VISUALIZADOR Y ANALIZADOR DE TRAZAS DE EJECUCIÓN SOBRE MODELOS CONCURRENTES

VISUALIZATION AND ANALYSIS OF EXECUTION TRACES OF CONCURRENT MODELS

Daniel Herranz Gómez

DOBLE GRADO EN INGENIERÍA INFORMÁTICA Y MATEMÁTICAS
FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo de fin de grado en Ingeniería Informática

14 de junio de 2021

Directores:

Elvira Albert Albiol
Miguel Isabel Márquez

Resumen

La herramienta SYCO (*Systematic testing tool for Concurrent Objects*) es un analizador de modelos concurrentes que recibe un programa concurrente a ejecutar con unos determinados parámetros y provee a la salida la información de todos los posibles entrelazados del programa (trazas), junto con la secuencia de acciones efectuadas para llegar a cada uno de los estados finales. Sin embargo, la salida proporcionada por dicha herramienta no es visual, lo que complica la interpretación de los diferentes entrelazados posibles, más aún cuando el número de posibilidades crece de forma exponencial, en lo que se conoce como el *problema de la explosión de estados*.

El objetivo de este Trabajo de Final de Grado consiste en desarrollar un visualizador/analizador de las diferentes trazas que obtiene el programa SYCO de forma que sean fácilmente entendibles por un programador. Dado que la salida se puede interpretar inherentemente como un árbol, se mostrará un árbol con capacidad interactiva, con diferentes opciones para el usuario, tanto en relación a recorridos (se ofrecen diferentes opciones para navegar por el árbol) como en relación a información mostrada (se permite elegir la información de variables, de las propiedades de ejecución o de estado y el número de trazas a mostrar, e incluso se añade una leyenda para dirigir la atención a determinados nodos, según las necesidades del usuario).

En esta memoria se explicarán los pasos necesarios para instalar SYCO (capítulo 3) y se documentará todo el proceso de diseño del visualizador, desde su inicio con la decisión sobre el formato de imagen, el lenguaje usado (**SVG** y ***JavaScript***, respectivamente) y el formato de entrada requerido para el visualizador (capítulos 4 y 5), hasta la toma de decisiones sobre la utilización de diversos componentes, su posicionamiento y su estilo (capítulo 6).

Una vez realizada la implementación, hemos trabajado en que el visualizador funcione de forma conjunta con SYCO, como si fueran un mismo componente, para evitar la ejecución manual del visualizador tras obtener la salida de SYCO. Previo a la existencia de este trabajo, SYCO ya funcionaba con un programa con interfaz web llamado EasyInterface, que se apoya en el uso de servidores provistos por Apache. Así, la mejor opción ha sido utilizar este componente auxiliar para integrar ambos componentes satisfactoriamente, detallando el proceso en el capítulo 7.

La implementación del visualizador puede encontrarse en el repositorio GitHub dado por el enlace `https://github.com/dhg98/Execution-traces-visualizer`.

Palabras clave

Programa concurrente, SYCO, actor, tarea, visualizador/analizador, árbol, traza

Abstract

SYCO (*Systematic testing tool for Concurrent Objects*) is an analyzer of concurrent models that receives a concurrent program with specific parameters as input and provides as output all the possible interleavings of the program (traces), together with the sequence of actions taken to reach each of the final states. However, the output generated by this tool is not visual, what threatens its usability when the number of possibilities grows exponentially in what is known as *state explosion problem*.

The objective of this end of degree project consists in developing a visualizer/analyzer of all the different traces that SYCO finds in an understandable way by a software developer. Since the output can be interpreted inherently as a tree, we will show an interactive tree, with different options for the users, both related to its traversal (we offer different ways to traverse over the tree) and related to the information shown (we allow the user to choose variables information, execution and state properties and the number of traces to show. Furthermore, we have also added a legend that directs the attention to specific nodes, according to the needs of the user).

In this thesis, we will explain the necessary steps to install SYCO (chapter 3) and we will document all the design process to create the visualizer, from the very beginning, with the decision of the image format, the programming language (**SVG** and ***JavaScript***, respectively) and the input format required for it (chapters 4 and 5) to the decisions taken over the utilization of different components, its positioning and its style (chapter 6).

After finishing the implementation, we attempted to make the visualizer work as a unit with SYCO, in order to avoid manual execution of it after obtaining the output from SYCO. Prior to this current work, SYCO was already working with EasyInterface, a program with a web interface that relies on servers supplied by Apache. For that reason, the best option has been to use this auxiliary tool in order to integrate both systems. This process is detailed in chapter 7.

The implementation of the visualizer can be found in the GitHub repository given by the link `https://github.com/dhg98/Execution-traces-visualizer`.

Keywords

Concurrent program, SYCO, worker, task, visualizer/analyzer, tree, trace

Índice general

Capítulo 1

Introducción

1.1. Antecedentes y motivación

SYCO (*Systematic testing tool for Concurrent Objects*) [6, 7] es una herramienta que sirve para analizar programas concurrentes en base a unos parámetros de ejecución concretos y provee a la salida la información sobre todos los entrelazados factibles en el programa (trazas), y la secuencia de acciones o tareas necesarias para alcanzar cada uno de los estados finales. Estas acciones son ejecutadas por unos determinados actores, que podrían asemejarse a los objetos en el paradigma de programación orientada a objetos, y cuyos métodos invocados serían las tareas.

Esta herramienta es muy útil para poder analizar programas concurrentes, puesto que el testeo de los mismos es mucho más complicado que el de los programas secuenciales (ya que aparecen determinados riesgos como *deadlocks*, condiciones de carrera o *livelocks*), y a medida que el programa crece se hace muy complicado considerar todos los posibles entrelazados factibles en un programa concurrente de forma manual.

SYCO ofrece una interfaz web (dada por EasyInterface [19, 20]) para realizar el testeo de los programas concurrentes y permite visualizar el resultado de la ejecución del programa en la propia interfaz. Esta salida consiste en unos diagramas de secuencia que se generan para cada una de las ejecuciones obtenidas, de forma que sea sencillo apreciar la secuencia de tareas ejecutadas para llegar hasta el estado final. Un ejemplo de estas trazas generadas se encuentra en el listing 1.1 (cada uno de los campos que aparecen en el mismo se explican en el apartado 5.1.1).

```
|------'Time: 0, Object: main, Task: 0:main'
|------'Time: 1, Object: Host_5, Task: 1:sendIn'
|------'Time: 2, Object: Switch_2, Task: 3:switchHandlePkt'
|------'Time: 3, Object: Controller_1, Task: 4:ctrlHandleMsg'
|------'Time: 4, Object: Switch_2, Task: 7:sendOut'
|------'Time: 5, Object: Switch_2, Task: 5:switchHandleMsg'
```

```
|------'Time: 6, Object: Switch_3, Task: 6:switchHandleMsg'
|------'Time: 7, Object: Host_8, Task: 2:sendIn'
|------'Time: 8, Object: Switch_2, Task: 8:switchHandlePkt'
|------'Time: 9, Object: Controller_1, Task: 9:ctrlHandleMsg'
|------'Time: 10, Object: Switch_2, Task: 12:sendOut'
|------'Time: 11, Object: Switch_2, Task: 10:switchHandleMsg'
|------'Time: 12, Object: Switch_4, Task: 11:switchHandleMsg'
```

Listing 1.1: Ejemplo de traza

A medida que el número de ejecuciones y la longitud de las mismas aumenta, crece también la dificultad para comprender cada una de estas ejecuciones. Además, muchas de ellas son redudantes para el usuario, puesto que no aportan nueva información para entender el funcionamiento del programa. Este elevado número de redundancias dificulta la tarea de encontrar las ejecuciones relevantes. Pero no sólo estamos interesados en poder analizar los resultados del programa una vez se ha desarrollado al completo, sino que buscamos también disponer de un mecanismo que facilite la identificación de *bugs* de programación durante el desarrollo del mismo. Finalmente, SYCO solamente muestra información relativa a los estados finales (los valores finales de cada uno de los campos de los actores en la ejecución), pero, a menudo, existen estados intermedios donde se manifiestan estos *bugs*, que ayudan a entender donde se están cometiendo fallos en el programa.

Así pues, se hace necesario buscar alternativas que faciliten estas tareas, aportando una vista gráfica de todas las ejecuciones obtenidas y otra información adicional que pueda ser relevante en la visualización y el análisis del programa (como por ejemplo el estado de las variables de los diferentes actores que participan en el programa, entre otros).

1.2. Objetivos del trabajo

El objetivo de este Trabajo de Final de Grado es **desarrollar un visualizador/analizador de trazas de modelos concurrentes** basado en la salida de SYCO, e **integrarlo con este programa** para que funcionen como una única entidad **haciendo uso de EasyInterface**, una herramienta que dispone de interfaz web y con la que SYCO ya estaba integrada antes del inicio de este Trabajo.

Para ello, fue necesario comprender qué estructura estábamos tratando de representar al hablar de trazas, y pensar cuáles eran las características que deseábamos que tuviera nuestro visualizador y qué información íbamos a mostrar en dicho árbol. En base a esto hubo que decidir el formato de imagen utilizado para representarlo, y el lenguaje de programación elegido para generar la imagen. En el capítulo 4 veremos los requisitos que establecimos para el árbol, y los motivos que nos llevaron a elegir **SVG** como formato de imagen, y ***JavaScript*** como lenguaje de programación.

Adicionalmente, para poder representar árboles basados en la salida de SYCO, es necesario

instalar la herramienta para poder ejecutarla y así lograr comprender qué información proporciona, de forma que podamos definir el formato de entrada del visualizador, que coincidirá con la (nueva) salida de SYCO (adicional a la estándar ya existente antes de la creación de este visualizador). En el capítulo 3 daremos un breve tutorial sobre la instalación de la herramienta, y en el capítulo 5 veremos por qué se utiliza un **fichero JSON** y elegiremos la estructura del mismo.

Por último, una vez se ha tomado la decisión sobre el formato de imagen y el lenguaje de programación utilizado, y se ha definido el formato de entrada, es posible proceder con la implementación del visualizador, que se justificará en el capítulo 6, explicando el funcionamiento de todos los componentes introducidos, con su respectivo estilo y disposición en pantalla, y con la integración con SYCO usando EasyInterface, detallada en el capítulo 7. La apariencia final del visualizador que tiene en cuenta todas las consideraciones detalladas en el capítulo 6 puede encontrarse en la figura 1.1.

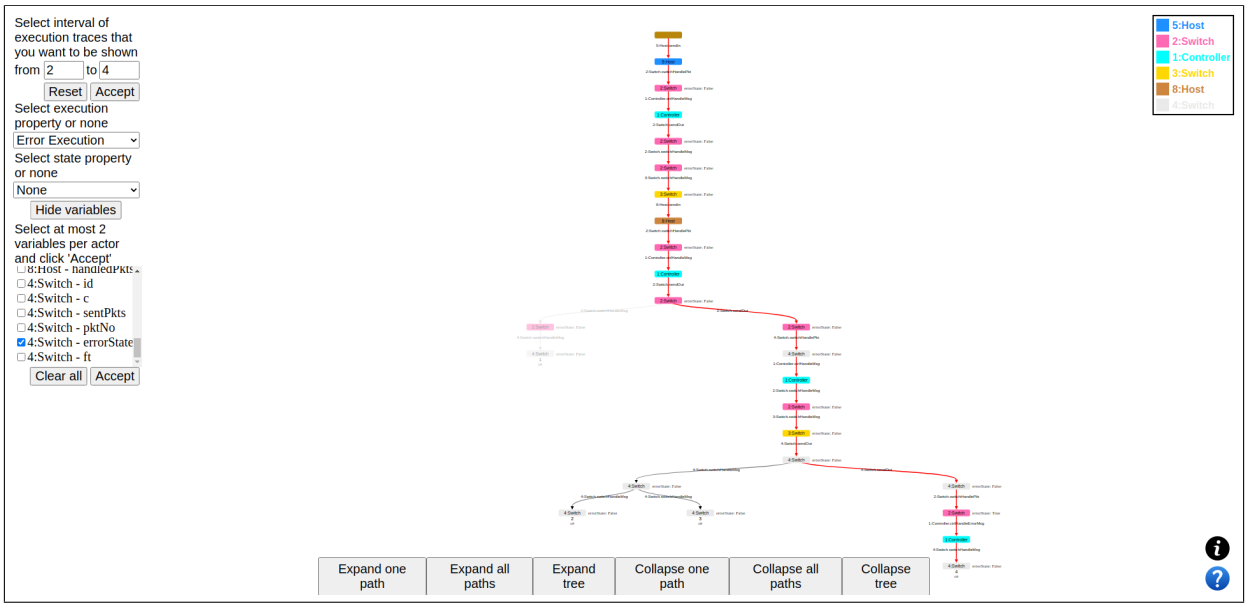

Figura 1.1: Apariencia del visualizador

Como puede apreciarse en dicha figura, resulta mucho más visual para analizar la ejecución de un programa el hecho de disponer de un árbol, en comparación con la información que disponemos en el listing 1.1.

1.3. Plan de trabajo

En primer lugar se instalará la herramienta en la que se va a centrar todo este Trabajo (SYCO), y se documentará el proceso. Este proceso de instalación nos permitirá descubrir

qué información tenemos disponible para decidir así qué vamos a mostrar en el árbol, y el formato en el que esta información va a venir dada.

Tras haber instalado el programa, y tener claras las características que debe tener nuestro visualizador/analizador, decidiremos cual va a ser el formato de imagen usado para visualizar el árbol de trazas, y qué lenguaje de programación vamos a usar para generar dicha imagen.

Una vez tengamos hecho esto, nos centraremos primero en la implementación del visualizador basado en la entrada previamente definida, cuyas características fundamentales son que podamos hacer *zoom* y que podamos mostrar información adicional en los nodos.

Finalizado lo anterior, estaremos en disposición de implementar parte del analizador, en el que permitiremos diferentes formas de explorar los árboles, como por ejemplo exploración camino a camino, o colapsado de un camino, mostrando únicamente su hoja, entre otros.

A continuación añadiremos la parte restante del analizador, que consiste en poder definir diferentes propiedades que cumplen los estados y las ejecuciones, de forma que sean visibles en el árbol a elección del usuario.

Tras completar la implementación del visualizador/analizador, lo integraremos con SYCO mediante el uso de EasyInterface.

Por último, añadiremos una serie de funcionalidades extras como son la leyenda para los actores o la selección del número de ejecuciones a mostrar en el árbol.

1.4. Material utilizado

La implementación del visualizador puede encontrarse en el repositorio de GitHub dado en el enlace `https://github.com/dhg98/Execution-traces-visualizer`. Este repositorio está estructurado de la siguiente manera:

- Un fichero HTML llamado *index.html*, que es el que va a ejecutar el navegador.
- Una carpeta *style* en la que se encuentran los ficheros CSS que definen el estilo de casi todo lo que se va a mostrar en el HTML.
- Una carpeta *examples* donde se encuentran unos ficheros en formato JSON con ejemplos de entrada para el visualizador.
- Una carpeta *tmp* donde EasyInterface va a escribir el fichero JSON obtenido de la ejecución de SYCO. Esta carpeta debe contar con los permisos 0777.
- Una carpeta *src* donde se encuentra todo el código *JavaScript* que controla el comportamiento del visualizador.
- Una carpeta *img* donde se encuentra una imagen con el estado del visualizador.

Capítulo 2

Introduction

2.1. History and motivation

SYCO (*Systematic texting tool for Concurrent Objects*) [6, 7] is a tool used to analyze concurrent programs based on specific execution parameters, and gives as output all possible interleavings in the program (traces), together with the sequence of actions or tasks that are needed to reach each of the final states. These actions are executed by specific workers that could look like objects in object oriented programming, and whose methods would be the tasks mentioned before.

This tool is very useful to analyze concurrent programs because the process of testing them is much more complicated than testing sequential programs (considering that there are lots of potential risks such as deadlocks, race conditions or livelocks), even more when the program grows because it becomes very difficult to consider all possible interleavings of a concurrent program manually.

SYCO offers a web interface provided by EasyInterface [19, 20] to test concurrent programs, and allows visualizing the result of the execution of the program in the same interface. This output consists of sequence diagrams that are generated for each of the obtained executions, so that it is easy to understand the sequence of actions that we have taken to reach the final state. One example of these sequence diagrams can be seen in listing 2.1 (all the fields that are in it will be explained in section 5.1.1).

```
|------'Time: 0, Object: main, Task: 0:main'
|------'Time: 1, Object: Host_5, Task: 1:sendIn'
|------'Time: 2, Object: Switch_2, Task: 3:switchHandlePkt'
|------'Time: 3, Object: Controller_1, Task: 4:ctrlHandleMsg'
|------'Time: 4, Object: Switch_2, Task: 7:sendOut'
|------'Time: 5, Object: Switch_2, Task: 5:switchHandleMsg'
|------'Time: 6, Object: Switch_3, Task: 6:switchHandleMsg'
|------'Time: 7, Object: Host_8, Task: 2:sendIn'
```

```
|------'Time: 8, Object: Switch_2, Task: 8:switchHandlePkt'
|------'Time: 9, Object: Controller_1, Task: 9:ctrlHandleMsg'
|------'Time: 10, Object: Switch_2, Task: 12:sendOut'
|------'Time: 11, Object: Switch_2, Task: 10:switchHandleMsg'
|------'Time: 12, Object: Switch_4, Task: 11:switchHandleMsg'
```

Listing 2.1: Example trace

As the the number of executions and the length of them increases, also the difficulty to understand all the different executions grows. In addition, most of them are superfluous for the user, because they don't provide new information to understand the functioning of the program. This high number of redundancies complicates the task of finding relevant executions. In fact, we are not only interested in analyzing the results of the program after it has been developed completely, but we also want to have a mechanism that makes the identification of bugs while developing the program easier. Finally, SYCO only shows information relative to final states (final values for each of the fields of the workers in the execution), but frequently there exist intermediate states where these bugs are more evident, which can help us understand where the bugs in the program are.

For that reason, it is necessary to look for different options that facilitate this task, creating a graphical view of all the obtained executions and other important information that could be relevant in the visualization and analysis of the program (such as variable states of all the different workers that take part in the program, among others).

2.2. Objectives of the project

The objective of this end of degree project is to **develop a visualizer/analyzer of traces of concurrent models** based on SYCO's output, and to **integrate it into this tool** in order for them to work as a unit **making use of EasyInterface**, a tool that has a web interface and with which SYCO was already working before the start of this project.

In order to do so, it was necessary to understand what the structure we want to represent when talking about traces is, to think about what all the features that we want our visualizer to have are, and what information we want to show in the tree structure. Because of this, we need to decide on the image format used to represent the tree and the programming language to generate the image. In chapter 4 we will see all the requirements that we set for the tree, and the reasons that have led us to choose **SVG** as image format and ***JavaScript*** as programming language.

In order to be able to represent trees based on SYCO's output, it has been necessary to install the tool so that we can understand what the information that is generated is, and we can define the input format of the visualizer, which matches with SYCO's (new) output (added to standard output, that existed before the creation of the visualizer). In chapter 3 we will give a small tutorial about the installation of SYCO, and in chapter 5 we will see

why we are going to use a **JSON file** and we will choose the structure of it.

Finally, once we have taken the decision over which image format and programming language to use, and we have defined the input format, it is possible to start with the implementation of the visualizer, which will be justified in chapter 6, explaining the functioning of all the different components that we have added, with their respective style and positioning, and with the integration of SYCO using EasyInterface, detailed in chapter 7. The final appearance of the visualizer that takes into account all the considerations made in chapter 6 can be found in figure 2.1.

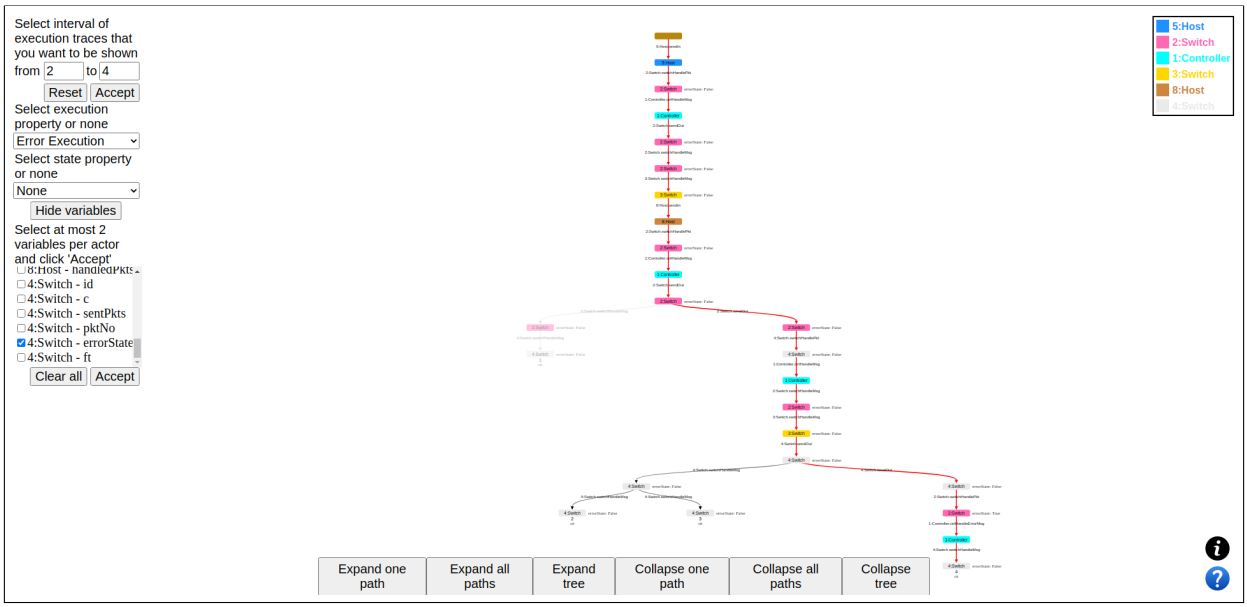

Figura 2.1: Visualizer appearance

As it can be seen in the figure, it is much more visual to have a tree in order to analyze the executions of a program, in comparison to the information we have in listing 2.1.

2.3. Work plan

First, we will install the tool on which this project is going to focus (SYCO), and we will write down all the process. This installation process will allow us to discover what is the information that we have available in order to decide what will be shown in the tree, and the format in which this information will be given.

After installing the program, and having clear all the features that our visualizer/analyzer should have, we will decide what the image format used to represent our traces tree will be, and which programming language will be used to generate that image.

Once that has been done, we will focus first on the implementation of the visualizer based on the input that has been previously defined, with zooming and additional information shown in nodes as basic features.

After that, we are prepared to start implementing the part of the visualizer that allows different ways to traverse past the tree, such as exploring trace by trace, or merging one trace, showing only its final state, among others.

Next, we will add the rest of the analyzer, that consists in allowing the definition of different properties that hold on states or executions, being shown in the tree at the request of the user.

After the implementation of the visualizer/analyzer is finished, we will integrate it with SYCO using EasyInterface.

Finally, we will add extra functionalities such as the legend for the workers or the selection of the number of executions we want to show in the tree.

2.4. Material used

The implementation of the visualizer can be found in the GitHub repository given by the link https://github.com/dhg98/Execution-traces-visualizer. This repository is structured as follows:

- A HTML file called *index.html*, which is executed by the browser.
- A *style* folder in which we have stored all the CSS files that define the style of almost every component that will be shown in the HTML.
- An *examples* folder where we will have some JSON files with input samples for the visualizer.
- A *tmp* folder where EasyInterface will write the JSON file obtained from the execution of SYCO. This folder should have 0777 permissions.
- A *src* folder where all the *JavaScript* code is stored, which is responsible for the behaviour of the visualizer.
- A *img* folder in which we have an image of the current state of the visualizer.

Capítulo 3

Instalación de Costa y Pet

Una de las etapas previas a la implementación fue la instalación de las herramientas de Costa [1, 2] y Pet [8, 9], puesto que en Pet se encuentra la herramienta SYCO de la cual vamos a obtener la entrada para el visualizador, y Pet utiliza Costa para funcionar.

Esta instalación es compleja, y consta de algunos pasos "delicados", por lo que va a documentarse en este capítulo para otras personas que necesiten utilizar esta herramienta. Mi intención es dar una breve guía de instalación sobre ambas herramientas, llamando la atención sobre los pasos críticos que no se deben obviar.

Los pasos que se van a anunciar fueron realizados en un ordenador con sistema operativo *Linux*, con una distribución Ubuntu. Previo a la instalación de Costa y Pet es necesario instalar algunos paquetes utilizando APT, como *subversion*, *graphviz*, *swi-prolog* o *default-jdk*, que son utilizados en la instalación de SYCO o en la ejecución del mismo.

Cabe destacar que SYCO utiliza *Python* para funcionar, por lo que también será necesario instalarlo. Sin embargo, no basta con instalar la última versión, puesto que SYCO no la utiliza (usa *Python 2* para funcionar).

3.1. Instalación de Costa

Los pasos que se indican más abajo asumen que se ha instalado previamente y con éxito la herramienta PPL [12], puesto que Costa hace uso de la misma. Los pasos para instalar esta herramienta no se indican aquí por estar bien documentados en http://costa.fdi.ucm.es/web/developer.php?subitem=3.

Paso 1: Hacer *checkout* del repositorio costa. Para ello es necesario ejecutar el comando

```
$ svn co https://costa.fdi.ucm.es/svn/Systems/costa/trunk ~/Systems/costa
```

Es importante que este comando se ejecute tal cual figura en esta guía. En general, un comando *svn checkout (svn co)* posee dos argumentos: la dirección del reposi-

torio y la dirección destino (en el ordenador) donde se va a copiar el repositorio. Si este segundo argumento no se proporciona, *svn* asumirá que la dirección destino es el directorio actual, haciendo el *checkout* en un directorio equivocado, lo cual provocará errores en la instalación de Pet.

Paso 2: Ir a la carpeta donde hemos hecho *checkout* del repositorio Costa. Para ello, es necesario ejecutar el comando

```
$ cd ~/Systems/costa
```

Paso 3: Hacer uso del fichero *Makefile* creado en dicha carpeta. Para ello, ejecutamos

```
$ make
```

Este proceso tardará unos minutos. Además, en mi caso, la instalación entró en un bucle infinito, por lo que fue necesario abortarla y volver a iniciarla, incluso más de una vez.

Paso 4: Ir a la carpeta *costabs*. Para ello, ejecutamos el comando

```
$ cd costabs/
```

Paso 5: Hacer uso de la regla *install* del fichero *Makefile*. Ejecutamos

```
$ make install
```

Paso 6: Ir a la carpeta *bin*. Para ello ejecutamos

```
$ cd bin/
```

Paso 7: Darle permisos de ejecución al fichero *generateProlog*. Para ello, ejecutamos

```
$ chmod +x generateProlog
```

Una vez se han realizado todos estos pasos correctamente, Costa debería funcionar correctamente. Para comprobarlo, véase la sección 3.3.

3.2. Instalación de Pet

Los pasos a seguir son los siguientes:

Paso 1: Hacer *checkout* del repositorio pet. Para ello es necesario ejecutar el comando

```
$ svn co https://costa.fdi.ucm.es/svn/Systems/pet/trunk ~/Systems/pet
```

La carpeta destino es *~/Systems/pet*. De nuevo, esta carpeta no debe ser cambiada, por el mismo motivo que se dio en la sección 3.1.

Paso 2: Ir a la carpeta donde hemos obtenido el repositorio pet, ejecutando

```
$ cd ~/Systems/pet
```

Paso 3: Hacer uso del fichero *Makefile* creado en dicha carpeta. Para ello, ejecutamos

```
$ make
```

Una vez se han ejecutado estos pasos, puede darse la instalación por finalizada. De nuevo, para comprobarlo, véase la sección 3.3.

3.3. Comprobación de funcionamiento

Una vez se ha instalado tanto Costa como Pet, es necesario asegurarse de que su instalación se ha completado con éxito, y que no ha fallado ningún paso. Esta comprobación es ineludible, puesto que sin Pet funcionando (más concretamente SYCO), no podremos visualizar la salida que pretendemos generar. Mediante la ejecución de los siguientes comandos podremos asegurarnos de que la instalación ha sido completada con éxito.

Paso 1: Ir a la carpeta donde se ha instalado *costa*. Para ello, ejecutamos el comando

```
$ cd ~/Systems/costa
```

Paso 2: Comprobar que Costa funciona correctamente, ejecutando

```
$ costa -c complexityClasses/Linear -M
```

La salida debería incluir un fragmento con una apariencia similar a la siguiente:

```
FINAL RESULTS FOR complexityClasses/Linear_<init>()V

Total analysis time: 0.017 secs

The Upper Bound for 'complexityClasses/Linear_<init>()V'(this) is
3+c(java/lang/Object_<init>()V) instructions

Method 'complexityClasses/Linear_<init>()V' terminates?: yes

Analysis failed: error(jbcr_user_classes,jpath_resolve_class_name/2,class
_file_not_found,[file=java/lang/Object])
```

Paso 3: Ir a la carpeta donde se encuentra SYCO. Para ello, ejecutamos el comando

```
$ cd ~/Systems/pet/src/interfaces/shell
```

Paso 4: Comprobar que SYCO funciona correctamente, ejecutando

```
$ ./syco ~/Systems/pet/examples/apet/misc/abdulla/floating_read.abs -obj_
sel_policy nondet -sched nondet
```

La salida debería incluir un fragmento con una apariencia similar a la siguiente:

```
CLP generated in 16.00 ms. (stored in /tmp/pet/main.pl)
6 executions in 15 ms.

Total time: 15 ms.
Context-Sensitive DPOR has explored the program ['/home/*****/Systems/pet
/examples/apet/misc/abdulla/floating_read.abs'] in 15 ms.
   +  Number of complete executions: 6
   +  Number of states explored: 11
```

donde ***** hacen referencia al nombre de usuario que está realizando la instalación. Adicionalmente, este último paso genera una serie de diagramas en formato SVG que pueden visualizarse yendo a la carpeta */tmp/pet*, y ejecutando el comando

```
$ xdg-open exec_tree.svg
```

En este caso, obtendremos un gráfico en formato SVG como el siguiente.

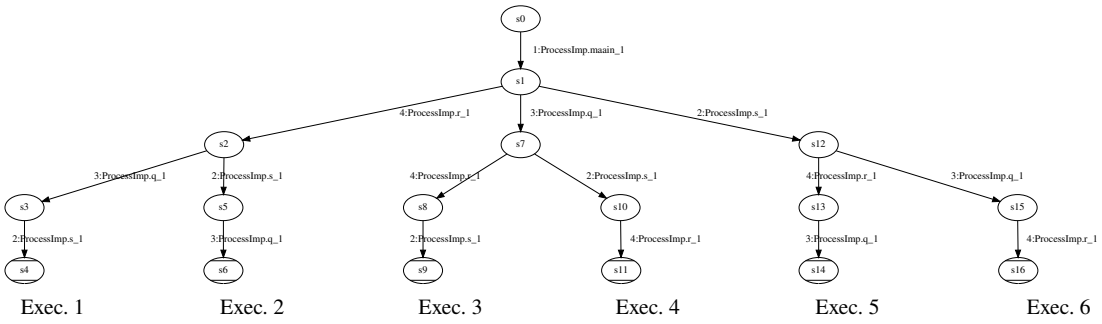

Figura 3.1: Ejemplo de árbol de ejecución

Es posible que la imagen aparezca cortada, dependiendo de si el archivo SVG tiene un parámetro de escala en el grupo *g* con id asociado *graph0* fijado a un tamaño mayor que 1. Aunque éste sea el caso, la instalación se ha completado con éxito. Si aún así deseásemos visualizar el árbol al completo, bastaría con editar el factor de escala a 1, en lugar de a su valor por defecto (1.36) como se aprecia en el siguiente ejemplo.

```
transform="scale(1.36 1.36) rotate(0) translate(4 493)"
```

Capítulo 4

Estado del arte en técnicas de visualización

Este trabajo consiste en diseñar un visualizador de modelos concurrentes, por lo que es necesario determinar previamente qué tecnología va a ser utilizada para representar la salida del programa (formato de imagen), así como el lenguaje de programación y las librerías utilizadas.

4.1. Comparación de formatos de imagen

Existen diferentes formatos para representar imágenes. Entre los más importantes y conocidos destacan los **formatos ráster** (también conocidos como mapas de bits) y los **formatos vectoriales**.

Una **imagen ráster** es un fichero de datos representado por una matriz bidimensional de píxeles que puede visualizarse en un monitor. Viene definida por su altura y anchura (en píxeles) y profundidad de color (en bits por píxel).

La principal desventaja de este tipo de imágenes es que **no escala correctamente** en el caso en el que deseemos ampliar su tamaño, puesto que al ser números fijos, el número de píxeles de la imagen seguirá siendo el mismo a pesar de que la imagen sea más grande, lo que provocará una pérdida de resolución. Es por eso que este tipo de imágenes no pueden ser utilizadas para el propósito que perseguimos, puesto que las estructuras de árbol que deseamos representar van a tener un tamaño arbitrario (tanto en altura del árbol como número de hijos por nodo), lo que va a hacer que necesitemos ampliar la imagen para poder leer el contenido de los nodos, con su consiguiente pérdida de calidad.

Además, presentan otra desventaja muy clara, y es que no es posible incorporar transiciones a este tipo de imágenes, por lo que no podríamos hacer expansión y colapsado de nodos, algo vital en este visualizador, como veremos más adelante.

Una **imagen vectorial** es un tipo de imagen definida en el plano formada por objetos geométricos lineales y curvos, basados en fórmulas matemáticas. Aunque a la hora de mostrar este tipo de imágenes en pantalla los ordenadores traduzcan este tipo de imágenes en imágenes ráster, el hecho de que este tipo de imágenes estén basadas en fórmulas matemáticas hace que sea **posible modificar su tamaño sin perder calidad**, lo cual es una gran ventaja a la hora de representar figuras moderadamente grandes en las que se busque también poder leer fragmentos de texto de tamaño pequeño.

Además, este tipo de imágenes son más ligeras que los formatos ráster, por lo que su procesamiento será mucho más rápido (véase [34]).

En la siguiente figura se puede apreciar una recreación de la diferencia de calidad entre los dos tipos de imágenes introducidos anteriormente:

Figura 4.1: Diferencia entre imagen ráster e imagen vectorial al hacer *zoom*

Vista esta diferencia, para representar los árboles del programa se utilizarán imágenes con formato vectorial. Existen diferentes formatos de imagen de tipo vectorial, entre los que destacan CGM o SVG. Sin duda alguna, el formato más utilizado y conocido de este tipo es el de SVG.

Entre las múltiples ventajas del formato **SVG**, destacan las siguientes:

- El comportamiento de las imágenes en este formato está definido a través de XML. Así, las imágenes SVG pueden ser indexadas, comprimidas y **programadas**.
- Es una de las recomendaciones del W3C (*World Wide Web Consortium*) desde 2001, lo que hace que sea un formato **soportado por todos los navegadores Web**.
- Es interactivo, en el sentido de que **permite incluir *scripts*** para generar caminos dinámicos. Así, permite generar diferentes acciones en base a eventos, como podría ser hacer *click*, posicionarse encima de un objeto concreto, hacer *zoom*, etc. e incluso combinarlo con el pulsado de teclas.

Las características que buscamos en el visualizador de trazas se detallan a continuación. Podemos ver que el formato de imagen SVG cubre todas esas características que buscamos, y por lo tanto es un formato idóneo para representar las trazas de los modelos concurrentes, y es el que utilizaremos en la implementación. Entre los requisitos definidos para los formatos de imagen, destacan:

- Hacer *zoom* sobre diferentes partes del árbol: esta característica la tienen todos los formatos de imagen, pero además SVG, al ser vectorial, no distorsiona la imagen, por lo que los textos presentes en el árbol serán legibles, independientemente del tamaño de letra que se escoja (aunque evidentemente cuanto más pequeña sea la letra, mayor será el zoom que tendremos que hacer para poder visualizar la información).

- Pulsar o situarnos sobre un nodo para ver la información del estado asociado al nodo: SVG es dinámico y permite expandir y contraer objetos. Así, permite expandir la información sobre un nodo, ya sea al clicar o posicionarnos sobre el nodo.

- Navegar por el árbol: SVG es dinámico y permite expandir y contraer objetos, como podrían ser los nodos. Pero además, por el hecho de ser programable, admite la utilización de distintos recorridos que modifiquen la estructura del árbol a mostrar. Distintos ejemplos pueden ser:

  - Mostrar todos los hijos desde un nodo.
  - Expandir un único camino desde un nodo dado hasta una hoja en su subárbol (si hubiera alguna sin expandir).
  - Expandir todos los caminos desde un nodo dado hasta todas las hojas en su subárbol.
  - Colapsar todos los hijos de un nodo.
  - Colapsar un camino desde un nodo dado hasta una hoja en su subárbol (si hubiera alguna ya expandida).
  - Colapsar todos los caminos desde un nodo dado hasta las hojas en su subárbol.
  - Expandir todo el árbol.
  - Colapsar todo el árbol.
  - Colapsar un camino desde un nodo en dicho nodo y la hoja, de manera que se oculten todos los nodos intermedios y la información de las aristas se compacte en la visible.

- Colorear trazas o nodos que cumplan una determinada propiedad: esta característica la tienen todos los formatos de imagen.

4.2. Comparación de lenguajes y librerías

Tras seleccionar el formato de imagen que se va a utilizar para representar la información, comparando las diversas opciones disponibles, es necesario determinar el lenguaje de programación más adecuado para generar este tipo de imágenes, y la librería utilizada, en caso de querer utilizar alguna.

Hay que tener en cuenta diferentes criterios a la hora de elegir el lenguaje de programación adecuado:

- **Existencia de librerías** que faciliten la labor: aunque casi cualquier lenguaje permite la creación de imágenes SVG, no todos disponen de librerías gráficas en las que sea sencillo generar este tipo de imágenes, por lo que los lenguajes que no dispongan de este tipo de herramientas, o estén poco difundidas son descartados, ya que las librerías ya implementadas pueden reducir el tiempo de implementación del visualizador.
- **Conocimiento del lenguaje**: dado que el tiempo es limitado, cuanto más conocimiento previo se tenga del lenguaje, más sencilla será la creación de las imágenes SVG, puesto que surgirán menos problemas de programación. Aún así, este factor no es determinante.
- **Existencia de una gran comunidad**: de nuevo, el tiempo es limitado, por lo que cualquier problema de implementación que no se pueda resolver fácilmente puede comprometer la entrega a tiempo. Por lo tanto, es muy relevante utilizar un lenguaje de programación que sea muy utilizado, de forma que, en caso de ser necesario, se pueda acceder a ejemplos y hacer preguntas en foros sobre los problemas que aparezcan. Más aún, el uso de una librería que este muy extendida puede favorecer la búsqueda de ejemplos y resolución de dudas que no estén relacionados con el lenguaje en sí sino con la librería.
- **Creación de elementos de interacción con el usuario**: como se ha indicado en la sección anterior, una de las acciones que deseamos permitir es la de analizar las trazas de ejecución. Para ello, debemos permitir seleccionar qué propiedades queremos que se muestren en el árbol, y para ello debemos añadir ciertos elementos para que el usuario interactúe con ellos (botones, desplegables, etc.). Por ello, es deseable que el lenguaje elegido permita añadir objetos de este tipo de forma sencilla.

Con todas estos factores en cuenta, busqué diferentes lenguajes que favorecieran la creación de formatos de imagen SVG, e incluso busqué librerías que ayudaran en la creación de árboles con las características anunciadas anteriormente. Entre mis hallazgos, destacan:

- *OpenEye* para *Python*. Para usar esta librería era necesario disponer de una licencia, con lo cual fue descartada automáticamente. Sin embargo, parece que disponía de todas las características necesarias.

- *Plotly* para *Python*. Librería muy potente, con una gran comunidad detrás y muy utilizada a nivel empresarial, sobre todo para *DataScience*.
- *Raphael* para *JavaScript*. Librería sencilla de utilizar y con una cierta comunidad detrás. Permite crear formatos de imagen SVG, pero tiene funcionalidades limitadas, lo que podía comprometer el proyecto en caso de querer realizar algo demasiado complejo.
- *D3* para *JavaScript*. Librería muy potente con una gran comunidad detrás (alrededor de 40.000 preguntas nuevas en *Stack Overflow*), que incluso se utiliza a nivel de empresa. Dispone de todas las herramientas necesarias para dibujar árboles, aunque la curva de aprendizaje es más larga en comparación con la de otras librerías.
- *GoJS* para *JavaScript*. Librería que permite crear todo tipo de diagramas, incluido árboles. Sin embargo, parece que no es demasiado utilizada, puesto que en *StackOverflow* hay pocas preguntas realizadas sobre esta librería, y cerca de un tercio de ellas está todavía sin responder.

Teniendo en cuenta la gran variedad de posibilidades y sus características, la librería utilizada fue *D3* en *JavaScript*, puesto que antes de empezar con la implementación fui capaz de encontrar una gran cantidad de ejemplos similares a lo que deseábamos hacer, lo que aseguraba que las funcionalidades que buscábamos incorporar al visualizador eran factibles sin necesidad de hacer pruebas anteriormente. Además, como ya se ha dicho anteriormente tiene una gran comunidad, y tiene licencia *BSD*, con lo cual su uso no está muy restringido.

Concretamente, la librería facilita:

- La expansión y el colapsado de los descendientes de un nodo.
- Pulsar o situarnos sobre un nodo para ver la información del estado asociado a él.
- La actualización de la imagen sin necesidad de regenerarla completamente, sino solo modificando lo mínimo. Esto permite que la generación de imágenes sea rápida, sobre todo cuando el árbol crece.

No sólo contamos con las posibilidades que nos ofrece la librería *D3*, sino que también disponemos de grandes facilidades con la utilización de *JavaScript*, como es la integración nativa con *HTML* y *CSS*. *HTML* facilitará en gran medida la interacción con el usuario, ya que podemos hacer uso de diferentes elementos para generar cambios en el árbol, y *CSS* ayudará en la definición del estilo, al compactar la definición de estilo de los objetos utilizados (tamaño de la letra, colores, etc.), permitiendo así que sea más sencillo cambiar estos valores.

Además, el código *JavaScript* es ejecutable en todos los navegadores modernos, lo que evita que los potenciales usuarios de la herramienta deban instalar apenas nada para poder comenzar a visionar y a interactuar con los árboles.

La mayor desventaja de utilizar este lenguaje es que antes de comenzar la implementación no disponía de conocimientos sobre él, pero es un lenguaje que es bastante utilizado, con lo que hay mucha documentación que se puede consultar y utilizar.

Capítulo 5

Especificación de la entrada del visualizador

El visualizador desarrollado en este trabajo va a tomar la salida del sistema SYCO y la va a usar como entrada para dibujar el árbol. Por este motivo, es necesario especificar cuál va a ser la entrada del visualizador. Para ello, se van a tener en cuenta dos factores, como es la forma que tiene SYCO de explorar los estados de ejecución, y también la entrada que exige la librería *hierarchy* de *D3* para generar el árbol. Estas dos especificaciones no van a coincidir, lo cual obliga a generar un algoritmo que transforme la salida de SYCO en la entrada de *D3* de forma transparente al usuario.

5.1. Especificación de la entrada

Dado que el lenguaje utilizado va a ser *JavaScript*, es deseable que el fichero de entrada del programa venga dado en un formato que sea legible sin necesidad de hacer ninguna transformación por parte del programa. Entre las diversas opciones existentes, destacan los formatos JSON, YAML o XML. Todos estos distintos formatos tienen ventajas y desventajas. Sin embargo, puesto que estoy familiarizado con la utilización de ficheros JSON (*JavaScript Object Notation*), decidí usar este formato.

Al contrario de lo que pudiera parecer por su nombre, los ficheros JSON no sólo se usan en la programación con *JavaScript*, sino que están muy extendidos en el mundo de la informática como formato de fichero plano. Por ejemplo, Python dispone de una librería para parsearlos.

5.1.1. Salida de SYCO

El sistema SYCO, proporciona información sobre las diversas trazas (ejecuciones de un programa) exploradas, en un formato plano. Así, para cada traza, proporciona una lista de

todos los actores que se han ejecutado, junto con la tarea que han ejecutado para alcanzar el siguiente estado. Este formato se repite hasta que se alcanza una hoja. Por ejemplo, una posible traza podría tener la apariencia del listing 1.1.

Podemos apreciar en dicho listing que contamos con 13 elementos de tiempo que aparecen en orden de ejecución. Estos elementos tienen asociado el objeto que ha ejecutado, y la tarea que ejecutan. Cada elemento de tiempo que figura en la lista será un nodo, llevando asociado también el objeto que ha ejecutado en ese instante de tiempo, mientras que la tarea que se ejecuta será la etiqueta de la arista que une dicho nodo con su padre.

Como es de esperar, las ejecuciones de cada programa se pueden agrupar en forma de árbol, y todas tienen nodo común, como mínimo, al que denominaremos raíz. Los restantes nodos serán nodos internos, a excepción del último de cada ejecución, que se conocerá por nodo hoja. En el caso del listing 1.1, el nodo hoja de dicha ejecución sería el que tiene como objeto a *Switch_4*, y el nodo raíz será el que tiene como objeto a *main*.

Además de esta información, SYCO proporciona información general sobre la exploración del árbol, como por ejemplo el número de ejecuciones encontradas, el número de nodos explorados, el tiempo que se ha necesitado para explorar el árbol, etc. El listing 5.1 muestra un ejemplo de esta información general.

```
Independence constraints generated in 1048 ms.
Number of executions: 4
Total time: 93
Total number of states explored during 4 executions: 25
Total number of tasks executed during 4 executions: 20
```

Listing 5.1: Ejemplo de información general

SYCO también puede proporcionar información adicional en los nodos, como:

- Información del estado de las variables para ese objeto en ese instante de tiempo.
- Información de ciertos conjuntos que dependen del algoritmo de *model-checking* (véase [31]) utilizado para la exploración de las trazas del programa concurrente. Por ejemplo, el *backtrack set* o el *sleep set* (véase [17]).

5.1.2. Entrada del visualizador

Dada la información de la sección 5.1.1, hemos de definir una estructura fija para la entrada del visualizador en formato JSON (que será generado automáticamente por SYCO y almacenado en la carpeta */tmp/pet*).

Adicionalmente a toda esa información, deseamos poder analizar trazas y estados de los programas concurrentes. Actualmente, SYCO no proporciona esta información en su salida

estándar, pero para que el visualizador permita tal función, deberá proporcionarse información en el fichero JSON sobre las propiedades que cumple cada uno de los nodos del árbol y cada una de las ejecuciones o trazas que componen el árbol.

Teniendo toda esta información en cuenta, la estructura del fichero JSON elegida fue la siguiente:

```
{
    "?sets": [
        {
            "nodeName": "node 1",
            "type": "backtrackSet" or "sleepSet",
            "content": [
                "element 1₁",...,"element N₁"
            ]
        },
        ⋮
        {
            "nodeName": "node M",
            "type": "backtrackSet" or "sleepSet",
            "content": [
                "element 1ₘ",...,"element Nₘ"
            ]
        }
    ],
    "?states": [
        {
            "nodeName": "node 1",
            "?vars": {
                "var1₁": "value Var1₁",
                ⋮
                "varN₁": "value VarN₁"
            },
            "?properties": [
                "property1₁",...,"propertyM₁"
            ],
            "?extra": "Extra information of the node 1"
        },
        ⋮
        {
            "nodeName": "node K",
            "?vars": {
```

```
                "var1_K": "value Var1_K",
                ⋮
                "varN_K": "value VarN_K"
            },
            "?properties": [
                "property1_K",...,"propertyM_K"
            ],
            "?extra": "Extra information of the node K"
        }
    ],
    "edges": [
        {
            "parent": "node i",
            "child": "node j",
            "label": "Task in the form actor.task that needs to
                be executed from the parent to arrive to the
               child",
            "?extra": "Extra information of the link that
               connects node i to node j"
        }
    ],
    "executions": [
        {
            "executionName": "Execution 1",
            "?executionLabel": "label Execution 1",
            "executionLeaf": "leafNode name of Execution 1",
            "?properties": [
                "property1_1",...,"propertyM_1"
            ]
        },
        ⋮
        {
            "executionName": "Execution L",
            "?executionLabel": "label Execution L",
            "executionLeaf": "leafNode name of Execution L",
            "?properties": [
                "property1_L",...,"propertyM_L"
            ]
        }
    ],
    "?info": "HTML formatted text with similar content to
       Listing 5.1"
```

```
74 }
```

Listing 5.2: Especificación de entrada al visualizador

Pasamos ahora a explicar cada uno de los campos presentes en el fichero JSON.

El campo ***sets*** consiste en una lista opcional de objetos. En caso de que no sea vacía, cada objeto de la lista consistirá en: campo *nodeName* que identifica unívocamente al nodo en cuestión; *type*, que identifica el tipo de conjunto que estamos tratando, ya sea *backtrack set* o *sleep set* (o cualquier otro que se quiera añadir en el futuro); y una lista no vacía *content* donde tendremos las diferentes cadenas de caracteres que formarán parte del conjunto del nodo en cuestión.

El campo ***states*** consiste en una lista opcional de objetos. En caso de que no sea vacía, cada objeto de la lista consistirá en: campo *nodeName*, que identifica unívocamente al nodo; *vars*, que es un objeto del tipo clave-valor, donde se indica, para cada identificador de variable asociado a dicho nodo, su valor en ese instante de tiempo; y *properties*, una lista que nos indica qué propiedades de estado cumple dicho nodo. Estos dos últimos objetos son opcionales, pero para que un objeto en *states* aporte información útil para el visualizador deberá contener al menos uno de los dos (de lo contrario será ignorado). Por último, incorporamos un campo *extra* que es una cadena de caracteres opcional que contiene la información adicional que deseamos mostrar.

El campo ***edges*** consiste en una lista de objetos (aristas). Cada objeto de la lista consiste en: campo *parent*, un identificador único de nodo, que actuará como padre del nodo dado en *child*. Estos dos campos no pueden coincidir, pero además deberán tener una estructura coherente (en el sentido de que no puede haber ciclos, solamente puede haber una raíz, etc.). Cada arista tiene un campo *label*, que indica el texto que aparecerá en la línea que una ambos nodos. En el listing 1.1, eso sería lo que está etiquetado con el nombre de *Task*. Finalmente, cada objeto de *edges* tiene un atributo *extra* que permite agregar información adicional para mostrarla en la arista.

El campo ***executions*** consiste en una lista de objetos. Cada objeto de la lista proporciona una ejecución (un camino de la raíz a una hoja). Este camino viene identificado unívocamente por la hoja (*executionLeaf*). Adicionalmente, se proporciona una etiqueta que nombra a la ejecución (*executionName*) y una etiqueta opcional (*executionLabel*) para dar algo más de información sobre dicha ejecución (por ejemplo, se puede indicar si esa ejecución ha llevado a un *Deadlock*, o cualquier otra información relevante que SYCO considere). Por último, de forma opcional se puede proporcionar una lista de propiedades de ejecución (*properties*) que cumple la ejecución en cuestión. Estas propiedades las cumple toda la ejecución, no únicamente el nodo hoja que figura en dicho objeto.

El campo ***info*** es opcional, y consiste en una cadena de caracteres en formato HTML que da información general sobre la exploración del árbol. Un ejemplo de la información que podríamos encontrar en este campo sería el listing 5.1.

La representación elegida no es casual, y viene propiciada por diversas decisiones de diseño,

como las siguientes:

- El programa SYCO es *state less*, lo que significa que va usando el estado a medida que va computando los diferentes caminos y lo borra cuando termina de explorar dicho nodo, para evitar el gasto masivo de memoria. Por este motivo, es recomendable que el estado de cada nodo vaya separado de los conjuntos que tiene asociados dicho nodo, puesto que el momento de cálculo de estos objetos se realiza en diferentes momentos, lo cual implicaría un elevado gasto de tiempo innecesario en la generación del fichero en caso de desear generarlos conjuntamente. Esto justifica el por qué tenemos un campo *sets* y otro *states*, cuando aparentemente podrían estar fusionados en uno solo.
- Un nodo puede tener varios tipos de conjuntos asociados (*backtrack set*, *sleep set*, etc.), pero todos tienen la misma estructura: un nodo asociado y el contenido del conjunto que representan. Por lo tanto, es innecesario separar cada uno de estos conjuntos y generar un campo diferente. En lugar de eso, añadimos un campo "*type*" que es el encargado de indicar el tipo de conjunto con el que se está tratando. A nivel conceptual, podría decirse que este campo es un enumerado con los diferentes tipos de conjuntos que podemos reconocer. De esta forma, unificamos criterios.
- Una ejecución es un camino desde la raíz hasta una de las hojas del árbol, sin repetir. De esta forma, para dar cada una de las ejecuciones no es necesario aportar la sucesión de aristas seguidas, puesto que solamente existe un camino desde cada hoja a la raíz. Así, nos basta con disponer del identificador del nodo hoja para identificar cada ejecución de forma unívoca, ya que, al disponer de las aristas del árbol (campo *edges*), podemos generar el camino hasta la raíz.
- Con la intención de hacer el visualizador más general, se incorpora un campo *extra* tanto en los nodos como en las aristas. Su objetivo es poder mostrar la información que el usuario desee libremente tanto en nodos como en enlaces.

Como ya se ha indicado anteriormente, aunque ésta sea la entrada del visualizador, esta entrada no puede ser usada directamente para generar el árbol que el usuario visualizará. Por ello, será necesario crear un algoritmo que transforme la entrada dada en la sección 5.1.2 en la que daremos en la sección 5.1.3. Este algoritmo puede encontrarse en la sección 5.2.

En el apéndice C puede encontrarse una breve explicación del proceso seguido para generar el fichero JSON con la estructura dada en el listing 5.2 por parte de SYCO.

5.1.3. Entrada de *D3 hierarchy*

La entrada de la librería *D3 hierarchy* es completamente diferente de la que recibe el visualizador. Ésta puede ser proporcionada como un objeto con estructura JSON anidada como el siguiente

```
{
    "name": "Node 1",
    "executionProperties": [
        "property1_1",...,"propertyM_1"
    ],
    "stateProperties": [
        "property1_1",...,"propertyM'_1"
    ],
    "vars": {
        "var1_1": "value Var1_1",
        ⋮
        "varN_1": "value VarN_1"
    },
    "backtrackSet": [
        "element 1_1",...,"element M''_1"
    ],
    "sleepSet": [
        "element 1_1",...,"element M'''_1"
    ],
    "executionNumbers": [1, 2],
    "children": [
        {
            "name": "Node 2",
            "link": "Label for link that connects Node 1 with
               Node 2"
            "executionProperties": [
                "property1_2",...,"propertyM_2"
            ],
            "stateProperties": [
                "property1_2",...,"propertyM'_2"
            ],
            "vars": {
                "var1_2": "value Var1_2",
                ⋮
                "varN_2": "value VarN_2"
            },
            "backtrackSet": [
                "element 1_2",...,"element M''_2"
            ],
            "sleepSet": [
                "element 1_2",...,"element M'''_2"
```

```
            ],
            "executionNumbers": [1],
            "children": [
                {
                    "name": "Node 4",
                    "link": "Label for link that connects Node
                       2 with Node 4"
                    "executionProperties": [
                        "property1_4",...,"propertyM_4"
                    ],
                    "stateProperties": [
                        "property1_4",...,"propertyM'_4"
                    ],
                    "vars": {
                        "var1_4": "value Var1_4",
                        ⋮
                        "varN_4": "value VarN_4"
                    },
                    "backtrackSet": [
                        "element 1_4",...,"element M''_4"
                    ],
                    "sleepSet": [
                        "element 1_4",...,"element M'''_4"
                    ],
                    "executionName": "Execution 1",
                    "executionLabel": "label Execution 1",
                    "executionNumbers": [1]
                }
            ]
        },
        {
            "name": "Node 3",
            "link": "Label for link that connects Node 1 with
              Node 3"
            "executionProperties": [
                "property1_3",...,"propertyM_3"
            ],
            "stateProperties": [
                "property1_3",...,"propertyM'_3"
            ],
            "vars": {
                "var1_3": "value Var1_3",
```

```
                    ⋮
                    "varN3": "value VarN3"
                },
                "backtrackSet": [
                    "element 13",...,"element M3''"
                ],
                "sleepSet": [
                    "element 13",...,"element M3'''"
                ],
                "executionName": "Execution 2",
                "executionLabel": "label Execution 2",
                "executionNumbers": [2]
            }
        ]
}
```

Listing 5.3: Especificación de la entrada de *D3 hierarchy*

Podemos apreciar que los campos que utilizamos son análogos a los utilizados anteriormente, excepto el referente a las propiedades de ejecución: una propiedad de ejecución aparecerá en un nodo si dicho nodo forma parte de una ejecución que tiene esa propiedad, es decir, si alguna hoja del subárbol que tiene a ese nodo como raíz tiene en el campo *executions* del listing 5.2 dicha propiedad.

También tenemos un campo *executionNumbers* en cada nodo, que nos indica qué ejecuciones pasan por dicho nodo, habiendo tomado los números de ejecución de izquierda a derecha en el árbol.

5.2. Transformación de la entrada

Como ya se ha comentado a lo largo de este capítulo, y se ha podido apreciar en los apartados 5.1.2 y 5.1.3, las representaciones utilizadas son muy diferentes entre sí. Por lo tanto, es necesario transformarlas para poder representar el árbol.

La idea para transformarlas es relativamente sencilla, y se va a explicar a continuación, teniendo en cuenta que la intención es transformar una lista plana de aristas en una lista anidada de nodos que mantenga esta relación padre-hijo dada en la lista de aristas.

En primer lugar, hay que tener en cuenta que los identificadores únicos de los nodos no tienen por qué tener la forma que hemos dado en las secciones anteriores (es decir, el identificador 0 está asociado a la raíz, y así sucesivamente con los hijos), por lo que en un principio desconocemos cuál va a ser la única raíz del árbol. En caso de que conozcamos cuál es dicha raíz (en 5.2.1 se explica detalladamente el procedimiento para encontrarla), utilizando la

estructura *sets* dada en el listing 5.2, podemos comenzar con la raíz para construir un árbol en el que tengamos como únicos nodos a la raíz y a sus descendientes directos, utilizando las aristas en las que dicho nodo está como padre. Haciendo el mismo proceso con los hijos directos, podemos generar los sucesores de segundo nivel de la raíz. Continuando de esta manera, generaríamos el árbol completo. Este proceso se detalla en el apartado 5.2.2.

Pero no sólo deseamos crear esta estructura de árbol, sino que después tenemos que recorrer las restantes estructuras (*sets*, *states* y *executions*) rellenando los restantes datos que faltan en los diversos nodos. Sin embargo, no resulta admisible que, para cada elemento de estas tres estructuras, tengamos que buscar el nodo al que pertenece en el árbol y agregarle dicho atributo, puesto que sería muy ineficiente. Para ello, una posible solución es guardar, para cada identificador (único) de nodo, el subárbol que tiene a dicho nodo como raíz. De esta forma, cuando encontremos un atributo asociado a un nodo simplemente tendremos que acceder a esta estructura y añadir el atributo. De esta forma, tendremos un coste de $O(1)$ para añadir cada atributo, en lugar de un coste de $O(n)$ (donde n es el número de nodos del árbol) con la otra aproximación (véase 5.2.3).

5.2.1. Encontrando la raíz

En base a lo explicado anteriormente, nuestro primer interés es, por tanto, encontrar la raíz del árbol. Pero además podemos preprocesar ligeramente la información mientras encontramos la raíz. Así, también generaremos una estructura donde, para cada identificador de nodo guardemos una lista con todas las aristas en las que dicho nodo aparece como padre. Esto nos va a ser de gran utilidad más adelante para generar el árbol, puesto que no tendremos que buscar a cada nodo en la lista de aristas para generar sus subárboles.

Para encontrar dicha raíz utilizaremos dos conjuntos. El primero de ellos nos dirá qué nodos son descartados como raíces. Estos nodos son todos aquellos que aparecen como hijos de alguna arista en el árbol. El segundo de ellos nos dirá qué nodos de los que hemos encontrado hasta el momento son candidatos a raíz. Estos nodos serán aquellos que, en el momento de aparecer como padres de una arista, no han aparecido como hijos en ninguna otra anteriormente. En el caso en el que ya hayan aparecido no se añaden, y si aparecen posteriormente se eliminan. Así, tras haber recorrido todas las aristas deberíamos disponer en el conjunto de raíces candidatas solo un identificador de nodo, que va a ser la raíz del árbol. En caso de que no sea así, se deberá a que la información recibida por la entrada es incorrecta, y no habrá árbol que generar.

5.2.2. Generando el árbol

Una vez ya hemos descubierto cuál es el nodo raíz, y ya tenemos procesada la información, vamos recursivamente generando el árbol desde las hojas hacia la raíz. Por ello, nuestro caso base de la recursión serán las hojas, en las que únicamente crearemos un árbol con un único

nodo. La forma de detectar una hoja es en el caso en el que para dicho nodo, no tengamos ninguna arista en la que aparezca como padre. Esta comprobación se puede hacer con un coste constante utilizando la estructura de aristas.

Una vez tenemos generados los subárboles para cada hijo de un nodo dado, simplemente añadimos en el atributo *children* todos estos hijos y tendremos creado el árbol de una altura una unidad superior al anterior. Continuando hasta la raíz generaremos el árbol deseado.

Adicionalmente a esto, guardamos para cada nodo una referencia al árbol del padre. Esto no aparece en el listing 5.3 por simplicidad, y en general *D3 hierarchy* no necesita esta referencia. Sin embargo, nosotros la necesitamos para poder propagar eficientemente las propiedades de cada ejecución desde la hoja en cuestión (no olvidemos que estas propiedades se proporcionan en el campo *executions*) hasta la raíz.

Además, para cada nodo guardaremos qué ejecuciones pasan por dicho nodo, numeradas de izquierda a derecha. Así, para cada nodo tendremos que unir los conjuntos que hemos obtenido para cada uno de sus hijos. En el caso base (una hoja), será un conjunto con un único elemento que nos indica la posición de dicha hoja contándolas de izquierda a derecha. Esta información simplificará enormemente la selección de ejecuciones que se detallará en el apartado 6.3.3.

5.2.3. Añadiendo la información a cada nodo

Tras generar la estructura de árbol, recorremos las estructuras *sets*, *states* y *executions* y vamos agregando a cada nodo la información que encontramos sobre él. Este proceso es sencillo y eficiente, puesto que contamos con una estructura en la que, para cada nodo, tenemos el árbol que tiene a dicho nodo como raíz.

El único atributo que requiere computación adicional a la asignación de referencias es el de las propiedades de ejecución. En este caso, para cada ejecución guardaremos las propiedades en el nodo y las pasaremos hacia arriba en el árbol utilizando el puntero al padre que tiene cada nodo.

Por último, también guardamos en dos listas bien diferenciadas e independientes del árbol generado todas las propiedades que han aparecido la generación del mismo, puesto que tendrán su utilidad más adelante (véase 6.3.2)

Capítulo 6

Decisiones de diseño del visualizador

En el proceso de implementación del visualizador, fue necesario tomar diversas decisiones de diseño para mantener un balance entre sencillez, estética y amigabilidad con el usuario.

La apariencia de la aplicación puede verse en la figura 1.1 (nótese que la apariencia y tamaño de los diversos componentes HTML pueden variar dependiendo del navegador que se utilice, pero su disposición en la pantalla será similar).

Si observamos con más detenimiento la figura, podemos ver que el árbol ocupa la gran mayoría de la pantalla, y lo que más llama la atención del mismo son los nodos, que son los principales elementos interactivos. Por lo tanto, la gran mayoría de las decisiones detalladas en este capítulo se van a centrar en estos componentes. Sin embargo, también va a haber decisiones que tendrán que ver con los enlaces del árbol o con el menú izquierdo, y que iremos detallando a lo largo del capítulo. En caso de que tuviéramos diversas opciones para abordar un problema, explicaremos todas y también el motivo por el que finalmente elegimos una concreta.

6.1. Consideraciones sobre los nodos

6.1.1. Información mostrada en los nodos

Como ya se ha visto en el capítulo 5, los nodos contienen diversa información, como

- El identificador del nodo.
- El estado de las variables para ese nodo.
- El contenido de los conjuntos *backtrack set* y *sleep set*.
- El ejecutor de la tarea para llegar hasta dicho nodo. Podemos apreciar en el Listing 5.2 que ésta será la parte anterior al punto en la etiqueta de la arista que incide en

dicho nodo (el nodo es el hijo en la arista correspondiente).

- El nombre de ejecución y una etiqueta opcional, en caso de que sea una hoja.
- Información extra que no se haya considerado en los otros campos.

Una vez conocemos toda la información que puede estar disponible en cada nodo (es posible que en algún nodo parte de esta información no esté disponible), necesitamos decidir qué información se va a mostrar y dónde va a ser mostrada en el nodo, también teniendo en cuenta los requisitos indicados en el capítulo 4.

Procedemos por partes en los elementos de la lista anterior.

Identificador de nodo y ejecutor de la tarea

En relación con el identificador del nodo, aunque éste nos ha servido para poder generar el árbol, no consideramos que sea una información que aporte nada sobre el algoritmo utilizado, y decidimos descartarla.

Pero sin mostrar los identificadores en el nodo, éstos iban a quedarse vacíos, lo cual teníamos claro que no era una buena opción. Por tanto, decidimos que la mejor información a mostrar iba a ser el ejecutor de la tarea que nos había llevado hasta dicho nodo (de esta forma, el nodo raíz es el único nodo que aparece en blanco).

Por ello, en el interior de los rectángulos se muestra únicamente el actor que ejecuta la tarea. Pero dado que este texto depende del nodo, si fijáramos un tamaño fijo para el nodo podría ocurrir que el texto excediera dichos límites. Como este texto es conocido antes de crear el árbol, y además es fijo (en el sentido de que no cambia en el árbol), en el momento en el que se crea el árbol podemos calcular la longitud del texto de cada nodo y fijar el ancho del rectángulo al máximo de todos los textos. De esta forma, nos aseguramos de que todos los nodos tendrán espacio suficiente para el texto que van a contener en su interior.

Conjuntos

Desde el inicio pensamos que incorporar toda la información sobre los diversos **conjuntos** de cada nodo no era posible, puesto que la longitud de estos conjuntos es arbitraria, lo que complica poder disponer de un espacio fijo en el nodo para mostrar dicha información. Sin embargo, es necesario mostrar estos conjuntos, puesto que contienen información relevante del algoritmo, pero no es necesario tenerlos siempre a la vista, es decir, pueden estar ocultos y mostrarlos al pulsar un botón, por ejemplo. Así, lo que se decidió fue mantenerlos ocultos, y únicamente mostrarlos cuando nos ubicamos encima de un nodo (sin hacer *click*), ocultándolos de nuevo al mover el ratón fuera del nodo.

Además, podía decidirse si aplicar el mismo zoom a estos elementos que al árbol completo. Se decidió que así fuera, puesto que de lo contrario podía ser que el árbol estuviera muy

ampliado y estos paneles fueran muy pequeños, lo que era un tanto extraño.

Variables

En el caso de las **variables**, el problema era el mismo que con la información de conjuntos, puesto que cada nodo puede tener un número indefinido de variables. La solución más coherente era mostrar las variables de igual forma que se ha hecho con los conjuntos. Así, estas variables se van a mostrar si al ubicarnos encima de un nodo, mantenemos pulsada la tecla *Shift*. Ahora bien, esto provoca que si deseamos ver la información de las variables en una traza concreta, tengamos que ir nodo a nodo situándonos encima para visualizar el contenido de las variables. Como es muy probable que se desee ver la información de las variables en varios nodos simultáneamente, era necesario encontrar una solución alternativa al problema.

La opción más inmediata era incorporar la información de estas variables al nodo. Sin embargo, esta opción ya se ha comentado que no era factible. Más aún, el hecho de incorporar un número indefinido de variables al interior del nodo podía desnivelar el árbol y provocar un efecto indeseado. Por lo tanto, decidimos permitir mostrar un número limitado (su valor actualmente está fijado a 2) de variables por nodo, pero dando al usuario la posibilidad de decidir cuántas variables quiere mostrar dentro del rango 0-2 (inicialmente no se muestra ninguna).

Como todos los nodos asociados a un mismo actor tienen el mismo número de variables, la forma de seleccionar las variables es por actor, y se mostrarán en todos los nodos de dicho actor.

Una vez que ya se decidió que las variables seleccionadas iban a ser mostradas en el nodo, había que decidir dónde exactamente. La idea original era que fueran dentro del rectángulo. Se nos ocurrieron dos opciones, con sus respectivos problemas:

1. Crear los rectángulos de forma que se ajustaran al tamaño del texto en cada nodo. De esta forma, cada nodo iba a tener un tamaño diferente, pudiendo provocar una descompensación en el árbol.

2. Crear todos los rectángulos del mismo tamaño. Así, en el momento en el que añadiéramos variables al nodo, todos los nodos cambiarían su tamaño (al máximo de todos los textos a mostrar de cada nodo). Sin embargo, ese espacio libre podía hacer que en ciertos nodos hubiera una gran descompensación entre la altura del rectángulo y la anchura del mismo. Además, no era sencillo modificar el tamaño del rectángulo, puesto que esto obligaba a recalcular todas las posiciones de nodos y aristas en el árbol. Finalmente, también se descartó esta idea.

En base a esto, decidimos que las **variables** aparecieran **en el lado derecho del nodo**, fuera del rectángulo (pero manteniendo el mostrar todas cuando nos situamos sobre un nodo y pulsamos *Shift*).

Colores

Hasta ahora, no se ha comentado nada sobre los colores, puesto que en un principio teníamos todos los nodos del mismo color, hasta que tomamos una decisión al respecto. De igual forma que en cada nodo íbamos a mostrar al actor que había ejecutado la tarea para llegar hasta dicho nodo, pensamos que sería buena idea si reafirmábamos ese concepto asignando un color diferente a cada actor. De esta forma, sería sencillo identificar los nodos que corresponden al mismo actor.

Ahora bien, la elección de colores no puede ser estática, puesto que desconocemos el número de actores que podemos tener en cada árbol, y necesitamos encontrar una solución que valga para todos.

El problema de encontrar un número arbitrario de colores que sean distinguibles por el ojo humano no es un problema fácil, y por supuesto aumenta a medida que el número de colores aumenta. Además, la obtención de estos colores debe ser realizada antes de mostrar el árbol, lo que, dependiendo del algoritmo utilizado, puede llevar un tiempo inasumible para el usuario. Por ejemplo, el algoritmo propuesto en [13] tiene un coste de $O(n^3)$, siendo n el número de colores que deseamos generar, que para n grande, dependiendo de la capacidad de proceso del ordenador del usuario, puede llevar mucho tiempo, más del admisible.

Por lo tanto, no es posible calcular los colores en función del número de actores existentes de forma dinámica. Así, la decisión (temporal) a la que se llegó fue generar un número prefijado de colores lo más distintos entre sí posible e irlos asignando a los actores según van apareciendo, de forma que si nos quedamos sin colores, asignamos el último disponible a todos los restantes actores que aparezcan.

Esta elección puede parecer que no es del todo acertada, puesto que no va a ser posible diferenciar todos estos actores con el mismo color, y la utilización de colores carecerá de sentido. En este caso será aun más de utilidad la leyenda introducida en el apartado 6.5, puesto que permite diferenciar todos los actores que tienen la misma apariencia.

Contorno de los nodos

El contorno del nodo es una parte relevante del mismo, puesto que nos puede aportar información de importancia. En nuestro caso, decidimos que el contorno del nodo indicara si dicho nodo tenía todavía caminos por expandir. Así, en caso de que los tenga, el nodo tendrá un contorno negro, y si no los tiene (es decir, todas las hojas alcanzables desde dicho subárbol son visibles), entonces el nodo no tiene contorno.

Aunque pueda parecer que este elemento visual no aporta apenas nada al árbol, sí que tiene una gran importancia, puesto que habitualmente un nodo va a tener diversos hijos, y si no dispusiéramos de esta herramienta, no sabríamos qué esperar al pulsar sobre dicho nodo (es un nodo hoja, es un nodo interno con todos sus caminos expandidos...) (véase 6.1.3).

Nombre y etiqueta de ejecución

En el caso de las hojas, tenemos información sobre las ejecuciones y una etiqueta opcional para dar algo más de información sobre dicha ejecución.

Mientras que en la cadena sobre el nombre se espera que sea corta (por ejemplo, "*Exec. N*"), en el caso de la etiqueta, esta puede tener una longitud arbitraria. Así, en el caso en el que haya dos hojas muy cercanas entre sí, puede ser que este texto solape, o que el texto de un nodo solape otra ejecución diferente, puesto que no se ha tenido en cuenta para los cálculos realizados en la sección 6.1.2. Por lo tanto, será necesario crear diferentes renglones en este texto en el caso en el que el texto supere una anchura determinada (en este caso, la anchura del rectángulo más la mitad de una constante prefijada que se utiliza en la sección 6.1.2).

Extras

En la sección 5.1.2 introdujimos un campo *extra* en la lista de *states*. Como ya se dijo, este campo permite aportar libertad en el árbol para mostrar la información que se desee en cada nodo (o ninguna si no se aporta).

Dado que desconocemos la longitud de dicho texto, no podemos mostrarlo permanentemente en el árbol. Así, vamos a usar la estrategia que hemos usado anteriormente con los conjuntos y las variables, y vamos a mostrarlo cuando nos ubicamos encima del nodo (y presionamos una tecla). Así, si pulsamos la tecla *Alt* al ubicarnos encima de un nodo que tiene tal información, será mostrada. Si no dispone de esta información no se hará nada.

6.1.2. Separación de los nodos

Para el árbol que queremos generar, estamos interesados en determinar la separación entre los nodos, de forma que toda la información que mostramos en cada momento no se solape (por ejemplo, si no tenemos cuidado es posible que si existe alguna variable con un nombre largo, el texto mostrado solape con el de un nodo con la misma altura que esté a la derecha). Este problema únicamente se presenta en el eje horizontal, puesto que en el vertical nada cambia de tamaño (a excepción de las variables que salen apiladas una encima de la otra, pero que no repercuten en la separación vertical de nodos). Por tanto, aunque no se especifique en lo que sigue de la sección, todas las consideraciones se hacen en el eje horizontal.

D3 ofrece dos formas de elegir la distribución de los nodos.

- Indicar el tamaño del árbol.
- Indicar el tamaño de cada nodo.

En nuestro caso, estamos interesados en proporcionar el tamaño de cada nodo (que no va a ser en ningún caso el tamaño del rectángulo, puesto que *D3* también incluye ahí la separación

entre nodos). Para calcular este valor, debemos tener diversas longitudes en cuenta, como son la anchura del rectángulo y el tamaño del texto de las variables mostradas por un lado, y la longitud del texto de las aristas por otro (podría parecer que este último caso no se va a dar, pero en caso de que el texto de dos aristas que inciden en dos hermanos sea muy largo, es posible que se solapen entre sí).

Este valor de separación entre nodos debe ser recalculado continuamente cada vez que se modifica el árbol, puesto que, dependiendo de los contenidos en cada momento, deberemos disminuir o aumentar el tamaño del nodo, de forma que en ningún caso dejemos más espacio del necesario ni dejemos menos y haya solapamiento.

6.1.3. Comandos sobre los nodos

Como se puede apreciar en la figura 1.1, disponemos de nodos en estructura de árbol. Sin embargo, como ya se indicó en el capítulo 4, los árboles pueden ser muy grandes, por lo que necesitamos que sea posible expandir y colapsar nodos. Para ello, en *JavaScript* existe un evento conocido como *click*, con el cual podemos modificar el comportamiento de un objeto cuando se pulsa sobre él. Distinguimos dos comportamientos distintos al clicar sobre los nodos:

- Expandir un nodo (mostrar sus hijos) si se encuentra colapsado (no se están mostrando sus hijos).
- Colapsar un nodo (ocultar todos sus descendientes) si se encuentra expandido (se están mostrando, al menos, algún descendiente, puesto que puede ser que estos nodos se encuentren también expandidos).

Sin embargo, nosotros estábamos interesados en permitir comportamientos adicionales, como:

- Expandir progresivamente (es decir, con una transición) los caminos no visibles desde dicho nodo.
- Colapsar progresivamente (es decir, con una transición) los caminos visibles desde dicho nodo.
- Colapsar un camino desde un nodo y mostrar únicamente la hoja.

Evidentemente, para poder implementar estos comportamientos iba a ser necesario encontrar una alternativa al evento *click*. Así, decidí que, adicionalmente a la utilización del ratón, se pudiera utilizar ciertas teclas para seleccionar los diferentes opciones de expansión y colapsado de nodos. En concreto, se utilizarán, las teclas *Control*, *Alt* y *Shift*, junto con el uso del botón izquierdo del ratón. Los comandos definidos son los siguientes:

- *Click* sobre un nodo, pulsando la tecla *Shift*: expansión del camino invisible más a la izquierda desde el nodo en el que se pulsó, si existiera alguno no visible.
- *Click* sobre un nodo, pulsando simultáneamente la tecla *Shift* y la tecla *Control*: expansión de todos los caminos desde el nodo en el que se pulsó (comenzando por el de más a la izquierda sin expandir), si existiera alguno no visible, utilizando una transición para expandir cada camino.
- *Click* sobre un nodo, pulsando simultáneamente la tecla *Alt*: colapsado del camino visible más a la derecha desde el nodo en el que se pulsó, si existiera alguno visible.
- *Click* sobre un nodo, pulsando simultáneamente la tecla *Alt* y la tecla *Control*: colapsado de todos los caminos visibles desde el nodo en el que se pulsó (comenzando por el visible de más a la derecha), si existiera alguno visible, utilizando una transición para colapsar cada camino.
- *Click* sobre un nodo, pulsando simultáneamente la tecla *Control*: Colapsado del camino visible más a la izquierda desde el nodo en el que se pulsó en la correspondiente hoja, si existiera alguno visible.
- *Click* sobre un nodo, sin usar ninguna combinación de las anteriores. En este caso, tenemos dos comportamientos diferentes, dependiendo del estado del árbol.
    - Si el nodo sobre el que pulsamos está mostrando al menos un descendiente (puede ser descendiente directo o alguna hoja de los caminos que terminan en su subárbol), entonces colapsamos dicho nodo, es decir, ocultamos todos sus descendientes.
    - Si el nodo sobre el que pulsamos no está mostrando ningún descendiente, entonces expandimos dicho nodo, es decir, mostramos sus descendientes directos.

6.1.4. Icono de ayuda

Toda la información referente a los comandos sobre los nodos detallada en la sección 6.1.3 y otros elementos importantes detallados en las secciones 6.1.1 y 6.2 está presente en la documentación de la herramienta. Sin embargo, dado que en muchas ocasiones los usuarios no leen las instrucciones de uso, consideramos conveniente añadir estos comandos en la pantalla, para que el usuario los tenga siempre a mano por si los necesita en algún momento. Sin embargo, puesto que esta herramienta no se prevé que vaya a ser utilizada constantemente, consideramos que no era buena idea que ocupara demasiado espacio en la pantalla. Por lo tanto, decidimos incorporar un símbolo bastante conocido, como es el de la interrogación azul, que todos interpretamos como "Ayuda".

Para mostrar el texto que informa sobre los diversos comandos y opciones disponibles, basta con colocar el ratón encima del icono. En el momento que el ratón se posicione fuera de este objeto, el texto desaparecerá, liberando la pantalla para permitir la interacción con el árbol.

6.1.5. Botones de recorrido del árbol

Ya se han detallado en la sección 6.1.3 los diferentes comandos que existen desde cada nodo para navegar por el árbol. Sin embargo, en árboles muy grandes, esto puede ser un atraso, puesto que necesitaremos buscar primero el nodo desde el que queremos expandir o colapsar un camino, lo cual puede ser costoso.

Lo cierto es que, para navegar por el árbol, basta con utilizar estos comandos desde el nodo raíz, puesto que este nodo contiene todos los caminos en el árbol. Sin embargo, al igual que ocurre con los restantes nodos, no siempre será rápido o fácil de encontrar.

Por ello, pensamos que sería una buena idea colocar unos botones en alguna parte de la pantalla que simularan este comportamiento, es decir, que cuando el usuario los pulse es como si estuviera haciendo *click* sobre la raíz utilizando uno de los comandos ya explicados en la sección 6.1.3. Estos cuatro botones (*expand one path*, *expand all paths*, *collapse one path* y *collapse all paths*) se han añadido, junto con otros dos nuevos (*expand tree* y *collapse tree*) en la parte inferior de la pantalla, como se puede ver en la figura 1.1.

La razón por la que se han añadido estos dos nuevos botones se detalla a continuación:

Botón de expandir

Con los actuales comandos era muy lento expandir todo el árbol en caso de que tuviera muchos nodos con muchos caminos (es decir, muchas hojas). La forma más aceptable de hacerlo era pulsar el botón "*Expand all paths*", pero este botón utiliza una transición con un tiempo fijo para expandir cada camino, por lo que podía tardar demasiado tiempo en expandir el árbol completo. Tampoco era una opción ir nodo a nodo expandiendo todos los hijos.

Con esta opción, creemos que será más fácil explorar todo el árbol en caso de que el usuario lo desee.

Botón de colapsar

El botón de colapsar tiene un comportamiento idéntico a hacer un *click* estándar sobre la raíz, siempre y cuando ésta esté mostrando nodos, con la salvedad de que este botón siempre se encuentra a la vista del usuario, mientras que la raíz puede salir de la vista del mismo. Por ello, consideramos que este botón era necesario para poder colapsar el árbol al completo, sin necesidad de buscar la raíz o bien pulsar el botón "*Collapse all paths*", que al igual que el botón de "*Expand all paths*" tarda un tiempo en colapsar cada camino.

6.2. Consideraciones sobre los enlaces

Ya sabemos del capítulo 5 que el campo *label* de *edges* es la información de la que disponemos para etiquetar las aristas. Por tanto, esta será la información que mostraremos. Por lo que respecta a su posición, este texto estará centrado en la arista, tanto en altura como en anchura.

Sin embargo, queda un caso a considerar, y es el de qué ocurre si colapsamos un camino en una hoja. En ese caso, tenemos más de una etiqueta que deberíamos mostrar. La opción más evidente es mostrar ese texto conjuntamente. Pero si el número de aristas colapsadas es grande, este texto solaparía con la información del nodo, haciendo que la apariencia no sea la deseada. Por este motivo, esta solución no fue la que se llevó a cabo.

Finalmente, decidimos que la solución más efectiva era añadir el efecto que ya estaba en otros objetos, como los nodos, de situarse encima del objeto para mostrar este texto, y ubicarse fuera del objeto para ocultarlo. De esta forma, esa información no perturba, puesto que solo se muestra si nos ponemos encima del enlace.

Adicionalmente, para que se distinga este caso frente al caso habitual de enlace, decidimos que los enlaces colapsados fueran discontinuos, de forma que sea más claro ver que esos enlaces son de un tipo distinto. También modificamos el texto. En el caso de estos enlaces, mostramos un asterisco en lugar del texto habitual para hacer referencia a que hay información oculta. En la figura 6.1 puede verse un ejemplo de un camino colapsado en una hoja.

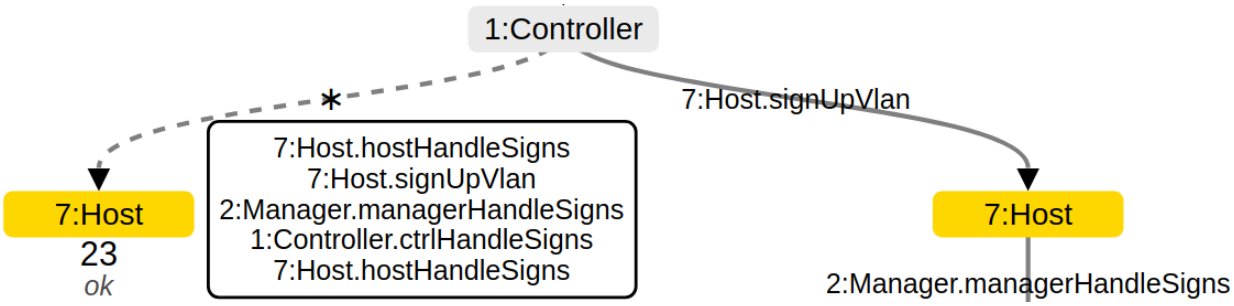

Figura 6.1: Ejemplo de enlaces colapsados

Extras

En la sección 5.1.2 introdujimos un campo *extra* en la lista de *edges*. Este campo permite aportar libertad en el árbol para mostrar la información que se desee en cada arista (o ninguna si no se aporta).

Dado que desconocemos la longitud de dicho texto, no podemos mostrarlo permanentemente en el árbol. Así, vamos a usar la estrategia que hemos usado anteriormente con muchos elementos, como por ejemplo la información de las aristas colapsadas, y vamos a mostrarlo

cuando nos ubicamos encima del enlace (y presionamos una tecla). Así, si pulsamos la tecla *Alt* al ubicarnos encima de una arista que tiene tal información, será mostrada. Si no dispone de esta información no se hará nada.

6.3. Consideraciones del menú izquierdo

En el menú izquierdo permitimos seleccionar variables, propiedades de ejecución y de estado y trazas de ejecución a mostrar.

6.3.1. Variables

Como ya se ha dicho en 6.1.1, vamos a dar la opción al usuario de elegir qué variables quiere que se muestren en el árbol. Por lo tanto, debemos incorporar un menú que admita dicha interacción. La posición de este menú será en la parte superior izquierda de la pantalla, puesto que no deseamos que interfiera demasiado en la interacción del usuario con el árbol.

Puesto que permitimos al usuario elegir más de una variable por actor, lo más coherente es utilizar *checkboxes* para tal fin. Así, mostraremos una lista con todas las variables que se han encontrado en los nodos del fichero de entrada, junto con el actor al que pertenecen. Para que el usuario sepa en todo momento qué es lo que se espera que haga, se añade un breve texto explicativo justo encima de los *checkboxes*.

En el momento en el que el usuario está decidido con la elección que ha realizado, puede pulsar el botón de "Aceptar" que se encuentra abajo y a la derecha de la lista de variables. El comportamiento de este botón es el evidente, es decir, cuando se presiona aceptar se muestran todas las variables que estén seleccionadas, o se muestra un mensaje en caso de que se supere el límite en algún nodo (en caso de que sea así, en el mensaje se indicará qué actor es el que está incumpliendo el límite). Además de este botón, dado que es posible que haya muchas variables diferentes, se ha añadido un botón de "*Clear all*", que permite al usuario limpiar todas las casillas previamente seleccionadas. Por este último motivo también se ha limitado a un tamaño máximo fijo la longitud que se muestra de la lista, teniendo que hacer *scroll* en caso de sobrepasar esa longitud límite.

Por último, dado que ese menú de variables ocupa un tamaño relativamente grande, pensamos que sería buena idea que se pudiera ocultar dicho menú. Para ello, se añade un botón en la parte superior que tiene un comportamiento similar al de un interruptor, puesto que cambia su texto en función del estado entre "*Show variables*" y "*Hide variables*", y se comporta acorde al texto, es decir, en el primer caso muestra el menú de las variables, y en el segundo las oculta, de forma que no ocupa espacio innecesario en la pantalla, y permite al usuario centrarse en lo más importante del visualizador, el árbol.

6.3.2. Propiedades

Ya indicamos en el capítulo 5 que vamos a permitir al usuario interactuar con las propiedades que se pasen al visualizador. Desde el primer momento tuvimos claro que el tratamiento de las propiedades de estado y las de ejecución debía ser idéntico, pero claramente diferenciado. En el caso de estas propiedades, solamente permitimos seleccionar una de cada tipo, o ninguna ("*None*").

Una posible opción para las propiedades de estado (respectivamente con las de ejecución) es añadir un *radio button* por cada propiedad de estado que haya aparecido en el árbol. Sin embargo, esto ocupa mucho espacio, e incluso puede llegar a agobiar en el caso en el que haya muchas propiedades disponibles para seleccionar. Por tanto, descartamos esta opción.

Finalmente nos decidimos por utilizar una barra de selección. En esta barra, añadimos como primera opción (y seleccionada por defecto) la de no mostrar ninguna propiedad ("*None*"). Las restantes opciones serán todas las propiedades que han aparecido en el procesamiento de la entrada (véase 5.2.3).

Pero estas barras sin información adicional pueden resultar un tanto sorprendentes para el usuario de la aplicación, pues ni siquiera indican cuál hace referencia a las propiedades de estado y cuál a las de ejecución. Por lo tanto, al igual que ocurrió con el menú de variables (6.3.1), vamos a agregar un breve texto encima de cada barra de selección para que el usuario sea consciente de qué es lo que se espera que haga con esa barra.

Por lo que respecta al posicionamiento, desde el primer momento tuvimos claro que estos desplegables debían estar junto al menú de variables, por lo que lo añadimos inmediatamente encima, para que no hubiera un efecto extraño en la aparición y desaparición del menú de variables.

Comportamiento tras la selección

Es evidente que, en el momento en el que el usuario cambia alguno de los dos desplegables añadidos a la pantalla, el estado del árbol debe verse afectado. Así, distinguimos dos casos:

- El caso en el que el cambio se produzca en las **propiedades de estado**, deberemos cambiar el estilo de los nodos que cumplan dicha propiedad, para que se note la diferencia, y así el usuario sepa distinguir qué nodos cumplen la propiedad de los que no. La forma que decidimos para hacer este cambio fue modificar el color de todos los nodos, tanto de los que cumplen la propiedad como los que no.
  - Si el nodo **cumple la propiedad**, se pinta el rectángulo de negro, y se cambia la letra del interior a blanco.
  - Si el nodo **no cumple la propiedad**, se pinta el rectángulo de un color muy claro, casi blanco. No se usa el blanco porque entonces las aristas y los nodos daban la sensación de estar "volando".

De esta forma se podrá apreciar un gran contraste entre los nodos que cumplen la propiedad, y los que no.

- El caso en el que el cambio se produzca en las **propiedades de ejecución**, se nos ocurrieron dos opciones.

  - Realizar un cambio de estilo en todos los nodos de la ejecución. Sin embargo, de hacer esto, sería difícil poder combinar la selección de propiedades de estado con las propiedades de ejecución. Así, esta opción quedó descartada.
  - Realizar un cambio de estilo en todos los enlaces de la ejecución. Ésta fue la opción elegida, puesto que no entraba en conflicto con la selección de propiedades de estado, lo cual era nuestro propósito desde un inicio.

  Así, decidimos que lo óptimo era modificar el color de los enlaces implicados. Entre todos los colores disponibles, decidimos que el rojo era el color que más se ajustaba para ello, puesto que buscamos que estas ejecuciones resalten sobre las restantes. Adicionalmente, para añadir aún más resaltado, engrosamos ligeramente los enlaces, para que se aprecie un mayor contraste.

6.3.3. Trazas

Tras completar la implementación del visualizador, lo testeamos con diferentes programas de entrada y nos dimos cuenta de que el rendimiento del visualizador disminuye cuando los árboles a visualizar crecen de tamaño, llegando a aparecer un *lag* bastante apreciable para el usuario. El motivo por el que esto sucede es porque se están mostrando demasiados nodos en el SVG, aunque están ocultos por el límite en el tamaño de la pantalla, y el navegador no es capaz de manejarlo eficientemente. Para paliar este problema, se nos ocurrió permitir al usuario seleccionar un intervalo de ejecuciones a mostrar, de forma que pueda ir viéndolas progresivamente si así lo desea.

Para ello, añadimos dos elementos de entrada de texto que permiten al usuario introducir únicamente números. En un inicio, los números que aparecen en estos elementos son el número mínimo de traza (1) y el número máximo, de forma que el usuario sepa desde el comienzo cuántas trazas va a visualizar, y así pueda decidir si desea limitar el rango. Para resetear estos valores, disponemos de un botón de *reset*, que restablece los valores a 1 y al máximo número de ejecuciones.

También disponemos de un botón de aceptar, que aplica la selección del usuario al árbol, en caso de que sea coherente (es decir, el intervalo no sea vacío y esté comprendido dentro de los límites de las trazas).

A su vez, pensamos que podía ser interesante hacer saber al usuario que el árbol tiene trazas ocultas (a la izquierda o a la derecha de las que se están mostrando, o a ambos lados). Nótese que aunque esto esté claro a la izquierda, pues es fácil recordar que el menor número de

traza disponible es el 1, a la derecha no es tan sencillo en caso de que dispongamos de un gran número de trazas. Para paliarlo, se nos ocurrieron dos opciones:

- Añadir un nodo invisible, de forma que solamente apareciera la arista, en la que como texto se indicara que a ese lado había trazas ocultas.
- Mostrar una traza más con una opacidad baja, para que haya un efecto de degradado. Esta opacidad se aplicaría a nodos, a enlaces y a los cuadros de texto que se muestran cuando nos situamos encima de los nodos pertenecientes a estas trazas.

Entre estas dos opciones, consideramos que la mejor de las dos era la de añadir una traza con menor opacidad, puesto que es mucho más visual, y nos permite visualizar una más si así lo deseamos sin necesidad de introducir nuevos valores a los elementos de texto.

Por último, para evitar incoherencias en el árbol decidimos que en el momento de aplicar el cambio de intervalo se colapse el árbol al completo, lo cual pone de manifiesto (en caso de que estuviera parcialmente expandido anteriormente) que se han aplicado los cambios con éxito.

6.4. Consideraciones sobre la información del algoritmo

Ya se indicó en las sección 5.1.1, en el Listing 5.1, que SYCO proporciona información general sobre la ejecución del algoritmo de exploración. Esta información, ya que es proporcionada por SYCO, veíamos razonable agregarla al visualizador.

Como es de esperar, esta información no es del todo relevante, puesto que incluso ya se muestra parte por la salida estándar, por lo tanto no va a tener tanta relevancia como tienen otros elementos en la pantalla. Así, decidimos que estuviera abajo a la derecha, junto con el botón de "Ayuda". En este caso, el icono va a ser el de una i de información, con un círculo negro alrededor, para que destaque.

Su comportamiento va a ser análogo al del icono de "Ayuda": si ubicamos encima el ratón, se mostrará la información recibida. Si nos salimos de dicho icono, la información se ocultará.

Por último, decidimos que la información recibida tuviera formato HTML porque va a ser incluida en un panel HTML, y dado que no podemos hacer suposiciones de los elementos que se van a recibir, preferimos que la información venga ya preparada de SYCO, de forma que pueda ser mostrada directamente.

6.5. Consideraciones sobre la leyenda

La funcionalidad de la leyenda no fue algo que consideráramos al inicio, puesto que nuestra atención estaba enfocada en conseguir mostrar el árbol de forma satisfactoria, con todas las

posibilidades que finalmente se han terminado añadiendo y que han sido detalladas a lo largo de este capítulo. Sin embargo, dado que las restantes funcionalidades fueron terminadas con tiempo suficiente, consideramos que era una característica interesante para ser añadida, puesto que permite condensar en un espacio bien definido la asignación de colores a cada nodo (veáse el apartado 6.1.1).

Adicionalmente, la intención era que esta leyenda fuera completamente funcional, es decir, que no proporcionara información estática, sino que ciertos eventos sobre ella permitiesen cambiar el estado del árbol. Sin embargo, en el momento en el que se están analizando propiedades de estado, los actores que realizan cada tarea pasan a un segundo plano, por lo que en ese caso la leyenda se comporta como un elemento estático.

Los eventos que dinamizan la leyenda (y por consiguiente el árbol) son:

- Situarnos encima del elemento de la leyenda que referencia a un actor concreto: consideramos que situarnos encima de algún actor implica que estamos interesados en resaltar dicho actor en el árbol (y ningún otro). Así, lo que hacemos es aclarar los restantes, y mantener el color del actor actual, logrando el efecto deseado.
- *Click* sobre un elemento de la leyenda: funciona como un interruptor, es decir, nos permite "desactivar" todos los nodos del mismo actor en caso de que estén siendo mostrados, aclarando sus rectángulos, o bien mostrarlos de nuevo en caso de que estuvieran desactivados.
- Salirnos del elemento sobre el que nos habíamos situado: tras mover el ratón fuera del elemento sobre el que nos encontrábamos, se entiende que no tenemos interés en mostrar la información de dicho nodo. Por ello, los nodos tendrán el mismo color que el elemento de la leyenda que haga referencia al mismo actor que el nodo.

Por último, dado que el número de actores que podemos visualizar puede ser arbitrariamente grande, necesitamos limitar el tamaño de la leyenda (pues de lo contrario podría llegar a sobrepasar la altura de la pantalla, ocultando la información). Así, limitamos el tamaño de esta leyenda a la mitad de la altura de la pantalla.

Capítulo 7

Integración con EasyInterface

En el capítulo 5 indicamos que SYCO generaba una salida, y comentamos que ésta era tomada por el visualizador, sin especificar realmente cómo se podía realizar esta conexión.

En realidad, estos dos componentes pueden funcionar independientemente, aunque el visualizador se ha desarrollado específicamente para SYCO (y la entrada, por lo tanto, está basada en la salida que SYCO genera), por lo que está pensado para que sea ejecutado tras haber ejecutado SYCO, y lo ideal es que sea de forma automática.

La opción más sencilla para conectar ambos servicios consiste en copiar el fichero JSON que SYCO genera en */tmp/pet* en la carpeta *tmp* del visualizador. De esta forma, si arrancamos el visualizador y le indicamos el fichero correcto de forma manual, deberíamos ser capaces de visualizar el árbol de ejecución. Sin embargo, esta forma es lenta, sobre todo en el caso en el que se estén haciendo pruebas en las que se requiera visualizar muchas veces un mismo árbol, o árboles diferentes.

Por lo tanto, es necesario encontrar una solución alternativa para realizar esta conexión, preferiblemente de forma transparente para el usuario (en el sentido de que únicamente necesite seguir los pasos una vez, no múltiples).

A lo largo de esta capítulo daremos dos formas diferentes de poder conectar ambos componentes.

7.1. Configurando Apache

Las formas de conectar ambos componentes que detallaremos en este capítulo hacen uso de un servidor Apache. Por tanto, antes de comentar estos métodos es necesario explicar cómo configurar Apache para evitar errores posteriores.

En primer lugar, es necesario instalar Apache correctamente. No se va a detallar aquí, pero puede seguirse el tutorial dado en `https://github.com/abstools/easyinterface/blob/`

master/INSTALL.md.

Una vez que se ha hecho esto, y asumiendo que disponemos de acceso a un servidor, por ejemplo, el de Costa, vamos a activar una funcionalidad que nos permite alojar nuestras páginas web para poder ejecutarlas en el servidor remoto utilizando Apache.

Para ello, es necesario seguir los siguientes pasos:

Paso 1: Habilitar el módulo *userdir* ejecutando el comando

```
$ sudo a2enmod userdir
```

Paso 2: Reiniciar el servidor Apache ejecutando

```
$ sudo service apache2 restart
```

Paso 3: Crear una carpeta *public_html* en el directorio *home*. Para ello, ejecutar

```
$ mkdir ~/public_html
```

Paso 4: Darle los permisos necesarios a la carpeta *public_html* (generalmente ya tendrá estos permisos)

```
$ chmod 0775 ~/public_html
```

Paso 5: Copiar la carpeta con el código del visualizador en el interior de *public_html*. Supondremos en lo que sigue que el código del visualizador se encuentra en una carpeta *code*, y en su interior se encuentra el fichero *index.html*. Suponiendo que estamos en la carpeta donde está *code*, hacemos

```
$ cp -ar ./code ~/public_html
```

Evidentemente la carpeta con el código del visualizador puede situarse en cualquier ruta, siempre que esté dentro de la carpeta *public_html*.

Paso 6: Darle todos los permisos a la carpeta *tmp*, ejecutando

```
$ chmod 0777 ~/public_html/code/tmp
```

Una vez se ha hecho esto, ya deberíamos tener Apache configurado correctamente, y nuestro código en la carpeta correcta, preparado para ser ejecutado.

7.2. Integración utilizando Apache

Una vez ya hemos completado los pasos de la sección 7.1, estamos preparados para integrar ambos servicios.

Antes de eso, vamos a explicar cómo se ha hecho la integración. Tras haber instalado Apache, y tener acceso a un servidor, como *localhost* (aunque realmente funcionaría con cualquier otro al que tengamos acceso), para acceder a nuestro fichero *index.html* donde se va a visualizar el árbol, simplemente tenemos que acceder a

```
http://localhost/~*****/code?token=$token
```

Listing 7.1: Ruta del fichero *index.html*

donde ***** hace referencia al nombre de usuario.

Lo más importante de lo anterior es el uso de un *token*. Este token se genera usando el PID del proceso que ejecuta, al que se le aplica un *hash* utilizando el programa UNIX *md5sum*. Esto generará un número 128 bits (expresado en hexadecimal), del que solamente utilizamos las 8 primeras cifras hexadecimales. Así, lo que hacemos es que el fichero JSON que SYCO genera se copie automáticamente en la carpeta *code/tmp/* que ya indicamos anteriormente, con el nombre *data_{token}.json*.

Una vez hecho esto, solo nos queda buscar una forma automática de indicarle a nuestro código *JavaScript* que el fichero que queremos abrir es el que utiliza el token que hemos creado. Aquí es donde se utiliza la ruta del Listing 7.1. En ella, pasamos el *token* utilizado, de forma que el código *JavaScript* pueda obtenerlo, y utilizarlo para abrir el fichero JSON correcto.

Disponemos de dos métodos relativamente similares para conectar las dos herramientas, que van a ser detallados a continuación.

7.2.1. Integración en local

Para este método, lo único que es necesario es haber seguido los pasos del capítulo 3 y de la sección 7.1. Una vez hecho eso, vamos a crear un *script* de *shell* en el que vamos a recibir por parámetro todos los *flags* que queremos pasarle a SYCO. Tras ello, copiaremos el fichero JSON que SYCO genera en la carpeta *code/tmp* ya nombrada antes, y abriremos dicho HTML. El código del *script* es el siguiente:

```
#!/bin/bash

# delete files older than one day
find /home/*****/public_html/code/tmp/ -name "data_*" -mtime +1 -delete

# create a token and temporary file name
```

```
date="`date`"
token="`echo "$$ $date" | md5sum | cut -c1-8`"
datafile="data_$token.json"

timeout 60s /home/*****/Systems/pet/src/interfaces/shell/syco $@  &> /tmp/x

if [ $? == 0 ]; then
    cp /tmp/pet/execs_main.json /home/*****/public_html/code/tmp/$datafile

    if [ $? == 0 ]; then
        xdg-open http://localhost/~*****/code?token=$token
    else
        echo "SYCO executed correctly, but there was an error copying JSON output
             file to public_html folder."
    fi
fi
# get file content
cat /tmp/x
```

Listing 7.2: *Script* de *shell* para conectar SYCO y el visualizador en local

Para ejecutar este *script*, es necesario seguir los siguientes pasos:

Paso 1: Crear el fichero ejecutando

```
$ touch nombrefichero.sh
```

Paso 2: Darle permisos de ejecución

```
$ chmod +x nombrefichero.sh
```

Paso 3: Abrir el fichero y copiar el contenido del Listing 7.2.

Paso 4: Ejecutar el *script* con los argumentos que deseemos. Siguiendo el ejemplo de la sección 3.3, basta con ejecutar

```
$ ./nombrefichero.sh ~/Systems/pet/examples/apet/misc/abdulla/floating_re
ad.abs -obj_sel_policy nondet -sched nondet
```

lo cual, si todo va según lo esperado, debería abrir el navegador por defecto configurado en el sistema y mostrarnos el árbol.

7.2.2. Integración con EasyInterface

El método descrito en la sección 7.2.1 tiene una desventaja clara, y es que necesitamos conocer los distintos argumentos que SYCO admite para ejecutar correctamente, o consultar

la ayuda de SYCO, y todo esto desde la ventana de comandos. Aunque deberíamos estar familiarizados con esta técnica, puede ser que no nos sintamos del todo cómodos con ella. Por lo tanto, sería beneficioso poder utilizar una herramienta que aporte una interfaz gráfica, y que permita la integración de ambos componentes.

Por suerte, existe una herramienta (en la que ya está funcionando SYCO previamente) que puede facilitar la integración. Esta herramienta es EasyInterface, y puede encontrarse en https://github.com/abstools/easyinterface. Su instalación es relativamente sencilla, y la gran mayoría ya debería estar completada de la sección 7.1.

Tras completar la instalación de EasyInterface, lo último que vamos a necesitar va a ser modificar algunos ficheros de configuración de la herramienta siguiendo los pasos detallados a continuación. En estos pasos, asumiremos que se ha hecho *clone* del repositorio en la ruta *~/Systems*.

Paso 1: Crear dos carpetas *pet* en la ruta *~/Systems/easyinterface/server/config* y *~/Systems/easyinterface/server/bin*

```
$ mkdir ~/Systems/easyinterface/server/config/pet
$ mkdir ~/Systems/easyinterface/server/bin/pet
```

Paso 2: Modificar el fichero *eiserver.default.cfg* de *config* con el siguiente contenido:

```
<eiserver>
  <apps       src="./pet/apps.cfg"     />
  <examples   src="./default/examples.cfg" />
  <sandbox>
    <sandboxprop name="timeout" value="300"/>
    <sandboxprop name="max_proc" value="30"/>
    <sandboxprop name="logpath" value="./exec.log"/>
  </sandbox>
</eiserver>
```

Paso 3: En la carpeta *pet* de *config*, crear un fichero *apps.cfg* con el siguiente contenido:

```
<apps>
 <app id="syco"             src="./pet/syco/syco.cfg"  />
 <app id="echo"             src="./default/echo.cfg"   />
</apps>
```

Paso 4: Crear la carpeta *syco* en *config/pet* y crear el fichero *syco.cfg* con el contenido que se encuentra en el link https://mega.nz/file/bE4BmS6K#ZeDZF5lK40M0yQ6S07d_hoI5Akly3VKGasEepzhA2d8.

Paso 5: En la carpeta *pet* de *bin*, crear el fichero *envisage_settings.sh* con permisos 0755 y con el contenido del fichero que puede encontrarse en https://mega.nz/file/rJohySrR#TfNbSJYl2fVj-1VUar0d7b5O86vRIMpqkqdXuKtgt-M.

Paso 6: En la carpeta *syco* de *bin/pet*, crear un fichero syco.sh con permisos 0777 y con el contenido del fichero siguiente:

```
#!/bin/bash

# execute saco_settings.sh to set some environment variables needed by
# costabs, etc.

# costabs configuration
export COSTABSHOME=/home/****/Systems/costa/costabs

export LD_LIBRARY_PATH=/usr/local/lib:/usr/local/lib/ppl

# delete files older than one day
find /home/****/public_html/code/tmp/ -name "data_*" -mtime +1 -delete

# create a token and temporary file name
date="`date`"
token="`echo "$$ $date" | md5sum | cut -c1-8`"
datafile="data_$token.json"

#echo "/home/****/Systems/pet/src/interfaces/shell/syco $@"
timeout 60s /home/****/Systems/pet/src/interfaces/shell/syco $@ -json_dir
_ei http://localhost/~****/code?token=$token  &> /tmp/x

# If costabs exit with exit-code 0 we just print the output to the
# stdout, otherwise we print an error message to the stdout as well
if [ $? == 0 ]; then
    cp /tmp/pet/execs_main.json /home/****/public_html/code/tmp/$datafile

    if [ $? == 0 ]; then
        cat /tmp/pet/output.xml
    else
        echo "SYCO executed correctly, but there was an error copying"
        echo "JSON to public_html folder"
        echo "output won't be shown. Solve the error and try again"
    fi
else
    echo "<eiout>"
    echo "<eicommands>"
    echo "<printonconsole consoleid='Error'>"
    echo "<content format='text'>"
    cat /tmp/x
    echo "</content>"
    echo "</printonconsole>"
    echo "<dialogbox boxtitle='Execution Error' boxwidth='350'>"
```

```
    echo "<content format='html'>"
    echo "<span style='color:red;' >syco exit with non-zero exit code: $R
        </span>"
    echo "</content>"
    echo "</dialogbox>"
    echo "</eicommands>"
    echo "</eiout>"
fi
```

donde *** hace referencia al nombre de usuario. Nótese que el contenido de este fichero es muy similar al de la sección 7.2.1.

Tras haber seguido todos estos pasos de configuración, solamente quedaría ejecutar los programas y visualizar los árboles, directamente desde la interfaz web. Para ello, acceder a

```
http://localhost/ei/clients/web/
```

Listing 7.3: Ruta de EasyInterface

y ejecutar un programa cualquiera, por ejemplo el que dimos en la sección 3.3 cuya ruta es *~/Systems/pet/examples/apet/misc/abdulla/floating_read.abs*. Deberíamos obtener la siguiente salida:

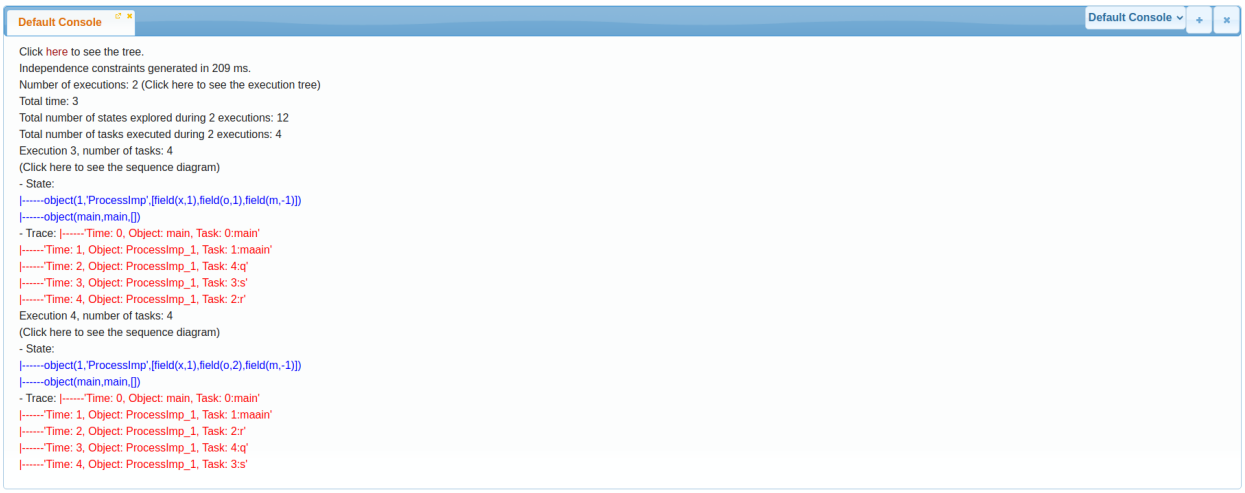

Figura 7.1: Ejemplo de salida de EasyInterface

Pulsando sobre *here*, en la parte superior deberíamos visualizar el árbol con una apariencia similar a la de la figura 1.1.

Capítulo 8

Conclusiones

8.1. Resumen del trabajo realizado

La obtención del visualizador/analizador ha sido fruto del trabajo de varios meses. Para lograr obtener una versión funcional de dicho visualizador, fue necesario tomar diversas decisiones a lo largo de la implementación del mismo, buscando siempre la sencillez, la intuitividad, la claridad, la utilidad y la estética con cada elemento añadido. Así, las decisiones no solamente hacen referencia a qué elemento se añade, sino también a su estilo (color, tamaño, forma, etc.) y a su posicionamiento. Las decisiones detalladas en el capítulo 6 se recogen en la figura 1.1, que muestra la versión final del visualizador. Podemos ver en dicha figura que el elemento central del visualizador es el árbol, aunque hay otros elementos en los bordes de la imagen que tienen diversos objetivos como:

- Cambiar el estado del visualizador/analizador: botones para elegir diversos recorridos, *checkboxes* para la selección de variables, barras de selección para la selección de propiedades, tanto de estado como de ejecución y elementos de entrada de texto para elegir el intervalo de trazas a mostrar.
- Proporcionar información sobre la asignación de colores a cada actor: leyenda.
- Proporcionar información general: icono de información.
- Ayudar al usuario a utilizar el visualizador: icono de ayuda.

Antes de comenzar con esta etapa de implementación del visualizador/analizador, tomamos diversas decisiones que nos permitieron comenzar con dicha etapa. Entre esas decisiones, destacan la elección de **SVG** como formato de imagen y ***JavaScript*** como lenguaje de programación, junto con la especificación de la salida de SYCO, que por tanto iba a ser la entrada del visualizador. En los capítulos 4 y 5 se detallan todas estas decisiones, junto con algunas otras tomadas antes del inicio de la implementación.

Con todo lo anterior, solamente disponíamos de un visualizador que funcionaba de forma independiente de SYCO, pero que tomaba como entrada la salida que generaba. La intención inicial era que este visualizador funcionase de forma integrada con SYCO, por lo que fue necesario invertir parte del tiempo dedicado a este Trabajo de Final de Grado para integrar el visualizador con EasyInterface, una herramienta con interfaz web desarrollada por el grupo de investigación Costa en la que SYCO ya funcionaba previamente. Dado que mis conocimientos sobre esta herramienta eran limitados, pues no la había utilizado nunca, decidimos pedir ayuda al profesor Samir Genaim, también miembro del grupo de investigación, que nos dio el soporte necesario para integrar ambos componentes de la manera más sencilla posible. La explicación de este proceso puede encontrarse en el apartado 7.2.2.

8.2. Trabajo futuro

Como ya se dijo en el capítulo 4, antes de comenzar con la implementación del visualizador se pensaron diversas funcionalidades que podían ser útiles y necesarias, llegando a implementar todas ellas, e incluso algunas más que no estaban pensadas de inicio (como la leyenda, o la selección de trazas). Tras finalizar esta parte, tratamos de pensar en diversas mejoras y modificaciones del visualizador/analizador, ayudados de otros miembros del grupo de investigación Costa, y llegamos a la conclusión de que existían diversas ramas en las que se podía mejorar.

En primer lugar, aunque actualmente solo se han utilizado *backtrack set* y *sleep set* como conjuntos para los nodos, en un futuro podemos estar interesados en añadir nuevos conjuntos (como los *don't-do sets*, véase [5]). En este caso, dependiendo de si estos conjuntos se van a añadir definitivamente o solamente aparecen en algunos casos, necesitaremos hacer cambios o no en el código del visualizador.

En el segundo caso, únicamente con agregar esta información al campo *extra* de los nodos ya podríamos visualizar dicha información en el nodo. Sin embargo, dado que ya mostramos la información de los conjuntos en un cuadro de texto distinto, podría quedar confuso tener que mostrar dos cuadros de texto distintos para observar información que conceptualmente debería ir unida.

La solución óptima pasaría por implementar un reconocedor de conjuntos, de forma que en el campo *sets* descrito en el apartado 5.1.2 podamos adaptarnos a todos los conjuntos distintos que se aporten, mostrando toda esa información que recibimos para cada nodo en el cuadro de texto designado para ello.

En segundo lugar, el visualizador se ha creado únicamente para analizar trazas generadas por SYCO, y por lo tanto su entrada está muy centrada en facilitar el intercambio de información entre SYCO y el visualizador. También los componentes adicionales (por ejemplo, selección de variables) se han agregado porque el visualizador que buscábamos desarrollar era para visualizar la salida de SYCO.

Sin embargo, el grupo de investigación Costa (que es el propietario de SYCO) también utiliza otras herramientas diferentes que generan árboles (como Nidhugg [23] y aPET [3, 4]), con sus correspondientes particularidades. Así, puesto que la herramienta está pensada para funcionar con SYCO, no podemos utilizarla directamente con estos nuevos componentes.

Aún así, es conveniente indicar que, dado que en ambos casos se desea mostrar un árbol, lo más apropiado es reutilizar este trabajo, ya sea solo a nivel conceptual, tomándolo como ejemplo para hacer un visualizador independiente para las otras herramientas, o bien adaptando éste para que pueda funcionar con otros componentes.

Si deseamos adaptar la herramienta, es necesario tener en cuenta que dependiendo de qué herramienta deseemos utilizar los cambios a realizar serán mayores (pero no por ello complejos). Por ejemplo, en el caso de Nidhugg los cambios serían mayores que en el de aPET, puesto que el paradigma de funcionamiento de este componente y el de SYCO son distintos (Nidhugg trabaja con paradigma *multithread* mientras que SYCO lo hace con paradigma de actores). De esta forma, también deberíamos adaptar ciertos conceptos del árbol (por ejemplo, la información mostrada en las aristas tendría otro enfoque diferente, pues pasaríamos de mostrar tareas ejecutadas por actores a instrucciones ejecutadas por *threads*).

En tercer lugar, aunque el visualizador ya incorpora una funcionalidad para solucionar los problemas de *lag* del visualizador cuando el número de nodos a mostrar es grande (véase 6.3.3), existen otras formas para mejorar la experiencia del usuario cuando el número de trazas es grande. Entre ellas, destaca el permitir que el visualizador reciba diversos archivos JSON, de forma que el usuario pueda visualizar las trazas exploradas progresivamente, sin necesidad de esperar a que finalice la exploración completa del programa introducido para mostrar el fichero JSON completo, lo cual para ficheros grandes puede llevar un tiempo considerable.

En cuarto y último lugar, en el visualizador se han incorporado diferentes recorridos y formas de simplificar lo que se muestra en el árbol (véase apéndice B). Sin embargo, existen otros tipos de recorridos que pueden facilitar la comprensión y condensar la información que se está mostrando, y que además pueden contribuir a mejorar el rendimiento del visualizador. Entre ellas, destaca:

- Mostrar solamente nodos de bifurcación y las hojas de cada ejecución, donde entendemos por bifurcación aquellos nodos que tienen al menos dos hijos. De esta forma, colapsaríamos todos los nodos intermedios que solamente disponen de un hijo, y mostraríamos la información de las aristas de la misma forma que hacemos con los caminos colapsados en las hojas: al situarnos sobre el enlace que une ambos nodos.

- Colapsar todas las aristas que tengan al mismo actor. En determinadas ocasiones, un mismo actor ejecuta varias veces consecutivas, realizando diferentes tareas. Podría ser interesante acortar todas estas tareas que ejecuta un mismo actor en un solo nodo, de forma que se muestre en las aristas la información de las tareas que hemos omitido al ubicarnos sobre ella.

En el caso del paradigma *multithread* (como el que utiliza Nidhugg), esto se traduciría en ocultar todas las instrucciones que son ejecutadas por un mismo *thread* en una única arista.

Capítulo 9

Conclusions

9.1. Summary of the work done

Obtaining this visualizer/analyzer has been the result of the work of several months. In order to obtain a functional version of this visualizer, it was necessary to take diverse decisions during the implementation of it, always looking for simplicity, intuitiveness, clarity, utility and aesthetic with every added element. Because of that, the decisions not only refer to which element was added, but also to its style (color, size, shape, etc.) and its positioning. All the decisions detailed in chapter 6 are fulfilled in figure 1.1, which shows the final version of the visualizer. We can see in that picture that the central element of the visualizer is the tree, but there are other elements at the borders of the screen that have diverse objectives, such as:

- Change the state of the visualizer/analyzer: buttons to choose different path traverses, check-boxes to select variables, selections to choose execution and state properties and text input elements to choose the interval of traces to display.
- Give information about the mapping of workers to colors in an interactive way: legend.
- Give general information about the exploration: information icon.
- Guide the user in the usage of the visualizer: help icon.

Before starting with the implementation of the visualizer/analyzer, diverse decisions were taken that allowed us to start with this stage. Among these decisions, we can highlight choosing **SVG** as image format and ***JavaScript*** as programming language, together with specifying SYCO's output (because it was also input of the visualizer). In chapters 4 and 5 we detail all these decisions, among others that were taken before the start of the implementation.

With all the above, we only had a visualizer that worked independently from SYCO, but that took the output that SYCO generated. The initial idea was that this visualizer worked as a unit with SYCO, so we needed to invest some of the time dedicated to this end of degree project on integrating the visualizer with EasyInterface, a tool developed by the Costa group that has a web interface, in which SYCO was already working previously. However, provided that my knowledge of this tool was limited, because I had never used it before, we decided to ask for help to professor Samir Genaim, also a member of Costa, who gave us the needed support to integrate both components in the easiest way. The explanation of this integration process can be found in part 7.2.2.

9.2. Future work

As it has already been said in chapter 4, before starting the implementation of the visualizer we thought about different features that could be useful and necessary, implementing them all and even some others that were not considered at the beginning (like the legend). After finishing this part, we tried to think of diverse improvements and modifications of the visualizer/analyzer, with the help of other Costa members, and we reached the conclusion that there were several fields over which this tool could be improved.

First of all, even though right now we are only using *backtrack set* and *sleep set* as the nodes sets, in the future we may be interested in adding new sets (like *don't-do sets*, see [5]). In that case, depending on if those sets are going to be added permanently or not, we will need to change the code of the visualizer.

If we are not adding them permanently, only by adding this information to the node *extra* field we would be able to visualize this information inside the node. However, provided that we are showing the information of the sets in a different text box, it would be confusing to show two different text boxes that contain information that should go together.

The optimal solution would be to implement a *set recognizer*, so that in *sets* field described in section 5.1.2 we are able to adapt to every type of set that is provided, showing all the information that we receive in the text box designed for it.

As a second direction, the visualizer has been created to analyze traces generated by SYCO, and because of that, its input is focused on facilitating the interchange of information between SYCO and the visualizer. Furthermore, all the additional components, such as variable selection, have been added because the visualizer we were trying to develop was to visualize SYCO's output.

However, the Costa group (SYCO's propietary) also uses other tools that generate trees (such as Nidhugg [23] and aPET [3, 4]), with its particularities. For that reason, because the visualizer is thought to work with SYCO, we can't use the tool straightaway for this other tools.

Still, it is convenient to say that, provided that in both cases we want to represent a tree,

the most appropriate approach is to reuse this work, either only at concept level, taking it as an example to create new independent visualizers for the other tools, or adapting it so that it can work with the other components.

If we want to adapt the tool, it is necessary to take into account that depending on which tool we want to use the changes will be bigger (but not for that reason more complex). For example, in the case of Nidhugg changes would be bigger than in the case of aPET, because the working model of this component is different from SYCO's (Nidhugg works with multi-thread model while SYCO works with workers model). Thus, we should also adapt some of the tree concepts (for example, the information shown in the edges has a different approach, because we will transition from showing tasks executed by workers to instructions executed by threads).

The third direction is that, even though the visualizer has a functionality to solve the lag problems when the number of nodes to show is big (see 6.3.3), there are other ways to improve the user experience when the number of traces is big. Above all, we can highlight one: the visualizer is prepared to receive multiple JSON files, so that the user can visualize the explored traces progressively, without needing to wait for SYCO to finish exploring all the different traces to create a JSON file containing all of them, which for big files can take long.

The last direction is that the visualizer has different traverses and ways to simplify what is shown in the tree (see appendix B). However, there are other traverses that can facilitate the understanding and can condense the information that is being shown, and that can even increase the performance of the visualizer. Among them, we can highlight:

- Show only bifurcation nodes and leaves of each execution, where we understand by bifurcation nodes all these nodes that have at least two children. Thus, we will merge all the intermediate nodes that only have one child, and we will show the information of the edges in the same way we do it in merged paths in leaves: by mousing over the edge that connects both nodes.

- Merge all the edges that have the same worker. In some circumstances, the same worker executes several tasks in a row. It may be interesting to shorten all these tasks that are executed by the same worker in only one node, so that we show in the edge the information of all the edges that have been merged.

  In multi-thread model (as Nidhugg uses), this could be translated to hide all the instructions that are executed by the same thread in only one edge.

Bibliografía

[1] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. COSTA (Cost and Termination Analyzer) Tool. https://costa.fdi.ucm.es/saco/web/.

[2] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. COSTA: A Cost and Termination Analyzer for Java Bytecode. In *Proceedings of the Workshop on Bytecode Semantics, Verification, Analysis and Trans-formation (Bytecode)*, Budapest, Hungary, 2008. Electronic Notes in Theoretical Computer Science.

[3] E. Albert, P. Arenas, M. Gómez-Zamalloa, and P. Y. H. Wong. aPET tool. https://costa.fdi.ucm.es/apet/.

[4] E. Albert, P. Arenas, M. Gómez-Zamalloa, and P. Y. H. Wong. aPET: a test case generation tool for concurrent objects. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 595–598, New York, NY, USA, 2013. Association for Computing Machinery.

[5] E. Albert, M. G. de la Banda, M. Gómez-Zamalloa, M. Isabel, and P. J. Stuckey. Optimal context-sensitive dynamic partial order reduction with observers. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, pages 352–362, New York, NY, USA, 2019. Association for Computing Machinery.

[6] E. Albert, M. Gómez-Zamalloa, and M. Isabel. SYCO (Systematic Testing for Concurrent Objects) Tool. http://costa.fdi.ucm.es/syco.

[7] E. Albert, M. Gómez-Zamalloa, and M. Isabel. SYCO: A systematic testing tool for concurrent objects. In *Proceedings of CC 2016*, pages 269–270. ACM, 2016.

[8] E. Albert, M. Gómez-Zamalloa, and G. Puebla. PET (Partial Evaluation-based Test Case Generator) Tool. https://costa.fdi.ucm.es/pet/pet.php.

[9] E. Albert, M. Gómez-Zamalloa, and G. Puebla. PET: A Partial Evaluation-based Test Case Generation Tool for Java Bytecode. In *Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '10, pages 25–28, New York, NY, USA, 2010. Association for Computing Machinery.

[10] M. Bostock. Collapsible Tree. https://observablehq.com/@d3/collapsible-tree.

[11] M. Bostock. D3 Hierarchy. https://github.com/d3/d3-hierarchy.

[12] Bugseng. Parma Polyhedra Library. https://www.bugseng.com/parma-polyhedra-library.

[13] P. Campadelli, R. Posenato, and R. Schettini. An algorithm for the selection of high-contrast color sets, 1999.

[14] CodeWall. How to make tree visualizations with JavaScript and SVG tutorial. https://www.codewall.co.uk/how-to-make-tree-visualizations-with-javascript-and-svg-tutorial/.

[15] CSSScript. SVGTree Demos. https://www.cssscript.com/demo/create-svg-based-tree-structure-using-javascript-svgtree/.

[16] Open Eye. Generating interactive SVG images. https://docs.eyesopen.com/toolkits/python/depicttk/svg.html.

[17] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 110–121, New York, NY, USA, 2005. Association for Computing Machinery.

[18] Aula Formativa. Definición y usos comunes del formato SVG en la web. https://blog.aulaformativa.com/definicion-usos-comunes-formato-svg-web/.

[19] S. Genaim and J. Domenéch. The EasyInterface Framework. http://github.com/abstools/easyinterface.

[20] S. Genaim, J. Domenéch, Johnsen E.B., and Schlatte R. EasyInterface: A Toolkit for Rapid Development of GUIs for Research Prototype Tools. In M. Huisman and J. Rubin, editors, *Fundamental Approaches to Software Engineering*, pages 379–383, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.

[21] GoJS. Introduction to GoJS Diagramming Components. https://gojs.net/latest/intro/.

[22] Gràffica. ¿Qué es y qué ventajas tiene el formato SVG? https://graffica.info/formato-svg-ventajas/.

[23] Nidhugg. Nidhugg. https://github.com/nidhugg/nidhugg.

[24] A. Ostrovski. SVGTree. https://github.com/slowli/SVGTree.

[25] Plotly. Tree-plots in Python. https://plotly.com/python/tree-plots/.

[26] Raphael. An Intro to Raphaël. http://raphaeljs.com/.

[27] tjklemz. SVG alternatives? https://stackoverflow.com/a/7537085/14461609.

[28] All trees. tree-svg. https://github.com/all-trees/tree-svg.

[29] J. Wick. Ráster vs. imágenes vectoriales - ¿Cuál es la diferencia? https://www.stickermule.com/es/blog/raster-vs-vector-images.

[30] Wikipedia, The Free Encyclopedia. Image file formats. https://en.wikipedia.org/wiki/Image_file_formats.

[31] Wikipedia, The Free Encyclopedia. Model checking. https://en.wikipedia.org/wiki/Model_checking.

[32] Wikipedia, The Free Encyclopedia. Raster graphics. https://en.wikipedia.org/wiki/Raster_graphics.

[33] Wikipedia, The Free Encyclopedia. Vector graphics. https://en.wikipedia.org/wiki/Vector_graphics.

[34] T. Yip. Comparing SVG and PNG File Sizes. https://vecta.io/blog/comparing-svg-and-png-file-sizes.

Apéndice A

Material adicional e instalación de la herramienta

En este trabajo se ha utilizado la librería *D3* para *JavaScript*, que puede encontrarse en https://github.com/d3/d3. Concretamente, se ha utilizado el paquete *hierarchy* y el paquete *selection-multi*, que pueden visualizarse en https://github.com/d3/d3-hierarchy y en https://github.com/d3/d3-selection-multi, respectivamente.

Adicionalmente, para realizar la integración entre SYCO y el visualizador desarrollado en el trabajo se ha usado EasyInterface, que puede encontrarse en [19, 20].

A.1. Instalación de la herramienta

La instalación del visualizador es muy sencilla, lo cual es una de las grandes ventajas de haber utilizado *JavaScript*, como ya se indicó en la sección 4.2. Si deseamos utilizar el visualizador integrado con SYCO, la mejor opción es seguir los pasos detallados en el capítulo 7. Sin embargo, es posible que deseemos visualizar un árbol que ya tengamos generado anteriormente. En ese caso, es necesario seguir los siguientes pasos:

Paso 1 Instalar **Node.js** en el sistema, siguiendo el tutorial detallado en https://www.geeksforgeeks.org/installation-of-node-js-on-linux/.

Paso 2 Instalar *HTTP-Server* siguiendo los pasos descritos en https://www.npmjs.com/package/http-server.

Paso 3 Navegar a la carpeta donde se encuentra el fichero *index.html* que queremos ejecutar y ejecutar el comando

```
$ http-server -p 5000
```

donde el *flag -p* indica el puerto en el que queremos crear el servidor, en este caso será el puerto 5000.

Paso 4 Ir al navegador deseado y en la barra de búsqueda escribir *localhost:5000*.

Tras haber ejecutado estos pasos, en el navegador debería mostrarse algo similar a la figura 1.1 (dependerá del fichero de entrada utilizado).

Nótese que no basta con abrir el fichero *index.html* con el navegador deseado, puesto que para abrir el JSON se está utilizando una función de *D3* para leer ficheros JSON que únicamente funciona usando el protocolo HTTP. Por lo tanto, la única opción es crear un servidor local y ejecutar de esa manera, siguiendo los pasos indicados anteriormente.

Apéndice B

Recorridos del árbol

En la implementación de este trabajo de final de grado se han realizado diversos algoritmos de recorrido de árboles para lograr que funcionen los comandos detallados en la sección 6.1.3. Estos recorridos tienen ideas relativamente sencillas y muy similares entre sí, pero deben tener en cuenta los restantes recorridos, de forma que podamos ejecutar todos los recorridos en diversos órdenes sin que se produzca ningún fallo. Esto complicó ligeramente la programación de los algoritmos.

El objetivo de este apéndice es explicar todos los algoritmos utilizados para recorrer los árboles, junto con los diversos problemas a los que nos hemos enfrentado en el proceso de desarrollo.

B.1. Atributos del nodo

En esta sección vamos a explicar brevemente la estructura de árbol que tenemos en *D3*, siguiendo lo ya explicado en la sección 5.1.3. En general, *D3* toma el objeto JSON generado en dicha sección y genera un nuevo objeto, que apunta a la raíz con información adicional e independiente del anterior. En concreto, contiene los atributos (aunque es posible añadirle otros atributos que nosotros consideremos útiles):

- ***children***: lista de referencias a los nodos que son hijos de dicho nodo. De este atributo es de donde *D3* va a tomar los hijos que mostrará. Así, no podemos utilizar únicamente este atributo, puesto que si no mostramos ciertos hijos, deberemos eliminarlos de esta lista, y si no los guardamos en alguna otra variable, los perderemos. Para salvar este escollo, cada nodo va a disponer desde el inicio con una nueva lista independiente de la lista *children* (en el sentido de que son dos *arrays* diferentes, que ocupan posiciones de memoria distintas) llamada *_children* que va a guardar referencias a todos los hijos del nodo, ya sean visibles o no.
- ***depth***: entero que indica la distancia de dicho nodo a la raíz. Este atributo será tenido

en cuenta para el algoritmo que utiliza *D3* para el posicionamiento de los nodos en el *canvas*, es decir, que si un nodo cambia de padre (por ejemplo, una hoja colapsada), deberá cambiar su profundidad en concordancia, pues de lo contrario se mostrará a mayor distancia de la usual de su nuevo padre, aunque la conexión entre ambos sea directa. Por lo tanto, utilizaremos un atributo adicional *realDepth* solamente en las hojas que nos indica la profundidad real de dicha hoja. En ocasiones el valor será igual a *depth*, y en otras será menor, si la hoja se encuentra colapsada.

- ***height***: entero que indica la distancia de dicho nodo a la hoja más lejana en su subárbol.

- ***parent***: referencia al nodo que es el padre actual de dicho nodo. Al igual que ocurre con *depth*, debemos actualizar este atributo en las hojas para poder hacer un recorrido natural desde la hoja hasta la raíz. Por lo tanto, necesitamos guardar el padre real en las hojas en un atributo auxiliar que hemos llamado *realParent*. De nuevo, en muchas ocasiones será el padre real, pero en otras será un ancestro suyo.

  Más aún, necesitamos modificar el atributo *parent* (y por lo tanto disponer del atributo *realParent*), puesto que de lo contrario tendríamos que ir nodo a nodo en el camino de la hoja a la raíz comprobando si dicho nodo está en el *array children* para saber cual es el padre actual, algo que es muy costoso computacionalmente.

De forma adicional a los atributos ya mencionados anteriormente, cada nodo va a disponer además de un atributo booleano ***visible***, que nos va a indicar que dicho nodo es visible. En general, un nodo será visible si se encuentra en el *array children* del padre. Sin embargo, es mucho menos costoso tener un atributo booleano que lo indique. Además, en el caso especial de los caminos colapsados en las hojas, será necesario comprobar que está en el *array children* del padre "ficticio".

Además, los nodos hoja van a disponer de un atributo booleano ***collapsed***, que nos va a indicar que dicha hoja ha sido colapsada. Necesariamente cuando este atributo está a verdadero el nodo será visible y estará colapsado, por lo que sus atributos *realDepth* y *realParent* pueden ser diferentes de *depth* y *parent* respectivamente (es posible que no lo sean, puesto que si por ejemplo colapsamos un camino desde el padre anterior a una hoja, el camino estará colapsado y estos atributos serán iguales). Con este atributo podremos saber que dicha hoja se encuentra (posiblemente) en el *array children* de un nodo que no es su padre real.

Para manejar el comportamiento de las trazas ocultas, añadiremos además dos atributos booleanos ***cut*** y ***faded***. El primero de ellos nos indica si un nodo está cortado, en el sentido de que no debe ser mostrado porque no está en el intervalo elegido por el usuario, mientras que el segundo solamente tiene relevancia para saber si un nodo cortado debe ser mostrado con una opacidad baja. Inicialmente estos atributos se encuentran a *false*, puesto que por defecto se muestran todas las trazas al inicio, con lo cual no hay nodos "cortados".

B.2. Algoritmo de colapsado de caminos

En primer lugar, el algoritmo de colapsado recibe como entrada un nodo, que será el nodo sobre el que se ha utilizado el comando *Alt* + *Click*, o bien será la raíz en caso de haber utilizado el botón que colapsa un camino o todos los caminos.

El algoritmo de colapsado de caminos sigue los siguientes pasos

Paso 1 En el caso en el que el nodo recibido sea una hoja, no hacemos nada, puesto no hay nada que colapsar. Devolvemos *false*.

Paso 2 En caso contrario, buscamos la última hoja (la de más a la derecha en el subárbol) que sea visible. En caso de que no exista tal hoja, devolveremos *false*, y no haremos nada más.

Paso 3 Asumiendo que en el paso anterior logramos encontrar una hoja visible, debemos subir en el camino hacia la raíz (siempre teniendo en cuenta que tenemos como límite el nodo sobre el que el usuario hizo *click*), para encontrar el primer ancestro que no deba ser colapsado, es decir, aquel ancestro que tenga mínimo algún hijo (o descendiente, porque puede ser una hoja colapsada) visible diferente del que se encuentra en el camino que estamos explorando.

Paso 4 Una vez tenemos el ancestro, colapsamos todo el camino desde el ancestro a la hoja.

Paso 5 Hasta el momento solo hemos modificado atributos del árbol, pero no hemos actualizado su apariencia, por lo que lo hacemos, ocultando todos los nodos del camino en el ancestro.

Paso 6 Centramos el árbol, mostrando en el medio al ancestro.

Paso 7 Devolvemos *true*.

El algoritmo que implementa los pasos anteriores es

```
function hidePath(d) {
    if (!d._children) return false;
    let leaf = findLastVisibleLeaf(d);
    // Leaf that has been found
    if (leaf) {
        let ancestor = findFirstBifurcation(leaf, d);
        makeInvisible(leaf, ancestor);
        // All the nodes need to be hidden at the position of ancestor
        update(ancestor);
        centerNode(ancestor);
```

```
        return true;
    }
    return false;
}
```

A continuación vamos a detallar los pasos 2, 3 y 4 introducidos anteriormente.

B.2.1. Encontrar la última hoja visible

El algoritmo utilizado para este fin es recursivo. Antes de comenzar, no exploramos todos aquellos nodos que estén cortados y que no estén difuminados.

Evidentemente, el caso base es el de una hoja. Puesto que buscamos que la hoja sea visible, en caso de que sea visible (atributo *visible* está a *true*) devolvemos dicha hoja, y en caso de que no lo sea devolvemos *null*.

El caso recursivo también es bastante evidente. En el caso en el que nos encontremos en un nodo interno, llamamos a la misma función para cada hijo del *array* _*children* en orden inverso. En el momento en el que uno de ellos nos devuelva un nodo no nulo, mandaremos dicho nodo hacia arriba, y terminaremos de explorar el nodo. En el caso en el que recorramos todos los nodos sin encontrar un nodo no nulo, devolveremos *null*, que significará que dicho subárbol no tiene hojas visibles.

El código *JavaScript* que implementa este algoritmo es el siguiente:

```
function findLastVisibleLeaf(node) {
    function findLastVisibleLeaf(node) {
    if (node.cut && !node.faded) return null;
    // Leaf
    if (!node._children) {
        if (node.visible) return node;
        else return null;
    }
    // Internal node
    for (let i = node._children.length - 1; i >= 0; --i) {
        let ch = findLastVisibleLeaf(node._children[i]);
        if (ch) return ch;
    }
    // If we have reached the end, we don't have visible leaves in the subtree
    // generated by the given node
    return null;
}
}
```

B.2.2. Búsqueda del primer punto de bifurcación

El algoritmo que realiza esta tarea es recursivo. El caso base se divide en dos casos. El primero es que es posible que hayamos alcanzado la raíz, en cuyo caso no podremos seguir buscando ancestros, y este será el ancestro que devolvamos. El segundo caso es que hayamos alcanzado el límite fijado (el nodo sobre el que se ha hecho *click*), en cuyo caso no debemos seguir buscando ancestros, pues este va a ser el único nodo que va a quedar visible.

Una vez descartado estos dos casos, el caso recursivo es claro: debemos comprobar que dicho nodo no tiene hijos visibles diferentes del hijo que nos lleva a la hoja considerada, y que tampoco tiene otros caminos colapsados. Para ello, dividimos en dos pasos.

En primer lugar, recorremos todo el *array* _*children* en búsqueda de hijos directos que sean nodos internos o de hijos directos que sean hojas no colapsadas y que además sean visibles. Si lo son, entonces este será el ancestro buscado.

En segundo lugar, tras haber recorrido todo el *array* anterior sin éxito, recorremos todo el *array children* en búsqueda de hojas colapsadas en dicho nodo. Si encontramos alguna, entonces este será el ancestro buscado.

Si tras haber hecho estos dos pasos no hemos encontrado ningún nodo que cumpla estos requisitos, entonces podemos continuar hacia arriba en el camino.

El algoritmo considerado es el siguiente:

```
function findFirstBifurcation(node, limit) {
    let parent = node.parent;
    if (!parent) return node; // There is no parent (we have reached the root)
    if (node === limit) return limit; // If we have reached the limit, we stop

    let index = parent._children.indexOf(node);
    for (let i = 0; i < parent._children.length; ++i) {
        // If children is visible and (is internal node or is a not-collapsed
        // leaf, parent is the ancestor we are looking for
        if ((i != index && parent._children[i].visible &&
                parent._children[i]._children) ||
            (i != index && parent._children[i].visible &&
                !parent._children[i]._children &&
                !parent._children[i].collapsed)) return parent;
    }

    // It is also possible that the parent has collapsed children. These
    // children (leaves), won't be at _children array
    if (parent.children) {
        index = parent.children.indexOf(node);
        for (let i = 0; i < parent.children.length; ++i) {
```

```
            if (i != index && !parent.children[i]._children &&
                parent.children[i].collapsed)
                return parent;
        }
    }
    // If we haven't found a node at this point, the ancestor can be hidden
    return findFirstBifurcation(parent, limit);
}
```

B.2.3. Colapsar el camino entre el ancestro y la hoja

El algoritmo que realiza esta tarea es recursivo . El caso base es que hayamos alcanzado el límite (el ancestro encontrado en la sección B.2.2). En ese caso, no hay nada que hacer. En caso contrario, eliminaremos el nodo actual del *array children* del padre y haremos dicho nodo invisible, y continuaremos con el padre. Por último, en caso de que el nodo sea una hoja, una vez hemos subido ya hacia arriba y modificado todos los atributos de los restantes nodos, restauramos los atributos *depth*, *collapsed* y *parent* para que el árbol se encuentre en el estado original.

El código *JavaScript* encargado de esta tarea es el siguiente:

```
function makeInvisible(node, limit) {
    if (!node.parent || node === limit) return; // Root or reached limit
    else {
        // Node is inside the parent children list. To make it invisible, we
        // need to remove it from the list
        let index = node.parent.children.indexOf(node);
        node.parent.children.splice(index, 1);
        if (node.parent.children.length == 0) node.parent.children = null
        node.visible = false;

        makeInvisible(node.parent, limit);
        if (node.collapsed) {
            node.collapsed = false;
            node.depth = node.realDepth;
            node.parent = node.realParent;
        }
    }
}
```

B.3. Algoritmo de colapsado de caminos en hojas

El algoritmo de esta sección es muy similar al de la sección B.2, con ligeras modificaciones que se van a detallar a continuación.

De nuevo, este algoritmo recibe como entrada un nodo, que será el nodo sobre el que se ha utilizado el comando *Ctrl + Click*.

El algoritmo de colapsado de caminos en hojas sigue los siguientes pasos:

Paso 1 En el caso en el que el nodo recibido sea una hoja, no hacemos nada, puesto que no hay nada que colapsar en la hoja.

Paso 2 En caso contrario, buscamos la última hoja (la de más a la derecha en el subárbol) que sea visible y que no esté colapsada. En caso de que no exista tal hoja, devolveremos *false*

Paso 3 Asumiendo que en el paso anterior logramos encontrar una hoja visible y no colapsada, debemos subir en el camino hacia la raíz al igual que en el **Paso 3** de la sección B.2.

Paso 4 Una vez tenemos el ancestro, colapsamos todo el camino desde el ancestro a la hoja y agregamos como hijo del ancestro a dicha hoja.

Paso 5 En caso de que realmente hayamos colapsado, es decir, que el nuevo padre de la hoja no sea el antiguo, actualizamos la apariencia del árbol, ocultando todos los nodos del camino en el ancestro, excepto la hoja, y centramos el árbol en el ancestro.

Paso 6 En el caso en el que hayamos realmente colapsado algún camino, devolvemos *true*. En caso contrario, retornamos al **Paso 2**.

El algoritmo que implementa el colapsado de caminos en hojas es

```
function collapsePathToLeaf(d) {
    if (d._children) { // Not leaf
        let leaf, ancestor;
        do {
            leaf = findFirstNonCollapsedPathToLeaf(d);
            if (leaf) {
                ancestor = findFirstBifurcation(leaf, d);
                collapseLeafPath(leaf, leaf, ancestor);
                if (leaf.realParent !== ancestor) {
                    update(ancestor);
                    centerNode(ancestor);
```

```
                }
            }
        } while (leaf && leaf.realParent === ancestor)
    }
}
```

A continuación vamos a detallar los pasos 2 y 4 introducidos anteriormente.

B.3.1. Encontrar la última hoja visible y no colapsada

El algoritmo utilizado para este fin es recursivo. Antes de comenzar a hacer comprobaciones sobre los nodos, no exploramos aquellos nodos que están cortados (nótese que no permitimos colapsar los caminos en hojas para los que están difuminados).

Evidentemente, el caso base es el de una hoja. Puesto que buscamos que la hoja sea visible y no esté colapsada, en caso de que sea visible (atributo *visible* está a *true*) y no colapsada (atributo *collapsed* está a false) devolvemos dicha hoja, y en caso de que no lo sea devolvemos *null*.

El caso recursivo también es bastante evidente. En el caso en el que nos encontremos en un nodo interno, llamamos a la misma función para cada hijo del *array* _*children* en orden inverso. En el momento en el que uno de ellos nos devuelva un nodo no nulo, mandaremos dicho nodo hacia arriba, y terminaremos de explorar el nodo. En el caso en el que recorramos todos los nodos sin encontrar un nodo no nulo, devolveremos *null*, que significará que dicho subárbol no tiene hojas visibles.

El código *JavaScript* para el algoritmo es el siguiente:

```
function findFirstNonCollapsedPathToLeaf(node) {
    if (node.cut) return null;
    // Leaf
    if (!node._children) {
        if (!node.collapsed && node.visible) // Non-collapsed and visible
            return node;
        else return null;
    }

    // Internal node
    for (let i = 0; i < node._children.length; ++i) {
        let ch = findFirstNonCollapsedPathToLeaf(node._children[i]);
        if (ch) return ch;
    }
    // If we have reached the end, we don't have non collapsed leaves in the
    // subtree generated by the given node
```

```
    return null;
}
```

B.3.2. Colapsar el camino entre el ancestro y la hoja en la hoja

El algoritmo que se va a detallar a continuación es un algoritmo recursivo. Antes de chequear si estamos en el caso base o no, cambiamos el atributo *visible* a false, puesto que dicho nodo no va a ser visible, a excepción de la hoja que ya se cambiará en su momento.

El caso base se alcanza cuando el padre del nodo en el que nos encontramos actualmente es el ancestro que hemos encontrado en el paso 3. En ese caso, necesitamos actualizar las conexiones, y para ello lo que vamos a hacer es sustituir en el padre a su verdadero hijo por la hoja que estamos colapsando. Una vez hemos hecho eso, necesitamos actualizar los atributos de la hoja para que sepa cual es su nuevo padre y cual es su nueva profundidad en el árbol, y le indicamos que está colapsada (atributo *collapsed* a *true*) y que es visible (atributo *visible* a *true*).

Para el caso recursivo, simplemente buscamos al nodo en el *array* de *children* del padre y lo eliminamos, y continuamos hacia arriba en el árbol.

El código *JavaScript* es el siguiente:

```
function collapseLeafPath(node, leaf, ancestor) {
    // Each node in the path will be invisible (except from leaf and ancestor)
    node.visible = false;
    // If we have reached the node that is the previous ancestor of ancestor of
    // the leaf, we need to update the connections
    if (node.parent === ancestor) {
        if (node.parent.children) {
            let index = node.parent.children.indexOf(node);
            if (index != -1) node.parent.children[index] = leaf;
            else node.parent.children.push(leaf);
        } else node.parent.children = [leaf];
        // Update leaf attributes so that it is printed correctly
        leaf.parent = ancestor;
        leaf.depth = ancestor.depth + 1;
        leaf.collapsed = true;
        leaf.visible = true;
    } else {
        if (node.parent.children) {
            // We need to collapse the path, so children should be erased from
            // parent
            let index = node.parent.children.indexOf(node);
            if (index != -1) node.parent.children.splice(index, 1);
```

```
            if (node.parent.children.length == 0) node.parent.children = null;
        }
        collapseLeafPath(node.parent, leaf, ancestor);
    }
}
```

B.4. Algoritmo de expansión de caminos

En primer lugar, el algoritmo de expansión recibe como entrada un nodo, que será el nodo sobre el que se ha utilizado el comando *Shift* + *Click*, o bien será la raíz en caso de haber utilizado el botón que expande un camino o todos los caminos.

El algoritmo de expansión de caminos sigue los siguientes pasos:

Paso 1 En el caso en el que el nodo recibido sea una hoja no hacemos nada, puesto que no hay nada que expandir. Devolvemos *false*.

Paso 2 En caso contrario, buscamos la primera hoja (la de más a la izquierda en el subárbol) que sea invisible, o bien que sea visible pero esté colapsada. En caso de que no exista tal hoja, devolveremos *false*.

Paso 3 Asumiendo que en el paso anterior logramos encontrar una hoja invisible, o bien visible pero colapsada, deberemos asegurarnos de que esta hoja aunque esté colapsada tenga nodos intermedios hasta su padre real que están colapsados. Si no los tuviera, la marcamos como no colapsada (atributo *collapsed* a *false*) y volvemos al **Paso 2**, puesto que en realidad no hemos cambiado nada de la apariencia en el árbol.

Paso 4 Una vez tenemos una hoja invisible o bien visible pero colapsada, y de forma que no está colapsada en su padre, diferenciamos dos casos:

- **Caso 1** Si la hoja estaba colapsada, entonces conocemos cual va a ser el ancestro (el padre falso de la hoja), y conocemos también el índice en el que debe ir el verdadero hijo en el camino a la hoja (pues si no utilizamos este índice, estaríamos cambiando de orden a los hijos).
- **Caso 2** Si la hoja no estaba colapsada, entonces necesitamos buscar el primer ancestro visible.

Paso 5 Una vez tenemos identificado al ancestro, necesitamos expandir el camino hasta el ancestro desde la hoja.

Paso 6 En caso de que realmente hayamos expandido, actualizamos la apariencia del árbol, ocultando todos los nodos del camino en el ancestro, excepto la hoja, y centramos el árbol en el ancestro.

Paso 7 En el caso en el que hayamos expandido algún camino, devolvemos *true*. En caso contrario, devolveremos *false*.

El algoritmo que implementa la expansión de caminos es

```
function showPath(d) {
    // Leaf should be discarded
    if (!d._children) return false;
    let leaf;
    while (1) {
        leaf = findFirstNotVisibleLeaf(d);
        if (leaf) {
            if (leaf.collapsed && leaf.realDepth == leaf.depth) {
                leaf.collapsed = false;
                continue;
                // Collapsed leaves that are correctly positioned should
                // not be taken into account
            }
        }
        break;
    }

    if (leaf) {
        let ancestor;
        if (leaf.collapsed) {
            leaf.collapsed = false;
            leaf.visible = false;
            ancestor = leaf.parent;
            // Index is important to avoid shifting positions on parent
            let index = leaf.parent.children.indexOf(leaf);
            leaf.parent = leaf.realParent;
            leaf.depth = leaf.realDepth;

            expandPath(leaf, ancestor, true, index);
        } else {
            ancestor = findVisibleAncestor(leaf, d);
            expandPath(leaf, ancestor, false);
        }
        update(ancestor);
        centerNode(ancestor);
        return true;
    }
```

```
    return false;
}
```

Las funciones utilizadas en el código anterior, se detallan a continuación.

B.4.1. Encontrar la primera hoja invisible o visible pero colapsada

El algoritmo que resuelve este problema es recursivo. Antes de comenzar a hacer comprobaciones sobre los nodos, eliminamos el caso en el que los nodos están cortados y no están difuminados, puesto que en ese caso esos nodos no deben ser considerados (nótese que no pasaría nada por explorar estos nodos, pues son invisibles, pero estaríamos derrochando recursos innecesariamente).

Es claro que el caso base es el de una hoja. Como buscamos que sea invisible o bien que sea visible pero esté colapsada, en caso de que sea invisible (atributo *visible* está a *false*) o bien sea visible (atributo *visible* está a *true*) pero esté colapsada (atributo *collapsed* está a *true*), devolvemos dicha hoja. En caso contrario, devolvemos *null*.

El caso recursivo corresponde a los nodos internos. En ellos, recorremos su *array* _*children* de izquierda a derecha en busca de que algún subárbol que tenga como raíz a alguno de sus hijos tenga una hoja con esas características. Si lo encontramos, lo devolvemos hacia arriba, y si no devolvemos *null* de nuevo, que indica que dicho subárbol no tiene hojas con esas condiciones.

El algoritmo, implementado en *JavaScript* es el siguiente:

```
function findFirstNotVisibleLeaf(node) {
    if (node.cut && !node.faded) return null;
    // Leaf
    if (!node._children) {
        if (!node.visible || (node.visible && node.collapsed)) return node;
        else return null;
    }
    // Internal
    for (let i = 0; i < node._children.length; ++i) {
        let ch = findFirstNotVisibleLeaf(node._children[i]);
        // If there is any, we return it. If not, we continue with the loop
        if (ch != null) return ch;
    }
    // There is not any coincidence in this subtree
    return null;
}
```

B.4.2. Búsqueda del primer ancestro visible

El algoritmo para encontrar el primer ancestro visible desde una hoja es sencillo. Disponemos, al igual que en todos los algoritmos presentados anteriormente para buscar ancestros de un límite, que será el caso base, junto con el hecho de haber alcanzado la raíz. También tenemos en el caso base la comprobación de que el nodo en el que nos encontramos actualmente es visible (atributo *visible* está a *true*). En ese caso, este es el ancestro que estamos buscando.

El caso recursivo consiste básicamente en continuar hacia arriba en el árbol buscando algún nodo que cumpla el caso base.

El código en *JavaScript* para este problema es:

```
function findVisibleAncestor(node, limit) {
    let parent = node.parent;
    if (!parent) return node; // There is no parent (we have reached the root)
    if (node === limit) return limit; // If we have reached the limit, we stop

    if (node.visible) return node;

    // If node is not visible, we go up in the path
    return findVisibleAncestor(parent, limit);
}
```

B.4.3. Expansión del camino entre el ancestro y la hoja

El algoritmo que realiza esta tarea es recursivo. El caso base es que hayamos alcanzado el ancestro, en cuyo caso no hay nada que hacer.

En el caso recursivo, chequeamos que el nodo no esté en el *array children* del padre. Si ya estaba, entonces no hay que hacer nada. Si no estaba, entonces hay que diferenciar dos casos

Caso 1 Si el camino estaba colapsado, y hemos alcanzado el ancestro, hay que sustituir el verdadero hijo por la hoja, y para eso utilizamos el índice donde estaba la hoja.

Caso 2 Si el camino no estaba colapsado, hay que insertar el hijo en el mismo índice que se encuentra en el *array* de *_children*, pues si estamos expandiendo este camino todos los que se encuentran a su izquierda están ya expandidos, y por lo tanto conciden los *arrays children* y *_children* desde 0 hasta el índice menos 1.

Por último, indicamos que el nodo es visible, y continuamos hacia arriba en el árbol.

El algoritmo para este caso es:

```
function expandPath(node, ancestor, collapsed, index = null) {
    if (node === ancestor) return;
    let parent = node.parent;
    parent.children = parent.children || [];
    if (!parent.children.includes(node)) {
        // Special case: index should be taken into account to add the real
        // children in the correct position
        if (collapsed && parent === ancestor)
            parent.children[index] = node;
        else {
            // If we are expanding this path, children and _children array are
            // the same from 0 to index - 1
            let index = parent._children.indexOf(node); // Not -1
            node.parent.children.splice(index, 0, node);
        }
    }
    node.visible = true;

    expandPath(node.parent, ancestor, collapsed, index);
}
```

Apéndice C

Generación del fichero JSON en SYCO

El objetivo de este apéndice es explicar brevemente cómo se genera el fichero JSON basado en la salida de SYCO que va a ser interpretado por el visualizador, cuya estructura se encuentra dada en el listing 5.2. La creación de este fichero JSON fue implementada en Prolog por el profesor Miguel Isabel, miembro del grupo de investigación Costa y codirector de este Trabajo.

En primer lugar, es importante resaltar que, como ya se indicó en la sección 5.1.2, la información de los estados se va generando a medida que se va explorando el árbol, al contrario que la restante información (información de aristas, de ejecuciones y de conjuntos), que se puede obtener tras finalizar la exploración.

Por lo que respecta a la información referente a los estados, ésta ya estaba disponible con anterioridad, salvo que no se utilizaba. Así pues, ahora durante el procesamiento se toma la información sobre los *fields* de cada actor (atributos de los actores) y se añaden como hechos en predicados para después ser añadidos al fichero JSON.

La generación del JSON se divide en **dos fases**, que se realizan una a continuación de la otra. El paso de una a la otra se hace mediante el uso de la técnica conocida como negación por fallo, muy extendida y utilizada en Prolog.

En la primera fase obtenemos toda la información que necesitamos escribir en el JSON, agregándola como hechos a ciertos predicados (excepto la referente a los estados, que como ya se ha dicho se hace durante la exploración del programa a ejecutar), y en la segunda tomamos todos esos hechos y los escribimos con la estructura adecuada en el JSON (es decir, agregando los signos de puntuación adecuados, y los saltos de línea para favorecer la legibilidad).

En la primera fase también se puede incorporar lógica para identificar algunas propiedades de ejecución y/o de estado. Por ejemplo, suponiendo que disponemos de un actor *Controller* que dispone de una tarea cuyo nombre es *ctrlHandleErrorMsg*, que se encarga de gestionar un caso de error, podemos chequear si en alguna ejecución y/o estado se ha ejecutado dicha tarea, para marcar la ejecución y/o el nodo como erróneos. Un ejemplo en el que se dispone

de actores *Controller* puede ser, entre otros, el que se encuentra la figura 1.1

También en esta etapa se toma la información general de ejecución del programa por parte de SYCO, que se agregará al campo *info*. Recuérdese que esta información viene dada en formato HTML, por lo que hay que prestar especial atención a la misma (puesto que el visualizador no va a hacer ningún procesamiento específico). Además, es beneficioso aportar como información el número de ejecuciones que se han encontrado en el programa (tal y como se muestra en EasyInterface), puesto que puede utilizarse para la selección de variables dada en el apartado 6.3.3. Así, dado que no disponemos de esta información, es necesario calcularla. Para ello, simplemente obtenemos la longitud de la lista de trazas que se han encontrado en el programa, que nos devolverá trivialmente el número de ejecuciones exploradas.