cf",{“Signal_ID”,"Concentration","Emission_Type"});
```

Las dos acciones usan el mismo bloque y los mismos parámetros.

Parámetros:

-Signal_ID (String): El identificador de la señal a emitir.

-Concentration (Float): La concentración de señal a emitir.

-Emission_Type (Desplegable): La forma de emitir la señal. Exact significa que la emisión solo se hará en el espacio en el que se encuentra la bacteria, Area que la señal se emitirá en todos los espacios del área de la bacteria, y Average emite de la misma forma que Area, pero la concentración es la media de la concentración dada y el número de espacios.

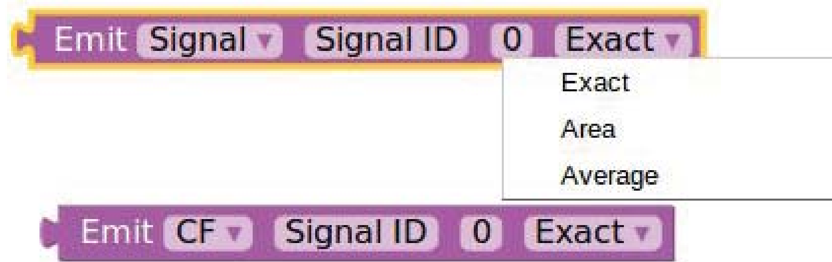


Figura 20. Bloques de emitir señal y emitir señal CF

11. **Absorber Quorum Sensing (s_absorb_QS):** La bacteria absorbe una cantidad determinada de una señal, y si la cantidad absorbida es mayor o menor (según el signo que se pase como parámetro) que el umbral, activa la proteína elegida. Si no, la desactiva.

Código:

```
action({"...},
"s_absorb_QS",{“Signal_ID”,"Comparison","Threshold",”Protein”}
)
```

12. **Leer QS (s_get_QS):** Igual que la acción anterior, pero sin absorber la señal, solo la lee.

Código:

```
action({"...},
"s_get_QS",{“Signal_ID”,"Comparison","Threshold",”Protein”})
```

Las dos acciones usan el mismo bloque y los mismos parámetros.

Parámetros:

-Signal_ID (String): El identificador de la señal a leer/absorber.

-Comparison (Desplegable): El signo para la comparación (> ó <).

-Threshold (Float): El umbral usado para la comparación.

-Protein (String): El nombre de la proteína a activar/desactivar.

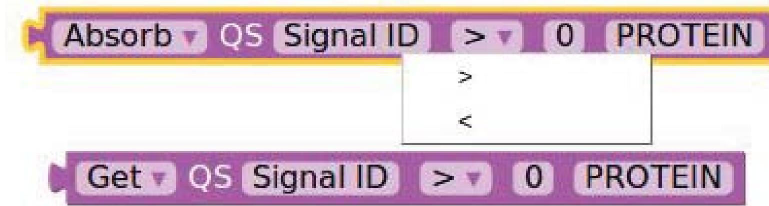


Figura 21. Bloques de absorber y leer QS

Otros bloques

Estos bloques son opcionales y añaden funcionalidades extra a la simulación.

1. **Bloque de módulo de nutrientes:**

   Añade un bloque de nutrientes a la simulación, con todos los parámetros de configuración del módulo de nutrientes. Sólo se puede tener uno como máximo. Nota: Incluir el bloque de nutrientes indica que se usa el módulo de nutrientes.

   Conexiones:

   -Externas: Bloque de experimento, bloque de acción principal, bloque de señales.

   -Internas: N/A

   Parámetros:

   -Nutrients_amount (Float): Cantidad de unidades de nutriente disponible en cada grupo de células.

   -Nutrient_Consumption_Rate (Float): Tasa de consumo de nutrientes. En unidades de nutriente por paso de tiempo (units/dt).

   -Nutrient_Grid_Length (Float): Número de células en el espacio de nutrientes.

   -Nutrient_Grid_Cell_Size (Float): Tamaño del lado del espacio de nutrientes. En píxeles.

   -Nutrient_Consumption_Mode (Float): Modo de consumo de nutrientes. Puede ser 0 (Homogéneo), 1 (Por proximidad) o 2 (Aleatorio).

Figura 22. Bloque de módulo de nutrientes

Código:

```
set("nutrients", 1.0);
set("nutrients_amount",0);
set("nutrient_consumption_rate",0);
set("nutrient_grid_length",0);
set("nutrient_grid_cell_size",0);
set("nutrient_consumption_mode",0);
```

Las instrucciones corresponden a los parámetros del bloque en orden, empezando por la segunda instrucción. La primera indica la presencia del bloque de nutrientes (o del módulo en Gro). No son necesarias todas las instrucciones (por defecto están todos los parámetros a 0).

2. Bloque de módulo de señales:

Añade un bloque de señales a la simulación, con todos los parámetros de configuración del módulo de señales. Sólo se puede tener uno como máximo. Nota: Incluir el bloque de nutrientes indica que se usa el módulo de nutrientes.

Conexiones:

-Externas: Bloque de experimento, bloque de nutrientes, bloque de acción principal.

-Internas: N/A.

Parámetros:

-Draw_Signals (Desplegable): Establece si las señales se mostrarán gráficamente (ON) o no (OFF).

-Length (Float): Número de células en el espacio de señales.

-Cell_Size (Float): Tamaño del lado del espacio de señales. En píxeles.

-Neighborhood (Float): Tamaño de las inmediaciones del espacio de señales.

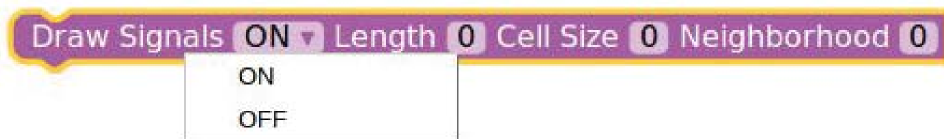


Figura 23. Bloque de módulo de señales

Código:

```
set("signals", 1.0);
set("signals_draw",1.0);
set_param("signals_grid_length",0);
set_param("signals_grid_cell_size",0);
set_param("signals_grid_neighborhood",0);
```

Las instrucciones corresponden a los parámetros del bloque en orden, empezando por la segunda instrucción. La primera indica la presencia del bloque de señales (o del módulo en Gro). No son necesarias todas las instrucciones (por defecto están todos los parámetros a 0, excepto `signals_draw`, que está a 1.0).

3. Bloque de creación de señal:

Añade una señal a la simulación, que se puede usar para interaccionar con las bacterias.

Conexiones:

-Superior: Bloque de experimentos, bloque de nutrientes, bloque de señales.

-Inferior: Otros bloques de señales.

Parámetros:

-Name (String): Nombre de la señal.

-kDiff (Float): Coeficiente de difusión de la señal.

-kDeg (Float): Coeficiente de degradación de la señal.

Figura 24. Bloque de creación de señal

Código:

```
Name:= s_signal([kdiff:=0,kdeg:=0]);
```

Los parámetros son los mismos que para el bloque.

4. **Bloque de Dump Single:**

Indica que al realizar la simulación, los datos resultantes de dicha simulación se guardarán en el fichero especificado. Técnicamente, es una acción, pero como no está relacionada con ningún agente y tiene una estructura distinta al resto de acciones, la incluímos aquí.

Conexiones:

Se coloca debajo del bloque de experimento (no puede ir en otro sitio) y debajo se puede conectar cualquier bloque que se pueda conectar con el bloque de experimento.

Parámetros:

-Route (String): Ruta absoluta de la localización del fichero en el que se van a guardar los datos (sin incluir el fichero).

-File_Name (String): Nombre del fichero a utilizar.

-El tercer parámetro es un entero que indica si hay que añadirle un número al final del nombre del fichero (útil si queremos guardar los datos en múltiples momentos de la simulación usando un bucle).

Figura 25. Bloque de dump single

Código:

```
program p() :={ selected:{ dump_single(fopen(
     Route <> File Name <> tostring((0+1)) <> ".csv", "w"
));}};
```

Los parámetros son los mismos que en el bloque. La extensión del fichero es por defecto csv, y en Blockly no se puede cambiar.

PRUEBAS DE VALIDACIÓN

1. Prueba básica

```
include gro
set("dt",0);
set("population_max",0);
genes([name:="Name",proteins:={"Name"},
      promoter:=[function:="TRUE",transcription_factors:={},
      noise:=[toOff:=0,toOn:=0,noise_time:=0]],
      prot_act_times:=[times:={0},letiabilities:={0}],
      prot_deg_times:=[times:={0},letiabilities:={0}]]);
plasmids_genes([Name:={"Name"}]);
program p():={skip();};program main():={c_ecolis(0,0,0,0,{"Name"},
      program p());};
```

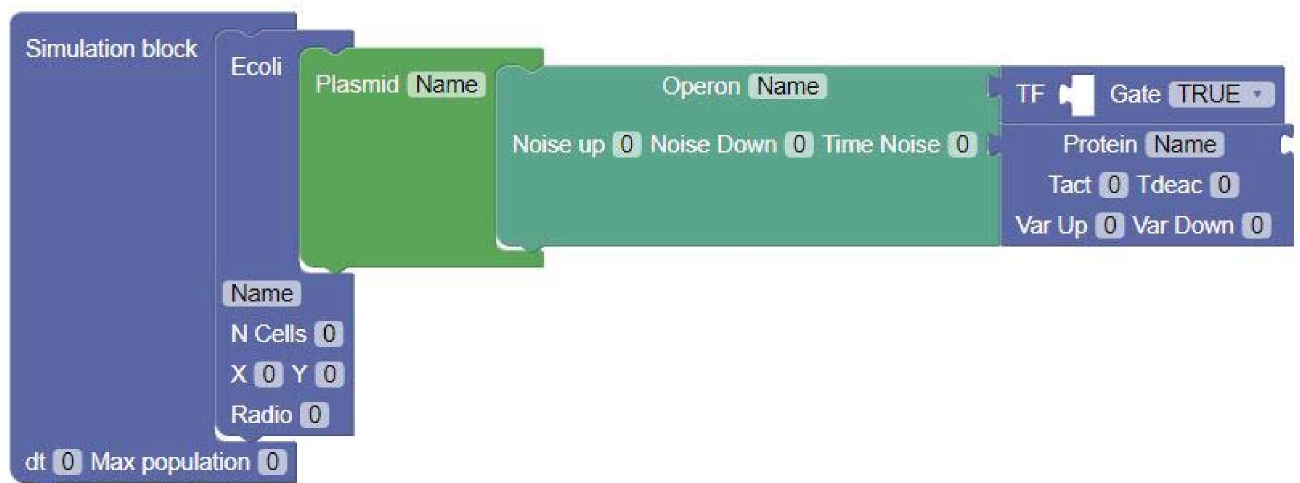


Figura 26. Resultado de traducir el código de prueba nº1

2. Prueba con bloque de nutrientes

```
include gro
set("dt",0);
set("population_max",0);
set("nutrients", 1.0);
set("nutrients_amount",100);
set("nutrient_consumption_rate",1);
set("nutrient_grid_length",10);
set("nutrient_grid_cell_size",10);
set("nutrient_consumption_mode",1);

genes([name:="Name",proteins:={"Name"},
      promoter:=[function:="TRUE",transcription_factors:={},
      noise:=[toOff:=0,toOn:=0,noise_time:=0]],
      prot_act_times:=[times:={0},letiabilities:={0}],
      prot_deg_times:=[times:={0},letiabilities:={0}]]);
plasmids_genes([Name:={"Name"}]);
program p():={skip();};program main():={c_ecolis(0,0,0,0,{"Name"},
      program p());};
```

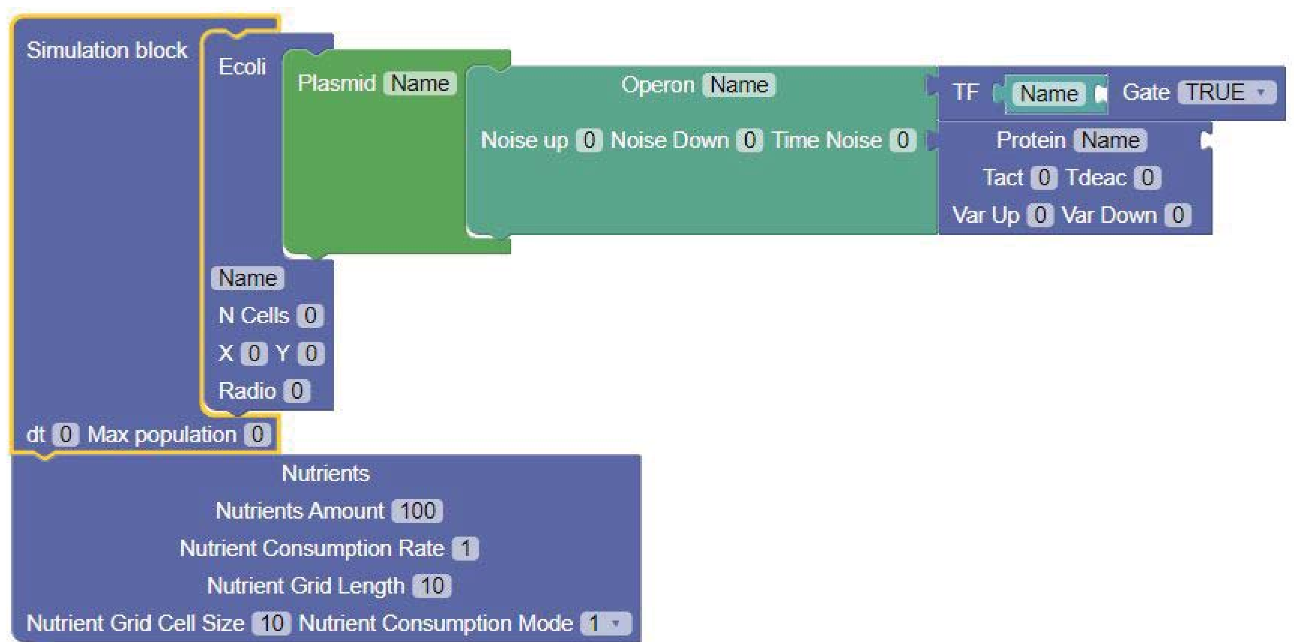


Figura 27. Resultado de traducir el código de prueba nº2

3. Prueba con acción

```
include gro
set("dt",0);
set("population_max",0);

genes([name:="Name",proteins:={"Name"},
      promoter:=[function:="TRUE",transcription_factors:={"Name"},
      noise:=[toOff:=0,toOn:=0,noise_time:=0]],
      prot_act_times:=[times:={0},letiabilities:={0}],
      prot_deg_times:=[times:={0},letiabilities:={0}]]);
plasmids_genes([Name:={"Name"}]);
action({"prot1"},"paint",{"0","3200","0","0"});

program p():={skip();};program main():={c_ecolis(0,0,0,0,{"Name"},
      program p());};
```

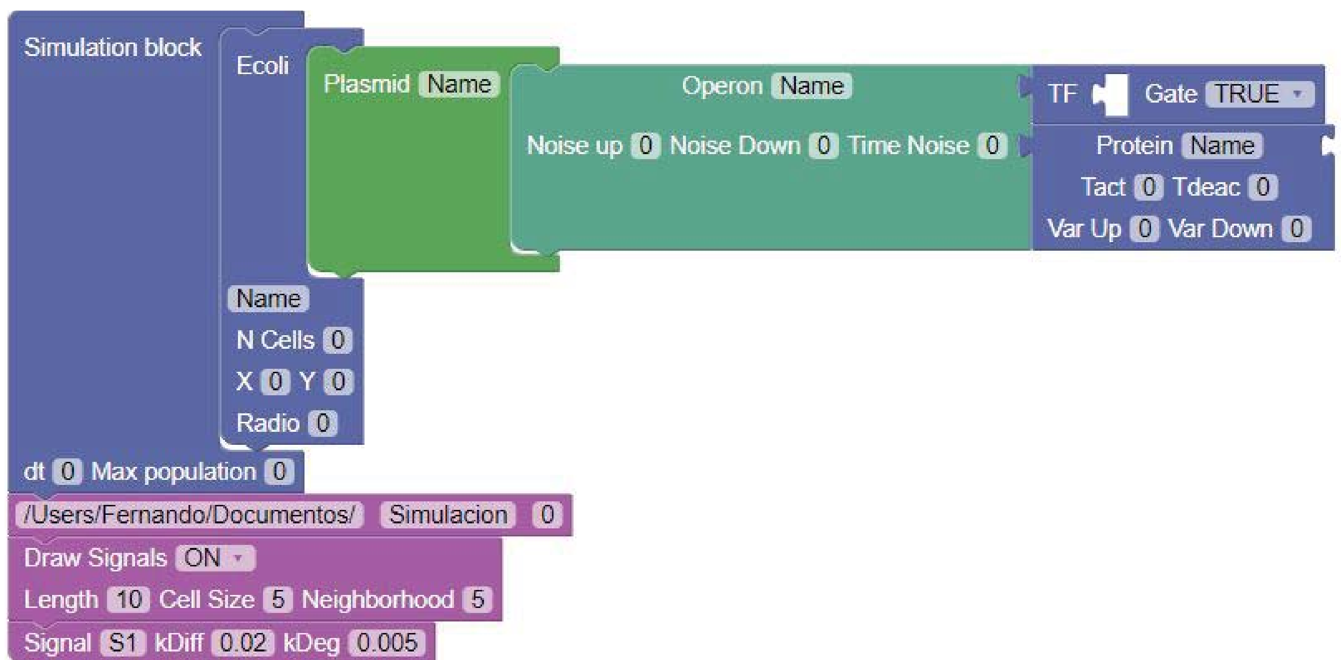


Figura 28. Resultado de traducir el código de prueba nº3

4. Prueba con señal y dump single

```
include gro
set("dt",0);
set("population_max",0);

set("signals", 1.0);
set("signals_draw",1.0);
set_param("signals_grid_length",10);
set_param("signals_grid_cell_size",5);
set_param("signals_grid_neighborhood",5);
S1:= s_signal([kdiff:=0.02,kdeg:=0.005]);

genes([name:="Name",proteins:={"Name"},
      promoter:=[function:="TRUE",transcription_factors:={},
      noise:=[toOff:=0,toOn:=0,noise_time:=0]],
      prot_act_times:=[times:={0},letiabilities:={0}],
      prot_deg_times:=[times:={0},letiabilities:={0}]]);
plasmids_genes([Name:={"Name"}]);

program p() :={ selected:{ dump_single( fopen(/Users/Fernando/Documentos/
<> Simulacion <> tostring((0+1)) <> ".csv", "w"));}};
program main():={c_ecolis(0,0,0,0,{"Name"},program p());};
```

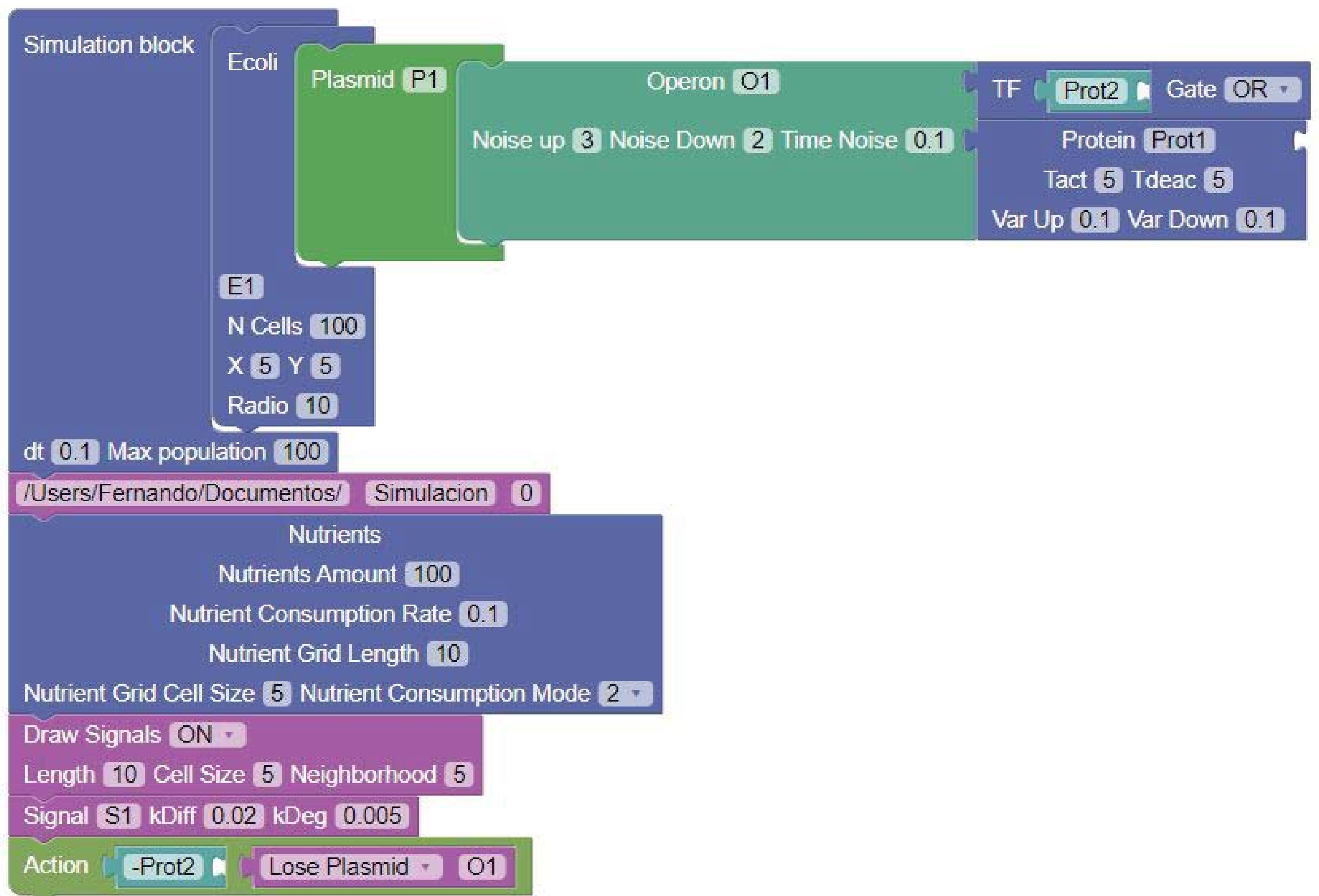


Figura 29. Resultado de traducir el código de prueba nº4

5. Prueba con todos los bloques

```
include gro
set("dt",0.1);
set("population_max",100);

set("nutrients", 1.0);
set("nutrients_amount",100);
set("nutrient_consumption_rate",0.1);
set("nutrient_grid_length",10);
set("nutrient_grid_cell_size",5);
set("nutrient_consumption_mode",2);

set("signals", 1.0);
set("signals_draw",1.0);
set_param("signals_grid_length",10);
set_param("signals_grid_cell_size",5);
set_param("signals_grid_neighborhood",5);
S1:= s_signal([kdiff:=0.02,kdeg:=0.005]);

genes([name:="O1",proteins:={"Prot1"},
      promoter:=[function:="OR",transcription_factors:={"Prot2"},
      noise:=[toOff:=3,toOn:=2,noise_time:=0.1]],
      prot_act_times:=[times:={5},letiabilities:={0.1}],
      prot_deg_times:=[times:={5},letiabilities:={0.1}]]);
plasmids_genes([P1:={"O1"}]);
action({"-Prot2"},"lose_plasmid",{"O1"});

program p() :={ selected:{ dump_single( fopen(/Users/Fernando/Documentos/
<> Simulacion <> tostring((0+1)) <> ".csv", "w"));}};
program main():={c_ecolis(100,5,5,10,{"P1"},program p());};
```

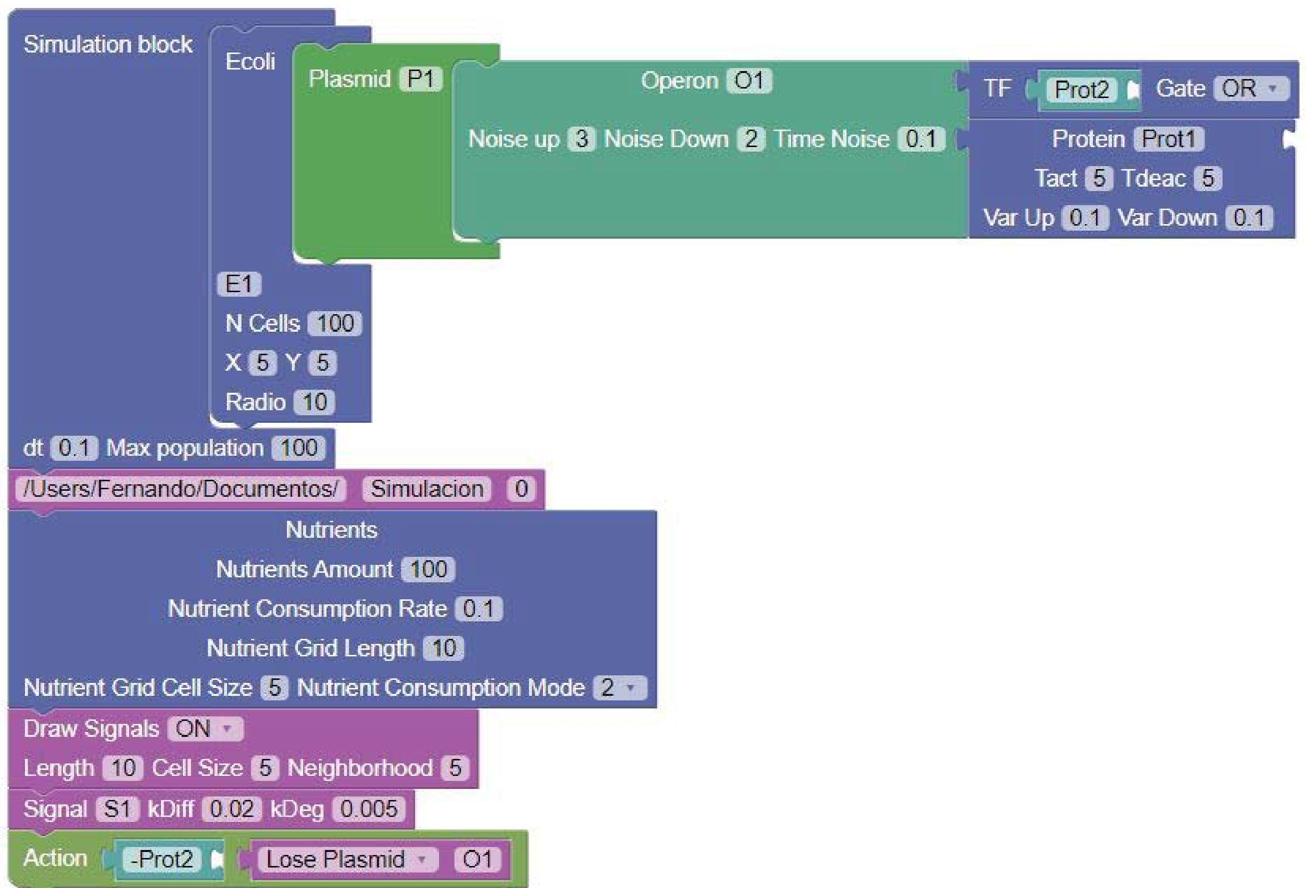


Figura 30. Resultado de traducir el código de prueba nº5

6. Prueba con múltiples instancias de todos los bloques posibles

```
include gro
set("dt",0.1);
set("population_max",100);

set("nutrients", 1.0);
set("nutrients_amount",100);
set("nutrient_consumption_rate",0.1);
set("nutrient_grid_length",10);
set("nutrient_grid_cell_size",5);
set("nutrient_consumption_mode",2);

set("signals", 1.0);
set("signals_draw",1.0);
set_param("signals_grid_length",10);
set_param("signals_grid_cell_size",5);
set_param("signals_grid_neighborhood",5);
S1:= s_signal([kdiff:=0.02,kdeg:=0.005]);

genes([name:="O1",proteins:={"Prot1","Prot2"},
      promoter:=[function:="NOT",transcription_factors:={"Prot2"},
      noise:=[toOff:=3,toOn:=2,noise_time:=0.1]],
      prot_act_times:=[times:={5,5},letiabilities:={0.1,0.1}],
      prot_deg_times:=[times:={5,5},letiabilities:={0.1,0.1}]]);
genes([name:="O1",proteins:={"Prot1","Prot2"},
      promoter:=[function:="NOT",transcription_factors:={"Prot2"},
      noise:=[toOff:=1,toOn:=0.5,noise_time:=0.1]],
      prot_act_times:=[times:={5,5},letiabilities:={0.1,0.1}],
      prot_deg_times:=[times:={5,5},letiabilities:={0.1,0.1}]]);
genes([name:="O3",proteins:={"Prot1","Prot3"},
      promoter:=[function:="TRUE",transcription_factors:={},
      noise:=[toOff:=3,toOn:=2,noise_time:=0.5]],
      prot_act_times:=[times:={5,4},letiabilities:={0.1,0.1}],
      prot_deg_times:=[times:={5,6},letiabilities:={0.1,0.1}]]);
```

```
genes([name:="O2",proteins:={"Prot1"},
      promoter:=[function:="OR",transcription_factors:={"Prot2","Prot3"},
      noise:=[toOff:=2,toOn:=3,noise_time:=1]],
      prot_act_times:=[times:={5},letiabilities:={0.1}],
      prot_deg_times:=[times:={5},letiabilities:={0.1}]]);
plasmids_genes([P1:={"O1"},P1:={"O1","O2"},P2:={"O3"}]);

action({,"-Prot2"},"lose_plasmid",{"O1"});
action({"Prot3"},"paint",{"3200","0","0","0"});

program p() :={ selected:{ dump_single( fopen(/Users/Fernando/Documentos/
<> Simulacion <> tostring((0+1)) <> ".csv", "w"));}};
program main():={c_ecolis(100,5,5,10,{"P1"},program p());
      c_ecolis(50,20,20,5,{"P1","P2"},program p());};
```

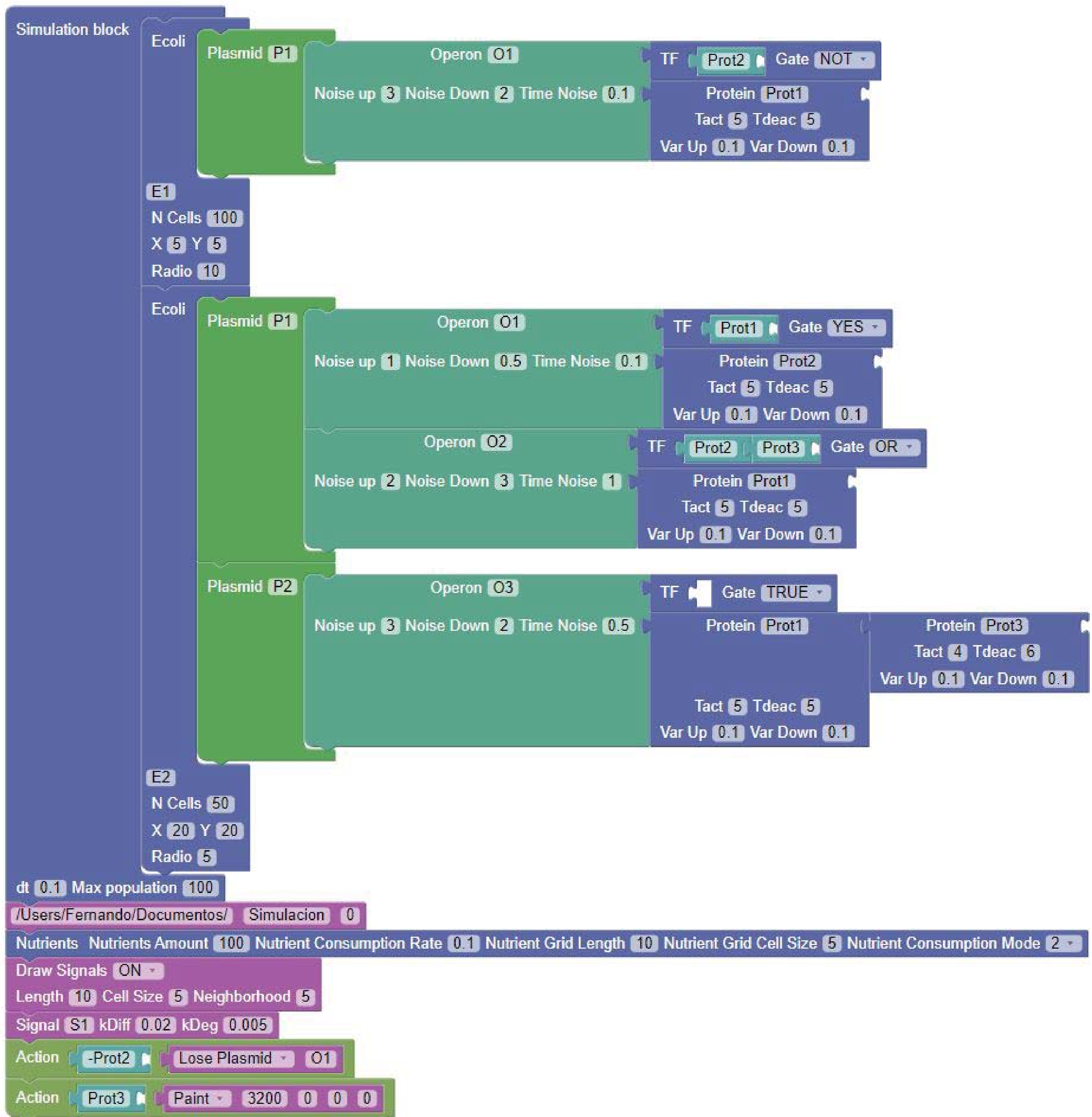


Figura 31. Resultado de traducir el código de prueba nº6

CONCLUSIONES

Finalmente, voy a exponer las conclusiones a las que he llegado tras desarrollar el proyecto realizado en este Trabajo de Fin de Grado.
Primero, vamos a recapitular cuáles eran los objetivos del proyecto. El objetivo principal facilitar el uso del lenguaje Gro a usuarios (principalmente, biólogos y estudiantes de biología y similares) con conocimientos limitados o nulos de programación.

Teniendo en cuenta cuales eran los objetivos, creo que se han alcanzado satisfactoriamente, ya que hemos conseguido un software que posibilita que un usuario pueda tomar provecho de experimentos y programas escritos en .gro ya existentes, y pueda simularlos, comprenderlos, o editarlos.

LÍNEAS FUTURAS DE TRABAJO

Por último, discutiré las posibles mejoras y ampliaciones que se pueden realizar sobre el proyecto.

Si tomamos como ejemplo de software al que aspirar el proyecto Bioblocks, es evidente que todavía hay margen de mejora, aunque más para Blockly que para el traductor, lo cual no se trata en este proyecto.

En primer lugar, se podría mejorar el parser para que reconociera mejor el código, sin necesidad de separarlo en bloques o líneas, aunque considero que esto es secundario, ya que la mayoría del código que se escribe está correctamente estructurado e indentado.

En segundo lugar, si Blockly se ampliara para reconocer más acciones o características de Gro, habría también que ampliar el traductor para ser capaz de trabajar con dichas características.

REFERENCIAS

[1] LIA-UPM, Página web del Laboratorio de Inteligencia Artificial de la UPM. [25/4/2018] (En línea) Disponible en: www.lia.upm.es

[2] The Klavins Lab, Página web del lenguaje GRO de la Universidad de Washington [24/4/2018] (En línea) Disponible en: https://depts.washington.edu/soslab/gro

[3] A New Improved and Extended Version of the Multicell Bacterial Simulator gro
Martín Gutiérrez, Paula Gregorio-Godoy, Guillermo Pérez del Pulgar, Luis E. Muñoz, Sandra Sáez, y Alfonso Rodríguez-Patón
ACS Synthetic Biology 2017 6 (8), 1496-1508
DOI: 10.1021/acssynbio.7b00003

[4] BioBlocks: Programming Protocols in Biology Made Easier
Vishal Gupta, Jesús Irimia, Iván Pau, y Alfonso Rodríguez-Patón
ACS Synthetic Biology 2017 6 (7), 1230-1232
DOI: 10.1021/acssynbio.6b00304

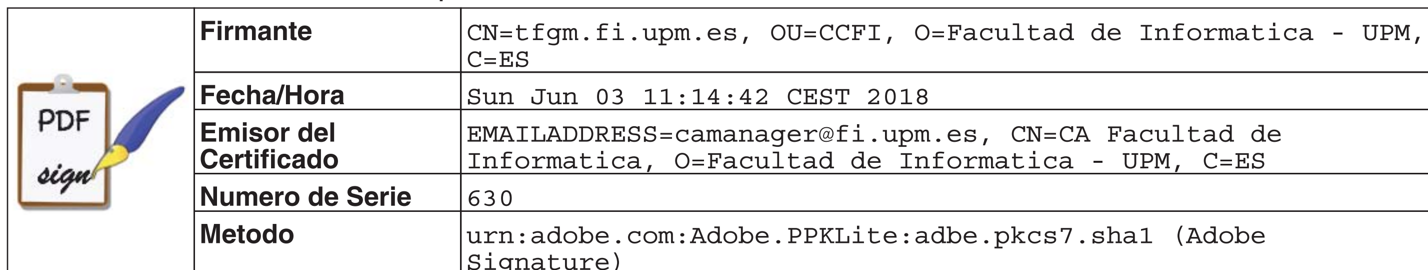
[5] Moya López, Aitor (2018). Desarrollo de una plataforma web para la especificación de protocolos biológicos basados en BIOBLOCKS. Proyecto Fin de Carrera / Trabajo Fin de Grado, E.T.S. de Ingenieros Informáticos (UPM), Madrid, España.

[6] Gutiérrez Pescarmona, Martín E., ProSpec specification reference. [23/05/2018] (En línea) Disponible en:
https://github.com/liaupm/GRO-LIA/blob/master/Documentation/ProSpecRef.pdf

[7] Google, Página web de Blockly. [24/5/2018] (En línea)
Disponible en: https://developers.google.com/blockly/

[8] Node.js Foundation, Página del entorno Node.js. [24/5/2018] (En línea)
Disponible en: https://nodejs.org/es/

Este documento esta firmado por



	Firmante	CN=tfgm.fi.upm.es, OU=CCFI, O=Facultad de Informatica - UPM, C=ES
	Fecha/Hora	Sun Jun 03 11:14:42 CEST 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)