Facultad de Ciencias

Desarrollo de un simulador de robots basados en Arduino

Development of an Arduino-based robot simulator

Trabajo de fin de grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Sergio Palomera Salas

Director: Domingo Gómez Pérez

Julio de 2021

Agradecimientos

A mi familia, por haber estado siempre ahí para apoyarme y aconsejarme.
A mis compañeros de clase y amigos, especialmente a Emilio, con el que más tiempo y momentos he compartido en mi paso por la Universidad.
A Domingo, el director de este proyecto, que me ha ayudado en todo momento durante el desarrollo de este trabajo.

Desarrollo de un simulador de robots basados en Arduino

Resumen

palabras clave: Arduino, Simulador, Compilador, Educación, STEM

La teoría de grafos es clave en la enseñanza de los estudios STEM (Ciencia, Tecnología, Ingeniería y Matemáticas), que agrupan las cuatro áreas de conocimiento científico-técnico. Una innovación en los últimos veinte años es la experimentación con tecnologías en el ámbito de la educación informática. El uso de robots para utilizarlos en problemas de encaminamiento de grafos es una propuesta atractiva por su potencial visual y su valor didáctico. En el grado de ingeniería informática de la Universidad de Cantabria ya se han realizado prácticas en las que se han utilizado robots. Aunque la experiencia en general es positiva, se detectó una gran carga de trabajo extra para los estudiantes, ya que aparte de la complejidad conceptual de los algoritmos, se sumaban las dificultades técnicas inherentes a la utilización de dispositivos físicos como pueden ser robots, microcontroladores, etc. Esto suponía un problema, ya que gran parte del tiempo se empleaba en solventar errores de uso de los robots en vez de dedicarlo al aprendizaje de algoritmos de encaminamiento.

Este proyecto resuelve varias de las carencias detectadas en las experiencias piloto con esta tecnología. El simulador que se ha programado soluciona parte de los problemas presentados en las prácticas como la detección de errores en la programación de los robots y permite la abstracción de la programación del encaminamiento. Indirectamente, permite la realización de las prácticas con menos materiales y reducir el tiempo necesario para superarlas.

El simulador está programado en Python, incluye un traductor de C a Python, es de código libre y permite la depuración de errores paso a paso con una interfaz intuitiva.

Development of an Arduino-based robot simulator

Abstract

keywords: Arduino, Simulator, Compiler, Education, STEM

Graph theory is key to the teaching of STEM education (Science, Technology, Engineering and Mathematics), which groups the four areas of scientific-technical knowledge. One innovation in the last twenty years is the experimentation with technologies in the computer education scope. The use of robots in graph routing problems is an attractive proposal due to their visual potential and didactic value. Practices involving robots have already been carried out in the computer engineering degree at the University of Cantabria. Even though the overall experience is positive, a great extra workload for the students was detected, considering that, besides the conceptual complexity of the algorithms, there were also technical difficulties attached to the use of physical devices such as robots, microcontrollers, etc. This meant a problem, because the majority of the time was employed to solve robot-usage errors instead of devote it to the learning of routing algorithms.

This project resolves some of the shortages detected in the pilot experiences using this technology. The simulator solves part of the problems arised during practices like error detection during robot programming and allows the routing programming abstraction. Indirectly, it allows to carry out the practices with less materials and reduces the required time to finish them.

The simulator is programmed in Python, includes a C-to-Python translator, is free software and allows step-by-step debugging with an intuitive interface.

Índice

1 Introducción

El uso de algoritmos y su correspondiente aplicación en la resolución de problemas han sido siempre conceptos difíciles de explicar, además de ser percibidos como complejos por parte de los alumnos, debido a que requieren de la adquisición de pensamiento algorítmico para poder traducirlos en forma de programas [Futschek and Moschitz, 2010]. Uno de los muchos casos utilizados para ilustrar el uso de algoritmos ha sido el recorrido de grafos. Además de mediante una aproximación teórica, una forma práctica de mostrar el recorrido de grafos ha sido el uso de robots, programados utilizando la tecnología Arduino (debido a su bajo coste y a la gran cantidad de recursos facilitados por su comunidad de usuarios). Pero esto supone una dificultad adicional al uso de los propios algoritmos, y es que la comprobación del correcto funcionamiento de los robots es un proceso más largo y complicado: además de tener que proporcionar el nuevo código al robot cada vez que se realice algún cambio en la implementación, hay que esperar a comprobar si este se comporta como debería, lo que resulta en un proceso de pruebas extenso y tedioso.

Esto se pudo comprobar en las prácticas realizadas en la asignatura Algorítmica y Complejidad, en las que cada recorrido del robot consumía una gran cantidad de tiempo teniendo en cuenta que muchas veces el código creado por los alumnos debía modificarse, al no cumplir con todos los posibles casos presentados en los circuitos. Esto resultaba en que una gran parte del tiempo de la sesión se consumía únicamente en asegurar el correcto funcionamiento del programa desarrollado.

1.1. Descripción de las prácticas

Las prácticas realizadas en Algorítmica y Complejidad consisten en la programación de robots para recorrer un circuito determinado. Este circuito es un grafo conexo no dirigido, en el que los nodos que lo componen se representan como curvas cerradas. El objetivo es recorrer el circuito entero, guardando los vértices visitados. El grafo puede estar compuesto por hasta ocho nodos, numerados entre el 0 y el 7. En cada uno de ellos se codifica una etiqueta que lo identifica de forma única.

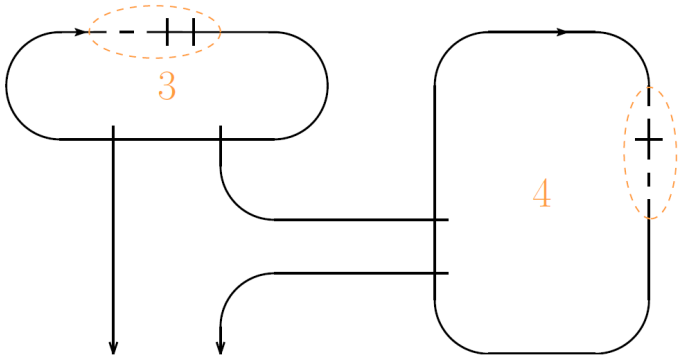

Ilustración 1: Disposición de los vértices en el circuito

Para que el robot pueda reconocer las aristas y etiquetas del circuito, estas se presentan mediante el uso de bits. Las aristas equivalen a un bit uno y las etiquetas son una serie de cuatro bits. Esta serie es el valor binario del número del nodo, y su primer bit siempre es cero, indicando su comienzo. En esta implementación, los ceros se corresponden a una discontinuidad en la cinta aislante, mientras que los unos se muestran como un segmento de cinta que corta la curva del nodo de forma ortogonal. El robot utiliza los bits como guía para explorar el grafo, siguiendo un recorrido dextrógiro. En la ilustración 1 se muestra la disposición de los vértices en un circuito, cada uno identificado por su etiqueta y conectado con los demás con aristas. La ilustración 2 exhibe los diferentes bits que presenta un nodo y cómo estos sirven para reconocer la etiqueta (comienza con un hueco en la cinta) y las aristas, las cuales indican al coche dónde se encuentra.

Los robots son programados utilizando el lenguaje Arduino y la librería «Cnosos», que cuenta con las funciones necesarias para que puedan desplazarse en el circuito. Al emplear Arduino, los programas desarrollados deben incluir dos funciones: «setup()», en la que se incluyen las tareas iniciales a realizar y «loop()», en la que se especifican acciones que van a repetirse de manera indefinida. Las funciones incluidas en la librería «Cnosos» son las siguientes:

- «lee_nodo(etiqueta, grado, entrada)»: el robot recorre el perímetro de un nodo y devuelve la etiqueta del nodo, su grado (el número de aristas incidentes con dicho nodo) y el número de orden de la arista en la que comienza la función (si está empieza tras la etiqueta, el valor es 0).
- «sal_aqui()»: el robot toma la arista recién encontrada y se desplaza hasta vértice de destino.
- «sal(int)»: el robot toma la arista especificada por el parámetro, siendo 1 la primera tras la etiqueta. Esta función supone que el robot acaba de pasar el bit de comienzo de la etiqueta.
- «siguiente()»: el robot avanza por el circuito hasta encontrar un hueco en la cinta (devuelve «False») o un trozo de cinta atravesada (devuelve «True») y sobrepasa ese bit.
- «sal_izq()»: el robot toma una salida a la izquierda.
- «lee_numero()»: el robot lee la etiqueta de un nodo. Esta función supone que el robot ha pasado el bit de comienzo de la etiqueta.
- «luce_numero(int)»: muestra un número a través del LED del robot.
- «error()»: indica un acontecimiento inesperado en el comportamiento del robot.

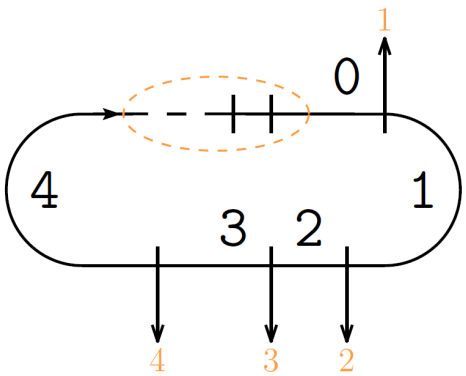

Ilustración 2: Disposición de los bits en un nodo

1.2. Objetivo

El objetivo principal de este trabajo es el desarrollo de un simulador que permita a los alumnos realizar esas mismas prácticas. Para ello, el simulador recibe el código desarrollado por el usuario, el cual está programado en C (aunque utilizando las funciones «setup()» y «loop()» de Arduino), y, mediante el uso de un compilador, procesa las instrucciones especificadas, exhibiendo en una interfaz gráfica los movimientos que realizaría el robot en un circuito elegido por el propio usuario.

Hemos elegido desarrollar un simulador para cumplir esta tarea porque, como exponen Cobos et al. [2020], el uso en el aula de un simulador facilita la comprensión de los distintos algoritmos por parte de los estudiantes, al mostrar este un entorno de fácil comprensión para poder practicar el desarrollo de algoritmos, además de observar los resultados.

Lo que buscamos con este proyecto es complementar el uso de los procesadores Arduino con un simulador que sirva a los alumnos para implementar y revisar sus soluciones, reservando el uso de los robots para el momento en que estas resuelvan el problema planteado.

Organizar las prácticas de esta manera facilita el proceso de pruebas de los algoritmos, permitiendo al alumno centrarse en la escritura del código, al evitar el constante traspaso de código al robot. Además, se agilizan las comprobaciones, al no tener que esperar a que el robot recorra el circuito, y también se evita tener que reservar espacios tales como aulas o laboratorios para permitir el movimiento de los robots (al menos durante este periodo de ensayos).

Una vez que los alumnos verifican sus algoritmos en el simulador, pueden transferirlos a los robots, evaluando si proceden de la misma forma. A pesar de emplearse solo en el último tramo de las prácticas, mantenemos el uso de los robots en el aula, ya que aportan una experiencia más visual, que una interfaz gráfica no puede igualar.

2 Requisitos

Los requisitos de un sistema son las descripciones de los servicios que debería proporcionar, además de las restricciones durante su uso [Sommerville, 2016, cap. 4.1]. Estos requisitos suelen clasificarse en dos categorías: funcionales y no funcionales. En este capítulo se detallarán los diferentes requisitos del proyecto.

2.1. Requisitos funcionales

Los requisitos funcionales describen los servicios que el sistema debe suministrar a los usuarios. En la siguiente tabla se muestran los requisitos del simulador:

Identificación	Descripción
RF01	El simulador permitirá al usuario elegir los nodos que compondrán el circuito.
RF02	El simulador permitirá al usuario elegir las aristas que unan los nodos del circuito.
RF03	El simulador permitirá al usuario elegir el fichero cuyas instrucciones se van a ejecutar.
RF04	El simulador permitirá al usuario elegir en qué nodo iniciar la simulación.
RF05	El simulador permitirá al usuario elegir si desea realizar una ejecución paso a paso.
RF06	El simulador permitirá al usuario avanzar al siguiente paso de la simulación.
RF07	El simulador permitirá al usuario borrar uno o varios de los nodos creados.
RF08	El simulador permitirá al usuario eliminar los parámetros proporcionados a la simulación, siendo estos todos los nodos creados, las aristas y el fichero indicado.
RF09	El simulador permitirá al usuario iniciar la simulación.

Cuadro 1: Requisitos funcionales

2.2. Requisitos no funcionales

Los requisitos no funcionales son limitaciones en los servicios ofrecidos por el sistema. En el cuadro 2 se muestran los requisitos no funcionales del proyecto.

Los motivos que han llevado a especificar estos requisitos son los siguientes:

RNF01 El simulador debe estar escrito en Python a petición del director del proyecto, Domingo, que solicitó expresamente el uso de este lenguaje de programación, por si tuviera que realizar cambios en el código.

RNF02 Al ser este proyecto un simulador de Arduino, debe recibir un fichero con instrucciones de la misma forma que lo haría un auténtico procesador Arduino.

RNF03 Los números de los diferentes nodos del circuito deben estar entre el 0 y el 7 debido a las propias restricciones establecidas por las prácticas de laboratorio que el simulador busca emular, en las cuales cada nodo se identifica mediante una etiqueta compuesta por cuatro bits, siendo el primero siempre 0.

Identificación	Tipo	Descripción
RNF01	Desarrollo	El simulador debe estar escrito en Python.
RNF02	Usabilidad	El usuario debe elegir un fichero para poder iniciar la simulación.
RNF03	Usabilidad	El usuario solo puede elegir nodos del 0 al 7.
RNF04	Usabilidad	El archivo con las instrucciones del coche puede encontrarse en cualquier directorio del sistema.
RNF05	Compatibilidad	El archivo con las instrucciones del coche debe tener extensión .c o .ino, debe ser un fichero de texto y con codificación ASCII.

Cuadro 2: Requisitos no funcionales

RNF04 El fichero con las instrucciones del coche puede estar almacenado en cualquier parte del sistema de archivos de modo que el usuario pueda colocarlo donde desee, no necesariamente en el directorio en el que estén los archivos del simulador.

RNF05 De igual forma que un coche Arduino recibe un archivo con extensión .ino, el simulador solo compila ficheros con esa misma extensión. También se aceptan archivos .c.

3 Herramientas

3.1. SLY

SLY (Sly Lex Yacc) es una librería de Python que implementa las herramientas necesarias para escribir analizadores sintácticos y compiladores. Orientada únicamente a la escritura del código por parte del usuario, SLY prueba los compiladores creados sin necesidad de generar ningún nuevo archivo ni ejecutar otros pasos que no se correspondan al propio compilador. Puede soportar gramáticas compuestas por cientos de reglas y proporciona un amplio informe y diagnóstico de errores como asistencia a la escritura del analizador sintáctico.

Para poder desarrollar el compilador, SLY proporciona dos clases diferentes: Lexer y Parser. La clase Lexer divide el texto de entrada en un grupo de tokens según una serie de expresiones regulares, mientras que la clase Parser reconoce la sintaxis del lenguaje, especificada como una gramática libre de contexto.

Hemos elegido SLY como herramienta para la creación del compilador por los siguientes motivos: el primero es que es una librería que procesa código escrito en Python, requisito fundamental para la implementación del proyecto; en segundo lugar, es una librería con la que ya estábamos familiarizados; por último, SLY es una adaptación modernizada de PLY, librería destacada por su simplicidad. De este modo, aunque existan herramientas para análisis sintáctico y léxico más sofisticadas, tales como ANTLR o Lark, estas poseen funciones que no son necesarias para la creación del compilador requerido por el proyecto, por lo que se han descartado.

3.2. Pygame

Pygame es una librería de Python para el diseño de aplicaciones multimedia, tales como videojuegos (exclusivamente en 2D) de manera sencilla. Hace uso de la librería SDL, por lo que puede procesar imágenes, reproducir sonidos y controlar el uso de periféricos.

En cualquier programa creado utilizando Pygame se inicializa la interfaz y se crea un bucle infinito, en el cual se procesan todos los eventos que afecten al programa, tales como uso del teclado o del ratón, para ejecutar las acciones que haya especificado el usuario.

Hemos utilizado Pygame para el desarrollo de la interfaz por las siguientes razones: al igual que sucede con SLY, la librería empleada debe estar escrita en Python, para así cumplir con los requisitos no funcionales del proyecto; además, Pygame es una de las herramientas más sencillas de utilizar en este lenguaje, ya que otras como BYOND o Godot no están disponibles en Python. La mejor alternativa sería Pyglet, la cual, a pesar de suministrar características que Pygame no posee, como soporte 3D o uso de múltiples ventanas al mismo tiempo, cuenta con una comunidad y popularidad mucho más reducidas, lo que supone una menor cantidad de material y ejemplos a disposición del usuario.

4 Casos de uso

Los casos de uso son una forma de describir todas las posibles interacciones entre los usuarios y un sistema, utilizando tanto modelos gráficos como texto [Sommerville, 2016, cap. 5.2]. En este capítulo se muestran los diferentes casos de uso del simulador utilizando un diagrama, así como plantillas para describirlos con más detalle.

4.1. Diagrama de casos de uso

En un diagrama de casos de uso se muestran los posibles actores, enlazados con todas las acciones que pueden realizar al interactuar con el sistema. En este proyecto solo existe un tipo de actor, el usuario. En la ilustración 3 se muestran los casos de uso del simulador.

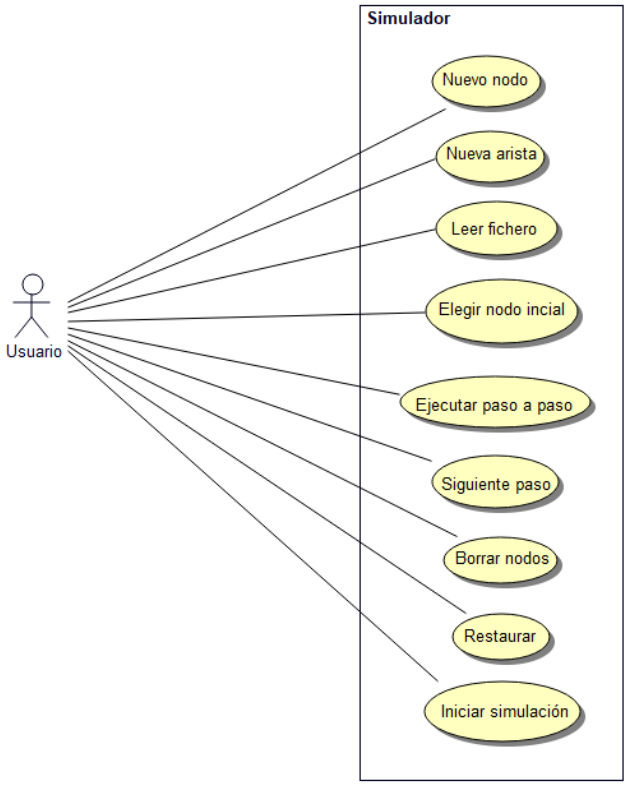

Ilustración 3: Casos de uso del simulador

4.2. Plantilla de casos de uso

Los diagramas de casos de uso proporcionan solo un simple resumen de las interacciones entre los actores y el sistema, por lo que es necesario añadir más información para completar las descripciones de cada caso de uso. En este proyecto, dicha información se muestra en forma de tablas.

En el cuadro 3 se muestran las variables y estructuras de datos que maneja el simulador y que son modificadas durante su uso.

Nombre	**Descripción**
Lista de nodos	Lista con los números de los nodos que se han añadido al grafo.
Lista de aristas	Lista de listas con las adyacencias, esto es, la posición de cada lista da el nodo origen y la lista da los nodos adyacentes.
Fichero	Ruta del fichero a compilar.
Nodo inicial	Número del nodo en el que comienza la simulación.
Modo paso a paso	Variable booleana que indica si la simulación se realizará paso a paso o no.

Cuadro 3: Datos manejados por el simulador

A continuación se muestra una serie de tablas con los datos de cada caso de uso del sistema.

Nombre	Iniciar simulación.
Descripción	El usuario inicia la simulación del programa.
Actores primarios	Usuario.
Actores secundarios	-
Evento de activación	El usuario pulsa el botón “Simulación”.
Precondiciones	1. El circuito debe tener al menos un nodo. 2. El usuario debe haber especificado el fichero a compilar.
Flujo principal	1. El simulador muestra el circuito en la interfaz. 2. El simulador muestra el código del fichero elegido en la interfaz. 3. El simulador compila el fichero de entrada, creando la traducción a Python. 4. El simulador ejecuta el fichero Python, mostrando en la interfaz el movimiento del coche a lo largo del circuito.
Postcondiciones	-
Flujos alternativos	-

Cuadro 4: Iniciar simulación

Nombre	Nuevo nodo.
Descripción	El usuario añade un nuevo nodo al circuito.
Actores primarios	Usuario.
Actores secundarios	-
Evento de activación	El usuario pulsa el botón “Nuevo nodo”.
Precondiciones	-
Flujo principal	1. El simulador muestra el cuadro de texto “Nodo” para introducir el número del nuevo nodo. 2. El usuario introduce el número del nodo. 3. El simulador comprueba que el carácter introducido sea válido (un número entre 0 y 7). 4. El simulador comprueba que no exista un nodo con ese número. 5. El simulador añade el nuevo nodo a la lista de nodos. 6. El simulador muestra en la interfaz un nuevo cuadro de texto, “Aristas”, que tiene asociado el número del nodo recién añadido. 7. El simulador muestra en la interfaz, bajo el nuevo cuadro de texto, el número del nuevo nodo, además de escribirlo por consola. 8. El simulador muestra en la interfaz un botón con el número del nodo. 9. Si el nodo añadido es el único del circuito, entonces: 9.1. Se establece el nuevo nodo como nodo inicial. 9.2. El botón correspondiente cambia de color inactivo a activo, indicando que el número mostrado por el botón es el del nodo inicial.
Postcondiciones	El nodo queda incluido en la lista de nodos.
Flujos alternativos	3a. El carácter introducido no es un número entre 0 y 7. 1. El simulador muestra un mensaje indicando el error, tanto por consola como en el cuadro de texto. 4a. Ya existe un nodo con ese número. 1. El simulador muestra un mensaje indicando el error, tanto por consola como en el cuadro de texto.

Cuadro 5: Nuevo nodo

Nombre	Nueva arista.
Descripción	El usuario añade una nueva arista entre dos nodos.
Actores primarios	Usuario.
Actores secundarios	-
Evento de activación	El usuario introduce el nodo de destino de una arista en el cuadro de texto "Aristas".
Precondiciones	-
Flujo principal	1. El simulador comprueba que el carácter introducido sea válido (un número entre 0 y 7). 2. El simulador comprueba que los nodo de origen y destino no sean iguales. 3. El simulador comprueba que no haya ya una arista entre esos dos nodos. 4. El simulador añade la nueva arista a la lista de aristas. 5. El simulador muestra en la interfaz, bajo el cuadro de texto "Aristas" utilizado, los nodos de origen y destino de la nueva arista, además de escribirlos por consola. 6. Si el nodo de destino no existe, entonces: 6.1. El simulador muestra en la interfaz un nuevo cuadro de texto con el número del nodo de destino. 6.2. El simulador muestra en la interfaz un nuevo cuadro "Aristas", que tiene asociado el nodo de destino. 6.3. El simulador muestra en la interfaz, bajo los nuevos cuadros de texto, el número del nodo de destino. 6.4. El simulador muestra en la interfaz un botón con el número del nodo de destino.
Postcondiciones	El nodo de destino queda añadido en la lista de nodos (en caso de no existir previamente) y la nueva arista se añade a la lista de aristas.
Flujos alternativos	1a. El carácter introducido no es un número entre 0 y 7. 1. El simulador muestra un mensaje indicando el error, tanto por consola como en el cuadro de texto. 2a. Los nodos de origen y destino de la arista coinciden. 1. El simulador muestra un mensaje indicando el error, tanto por consola como en el cuadro de texto. 3a. La arista especificada ya existe. 1. El simulador muestra un mensaje indicando el error, tanto por consola como en el cuadro de texto.

Cuadro 6: Nueva arista

Nombre	Leer fichero.
Descripción	El usuario escoge el fichero que compilará el simulador.
Actores primarios	Usuario.
Actores secundarios	-
Evento de activación	El usuario pulsa el botón “Leer fichero”.
Precondiciones	-
Flujo principal	1. El simulador muestra una ventana que permite al usuario buscar en los archivos locales el fichero que quiere utilizar en la simulación (solo ficheros .c o .ino). 2. El usuario escoge el fichero. 3. El simulador cierra la ventana de búsqueda. 4. El simulador muestra en la interfaz y por consola el archivo elegido.
Postcondiciones	El fichero seleccionado se establece como archivo a analizar por el compilador.
Flujos alternativos	El usuario puede cerrar la ventana de búsqueda sin haber seleccionado ningún archivo.

Cuadro 7: Leer fichero

Nombre	Elegir nodo inicial.
Descripción	El usuario elige el nodo en el que comenzará la simulación.
Actores primarios	Usuario.
Actores secundarios	-
Evento de activación	El usuario pulsa uno de los botones con los números de los nodos.
Precondiciones	-
Flujo principal	1. Si el botón elegido no se corresponde con el del nodo inicial, entonces: 1.1. El número mostrado en el botón seleccionado se establece como nodo inicial de la simulación. 1.2. El botón seleccionado cambia de color inactivo a activo, indicando que el número que muestra es el nuevo nodo inicial. 1.3. El botón del nodo inicial anterior cambia de color de activo a inactivo. 1.4. El número del nuevo nodo inicial se muestra por consola.
Postcondiciones	El nodo seleccionado se establece como nodo inicial.
Flujos alternativos	-

Cuadro 8: Elegir nodo inicial

Nombre	Ejecutar paso a paso.
Descripción	El usuario decide si la ejecución de la simulación se realizará paso a paso o no.
Actores primarios	Usuario.
Actores secundarios	-
Evento de activación	El usuario pulsa uno de los dos botones ("Sí" o "No") que controlan la ejecución paso a paso.
Precondiciones	-
Flujo principal	1. Si el botón no se corresponde con el estado actual de la ejecución paso a paso: 1.1. Si el botón es "No", entonces: 1.1.1. El color del botón "No" cambia de inactivo a activo, indicando que no se realizará la ejecución paso a paso. 1.1.2. El color del botón "Sí" cambia de activo a inactivo. 1.1.3. El simulador muestra un mensaje por consola, indicando que la simulación no se realizará paso a paso. 1.2. Si el botón es "Sí", entonces: 1.2.1. El color del botón "Sí" cambia de inactivo a activo, indicando que la ejecución se realizará paso a paso. 1.2.2. El color del botón "No" cambia de activo a inactivo. 1.2.3. El simulador muestra un mensaje por consola, indicando que la simulación se realizará paso a paso.
Postcondiciones	El modo de ejecución de la simulación queda modificado según la elección del usuario.
Flujos alternativos	-

Cuadro 9: Ejecutar paso a paso

Nombre	Siguiente paso.
Descripción	El usuario avanza la simulación hasta el siguiente paso.
Actores primarios	Usuario.
Actores secundarios	-
Evento de activación	El usuario pulsa el botón "Siguiente paso".
Precondiciones	1. La simulación debe haber comenzado. 2. La ejecución paso a paso debe estar activada.
Flujo principal	1. La simulación avanza hasta encontrar la siguiente instrucción de la librería del coche o termina, en caso de que no queden más instrucciones que ejecutar.
Postcondiciones	-
Flujos alternativos	-

Cuadro 10: Siguiente paso

Nombre	Borrar nodos.
Descripción	El usuario borra uno o más nodos del circuito.
Actores primarios	Usuario.
Actores secundarios	-
Evento de activación	El usuario pulsa el botón “Borrar nodos”.
Precondiciones	El usuario debe haber creado al menos un nodo.
Flujo principal	1. El botón “Borrar nodos” cambia de color. 2. Se muestra un nuevo botón, “Cancelar”, que permite a los usuarios cancelar la operación de borrado. 3. Los botones correspondientes a los diferentes nodos del circuito cambian de color, indicando que los nodos pueden ser seleccionados para ser borrados. 4. El usuario elige los nodos que quiere borrar. 5. El usuario pulsa el botón “Borrar nodos”. 6. Los nodos seleccionados son eliminados, al igual que las aristas incidentes y los botones que representan dichos nodos. 7. Si quedan nodos sin borrar, entonces: 7.1. Los botones de los nodos restantes son colocados en orden en la interfaz. 7.2. Si el nodo inicial ha sido borrado, entonces: 7.2.1. El primer nodo de los restantes es establecido como el nuevo inicial. 7.3. Los botones de los nodos restantes recuperan sus colores anteriores: el color activo en el caso del nodo inicial y el inactivo en el caso de los demás. 8. Si no: 8.1. El nodo inicial queda sin asignar.
Postcondiciones	Los nodos especificados y sus aristas son eliminados.
Flujos alternativos	El usuario puede cancelar el borrado pulsando el botón “Cancelar” o, si no ha seleccionado ningún nodo, utilizando el botón “Borrar nodos”.

Cuadro 11: Borrar nodos

Nombre	Restaurar.
Descripción	El simulador vuelve a su estado inicial, eliminando todos los cambios introducidos por el usuario.
Actores primarios	Usuario
Actores secundarios	-
Evento de activación	El usuario pulsa el botón “Restaurar”.
Precondiciones	El usuario debe haber proporcionado algún dato al simulador, ya sean los nodos del circuito, el fichero a compilar o haber cambiado el modo de ejecución paso a paso.
Flujo principal	1. Si el circuito tiene algún nodo, entonces: 1.1. Los nodos y aristas del circuito son eliminados. 2. Si el usuario ha elegido un fichero para ser compilado, entonces: 2.1. El archivo a compilar es eliminado. 3. Si el usuario ha cambiado el modo de ejecución paso a paso, entonces: 3.1. El modo de ejecución paso a paso vuelve a su valor por defecto. 4. Si la simulación ya se ha realizado, entonces: 4.1. El dibujo del circuito y el código empleado son borrados de la interfaz.
Postcondiciones	Todos los datos anteriormente suministrados al simulador quedan eliminados.
Flujos alternativos	-

Cuadro 12: Restaurar

5 Desarrollo

5.1. Modelo de desarrollo

Para el desarrollo de este proyecto se ha utilizado el modelo incremental, que se basa en la creación de una implementación inicial del proyecto con la que conseguir retroalimentación por parte de los usuarios o clientes, para de este modo poder modificarlo, desarrollando sucesivas versiones hasta conseguir la implementación deseada. En este modelo, las fases de especificación, desarrollo y validación se realizan de forma intercalada, compartiendo la información obtenida en cada una con las demás [Sommerville, 2016, cap. 2.1]. En la ilustración 4 se muestra el funcionamiento del modelo incremental.

Hemos optado por el desarrollo incremental y no en cascada. En este último modelo, las diferentes fases se realizan de forma independiente, de modo que no se empieza una nueva etapa hasta haber terminado la anterior. Si ocurren cambios en los requisitos del proyecto durante las últimas fases, deben retomarse las anteriores. En el caso de este simulador, durante su desarrollo se han realizado algunos cambios en sus especificaciones que en un principio no se habían previsto, por lo que el modelo de cascada habría sido ineficaz.

Cada una de las diferentes versiones del proyecto, también conocidas como incrementos, incorpora nuevas funcionalidades requeridas por los clientes, aunque, en general, los incrementos más tempranos son los que implementan las características más importantes, de forma que el sistema se pueda evaluar para comprobar si cuenta con las funcionalidades necesarias. Las versiones posteriores son las que realizan cambios menores, adaptándose a las necesidades concretas de los clientes y modificando los requisitos que estos indiquen.

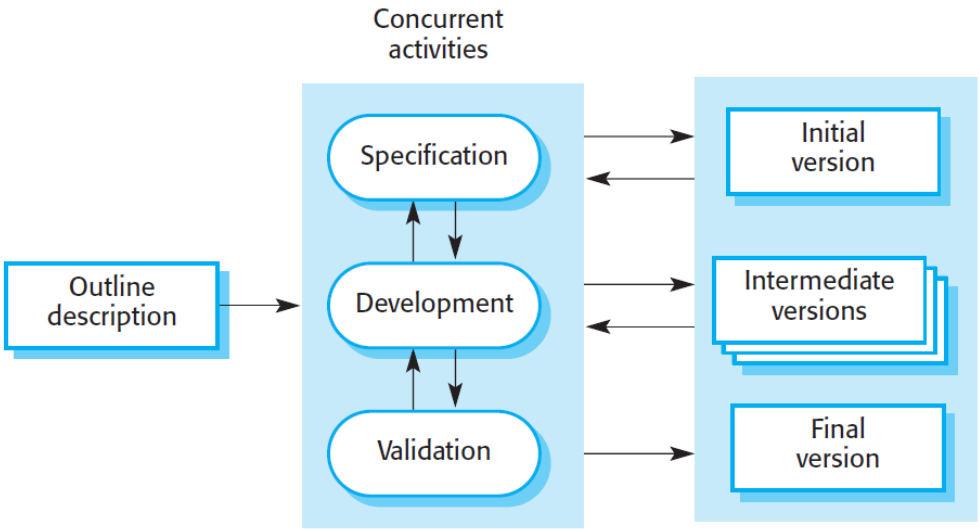

Ilustración 4: Desarrollo incremental [Sommerville, 2016]

Partiendo de la descripción del proyecto y la especificación de requisitos, se desarrolló la versión

inicial del simulador, que contaba con las características y componentes imprescindibles para su funcionamiento. Estos eran el compilador y la interfaz, los cuales, a pesar de no estar completados entonces, eran capaces de simular el comportamiento del coche Arduino. Tras haber finalizado este incremento inicial, se realizaron varias reuniones, a modo de fases de validación, en las que el director del proyecto indicó los servicios a implementar, siendo estos los constituyentes de las diferentes versiones intermedias. Por último, tras haber cumplido con todos los requisitos y características solicitadas, se validó el simulador, alcanzado este su versión final.

5.2. Incrementos del sistema

El desarrollo del simulador se ha llevado a cabo en cuatro versiones o incrementos principales, siendo cada uno de estos verificado por el director del trabajo:

1. El primer incremento realizado fue la creación del compilador, compuesto por los analizadores léxico y sintáctico, y el traductor de código C a código Python. Se comenzó el proyecto con el desarrollo del compilador, al ser este componente imprescindible para poder procesar las instrucciones de los ficheros suministrados al simulador.

2. El siguiente componente desarrollado fue la interfaz gráfica, la cual, en su versión inicial, permitía especificar los nodos y aristas del circuito, dibujarlo y probar el recorrido del coche utilizando las diferentes instrucciones de su librería, aunque con código incluido dentro de la propia interfaz, ya que aún no se había integrado este componente con el compilador.

3. Tras realizar la primera reunión, en la que se estableció el comportamiento de los elementos desarrollados como válido, se procedió con la siguiente versión del simulador, que integró los dos componentes anteriores (el compilador y la interfaz) permitiendo así seleccionar y ejecutar los ficheros con instrucciones del coche desde la propia interfaz y de este modo comprobar el funcionamiento del vehículo a lo largo del circuito. Además de la integración, también se realizaron ciertos cambios acordados en dicha reunión, como la inclusión de nuevos tipos de datos o el tratamiento de punteros (entre otros) en lo referente al compilador, y la inclusión de la ejecución paso a paso y la selección del nodo inicial de la simulación en el caso de la interfaz.

4. Después de verificar los cambios incluidos en la versión anterior, el último incremento del simulador consistió en emplear las directrices enunciadas por Johnson [2014] para el correcto diseño de interfaces, de modo que la empleada en el proyecto fuera fácil de comprender y utilizar. Los cambios realizados siguiendo las instrucciones de dicho libro se exponen en el capítulo referente a la interfaz.

6 Compilador

A pesar de que el simulador esté escrito en lenguaje Python, los ficheros de entrada con las instrucciones están escritos en C (deben incluir las funciones «setup()» y «loop()», de la misma forma que los programas Arduino). Esto hace necesaria una traducción de C a Python para que el simulador pueda ejecutar los archivos. Para llevarla a cabo, desarrollamos un compilador, que recibe un fichero como entrada, sobre el cual realiza los análisis léxico y sintáctico. Tras estos procesos, se emplea un traductor para generar un nuevo fichero compuesto por las instrucciones equivalentes en lenguaje Python. En la ilustración a continuación se muestra el funcionamiento del compilador, con las tres cajas identificando las partes en las que se divide. Las flechas a la izquierda de cada componente indican la entrada que estos reciben, mientras que las que se encuentran a la derecha identifican la salida generada por cada uno.

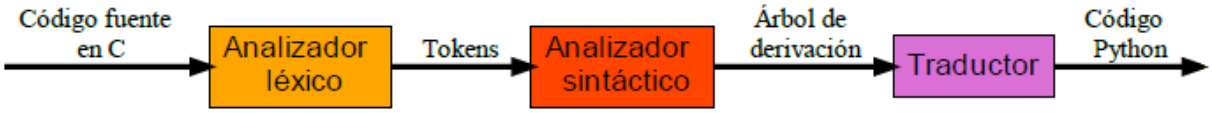

Ilustración 5: Funcionamiento del compilador

6.1. Analizador léxico

El primer paso para la creación del compilador consiste en el análisis léxico del código de entrada, que divide dicho código en tokens. Para conseguir esto empleamos la clase Lexer proporcionada por la herramienta SLY, que obtiene los tokens a partir del código empleando una serie de expresiones regulares.

El «lexer» debe especificar los tipos de tokens en que descompone el código (en el caso de SLY, los nombres especificados en el conjunto deben estar escritos en mayúsculas). Los tokens deben especificarse mediante el uso de reglas de expresiones regulares. Estas expresiones regulares deben de ser compatibles con el módulo re de Python para poder ser identificadas al ejecutar el analizador léxico. Cada token tiene un tipo y un valor: el primero se corresponde con el nombre de la expresión regular, mientras que el segundo equivale al conjunto de caracteres que coinciden con dicha expresión. Por ejemplo, si el nombre de la expresión regular es 'STRING', los posibles valores serán cadenas de caracteres entre comillas, mientras que si la expresión regular se llama 'INT', los valores correspondientes serán números enteros. En caso de que se quiera devolver algún carácter concreto ('+', '*', por ejemplo), este debe de ser incluido en el conjunto «literals», que iguala tanto el valor como el tipo de dicho token al propio carácter.

La clase Lexer, además, permite escribir funciones que realicen ciertas instrucciones al encontrar un token concreto, por ejemplo, cambiar el tipo del token o ignorar partes del texto, tales como

comentarios o saltos de línea. Esto puede hacerse escribiendo la expresión regular específica del token dentro del decorador «@_()». Este tipo de funciones resultan especialmente útiles para la localización de errores, ya que el «lexer» por sí solo no es capaz de identificar en qué posición del fichero se encuentran los tokens, por lo que es necesario crear métodos que proporcionen esta información. Por ejemplo, se puede crear una función que cambie el atributo «lineno», correspondiente al número de línea en la que se encuentra el analizador en un momento concreto de su ejecución, cada vez que se active la expresión regular correspondiente al salto de línea. Al saber en qué línea se encuentra el analizador léxico, se pueden reportar los errores no solo indicando el código erróneo, sino también especificando en qué posición del fichero se encuentran los fallos.

Si el «lexer» localiza un carácter erróneo durante la ejecución, esta se detiene. Si se quiere habilitar el tratamiento de errores, se puede incluir una función «error()», en la que se especifique qué hacer cuando se encuentre uno de estos caracteres. Esta función recibe como parámetro el resto del texto de entrada que falta por tokenizar, lo que permite proseguir la ejecución en busca de errores más adelante, localizando en una sola llamada al analizador todos los fallos presentes en el código.

Una vez que se han especificado las reglas, los literales y las funciones del analizador léxico, se puede llamar a la función «tokenize()», que es la única función pública proporcionada por la clase Lexer y es la encargada de producir y devolver el conjunto de tokens obtenidos al recorrer el texto de entrada, cada uno con su tipo y valor correspondiente.

6.2. Analizador sintáctico

El siguiente paso en el desarrollo del compilador es la creación del analizador sintáctico, el cual se encarga de reconocer la sintaxis del lenguaje, en este caso C. Esto lo hemos llevado a cabo utilizando la clase Parser proporcionada por SLY.

Para poder reconocer las diferentes reglas de la sintaxis de C, el «parser» se vale de los conjuntos de tokens y literales previamente definidos por el «lexer». Para identificar cada una de estas reglas, el analizador sintáctico implementa una serie de métodos, encabezados todos ellos por el decorador «@_(rule)». A este decorador se le indica la regla sobre la cual tiene que actuar, la cual puede contener otras reglas de la propia gramática, tokens o literales especificados por el «lexer», aunque estos últimos, a diferencia de los otros dos, deben especificarse entre comillas. Los métodos del «parser» se ordenan según su posición correspondiente en la gramática, de modo que la regla correspondiente al programa entero se coloca primero, terminando con los símbolos terminales, tales como números o caracteres. Independientemente de las instrucciones que se ejecuten en las funciones, estas deben retornar un valor, el cual quedará asociado a esa aplicación concreta de una regla. Estos valores pueden devolverse de diferentes formas. Por ejemplo, si se quiere retornar el resultado de una suma, se puede devolver únicamente ese resultado o una tupla compuesta por los sumandos y el símbolo '+', entre otras posibilidades.

El «parser» también puede incluir producciones vacías, ya que SLY proporciona soporte para las mismas. Esto permite especificar casos tales como los parámetros de una función, ya que un puede no recibir ninguno.

En muchos casos, la escritura de las reglas puede resultar en la especificación de gramáticas ambiguas. Cuando se genera una de estas gramáticas, el «parser» avisa de la existencia de conflictos. Para solucionar los problemas de ambigüedad, la clase Parser permite el uso de la variable «precedence», la cual incluye el orden de precedencia y asociatividad (izquierda, derecha o ninguna) que se quiere dar a las diferentes reglas, especificando primero las de mayor preferencia. En el caso de este compilador, hemos empleado el orden de precedencia establecido por C.

6.3. Traducción a código Python

Por último, se devuelve el código C traducido a Python. Para esta implementación hemos utilizado el módulo dataclasses, que permite la creación de clases de forma más sencilla, sin necesidad de

implementar funciones tales como «___init___()» o «___repr___()». Dependiendo de los parámetros elegidos al utilizar el decorador «@dataclass» proporcionado por la librería, se le puede indicar al módulo que añada dichos métodos a la clase o no. Al crear una clase se especifican los atributos y su tipo, además de las funciones.

Como hemos explicado anteriormente, las funciones del «parser» deben retornar el valor de las reglas de la gramática de alguna forma. En esta implementación, cada método devuelve una clase, definida mediante «dataclasses». Los atributos de cada clase dependen de la parte de la gramática que representen. De este modo, la clase correspondiente a las constantes recibe como parámetros el nombre, el tipo y el valor, mientras que la clase que representa a las funciones recibe el tipo de la función, el nombre, los parámetros y el cuerpo. Esto mismo se aplica para todas las demás clases: atributos, operaciones aritméticas, variables, etc.

Cada una de estas clases, además de los ya mencionados atributos, cuenta con una función «str(n)», la cual, en este proyecto, es la encargada de expresar esa regla de la gramática según su equivalencia en Python, utilizando los atributos de la propia clase. El parámetro «n» de esta función representa el número de veces que es necesario sangrar el texto obtenido dentro del nuevo fichero Python, por lo que si se tratase de una variable global, «n» valdría cero, pero si fuese una instrucción dentro de una función, «n» valdría uno.

Además de la traducción de lenguaje C a Python, en algunos casos ha sido necesaria la inclusión de instrucciones exclusivas de Python, para de esta forma implementar correctamente las instrucciones del fichero original.

Al emplear variables globales en C, su valor se puede cambiar directamente dentro de una función, pero en Python no ocurre esto, ya que al hacer la asignación, se interpreta como la creación de una nueva variable con el mismo nombre que la variable global, de modo que el valor de esta última no se ve afectado al finalizar la ejecución. Para solucionar esto hemos utilizado una instrucción «global», que hemos antepuesto a cada cambio de valor de una variable global realizado dentro de una función.

Algunas funcionalidades permitidas por el compilador solo pueden utilizarse en Python mediante la importación de módulos, tal es el caso de las estructuras y la función «sleep()». Para implementar las primeras de la forma más fiel a C hemos utilizado la librería ctypes, la cual permite emplear funcionalidades de C en Python. Utilizando ctypes, los campos de las estructuras se especifican dentro de una lista compuesta por tuplas, siendo cada uno de sus elementos el nombre del campo y su tipo, respectivamente. En el caso de la función «sleep()», es necesario importar el módulo time.

Los «switch statement» de C se han sustituido por instrucciones condicionales «'if-elif'» anidadas.

El último cambio realizado en la traducción ha sido la inclusión de la función «espera()», encargada de la ejecución paso a paso de la simulación. Esta función se añade en el código Python después de cualquiera de las funciones propias de la librería para el control del coche, de modo que, al indicar a la simulación que continúe con el siguiente paso (en caso de que la simulación paso a paso esté habilitada, ya que en caso contrario la función se ignora), esta lo hará hasta encontrar la siguiente función de la librería. Detallamos el funcionamiento de «espera()» en el capítulo referente a la interfaz del simulador.

6.4. Cambios con respecto a C

Los ficheros que emplea el simulador deben estar escritos en C, pero solo hemos implementado un subconjunto de la gramática de dicho lenguaje en el compilador. Sin embargo, sí hemos incluido algunos recursos ajenos a C para facilitar la escritura del código.

Estos añadidos son los tipos de datos «bool» y «byte», las funciones «printf()» y «delay()» y los métodos definidos en la librería para el manejo del coche. Todas ellas pueden utilizarse sin necesidad de añadir los «#include» correspondientes.

Otras «keywords» de C, en cambio, no son indispensables para el simulador, por lo que no las hemos incluido, simplificando así el compilador. Algunas de las funcionalidades omitidas son «auto», «extern», «register» o «volatile». Sin embargo, sí hemos incluido la directiva «#include», pero de una forma simplificada, de modo que en el fichero generado en lenguaje Python se muestran como

comentarios los «include» empleados, sin ninguna funcionalidad real. De forma análoga a la palabra clave «include», los prototipos de funciones en C se muestran como comentarios en la traducción, ya que en Python no se hace uso de prototipos.

Por último, los punteros (tanto de dirección como indirección) son analizados por el compilador, pero tratados como cualquier otra variable, de modo que sus funcionalidades como contenedores de direcciones de memoria resultan ignoradas.

7 Interfaz

Para mostrar al usuario cómo se comportaría el robot Arduino, se ha creado una interfaz en la que el usuario puede especificar tanto el circuito a recorrer como el fichero con el código que el coche emplearía. Tras haber comprobado el correcto funcionamiento del simulador hemos realizado cambios en la interfaz siguiendo las directrices indicadas por Johnson [2014].

7.1. Diseño de la interfaz

Al iniciar el simulador, este muestra la interfaz gráfica, la cual, en su estado inicial, cuenta con los botones y datos necesarios para preparar los parámetros de la simulación, tales como los nodos y aristas del circuito o el fichero a ejecutar. Conforme el usuario cambie estos parámetros, la interfaz presentará la nueva información en forma de botones, texto y demás elementos, además de mostrar los mensajes, tanto de error como de éxito, en la interfaz y por consola.

7.1.1. Disposición inicial

El diseño inicial de la interfaz se muestra en la ilustración 6. En la parte superior izquierda se encuentran tres botones, etiquetados como “Nuevo nodo”, “Leer fichero” y “Simulación”, los cuales se encargan de suministrar las opciones necesarias para mostrar la simulación del vehículo. Estas permiten al usuario incluir nuevos nodos en el circuito, elegir el fichero cuyas instrucciones serán compiladas y ejecutadas por el simulador, e iniciar la ejecución de la simulación, respectivamente. En la parte superior derecha de la interfaz, en cambio, están localizadas las funciones opcionales cambiar el nodo inicial y habilitar el modo de simulación paso a paso. En la parte derecha de la interfaz también aparece una etiqueta de texto, “Código”, bajo la cual se mostrarán la dirección de memoria del fichero elegido y su contenido.

Ilustración 6: Interfaz al iniciar la simulación

7.1.2. Interfaz antes de la simulación

Para poder comenzar la simulación, es imprescindible que el usuario especifique tanto el circuito a recorrer como el fichero a procesar, generando ambas elecciones cambios directos en los elementos mostrados en la interfaz (ilustración 7).

Al pulsar el botón “Nuevo nodo”, se añade a la interfaz un cuadro de texto etiquetado con el texto “Nodo”, en el cual el usuario puede introducir el número del nodo que quiere crear. Si se introduce un número válido, se incluyen nuevos elementos en la interfaz: un cuadro de texto acompañado de la etiqueta “Aristas” y, bajo dicho cuadro, el número del nodo recién añadido. Desde este nuevo cuadro se pueden añadir aristas con origen en el nodo asociado al cuadro, especificando el destino. Al crearse una nueva arista, se muestran en la interfaz su origen y su destino y, en caso de que el nodo final no existiese, se incorporan a la interfaz dos cuadros de texto (“Nodo” y “Aristas”) correspondientes a ese nuevo nodo.

Cada vez que se incluye un nuevo nodo en el circuito, además de las acciones ya mencionadas, el simulador crea un nuevo botón, mostrado en la parte derecha de la interfaz, bajo la leyenda “Nodo inicial”. Cada uno de estos botones tiene como texto el número de su nodo correspondiente. El botón activo (marcado con un color diferente) indica el nodo del circuito donde comenzará la simulación.

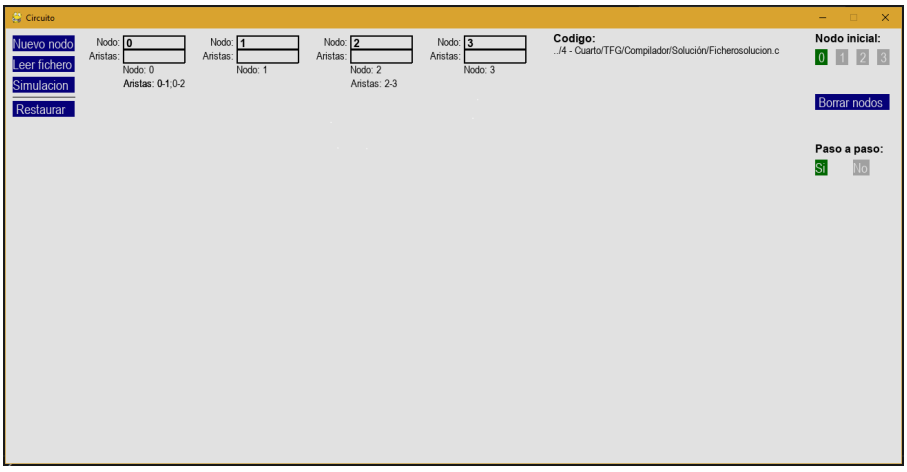

Ilustración 7: Interfaz antes de la simulación

La ejecución paso a paso tiene dos botones asociados: “Sí” y “No”. Al seleccionar uno u otro se activa o desactiva la simulación paso a paso, respectivamente. Al igual de lo que ocurre para los botones que seleccionan el nodo de arranque de la simulación, el botón marcado se señala con un color distinguido. Esto supone que, al seleccionar el 4 como comienzo de la simulación, por ejemplo, este será el único botón que tenga el color activado. Lo mismo sucede con las opciones “Sí” y “No”: al seleccionar una, la otra mostrará el tono asociado a la acción deshabilitada.

El botón “Leer fichero” abre una ventana que permite al usuario buscar el archivo que quiere compilar. Una vez elegido, la ventana se cierra, y la dirección de memoria del fichero se muestra en pantalla, bajo la leyenda “Código”.

Al realizar el usuario cambios en los parámetros de la simulación, dos nuevos botones aparecen: “Borrar nodos” y “Restaurar”. El primero aparece específicamente al añadir nodos al circuito, colocándose bajo los botones para la selección del nodo inicial; mientras que el segundo se muestra además tras la selección del fichero con las instrucciones del coche o el cambio del modo de la ejecución paso a paso, ocupando la posición bajo el botón “Simulación”.

Si el usuario quiere eliminar alguno de los nodos del circuito, así como las aristas incidentes, debe usar el botón “Borrar nodos”. Este componente, al pulsarse, desencadena tres modificaciones en la interfaz: cambia el color de los botones de los nodos; añade un nuevo botón, “Cancelar”, en caso de que se quiera abortar la operación de borrado, y cambia su propio color. Durante el proceso de eliminación de nodos, sus distintos botones asociados utilizan dos colores diferentes, para distinguir los

seleccionados para eliminarse. Ninguno de estos dos colores coincide con los empleados para diferenciar el nodo inicial de los demás, evitando así confusiones. Además, el tono empleado para distinguir los nodos que serán eliminados es el mismo que el que toma el botón “Borrar nodos” durante el proceso. Una vez que se lleva a cabo el borrado (pulsando de nuevo “Borrar nodos”), ocurren los siguientes cambios en la interfaz: el botón “Cancelar” desaparece, “Borrar nodos” recupera su color anterior y tanto los cuadros de texto como los botones relacionados con los nodos desaparecidos se eliminan de la interfaz, colocando los componentes restantes sin dejar huecos intermedios. Otras modificaciones menores son las etiquetas de texto con las aristas del circuito, que son alteradas para que dejen de mostrar los nodos eliminados. Por otra parte, si aún quedan nodos pero el nodo inicial ha sido suprimido, su papel como comienzo de la simulación es tomado por el actual primer nodo de la lista. El botón asociado a este nuevo nodo inicial cambia al color correspondiente.

Por último, el botón “Restaurar” elimina todos los datos proporcionados a la simulación, siendo estos todos los nodos y aristas del circuito, el fichero elegido y la configuración paso a paso, devolviendo al simulador a su estado inicial. Lo mismo ocurre con la interfaz, ya que todos los componentes mostrados en ella a causa de las acciones del usuario son descartados, incluyendo la representación del circuito y el código empleado, en caso de haberse ejecutado la simulación, lo que deja a la interfaz con la misma apariencia que al inicio de la ejecución.

7.1.3. Interfaz durante la simulación

Los últimos cambios que sufre la interfaz son causados por el uso del botón “Simulación”, que hace que comience el recorrido del coche por del circuito especificado por el usuario. Esto trae consigo tres nuevos añadidos a la interfaz: una representación gráfica del circuito, el código del fichero elegido y el botón “Siguiente Paso” (ilustración 8).

El circuito se muestra con sus nodos agrupados en dos filas, cada una de estas con capacidad para hasta cuatro nodos, aunque solo se hace uso de la fila inferior si el circuito tiene más de cuatro vértices. El recorrido del coche comienza inmediatamente, empezando por el nodo establecido como inicial y desplazándose por el circuito según las instrucciones especificadas por el usuario.

A la derecha del circuito se coloca el código C con esas instrucciones. Esta información se incluye en la interfaz porque será útil para indicar al usuario en qué parte de la ejecución se encuentra el programa: tras llamar a una de las funciones de la librería del coche, se coloca un marcador al lado de dicha instrucción, indicando que es la que acaba de ser ejecutada.

La disposición de esta información se realiza en dos pasos. En primer lugar la escritura del código en la interfaz, la cual se lleva a cabo antes de la propia ejecución del fichero; después de esto, la inclusión del indicador del paso actual de la simulación, actualizado durante todo el proceso mediante la función «espera()», presentada en el capítulo referente al compilador.

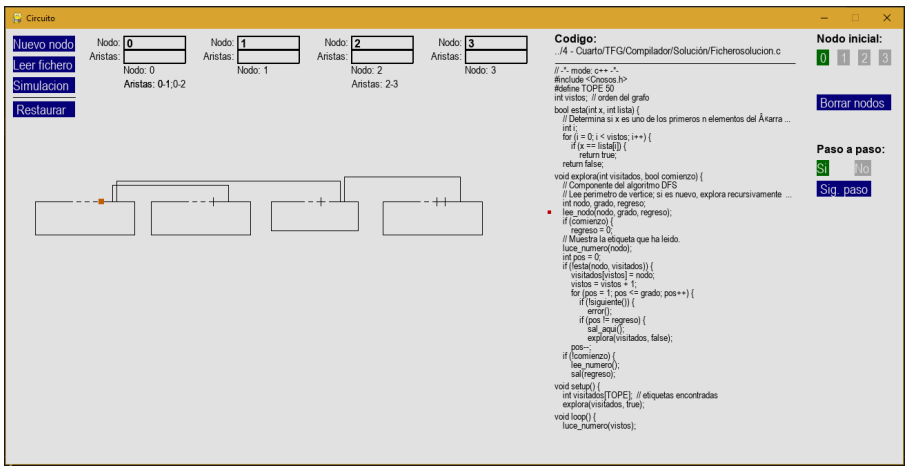

Ilustración 8: Interfaz durante la simulación

El proceso para mostrar en la interfaz el código se encarga, además de escribir las propias líneas, de procesar ambos ficheros, tanto el escrito en C como su equivalente en Python, para más adelante proporcionar los datos necesarios a la función «espera()» y que esta pueda indicar la instrucción que ha ejecutando el simulador en ese momento.

Este procedimiento se realiza en varios pasos, siendo el primero recorrer el fichero C, leyendo las líneas a copiar en la interfaz, las coordenadas en las que se encuentra cada una y varias listas con las posiciones ocupadas por cada una de las instrucciones de la librería del coche dentro del fichero (tanto la posición de las líneas como sus coordenadas serán empleados más adelante por el simulador). A continuación, el compilador realiza un análisis léxico y sintáctico del fichero C y genera la traducción correspondiente en Python. Una vez que se ha realizado la traducción, se recorre el nuevo fichero para crear las listas en las que almacenar las posiciones de cada una de las instrucciones de la librería del coche, de la misma forma que se hizo con el archivo original.

El siguiente paso consiste en importar el fichero contenedor del simulador y el módulo inspect en el archivo Python. El primero cuenta con todas las funciones relacionadas con el funcionamiento de la interfaz, incluyendo aquellas que adaptan las funciones de la librería del coche para su correcta disposición en la simulación. El módulo «inspect», en cambio, se utiliza dentro de la función «espera()» para obtener el número de línea desde el cual ha sido llamada.

Esto lleva al último paso del procedimiento, consistente en mostrar qué parte del código acaba de ejecutar el simulador. Al invocar la función «espera()», esta devuelve el número de línea en el cual se ha producido la llamada. Como se ha explicado en el capítulo referente al compilador, «espera()» se añade una línea después de cualquiera de las otras funciones de la librería del coche, de modo que, al recibir el resultado devuelto por «espera()», el simulador le resta uno para situar en el código Python la función ejecutada. Este dato se compara con cada una de las listas creadas al recorrer el archivo Python, para determinar tanto en qué lista se encuentra como la posición que ocupa dentro de ella. Con estos dos datos, se obtiene la misma posición de la lista análoga del archivo C, que indica la posición de la instrucción correspondiente en el fichero C. Así, el simulador puede señalarla en la interfaz.

En caso de que la ejecución paso a paso esté habilitada, el botón “Siguiente paso” aparece en la interfaz. Este tipo de ejecución implica que, cada vez que el simulador llame a una de las instrucciones del coche, la simulación se detendrá, colocando junto a dicha instrucción el marcador explicado anteriormente. Cuando el usuario pulse el botón, el coche reanudará su movimiento hasta que vuelva a ejecutar otra de las instrucciones de su librería.

7.2. Reglas de diseño de la interfaz

Una vez comprobado el correcto funcionamiento de los distintos componentes y características de la interfaz, la revisamos para verificar si el diseño es el adecuado para que los usuarios la utilicen de la forma más intuitiva posible. Para llevar a cabo esta evaluación, hemos seguido las directrices expuestas por Johnson [2014], cambiando la interfaz en caso de que no se cumpliese alguna de las directrices que plantea.

7.2.1. Percepción del usuario

El primer concepto presentado en el libro es el sesgo en la percepción de los usuarios y cómo el diseño de la interfaz debería tener este factor en cuenta [Johnson, 2014, cap.1]. Para evitar problemas durante su uso, una buena interfaz debería evitar ambigüedad, de modo que todos los usuarios la interpreten siempre de la misma forma; ser consistente, mostrando la información y los componentes relacionados en la misma posición y con similares características como color o fuente, y entender las metas del usuario, ya que lo que este percibe está fuertemente influenciado por sus objetivos, por lo que se debe mostrar la información relacionada con estos en todo momento. La interfaz empleada en este proyecto cumple los tres requisitos, al estar diseñada para cumplir una función muy concreta y contar con pocos componentes. De este modo, estos siempre ocupan la misma posición, mostrando al

usuario en todo momento los progresos que ha realizado con los parámetros de la simulación. Además, la disposición del circuito se realiza colocando los nodos en el orden en que los ha introducido el usuario, de modo que la representación del grafo se ajusta a la percepción del usuario.

Para diferenciar aún más los elementos de la interfaz entre sí y, a su vez, distinguir estos del fondo, es muy importante hacer un uso correcto del color [Johnson, 2014, cap.4]. Para ello hay que tener en cuenta los contrastes entre colores, el tamaño de los objetos y el texto (que debe ser lo suficientemente grande para poder distinguir su color) y la separación que hay entre ellos (cuanto más cerca se encuentren dos elementos, más fácil será diferenciar sus colores). Además, si se quiere llamar la atención del usuario por medio del color, la forma más eficaz de hacerlo es utilizar colores distintivos como blanco, negro, rojo, azul, amarillo o verde, que son los que mayor impacto causan en el sistema visual. Al utilizar estos colores, hay que evitar colocarlos junto a sus opuestos, ya que estas combinaciones de colores resultan molestas.

Hemos empleado estas directrices de color en el proyecto, eligiendo los tonos dependiendo del tipo de componente. El circuito, así como las etiquetas y cuadros de texto se muestran en color negro, para destacar sobre el fondo gris claro (no hemos utilizado el blanco porque puede resultar molesto, especialmente en entornos poco iluminados). Los colores de los botones vienen determinados por su función, si realizan una tarea como añadir o borrar nodos, iniciar la simulación o pasar a su siguiente paso. Estos botones se muestran con un color fijo (azul) de modo que contrasten con el fondo y el usuario interprete que están habilitados para poder usarlos. En cambio, los botones que suponen una elección entre varias opciones (siendo estas elegir el nodo inicial y el modo de ejecución) pueden tener dos colores, dependiendo de si la opción que representan está activa o no. Los dos colores son verde (para la opción marcada) y gris. Esta elección de colores se ha elegido con un propósito claro, que el usuario vea que las funciones que cumplen son diferentes. Al utilizar la función de borrado de nodos, los diferentes botones asociados a estos utilizan dos nuevos colores, distintos del gris y el verde empleados en la elección del nodo inicial (ilustración 9). Estos dos nuevos colores son rosa y rojo, siendo el rojo el que señala los nodos marcados para borrar, como también es rojo el color que adopta el botón “Borrar nodos”.

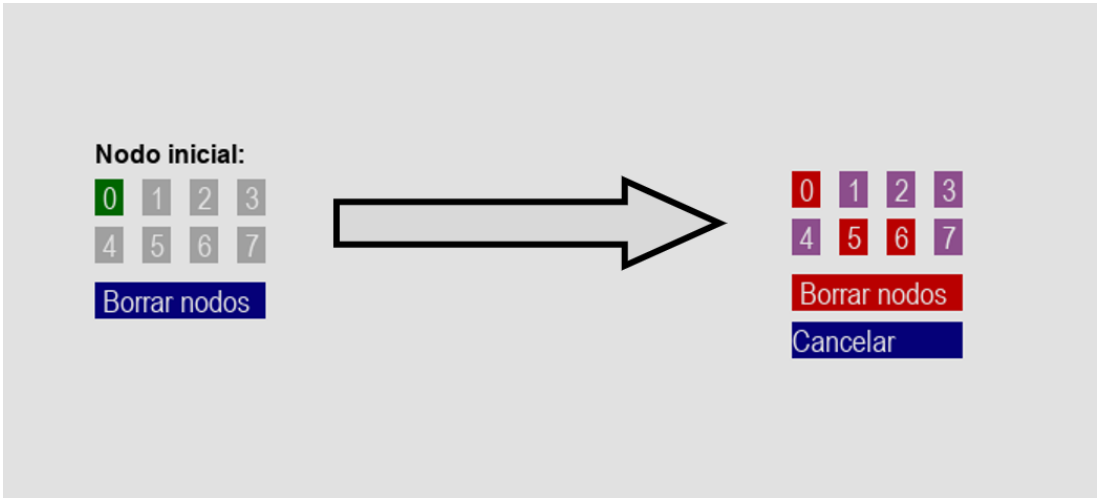

Ilustración 9: Uso de los colores al borrar los nodos

El coche y el indicador de la instrucción recién ejecutada, en cambio, están representadas en naranja y rojo, respectivamente, de modo que, aunque sus colores sean ligeramente diferentes, se establezca una relación entre ellos, al ser el movimiento del coche el causante directo de los movimientos del indicador a lo largo de las instrucciones del código. Esta forma de implementarlo en el simulador sirve para depurar la ejecución paso a paso.

7.2.2. Visión del usuario

La visión humana está optimizada para agrupar los objetos en estructuras, especialmente al estar estos relacionados por factores como proximidad, similitud, continuidad o simetría, entre otros [Johnson, 2014, cap.2]. En el diseño de esta interfaz, solo se han empleado los dos primeros, al ser los más determinantes. Los botones están agrupados según su funcionalidad, encontrándose cada grupo en extremos opuestos de la interfaz. Los botones relacionados con parámetros imprescindibles de la simulación (añadir nodos al circuito, elegir el fichero a compilar o iniciar la propia simulación) se encuentran agrupados a la izquierda, mientras que los encargados de especificar parámetros opcionales (elegir el nodo inicial, eliminar nodos o ejecutar la simulación paso a paso) están en la parte derecha. La única excepción a esta regla es el botón "Restaurar", el cual, a pesar de no ser necesario para poder probar el funcionamiento del vehículo, se ha colocado en la parte izquierda, junto a los botones imprescindibles, para permitir al usuario empezar una nueva simulación de forma rápida. Los cuadros de texto, en cambio, están agrupados todos juntos al cumplir la misma función (añadir nodos o aristas al circuito).

La visión periférica de las personas es pobre: si se muestra información en la interfaz lejos del punto al que están mirando (generalmente el cursor o el próximo lugar al que lo piensan desplazar), es muy posible que los usuarios pasen esos nuevos datos por alto [Johnson, 2014, cap.5]. Si se quiere que los nuevos mensajes incluidos en la interfaz sean percibidos, deben colocarse cerca de donde el usuario esté mirando y, para el caso de que sean mensajes de error, mostrarlos con un color característico, generalmente rojo.

En la interfaz de este proyecto, el botón para añadir nodos al circuito está colocado a la izquierda, de modo que los cuadros de texto para especificar nuevos nodos y aristas aparezcan a la derecha de dicho botón, siguiendo así el orden de lectura habitual (de izquierda a derecha) y facilitando que el usuario se percate de la aparición de los nuevos elementos en la interfaz. Tras haber añadido cuatro nodos, los cuadros de texto y etiquetas relacionados con los siguientes aparecen en una nueva fila, debajo de los cuatro anteriores, manteniendo así el orden de lectura y facilitando la visión de los datos añadidos al mostrarse en dos filas en vez de solo una muy larga. De forma similar, la dirección en la que se encuentra el fichero elegido aparece a la derecha de los cuadros de texto correspondientes a los nodos, de modo que mirando en un único sentido el usuario obtiene toda la información sobre los parámetros imprescindibles de la simulación. Al incluir el usuario nuevos nodos en el circuito aparecen los botones correspondientes para elegir el nodo inicial de la simulación. Estos botones se encuentran a la derecha de la interfaz, lejos de la zona de atención del usuario, al no ser una característica imprescindible para el funcionamiento del simulador. El circuito y el código del fichero elegido tampoco aparecen cerca del botón que los muestra, pero esto se debe a que ambos elementos ocupan el suficiente espacio como para que el usuario los detecte con su visión periférica.

En lo referente a los mensajes de error, en un principio se mostraban solo por consola, pero, al caber la posibilidad de que los usuarios no le prestasen atención, se han incluido también en la interfaz. Si los fallos en la simulación están causado por no haber añadido ningún nodo al circuito o no haber elegido el fichero de código, las descripciones de esos problemas aparecen justo debajo del botón "Simulación" (ilustración 10), ya que es su uso el que causa que los errores se manifiesten. Los mensajes de fallo no desaparecen hasta que el usuario proporcione esos datos, dependiendo de cuál sea el problema que indique la interfaz. Los mensajes de error referentes a la inclusión de nodos ya creados o aristas repetidas, entre otros, se muestran dentro del cuadro de texto cuyo uso ha disparado el error, manteniéndose hasta que el usuario pulse alguna tecla. Todos los mensajes de error se escriben en rojo, para contrastar con el resto del texto (escrito en negro).

7.2.3. Capacidades del usuario: atención y memoria

Al diseñar una interfaz, hay que tener en cuenta que la atención que pueden prestar los usuarios a sus tareas es limitada y su memoria, imperfecta [Johnson, 2014, cap.7]. Por estos dos motivos hay que diseñar las interfaces de forma que ayuden a los usuarios a lo largo de todos los pasos que realicen hasta conseguir sus objetivos. Esto se consigue limitando la información presentada, en el caso de la

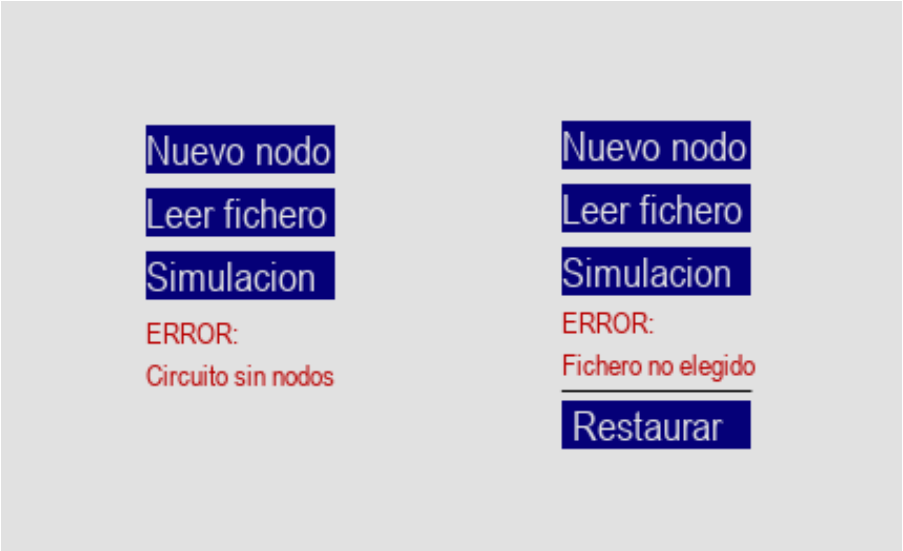

Ilustración 10: Mensajes de error al iniciar la simulación

atención, y mostrando en todo momento a los usuarios los datos que necesiten para avanzar, en el caso de la memoria, de modo que no tengan que centrarse en recordar, sino en las metas a completar usando la interfaz. No es buena práctica para facilitar las tareas del usuario el uso de diferentes modos para los mismos componentes (como tomar fotos o grabar vídeos usando el mismo botón), de modo que los usuarios tengan que recordar qué es lo que hace cada uno, como tampoco lo es la creación de jerarquías con muchos niveles, ya que los usuarios deben recordar no solo donde están, sino a dónde quieren llegar.

Este nivel de sencillez ha sido fácil de implementar, al no ser necesarios demasiados componentes para hacer funcionar al simulador, lo cual implica que no ha habido la necesidad de programar la interfaz con diferentes modos de funcionamiento ni organizar sus datos en varios niveles. Los únicos componentes que tienen varios modos de uso son los botones de los nodos, que pueden usarse para elegir el comienzo del circuito o para eliminar los nodos creados. Estos botones están diseñado de esa forma por dos motivos. Por una parte, la diferencia entre un modo y otro es clara, al ser necesario interactuar con el botón “Borrar nodos” y cambiar los botones de los nodos de color. Por otra, si se hubieran desplazado los botones o creado otros diferentes para la función de borrado, el usuario tendría que prestar atención a más elementos en pantalla.

Continuando con lo expuesto sobre la atención de las personas, si estas quieren llevar a cabo una tarea empleando un determinado software, es muy posible que, si dicha tarea se desarrolla en varios pasos, los usuarios se olviden de la parte de la ejecución donde están o qué acciones quedan por realizar [Johnson, 2014, cap.8]. Por este motivo, además de crear interfaces que no distraigan a los usuarios de sus objetivos, es igual de importante guiarlos a lo largo de todo el proceso, indicando lo que se ha hecho y lo que aún queda por hacer para finalizar las tareas. El ser humano piensa de forma cíclica a la hora de completar una tarea, dividiéndose dicho ciclo en tres etapas: escoger una meta, ejecutar acciones que nos acerquen a ella y evaluar si se ha alcanzado el objetivo. Estas tres fases se repiten hasta que se complete la tarea o se establezca como inalcanzable. Para apoyar este ciclo de pensamiento, la interfaz debe proporcionar caminos claros para lograr el objetivo facilitado por el software, dar información a los usuarios que los acerque a su meta y mostrar el progreso o finalización de la tarea en la interfaz.

La interfaz del simulador está diseñada de forma que respalda cada una de las fases de este ciclo de pensamiento, aunque ha sido necesario modificarla para facilitar las fases de ejecución y evaluación. La meta del usuario es comenzar la simulación, empezando esta al pulsar el botón “Simulación”; el camino para lograr este objetivo se presenta de forma que dicho botón se encuentra bajo otros dos, que son los que permiten al usuario establecer los parámetros necesarios para llevar a cabo la simulación, de modo que, siguiendo el orden natural de lectura de arriba abajo, los usuarios empleen primero los

botones superiores antes de simular. La fase de ejecución se apoya mostrando a los usuarios los nodos y aristas que añaden al circuito, además del fichero que escogen (mostrar la dirección del fichero es una de las características que ha sido necesario añadir para facilitar esta fase), de modo que sepan en todo momento que la información introducida será utilizada por el simulador. Por último, en la etapa de evaluación, al pulsar el botón "Simulación", el usuario puede ver si esta ha comenzado con éxito o si ha habido algún error que requiere solución, ya que, en el primer caso se mostrarán el circuito y el coche recorriéndolo, mientras que en el segundo aparecerá un mensaje de error indicando el problema.

El ser humano no está programado para leer, por lo que las interfaces deberían incluir la menor cantidad de texto posible, evitando así desviar a los usuarios de su objetivo [Johnson, 2014, cap.6]. Además de incluir pocas palabras, es importante evitar las siguientes elecciones de diseño: incluir vocabulario poco habitual o extraño para los usuarios, utilizar fuentes y colores difíciles de leer, mostrar las palabras con un tamaño insuficiente o colocar los textos sobre fondos que dificulten la lectura, entre otras.

Estas directrices han sido fácilmente aplicables en el simulador, al estar pensado para cumplir una tarea muy específica con un vocabulario muy concreto, por lo que la cantidad de texto necesario para informar al usuario es mínima, empleándose solo para identificar los diferentes componentes, la información introducida por los usuarios y los mensajes de error. Esto hace que el usuario pueda centrarse completamente en preparar la simulación, ya que no tiene que leer ningún texto complicado o entender unas instrucciones extensas. En lo que respecta a la claridad de las palabras para su lectura, estas siempre utilizan un color que contraste con su entorno, que tiene un color fijo, sin ninguna imagen o elemento similar que complique la lectura. Además, se emplea una fuente sencilla con un tamaño lo suficientemente grande para poder verse sin problemas.

7.2.4. Coordinación del usuario

Cuando las personas usan interfaces gráficas, en algún momento tendrán que hacer uso de su coordinación ojo-mano, ya sea para pulsar un enlace, seleccionar una imagen o desplegar un menú, entre otros [Johnson, 2014, cap.13]. Esta coordinación sigue determinadas leyes, siendo la más importante para este proyecto la ley de Fitts, la cual establece que, cuando se quiere apuntar a un determinado objetivo, se alcanzará antes cuanto mayor sea su tamaño y más cerca se esté de él. El tiempo necesario disminuirá al reducirse la distancia al objetivo o aumentar su tamaño, hasta llegar a un límite en el que el tiempo permanecerá invariable, por mucho que se reduzca el recorrido o se incrementen las dimensiones del objetivo. Si se busca que los usuarios empleen el mínimo tiempo posible alcanzando los diferentes elementos interactivos de la interfaz, estos deben cumplir una serie de características: ser lo suficientemente grandes, para poder ser utilizados por los usuarios de forma sencilla; hacer la zona interactiva tan grande como lo sea el componente, de modo que el usuario no se frustre al clicar en un componente que no reacciona; y dejar suficiente separación entre botones, enlaces y otros elementos, de modo que el usuario no utilice erróneamente componentes cercanos entre sí.

Los principios de diseño de la ley de Fitts anteriormente descritos se cumplen en nuestra interfaz. Tanto los botones como los cuadros de texto tienen el suficiente tamaño como para ser fácilmente utilizados y mostrar su información de forma clara. Además, la zona hábil de ambos tipos de componentes se corresponde con las dimensiones mostradas, de modo que el usuario pueda hacer clic sobre ellos en cualquier parte de su superficie para iniciar su correspondiente acción. Por último, la separación entre todos los botones y cuadros de texto es la suficiente para diferenciar unos de otros, excepto en un caso, el de la pareja de cuadros de texto de un nodo y sus aristas, los cuales se encuentran inmediatamente uno encima de otro, para mostrar así su estrecha relación. Sin embargo, cada pareja de cuadros de texto está bien separada de las demás, haciendo claramente distinguibles unos nodos de otros.

7.2.5. Prevención de errores y capacidad de respuesta de la interfaz

Si al diseñar un producto software se busca que los usuarios aprendan a utilizarlo lo más rápido posible, conociendo todas las herramientas que el sistema proporciona, este debe tolerar los errores cometidos por los usuarios, haciéndolos fáciles de solucionar [Johnson, 2014, cap.11]. Esto se debe a

que la productividad de las personas se ve reducida si el software utilizado hace que los usuarios se equivoquen frecuentemente, no puedan arreglar los errores cometidos o sí que puedan, pero empleando mucho tiempo en corregirlos. Si al usar un sistema los usuarios pierden mucho tiempo lidiando con sus fallos, lo que se conoce como sistemas de alto riesgo, no tratarán de aprender a utilizar todas las funcionalidades que el software provee, sino que utilizarán solo las características que ya conocen. Por el contrario, en un sistema de bajo riesgo, los fallos son difíciles de cometer, además de fáciles de solucionar, lo que reduce el estrés del usuario, facilitando el aprendizaje de las características del software. Para crear un sistema de bajo riesgo, es necesario implementar las siguientes medidas: prevenir los errores siempre que sea posible, desactivar comandos inválidos, hacer los fallos fáciles de detectar indicando a los usuarios las acciones que los han causado y permitir revertir los errores de forma sencilla.

En este proyecto hemos hecho algunos cambios para poder incluir características propias de sistemas de bajo riesgo, que en un principio no habían sido incluidas al no considerarse necesarias, pero que hemos añadido para facilitar la interacción de los usuarios con la interfaz. En lo que respecta a la prevención de errores, en la versión inicial del sistema, al especificar el número de un nodo se aceptaba como entrada cualquier carácter o grupo de caracteres, informando después al usuario del error en caso de no haber introducido un número en el rango correcto (de 0 a 7). En la versión final, en cambio, solo se permite introducir un único número, bloqueando el teclado en caso de que el usuario quiera escribir más y, si la entrada no es un número entre 0 y 7, se notifica el error inmediatamente (ilustración 11). Esta nueva notificación de errores también se utiliza al querer incluir un nodo repetido, una arista ya utilizada o una cuyo nodo de origen y destino sea el mismo. En la versión final del proyecto, además de mostrar textos descriptivos explicando los errores cometidos dentro o cerca de los componentes de la interfaz que los han provocado (botones o cuadros de texto), hemos implementado nuevas funcionalidades para que los usuarios puedan revertir los cambios introducidos, siendo estas opciones borrar los nodos creados y eliminar todos los parámetros modificados. Ambas permiten al usuario interactuar con el simulador de forma más rápida, al poder realizar cambios sin tener que reiniciar el programa.

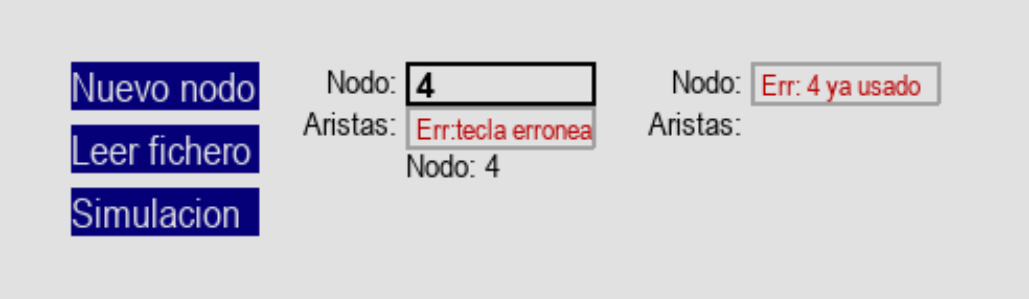

Ilustración 11: Prevención de los errores del usuario

El último de los factores a tener en cuenta a la hora de diseñar interfaces es la capacidad de respuesta, siendo este el más importante al determinar la satisfacción de los usuarios, más incluso que la eficiencia. Esto se debe a que, por muy rápido y eficaz que sea un programa informático al cumplir sus tareas, si este no informa a los usuarios de lo que sucede, lo percibirán como menos efectivo al no tener claro si sus acciones están siendo tomadas en cuenta por el sistema [Johnson, 2014, cap.14]. Para mejorar la capacidad de respuesta de una interfaz, esta debe adaptarse a los requerimientos de tiempo de los usuarios, lo cual se consigue informándoles de todo lo que sucede, incluso aunque sus peticiones no puedan ser cumplidas de forma inmediata. Algunos ejemplos de buena capacidad de respuesta son hacer saber al usuario de forma inmediata que la información introducida se ha recibido o indicar cuánto tiempo tardará la tarea actual en completarse. En cambio, algunos de los ejemplos que causan que una interfaz se perciba como ineficiente son proporcionar retroalimentación de forma tardía al pulsar botones, desplazar el cursor o manipular objetos; iniciar tareas que consumen recursos e impiden el uso de la interfaz pero no pueden ser canceladas; o no informar del tiempo restante para completar un proceso. La retroalimentación proporcionada por el sistema debe mostrarse de forma

instantánea para mantener la percepción de causa y efecto por parte de los usuarios, ya que, si estos reciben los datos con retraso, interpretarán que sus acciones no están relacionadas con los cambios en la interfaz, y percibirán el software como ineficaz.

La interfaz de este simulador tiene gran capacidad de respuesta al mostrar al usuario nuevo texto, botones y otros componentes de forma instantánea siempre que este realiza una acción. Como se ha dicho anteriormente, el utilizar los botones hará que estos cambien de color, aparezcan nuevos cuadros de texto y etiquetas o la simulación sea iniciada, dependiendo de cuál sea su función. Lo mismo sucede con los mensajes de error, que aparecen al realizar el usuario cualquier acción incompatible con las características de los parámetros del simulador, como puede ser intentar iniciar el recorrido del coche sin haber proporcionado un circuito o un fichero para compilar. A pesar de estas características, no hemos podido aplicar todas las directrices relativas a la capacidad de respuesta, debido a la propia implementación del simulador. En concreto, no hemos podido indicar el progreso o tiempo restante de la simulación, ya que no puede determinarse al ejecutarse la simulación de forma recursiva. Esto se debe a que no se puede saber cuántas instrucciones quedan por ejecutar. Tampoco hemos habilitado la cancelación de tareas largas, como puede ser la simulación del vehículo. Esto se debe a que el archivo es importado como un módulo Python, lo que hace que se ejecuten todas las instrucciones del coche.

8 Conclusiones

El simulador creado en este proyecto permite a los alumnos de Algorítmica y Complejidad probar de forma más rápida y sencilla el código desarrollado para las prácticas, utilizando los robots solo cuando hayan verificado su validez. Ejecutar el simulador evita tener que esperar a que el coche termine el recorrido, lo cual, dependiendo del número de nodos y aristas del circuito puede ser tedioso, especialmente si hay que repetirlo varias veces por fallos en el código. Durante estos ensayos tampoco se requiere reservar espacios físicos en las aulas para probar el funcionamiento de los vehículos. El ahorro de tiempo en el periodo de pruebas es especialmente notorio si el número de circuitos o coches disponibles es insuficiente para proveer a todos los estudiantes.

En cambio, mediante el uso del simulador, cada estudiante o grupo de estudiantes puede probar su propia implementación empleando únicamente un ordenador, lo que les permitiría modificar y verificar su código en cualquier lugar, no solo en el aula, además de acortar cada prueba individual, al ser el desplazamiento del coche simulado mucho más rápido que el del robot real.

El simulador proporciona dos herramientas diferentes: el compilador y la interfaz gráfica. El primero analiza el código creado por los alumnos, que debe estar escrito con el mismo lenguaje que un programa Arduino, mientras que la interfaz permite a los estudiantes diseñar el circuito a recorrer, eligiendo el número de nodos y las aristas entre ellos; seleccionar el nodo en el que comenzará la simulación; y elegir o no realizar la simulación paso a paso, para ver en detalle todos los movimientos que realiza el coche, observando así de forma más clara los comportamientos anómalos del coche.

8.1. Pruebas realizadas

Durante el desarrollo del proyecto llevamos a cabo tres tipos de pruebas: unitarias, de componentes y del sistema, con el fin de comprobar si tanto los elementos por separado como las interacciones entre ellos funcionan de la forma esperada. Al haber creado el simulador utilizando el modelo incremental, probamos cada nueva versión del proyecto no solo bajo nuevas situaciones, sino utilizando también las pruebas anteriores, para verificar que las funcionalidades añadidas no causaran problemas con las ya existentes.

8.1.1. Pruebas unitarias

Utilizamos las pruebas unitarias para comprobar el correcto funcionamiento del analizador léxico y la interfaz, ya que son los únicos elementos del proyecto que pueden probarse sin necesidad de interactuar con otros componentes.

En el caso del analizador léxico, realizamos las pruebas con el objetivo de verificar que los diferentes elementos de cualquier fichero de código se identifican correctamente (ya sean tipos de datos como «int» o «float»; «strings», números enteros, etc.). Para ello probamos archivos con código que contenían varios ejemplos de todas las expresiones regulares que incluye el analizador léxico, además de caracteres literales. En estas pruebas utilizamos código erróneo, ya que los errores sintácticos se identifican al ejecutar el «parser».

A la hora de probar la interfaz, ejecutamos todas sus funcionalidades: crear nodos, añadir aristas, cambiar el nodo inicial, etc. También hicimos pruebas con el recorrido del coche (pasándole las instrucciones directamente a la interfaz, sin emplear el compilador aún), utilizando diferentes circuitos para

ver el desempeño del vehículo en todos ellos. Realizamos todas las pruebas de forma tanto correcta como incorrecta, para asegurar así que los posibles errores fueran correctamente identificados y notificados. Al comprobar las funcionalidades de la interfaz, empleamos también los componentes asociados (botones, cuadros de texto), para así verificar que las interacciones del usuario con la interfaz fueran identificadas y tratadas correctamente.

8.1.2. Pruebas de componentes

Las pruebas de componentes consisten en probar la correcta interacción entre distintos objetos del sistema, para ver si las relaciones entre ellos se comportan de la forma esperada. Realizamos este tipo de pruebas para comprobar cómo procesa el analizador sintáctico los datos facilitados por el «lexer» y cómo interactúa el traductor de Python con las instrucciones identificadas por el «parser».

Para probar el analizador sintáctico, de forma similar a lo hecho con el «lexer», utilizamos diferentes ficheros con código, presentando cada uno de ellos varios ejemplos de las diferentes reglas implementadas en el «parser», además de instrucciones mal escritas, para comprobar si el analizador sintáctico era capaz tanto de identificar las diferentes reglas (sin confundir aquellas que tienen una sintaxis similar, por ejemplo) como de localizar errores en el código debidos a palabras mal escritas o instrucciones colocadas en el orden incorrecto, entre otros.

Tras haber verificado el correcto desempeño del analizador sintáctico, procedimos a probar el traductor a Python. Para ello empleamos código correctamente implementado, al detectarse los errores por el «parser». Ejecutamos los diferentes ficheros traducidos para asegurar que se generaran correctamente.

8.1.3. Pruebas del sistema

El último tipo de pruebas que realizamos fueron las pruebas del sistema, en las cuales todos los componentes del simulador se integran para comprobar que todos ellos son compatibles entre sí, interactúan de forma correcta y transfieren los datos entre ellos de forma adecuada. Al haber hecho pruebas previamente sobre el compilador, el traductor y la interfaz, la única característica del sistema sin comprobar en esta etapa era el tratamiento de los ficheros traducidos a Python por parte de la interfaz. Por este motivo, las pruebas del sistema consistieron en proporcionar diferentes ficheros para su posterior traducción y ejecución. El primer archivo utilizado fue uno proporcionado por el director del proyecto, Domingo, con el código necesario para que el coche recorriera el circuito entero. Una vez verificado ese, probamos otros con todas las instrucciones que puede llevar a cabo el coche, incluyendo la función para reportar errores. Tras haber comprobado todos ellos, dimos el proceso de pruebas del sistema por concluido.

8.2. Trabajos futuros

El simulador creado en este proyecto cumple con todos los cometidos para los que fue inicialmente concebido, por lo que no hay ninguna modificación más que realizar con el fin de mejorar su desempeño.

A pesar de esto, el simulador sí que podría ser modificado para poder adaptarse a los requisitos de otros proyectos que requieran alguna de las características presentes en este, como pueden ser el uso de un compilador o de una interfaz de usuario desarrollada en Python.

El código fuente del proyecto está en un repositorio en GitLab.

Referencias

M. Cobos, M. D. R-Moreno, and D. F. Barrero. R2P2: Un simulador robótico para la enseñanza de inteligencia artificial. 2020. URL http://isg.aut.uah.es/assets/pdfs/jenui2020.pdf.

G. Futschek and J. Moschitz. Developing algorithmic thinking by inventing and playing algorithms. 2010. URL https://www.researchgate.net/publication/228717382_Developing_Algorithmic_Thinking_by_Inventing_and_Playing_Algorithms.

J. Johnson. *Designing with the Mind in Mind. Simple Guide to Understanding User Interface Design Guidelines*. Morgan Kaufmann, 2014. ISBN 978-0-12-407914-4. URL http://gen.lib.rus.ec/book/index.php?md5=e1e9a6bd56f79f8b81f653801c7ed01e.

I. Sommerville. *Software Engineering*. Pearson, 10th edition, 2016. ISBN 978-0-13-394303-0. URL http://gen.lib.rus.ec/book/index.php?md5=d8405230999cdb6d280b493fa5458a5b.