UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Informática

Un entorno de modelado online para el model checker SPIN

Trabajo Fin de Grado

Grado en Ingeniería Informática

AUTOR/A: de Juan Achahuanco, Arturo Raúl

Tutor/a: Alpuente Frasnedo, María

Director/a Experimental: SAPIÑA SANCHIS, JULIA

CURSO ACADÉMICO: 2021/2022

Resum

Les eines de verificació de models existents ofereixen unes certes facilitats per a l'anàlisi i comprensió del procés de verificació de sistemes *hardware* i *software*. Aquest és el cas del verificador algorítmic SPIN, que permet a l'usuari inspeccionar detalls de la verificació formal a través d'interfícies gràfiques com les que ofereixen les eines iSpin o xSpin. Quan l'intent de verificació d'una propietat falla, aquests sistemes retornen, a manera de contraexemple, una traça de la seua execució que demostra que la propietat analitzada és falsa.

No obstant això, les interfícies actuals ofereixen facilitats d'inspecció limitades i el seu ús requereix uns coneixements avançats sobre el disseny de SPIN i de les diferents configuracions que el sistema pot suportar. L'objectiu d'aquest treball és desenvolupar una eina en línia que, a més de permetre a l'usuari modelar i verificar en SPIN les propietats temporals d'un sistema, suporte una exploració flexible i interactiva de les execucions del model checker. L'entorn estarà focalitzat a facilitar l'aprenentatge del funcionament de SPIN i la comprensió dels contraexemples generats, que poden ser visualitzats en forma gràfica.

L'eina permet a l'usuari interactuar amb aquesta representació per a ajudar-lo a comprendre les causes de les fallades de verificació. En primer lloc, s'ofereix una manera d'exploració global que permeta als usuaris obtindre una vista general del graf per a comprendre el seu funcionament a alt nivell. En segon lloc, l'eina suporta una manera d'exploració local que permeta al programador interactuar amb el verificador. A més de permetre l'escrutini dels estats de la traça, el sistema desenvolupat ofereix una interfície de consulta que permet filtrar i destacar regions específiques del graf..

Paraules clau: mètodes formals, verificació automatitzada, visualització de traces d'execució, comprovador de models SPIN

Resumen

Las herramientas de verificación de modelos *model checking* existentes ofrecen ciertas facilidades para el análisis y comprensión del proceso de verificación de sistemas *hardware* y *software*. Este es el caso del verificador algorítmico SPIN, que permite al usuario inspeccionar detalles de la verificación formal a través de interfaces gráficas como las que ofrecen las herramientas iSpin o xSpin. Cuando el intento de verificación de una propiedad falla, dichos sistemas devuelven, a modo de contraejemplo, una traza de su ejecución que demuestra que la propiedad analizada es falsa.

Sin embargo, las interfaces actuales ofrecen facilidades de inspección limitadas y su uso requiere unos conocimientos avanzados sobre el diseño de SPIN y de las diferentes configuraciones que el sistema puede soportar. El objetivo de este trabajo es desarrollar una herramienta online que, además de permitir al usuario modelar y verificar en SPIN las propiedades temporales de un sistema, soporte una exploración flexible e interactiva de las ejecuciones del *model checker*. El entorno estará focalizado en facilitar el aprendizaje del funcionamiento de SPIN y la comprensión de los contraejemplos generados, que pueden ser visualizados en forma gráfica.

La herramienta permite al usuario interactuar con dicha representación para ayudarle a comprender las causas de los fallos de verificación. En primer lugar, se ofrece un modo de exploración global que permite a los usuarios obtener una vista general del grafo para comprender su funcionamiento a alto nivel. En segundo lugar, la herramienta soporta un modo de exploración local que permite al programador interactuar con el verificador.

Además de permitir el escrutinio de los estados de la traza, el sistema desarrollado ofrece una interfaz de consulta que permite filtrar y destacar regiones específicas del grafo.

Palabras clave: métodos formales, verificación automatizada, visualización de trazas de ejecución, comprobador de modelos SPIN

Abstract

Model checking tools currently offer several facilities for the analysis and understanding of the verification process of hardware systems and software systems. This is the case of the algorithmic verifier SPIN, which allows the user to inspect specific details of the formal verification by means of a graphical interface such as the iSpin and xSpin ones. When the attempt to verify a property fails, these systems return, as a useful counterexample, an execution trace showing that the analyzed property is false.

However, the current interfaces offer limited inspection facilities, and moreover, their use requires advanced knowledge of the SPIN design and of the configurations it supports The objective of this work is to develop an online tool that, besides supporting the user in modeling and verifying the temporary properties of a system using SPIN, supports the interactive exploration of the model checker executions un a flexible way. The environment is aimed at helping non-expert users learn how to use SPIN and to understand the delivered counterexamples, which can be graphically displayed.

The user is allowed to interact with this graphical representation to understand the causes of a verification failure. First, a global inspection mode has been implemented that provides users with a general, high-level overview of the graph. Second, a local inspection mode facilitates the interactive use of the verifier. Besides allowing the trace states to be scrutinized, a query interface is also provided that allows the user to filter and highlight specific regions of the graph.

Key words: formal methods, automated verification, trace visualization, SPIN model checker

Índice general

Índice de figuras

Índice de tablas

CAPÍTULO 1
Introducción

En la actualidad, la gran mayoría de la población posee un teléfono inteligente, ha navegado por Internet o incluso ha utilizado los recursos de cómputo de alguna máquina con pantalla. Y en todos estos casos, las interfaces gráficas se han utilizado para que nosotros, las personas, podamos utilizarlos y que realicen alguna tarea determinada. En el ámbito de la informática, las encargadas de facilitar la comunicación máquina-persona son las interfaces gráficas de usuario (IGU).

Por un lado, la IGU es un programa cuya finalidad es facilitar la comunicación entre el usuario y la máquina a través de la representación visual de la información del sistema, así como las diferentes interacciones posibles. UNo de los principales propósitos de este tipo de interfaces es su facilidad de uso. De hecho, este tipo de interfaces aparecieron después de las interfaces de línea de comandos, o por sus siglas en inglés CLI [32], que aparecieron en la época de las primeras computadoras. Con estas, los usuarios podían escribir diferentes instrucciones que el ordenador podía interpretar y ejecutar una función. A día de hoy, se pueden seguir utilizando, pero a lo largo de la historia se fueron sustituyendo por las interfaces gráficas, gracias a la modernización de los sistemas operativos.

No fue hasta que empezaron a surgir los primeros estándares de software que que empezaron a alcanzar importancia las interfaces gráficas de calidad. Uno de los temas que tratan los estándares de calidad de software es la usabilidad. Esta característica y las interfaces gráficas están sumamente relacionadas porque se utiliza para medir la capacidad de un sistema de ser entendido, aprendido, usado y resultar atractivo para el usuario.

No obstante, a un usuario promedio que realiza tareas de ofimática le sería complicado empezar a utilizar una aplicación con CLI, porque en este tipo de aplicaciones es necesario conocer comandos específicos para realizar tareas en él. Sin embargo, en el ámbito en el que nos encontramos, los estudiantes del grado en ingeniería informática debemos estar familiarizados con estos entornos y en algún momento el alumnado tendrá que utilizar este tipo de interfaces. Por ejemplo, una de las aplicaciones basadas en terminal que se ha utilizado en el grado es el *model checker* Spin [22]. Los *model checkers* sirven para verificar la seguridad y correcto funcionamiento de sistemas críticos industriales [25] y el hecho de que esté basado en terminal quiere decir que los resultados que genera se mostrarán como una ristra de texto, lo que puede dificultar el estudio de los resultados del sistema modelado.

1.1 Motivación

La idea original de este trabajo surge a partir de la inspiración de cómo facilita la comprensión y análisis de la información la herramienta formal Anima [2], un sistema interactivo de exploración de trazas de ejecución de los modelos formales escritos en el lenguaje de altas prestaciones Maude [9]. Una de las principales características de esta herramienta es la posibilidad de mostrar los diferentes caminos que puede tomar el sistema modelado en forma de árboles de decisión. Cada nodo de este árbol que tiene más de un descendiente pone en evidencia que, desde ese estado, el sistema puede ejecutar más de una instrucción.

Maude es un lenguaje de verificación formal que se puede comparar con el que utiliza Spin para modelar los sistemas, llamado Promela [22]. Cabe destacar que Spin es el que se encarga de recorrer todos los caminos del sistema y, si una propiedad no se cumple, genera una traza de contraejemplo. Los contraejemplos se utilizan para comprender dónde ha surgido un error en el sistema modelado. El problema es que, al ser SPIN una aplicación de consola, todos los resultados se muestran en formato texto y esto puede resultar difícil a la hora de analizarlos y comprender el origen de los errores. Los contraejemplos son una lista de estados y, de forma similar a la forma en que Anima muestra los estados, se quiere conseguir una visualización gráfica e interactiva de los contraejemplos.

Por otro lado, la manera clásica de interactuar con Spin es a través de diferentes parámetros cuyas combinaciones pueden resultar muy difíciles de realizar y comprender. Es cierto que actualmente existen diferentes interfaces gráficas para Spin pero resultan muy complicadas para usuarios poco expertos.

1.2 Objetivos

El objetivo principal de este proyecto es conseguir una aplicación sencilla e intuitiva en la que el usuario no requiera conocimientos avanzados del *model checker* Spin. Siguiendo esta filosofía se considera que realizar una aplicación web es más accesible a para todos los usuarios que utilizar una aplicación de escritorio. Por lo tanto, el proyecto se divide en dos partes: desarrollar un *frontend* y un *backend* para la solución software propuesta.

En primer lugar, el *frontend* se encarga de mostrar un entorno en el que el usuario puede escribir o importar código Promela y encontrar las propiedades dentro del código para su posterior verificación. Después, de la verificación, es posible que algunas propiedades den como resultado trazas de contraejemplos y el *frontend* se encarga de mostrar de forma comprensible estas trazas. Además, también la aplicación muestra, en cada estado del contraejemplo, toda la información relevante del sistema en dicho momento.

Por último, el *backend* es una aplicación de servidor que se encarga de procesar solicitudes y ejecutar Spin. Las solicitudes permitidas son las de visualizar la página web, verificar las invariantes del código Promela y procesar contraejemplos de las invariantes que han fallado.

1.3 Impacto esperado

El desarrollo de este trabajo tiene como objetivo que usuarios no expertos (en particular, alumnos de cursos de iniciación en los métodos formales) puedan verificar sus modelos y obtener resultados fáciles de entender. Las principales diferencias con respec-

to a utilizar directamente Spin es la facilidad de uso al realizar la verificación de modelos y analizar su comportamiento y que no se requiere conocimientos de todas las opciones que ofrece Spin. De hecho, a diferencia de otras interfaces gráficas previas desarrolladas para este *model checker*, no se requiere de una configuración avanzada.

En conclusión, se espera que el usuario que utilice esta herramienta pueda sentirse cómodo y que tenga la sensación de control. De este modo, verificar sistemas será una tarea sencilla de realizar e incluso se espera que pueda potenciar un mayor interés por en este tipo de herramientas.

1.4 Metodología

La metodología de desarrollo que se utiliza en este proyecto es la llamada metodología en cascada [27]. Por lo tanto, la aplicación pasará por las siguientes fases:

1. Análisis: Entender el problema actual y proponer un sistema que abarque diferentes soluciones. Esta fase se encarga también de obtener los requisitos que definen si el proyecto está terminado, además de documentar las pruebas necesarias para verificar si los requisitos están completados.

2. Diseño: Definir la arquitectura de comunicación entre el *frontend* y el *backend* y la estructura de la aplicación web, es decir, donde se van a colocar los diferentes elementos visuales.

3. Desarrollo: Generar las líneas de código correspondientes para cumplir los requisitos descritos.

4. Pruebas: Ejecutar y verificar si los requisitos programados han superado las pruebas. Si una de estas pruebas da un resultado negativo, se volverá a la fase de desarrollo.

5. Implantación: Instalación de la aplicación web en un servidor real. Esto conlleva instalar los programas necesarios para levantar la aplicación.

Esta metodología es un proceso lineal, en el que al finalizar una fase comienza la siguiente. Una característica de esta metodología es que al desarrollar una fase esta puede generar un fallo en alguna fase anterior, esto ocasiona que hay que regresar a esa fase y solucionar el problema. Para concluir, al final de la última fase se genera lo que comúnmente se conoce como entregable. Este entregable se considera el resultado final del trabajo.

CAPÍTULO 2
Contexto tecnológico

Este capítulo sirve para familiarizar al lector con los conceptos que se usan a lo largo de la memoria. Se hace una breve introducción al lenguaje Promela y al sistema Spin. Para finalizar se presenta un apartado que compara este trabajo con diferentes tecnologías relacionadas.

2.1 Promela

Como ya se ha comentado, Promela es un lenguaje de modelado de sistemas que se utiliza para la verificación de modelos, también llamado *model checking* en inglés. Una de las características principales del lenguaje es que se puede modelar sistemas distribuidos y comprobar si funcionan correctamente.

El Listado 2.1 muestra parte de la gramática que define el lenguaje Promela. Siguiendo esta estructura, se puede construir el modelo formal de un sistema. Una descripción de la gramática completa se puede consultar en el siguiente enlace: https://spinroot.com/spin/Man/grammar.html

```
spec  : module [ module ] *

module  : proctype  /* proctype declaration */
  | init    /* init process       */
  | never   /* never claim         */
  | trace   /* event trace         */
  | utype   /* user defined types */
  | mtype   /* mtype declaration   */
  | decl_lst  /* global vars, chans */

proctype: [ active ] PROCTYPE name '(' [ decl_lst ]')' [ priority ] [ enabler ]
     '{' sequence '}'

init  : INIT [ priority ] '{' sequence '}'

never : NEVER '{' sequence '}'

mtype : MTYPE [ '=' ] '{' name [ ',' name ] * '}'

decl_lst: one_decl [ ';' one_decl ] *

one_decl: [ visible ] typename  ivar [',' ivar ] *
  | [ visible ] unsigned_decl

typename: BIT | BOOL | BYTE | SHORT | INT | MTYPE | CHAN
  | uname /* user defined type names (see utype) */

```

```
active  : ACTIVE [ '[' const ']' ]  /* instantiation */

sequence: step [ ';' step ] *

step    : stmnt [ UNLESS stmnt ]
  | decl_lst
  | XR varref [',' varref ] *
  | XS varref [',' varref ] *

ch_init : '[' const ']' OF '{' typename [ ',' typename ] * '}'

varref  : name [ '[' any_expr ']' ] [ '.' varref ]

assign  : varref '=' any_expr /* standard assignment */
  | varref '+' '+'  /* increment */
  | varref '-' '-'  /* decrement */

stmnt : IF options FI   /* selection */
  | DO options OD   /* iteration */
  | FOR '(' range ')' '{' sequence '}'    /* iteration */
  | ATOMIC '{' sequence '}' /* atomic sequence */
  | D_STEP '{' sequence '}' /* deterministic atomic */
  | SELECT '(' range ')'  /* non-deterministic value selection */
  | '{' sequence '}'  /* normal sequence */
  | send
  | receive
  | assign
  | ELSE      /* used inside options */
  | BREAK     /* used inside iterations */
  | GOTO name
  | name ':' stmnt  /* labeled statement */
  | PRINT '(' string [ ',' arg_lst ] ')'
  | ASSERT expr
  | expr      /* condition */
  | c_code '{' ... '}'  /* embedded C code */
  | c_expr '{' ... '}'
  | c_decl '{' ... '}'
  | c_track '{' ... '}'
  | c_state '{' ... '}'

expr  : any_expr
  | '(' expr ')'
  | expr andor expr
  | chanpoll '(' varref ')' /* may not be negated */

const : TRUE | FALSE | SKIP | number [ number ] *
```

Listado 2.1: Fragmento de la gramática de Promela

Para empezar, un sistema se puede entender como un conjunto de subsistemas que se pueden comunicar entre sí. En Promela estos componentes se denominan procesos. Cada proceso tiene su propio comportamiento y se comunican gracias a los canales de mensajes. Para guardar información relevante de los procesos se utilizan las variables. Estas variables se clasifican como locales, mientras que las del tipo global son las que se definen fuera de un proceso y puede servir para sincronización y comunicación entre diferentes procesos.

A continuación, es necesario configurar el comportamiento de los procesos. Promela ofrece para ello una serie de sentencias que se asemejan a las de los lenguajes de programación, particularmente el lenguaje C. Para que el modelo se comporte de una manera u otra dependiendo de una condición, se utilizan las instrucciones de selección. Por otra

parte, para simular un comportamiento iterativo se utilizan las instrucciones de repetición.

Por último, se necesita definir qué propiedades debe cumplir el sistema. Para ello se utilizan los *never claims* y las fórmulas LTL, entre otras. Estas propiedades, que también denominan llamar invariantes del sistema, serán procesadas y verificadas por Spin.

En el Listado 2.2 se muestra cómo se inician las constantes, los canales de mensajes y un proceso dado. En la línea 15 y 25, se puede observar la instrucción `do` y `od`, que se utiliza para modelar un comportamiento iterativo. El código completo, que se muestra en el Apéndice A (train.pml), es el que se utiliza como ejemplo conductor en el resto del documento.

```
1  #define N 4
2
3  mtype = { appr, leave, go, stop, Empty, Notempty, add, rem, hd };
4
5
6  chan g     = [N] of { mtype, pid };
7  chan qg    = [0] of { mtype, pid };
8  chan q     = [0] of { mtype, pid };
9  chan t[N] = [0] of { mtype };
10
11 active [N] proctype train()
12 {
13     assert(_pid >= 0 && _pid < N);
14
15 Safe:         do
16              :: g!appr(_pid);
17 Approaching:     if
18                 :: t[_pid]?go ->
19                     goto Start
20                 :: t[_pid]?stop
21                 fi;
22 Stopped:        t[_pid]?go;
23 Start:         skip; /* crossing */
24 Crossed:       g!leave(_pid)
25             od
26 }
```

Listado 2.2: Modelo del comportamiento de un tren en Promela

2.2 Spin

Spin es una herramienta que se utiliza para la verificación de modelos. Aparte de verificar propiedades, también ofrece varias opciones para el análisis del sistema modelado, incluyendo simulaciones interactivas, donde el usuario elige el camino que debe tomar el sistema, o guiadas, generadas a partir de un contraejemplo. Estos contraejemplos se generan automáticamente en la fase de verificación cuando una propiedad no se satisface. Esto significa que, por lo menos, existe un camino que puede conducir al sistema a un estado indeseado. Esta situación puede ocurrir ya sea por una mala abstracción del sistema o porque el sistema está modelado correctamente y sí puede alcanzar ese estado.

Otra característica destacable de Spin es que puede generar autómatas a partir de los procesos analizados. Informalmente los autómatas están compuestos por estados y transiciones de estados. Las transiciones hacen referencia a las sentencias ejecutables del proceso y los estados representan al proceso después de ejecutar la sentencia. Para facilitar el análisis, Spin oculta algunos estados que solo se usan en el algoritmo de verificación

y muestra una versión simplificada del autómata. La Figura 2.1 se comparan la versión simplificada y la completa de la traza de Spin.

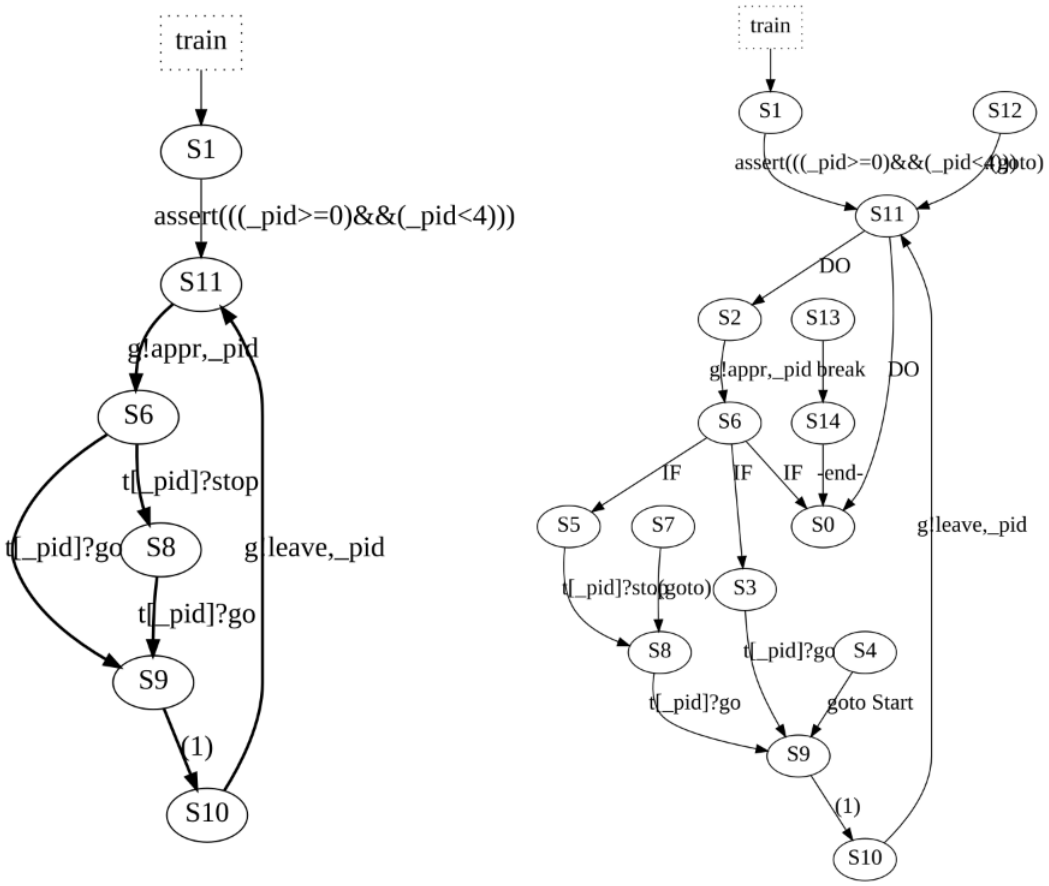

Figura 2.1: Versión simplificada (izquierda) y completa (derecha) de un autómata del Apéndice A

2.3 Alternativas anteriores a este trabajo

Para la realización de este trabajo se han probado diferentes herramientas que tienen un objetivo en común con nuestro trabajo: ofrecer una interfaz que reúna la mayoría de características que ofrece Spin. Las interfaces que se han considerado son: xSpin [33], iSpin [21], jSpin [4] y SpinRCP [6], siendo iSpin una versión actualizada de xSpin y ambos programados en Tcl/Tk [14], mientras que las otras lo están con el lenguaje Java [12].

En la Tabla 2.1 se confrontan las principales características que poseen las interfaces estudiadas y se ha añadido una columna para comparar con el trabajo actual. El nombre del sistema desarrollado en este trabajo es SME, del acrónimo *Spin Modeling Environment*. Las celdas marcadas con un sí o no un reflejan si el correspondiente sistema posee o no esa característica.

Características	xSpin	iSpin	SpinRCP	jSpin	SME
Editar código	Sí	Sí	Sí	Sí	Sí
Comprobar sintaxis	Sí	Sí	Sí	Sí	Sí
Marcar en línea error de sintaxis	No	No	Sí	No	No
Verificar propiedades	Sí	Sí	Sí	Sí	Sí
Verificar grupos de propiedades	No	No	No	No	Sí
Simulación aleatoria	Sí	Sí	Sí	Sí	No
Navegación interactiva entre estados del contraejemplo	Sí	Sí	Sí	Sí	Sí
Navegación automática	No	Sí	No	No	Sí
Mostrar transiciones de los procesos del contraejemplo	No	No	No	No	Sí
Aplicación web	No	No	No	No	Sí
Mostrar output por consola	Sí	Sí	Sí	Sí	Sí
Mostrar canales de mensaje	Sí	Sí	Sí	No	Sí
Mostrar variables locales y globales	No	Sí	Sí	No	Sí
Generar autómatas de los procesos	No	Sí	Sí	Sí	Sí
Conmutar entre contraejemplos	No	No	No	No	Sí

Tabla 2.1: Comparación de las características que ofrece cada herramienta

Las herramientas descritas tienen una opción propia para comprobar errores de sintaxis, mientras que el sistema SME comprueba la sintaxis cuando se realiza la verificación de propiedades. Otra diferencia que existe es que, en el momento de mostrar la comunicación entre los procesos, las interfaces tradicionales tienen una opción de mostrar un diagrama de secuencia [8], donde cada proceso es un actor del diagrama que envía y recibe mensajes. En cambio, SME se encarga de mostrar el contenido de cada canal en una sección de la interfaz a medida que se actualizan actualiza los estados del contraejemplo.

CAPÍTULO 3
Análisis del problema

El objetivo de este capítulo es identificar, definir y especificar los requisitos de los servicios que se implementan en el sistema SME. Se analizan los procesos que tienen que realizar los usuarios para verificar y analizar el sistema modelado de la manera tradicional, es decir, usando Spin sin y con las interfaces de usuario ya mencionadas en el anterior capítulo. Adicionalmente, se documenta los problemas que tienen las interfaces gráficas actuales de Spin lo cual conduce a las ideas inspiradoras para el desarrollo del proyecto.

3.1 Problema actual

Para empezar, las principales tareas a las que da soporte Spin son: verificar propiedades de un sistema modelado en el lenguaje Promela, mostrar trazas de contraejemplo, visualizar la comunicación entre procesos y mostrar la tabla de símbolos del código Promela, es decir, variables, canales de mensajes y procesos.

Para obtener los resultados de estas tareas se requiere conocer los diferentes comandos que ofrece Spin. El problema de esto es la complejidad de las combinaciones de parámetros necesarias, que sin cuya comprensión el análisis de los resultados resulta incompleto o difícil de entender. A esto se añade los problemas que conlleva utilizar una aplicación de consola y el hecho de que los resultados se muestran en texto plano sin ningún formato, complicando aún más el análisis.

3.1.1. Verificación de invariantes

(introducción a la verificación de invariantes). En la Figura 3.1 se muestra el *workflow* de los pasos que tiene que seguir el usuario para verificar las invariantes de un sistema modelado, una vez instalado el sistema.

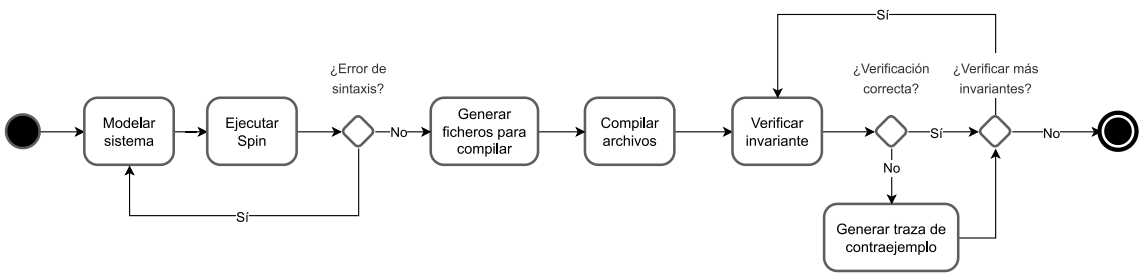

Figura 3.1: *Workflow* del proceso manual de verificación de propiedades

Primero, es necesario modelar el sistema con el lenguaje de Promela, creando los procesos y variables del sistema, además de añadir las propiedades que debe cumplir el sistema y que serán verificadas por Spin. Estas invariantes deben ser etiquetadas con un nombre único.

Antes de ejecutar Spin, es importante fijar correctamente los parámetros que se pueden utilizar para generar el proceso de *model checking* del sistema modelado. Después de ejecutarlo, si no hay ninguna excepción, se generarán ficheros en el lenguaje C, con sus respectivas cabeceras.

El sistema Spin se invoca con el siguiente comando:

```
$ spin -a nombre-fichero
```

Se utiliza un compilador para el lenguaje C y este genera un ejecutable que corresponde al *model checker* para el sistema modelado. Con el ejecutable se puede obtener diferentes resultados dependiendo, nuevamente, de los parámetros que se puedan utilizar. Para verificar una invariante del código escrito se debe utilizar el ejecutable junto a la etiqueta que identifica la propiedad que se quiere verificar:

```
$ ./pan -N etiqueta-invariante
```

Si se ha definido más invariantes, se debe realizar este proceso de forma iterativa con el resto. En el caso del código del Apéndice A, en las últimas líneas están escritas varias fórmulas LTL. Para verificar cada una de ellas habría que ejecutar los siguientes comandos:

```
$ ./pan -N c1
$ ./pan -N c2
$ ./pan -N c3
...
$ ./pan -N c8
```

También hay que tener en cuenta que si una invariante resulta falsificada durante la verificación, el programa genera la traza de contraejemplo. Sin embargo, si ocurre que otra invariante tampoco se cumple, el programa sobrescribe la traza de contraejemplo anterior.

Este proceso puede llegar a ser tedioso, porque cuando se quiere modificar algún detalle del programa se debe realizar nuevamente desde cero este proceso. Para solventar este problema se podría crear un *script* que organizase todos los comandos y resultados, pero este proceso es un inconveniente del que se debe encargar el propio usuario y no la herramienta, a pesar de tratarse de un problema de la propia herramienta.

3.1.2. Interpretación y visualización de trazas de contraejemplo

Después de verificar las invariantes se generan las trazas de contraejemplo de las que no han pasado el proceso de verificación. Para mostrar la traza se debe utilizar el comando `spin` junto a un parámetro que se utiliza para dicha función. Al ejecutar el comando se muestra el camino que puede recorrer el sistema hasta llegar al estado indeseado.

```
$ spin -t nombre-fichero
```

Al mostrar esta información, Spin no muestra la traza completa en un intento de focalizar la atención al usuario en el punto donde se genera el fallo del sistema. Pero si el usuario quisiera ver la traza completa debe utilizar los parámetros específicos para activarlo. Además, existen otros parámetros para mostrar las variables globales y locales de

cada proceso. En consecuencia, la traza que se ve por consola se vuelve un objeto muy difícil de analizar.

```
$ spin -t -v -r -s -l -g -w nombre-fichero
```

Si se ejecuta este comando con el código del modelo ejemplo, `train.pml`, el proceso de verificación de una de sus invariantes falla generando un contraejemplo. El resultado se muestra en el Listado 3.1. Se puede observar que algunas líneas contienen al principio un número. Este número hace referencia al índice del estado del contraejemplo en el que se encuentra el sistema en dicho momento. Por lo tanto, como el último número que aparece es el 502, quiere decir que existen 501 estados que se deben analizar para comprender totalmente el contraejemplo.

```
501:  proc  5 (queue:1) train.pml:70 Send 0 -> queue 1 (list)
501:  proc  5 (queue:1) train.pml:70 (state 9)  [list!who]
    queue 3 (g):
    queue 1 (list): [2][1][3][0]
    Occupied = 0
    Crossed = 0
    Stopped = 0
    Add1 = 0
    Add2 = 0
    Approaching = 0
    queue(5):x = 3
    queue(5):who = 0
502:  proc  - (c6:1) _spin_nvr.tmp:56 (state 1) [(!((len(list)<4)))]  <merge 0
    now @2>
    queue 3 (g):
    queue 1 (list): [2][1][3][0]
    Occupied = 0
    Crossed = 0
    Stopped = 0
    Add1 = 0
    Add2 = 0
    Approaching = 0
spin: _spin_nvr.tmp:56, Error: assertion violated
spin: text of failed assertion: assert(!(!((len(list)<4))))
502:  proc  - (c6:1) _spin_nvr.tmp:56 (state 2) [assert(!(!((len(list)<4))))]
    queue 3 (g):
    queue 1 (list): [2][1][3][0]
    Occupied = 0
    Crossed = 0
    Stopped = 0
    Add1 = 0
    Add2 = 0
    Approaching = 0
Never claim moves to line 56  [assert(!(!((len(list)<4))))]
spin: trail ends after 502 steps
#processes: 6
    queue 3 (g):
    queue 1 (list): [2][1][3][0]
    Occupied = 0
    Crossed = 0
    Stopped = 0
    Add1 = 0
    Add2 = 0
    Approaching = 0
502:  proc  5 (queue:1) train.pml:68 (state 14)
    queue(5):x = 3
    queue(5):who = 0
502:  proc  4 (gate:1) train.pml:46 (state 14)
    gate(4):who = 0
502:  proc  3 (train:1) train.pml:29 (state 8)
502:  proc  2 (train:1) train.pml:30 (state 9)
```

```
502:  proc  1 (train:1) train.pml:29 (state 8)
502:  proc  0 (train:1) train.pml:29 (state 8)
502:  proc  - (c6:1) _spin_nvr.tmp:55 (state 6)
6 processes created
```

Listado 3.1: Contraejemplo resultado de la violación de una invariante en el modelo del Apéndice A

Otro problema de este proceso es que funciona cuando solo hay una traza de contraejemplo. Para el caso de tener más trazas se debe utilizar un nuevo parámetro junto al nombre de la traza y el nombre del fichero que contiene el código. Y además se debe realizar estos pasos iterativamente hasta analizar todos los contraejemplos.

```
$ spin -k nombre-contraejemplo nombre-fichero
```

3.2 Identificación y definición de requisitos

A continuación, se documenta que métodos se han usado para la obtención de requisitos de este trabajo. Las técnicas utilizadas son una serie de procesos que se utilizan habitualmente en el ámbito de ingeniería de requisitos y que han sido adaptadas a las necesidades del proyecto.

3.2.1. Técnicas de elicitación

Las técnicas de elicitación son un recurso para poder identificar las necesidades y las características que debería tener un futuro sistema. Dependiendo de la técnica que se use se puede obtener diferente tipo de información. En el caso de nuestro trabajo, se han utilizado tres tipos de técnicas distintas: encuestas, escenarios y casos de uso [26].

La técnica de encuestas se utiliza para conocer diferentes puntos de vista, opiniones, problemas con el sistema actual y posibles ideas para un nuevo sistema. Los escenarios muestran cómo se debería comportar el sistema. Finalmente, los casos de uso descubren cómo se relacionan las funcionalidades.

3.2.1.1 Encuestas

Se ha creado una encuesta con una serie de preguntas relacionadas con la usabilidad del *model checker* Spin, una de las herramientas utilizadas en las prácticas de laboratorio de la asignatura de Métodos Formales Industriales (MFI) del 3º curso del grado en ingeniería informática de la Escuela Técnica Superior de Ingeniería Informática (ETSINF). Las encuestas han sido cumplimentadas por los alumnos de este curso de MFI.

La encuesta consta de cuatro frases en las que el entrevistado elige una de las cinco posibles respuestas, haciendo referencia a cuánto de acuerdo está con la afirmación escrita. Por último, se ha incluido una pregunta de respuesta abierta donde el entrevistado puede proponer ideas para la nueva aplicación.

Los enunciados de la encuesta son las siguientes:

1. En general, las aplicaciones de líneas de comandos son más difíciles de usar que las aplicaciones con interfaz gráfica.
2. Considero que los contraejemplos de Spin son difíciles de entender.
3. Representar gráficamente los contraejemplos de Spin ayudaría a entenderlos mejor.

4. Instalar una aplicación de escritorio es mejor que utilizar una aplicación web.

5. ¿Qué funcionalidades te gustaría que tuviera una interfaz gráfica para las prácticas de MFI con Spin?

El objetivo de la encuesta es encontrar el grado de satisfacción de utilizar la interfaz de consola de Spin y obtener nuevos requisitos para el nuevo sistema.

Como se ha comentado anteriormente, la encuesta fue entregada a los alumnos de MFI del curso 2021/2022 y hubo 73 participantes. A continuación, se presenta una breve explicación del significado de la pregunta, los resultados de la encuesta y la conclusión que hemos extraído:

Pregunta 1: En general, las aplicaciones de líneas de comandos son más difíciles de usar que las aplicaciones con interfaz gráfica.

La pregunta consiste en comparar entre las aplicaciones con una interfaz de línea de comandos y con una interfaz gráfica. Como se muestra en la Figura 3.2, se puede observar que la mayoría de los usuarios se sienten más cómodos utilizando una interfaz gráfica que una aplicación con interfaz de línea de comandos.

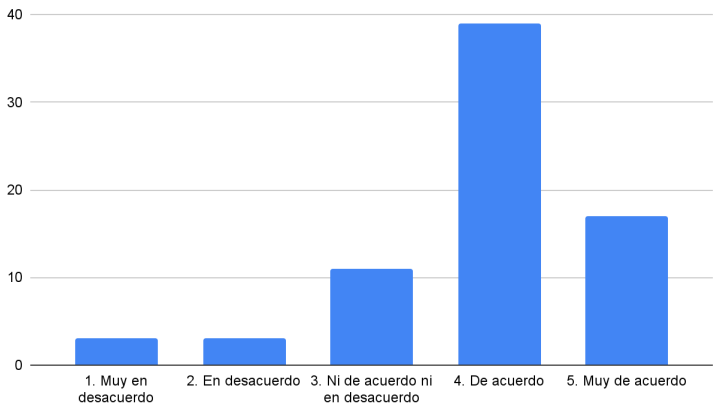

Figura 3.2: Gráfico de los resultados de la pregunta 1 de la encuesta

Pregunta 2: Considero que los contraejemplos de SPIN son difíciles de entender.

El resultado de esta pregunta de la encuesta es el que se muestra en la Figura 3.3. A partir de los resultados se puede concluir que para aproximadamente la mitad de los alumnos no es tan difícil entender el contraejemplo, pero sí que existe una gran cantidad de alumnos que sí que la encuentran difícil.

Pregunta 3: Representar gráficamente los contraejemplos de SPIN ayudaría a entenderlos mejor.

La tercera pregunta está relacionada con una de las características que se han incluido en el nuevo sistema. De este modo, se puede confirmar que el nuevo producto se adapta a las necesidades de los usuarios de Spin.

A partir de la Figura 3.4 se deduce que la mayoría de los entrevistados se sentirían más cómodos pudiendo visualizar el contraejemplo de manera gráfica y no sólo de la manera tradicional por la línea de comandos.

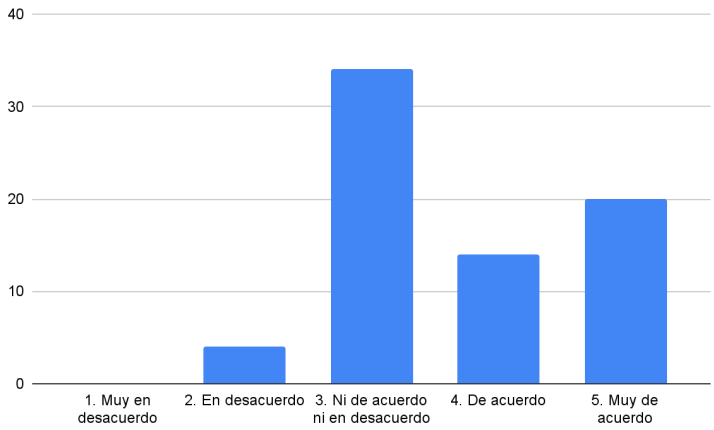

Figura 3.3: Gráfico de los resultados de la pregunta 2 de la encuesta

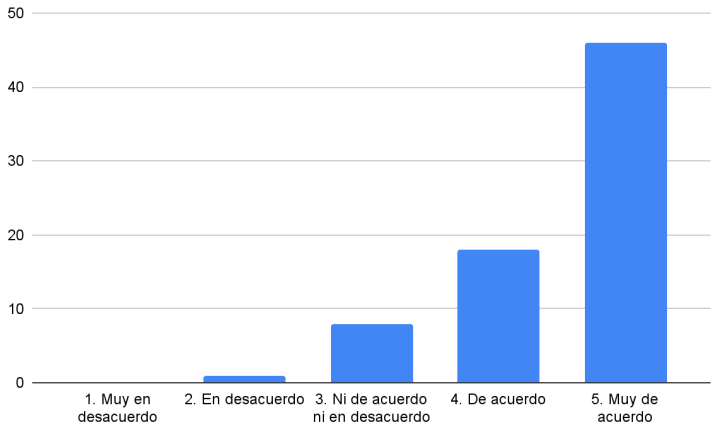

Figura 3.4: Gráfico de los resultados de la pregunta 3 de la encuesta

Pregunta 4: Instalar una aplicación de escritorio es mejor que utilizar una aplicación web.

En este caso, las opciones 1 y 2 hacen referencia a que es mejor utilizar una aplicación web a instalar una aplicación en el ordenador. Las opciones 4 y 5 significan que se está a favor a la pregunta, mientras que la opción 3 indica que no es mejor ni peor un tipo de aplicación u otro.

La Figura 3.5 indica que la mayoría de los alumnos considera que sí es mejor utilizar una aplicación desde el navegador a instalar una aplicación en el propio ordenador, si bien existe un gran porcentaje de neutralidad correspondiente a alumnos que, al contar con equipos potentes propios, mantienen que no es mejor ni peor.

Pregunta 5: ¿Qué funcionalidades te gustaría que tuviera una interfaz gráfica para las prácticas de MFI con SPIN?

El objetivo de esta pregunta es conseguir nuevos requisitos, tanto funcionales como no funcionales, de los alumnos entrevistados. El proceso que se ha seguido para obtener los requisitos es el siguiente:

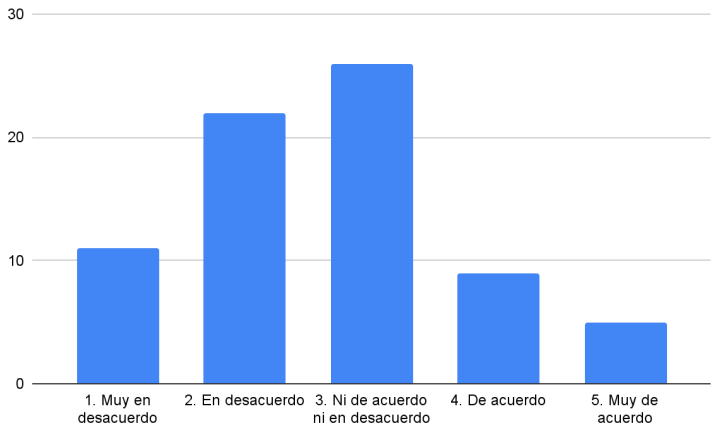

Figura 3.5: Gráfico de los resultados de la pregunta 4 de la encuesta

- Muchas de las respuestas contenían ideas parecidas; por lo tanto, se necesitó agruparlas para conocer qué requisitos y características solicitaban más los alumnos.

- Los resultados que se muestran en la Tabla 3.1 dan a entender que los entrevistados reclaman poder ver el contraejemplo de manera visual y no en forma de lista textual, que es como se muestra de la manera tradicional.

- También solicitan un editor de código en el que se pueda ver posibles errores de sintaxis y autocompletado de variables, funciones o palabras clave del lenguaje Promela.

Requisito	N.º de personas	Tipo de requisito
Mostrar visualmente la traza del contraejemplo	20	Funcional
Mostrar errores de sintaxis del código de Promela	9	Funcional
Mostrar posibles comandos de Spin	5	Funcional
Autocompletado Promela	4	Funcional
Ver variables paso a paso	4	Funcional
Código de ejemplo y tutorial	3	Funcional
Componentes manipulables	3	Funcional
Mostrar evolución del sistema por la traza de contraejemplo	3	Funcional
Interfaz intuitiva	3	No funcional
Errores explicativos	1	No funcional
Navegabilidad por los estados del contraejemplo botones	1	Funcional
Opción de añadir fórmulas LTL	1	Funcional
Editar código	1	Funcional
Comprobar conjuntos de fórmulas LTL	1	Funcional
Árboles de decisión para ver el camino	1	Funcional

Tabla 3.1: Requisitos obtenidos a partir de la pregunta 5

3.2.1.2 Escenarios

Con la técnica de escenarios se describe el comportamiento tanto de los actores que interactúan con el sistema como del propio sistema. Hay diferentes formas de documentar esta técnica y en nuestro caso, se utiliza la versión estructurada. Esta versión describe de manera ordenada los sucesos. Además, incluye un apartado para explicar si hay algún camino alternativo o cómo se debería tratar los errores.

Tras analizar el resultado de las encuestas, se proponen varios escenarios para comprender los posibles usos del nuevo sistema:

Verificar invariantes

La Tabla 3.2 indica la secuencia de tareas que realizan los diferentes actores del sistema, que son: el usuario, el sistema *frontend* y *backend*. Se puede observar que el usuario ejecuta una acción y el resto las deben ejecutar los sistemas. Además, en la tabla se indican rutas alternativas: el usuario puede importar un fichero externo y si sí el *frontend* no encuentra invariantes, este mostrará un mensaje; por el contrario, si el *backend* encuentra un error en el código, se mostrará un mensaje en una ventana emergente.

	Secuencia principal
1.	El usuario escribe el código en el editor de la página.
2.	El *frontend* busca en el código las invariantes escritas por el usuario.
3.	El *frontend* genera una lista de invariantes del código.
4.	El *frontend* muestra la lista en la página web.
5.	El usuario selecciona las invariantes que quiere verificar.
6.	El usuario pulsa el botón para verificar las invariantes seleccionadas.
7.	El *frontend* recoge el código y las invariantes.
8.	El *frontend* genera una solicitud para enviar al *backend*.
9.	El *backend* procesa la solicitud.
10.	El *backend* genera un fichero con el código.
11.	El *backend* ejecuta Spin con el fichero creado y genera ficheros para compilar.
12.	El *backend* compila los ficheros y genera un ejecutable.
13.	El *backend* elimina los ficheros, excepto el ejecutable y el código.
14.	El *backend* verifica los invariantes con el ejecutable.
15.	Si al verificar un invariante, éste no pasa la verificación, se generará un archivo con la traza de contraejemplo.
16.	El *backend* genera una lista con los invariantes que no han pasado la verificación.
17.	El *backend* genera una respuesta para enviar al *frontend*.
18.	El *frontend* procesa la respuesta
19.	El *frontend* muestra la lista de invariantes que no han pasado la verificación.
	Alternativas / Errores
1.1.	El usuario importa un fichero con código.
2.1.	Si el *frontend* no encuentra ninguna invariante, muestra un mensaje de error.
11.1.	Si al ejecutar Spin se encuentra un error en el código el sistema devuelve un error al *frontend* y este lo muestra al usuario.

Tabla 3.2: Escenario estructurado de verificar invariantes

Ver contraejemplo

La secuencia de la Tabla 3.3 representa las acciones que se llevan a cabo para mostrar un contraejemplo. En este caso no hay caminos alternativos.

Secuencia principal	
1.	El *frontend* muestra la lista de invariantes que no han pasado la verificación.
2.	El usuario solicita ver los contraejemplos
3.	El *frontend* recupera la respuesta del contraejemplo asociado.
4.	El *frontend* actualiza la lista de variables.
5.	El *frontend* muestra un grafo dirigido con los estados de la traza de contraejemplo.

Tabla 3.3: Escenario estructurado de ver contraejemplo

Analizar contraejemplo

Analizar un contraejemplo hace referencia a navegar entre los estados del contraejemplo para observar su comportamiento. Las acciones son las que se muestran en la Tabla 3.4

Secuencia principal	
1.	El usuario pulsa uno de los botones de control para navegar entre los estados del grafo de contraejemplo.
2.	El *frontend* actualiza las variables globales y locales.
Alternativas / Errores	
1.1.	El usuario selecciona uno de los nodos del grafo para cambiar de estado del grafo.

Tabla 3.4: Escenario estructurado de analizar contraejemplo

Cambiar de contraejemplo

La diferencia entre este escenario y el de "Ver contraejemplo" es que en este el usuario puede decidir cuál de los contraejemplos quiere ver. En la Tabla 3.5 indica cómo se debe hacer.

Secuencia principal	
1.	El usuario selecciona una traza de contraejemplo.
2.	El *frontend* recupera la respuesta del contraejemplo asociado.
3.	El *frontend* abre una nueva ventana.
4.	El *frontend* actualiza la lista de variables.

Tabla 3.5: Escenario estructurado de cambiar contraejemplo

3.2.1.3 Casos de uso

Los casos de uso representan qué tareas pueden realizar los usuarios del sistema. Estas tareas pueden estar relacionadas con otras subtareas que realiza el propio sistema

o subsistemas. Estas tareas reciben el nombre de la propia técnica: casos de uso. Para este trabajo se distinguen los dos sistemas: el *frontend* y *backend*. Cada sistema tiene sus propios casos de uso que se deben definir y especificar en las siguientes fases.

Verificar invariantes

El caso de uso de "Verificar invariantes", que se muestra en la Figura 3.6, complementa al escenario con el mismo título. En este apartado se muestra la relación de las tareas que puede realizar el usuario con las que realiza el servidor y qué pasos debe tomar el usuario para poder alcanzar la tarea de verificación con éxito.

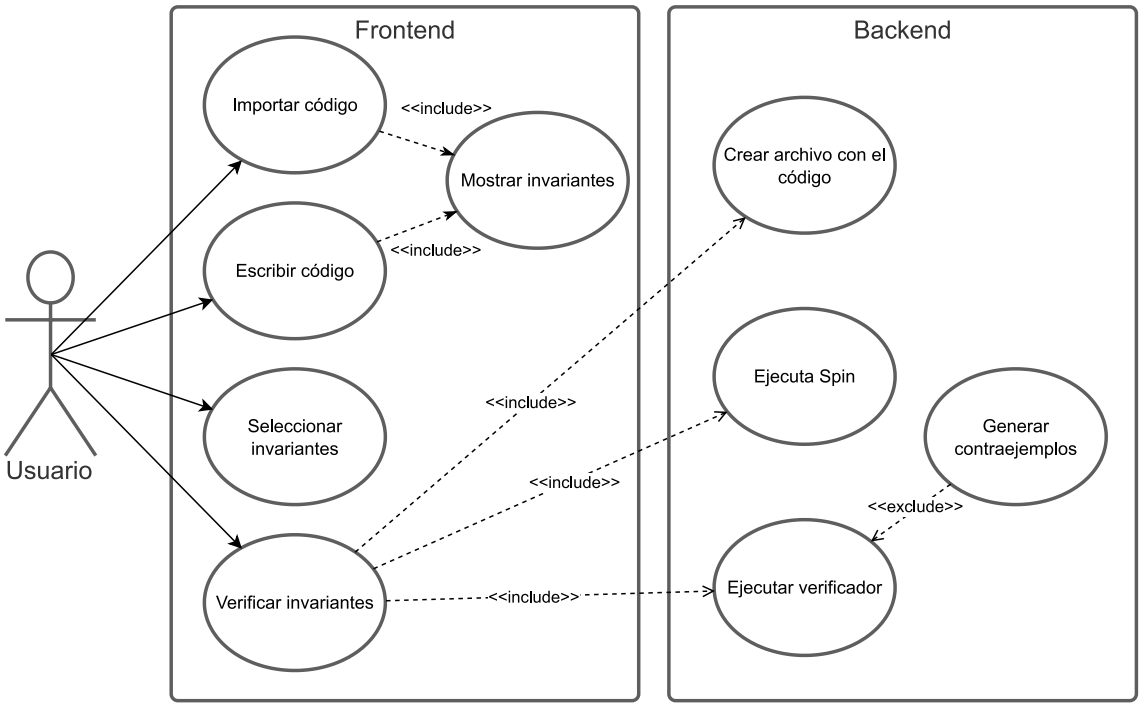

Figura 3.6: Caso de uso de verificar invariantes

Ver contraejemplo

La tarea de ver un contraejemplo se refiere a poder visualizar la traza, junto a las variables locales de los procesos, las variables globales del sistema y los mensajes que realizan los procesos. Los pasos que se siguen se describen en el apartado de escenarios con el mismo nombre y las relaciones entre las tareas se muestra en la Figura 3.7.

Analizar contraejemplo

Para complementar la Figura 3.8, hay que tener en cuenta que el usuario se encuentra en la página que se muestra el contraejemplo. La finalidad de este caso de uso es actualizar la lista de variables y procesos al estado que el usuario quiere visualizar.

Cambiar de contraejemplo

Del mismo modo que en la sección anterior, este caso de uso sucede en la página de mostrar el contraejemplo y se encarga de recuperar un contraejemplo distinto y actualizar la configuración. Del mismo modo que el caso anterior, son tareas que realiza la parte del *frontend* como se visualiza en la Figura 3.9

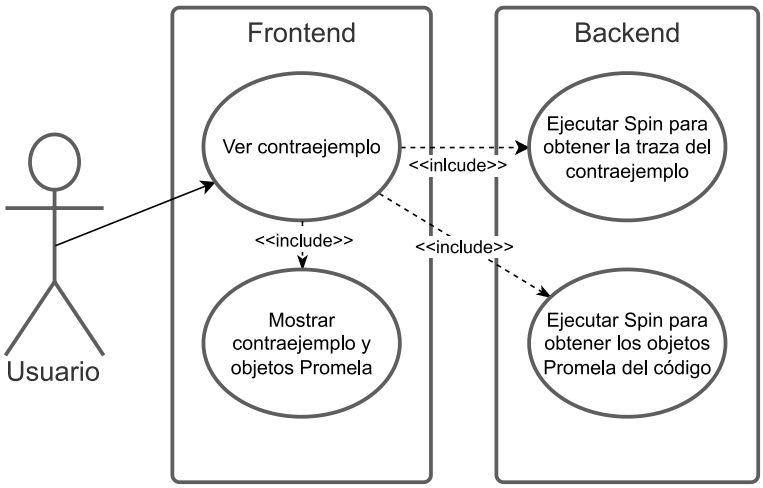

Figura 3.7: Caso de uso de ver contraejemplo

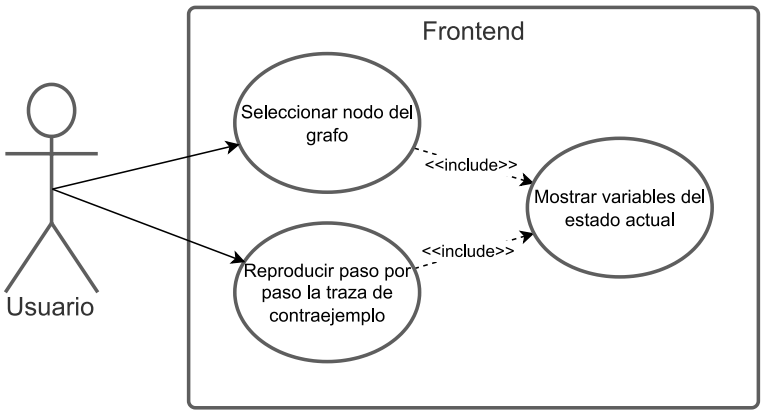

Figura 3.8: Caso de uso de analizar contraejemplo

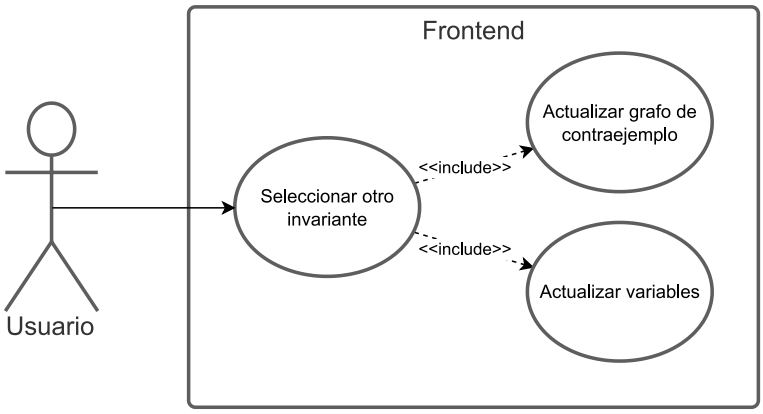

Figura 3.9: Caso de uso de cambiar de contraejemplo

3.2.2. Definición y negociación de requisitos

El proceso de definición abarca la tarea de recoger todos los requisitos de la fase de identificación. Posteriormente, la fase de negociación se ocupa de agrupar y priorizar los requisitos que pasarán a la siguiente fase. Los requisitos funcionales y no funcionales se han obtenido a partir de las técnicas empleadas en la sección anterior y se muestran en la Tabla 3.6.

Requisitos funcionales	Requisitos no funcionales
Mostrar visualmente la traza del contraejemplo.	Interfaz intuitiva.
Reproducir paso por paso la traza del contraejemplo.	Errores explicativos.
Mostrar errores de sintaxis del código de Promela.	Aplicación web.
Mostrar posibles comandos de Spin.	Soportar varias solicitudes.
Autocompletado Promela.	
Ver variables del contraejemplo paso a paso.	
Código de ejemplo y tutorial.	
Mostrar objetos Promela manipulables.	
Navegabilidad por los estados del contraejemplo con botones.	
Añadir fórmulas LTL.	
Escribir y editar código.	
Comprobar conjuntos de fórmulas LTL.	
Árboles de decisión para ver el camino.	
Seleccionar otra traza de contraejemplo.	
Actualizar variables.	
Actualizar contraejemplo.	
Seleccionar nodo del grafo.	
Obtener objetos Promela a partir del código.	
Obtener invariantes del código.	
Obtener traza de contraejemplo.	
Importar código.	

Tabla 3.6: Requisitos obtenidos a partir de las técnicas de elicitación

3.2.2.1 Priorizar requisitos

Los requisitos se han agrupado en tres características generales que debería cumplir este proyecto: edición de código Promela, verificar invariantes y analizar trazas de contraejemplo. Estas características son un conjunto de requisitos que el proyecto debe tratar.

- El editor de código, como su propio nombre indica, es un editor en línea para poder visualizar el sistema modelado en Promela. El editor abarca las tareas relacionadas con escribir, editar e importar código. Este editor en línea será accesible a través de la aplicación.
- La verificación de invariantes se encarga de analizar el código para encontrar invariantes en el mismo. Además, se muestran en formato lista las invariantes recogi-

das. En la lista se puede seleccionar qué invariantes se quiere verificar y el *backend* se encarga de ejecutar Spin y verificar las propiedades seleccionadas.

- Analizar un contraejemplo reúne los requisitos de mostrar y actualizar los estados del sistema y componentes Promela del código. Los estados y componentes, es decir, las variables y procesos, son obtenidos a través del proceso de verificación.

3.3 Solución propuesta

La solución que se propone para este trabajo es la creación de una aplicación web donde el usuario podrá editar código, verificar invariantes y analizar contraejemplos. El hecho de que se trata de una aplicación web facilita el acceso a los usuarios que tengan un navegador web y de este modo no requiere instalar una aplicación ocupando espacio en el disco.

Para conseguir estos propósitos, el trabajo se organiza siguiendo la estructura de sus dos partes fundamentales: el *frontend* y el *backend*. Adicionalmente, para la comunicación entre las dos partes se utilizarán los métodos de petición HTTP [29].

3.3.1. Propuesta de *frontend*

El *frontend* implementa la interfaz entre el usuario y el servidor. En la página web se puede escribir, editar e importar archivos Promela. Posteriormente, la aplicación se encarga de buscar las fórmulas LTL descritas en el código para mostrarlas en una lista.

En la lista, el usuario puede seleccionar las invariantes que necesita verificar. Después de la verificación, la página mostrará los mensajes correspondientes acerca de si ha pasado o no la verificación. El servidor mandará un mensaje de error si el código enviado contiene errores de sintaxis y el *frontend* también mostrará un mensaje de error.

Cuando una o varias invariantes no han pasado la verificación, la aplicación muestra una lista que contiene dichas invariantes y la opción de visualizar los contraejemplos generados.

En el apartado de analizar contraejemplos se muestra un grafo dirigido, las variables locales y globales del sistema y diferentes botones para navegar entre los nodos del grafo. El grafo representa el sistema a través del tiempo y los nodos indican el estado en el que se encuentra en el sistema en ese momento. Cuando se navega a través de los estados, la lista de variables se actualizará.

Otra opción para entender mejor el sistema es visualizar el propio funcionamiento de los procesos. Estos se muestran, de igual modo que el contraejemplo, en formato de grafo dirigido, donde cada nodo contiene el contador de programa del proceso y las aristas corresponden a una ejecución de sentencia relevante.

Una sentencia relevante hace referencia a aquellas sentencias que se encuentran dentro de un bucle o condicional, o un envío de mensajes entre procesos. Por el contrario, una serie de sentencias de inicializar variables se agrupan en un solo estado.

3.3.2. Propuesta de *backend*

La parte del *backend* consiste en una aplicación que se instala en el servidor para poder procesar solicitudes HTTP. Estas solicitudes pueden consistir en solicitar la página web y ejecutar Spin. También se encarga de distribuir los recursos del servidor para poder abarcar un mayor número de solicitudes.

Es común que se pueda solicitar la página web de la aplicación desde cualquier navegador, mientras que para ejecutar en el servidor Spin se necesita realizar una solicitud desde el *frontend*.

La aplicación de servidor se encarga de crear una instancia de Spin junto a los parámetros necesarios, ya que realizar la tarea de verificación requiere ciertos parámetros, mientras que para mostrar información del contraejemplo se necesitan otros.

Para todas las tareas que requieran una respuesta por parte del servidor, este se encarga de generar una respuesta adecuada. Tanto las respuestas como las solicitudes se representan en formato JSON, del acrónimo *JavaScript Object Notation* [5].

Cabe destacar que cuando se modela un sistema en Promela, también se puede incluir código C. Esto podría ocasionar una vulnerabilidad en el sistema del servidor, porque si se llega a ejecutar código malicioso, esto podría desencadenar un riesgo importante para la seguridad del sistema. Para impedir este problema, antes de ejecutar Spin, el servidor debe analizar el código y en el caso de encontrarse código C, devolverá un error al cliente indicando que no se puede incluir código C en el modelo.

CAPÍTULO 4
Diseño de SME

El propósito de este capítulo es doble: definir la estructura y comunicación entre los sistemas *frontend* y *backend*, del sistema SME y documentar los componentes que forman el diseño de la interfaz gráfica y su funcionamiento. Por último, se explican brevemente las tecnologías utilizadas para el desarrollo de la aplicación de servidor y de los elementos visuales.

4.1 Arquitectura del sistema

Una arquitectura software hace referencia a la estructura de un sistema software, abstrayendo los elementos que lo componen y la comunicación entre ellos. De hecho, cuando se diseña una nueva aplicación es muy importante observar una arquitectura porque ésta definirá el desarrollo de dicha aplicación.

En la actualidad existe una gran variedad de tipos de modelos que se pueden utilizar, cada uno con sus propias ventajas e inconvenientes. Por lo tanto, a la hora de elegir un tipo de modelo hay que tener en cuenta las características de cada proyecto y analizar la opción que mejor se adapte al mismo.

En el caso de este proyecto, se ha tenido en cuenta la fecha de entrega, el equipo de desarrollo y las funcionalidades a implementar. La opción que más se adapta a estas opciones es la conocida como cliente-servidor [31], o también nombrada modelo de dos capas [16].

Esta estructura, como su nombre indica, está compuesta por dos bloques principales, el cliente y el servidor o, como utilizamos también a lo largo del documento, *frontend* y *backend*. A grandes rasgos, las características principales de este diseño son gracias a la separación en dos capas, es más sencillo definir qué funcionalidades es responsabilidad de cada una y que, al añadir o modificar alguna funcionalidad, simplemente hay que comprobar la comunicación entre las capas y no el diseño por completo, como podría ocurrir en los modelos de una capa o monolíticos.

4.2 Comunicación entre capas

Para iniciar una comunicación entre las capas primero el cliente tiene que solicitar la información al servidor. Entonces, este procesa la información y ejecuta las funciones necesarias para generar una respuesta y, por último, se envía la información solicitada al cliente. Por ejemplo, ver una página web, descargar algún archivo o ver un vídeo, son solicitudes comunes que un usuario puede realizar al navegador. Del mismo modo,

el navegador solicita esta información a los servidores y estos a su vez pueden cursar solicitudes a otros servidores para completar su respuesta al usuario final.

La comunicación descrita anteriormente es una aproximación de cómo funciona la comunicación en internet. Ya que se podría entender que este tipo de estructura corresponde a de una arquitectura multicapa. No obstante, para este proyecto se utiliza una versión más simple porque, al haber dos componentes que se deben comunicar, sólo hay que ocuparse de un canal de mensajes.

Cabe destacar que la notación en que se envía y recibe la información en este proyecto es en formato JSON. Este formato está formado por objetos, que son un tipo de dato que se utiliza para almacenar información estructurada. Dichos objetos se componen de un conjunto de elementos, los cuales contienen un valor y una clave. El valor contiene información relevante: un número, un booleano, una lista o, incluso, otro objeto. Por una parte, la clave es el identificador del valor, que se utiliza para localizar y obtener dicho valor.

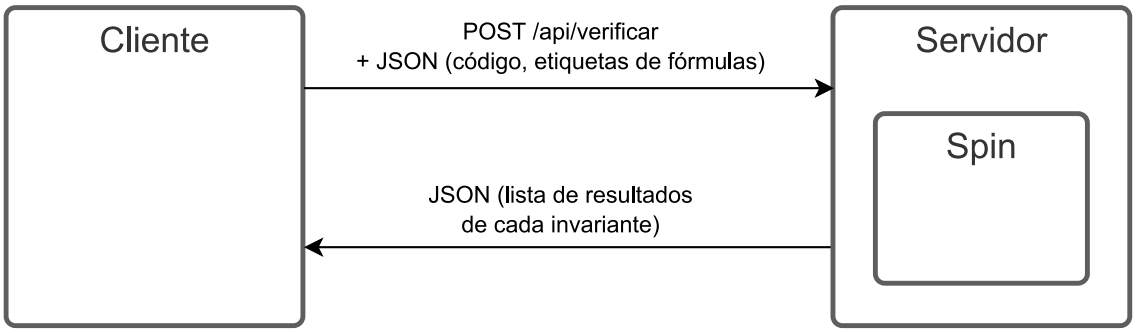

Figura 4.1: Comunicación verificar propiedades

En la Figura 4.1, se describe la comunicación entre el *frontend* y el *backend* para el requisito "Verificar propiedades". El *frontend* recoge en un objeto JSON del código del usuario y las propiedades a verificar, y lo envía al *backend*. Este procesa la información, ejecuta las funciones necesarias con Spin y genera una respuesta. Esta respuesta será procesada por el *frontend* y mostrará un mensaje al usuario.

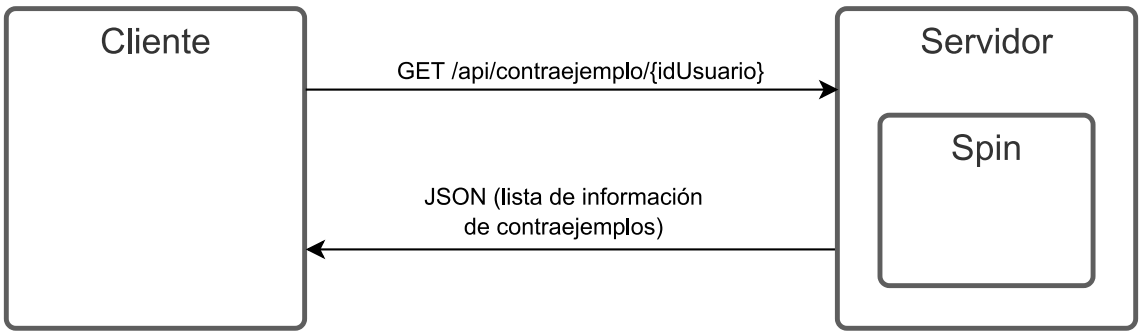

Figura 4.2: Comunicación para visualización de contraejemplos

Para finalizar, en la verificación, el *backend* puede generar una serie de elementos que posteriormente utiliza el *frontend*, como son: el contraejemplo, los procesos, las variables, tanto globales como locales, y los canales de mensajes del sistema modelado. En la Figura 4.2 se muestra cómo el *frontend* solicita esta información para procesarla y generar grafos y tablas.

4.3 Diseño de las páginas de la aplicación

Después de documentar la estructura y cómo se va a comunicar nuestro sistema, es turno del diseño de las interfaces que muestra la aplicación. En esta sección se explicarán las decisiones de diseño para la aplicación, cuál será su funcionamiento de cada componente y cómo el usuario puede utilizarlas.

La aplicación consta de dos páginas principales: un editor, donde el usuario escribe o importa código Promela y selecciona las invariantes, y una página donde se muestra el contraejemplo y los procesos en formato de grafo, y una lista de variables y canales de mensaje que utiliza el código. A esta página en adelante la llamaremos visualizador.

4.3.1. Editor

El editor es la página principal donde el usuario podrá definir un sistema modelado y verificar las propiedades. En la Figura 4.3, se puede distinguir tres partes: un encabezado, el propio editor y una barra lateral. El encabezado muestra el título del código, que si es importado, será el nombre del fichero utilizado. Además de la opción de importar código está la de descargar el código que se encuentra el editor, mientras que la barra lateral es donde se muestran las fórmulas LTL encontradas en el código.

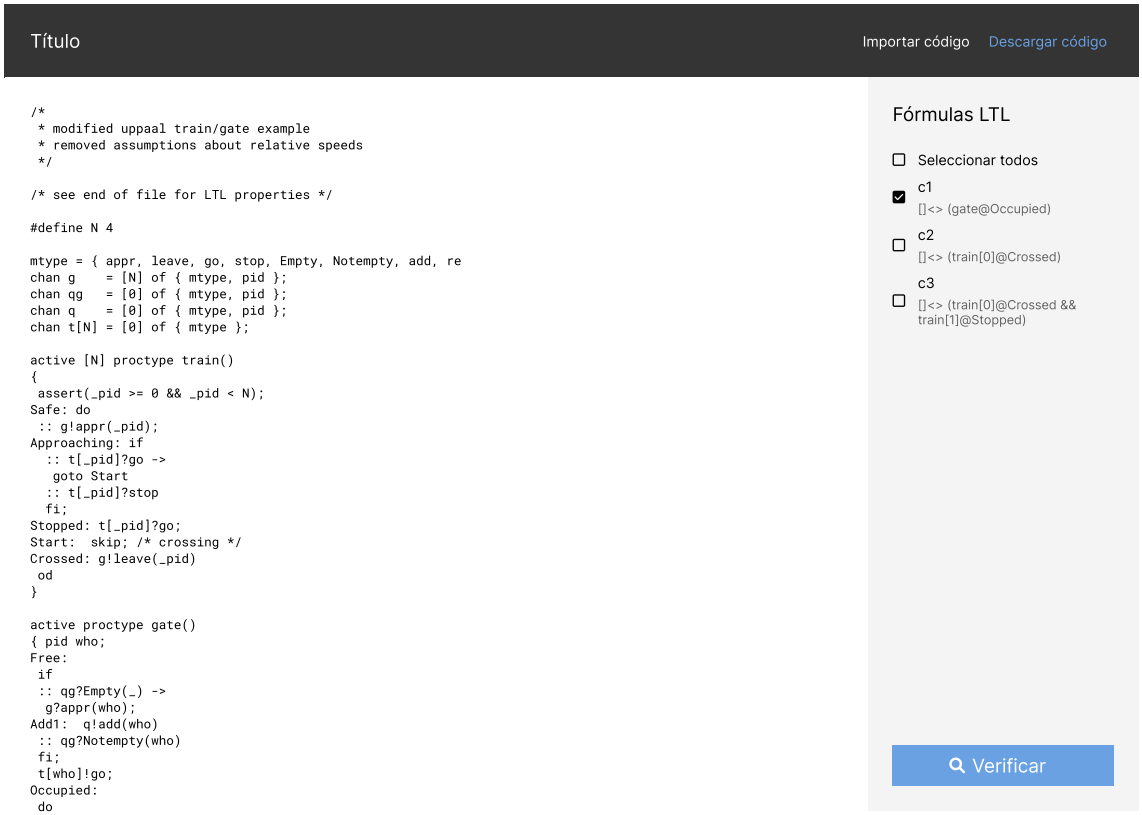

Figura 4.3: Interfaz del editor

Fórmulas LTL

La barra lateral consta de una lista de fórmulas LTL seleccionables y el botón para iniciar la verificación de propiedades LTL. La lista se muestra como en la Figura 4.4. Si el *frontend* no encuentra ninguna invariante mostrará un mensaje en la barra indicándolo,

mientras que, si existen fórmulas en el código, el *frontend* las mostrará para que el usuario pueda seleccionar las que desee verificar.

Por otro lado, si el usuario no ha seleccionado ninguna fórmula, el botón de verificar no estará disponible hasta que se haya seleccionado alguna. El panel también contiene una opción de seleccionar todas las invariantes, si bien solo aparece si en el código hay más de una fórmula descrita.

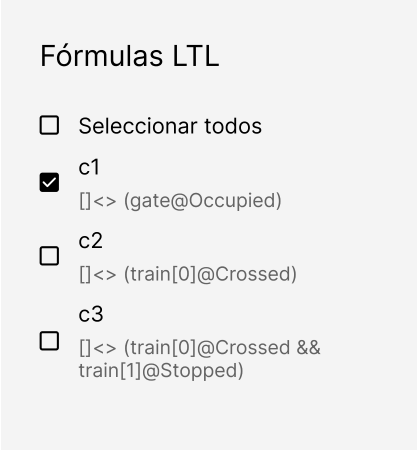

Figura 4.4: Columna donde se muestran las fórmulas LTL

Verificación del modelo

Tras pulsar el botón de verificar, se abrirá una ventana emergente similar a la que se muestra en la Figura 4.5, que informa al usuario que está en curso la verificación de las propiedades seleccionadas. El mensaje no cambiará hasta que el *backend* termine la verificación y envíe los resultados. En el caso de que exceda un tiempo determinado se mostrará un error. Si el resultado indica que todas las fórmulas LTL han pasado satisfactoriamente la verificación, el mensaje mostrará dicha información. Mientras que si existe alguna que no ha pasado se mostrará un mensaje indicando las que no han pasado, junto al resultado de la consola generado por Spin.

Tras el mensaje de que alguna propiedad no ha pasado la verificación, la ventana ofrece dos opciones: analizar los contraejemplos o cerrar la ventana. Si el usuario decide elegir la primera opción, pulsando en la ventana de la aplicación, pasará a la siguiente página, activando así el visualizador. Puede ocurrir que la ventana muestra una alerta si el código contiene errores de sintaxis o fragmentos de código escrito en C. En caso de un error de sintaxis, Spin devuelve un mensaje de error que indica en qué línea se puede encontrar el error.

4.3.2. Visualizador

Esta componente es la que se encarga de mostrar la información de los contraejemplos generados. En ella se tomó una de las decisiones más importantes del proyecto: cómo mostrar esa información. Para empezar, se tuvo en cuenta la manera en que la presentan las diferentes interfaces existentes y el propio Spin vía la consola. Estas, en general, muestran una lista de estados con texto plano, donde cada elemento de la lista contiene información del propio estado, las variables de cada proceso y los mensajes que se van generando.

Los estados del contraejemplo representan una transición de un estado a otro de un proceso interno. Spin tiene incorporado una función para generar un autómata de cada

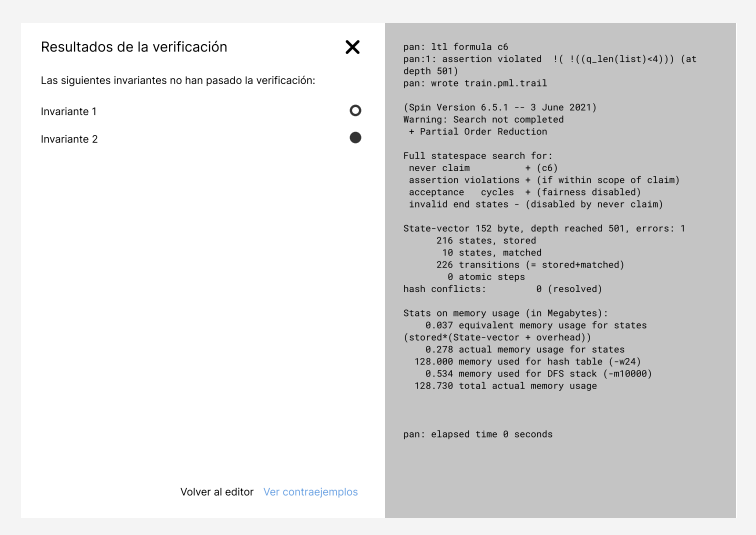

Figura 4.5: Ventana que muestra una o varias propiedades que no han pasado la verificación

proceso del código, donde cada transición corresponde generalmente a la ejecución de una instrucción de dicho proceso. A partir de esta información, la manera en la que se muestra en este proyecto toda la información relevante es en forma de grafos dirigidos.

Como se muestra en la Figura 4.6, la ventana se divide en varias secciones. La cabecera es similar a la del editor, pero con diferentes opciones. En la columna de la izquierda se visualiza el grafo del contraejemplo, mientras que la columna de la derecha está dividida en dos partes para separar las variables y los mensajes. Por último, hay una sección central para mostrar los autómatas de los procesos.

Cabecera

El encabezado que muestra la Figura 4.7 está formado por la etiqueta de la fórmula LTL que ha generado un contraejemplo junto a una serie de opciones para que el usuario pueda descargar los archivos generados por el *backend*, volver al editor o cambiar de contraejemplo. Esta última opción solamente aparece si en la anterior verificación ha aparecido más de una propiedad que ha fallado.

Al pulsar el botón de "Descargar ficheros" se abre una ventana con una lista de archivos que el usuario puede seleccionar para descargarlos, además de una opción de seleccionarlos todos. Está dividido en tres conjuntos: los archivos con la extensión `trail`, que son los datos que utiliza Spin para reconstruir la traza de contraejemplo, archivos SVG [13] de los autómatas de cada proceso y, por último, archivos en formato DOT [20, 11], que es una notación para definir grafos. Por último, si no se selecciona ningún archivo, el botón de descargar no se puede utilizar, como se muestra en la Figura 4.8, en ambas versiones.

Para terminar, la última opción de "Cambiar de contraejemplo" abre una ventana que contiene la lista de contraejemplos a los que puede acceder el usuario. Por ejemplo, la ventana podría contener algo similar a lo que muestra la Figura 4.9. Al cambiar de contraejemplo, la interfaz se encarga de actualizar el resto de componentes para recuperar la información de la correspondiente traza.

Contraejemplo

La traza de contraejemplo, que se ubica en la columna de la izquierda, muestra un grafo dirigido que lo representa, donde cada nodo representa un estado de un proceso

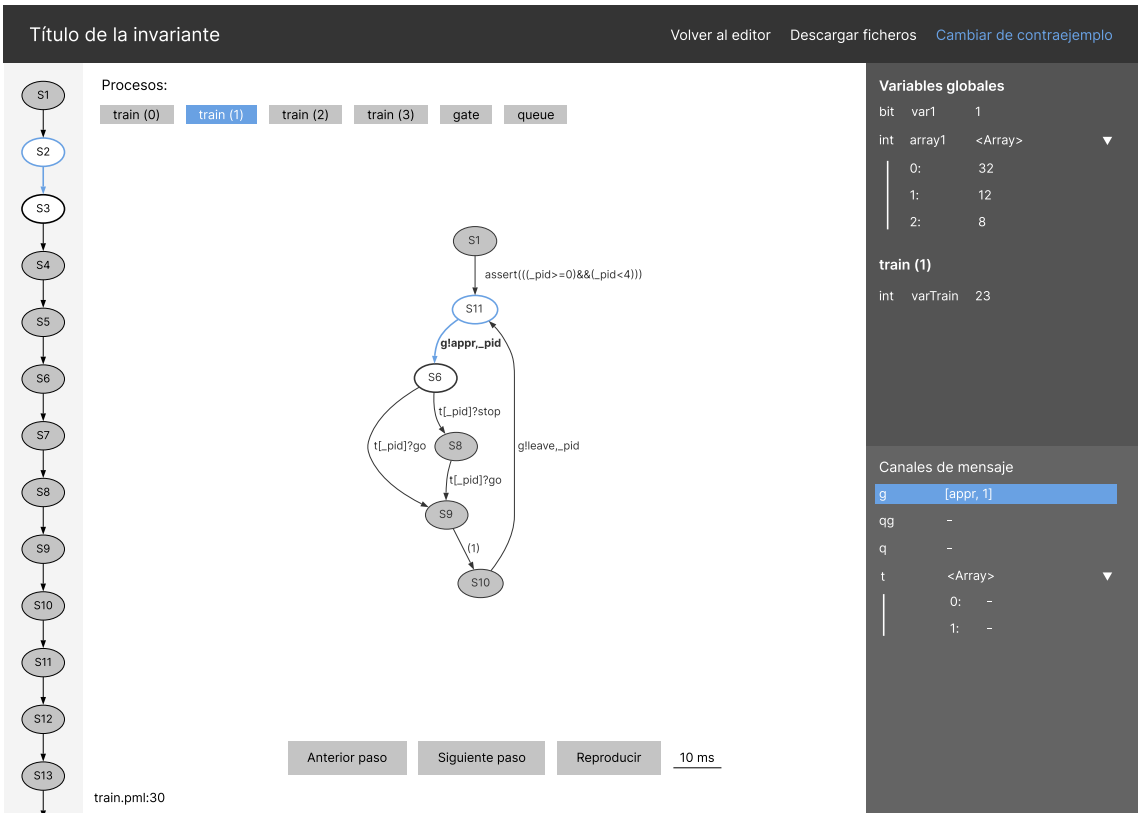

Figura 4.6: Interfaz del visualizador

Figura 4.7: Cabecera del visualizador

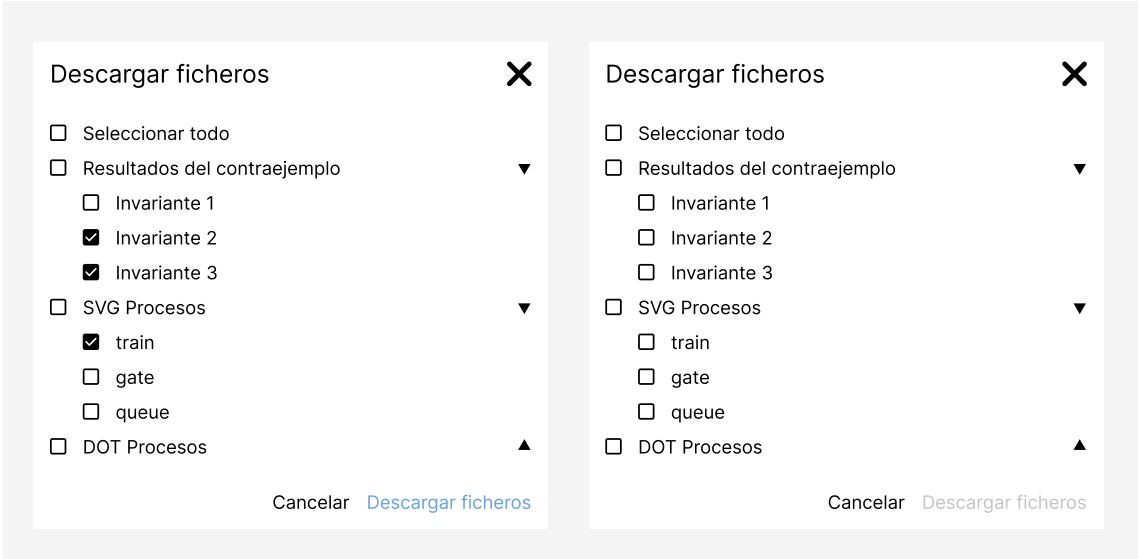

Figura 4.8: Versiones de la ventana para descargar ficheros

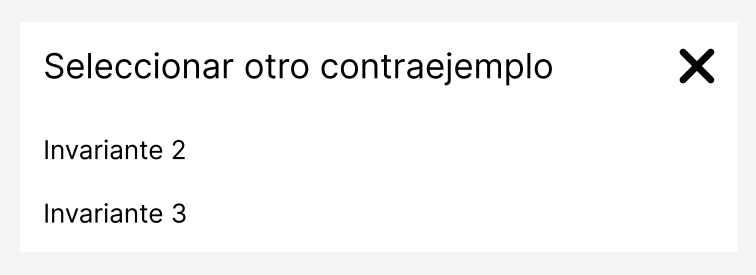

Figura 4.9: Ventana de seleccionar contraejemplo

y cada arista representa una transición de un estado del sistema a otro. Al cambiar de un estado a otro del contraejemplo, la sección de procesos se actualizará mostrando la transición de dicho proceso.

Tanto los nodos como las aristas se pueden seleccionar a través de la interfaz. La diferencia reside en que los nodos no permiten apreciar el cambio de un estado a otro porque los valores ya se han establecido, mientras que las aristas representan el cambio y los nuevos valores se indicarán en la tabla de símbolos, que es la columna de la derecha.

Como se muestra en la Figura 4.10, el primer grafo representa que el sistema se encuentra en el estado 1, mientras que el segundo grafo ilustra la transición de este al siguiente estado. El último grafo representa que ya se han establecido los datos. Para simbolizar donde se encuentra el sistema en el tiempo, se ha utilizado el color primario y un grosor distinto para distinguirlos del resto de nodos y aristas.

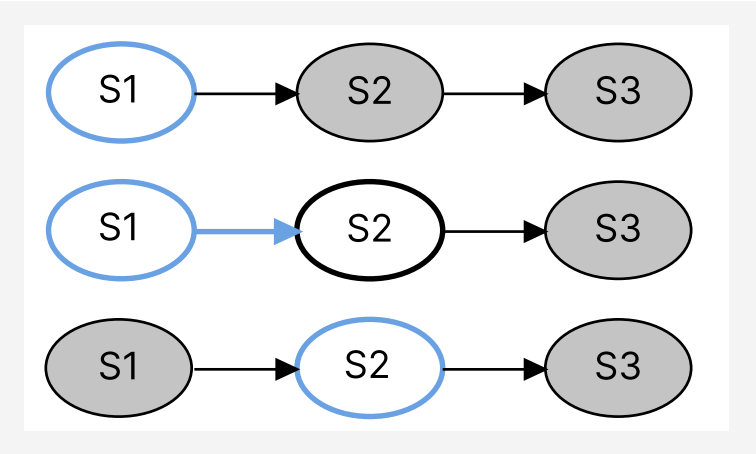

Figura 4.10: Las tres etapas de la transición entre dos estados

Cabe destacar que existen dos tipos de contraejemplos: los que alcanzan un estado indeseado o los que contienen un bucle que genera una situación indeseada. Para el primer tipo el estado indeseado se muestra en el último estado del grafo. Por otro lado, para el segundo tipo el ciclo se muestra explícitamente en el grafo y se indica con un nodo que tiene dos entradas, una es del flujo normal y la otra del ciclo. En la Figura 4.11 se pueden observar los dos tipos.

Navegación

En este apartado se detallan las opciones para navegar entre los diferentes estados del grafo. Está formado por tres botones y un campo de texto que están organizados como se indica en la Figura 4.12. Los botones son: “Anterior paso”, “Siguiente paso” y “Reproducir”. Los nombres hacen referencia a cómo se van a navegar entre los estados. Además, la

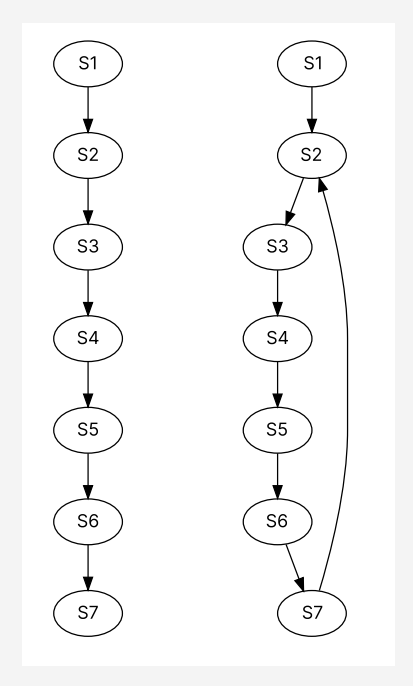

Figura 4.11: Contraejemplo con estado final y otro con un ciclo

última opción actualiza los estados de una transición a otra de manera automática. Además aparece un campo de texto adicional donde se define cuánto tiempo en milisegundos tiene que esperar el sistema entre transiciones.

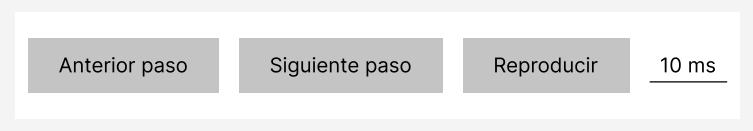

Figura 4.12: Opciones de navegación

En la Figura 4.13 se muestra que, al seleccionar el botón "Reproducir", este cambia el texto a "Detener" y el campo de texto tiene un color más claro para señalar que ya no se puede modificar. En este caso, las transiciones automáticas están activadas y para detener este proceso hay que pulsar nuevamente en el botón "Detener".

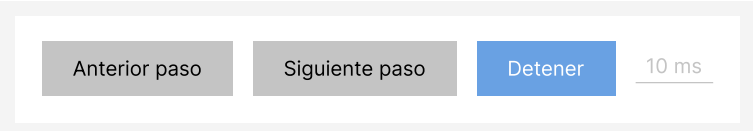

Figura 4.13: Opciones de navegación cuando se está reproduciendo el contraejemplo

Procesos

La sección de procesos está formada también por distintas partes: la lista de procesos para visualizar, el autómata del proceso seleccionado, las opciones de navegación ya explicadas y la línea de código del archivo. Como se muestra en la Figura 4.6, el foco central, tanto de este componente como el de la página, es el autómata del proceso porque es el eje central que estructura toda la información.

El grafo que se muestra es un conjunto de estados y transiciones donde cada transición contiene una etiqueta. Esta etiqueta hace referencia a la instrucción que debe realizar

el proceso para cambiar de un estado a otro. Al igual que el grafo del contraejemplo, se resaltan los estados y transiciones en los que se encuentra el proceso como se ilustra en la Figura 4.10.

Como se mencionó en la Sección 2.2, Spin simplifica el autómata original para eliminar estados y transiciones. Tras analizar varias trazas de contraejemplos, se observó que aparecen los estados eliminados. Por lo tanto, se decidió no utilizar ni el autómata simplificado ni el completo por Spin, sino una versión que une las dos partes, como indica la Figura 4.14. Este proceso se detalla más adelante, en el Capítulo 5.

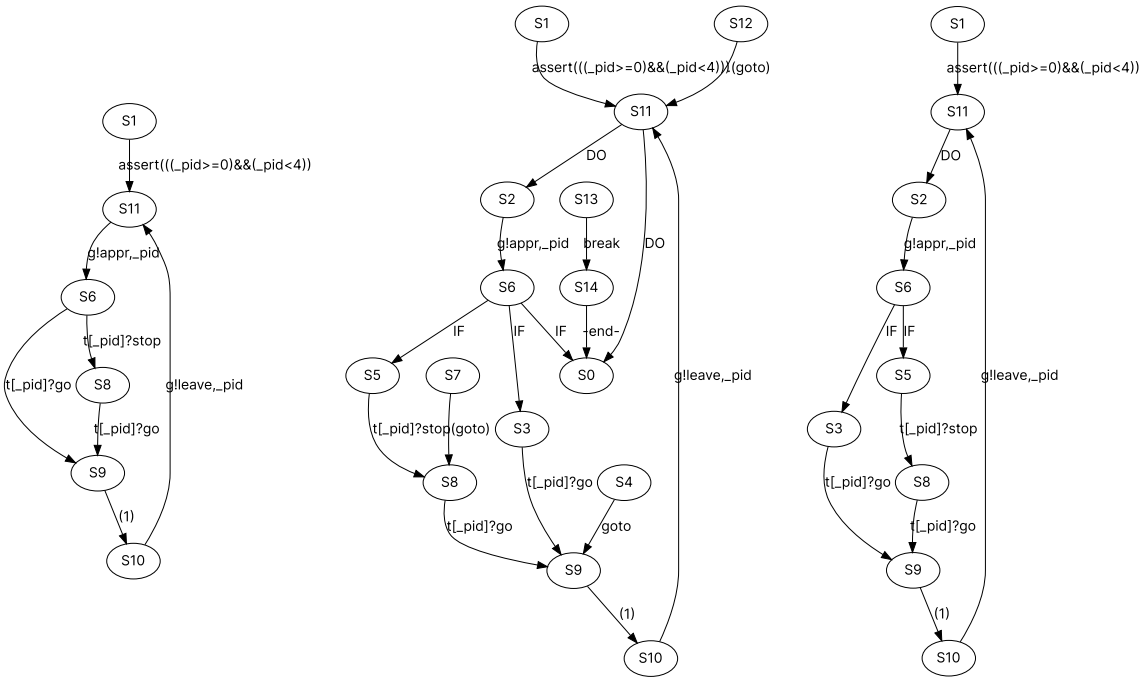

Figura 4.14: Versión simplificada, completa y la unión, respectivamente

En la parte superior de la ventana aparece la lista de procesos que contiene el código y su interfaz es la que se muestra en la Figura 4.15. Al seleccionar uno u otro, el cliente se encarga de actualizar los datos de las variables locales y mostrar el grafo correspondiente. Cabe destacar que los procesos que no han ejecutado ninguna instrucción se muestran sin ninguna transición marcada, pero sí que se marca el estado en el que se encuentra.

Una de las características de Promela es que permite iniciar varias instancias de un mismo proceso. Para que esto funcione internamente, Spin las diferencia porque cada instancia puede tener las mismas instrucciones, pero diferentes identificadores. De este modo, en la lista se muestran tantas opciones como el número de veces que se ha iniciado ese mismo proceso. Para diferenciarlos se coloca, junto al título del proceso y entre paréntesis, el identificador que proporciona Spin.

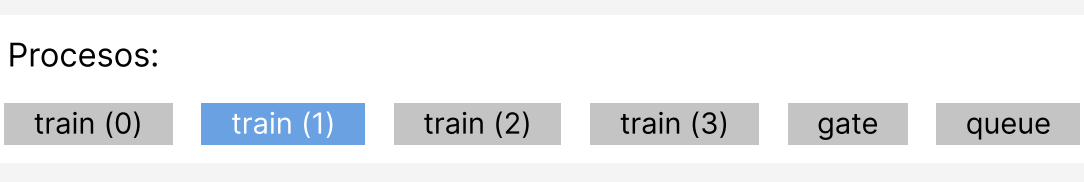

Figura 4.15: Lista de procesos

Para finalizar, en la parte inferior izquierda, al lado del contraejemplo, se muestra un texto que indica el título del código junto a un número. Este número representa en qué

línea del código se encuentra la instrucción que se ha ejecutado. Cómo indica Spin, este número puede variar del fichero original por razones de desarrollo. Como el *backend* es quien obtiene esta información de Spin, no se garantiza que dicha instrucción se encuentre donde indica este elemento.??

Variables

El listado con las variables del modelo se muestra en la sección superior de la columna de la derecha. Estas variables son las que contienen la mayoría de los datos del sistema, que se irán creando y modificando sus valores en función de las transiciones. En Promela existen dos tipos de variables: las variables globales y las variables locales. Las variables globales, como su propio nombre indica, permiten representar la información que es accesible para todos los procesos, mientras que las locales son exclusivas del proceso que las inicia.

Cada variable ocupa una fila en la sección de variables y cada una de ellas contiene el tipo de dato, su nombre y su valor. Estas filas son las que se irán modificando en función de los cambios de estado realizados. Para diferenciar del resto de variables, la interfaz resalta la fila modificada. En la Figura 4.16 se puede observar que se distinguen también dos partes. La primera parte es donde se encuentran las variables globales y la otra son las variables del proceso que se está visualizando en un momento dado por pantalla.

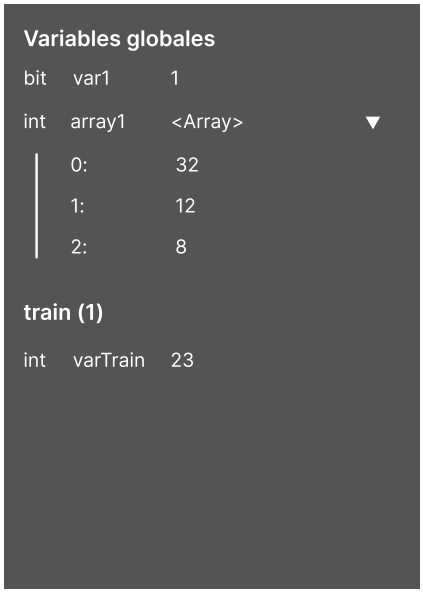

Figura 4.16: Lista de variables globales y locales del proceso `train (1)`

Existen varios tipos de datos que se pueden definir en Promela. Estos se representan con números enteros y se diferencian por el rango en que se pueden definir. Existe el *bit* (o el *bool*), el *byte*, el *short* y el *int*, siendo este último el de mayor rango. También se pueden definir constantes utilizando la palabra reservada *mtype*. Estas constantes no se muestran como variables, sino como valores de estas. Y en el caso de los mensajes, se muestran en el contenido de los mismos.

Para definir listas de tipos de datos, se utilizan los *arrays*, que son elementos que contienen conjuntos de datos del mismo tipo. Los elementos de *arrays* están ordenados y para acceder a uno de estos se utilizan los índices. El primer elemento, cómo en la mayoría de lenguajes de programación, comienza por el índice cero. En la lista de variables se muestra como un desplegable que contiene los elementos ordenados, indicando el índice

y el valor. Si existe alguna modificación en algún elemento del *array*, la interfaz resalta el elemento modificado.

Canales de mensajes

Los canales de mensajes aparecen en la sección inferior, debajo de las variables, y tienen el aspecto que ilustra la Figura 4.17. En ella se muestran los canales de mensajes que se crean en el código. Los canales de mensajes se utilizan para enviar y recibir información de diferentes procesos. Para realizar esta comunicación, se utilizan las instrucciones de enviar y recibir que se componen por el nombre del canal de mensaje y los datos que se quieren mandar o solicitar.

Los canales de mensajes contienen un *buffer* para almacenar mensajes. La cantidad de mensajes que puede contener viene definido en la inicialización del propio canal. Existe un caso concreto que merece destacarse: si el *buffer* se inicia a cero y un proceso quiere enviar un mensaje por ese canal, no se realizará si no existe algún proceso donde su siguiente instrucción corresponde a una recepción por ese mismo canal. Este tipo de canales se les conoce como canales síncronos. Las interfaces competidoras de SME que ya existen solamente muestran los mensajes que sí que tienen un *buffer* mayor que cero.

Del mismo modo que las variables, se pueden definir *arrays* de canales y siguen la misma estructura. Cada elemento del *array* es un canal disponible y la interfaz lo muestra como un desplegable parecido a los *arrays* de las variables. Cuando se guarda o se devuelve algún mensaje la interfaz se encarga también de resaltar el elemento.

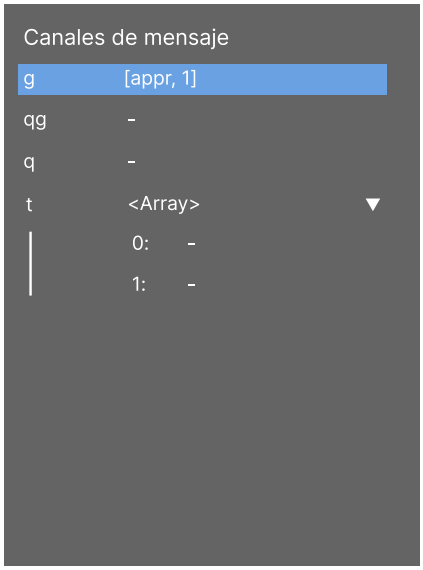

Figura 4.17: Lista de canales de mensaje

4.4 Tecnologías usadas

A día de hoy, es muy conocido que existe una gran variedad de herramientas y lenguajes para desarrollar una solución. Además, al utilizar la arquitectura cliente-servidor, se podría implementar el cliente con un lenguaje y el servidor con otro distinto. De todos modos, para el desarrollo de este trabajo se ha decidido utilizar el lenguaje JavaScript [18] en ambos casos. Entre las razones por las que se ha elegido este lenguaje, destacan que es un lenguaje popular y muy utilizado en el ámbito del desarrollo web, además de que existen entornos de desarrollo de JavaScript para crear servidores y APIs, acrónimo de

Application Programming Interface [30]. Uno de ellos es el llamado Node.js [7], que también se utiliza para el desarrollo de la API de Spin y de la aplicación de servidor.

Node.js ofrece una variedad de módulos para utilizar en los proyectos de los desarrolladores. Uno de ellos, Express [19], se utiliza para crear aplicaciones de servidor y *endpoints*. Es más, Node.js ofrece nativamente módulos para crear *clusters* y pruebas de aceptación. Esta información se amplía en el Capítulo 5 y Capítulo 6. Para complementar las pruebas de aceptación, se utiliza Mocha [15], un *framework* de JavaScript, que se puede utilizar junto a la librería de aserciones que ofrece Node.js.

Finalmente, para desarrollar el cliente se ha decidido utilizar React, que es un *framework* de JavaScript bastante popular para el desarrollo de interfaces gráficas. La principal característica reside en la reutilización de componentes, donde cada componente es un elemento visual de la página. Estos componentes tienen su propia funcionalidad y, gracias a que se pueden reutilizar en otras partes de la página, ayuda al mantenimiento de la aplicación.

Por último, se utiliza la librería de g3-graphviz [23], que está basado en la herramienta Graphviz [11]. Se utiliza para visualizar todo tipo de grafos y poder interactuar con sus elementos ofreciendo también la posibilidad de animarlos. Esta librería se encarga de generar los SVGs de los autómatas de los procesos para que luego el usuario pueda descargar dichos ficheros.

CAPÍTULO 5
Desarrollo de SME

En este capítulo se resaltan los requisitos y soluciones más relevantes que se han desarrollado para el sistema SME. Se especifica el procedimiento de cómo se ha llevado a cabo el desarrollo de estas y cuál es el resultado final.

5.1 Componentes reutilizables

Para desarrollar la interfaz, primero hay que analizarla y descomponerla en componentes hasta obtener la unidad mínima que la forma, es decir, el diseño está dividido por secciones, las secciones por subsecciones y, a su vez, estas están compuestas por elementos tales como imágenes o texto, que no se pueden dividir más. que no se pueden dividir más.

Al unir los elementos más básicos se obtienen componentes más complejos, como podrían ser los botones y formularios. Del mismo modo, al juntar componentes complejos se obtiene las secciones de la página y/o la propia página. Este proceso se muestra en la Figura 5.1. Gracias a este método se consigue un elemento visual que se puede reutilizar para nuevos componentes. Por consecuencia, si se quiere modificar alguna propiedad de una unidad básica, esta acción se propagará al resto de elementos que la componen.

Aunque este proceso facilita la creación de nuevos componentes, existe un inconveniente al crearlos, configurar los parámetros que los forman, ya que esto puede resultar una tarea muy complicada. Para solventar este problema se decidió utiliza la técnica de refactorización *Introduce Parameter Object* [17] que, en vez de recibir una larga lista de parámetros, se recibe un objeto clase que contenga estos parámetros. El resultado final es el que indica el Listado 5.1

```
1  // classes/DowloadBtnProps.js
2  class DownloadBtnProps {
3      constructor({ children, fileName, content }) {
4          this.children = children
5          this.fileName = fileName
6          this.content = content
7      }
8  }
9
10 // components/DownloadButton.jsx
11 const DownloadButton = (DownloadBtnProps) => {...}
```

Listado 5.1: Refactorización de *Introduce Parameter Object*

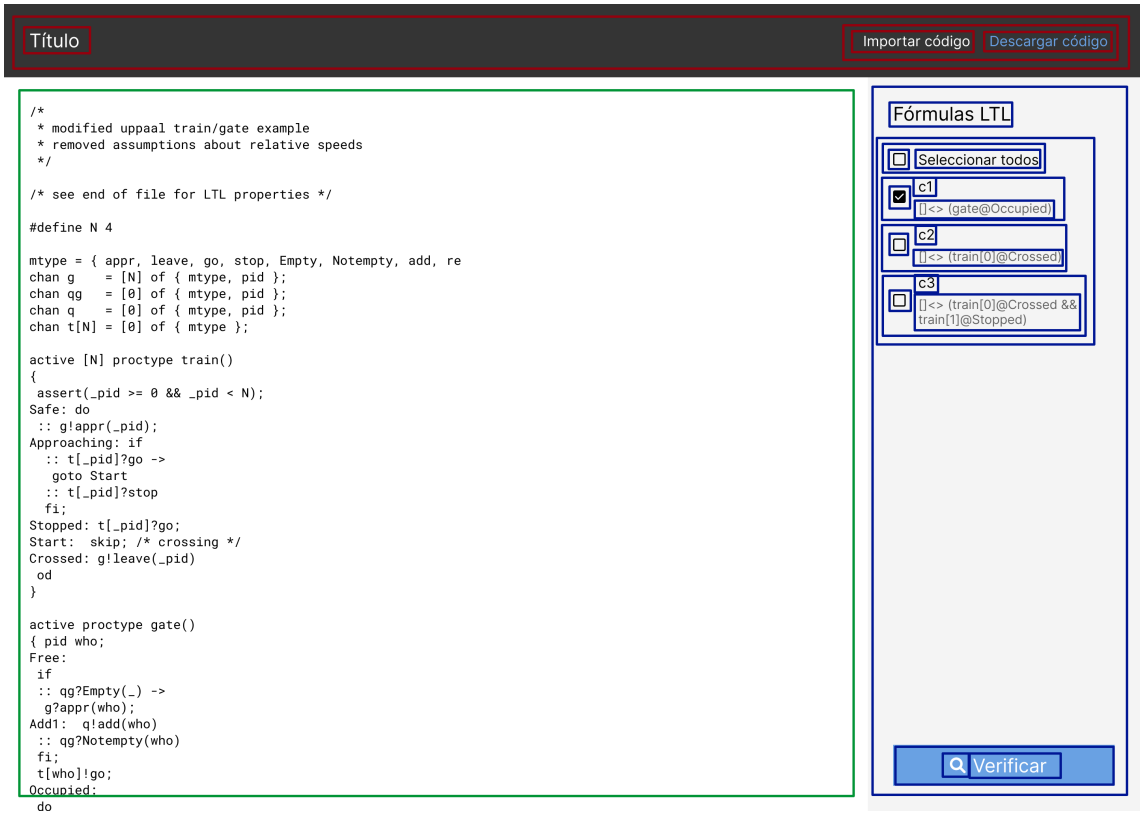

Figura 5.1: Interfaz del editor descompuesta en elementos básicos y complejos

5.2 Ejecutar Spin

Para poder ejecutar Spin en el *backend* es necesario utilizar el módulo de Node.js *Child process*. Este módulo contiene funciones para la creación y manejo de procesos a partir de un comando. Destacamos el método `exec` que es el que utilizamos para ejecutar Spin. Este método puede utilizar hasta 3 parámetros, siendo el primero obligatorio: el comando a ejecutar, la configuración y una función *callback* que se ejecuta al terminar el proceso creado. A su vez, el parámetro del *callback* debe ser una función que reciba 3 parámetros: el primero indica si ha ocurrido algún error, mientras el segundo y el tercero son, respectivamente, la salida estándar y el error estándar de la consola del proceso tras finalizar su ejecución.

El Listado 5.2 contiene las funciones para crear el *model checker* y obtener el resultado de una traza de contraejemplo. Se puede observar que la función principal, que es `getSpinbyParameterAndUserId`, devuelve una promesa [10]. Estas promesas son objetos de JavaScript que se utilizan para la creación de funciones asíncronas o para simular un comportamiento asíncrono. En nuestro caso nos es útil porque el método `exec`, al no devolver nada, ahora devuelve una promesa que, cuando se trate, permitirá obtener el *output* tras ejecutar el proceso.

```
1 import { promisify } from 'node:util'
2 import { exec } from 'node:child_process'
3
4 // Crea una Promesa a partir de una función donde uno de sus parámetros es un
      callback
5 const asyncExec = promisify(exec)
6
7 const PARAMETROS = {
8     verificar: '-search',
```

```
    contraejemplo: file => `-k ${file} -v -r -s -l -g -w`,
}

const getSpinByParameterAndUserId = (parameter, userId) => {
    // Obtiene la ruta de un usuario y el nombre del fichero .pml que contiene dado su ID
    const { userPath, file } = getUserPathAndPromelaFileByUserId(userId)

    // Para indicar donde se ejecuta el comando
    const config = { cwd: userPath }

    return asyncExec(`spin ${PARAMETROS[parameter]} ${file}`, config)
}

// Genera un ejecutable para realizar las verificaciones de propiedades
const crearVerificador = userId =>
    getSpinByParameterAndUserId(PARAMETROS.verificar, userId)

// Se obtiene en formato lista las transiciones del contraejemplo junto a las varialbes y canales de mensaje
const obtenerContraejemplo = (userId, counterFile) =>
    getSpinByParameterAndUserId(PARAMETROS.contraejemplo(counterFile), userId)

export { crearVerificador, obtenerContraejemplo }
```

Listado 5.2: Fichero de `getSpin.js`

5.3 Verificar propiedades

Antes de verificar una propiedad se debe crear el ejecutable generado por la función `crearVerificador` del Listado 5.2. A partir de este ejecutable, con el nombre `pan`, se puede verificar cualquier invariante a partir de su etiqueta. Como se muestra en el Listado 5.3, la función `verificarInvariante` indica si una invariante ha pasado o no la verificación. Esta analiza el resultado por consola y, si se detecta en una línea el texto `pan: wrote`, esto indica que no ha pasado la verificación y, en el caso contrario, sí que lo ha pasado.

```
import { promisify } from 'node:util'
import { exec } from 'node:child_process'

const asyncExec = promisify(exec)

const STATUS = {
    approved: 1,
    notApproved: 0,
}

const verificarInvariante = async (etiquetaInvariante, userPath) => {
    const config = { cwd: userPath }

    // Obtiene el resultado por consola de la verificación a partir de la etiqueta de una invariante
    const { stdout } = await asyncExec(
        `./pan -a -t${etiquetaInvariante}.trail -N ${etiquetaInvariante}`,
        config
    )

    return !stdout.includes('pan: wrote ')
        ? { status: STATUS.approved }
        : { status: STATUS.notApproved, output: stdout }
}

```

```
25 export default verificarInvariante
```

Listado 5.3: Fichero de `verificarInvariante.js`

La razón por la que el texto `pan: wrote` pone en evidencia que una propiedad no ha pasado es porque cuando una verificación resulta en fallo, el resultado por consola muestra unas líneas similares al Listado 5.4. Si la propiedad sí se cumple, no aparecerán estas líneas.

```
1 pan:1: assertion violated  !( !((q_len(list)<4))) (at depth 501)
2 pan: wrote train.pml.trail
```

Listado 5.4: Fragmento del *output* de la verificación de una invariante

5.4 Información para el visualizador

Para iniciar el visualizar se necesita la siguiente información: el grafo dirigido del contraejemplo, los grafos de los procesos, las variables y los canales de mensajes. También hay que tener en cuenta que entre los grafos de la traza y los de los procesos están relacionados a través de las transiciones, porque la transición de un estado a otro del contraejemplo se refleja como una transición de un proceso.

5.4.1. Grafos de los procesos

Como se menciona en capítulos anteriores, el grafo que va a visualizar el usuario no es un grafo que genera Spin de manera nativa. El grafo resultante es una unión entre los estados y transiciones del grafo simplificado y completo que sí que puede generar Spin. Este nuevo grafo debe contener todos los estados y transiciones del grafo simplificado y los que oculta Spin y que sí aparecen en la traza de contraejemplo. La solución para obtener esta versión nueva del grafo es partir del grafo completo y dejar únicamente como nodo que tiene padre al estado inicial del proceso. Para concluir, si el grafo simplificado no tiene un estado nombrado como S0, significa que es un proceso continuo. Por lo tanto, si es un proceso continuo, en el grafo completo también se elimina tanto el estado como las transiciones que lo alcanzan.

En el Listado 5.5 se muestra un fragmento de la salida del comando `./pan -d -d` que muestra todos los procesos del sistema junto a sus transiciones de la versión completa. La primera línea indica el nombre del proceso y sus siguientes líneas las transiciones que puede ejecutar el proceso. Analizando las líneas de las transiciones se obtiene el nodo origen y destino, la instrucción que se ejecuta y la línea donde se encuentra dicha instrucción.

```
1 STEP 2 train
2   state   1 -(tr  45)-> state  11  [id   0 tp   2] [----L] train.pml:21 =>
        assert(((_pid>=0)&&(_pid<4)))
3   state   2 -(tr  46)-> state   6  [id   1 tp   3] [----G] train.pml:23 => g!
        appr,_pid
4   state   3 -(tr  47)-> state   9  [id   2 tp 506] [----G] train.pml:25 => t[
        _pid]?go
5   state   4 -(tr   1)-> state   9  [id   3 tp   2] [----L] train.pml:26 => goto
         Start
6   state   5 -(tr  48)-> state   8  [id   4 tp 506] [----G] train.pml:27 => t[
        _pid]?stop
7   state   6 -(tr   0)-> state   0  [id   5 tp   2] [----G] train.pml:24 => IF
8   state   6 -(tr   0)-> state   3  [id   5 tp   2] [----L] train.pml:24 => IF
9   state   6 -(tr   0)-> state   5  [id   5 tp   2] [----L] train.pml:24 => IF
```

```
10  state  7 -(tr  1)-> state  8  [id  6 tp  2] [----G] train.pml:29 => .(
        goto)
11  state  8 -(tr 49)-> state  9  [id  7 tp 506] [----G] train.pml:29 => t[
        _pid]?go
12  state  9 -(tr  1)-> state 10  [id  8 tp  2] [----G] train.pml:30 => (1)
13  state 10 -(tr 50)-> state 11  [id  9 tp  3] [----G] train.pml:31 => g!
        leave,_pid
14  state 11 -(tr  0)-> state  0  [id 10 tp  2] [----L] train.pml:22 => DO
15  state 11 -(tr  0)-> state  2  [id 10 tp  2] [----L] train.pml:22 => DO
16  state 12 -(tr  1)-> state 11  [id 11 tp  2] [----L] train.pml:33 => .(
        goto)
17  state 13 -(tr  1)-> state 14  [id 12 tp  2] [----L] train.pml:22 =>
        break
18  state 14 -(tr 51)-> state  0  [id 13 tp 3500] [--e-L] train.pml:33 => -
        end-
```

Listado 5.5: Fragmento de la salida del comando `./pan -d -d -d`

5.4.2. Variables y canales de mensajes

El Listado 5.6 muestra un trozo del *output* del comando `spin -d train.pml`. Este fragmento es una tabla donde cada fila reproduce símbolos del código escrito, que están formados por las constantes (`mtype`), canales de mensajes (`chan`), procesos (`proctype`) y variables (`byte`). Cada columna indica una información del símbolo y varía según el tipo de símbolo. Por ejemplo, la tercera columna, que se puede entender como el tercer elemento separado por espacios en blanco, en el caso de los canales de mensajes indican el tamaño del *buffer*, mientras que el de procesos indica el número de instancias que se crean. Para finalizar, la información que se obtiene de esta tabla son los nombres de los símbolos, su valor y si se trata de variables globales o de un proceso.

```
1  mtype   appr  9 <:global:>  <constant>  {scope _}
2  mtype   leave 8 <:global:>  <constant>  {scope _}
3  mtype   go  7 <:global:>  <constant>  {scope _}
4  mtype   stop  6 <:global:>  <constant>  {scope _}
5  mtype   Empty 5 <:global:>  <constant>  {scope _}
6  mtype   Notempty  4 <:global:>  <constant>  {scope _}
7  mtype   add 3 <:global:>  <constant>  {scope _}
8  mtype   rem 2 <:global:>  <constant>  {scope _}
9  mtype   hd  1 <:global:>  <constant>  {scope _}
10 chan    g 4 <:global:>  <variable>  2 mtype   byte    {scope _}
11 chan    qg  0 <:global:>  <variable>  2 mtype   byte    {scope _}
12 chan    q 0 <:global:>  <variable>  2 mtype   byte    {scope _}
13 chan    t[4]  0 <:global:>  <array> 1 mtype   {scope _}
14 proctype  train 4 <:global:>  <variable>  {scope _}
15 proctype  gate  1 <:global:>  <variable>  {scope _}
16 byte    who 0 <gate>  <variable>  {scope _7_}
17 chan    list  4 <:global:>  <variable>  1 byte    {scope _}
18 proctype  queue 1 <:global:>  <variable>  {scope _}
19 byte    who 0 <queue> <variable>  {scope _9_}
20 byte    x 0 <queue> <variable>  {scope _9_}
```

Listado 5.6: Fragmento de la salida del comando `spin -d train.pml`

5.4.3. Relación entre contraejemplo y procesos

La relación entre las transiciones del contraejemplo y las de los procesos se puede obtener a partir de la traza de contraejemplo. En el Listado 5.7 se ilustra una transición del contraejemplo. El primer número indica el identificador del estado del contraejemplo en

el que se encuentra. Continua la palabra `proc`, seguido de un número que simboliza el identificador del proceso y, junto al contenido del primer paréntesis, muestra cual es el proceso relacionado con la transición. Antes del otro paréntesis, se encuentra el nombre del archivo junto a su extensión y el número de la línea donde está escrita la instrucción que está contenida al final de la línea, entre corchetes. Y el contenido del segundo paréntesis señala cuál es el estado de origen del proceso que ejecuta la instrucción. Las tres últimas líneas indican qué variables y mensajes se han modificado desde el anterior estado del contraejemplo.

```
31: proc 4 (gate:1) train.pml:52 (state 11) [g?leave,who]
    queue 3 (g):
    queue 1 (list): [3]
    gate(4):who = 3
```

Listado 5.7: Fragmento de la traza de contraejemplo

Para concluir, al analizar y procesar las salidas de los comandos descritos, se obtiene lo siguiente: el grafo de contraejemplo y los grafos de los procesos. El grafo de contraejemplo contiene los estados que, a su vez, contienen las variables o canales de mensajes que han sido modificadas y las transiciones que incluyen un estado origen y destino del contraejemplo, el identificador del proceso, el estado origen del proceso que ejecuta una instrucción y dicha instrucción. Del mismo modo, los grafos de los procesos están formados por estados y transiciones, aunque los estados solamente incluyen el nombre y las transiciones marcan un estado origen y destino, la línea donde se encuentra la instrucción y la propia instrucción que se ejecuta.

5.5 Seguridad de la aplicación de servidor

La aplicación de servidor del sistema SME tiene dos propósitos, mostrar la aplicación web y ejecutar Spin. Para ofrecer soporte a estas tareas se requieren de diferentes *endpoints* y estos son creados con las funciones que ofrece el *framework* Express. Una de las principales inquietudes de este proyecto era cómo prevenir los posibles fallos de seguridad y de rendimiento en el *backend*. Por esta razón hemos querido enfocarnos en varias posibles vulnerabilidades de nuestro sistema. Los fallos que se han contemplado son: no soportar una gran carga de solicitudes, inyectar código C, ocupar una gran cantidad de ficheros temporales y procesar solicitudes que no sean del propio sistema SME.

5.5.1. Soportar varias solicitudes al mismo tiempo

En primer lugar, la cantidad de solicitudes que puede procesar una aplicación de servidor depende de varios factores, como puede ser, los recursos de la propia máquina o cómo el programa utiliza dichos recursos de la manera más óptima posible. Como el primer factor depende de la propia máquina del servidor y no del desarrollo del sistema, nos hemos centrado en el segundo factor. Para ello Node.js ofrece un módulo nombrado *Cluster*, que se utiliza para crear subprocesos, llamados trabajadores, y repartir la carga del trabajo entre ellos. De este modo, cada trabajador contiene una instancia de la aplicación del servidor para recibir solicitudes externas y utilizar más recursos que una aplicación sin ellos [24].

5.5.2. Prevenir inyección de código a través del código Promela

Para el problema de la inyección de código en C, hay que aclarar que en Promela se puede incluir fragmentos de C para ampliar el comportamiento del sistema a modelar.

Al poder incluir dichos fragmentos, algún atacante podría generar algún fallo en el sistema SME. Como se indica desde la línea 61 hasta la 65 del Listado 2.1, se utilizan estas instrucciones para añadir fragmentos de C. Por lo tanto, para solucionar esta vulnerabilidad, tanto en el cliente como en el servidor se analiza el código escrito en busca de dichas instrucciones y fragmentos. El código que se encuentra en el Listado 5.8 devuelve *true* o *false* dependiendo si el código que se pasa por parámetro coincide o no con la expresión regular de la función, que son todas las instrucciones para incluir código C en Promela.

```
1 const tieneCodigoC = code =>
2     code.match(/c_decl|c_expr|c_decl|c_track|c_state/gm) !== null
```

Listado 5.8: Función para determinar si un código escrito en Promela contiene o no fragmentos en C

5.5.3. Eliminar archivos temporales

Otro problema a resaltar es que cuando un usuario realiza una verificación, el *backend* del sistema crea una carpeta y reconstruye el código que se quiere verificar. Después, mientras se realiza la verificación, también se genera un ejecutable y, si hay invariantes que fallan, trazas de contraejemplo. Esto ocasiona un problema de espacio si no se maneja adecuadamente este proceso. Para solucionarlo se ha decidido que, cada vez que se crea una carpeta, se utiliza la función nativa de JavaScript `setTimeout` que recibe dos parámetros: el primero es una función y el segundo marca, en milisegundos, en qué momento se va a ejecutar la función.

5.5.4. Procesar solicitudes que ejecutan Spin únicamente en el lado del cliente

Por último, otra medida de seguridad implementada es la de permitir que únicamente se puedan realizar a través del cliente las solicitudes de verificación y recuperación de la información de los contraejemplos. Esta solución trata de prevenir un ataque CSRF [3], de las siglas en inglés *Cross Site Request Forgery*. En palabras resumidas, es un ataque que se aprovecha de que algunas aplicaciones pueden aceptar y procesar solicitudes ilícitas, al no comprobar su origen.

Para prevenir este ataque se ha añadido el fragmento de código que muestra el Listado 5.9. A grandes rasgos, este fragmento es una función, que se conoce como *middleware*, que se ejecuta cuando se recibe una solicitud. Este *middleware* está configurado para comprobar el origen de cualquier solicitud que implique ejecutar Spin, verificando que coincide con la URL del *frontend*. De otro modo, el servidor rechazará la solicitud y enviará una respuesta con el código HTTP 403 (*Unauthorized*), que indica que el servidor ha recibido la solicitud, pero no la va a procesar porque carece de credenciales, que en nuestro caso indica que el usuario no está en la página del cliente.

```
1 const permitirDominio = (req, res, next) => {
2     // Donde process.env.clienteURL contiene la URL del frontend y se compara
          con el origen de la solicitud
3     if (process.env.clienteURL !== req.headers.origin) res.send(401, {auth:
          false})
4     if (req.method === 'OPTIONS') res.send(200)
5
6     res.header('Access-Control-Allow-Credentials', true)
7     res.header('Access-Control-Allow-Origin', req.headers.origin)
8     res.header('Access-Control-Allow-Methods', 'GET,POST,OPTIONS')
9     res.header('Access-Control-Allow-Headers', 'X-CSRF-Token, X-Requested-With,
          Accept, Accept-Version, Content-Length, Content-MD5, Content-Type,
          Date, X-Api-Version')
```

```
10
11      next()
12  }
```

Listado 5.9: *Middleware* para verificar si la solicitud proviene del cliente

5.6 Retos afrontados en el desarrollo de SME

A lo largo del proyecto se han encontrado algunas dificultades que ralentizaban el desarrollo del producto. Estos problemas estaban relacionados con la organización y el propio entendimiento del funcionamiento de Spin y los resultados que genera. A partir de aquí, se explica con detalle cuál fue el problema que lo ocasionó y cómo se ha solucionado.

5.6.1. Desarrollo y estructura del proyecto

Al no frecuentar el desarrollo de proyectos de la naturaleza del sistema desarrollado, el primer inconveniente apareció a la hora de dividir en dos subproyectos. En primera instancia, se planteó utilizar una herramienta que sirve para crear aplicaciones que contengan tanto la aplicación del servidor y la interfaz web. Esta herramienta es un *framework* de React llamada Next.js.

Como se explica en la Sección 4.4, React se utiliza para crear componentes e interfaces de usuario, mientras que para crear páginas web se necesita una aplicación de servidor. Si bien Next.js se encarga de ambos aspectos, al ser una tecnología que requería de una alta curva de aprendizaje, se decidió descartarla. La decisión final fue crear dos proyectos con sus controles de versiones respectivos y, a partir de ahí, crear los recursos y las pruebas para cada funcionalidad.

El siguiente problema relacionado con la estructura del proyecto es que las funciones eran bastante extensas y ocupaban gran parte de los ficheros que contienen código. Esto ocasionaba que el desarrollo se volviese más complicado en el momento de solucionar problemas que surgían mientras que se programaba. Esto ocurría tanto a la hora de crear componentes React como al crear funciones del servidor.

Para solventar este problema, se decidió separar las funciones en diferentes archivos, como muestra la Figura 5.2 donde se separa por un lado la creación del componente en un fichero y su funcionamiento en otro. De este modo se consigue que cada fichero tenga su propósito, es decir, el fichero del componente se encarga de la estructura de la interfaz, mientras que el de funcionalidad, define cómo se debe comportar dicho componente.

Por otro lado, seguía existiendo el problema de que las funciones eran bastante extensas. Por lo tanto, la decisión fue analizar cada función y separarla en subfunciones para que cada una de ellas tenga un propósito más específico. Esto se muestra en la Figura 5.3, que muestra el contenido de un fichero con el código que realiza una función específica pero a su vez tiene subfunciones que pueden entenderse como subtareas para llevar a cabo la tarea final.

5.6.2. Obtener información de los resultados de la consola

Uno de los principales retos era obtener el output de las ejecuciones de Spin y, a partir de ahí, sacar la información necesaria. La primera idea fue buscar alguna herramienta que pudiese compilar código en C a JavaScript y tener una versión de Spin implementada en

Figura 5.2: Separar estructura del componente de su funcionalidad

JavaScript. De este modo, la aplicación web simplemente serviría para visualizar la web y el cliente se encargaría del resto de funciones. Sin embargo, las herramientas existentes o estaban en fase de desarrollo y/o con muy poca documentación.

Por este motivo, esta idea se descartó en favor de utilizar el módulo *Child process* de Node.js. Este módulo se utiliza para crear y manejar procesos a partir de algún comando. Con los procesos creados por este módulo se puede obtener con facilidad los resultados que puede generar el comando utilizado. Al poder utilizar este módulo únicamente en la parte del servidor, se descartó la idea de ejecutar Spin en el cliente y su ejecución pasó a ser un trabajo del servidor.

Por último, la técnica más eficiente para definir y reconocer la sintaxis del lenguaje de modelado es utilizar las expresiones regulares. La propuesta original era utilizar Flex y Bison [28], que se utilizan para procesar un texto de entrada y obtener *tokens* que permiten posteriormente realizar alguna tarea. Uno de sus usos, por ejemplo, es crear compiladores para lenguajes de programación a partir de la gramática que define dicho lenguaje [1].

5.6.3. Obtener el grafo de unión

La solución final para este problema está documentada en la Sección 5.4. En él se define cómo Spin genera sus propios autómatas y la manera de simplificarlos para que el usuario final pueda analizar mejor el funcionamiento de sus procesos. El problema reside en que los contraejemplos que devuelven las verificaciones realizadas contienen estados ocultos que no muestra el autómata simplificado.

El objetivo es obtener una versión que junta el autómata simplificado con el autómata completo del proceso. Esta versión requiere de los estados intermedios que oculta Spin y corresponde a las instrucciones `if`, `do` y `atomic`. Por ejemplo, en la Figura 2.1 se puede observar que entre los estados S11 y S6 no hay ningún estado entre medias en el grafo simplificado, mientras que en el completo sí, el estado S2.

En primer lugar, se intentó utilizar los conocimientos de teoría de grafos aprendidos en el grado. Para ello se planteó crear un grafo con todos los nodos y aristas. En la Figura 5.4, en el caso de S11 a S6 se observa que hay dos posibles caminos: uno desde directo S11

```
> const getTransitionLineNumberFromLine = line => {…
  }

> const getInstructionFromLine = line => {…
  }

> const getStateLabelsFromLine = line => {…
  }

> /**…
> const lineHasTransition = line => {…
  }

> const getProcessGraphFromOutput = output => {…
  }

> const getProcessGraphsFromOutput = output => {…
  }

  export default getProcessGraphsFromOutput
```

Figura 5.3: Función de obtener los grafos a partir del *output*, dividida en subfunciones

a S6 y otro que recorre S11, S2 y S6. Para obtener el grafo de unión se necesita el camino más largo entre S11 y S6.

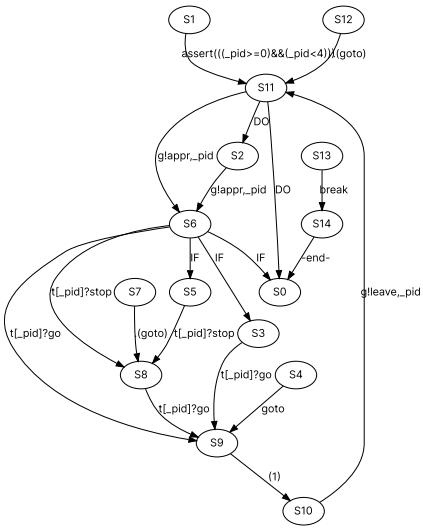

Figura 5.4: Unión de todas las transiciones y estados del grafo simplificado y completo

Sin embargo, esto plantea otro problema, ya que desde S6 a S9 hay 4 caminos posibles: [S6, S9], [S6, S8, S9], [S6, S3, S9] y [S6, S5, S8 y S9]. Pero esto ocurre porque entre el camino S6 a S9 existe un estado oculto y desde S6 a S8 hay otro. Por lo tanto, si se crea una función para obtener el camino más largo este obtendría resultados no deseados.

Por lo tanto, se rechazó esta aproximación y se empezó a plantear el problema desde otra perspectiva. Al final la solución era más simple y residía en la pregunta: qué diferencia hay entre los caminos del grafo simplificado y el grafo completo. Se observó que, a diferencia del grafo simplificado, no existe más de un nodo sin padre, es decir, el nodo inicial. Por lo tanto, había que eliminar los nodos sin padre y repetir este proceso hasta que solo el nodo inicial esté sin padre.

Además, todos los procesos tienen el estado final que se muestra como S0 pero, al poder crear procesos que no tienen fin, este último nodo, que sí aparece en el grafo com-

pleto, no debería aparecer en el grafo simplificado. Para concluir, si el grafo simplificado no tiene el nodo S0, la versión deseada tampoco debe tener este nodo, como se muestra en la Figura 4.14.

5.6.4. Obtener los procesos, las variables y los canales de mensajes

El último reto a superar era obtener la información de cada estado: los procesos, las variables y los canales de mensajes. Para ello, Spin ofrece diferentes formas de mostrar estos datos a través de utilizar diferentes parámetros, cómo se muestra en la Sección 5.4.

Para ello había que tener en cuenta varios aspectos, como, por ejemplo, cómo averiguamos si varios procesos se han iniciado a partir de un mismo código, si una variable es global o local de un proceso o cómo se actualiza la comunicación entre procesos. En primer lugar se plantearon dos ideas: la primera, obtener la información a través del código y la segunda, a través de los comandos de Spin.

La primera opción parece la más sencilla a primera vista pero, gracias a los parámetros que se pueden utilizar, se puede obtener una tabla de todos los símbolos e incluso (con las opciones correctas a partir del contraejemplo) obtener los datos en cada actualización del estado global. Por lo tanto, se descartó la primera idea y se empezó a desarrollar a partir de la segunda.

A partir de aquí, obtener las variables, tanto globales como locales, y los canales de mensajes es una tarea sencilla. El principal problema que se ocasionaba era cómo saber el identificador de cada instancia que se creaba de un mismo proceso. A partir del comando `spin -d nombre-fichero.pml` se puede ver el número de instancias que se crean, pero no sus identificadores.

Para solucionarlo, en el bucle para obtener cada estado del contraejemplo, se puede observar que la primera línea, que corresponde a la instrucción que se ha ejecutado, se encuentra la palabra `proc` seguido de un número y el nombre de un proceso. Ese número indica el identificador del proceso que, junto al número de instancias que hay del mismo, se debe guardar en una lista con el mismo tamaño del número de instancias.

CAPÍTULO 6
Pruebas

En este capítulo se describen las pruebas realizadas en el sistema SME de las tareas de servidor que consideramos más relevantes. Estas pruebas tratan de verificar que los resultados devueltos por el *backend* son correctas. Para ello hemos analizado de manera manual el modelo del Apéndice A para demostrar que los resultados generados por el sistema son iguales.

6.1 Preparación

Los requisitos que se van a comprobar si son correctos son: verificar invariantes y generar el grafo de unión. Para el primer requisito se requiere comprobar que, dado un modelo con diferentes propiedades definidas, las invariantes que hayan pasado y las que no, el servidor muestre el mismo resultado cuando se verifican de la manera tradicional. Para el segundo, es necesario definir el grafo deseado de manera manual analizando: un contraejemplo y los autómatas, tanto simplificados como completos, de los procesos modelados.

Para generar y organizar pruebas de funcionamiento, el *framework* Mocha ofrece funciones, tales como, `it` y `describe`. La primera función se utiliza para crear una prueba y el segundo comando permite agruparlas. Seguir esta estructura nos ayuda a la mantenibilidad de las pruebas, por ejemplo, para crear nuevas pruebas relacionadas o modificar alguna prueba de manera sencilla. Para crear estas pruebas se necesitan de algún modo definir cuando se ha pasado una prueba o no. Para ello se utilizan las librerías de aserciones, que en nuestro caso son las del módulo de Node.js *Assert*. De esta librería usamos las funciones `deepStrictEqual` y `strictEqual`. La función de `strictEqual` se utiliza para comprobar si dos valores son el mismo y se utilizan para las variables del tipo `String`, `Boolean` o `Number`, entre otros. La diferencia entre esta función y `deepStrictEqual` es que, en JavaScript, dos objetos no se pueden comparar del mismo modo que, por ejemplo, dos cadenas de texto y esto se debe a la propia naturaleza de JavaScript porque, al final, los requisitos que se comprueban dan como resultado un objeto.

6.2 Verificar invariantes

Como se menciona en la introducción del capítulo, para crear casos de prueba se utiliza el modelo del Apéndice A. Para los casos de prueba se han juntado las invariantes y los resultados esperados de sus verificaciones en un objeto, como el que debe devolver la función de `verificarInvariante`. Al ejecutar y comprobar que han pasado las pruebas descritas en el Listado 6.1 se puede concluir que el método funciona correctamente.

```
const resultado = {
    c1: { status: STATUS.approved },
    c2: { status: STATUS.notApproved },
    c3: { status: STATUS.notApproved },
    c4: { status: STATUS.notApproved },
    c5: { status: STATUS.approved },
    c6: { status: STATUS.notApproved },
    c7: { status: STATUS.approved },
    c8: { status: STATUS.approved },
}

describe('Verificar propiedades del modelo de ejemplo', () => {
    it('Invariante c1', async () => {
        const actual = await verificarInvariante('c1', userPath)
        const expected = resultado.c1

        assert.strictEqual(actual.status, expected.status)
    })

    it('Invariante c2', async () => {
        const actual = await verificarInvariante('c2', userPath)
        const expected = resultado.c2

        assert.strictEqual(actual.status, expected.status)
    })

    ...

    it('Invariante c8', async () => {...})
})
```

Listado 6.1: Pruebas para comprobar la verificación de propiedades

6.3 Generar el grafo de unión

Para los casos de prueba de generar el grafo que se desea mostrar como, por ejemplo, el que se muestra en la Figura 4.14, hay que crear un grafo a partir de las transiciones y estados deseados, junto a los nombres que también deben coincidir. Por lo tanto los casos de prueba del modelo del Apéndice A son los tres procesos modelados: `train`, `gate` y `queue`. Después de ejecutar los comandos que se mencionan en la Sección 5.4 para obtener la información de los grafos simplificados y completos de los procesos, se crea manualmente el grafo de unión deseado y se compara con el grafo generado por las funciones desarrolladas. En el Listado 6.2 se muestran tres funciones `it` que definen los tres procesos descritos.

```
describe('Verificar grafo de unión', () => {
    it('Proceso del train', () => {
        // Devuleve un array de ProcessGraph a partir del resultado de consola
            de los comandos ./pan -d y ./pan -d -d -d
        const graphs = getProcessGraphsFromOutput(outputTrain)

        // Devuelve el grafo de unión siendo graph[0] el grafo simplificado y
            graph[1] el completo
        const formatedGraph = getFormatedProcessGraph(graphs[0], graphs[1])

        // Crear un ProcessGraph donde trainStates y trainTransitions son los
            estados y transiciones que debe tener el grafo de unión esperado
        const expectedGraph = new ProcessGraph('train', trainStates,
            trainTransitions)
```

```
11
12          assert.deepStrictEqual(formatedGraph, expectedGraph)
13      })
14      it('Proceso del gate', () => {...})
15      it('Proceso del queue', () => {...})
16  })
```

Listado 6.2: Pruebas para comprobar la generación de grafos de unión

CAPÍTULO 7
Conclusiones

El objetivo de este capítulo es doble: exponer los resultados y conclusiones que se pueden extraer de este trabajo y definir qué conocimientos y utilidades se han podido utilizar en el proyecto gracias a las materias cursadas por el alumno en el grado en Ingeniería Informática. Para complementar los resultados también se mencionan algunos inconvenientes a los que el alumno se ha tenido que enfrentar para asegurar el correcto funcionamiento del sistema.

7.1 Resultados

Para empezar, en la Sección 1.2 y Sección 3.3, se han definido objetivos y requisitos que debe cumplir este trabajo. Recapitulando, SME es una aplicación web que está dividida en dos partes: el cliente y el servidor, donde el servidor gestiona el proceso de verificación y el cliente se encarga de visualizar los datos obtenidos a partir de los resultados del servidor.

El objetivo principal es el desarrollo de la aplicación web en su totalidad cumpliendo los requisitos definidos: que soporte la verificación de invariantes y visualizar los contraejemplos de una manera sencilla, entre otros aspectos. A través de las encuestas, se obtuvo que los alumnos requerían de poder visualizar de una manera gráfica los estados de los contraejemplos y se llegó a la conclusión que la mejor forma de hacerlo era utilizar grafos dirigidos.

Al terminar los dos sistemas y de comprobar que las pruebas a partir de los requisitos han resultado correctas, se puede concluir que el proyecto se ha terminado satisfactoriamente. A continuación, se explican los principales desafíos e inconvenientes que se han encontrado y qué propuestas han llevado a su resolución.

7.2 Relación del trabajo desarrollado con los estudios cursados

A lo largo de la carrera, el alumno de la rama de Ingeniería del Software obtiene una serie de conocimientos que se pueden llegar a apreciar en este trabajo. A continuación, voy a comentar qué asignaturas del grado y qué conocimientos he utilizado para terminar el proyecto con éxito, siendo estas asignaturas los pilares principales de mi conocimiento actual sobre desarrollo software y creación de proyectos.

- MFI: La primera asignatura que quiero destacar, es la de Métodos Formales Industriales. Sin los fundamentos estudiados en ella y las prácticas de laboratorio, este

trabajo ni se hubiera planteado. Me han enseñado sobre las características de la verificación de sistemas críticos y su importancia para la seguridad de las personas y las organizaciones. Mientras realizaba los ejercicios del laboratorio, la herramienta que más me llamó la atención fue Spin por sus diferentes opciones y que el lenguaje que usa, Promela, era a la vez potente y muy sencillo de aprender. Viendo el potencial de esta herramienta, sentía que si tuviese una interfaz más amigable, más personas se sentirían atraídas del mismo modo que yo.

- MAD y TAL: Gracias a estas asignaturas surgió la idea original de mostrar los procesos en el formato de grafo dirigido. Además, en MAD se enseñan los problemas y algoritmos más famosos de grafos, como, por ejemplo, el camino más corto, búsqueda en anchura (BFS) y búsqueda en profundidad (DFS), que fueron un buen comienzo para entender el problema del grafo de unión. También gracias a TAL pude entender de una manera más fácil los conceptos de autómatas que se documentan en los libros de Spin, además de que también se aprenden los conceptos de las expresiones regulares.

- IPC: En la asignatura de Interfaces Persona Computador aprendí los principios de un buen diseño de interfaz. Por ejemplo, en las interfaces diseñadas se puede apreciar que, para separar cada sección, cada una tiene un tono de gris distinto, que se puede percibir como un bloque de información diferente al resto.

- AER: A partir de aquí, el resto de asignaturas corresponden a la rama de Ingeniería de Software y están relacionadas a la creación y gestión de proyectos de una aplicación software. Por ejemplo, destacaría de Análisis y Especificación de Requisitos, cuyo objetivo es definir cómo se debe documentar y obtener las diferentes funcionalidades que debe tener la aplicación a través de técnicas estandarizadas.

- DDS: En de Diseño de Software se estudian los patrones de diseño para favorecer un mejor entendimiento del código. En este proyecto se usó estos conocimientos para obtener un código y estructura más organizado, como utilizar objetos para los parámetros de funciones y separar en subtareas las funciones. Este último, además ayudó a la creación de las pruebas.

- GPR, PSW y PIN: Las tres asignaturas tratan sobre las diferentes fases de la creación de principio a fin de una aplicación software, siendo Gestión de Proyectos el que muestra los conceptos y técnicas del desarrollo tradicional y Proceso Software y Proyecto de Ingeniería Software los encargados del desarrollo ágil.

Para concluir, mencionar algunas asignaturas que también han estado presentes, casi al mismo nivel que las anteriores.

- CSD: Concurrencia y Sistemas Distribuidos me enseñó las ventajas e inconvenientes de compartir la memoria entre diferentes procesos. En el caso del proyecto existe el módulo de Cluster de Node.js que se encarga de separar en diferentes "trabajadores" y repartir los recursos para tener un mejor rendimiento y abarcar más solicitudes.

- CSO: Por último, los conocimientos obtenidos a través de Calidad de Software me enseñaron las propiedades que debe cumplir una aplicación a través de estándares.

7.3 Trabajo futuro

Ahora que el trabajo ha finalizado, es momento de resumir algunas ideas que quedaron aparcadas y que podrían mejorar la aplicación. Algunas se derivan de los reclamos de los alumnos que realizaron la encuesta que se muestran en la Tabla 3.1 y otras son algunas características avanzadas que si se incluyen en las interfaces anteriores a este trabajo de la Tabla 2.1. Además, mientras se desarrollaba el proyecto, también surgieron nuevas ideas y en este capítulo se detallan las que consideramos más relevantes.

7.3.1. Verificar sintaxis y autocompletado on-line

Una de las características más solicitadas y relacionadas con el editor es la de ver los errores de sintaxis mientras se escribe y el autocompletado de instrucciones y variables. En el caso de verificar la sintaxis del código en tiempo real, se debería estudiar la gramática del lenguaje de Promela. Afortunadamente ya existe tal gramática que define cómo se estructura el código y que está disponible en:

`https://spinroot.com/spin/Man/grammar.html`

A partir de esta gramática se podría comprobar en tiempo real si el código está escrito correctamente. Después, si se encuentra algún error de sintaxis en el código, el editor lo marcaría y, si está configurado, mostraría una posible solución, como los editores actuales para algunos lenguajes de programación.

De hecho, algunos entornos de desarrollo actuales tienen implementados sistemas de autocompletado, que se encargan de reconocer nombres de variables, tipos de datos, nombres de funciones, parámetros de estas, etc. En el caso de Promela, se debería obtener primero las instrucciones disponibles y su estructura a partir de la gramática. Posteriormente, cuando el usuario escriba código, el cliente se encarga de guardar las palabras que ha usado para las variables, canales de mensaje y procesos que podría consultarse para ofrecer el autocompletado.

7.3.2. Ver el código desde el propio visualizador

Recordemos que cuando Spin muestra los estados del contraejemplo o cuando se obtiene los estados del autómata de un proceso, la transición de cada estado corresponde a la ejecución de una instrucción. Cada instrucción está seguida del nombre del fichero junto a su extensión y el número de la línea donde se encuentra. A partir de esta información, en el visualizador se podría mostrar, en cada transición, parte del código involucrado o el código completo para que el usuario final pueda seguir la traza.

7.3.3. Pruebas de aceptación a usuarios reales

Por motivos obvios, debido a la realización de este proyecto como trabajo fin de grado no han podido realizarse pruebas de aceptación de sus usuarios al haber concluido el curso en el momento de finalización del trabajo. El resultado de dichas pruebas podría permitir, en un futuro, mejorar este trabajo incorporando las mejoras obtenidas a partir del `feedback` de los potenciales destinatarios.

Bibliografía

[1] Anthony A Aaby. Compiler construction using flex and bison. *Walla Walla College*, 2003.

[2] María Alpuente, Demis Ballis, Francisco Frechina, and Julia Sapiña. Exploring conditional rewriting logic computations. *Journal of Symbolic Computation*, 69:3–39, 2015.

[3] Robert Auger. The Cross-Site Request Forgery (CSRF/XSRF) FAQ — cgisecurity.com. https://www.cgisecurity.com/csrf-faq.html, 2010. [Accessed 10-Jun-2022].

[4] Moti Ben-Ari. jspin–java gui for spin: User's guide, 2010.

[5] Tim Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 8259, December 2017.

[6] Zmago Brezočnik, Boštjan Vlaovič, and Aleksander Vreže. Spinrcp: The eclipse rich client platform integrated development environment for the spin model checker. In *Proceedings of the 2014 International SPIN Symposium on Model Checking of Software*, SPIN 2014, page 125–128, New York, NY, USA, 2014. Association for Computing Machinery.

[7] Mike Cantelon, Marc Harter, TJ Holowaychuk, and Nathan Rajlich. *Node. js in Action*. Manning Greenwich, 2014.

[8] J. Cardoso and C. Sibertin-Blanc. Ordering actions in sequence diagrams of uml. In *Proceedings of the 23rd International Conference on Information Technology Interfaces, 2001. ITI 2001.*, pages 3–14 vol.1, 2001.

[9] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martı-Oliet, José Meseguer, and Carolyn Talcott. Maude manual (version 2.1). *SRI International, Menlo Park (April 2005)*, 2004.

[10] Michael E Colley. Demystifying promises in javascript: How do they work? — betterprogramming.pub. https://betterprogramming.pub/demystifying-promises-in-javascript-how-do-they-work-a6b8c5dcb85f, 2021. [Accessed 10-Jun-2022].

[11] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C. North, and Gordon Woodhull. Graphviz— open source graph drawing tools. In Petra Mutzel, Michael Jünger, and Sebastian Leipert, editors, *Graph Drawing*, pages 483–484, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[12] Joyce Farrell. *Java programming*. Cengage Learning, 2011.

[13] Jon Ferraiolo, Fujisawa Jun, and Dean Jackson. *Scalable vector graphics (SVG) 1.0 specification*. iuniverse Bloomington, 2000.

[14] Clif Flynt. *Tcl/Tk: A Developer's Guide*. Elsevier, 2012.

[15] OpenJS Foundation. Mocha - the fun, simple, flexible JavaScript test framework — mochajs.org. https://mochajs.org/, 2022. [Accessed 10-Jul-2022].

[16] Martin Fowler. *Patterns of Enterprise Application Architecture: Pattern Enterpr Applica Arch*. Addison-Wesley, 2012.

[17] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[18] Juan Diego Gauchat. *El gran libro de HTML5, CSS3 y Javascript*. Marcombo, 2012.

[19] Evan Hahn. *Express in Action: Writing, building, and testing Node. js applications*. Simon and Schuster, 2016.

[20] Mark Hansen. DOT Language — graphviz.org. https://www.graphviz.org/doc/info/lang.html, 2022. [Accessed 10-Jun-2022].

[21] Gerard Holzmann. Getting Started: Using iSpin — spinroot.com. http://spinroot.com/spin/Man/3_SpinGUI.html, 2014. [Accessed 10-Jun-2022].

[22] Gerard J. Holzmann. The model checker spin. *IEEE Transactions on software engineering*, 23(5):279–295, 1997.

[23] Magnus Jacobsson. GitHub - magjac/d3-graphviz: Graphviz DOT rendering and animated transitions using D3 — github.com. https://github.com/magjac/d3-graphviz. [Accessed 10-Jun-2022].

[24] Eric Johansson. Using cluster module in node.js for increased performance. *Linnaeus University, Department of computer science and media technology (CM)*, 2021.

[25] John C. Knight. Safety critical systems: Challenges and directions. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, page 547–550, New York, NY, USA, 2002. Association for Computing Machinery.

[26] Gerald Kotonya and Ian Sommerville. *Requirements Engineering: Processes and Techniques*. John Wiley & Sons, 2004.

[27] Yu Beng Leau, Wooi Khong Loo, Wai Yip Tham, and Soo Fun Tan. Software development life cycle agile vs traditional approaches. In *International Conference on Information and Network Technology*, volume 37, pages 162–167. IACSIT Press, Singapore, 2012.

[28] John R. Levine. *Flex & Bison*. O'Reilly, 2013.

[29] Henrik Nielsen, Jeffrey Mogul, Larry M Masinter, Roy T. Fielding, Jim Gettys, Paul J. Leach, and Tim Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, June 1999.

[30] Joshua Ofoeda, Richard Boateng, and John Effah. Application programming interface (api) research: A review of the past to inform the future. *International Journal of Enterprise Information Systems (IJEIS)*, 15(3):76–95, 2019.

[31] Haroon Shakirat Oluwatosin. Client-server model. *IOSR Journal of Computer Engineering*, 16(1):67–71, 2014.

[32] Jeremy Reimer. A history of the gui. *Ars Technica*, 5:1–17, 2005.

[33] Theo C Ruys. Xspin/project-integrated validation management for xspin. In *International SPIN Workshop on Model Checking of Software*, pages 108–119. Springer, 1999.

APÉNDICE A
Modelo de un sistema ferroviario

El modelo es un ejemplo que describe el control ferroviario de un sistema que maneja el acceso a una estación para diferentes trenes. Este modelo se utiliza a lo largo de la memoria de modo de ejemplo. Contiene las instrucciones más relevantes: procesos, canales de mensajes, envío y recibo de mensajes, variables e instrucciones `if` y `do`

```
/*
 * modified uppaal train/gate example
 * removed assumptions about relative speeds
 */

/* see end of file for LTL properties */

#define N 4

mtype = { appr, leave, go, stop, Empty, Notempty, add, rem, hd };


chan g    = [N] of { mtype, pid };
chan qg   = [0] of { mtype, pid };
chan q    = [0] of { mtype, pid };
chan t[N] = [0] of { mtype };

active [N] proctype train()
{

  assert(_pid >= 0 && _pid < N);

Safe:   do
    :: g!appr(_pid);
Approaching:    if
       :: t[_pid]?go ->
         goto Start
       :: t[_pid]?stop
       fi;
Stopped:  t[_pid]?go;
Start:    skip; /* crossing */
Crossed:  g!leave(_pid)
    od
}

active proctype gate()
{ pid who;
Free:
  if
  :: qg?Empty(_) ->
    g?appr(who);
Add1:   q!add(who)
```

```
43    :: qg?Notempty(who)
44    fi;
45
46    t[who]!go;
47
48  Occupied: do
49      :: g?appr(who)  ->
50         t[who]!stop;
51  Add2:      q!add(who)
52      :: g?leave(who) ->
53         q!rem(who);
54         goto Free
55      od
56  }
57
58  chan list = [N] of { pid };
59
60  active proctype queue()
61  {
62
63  pid who, x;
64
65  Start:
66    if
67    :: nempty(list) ->
68       list?<who>;
69       qg!Notempty(who)
70    :: empty(list) ->
71       qg!Empty(0)
72    fi;
73    do
74    :: q?add(who) ->
75       list!who
76    :: q?rem(who) ->
77  Shiftdown:  list?x;
78       assert(x == who);
79       goto Start
80    od
81  }
82
83  /*
84   * ltl format specifies positive properties
85   * that should be satisfied -- spin will
86   * look for counter-examples to these properties
87   */
88
89  ltl c1  { []<> (gate@Occupied) }
90  ltl c2  { []<> (train[0]@Crossed) }
91  ltl c3  { []<> (train[0]@Crossed && train[1]@Stopped) }
92  ltl c4  { []<> (train[0]@Crossed && train[1]@Stopped && train[2]@Stopped &&
        train[3]@Stopped) }
93  ltl c5  { [] (train[0]@Crossed + train[1]@Crossed + train[2]@Crossed + train[3]
        @Crossed <= 1) }
94  ltl c6  { [] (len(list) < N) }
95
96  ltl c7  { [] (((gate@Add1 || gate@Add2)) -> (len(list) < N)) }
97  ltl c8  { [] (train[0]@Approaching) -> <> (train[0]@Crossed) }
```

ANEXO

OBJETIVOS DE DESARROLLO SOSTENIBLE

Grado de relación del trabajo con los Objetivos de Desarrollo Sostenible (ODS).

Objetivos de Desarrollo Sostenible	**Alto**	**Medio**	**Bajo**	**No procede**
ODS 1. **Fin de la pobreza.**				x
ODS 2. **Hambre cero.**				x
ODS 3. **Salud y bienestar.**				x
ODS 4. **Educación de calidad.**	x			
ODS 5. **Igualdad de género.**				x
ODS 6. **Agua limpia y saneamiento.**				x
ODS 7. **Energía asequible y no contaminante.**				x
ODS 8. **Trabajo decente y crecimiento económico.**	x			
ODS 9. **Industria, innovación e infraestructuras.**		x		
ODS 10. **Reducción de las desigualdades.**	x			
ODS 11. **Ciudades y comunidades sostenibles.**				x
ODS 12. **Producción y consumo responsables.**				x
ODS 13. **Acción por el clima.**				x
ODS 14. **Vida submarina.**			x	
ODS 15. **Vida de ecosistemas terrestres.**			x	
ODS 16. **Paz, justicia e instituciones sólidas.**				x
ODS 17. **Alianzas para lograr objetivos.**				x

- **Educación de calidad**, creo que está estrechamente relacionado con la creación de este proyecto. La razón es que la idea surgió del descontento al utilizar una aplicación que muestra los resultados por una consola y que, en el momento de analizar y comprender estos datos, se volvía una tarea que requería de conocimientos más avanzados. Este hecho dificultaba la tarea de utilizar los conceptos estudiados en teoría y que se aplican a dicha aplicación. Gracias a este trabajo se quiere conseguir que los nuevos usuarios sientan que tienen el control y centrarse en analizar y comprender sus modelos. Además, este nuevo sistema permite proporciona asistencia a un usuario que conoce los conceptos clave sin tener conocimientos más avanzados de la herramienta y los fundamentos en los que está basada.

- **Reducción de las desigualdades**, también considero que este proyecto cumple este ODS. La razón es porque el sistema desarrollado es una aplicación web, es decir, que se puede utilizar en cualquier dispositivo que tenga acceso a internet. Al desarrollar este tipo de aplicaciones se consigue que, independientemente del sistema operativo y de los recursos de los dispositivos, se puede utilizar sin ninguna complicación, que, a diferencia de las aplicaciones de escritorio, dependen de dichos factores.

- **Trabajo decente y crecimiento económico**, este ODS se puede relacionar con este proyecto porque está basado en una de las herramientas de verificación formal, Spin, que se utilizan para comprobar la seguridad y el correcto funcionamiento de sistemas críticos industriales. La importancia de estas herramientas es crucial porque aseguran la protección frente a fallos que podrían conllevar a grandes perdidas materiales e incluso humanas. El uso de esta aplicación con productos o servicios, que requieran un alto nivel de seguridad, puedan prevenir posibles fallos importantes.

- **Industria, innovación e infraestructuras**, se podría pensar que como este proyecto busca la facilidad de uso y que se pueda utilizar en cualquier lugar con acceso a internet, podría influir en la concienciación en invertir más recursos en crear este tipo de *software*. De este modo, las aplicaciones que se utilizan en las industrias o comercios que están desactualizadas o que tienen implementado interfaces difíciles de entender ayudaría que los usuarios tuvieran una mejor productividad.

- **Vida submarina** y **Vida de ecosistemas terrestres**, son ODSs que podrían estar relacionadas con el sistema desarrollado, porque existen sistemas críticos que podrían afectar negativamente al medio ambiente, como podría ser, sistemas de control de un navío que contiene alguna carga que puede contaminar o sistemas de seguridad de centrales nucleares y que si surge algún fallo informático podrían llevar a una catástrofe medioambiental. Por eso en este trabajo se fomenta el uso de las herramientas de verificación de propiedades.