ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADO EN INGENIERÍA DEL SOFTWARE

MODELADO, SIMULACIÓN Y ANÁLISIS DE UN SISTEMA COMPLEJO DE ASCENSORES

MODELING, SIMULATION, AND ANALYSIS OF A COMPLEX LIFT SYSTEM

Realizado por

MARINA SARABIA SEGURA

Tutorizado por

MARÍA DEL MAR GALLARDO

Departamento

LENGUAJES Y CIENCIAS DE LA COMPUTACIÓN

UNIVERSIDAD DE MÁLAGA
MÁLAGA, JUNIO 2019

Fecha defensa: Julio 2019

Fdo.: El Secretario del Tribunal

Resumen

La construcción de software complejo puede verse asistida por los llamados métodos formales que permiten detectar errores de diseño e implementación, especialmente con respecto a las propiedades críticas del sistema. Los métodos formales son técnicas con una base matemática muy fuerte que les hace muy apropiados para el análisis y verificación del software.

En este proyecto, hemos seleccionado un sistema complejo, como es la construcción de un sistema compuesto de varios ascensores que funcionan de manera coordinada. Este problema muestra muchos de los problemas típicos del desarrollo de software crítico. Por un lado, como cada ascensor funciona como un proceso autónomo que tiene que sincronizarse con el resto, la implementación del sistema puede contener los típicos errores de seguridad o viveza propios de los sistemas concurrentes. Por otro lado, la estructura del sistema de ascensores contiene varios componentes que se relacionan entre sí de múltiples formas. Finalmente, en el sistema de ascensores también hay propiedades de tiempo real que deberían satisfacerse para que su comportamiento sea aceptable.

Cada tipo de propiedad requiere utilizar una técnica formal diferente. La mejor técnica formal para detectar errores propios de la interacción incorrecta entre procesos es el “Model Checking”[1,2,3]. En el model checking, los actores principales son los estados del sistema, y las transiciones que hacen que el sistema evolucione. Sin embargo, para analizar propiedades estructurales de los sistemas es mucho más fácil y eficiente usar lenguajes en los que la noción de clase y relación/asociación sea más explícita.

En este proyecto, se ha utilizado el model checker SPIN, las herramientas Alloy, USE (para UML/OCL) y Uppaal para la descripción y análisis de las distintas propiedades deseables del sistema. Como resultado, se ha construido un sistema de ascensores que es correcto con respecto a las propiedades críticas del sistema.

Palabras clave:

Sistema de ascensores distribuido, Verificación de modelos, Model checking, SPIN, UPPAAL, Alloy analyzer, OCL, Lógica Temporal.

Abstract

The construction of complex software can be assisted by the so-called formal methods that allow the detection of design and implementation errors, especially with respect to the critical properties of the system. Formal methods are techniques with strong mathematical basis that makes them very appropriate for the analysis and verification of software.

In this project, we have selected a complex system, such as the construction of a system composed of several elevators that work in a coordinated manner. This problem shows many of the typical problems of complex software development. On the one hand, as each elevator works as an autonomous process that has to be synchronized with the rest, the implementation of the system may contain the typical safety and liveness errors of concurrent systems. On the other hand, the structure of the elevator system contains several components that are related to each other in multiple ways. Finally, in the elevator system there are also real-time properties that should be satisfied for their behavior to be acceptable.

Each type of property requires the use of a different formal technique. The best formal technique to detect errors of the incorrect interaction between processes is "Model Checking". In model checking, the main actors are the states of the system, and the transitions that make the system evolve. However, to analyze structural properties of systems, it is much easier and more efficient to use languages in which the notion of class and relation/association is more explicit.

In this project, the model checkers SPIN and Uppaal and the tools Alloy, USE (for UML / OCL) have been used for the description and analysis of different desirable properties of the proposal system. As a result, an elevator system has been constructed that is correct with respect to its most critical properties.

Keywords:

Distributed elevator system, Verification of models, Model checking, SPIN, UPPAAL, Alloy analyzer, OCL, Temporary Logic.

INDICE GENERAL

1 Introducción

La construcción de un sistema complejo como es el caso de un sistema que consta de un número arbitrario de ascensores en edificios de gran altura no es en absoluto una tarea trivial. El sistema debe garantizar que los usuarios esperen el mínimo tiempo imprescindible antes de subirse al ascensor y, una vez, dentro el ascensor debe llevarlos a su destino, también lo más rápido posible. Este comportamiento deseable está gobernado, usualmente, por una serie de reglas (planificación) que llevan implementados los controladores del sistema. Por ejemplo, es usual que la petición realizada desde dentro de un ascensor tenga prioridad con respecto a la que se realiza desde fuera del mismo. Pero, como las circunstancias pueden ser muy variables, pueden tenerse en cuenta técnicas de planificación diversas que afectan de forma directa al comportamiento del sistema.

Existen muchos trabajos enfocados al análisis de distintos aspectos de estos sistemas, desde características de bajo nivel relativos a las interacciones entre los ascensores, hasta aspectos más funcionales de su comportamiento. En cualquier caso, es indudable que la construcción de un sistema complejo como el descrito debe ir precedido de algunos análisis que garanticen que se satisfacen las propiedades más importantes.

En este trabajo, se ha llevado a cabo el modelado, análisis y verificación del comportamiento del sistema descrito a través de diversas técnicas formales, cada una de las cuales se centra en un tipo particular de propiedades.

En concreto, se ha utilizado la herramienta para la verificación lógica de software concurrente SPIN para analizar propiedades relativas de seguridad y viveza de los ascensores cuando estos se ven como procesos autónomos. Se ha utilizado la herramienta USE para analizar el comportamiento del sistema descrito en UML/OCL. Por otro lado, se ha utilizado el lenguaje de especificación declarativo Alloy para expresar restricciones y comportamientos estructurales más complejos del sistema de software. Finalmente, la herramienta Uppaal ha permitido analizar propiedades de tiempo real.

1.1 Descripción del problema

Desde el antiguo Egipto a la actualidad se encuentran registros de diversos esfuerzos de la humanidad para transportar cargas y personas. A través de los siglos, la tracción animal y la fuerza humana fueron sustituidas por la energía de vapor, una tecnología de la Revolución Industrial que había quedado restringida al transporte de cargas. Más tarde, con el surgimiento de nuevos mecanismos de seguridad, se construyó el primer ascensor de pasajeros. Los primeros ascensores movidos a vapor resultaron ser demasiado lentos. Para alcanzar la octava planta de un edificio un pasajero debía esperar una media de 2 minutos [4].

Actualmente, los ascensores pueden alcanzar una velocidad de medio kilómetro por minuto. Los circuitos eléctricos representaron un gran avance, pero con la llegada de la informática la satisfacción de los pasajeros ha aumentado considerablemente.

Los ascensores y las partes que los componen deben responder a las necesidades de los edificios en los que van a instalarse. Dependiendo de las plantas que contenga el edificio, y de la cantidad de personas que necesiten utilizarlo durante el día, sus características y especificación varían [5].

Los ascensores de hoy en día poseen una memoria que almacena las peticiones y las atiende priorizando las peticiones más cercanas a ellos o priorizando las peticiones internas respecto de las externas.

DESCRIPCIÓN DEL SISTEMA

En el sistema de control de ascensores (SCA) que se propone en el presente proyecto utiliza una memoria que prioriza las peticiones internas. Este SCA está compuesto por los siguientes componentes:

Edificio

Es la construcción de dimensiones variables que es destinada a servir de vivienda o es destinada a servir como espacio para llevar a cabo actividades humanas. Prácticamente todos los edificios con dos o más plantas contienen ascensores para facilitar la movilidad de las personas entre sus plantas.

Plantas

Se consideran plantas o pisos cada una de las alturas en las que está dividida un edificio.

Cabina del Ascensor

Es el encargado de transportar a las personas o a los objetos que se quieren trasladar de una planta a otra y es donde se encuentra un conjunto de botones a través de los que los usuarios realizan las peticiones internas.

Puertas de planta

Puertas situadas en cada planta del edificio y que pueden ser automáticas, semiautomáticas o manuales.

Puertas de ascensor

Las puertas de ascensor son parte de cada ascensor y pueden ser automáticas, semiautomáticas o manuales.

Botones de planta

Los botones de planta sirven al usuario para realizar peticiones externas. En nuestro sistema contamos con uno para indicar que la dirección es de subida y el otro para la dirección de bajada.

Botones de ascensor

La botonera que hay dentro del ascensor contiene un botón para cada planta a través de los cuales se realizan peticiones internas.

Controlador

El controlador atiende las llamadas desde las plantas o la cabina, haciendo que el ascensor suba o baje en función de éstas, y ocupándose también de que el ascensor se detenga en las plantas que el pasajero haya solicitado y de que se abran y cierren las puertas.

El controlador deberá ocuparse asimismo de la parada de los ascensores cuando se alcance la planta destino, siempre en función de las peticiones pendientes.

Las puertas deben abrirse automáticamente al llegar a una planta en la que se realice una parada. Las puertas se cierran automáticamente cuando se pulsa un botón de petición de planta desde el ascensor, o, si no se pulsa ningún botón dentro del ascensor, cuando haya transcurrido un determinado tiempo desde la apertura de las puertas.

El funcionamiento de los ascensores es el siguiente:

1. Cuando se pulsa un botón de llamada, se registra una petición de planta y se realiza la asignación de un ascensor que no esté ocupado.
2. Cuando se pulsa un botón de cabina, dentro del ascensor, se registra la petición y el ascensor responde a ésta, así como a las peticiones de planta asignadas al ascensor. En nuestro caso, siempre se priorizan las peticiones de cabina antes que las de planta.

1.2 Objetivos del proyecto

El objetivo de este trabajo es el modelado, simulación y análisis de un sistema complejo compuesto por varios ascensores que funcionan en edificios y que están gestionados mediante un controlador. Para ello, se utilizarán cuatro técnicas formales diferentes que permiten analizar distintos aspectos del sistema.

En primer lugar, se ha realizado el estudio y la construcción de un meta modelo UML del sistema de ascensores descrito, haciendo uso de una metodología iterativa e incremental y utilizando el lenguaje formal OCL para definir restricciones que proporcionen un modelo fiel del sistema que se pretende implementar. La consistencia y viabilidad del modelo desarrollado se ha analizado utilizando la herramienta USE.

Después de realizar la construcción del meta modelo, se han utilizado tres técnicas formales diferentes para analizarlo desde distintos puntos de vista. En primer lugar, el sistema se ha descrito utilizando el lenguaje de especificación declarativo Alloy. Con este lenguaje es posible describir propiedades estructurales del sistema. Alloy es, por un lado, un lenguaje de modelado estructural declarativo basado en conjuntos, relaciones entre conjuntos, y lógica de primer orden. Por otro lado, Alloy es también una herramienta de visualización y simulación de modelos, que permite comprobar la consistencia de los modelos desarrollados y su corrección con mucha más claridad que otros verificadores.

La siguiente herramienta utilizada ha sido el model checker Spin. En este caso, el objetivo ha sido la descripción de la interacción entre los ascensores, el controlador y el entorno desde el punto de vista de su ejecución concurrente. Para ello, el sistema se ha descrito utilizando el lenguaje Promela, mientras que las propiedades esperadas (de seguridad y viveza) se han especificado en la lógica temporal LTL (linear temporal logic). Con ambas especificaciones, Spin es capaz de analizar automáticamente si todos los comportamientos del sistema satisfacen cada una de las propiedades.

En último lugar, se ha utilizado la herramienta Uppaal para modelar aspectos de tiempo real del sistema utilizando autómatas temporizados, y para verificarlos frente a propiedades escritas en la lógica temporal TCTL (timed computational tree logic). Uppaal es, al igual que Spin, un model checker que utiliza abstracciones del espacio de estados, que contiene variables llamadas relojes con dominio en el conjunto de los números reales, para hacer decidible la verificación.

1.3 Metodología y medios empleados

En este proyecto, se han utilizado cuatro técnicas formales para analizar la corrección del diseño de un sistema complejo formado por varios ascensores. La metodología que se ha seguido es la habitual cuando se utilizan este tipo de métodos, la cual se puede sintetizar en tres pasos fundamentales:

1. Modelar el sistema utilizando un lenguaje de modelado.
2. Especificar las propiedades utilizando un lenguaje de especificación, normalmente basado en alguna variante de la lógica proposicional.
3. Analizar si el modelo satisface las propiedades utilizando una herramienta automática o semi-automática

En nuestro caso, hemos analizado el sistema desde distintos puntos de vista, utilizando diferentes técnicas formales, desde las características estructurales de más alto nivel, hasta las propiedades más específicas relativas al comportamiento concurrente o de tiempo real.

De esta forma, hemos comenzado realizando una especificación UML/OCL que nos ha permitido establecer las componentes principales del sistema y sus relaciones. Se ha analizado esta especificación de alto nivel con la herramienta USE. Sin embargo, es muy difícil asegurar que un modelo sea totalmente correcto utilizando únicamente UML/OCL ya que el lenguaje no es lo suficientemente expresivo como para describir ciertas características del sistema. Podrían utilizarse diagramas de casos de uso para analizar requisitos y comportamientos específicos del sistema, pero hay pocas herramientas que puedan realizar este trabajo de manera automática.

Para mejorar la descripción del modelo hemos utilizado el lenguaje y la herramienta Alloy que permite describir propiedades y comportamientos con la lógica de primer orden y analizarlos de forma automática.

A continuación, hemos realizado una especificación del sistema en el lenguaje de entrada del model checker Spin, que se denomina Promela. En este lenguaje, se pueden hacer explícitas las interacciones entre los procesos y analizar automáticamente propiedades de seguridad y de viveza descritas en la lógica LTL.

Finalmente, se ha realizado una descripción del sistema en autómatas temporizados, para poder analizar propiedades de tiempo real utilizando el model checker Uppaal.

Esta metodología sólo proporciona la corrección parcial de los sistemas, es decir, sólo se puede asegurar que se satisfacen las propiedades críticas que se han analizado ya que es imposible probar que este tipo de sistemas sean correctos al 100%.

En la Figura 1, se muestra la metodología general utilizada por las herramientas de model checking como son Spin y Uppaal. Como puede verse en la figura, la entrada a la herramienta es el modelo del sistema a analizar, y las propiedades esperadas del mismo. La herramienta analiza de manera automática el modelo frente a la propiedad y proporciona un veredicto. Si nos devuelve la respuesta “Sí” podemos inferir que el modelo satisface la propiedad. Si nos devuelve la respuesta “No”, proporciona también el comportamiento anómalo que ha violado la propiedad, que se denomina contraejemplo, y que es muy útil para depurar el modelo.

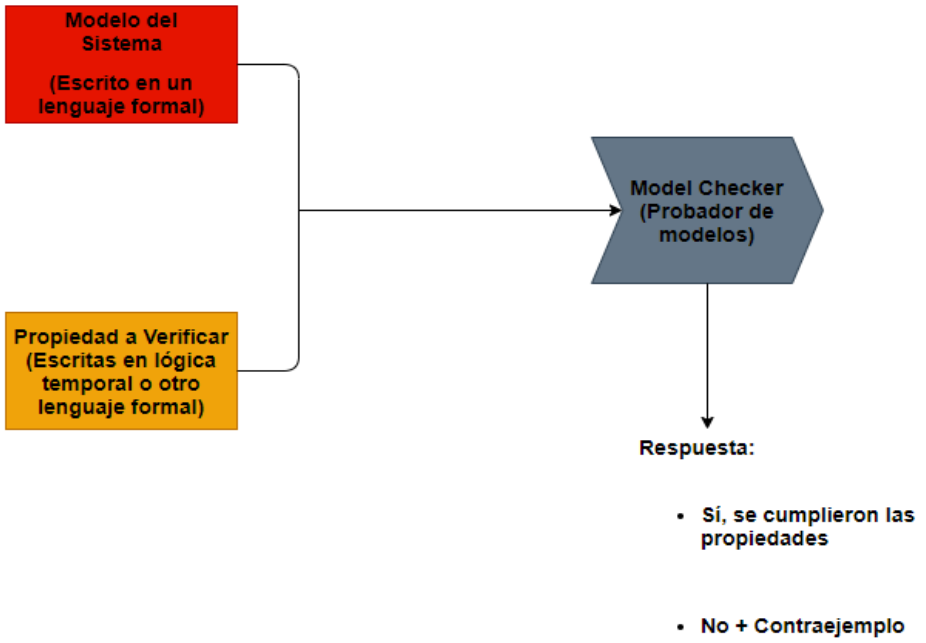

Figura 1: Diagrama proceso de verificación de modelos

2 Modelado del Sistema SCA

El principal objetivo de esta sección es dar construir diferentes elementos y estructuras que componen el ascensor y su funcionamiento, así como las relaciones entre ellos. Se ha realizado una especificación previa del modelo de datos en UML, utilizando casos de uso, diagramas de secuencia, diagramas de colaboración y diagramas de actividad.

2.1 Descripción del Sistema

SISTEMA DE CONTROL PARA UN GRUPO DE ELEVADORES

El sistema implementado está formado por varios ascensores y un único controlador que se encarga de decidir a qué cabina, que esté libre, se le asigna cada una de las llamadas realizadas desde las distintas plantas del edificio en el que se encuentra.

En nuestro sistema, como ya se ha comentado, las llamadas desde el interior de la cabina tendrán prioridad sobre las que se realizan desde fuera del ascensor, es decir, desde una planta del edificio.

El controlador, además, se ocupa de que el ascensor se detenga en las plantas desde las que se ha solicitado un ascensor, y que se abran y se cierren las puertas.

Los dispositivos físicos que el controlador debe manejar son los ascensores, las puertas, los botones, y la iluminación de estos, tal y como se muestra en la Figura 2. El controlador recibirá la información de las peticiones desde los ascensores y las plantas a través de estos botones y deberá ocuparse de parar el ascensor en cada planta del edificio cuando corresponda, en función de las llamadas pendientes. Si el ascensor debe detenerse en un piso, se debe ordenar la parada en el momento en que llegue.

Las puertas deben abrirse automáticamente al llegar a una planta desde la que se ha realizado una llamada. Las puertas del ascensor deben cerrarse cuando se hace una llamada desde el panel de botones del ascensor. Del mismo modo, si no se recibe ninguna petición ambas puertas se cerrarán tras un periodo de tiempo [6].

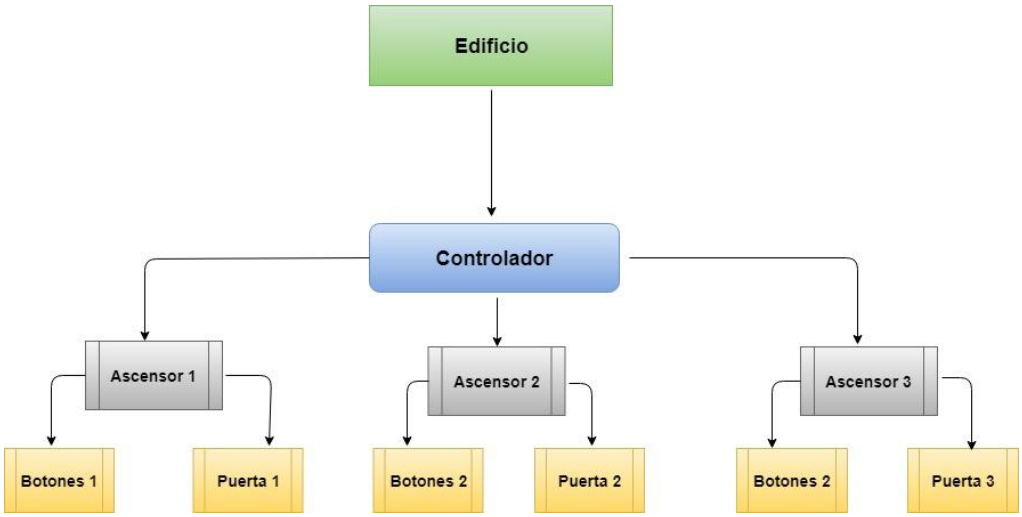

Figura 2: Diagrama controlador del ascensor

2.2 Análisis del modelo UML

Desde el punto de vista del usuario, un sistema complejo, como el que se describe en el presente proyecto, actúa como una entidad cerrada que se comporta de una forma predecible.

En UML, la mejor forma de describir la interacción entre el sistema y su entorno es mediante los denominados *casos de uso*. Un caso de uso describe los comportamientos de un sistema o una parte del mismo mediante la secuencia de acciones se ejecutan para producir un resultado observable para un actor [7].

Un diagrama de casos de uso modela la vista de diseño dinámico de los sistemas. Los diagramas de casos de uso son muy importantes para modelar el comportamiento de un sistema, un subsistema o una clase. El diagrama de caso de uso muestra un conjunto de casos de uso, actores y sus relaciones.

Los actores principales que intervienen en los casos de uso de nuestro sistema son “El pasajero” y el “Controlador del Ascensor”.

Diagramas UML

La Figura 3 muestra los requisitos del sistema mediante el diagrama de caso de uso del SCA:

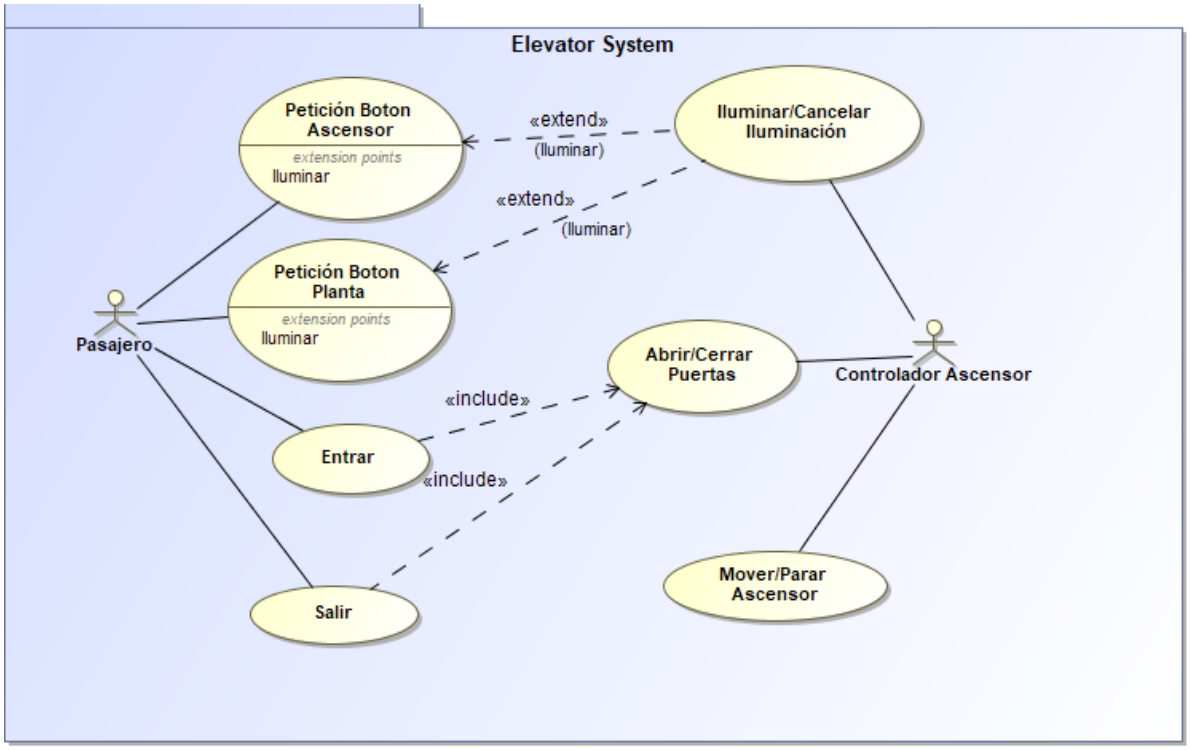

Figura 3: Diagrama de caso de uso del sistema ascensor SCA

El escenario básico que se muestra en el diagrama de caso de uso es el siguiente:

- Un pasajero pide un ascensor y presiona el botón de la planta en la que se encuentra.
- El controlador detecta que el botón de planta se ha presionado e ilumina dicho botón.
- El controlador envía un ascensor libre al piso seleccionado.
- Cuando el ascensor llega, abre sus puertas y el controlador apaga el botón presionado.
- El pasajero entra y presiona el botón del ascensor que indica la planta a la que se quiere dirigir.
- El botón se ilumina.
- El ascensor cierra las puertas.
- El ascensor se mueve a la planta seleccionada.
- El ascensor abre las puertas.
- El pasajero sale del ascensor.
- Tras un periodo de tiempo, las puertas del ascensor y de planta se cierran, a no ser que se produzca una llamada antes de ese periodo.

Los diagramas de secuencia y de colaboración proporcionan una información similar a la de los casos de uso, pero expresada desde un punto de vista diferente.

Un diagrama de secuencia muestra la secuencia explícita de mensajes adecuados para modelar un sistema en tiempo real, mientras que el diagrama de colaboración hace énfasis en las relaciones entre los objetos.

La Figura 4 muestra una secuencia de mensajes válida del sistema SCA. En la figura, un pasajero presiona desde el interior del ascensor un botón. A raíz de esta acción, se genera una petición que llega al controlador que iluminará el botón seleccionado y accionará un ascensor para que comience a moverse. Una vez que el ascensor llega a la planta destino, el controlador lo parará y cancelará la iluminación del botón que se encontraba presionado. Por último, se abrirá la puerta y tras un tiempo determinado, ésta se cerrará.

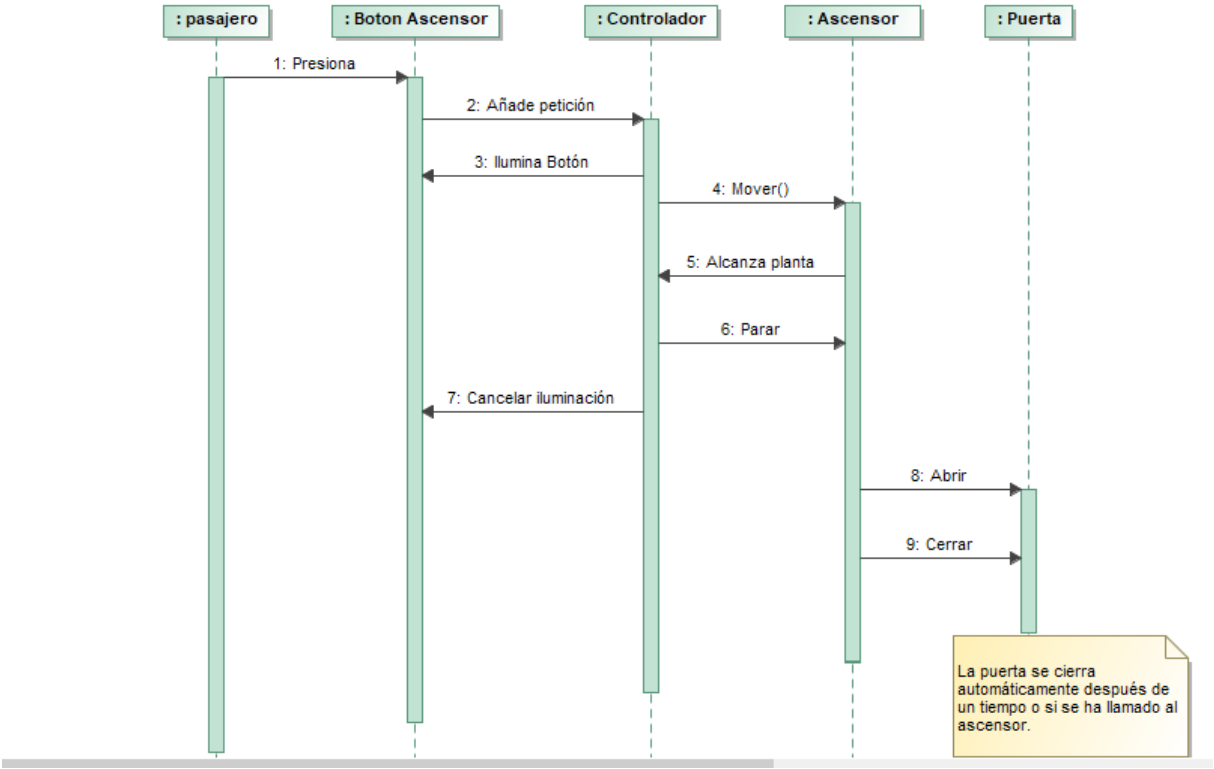

Figura 4: Diagrama de secuencia sistema ascensor

En la Figura 5, se muestra el conjunto de interacciones entre clases o tipos y las relaciones entre objetos para el caso de una petición desde dentro del ascensor. Es el mismo caso de uso que hemos descrito para el diagrama de secuencia.

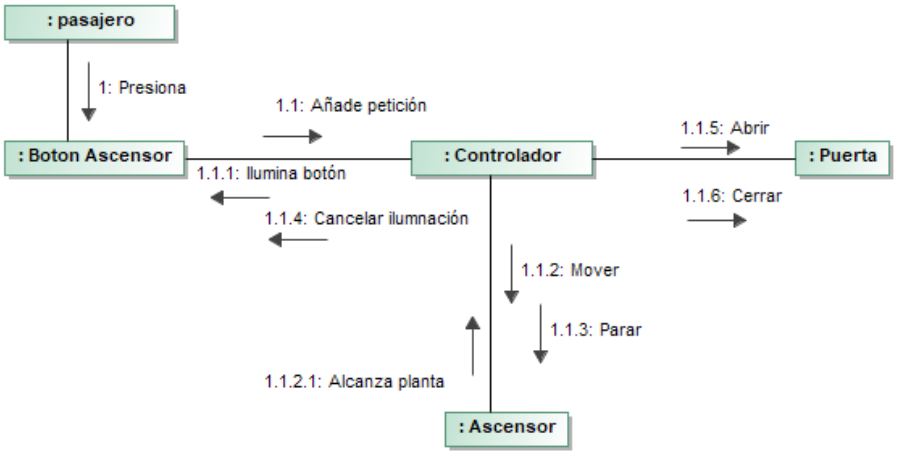

Figura 5: Diagrama de colaboración petición desde el ascensor

La Figura 6 muestra otro diagrama de colaboración que puede definirse para representar el caso en el que un pasajero presiona un botón desde una planta. Como se puede observar en la figura, el proceso es casi idéntico al pulsar un botón de ascensor.

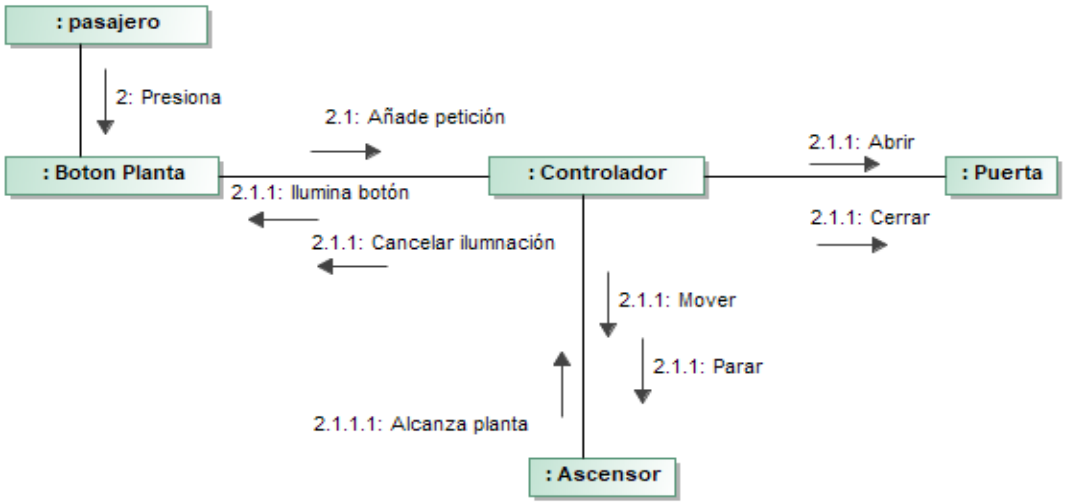

Figura 6: Diagrama de colaboración petición desde planta

Un diagrama de actividad es un diagrama de flujo del proceso multi-propósito que se usa para modelar el comportamiento del sistema. Los diagramas de actividad se pueden usar para modelar un caso de uso, o una clase, o un método complicado. En el diagrama que se muestra en la Figura 7, se ha definido el caso en el que un pasajero presiona un botón de planta. A continuación, el ascensor que se

ha asignado a la petición pasa a estar ocupado, se ilumina el botón y se cierra la puerta, si estuviese abierta. Si no hubiese ningún ascensor libre esto no ocurriría. Seguidamente, el ascensor comienza a moverse a la planta seleccionada. Hasta que no se alcanza la planta destino, cada planta intermedia por la que se pasa es comprobada por el control de planta. Cuando se encuentra en la planta destino se abre la puerta y el botón se apaga.

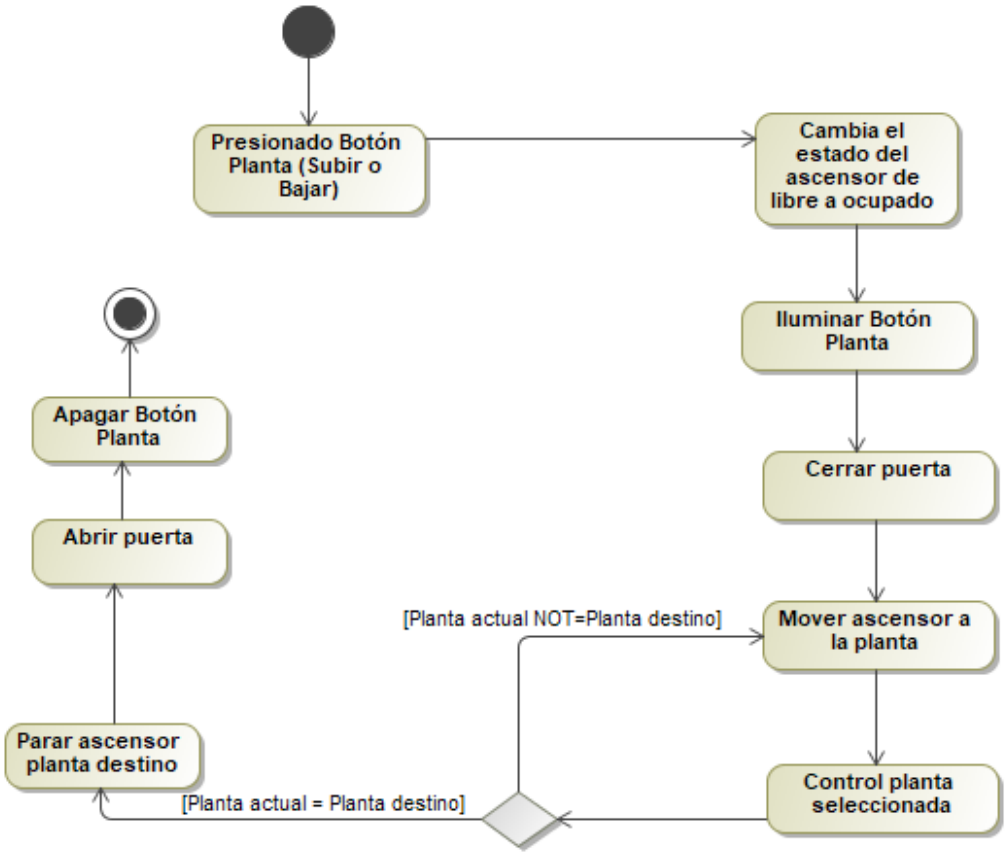

Figura 7: Diagrama de actividad petición desde planta

2.3 Modelado del sistema en UML/OCL

El modelado de sistemas software, por lo general, siempre ha consistido en la realización de diagramas, con información que, a menudo, es insuficiente, incompleta y, en muchos casos, inconsistente. Para una especificación más precisa pueden usarse lenguajes formales, que ofrecen numerosos beneficios tal y como se ha comentado anteriormente. Entre ellas, la principal es que las especificaciones pueden ser analizadas por herramientas automáticas (o semi-automáticas) para asegurar que no son erróneas ni inconsistentes con otros elementos de nuestro modelo.

OCL es un lenguaje de especificación basado en la lógica de primer orden para la descripción formal de restricciones sobre modelos UML. OCL permite especificar invariantes, precondiciones, postcondiciones, inicializaciones, guardas, etc.

OCL es un lenguaje que realiza sólo operaciones de consulta, referenciando valores o relaciones entre los objetos, por lo que no altera los objetos del modelo que se está analizando [8].

En nuestro trabajo se ha modelado el sistema utilizando tanto un enfoque imperativo como uno declarativo. El enfoque imperativo utiliza OCL para modelar las operaciones abstractas como solicitudPeticion(p:Peticion), abrirPuerta(), subir(), parar(), etc.

Por otro lado, las precondiciones y postcondiciones definen los requisitos de entrada/salida necesarios para garantizar que las operaciones funcionan correctamente. El enfoque declarativo especifica las operaciones abstractas como operaciones de consulta booleanas. Estas operaciones OCL representan las restricciones a través de predicados también definidos como booleanos.

La evolución del sistema se modela explícitamente como una secuencia de objetos abstractos en donde cada transición, que consiste en un objeto y su sucesor directo, representa implícitamente la ejecución de una de las operaciones.

La verificación de cada instancia consiste en comprobar si cada transición se corresponde con la ejecución correcta de una operación, es decir, para cada par de objetos sucesivos las pre y post condiciones de la operación ejecutada deben ser ciertas.

Las expresiones invariantes, así como las pre y post condiciones pueden acceder directamente a los valores definidos dentro de los objetos y a las relaciones entre ellos.

Un invariante es una expresión OCL que restringe las instancias válidas de una clase a aquellas sobre las que el valor de la expresión es cierto [9].

La Figura 8 muestra el diagrama de clases de la estructura estática del metamodelo final del SCA generado con la aplicación USE, su estructura interna, y sus relaciones:

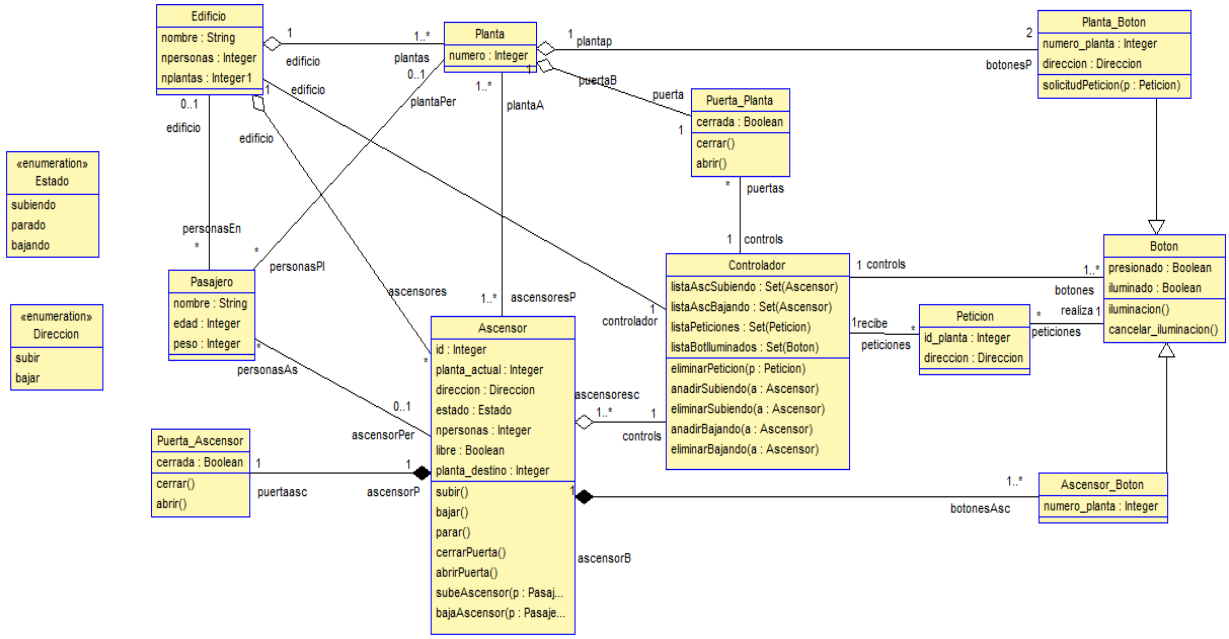

Figura 8: Metamodelo Sistema de Ascensores generado con USE

Como se puede observar en la imagen, el metamodelo descrito está compuesto por las siguientes clases, de las cuales las más significativas son las siguientes:

EDIFICIO

La clase Edificio se encuentra compuesta por los atributos *nombre*, *número de personas* y número *de plantas*.

Contiene una relación de composición con la clase Planta y una relación de asociación con las clases Ascensor, Pasajero y Controlador.

PLANTA

Esta clase contiene como único atributo *el número de planta* que es un identificador.

Contiene una relación de agregación con las clases Puerta_Planta y Planta_Boton y una relación de asociación con las clases Edificio, Pasajero y Ascensor.

ASCENSOR

La clase Ascensor está compuesta por los atributos *id*, *planta actual*, *dirección*, *estado*, *número de personas*, *libre*, y *planta destino*.

También contiene las operaciones *subir*, *bajar*, *parar*, *cerrar puerta*, *abrir puerta*, *sube al ascensor* y *baja del ascensor*. Cuando una de estas operaciones es ejecutada, el ascensor puede cambiar de estado.

Mantiene una relación de composición con las clases Ascensor_Boton, Puerta_Ascensor y de agregación con la clase Controlador. Con las clases Pasajero y Planta mantiene una relación de asociación.

CONTROLADOR

El controlador está compuesto por cuatro conjuntos que representan la lista de ascensores que se encuentran *subiendo* y *bajando*. También contiene la *lista de peticiones* que realizan los usuarios y una *lista de botones* que se encuentren pulsados.

Por otro lado, el controlador también cuenta con una serie de operaciones para *añadir* y *eliminar* de las listas sus correspondientes objetos.

PASAJERO

La clase Pasajero está compuesta por los atributos *nombre*, *edad*, peso.Tiene además una relación de asociación con las clases Edificio, *Planta*, y *Ascensor*.

PUERTA PLANTA Y ASCENSOR

Las clases Puerta tanto del Ascensor como de las Plantas, tienen el mismo comportamiento. En ambos casos, cada Puerta contiene el atributo *cerrada* de tipo Boolean que indica si la puerta se encuentra o abierta. Ambas clases cuentan con las operaciones *abrir* y *cerrar*.

BOTÓN PLANTA Y ASCENSOR

Estas dos clases heredan de la clase Botón que contiene dos atributos tipo Boolean *presionado* e *iluminado*, ambas indican el estado en el que se encuentran. Además, contiene las operaciones *iluminación* y *cancelar iluminación* que modifican el valor de dichos atributos.

Ambas clases cuentan con el atributo *número de planta* que indica la planta a la que pertenece el botón pulsado. La clase Botón Planta también cuenta con un atributo *dirección* y la operación *solicitud petición* que se explica detenidamente en una sección posterior.

PETICION

Esta clase contiene los atributos *id de planta* y *dirección*.

2.4 Modelo e invariantes en OCL

Como ya se ha comentado con anterioridad, un diagrama UML no es suficientemente detallado para proporcionar todos los aspectos relevantes de una especificación. Hay que incluir en el meta modelo algunas reglas adicionales que determinen cuándo un modelo está bien formado.

Para el caso del modelo SCA, se ha empleado el lenguaje de especificación formal OCL, que permite describir expresiones y restricciones en modelos sobre modelos UML.

El lenguaje OCL puede usarse con distintos propósitos:

1. Para especificar características estáticas sobre clases y tipos en un modelo de clases.
2. Para especificar características estáticas de tipo para Estereotipos.
3. Para especificar pre y post condiciones sobre operaciones y métodos.
4. Como lenguaje de navegación.
5. Para especificar restricciones sobre operaciones. En el documento Semántica del UML, se utiliza OCL en la sección reglas bien formuladas, como constantes estáticas sobre las metas clases en la sintaxis abstracta. En varios lugares también se usa para definir operaciones 'adicionales', que se toman en cuenta en la formación de reglas [10].

Antes presentar los invariantes definidos sobre el modelo SCA vamos a explicar brevemente la sintaxis de OCL [11].

SELF

En OCL, la palabra reservada self se utiliza para referirse a la instancia contextual.

Por ejemplo:

```
context Edificio
inv : self.nPersonas > 10
```

Es este caso, self hace referencia a una instancia de la clase Edificio. A través de la declaración *context* al comienzo de la expresión especificamos el contexto de una expresión OCL.

INVARIANTES

Los invariantes son condiciones que el sistema debe cumplir siempre excepto, quizás, durante las transiciones de estado.

La siguiente invariante especifica que en un edificio no hay ascensor si hay tiene menos de 3 plantas. Este invariante es válido para cualquier instancia de la clase Edificio.

```
context Edificio
inv: self.nplantas <3 implies self.ascensores->size()<=0
```

Precondiciones y Potscondiciones

Una precondición o postcondición está formada por expresiones OCL que se asocian a un método u operación. La palabra reservada *self* puede usarse en una de estas expresiones para referirse al objeto receptor de la operación. La palabra reservada *result* se refiere al resultado de la operación, en caso de hay alguno.

Las postcondiciones están formadas por expresiones *previas*, que pueden hacer referencia a dos conjuntos de valores:

- El valor de una propiedad al comenzar el método u operación. Para referirnos a este valor se usa la palabra “@pre”.

- El valor de la propiedad luego de completar la operación o método. Para referirnos a este valor se usa la palabra “@post”.

Tipos y valores básicos

En OCL existen varios tipos básicos predefinidos e independientes de cualquier modelo de objetos, con un conjunto de operaciones sobre ellos.

Por Ejemplo:

1. **Boolean: true, false**
2. **Integer: 1, -3, 2, 54, 78124, ..**
3. **Real: 1.2, 5.16, ...**
4. **String: 'esto es un String...'**
5. **Integer: *, +, -, /, abs()**
6. **Real: *, +, -, /, floor()**
7. **Boolean: and, or, xor, not, implies, if-then-else**
8. **String: toUpper(), concat()**

EXPRESIONES LET

La expresión *let* permite definir un atributo derivado o una operación para ser usada luego en otras expresiones OCL.

COLECCIONES

El tipo Collection está predefinido en OCL. Es un tipo abstracto que dispone de un conjunto de operaciones. Sus subtipos son colecciones concretas: Set, Sequence, y Bag.

1. Un tipo Set es el conjunto matemático, no contienen elementos repetidos.
2. Un tipo Bag es como un conjunto, pero puede contener elementos duplicados.
3. Un tipo Sequence es similar a un Bag, pero los elementos están ordenados.

OPERACIONES DE COLECCIONES

OCL define varias operaciones sobre los tipos Colecciones:

Select

La operación select especifica un subconjunto de una colección y su sintaxis es la siguiente:

```
colección->select(c:Objeto| expresión lógica)
```

El resultado de la operación select es la subcolección de todos los elementos de la colección para los que la expresión lógica se cumple. La variable c es llamada iterador. Cuando se evalúa select, c itera sobre la colección evaluando la expresión lógica para cada valor de c.

En el siguiente ejemplo, usamos select para indicar que en cada ascensor debe haber un pasajero mayor de edad:

```
context Ascensor
inv: self.personasAs->select(p: Pasajero| Edad >= 18) ->notEmpty()
```

Collect

La operación collect se utiliza para construir una colección a partir de otra colección, pero con objetos diferentes a la original. Por ejemplo**:**

```
context Ascensor
inv: self.personasAs->collect(p:Pasajero| p.peso)->sum() <=400)
```

En el ejemplo anterior, a partir de los pasajeros de un ascensor, construimos la colección (bolsa) de sus pesos y calculamos su suma. El invariante nos dice que la suma de los pesos de los pasajeros de cada ascensor no puede superar 400. En general, cuando accedemos a un atributo de una colección, OCL automáticamente asumirá que hay una operación collect sobre cada elemento de la colección con el atributo especificada.

ForAll

La operación forAll se utiliza para especificar una expresión booleana que debe cumplirse para cada uno de los elementos de una colección:

```
colección->forAll ( o: Objeto| expresión lógica)
```

La expresión lógica se evaluará a *true* sii es cierta para todos los elementos de la colección, o, dicho de otro modo, la expresión se valúa a *false* si algún elemento no satisface la expresión.

En el siguiente ejemplo, se dic que todas las personas que están en un ascensor también están en el edificio en el que se encuentra el ascensor.

```
context Edificio
inv:self.ascensores.personasAs->forAll(p:Pasajero|self.personasEn->includes(p))
```

Exists

La operación exists se usa para especificar una expresión booleana que debe ser cierta para, al menos, un objeto en la colección:

```
colección-> exists ( o: Objeto| expresión lógica)
```

El resultado es *true* sii la expresión lógica se cumple para uno o más elementos de la colección, o equivalentemente, la expresión lógica se evalúa a *false* si ninguno de los elementos de la colección la satisface.

Por ejemplo, el siguiente código describe las pre y post condiciones de la operación cerrar() (la puerta de un ascensor). Parte de la precondición afirma que debe haber algún botón del ascensor que haya sido presionado.

```
context Puerta_Ascensor::cerrar()
pre: self.cerrada=false and
self.ascensorP.botonesAsc->exists(b:Ascensor_Boton|b.presionado)
post: self.cerrada=true
```

OCL proporciona muchas otras operaciones como son size, includes, excludes, includesAll, excludesAll, isEmpty, notEmpty y any.

A continuación, presentamos las restricciones OCL sobre el modelo UML del SCA. Cada restricción se define como un invariante en el contexto de una clase.

Restricciones sobre el meta modelo

En esta sección, se presentan las restricciones que necesarias para enriquecer el metamodelo final del sistema de ascensores obtenido en la sección anterior, junto con su notación en leguaje OCL:

Todos los edificios deben tener el atributo **nombre** distinto, ya que no puede haber dos edificios con el mismo nombre. En este caso, usamos la función all Instances que proporciona la colección de todas las instancias de la clase edificio en el modelo.

```
context Edificio
inv: Edificio.allInstances()->forAll(e1,e2 |e1<>e2 implies e1.nombre <> e2.nombre)
```

El atributo **npersonas** contiene el número de personas que hay en el edificio.

```
context Edificio
inv: self.personasEn->size() = self.npersonas
```

Todos los ascensores deben tener el atributo **id** distinto, ya que no puede haber dos ascensores con el mismo id.

```
context Ascensor
inv: Ascensor.allInstances()->forAll(a1,a2 | a1<>a2 implies a1.id <> a2.id)
```

Las personas que están en los ascensores de cada edificio, están también en el edificio.

```
context Edificio
inv: self.ascensores.personasAs->forAll(p:Pasajero| self.personasEn->includes(p))
```

Un edificio no tiene ascensores si tiene menos de 3 plantas.

```
context Edificio
inv: self.nplantas <3 implies self.ascensores->size()<=0
```

Todas las plantas deben tener asociados números de planta diferentes.

```
context Planta
inv: Planta.allInstances()->forAll(p1,p2 |p1<>p2 implies p1.numero <> p2.numero)
```

El número de plantas de un edificio es mayor que el número de ascensores que hay en el edificio.

```
context Edificio
inv: self.ascensores->size() < self.nplantas
```

El peso de las personas que hay en un ascensor no debe exceder 400 kilos.

```
context Ascensor
inv: self.personasAs->collect(p:Pasajero| p.peso)->sum() <=400
```

Si el estado del ascensor es subiendo o bajando la puerta del ascensor debe estar cerrada.

```
context Ascensor
inv: (self.estado=#subiendo or self.estado=#bajando) implies self.puertaasc.cerrada
```

Si un ascensor está parado en una planta debe tener la puerta abierta durante un periodo de tiempo.

```
context Ascensor
inv: self.estado=#parado implies self.puertaasc.cerrada=false
```

Otra forma de especificar la invariante:

```
context Ascensor inv:

self.estado=#parado implies self.plantaA->forAll(p:Planta| self.planta_actual=p.numero implies p.puerta.cerrada=false)
```

Si un ascensor está subiendo, el botón de la planta destino con dirección subir debe estar iluminado.

```
context Ascensor inv:
self.estado=#subiendo implies self.plantaA->forAll(p:Planta|self.planta_destino=p.numero
implies p.botonesP->forAll(b:Planta_Boton| b.direccion=#subir implies b.iluminado))
```

Si un ascensor está bajando el botón de la planta destino con dirección bajar debe estar iluminado.

```
context Ascensor inv:
self.estado=#bajando implies self.plantaA->forAll(p:Planta|self.planta_destino=p.numero
implies p.botonesP->forAll(b:Planta_Boton| b.direccion=#bajar implies b.iluminado))
```

Pre condiciones y Post condiciones

OCL es un lenguaje de restricciones y de consultas. Como lenguaje de restricciones es usado para hacer más precisa la información de los modelos mediante la especificación de invariantes, pre- y post- condiciones de operaciones como ya hemos dicho anteriormente. Sobre nuestro SCA se han definido las siguientes operaciones:

El método subeAscensor (p: Pasajero) que hace que la persona p suba al ascensor receptor del mensaje. La precondición es que p está en el edificio en el que está en el ascensor, pero no debe estar en el ascensor. La post condición es que la persona está en el ascensor.

```
context Ascensor::subeAscensor(p:Pasajero)
pre : self.edificio.personasEn->includes(p)
pre : self.personasAs->excludes(p)
post: self.personasAs->includes(p)
```

El método bajaAscensor (p: Pasajero) que hace que la persona p baje del ascensor receptor del mensaje. La pre condición es que p está en el ascensor. La pos condición es que la persona p no está en el ascensor.

```
context Ascensor::bajaAscensor(p:Pasajero)
pre:self.personasAs->includes(p)
post:self.personasAs->excludes(p)
```

El método cerrar () y abrir () de la clase puerta del ascensor, hace que las puertas se abran o se cierren.

En el caso del método cerrar () la precondición es que la puerta esté abierta y que algún botón del panel ascensor haya sido presionado, la postcondición es que la puerta está cerrada.

```
context Puerta_Ascensor::cerrar()

pre: self.cerrada=false and self.ascensorP.botonesAsc->exists(b:Ascensor_Boton|b.presionado)
post: self.cerrada=true
```

En el caso del método abrir () la precondición es que la puerta está cerrada, y el ascensor haya llegado a la planta destino, la post condición es que la puerta está abierta.

```
context Puerta_Ascensor::abrir()

pre: self.cerrada=true and self.ascensorP.plantaA->forAll(p:Planta|(p.numero=self.ascensorP.planta_destino
and p.numero=self.ascensorP.planta_actual))
post: self.cerrada=false
```

Los métodos cerrar () y abrir () de la clase puerta de la planta, hacen que las puertas se abran o se cierren.

En el caso del método cerrar (), la precondición es que la puerta esté abierta y que algún botón del panel ascensor haya sido presionado, la post condición es que la puerta está cerrada.

```
context Puerta_Planta::cerrar()

pre: self.cerrada=false and self.plantap.ascensoresP.botonesAsc->exists(b:Ascensor_Boton|b.presionado)
post: self.cerrada=true
```

En el caso del método abrir (), la precondición es que la puerta esté cerrada, y el ascensor haya llegado a la planta destino, la postcondición es que la puerta está abierta.

```
context Puerta_Planta::abrir()
pre: self.cerrada=true and self.controls.ascensoresc->forAll(a:Ascensor|(a.planta_destino=self.plantap.numero
and a.planta_actual=self.plantap.numero))
post: self.cerrada=false
```

El método iluminacion() ilumina un botón cuando es presionado y cancelar_iluminacion() apaga dicho botón, cuando se llega a la planta destino.

```
context Boton::iluminacion()
pre: self.presionado=true and self.iluminado=false  and self.controls.listaBotIluminados->excludes(self)
post: self.iluminado=true and self.controls.listaBotIluminados->includes(self)
```

```
context Boton::cancelar_iluminacion()
pre: self.iluminado=true and self.controls.listaBotIluminados->includes(self)
post:  self.presionado=false and self.iluminado=false and self.controls.listaBotIluminados->excludes(self)
```

El método anadirSubiendo(a:Ascensor) añade a la lista “listaAscSubiendo” en el controlador los ascensores que están subiendo. La precondición obliga a que no esté incluido en dicha lista, y que algún botón del ascensor haya sido presionado con dirección subir. La postcondicion es que el ascensor esté incluido en dicha lista y que el ascensor está ocupado.

```
context Controlador::anadirSubiendo(a:Ascensor)
pre: self.listaAscSubiendo->excludes(a) and a.direccion=#subir
and a.botonesAsc->exists(b:Ascensor_Boton|b.presionado)
and a.libre
post: self.listaAscSubiendo->includes(a) and a.libre=false
```

El método eliminarSubiendo(a:Ascensor) elimina de la lista “listaAscSubiendo” en el controlador los ascensores que estaban subiendo. La precondición indica que a debe estar incluido en la lista, no está libre, y que su planta actual es igual a su planta destino. En la post condición el ascensor no debe estar incluido en la lista, y el ascensor debe estar libre.

```
context Controlador::eliminarSubiendo(a:Ascensor)
pre: self.listaAscSubiendo->includes(a) and a.libre=false
and a.planta_actual=a.planta_destino
post:self.listaAscSubiendo->excludes(a) and a.libre
```

El método anadirBajando(a:Ascensor) añade a la lista “listaAscBajando” en el controlador los ascensores que están bajando. La precondición establece que el ascensor a no debe estar incluido en dicha lista, y que algún botón del ascensor debe haber sido presionado con dirección bajar. La postcondicion es que el ascensor a está incluido en la lista y el ascensor está ocupado.

```
context Controlador::anadirBajando(a:Ascensor)
pre: self.listaAscSubiendo->excludes(a) and a.direccion=#bajar
and a.botonesAsc->exists(b:Ascensor_Boton|b.presionado) and a.libre
post:self.listaAscSubiendo->includes(a) and a.libre
```

El método eliminarBajando(a:Ascensor) elimina de la lista “listaAscBajando” en el controlador los ascensores que estaban bajando. La precondición afirma que el ascensor a deber estar en la lista, no deber estar libre, y su planta actual es igual a su planta destino. La postcondición indica que el ascensor a ya no debe estar incluido en la lista, y debe estar libre.

```
context Controlador::eliminarBajando(a:Ascensor)
pre: self.listaAscBajando->includes(a) and a.libre=false
and a.planta_actual=a.planta_destino
post:self.listaAscBajando->excludes(a) and a.libre
```

2.5 Verificación con USE

En esta sección mostramos cómo se puede validar las especificaciones OCL [12] descrita con la herramienta USE.

La ventana que aparece después de iniciar USE se puede ver en la siguiente captura de pantalla (Figura 9).

A la izquierda, hay una vista de árbol que muestra los contenidos (clases, asociaciones, invariantes y precondiciones y postcondiciones) del modelo.

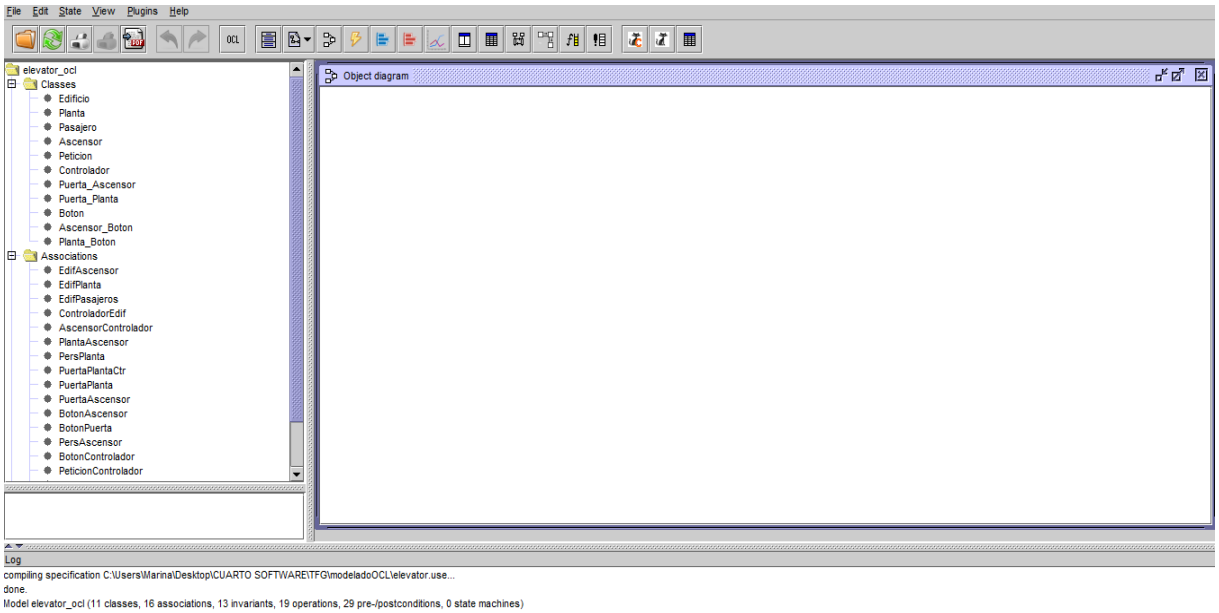

Figura 9: Panel contenido del modelo en la herramienta Use

La siguiente imagen (Figura 10) muestra el árbol expandido con todos los elementos del modelo. Si se selecciona uno de los invariantes, su definición se muestra en el panel debajo del árbol.

El área grande a la derecha es un espacio de trabajo donde se pueden abrir vistas que muestran diferentes aspectos del modelo. Las vistas se pueden crear en cualquier momento seleccionando una entrada del menú de vista o directamente mediante un botón de la barra de herramientas. No hay límite en las vistas activas. La siguiente captura de pantalla muestra la ventana principal después de la creación de una vista de nuestro modelo.

Después de crear nuestro diagrama de objetos si pulsamos sobre el rayo, se evaluarán los invariantes de nuestro modelo y se nos mostrará, en una ventana, si son o no satisfechos.

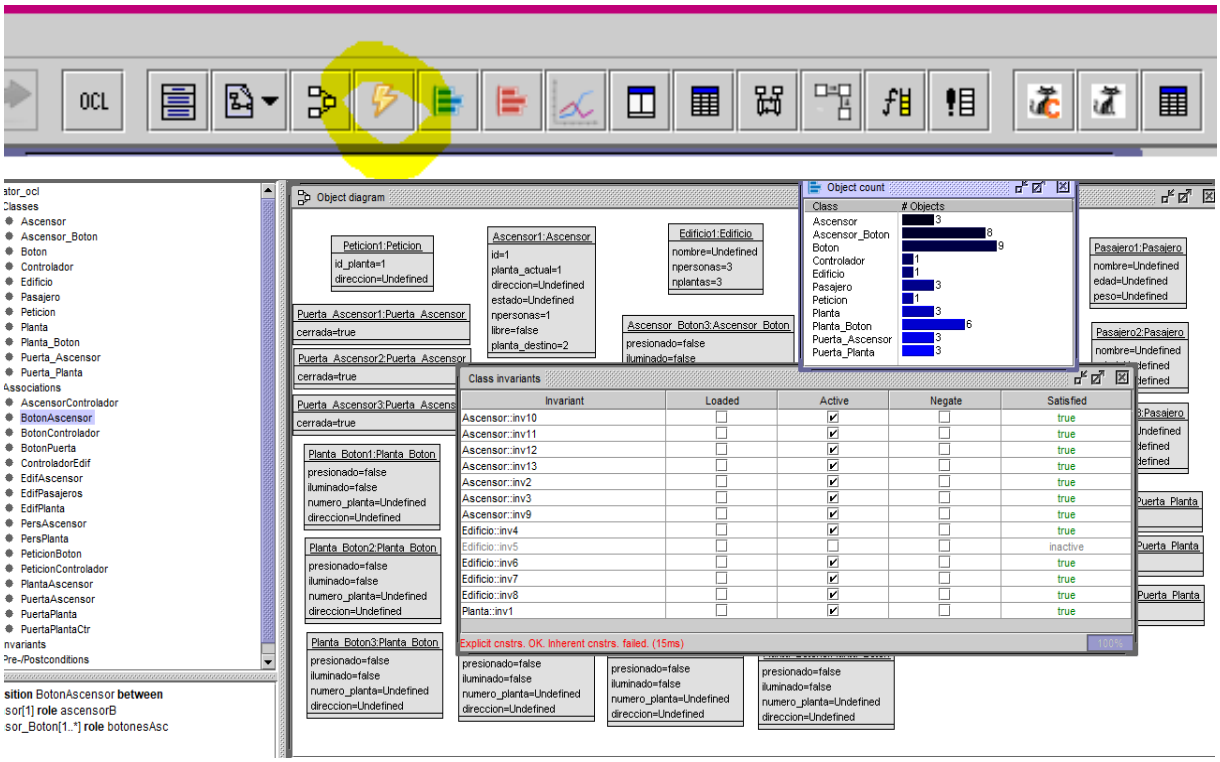

Figura 10: Ejemplo validación invariantes ascensor

El analizador de OCL e intérprete de USE permite la evaluación de expresiones OCL arbitrarias. El elemento señalado en la figura 10 abre un cuadro de diálogo donde se pueden ingresar y evaluar las expresiones.

El SCA es, aunque no lo parezca, a priori un sistema complejo por lo que resulta muy tedioso crear el diagrama de objetos sin utilizar alguna herramienta complementaria, que genere automáticamente los diagramas.

Gracias a OCL se ha logrado una especificación del SCA mucho más precisa usando expresiones sencillas, las cuáles nos han facilitado validar aspectos (Figura 11) y funcionalidades básicas del sistema que con el UML básico que ya conocemos no hubiese sido posible.

Los lenguajes de especificación son una gran parte de la Ingeniería de Software como ya hemos visto con OCL. Aprender cómo escribir un modelo riguroso es una parte fundamental del diseño de un buen código.

En la siguiente sección se explora el lenguaje de especificación de Alloy, su uso a través de la herramienta Alloy Analyzer [13] y su potencial a la hora de analizar y verificar modelos de datos. Este analizador puede utilizarse para analizar de manera automática propiedades sobre los modelos.

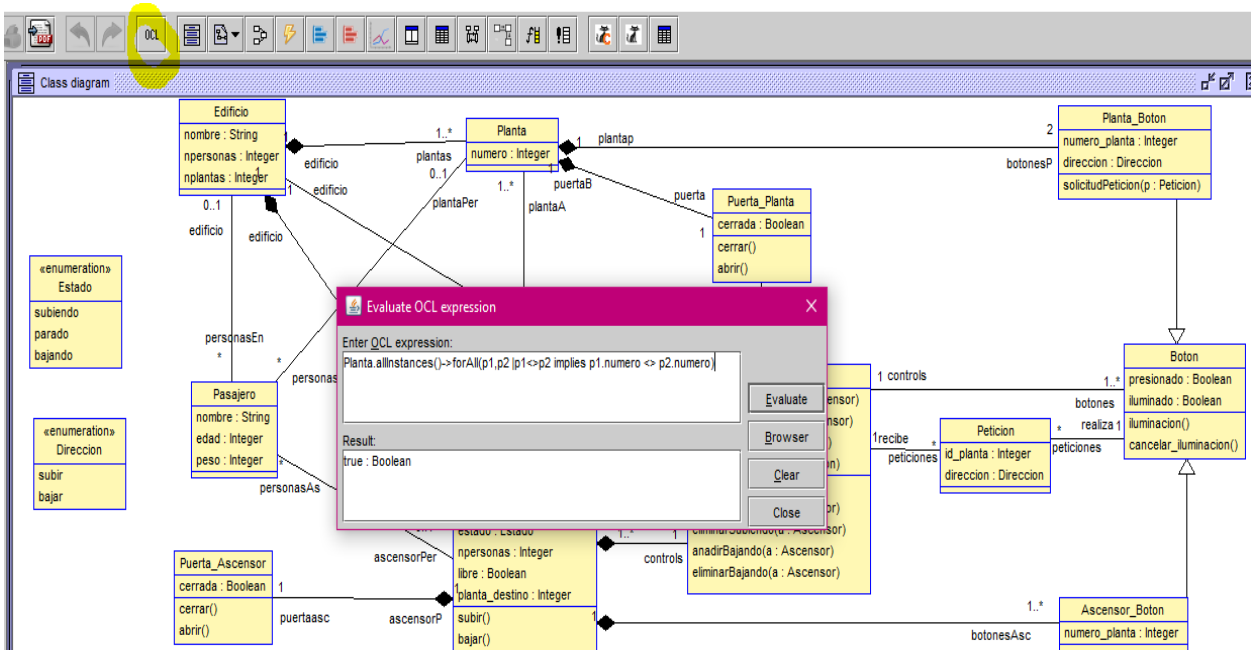

Figura 11: Ejemplo validación invariantes ascensor

3 Especificación y Análisis con Alloy

En esta sección del trabajo, se explicará la sintaxis del lenguaje de especificación de Alloy y su aplicación a nuestro SCA a través del Analizador de Alloy.

3.1 Alloy Analyzer

Alloy es lenguaje de código abierto para el modelado de software, así como una herramienta que permite la generación automática de instancias acordes con el modelo diseñado. Su lenguaje es de modelado estructural basado en lógica de primer orden relacional. Es un lenguaje textual que consiste en signaturas las cuales permiten la definición de relaciones, funciones, predicados y asertos.

Los predicados especificados y los asertos pueden utilizarse para la generación de modelos específicos y para encontrar errores de modelado. La herramienta utiliza KodKod Torlak y Jackson [14], un generador de modelos, para convertir la especificación en modelos utilizando unos límites de tamaño dados por el usuario, en una secuencia de cláusulas booleanas que son la entrada de un SAT solver [15]. Esta herramienta puede de forma iterativa proporcionar instancias válidas de la especificación, o bien un contraejemplo si se está comprobando un aserto.

Alloy ha demostrado su utilidad en diversas aplicaciones en las que se han encontrado fallos de diseño [16].

A diferencia de los model checker, que se mostrarán más adelante, Alloy no permite la demostración de la corrección de un modelo, ya que, el análisis está limitado en el tamaño de las instancias que genera, por lo que la no existencia de instancias erróneas no implica que el modelo sea correcto. En cualquier caso, los desarrolladores que utilizan Alloy suelen asumir la idea de que los errores de modelado se muestran en instancias “pequeñas” de los modelos, de manera que, en general, no hace falta crear instancias de gran tamaño.

En cualquier caso, las propiedades pueden ser probadas inicialmente con Alloy y si no se encuentran errores, se puede analizar más a fondo el sistema con otras herramientas.

Los modelos de Alloy utilizan en la noción de conjuntos y relaciones entre conjuntos. Están compuestos por varios tipos de declaraciones [17]:

Las signaturas son la base del lenguaje Alloy. Definen el vocabulario de un modelo creando nuevos conjuntos. Por ejemplo, la declaración ***sig Object {}*** denota un conjunto de objetos. Además, en la misma declaración pueden definirse relaciones como se muestra en el siguiente código:

```
sig Ascensor{
edificio: one Edificio,
planta_actual: Planta one -> Tiempo,
planta_destino: one Planta,
direccionAsc: Direccion lone -> Tiempo,
botones: set BotonAscensor,
puerta: one Puerta,
controlado_por: one Controlador
}{
```

La signatura Ascensor definida arriba tiene definidas varias relaciones. Las relaciones se establecen a través de campos y pueden ser binarias, ternarias, cuaternarias, etc.

Por ejemplo:

```
planta_actual: Planta one -> Tiempo
```

Declara la relación ternaria, que asocia ascensores, plantas y unidades de tiempo, así contiene valores del tipo {(a0, p0, t0), (a0, p1, t1)} donde a0 es un ascensor, p0 y p1 son plantas de un edificio y t0 y t1 son instantes de tiempo.

Los conjuntos formados por diferentes signaturas son disjuntos a no ser que se declare una signatura incluida dentro de otra. Por ejemplo, los conjuntos Subir y Bajar son subconjuntos singleton de Dirección:

```
abstract sig Direccion {}
one  sig Subir,Bajar extends Direccion {}
```

Para definir restricciones sobre el tamaño de los conjuntos, ya sean signaturas o relaciones, se usa la multiplicidad que puede definirse mediante las palabras reservadas lone, one, some, set.

Alloy también incluye una colección de cuantificadores, para la construcción de fórmulas de primer orden:

all -> all x: X| F, F se verifica para todos los x en X
some -> some x:X | F, F se verifica para algún x en X
no -> no x:X| F, F no se verifica para ningún en x en X
lone -> lone x:X | F, F se verifica para uno o ningún x en X
one -> one x:X | F, F es cierto para exactamente un x en X

Alloy cuenta también con Operadores de conjuntos:

+ => unión
& => intersección
- => diferencia
in => subconjunto
= => igualdad

y operaciones relacionales:

-> => producto
~ => traspuesta
. => join
[ ] => box join
^ => clausura transitiva
* => clausura reflexivo-transitiva **<:** => restricción de dominio
:> => restricción de rango
++ => override

Los ***hechos (facts)*** son restricciones que deben satisfacer todas las instancias del sistema. Por ejemplo, la siguiente fórmula afirma que, en cualquier instancia, los ascensores no comparten botones.

```
//Los Ascensores no comparten botones
all disj a1,a2:Ascensor| no a1.botones&a2.botones
```

Los ***predicados (pred)*** son restricciones parametrizadas y se pueden usar para representar operaciones. Por ejemplo, el predicado plantasEdif define las restricciones que deben cumplir las plantas de un edificio **e**. Una planta **p** de **e** debe ser la primera planta, la última o una intermedia. Las tres restricciones adicionales afirman que todas las plantas son alcanzables a través de las relaciones **panterior** y **psiguiente** utilizando la clausura transitiva (^).

```
pred plantasEdif(p:Planta,e:Edificio){
p in e.primeraPlanta + e.ultimaPlanta+e.intermedias
p = e.primeraPlanta implies p in e.ultimaPlanta.(^panterior)
p in e.intermedias implies  p in e.primeraPlanta.(^psiguiente)
p= e.ultimaPlanta implies  p in e.primeraPlanta.(^psiguiente)
}
```

Del mismo modo que se definen predicados, también es posible definir **funciones (fun),** que son expresiones que devuelven resultados. Los asertos **(assert)** son restricciones que deben derivarse de la especificación del modelo.

A continuación, en la Figura 12, se muestra un pequeño ejemplo que ilustra el uso de la herramienta Alloy.

```
open util/ordering[Tiempo]
sig Tiempo{}
sig Planta{}
sig Ascensor{
planta_actual: Planta one -> Tiempo
}

pred traza(){
  all t:Tiempo-last|
    let t' = next[t] |some a:Ascensor|a.planta_actual.t != a.planta_actual.t'

}

run traza for 4 Ascensor, 4 Planta,5 Tiempo
```

Figura 12: Ejemplo de un modelo en el lenguaje Alloy.

La figura muestra un modelo de ascensor con tres tipos de objetos: Ascensor, Planta y Tiempo. Los objetos de tipo Ascensor contienen una relación ternaria desde Ascensor a exactamente una Planta, que es la posición del ascensor en un instante de tiempo. El predicado traza indica que hay algún par de instantes t, t' y un ascensor

a, de modo que la planta en la que está en ascensor a en el instante t es diferente de la planta en la que se encuentra en el instante t'.

Después de compilar y ejecutar este modelo en la herramienta Alloy Analyzer encuentra una solución consistente en una lista de relaciones, sus tipos y los átomos que contienen Planta, Ascensor y Tiempo, que son las signaturas del modelo. En concreto, la salida textual de la instancia se muestra en la siguiente Figura 13:

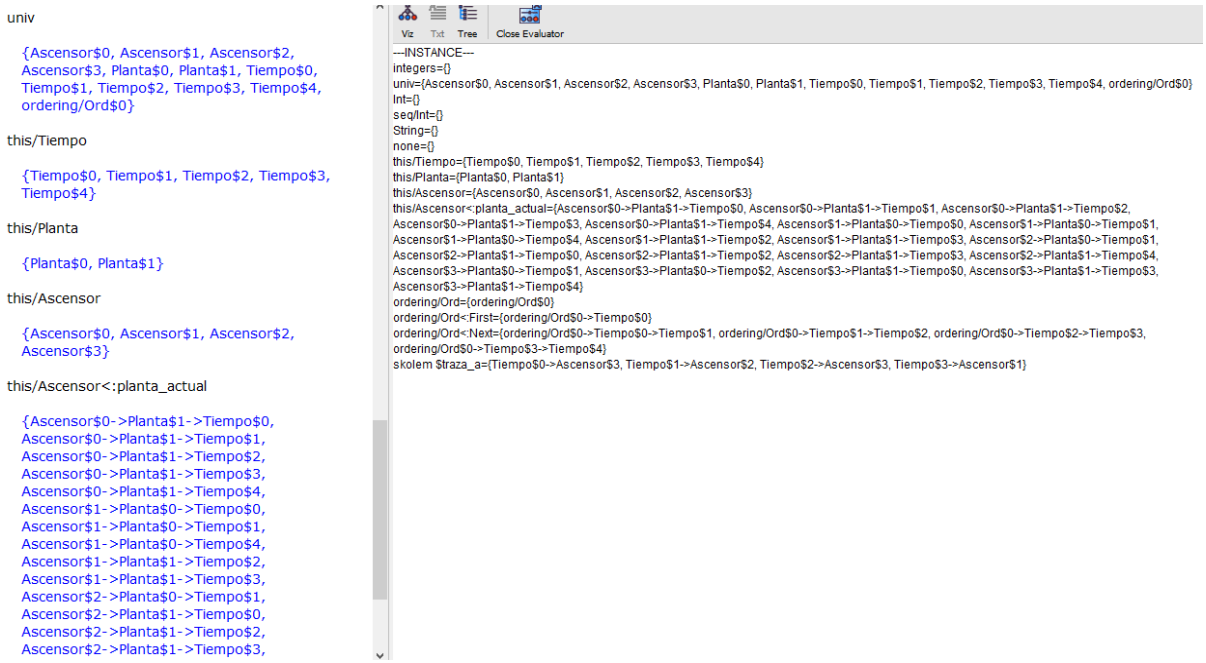

Figura 13: Solución que muestra relaciones, tipos y tuplas proporcionada por Alloy.

Una vista de la instancia se puede visualizar de forma gráfica tal y como se muestra en la siguiente figura 14.

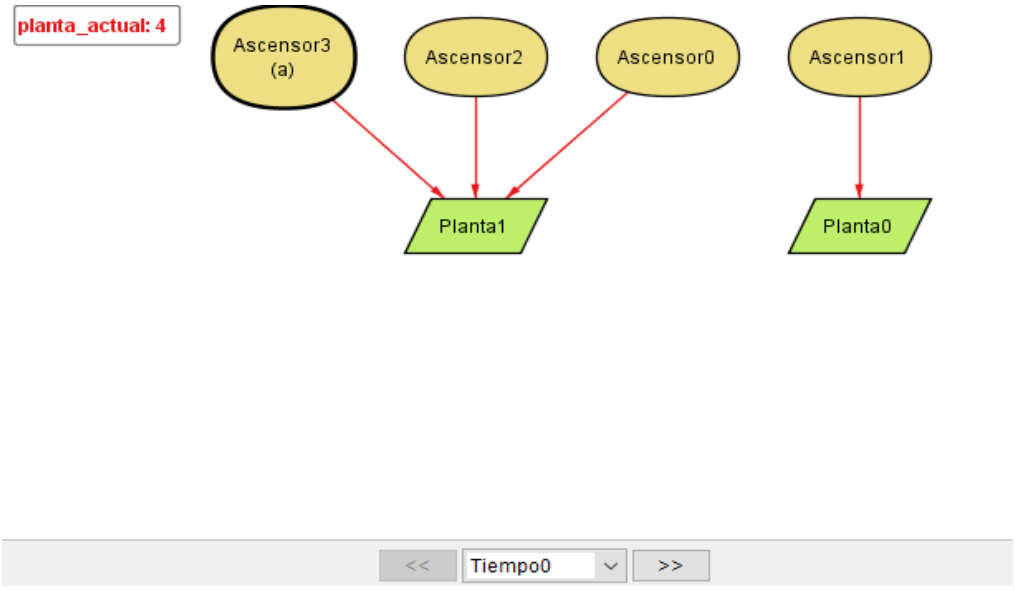

Figura 14: Solución gráfica proporcionada por Alloy.

3.2 Especificación: Sistema de ascensores

El modelo del sistema describe cómo un ascensor debe atender un conjunto de peticiones de acuerdo a un conjunto de restricciones y algunas políticas de servicio. En el modelo del ascensor, las componentes de interés son Edificio, Planta, Ascensor, Controlador, Puerta, Botón Ascensor, Botón Planta, Dirección y Estado de la puerta. Hay dos conjuntos de botones, los botones de llamada, que pertenecen a las plantas y tienen una dirección, y los botones de planta, que pertenecen a los ascensores y tienen una o más plantas de destino. Hay dos conjuntos singleton para las direcciones: Subir y Bajar. Los ascensores siempre deben estar exactamente en una planta.

Los ascensores también tienen una dirección de movimiento para lo que se usarán los conjuntos Subir y Bajar ya mencionados. La Tabla I describe con más detalles los distintos actores del sistema (signaturas):

Lista de conjuntos relevantes

Tiempo	***Instantes tiempo***
Edificio	Edificios del sistema
Planta	Plantas de los edificios
Ascensor	Conjunto de todos los ascensores
Controlador	Conjunto de todos los controladores
Botón Ascensor	Botones en los ascensores
Boton Planta	Botones en las plantas
Puerta	Conjunto de todas las puertas
Direccion	Conjunto de todas las direcciones
Subir	Dirección Subir
Bajar	Dirección Bajar
Estado puerta	Conjunto de todos los estados
Abierta	Estado puerta abierta
Cerrada	Estado puerta cerrada

Tabla 1: Lista de signaturas y conjuntos que componen el modelo

En la Tabla II se muestra la Lista de relaciones relevantes (entre paréntesis aparece la signatura en la que se define cada relación).

planta_actual (Ascensor)	Planta one -> Tiempo	Planta en la que se encuentra el ascensor en cada instante de tiempo
direccionAsc (Ascensor)	Direccion lone -> Tiempo	Dirección que tiene el ascensor en cada instante de tiempo
botones (Ascensor)	set BotonAscensor	Botones que contiene el ascensor
controlado_por (Ascensor)	one Controlador	Controlador del ascensor
Puerta (Ascensor)	one Puerta	Puerta del ascensor
dirección (Boton Planta)	Subir + Bajar	Dirección del botón de planta
botones_planta (Planta)	set BotonPlanta,	Botones que posee la planta
psiguiente (Planta)	lone Planta	Planta siguiente
panterior (Planta)	lone Planta	Planta anterior
estado (Puerta)	Estado_Puerta one->Tiempo	Estado de la puerta del ascensor
subiendo,bajando (Controlador)	Ascensor set ->Tiempo	Dirección del ascensor en algún instante de tiempo
iluminados_ascensor (Controlador)	BotonAscensor set -> Tiempo	botones de ascensores iluminados
iluminados_planta (Controlador)	BotonPlanta set -> Tiempo	Botones de plantas iluminados
peticiones_ascensor (Controlador)	BotonAscensor set -> Tiempo	Peticiones desde ascensor
Peticiones_planta (Controlador)	BotonPlanta set-> Tiempo	Peticiones desde planta

Tabla 2: Lista de relaciones que componen el modelo

ABSTRACCIONES BÁSICAS

Las plantas están ordenadas siguiendo el orden natural: primera planta, intermedias y última planta, y se relacionan entre sí a través de las relaciones psiguiente y panterior.

Hay botones tanto en el interior de los ascensores como en las plantas. Cada botón está asociado a una única planta, su iluminación puede cambiar en distintos instantes de tiempo.

Cada ascensor está, en cada instante de tiempo, en una sola planta, y tiene una única dirección, subir o bajar.

Hay un controlador por cada edificio que gestiona las peticiones desde las plantas, las realizadas desde el interior de los ascensores, así como la iluminación de los botones que han sido pulsados. También gestiona el estado de los ascensores, es decir, si se encuentran subiendo, bajando o parados.

En nuestro modelo, la política de servicio que se sigue es que ninguna petición desde dentro del ascensor puede rechazarse. Además, si un ascensor rechaza una petición realizada desde una planta, ésta será atendida por otro ascensor en un instante de tiempo posterior.

Restricciones de entorno estáticas

La Figura 15 muestra que cada edificio está compuesto por un conjunto de plantas, primera, última e intermedias, un conjunto de ascensores y un controlador. Las plantas primera y última deben ser distintas y además no pueden pertenecer al conjunto de las plantas intermedias. La dirección tendrá una única instancia subir y otra bajar, así como el estado de las puertas, cerrada o abierta.

```
open util/ordering[Tiempo]

sig Tiempo{}

sig Edificio{
primeraPlanta: Planta ,
ultimaPlanta : Planta ,
intermedias : set Planta,
ascensores: set Ascensor,
controlador: one Controlador,
}{
primeraPlanta!=ultimaPlanta
primeraPlanta+ultimaPlanta not in intermedias
}

abstract sig Direccion {}
one  sig Subir,Bajar extends Direccion {}

abstract sig Estado_Puerta{}
one sig Abierta extends Estado_Puerta{}
one sig Cerrada extends Estado_Puerta{}
```

Figura 15: Signaturas Tiempo, Edificio, Dirección, Estado Puerta en Alloy.

La Figura 16 contiene a los botones del ascensor que están asociados a una única planta. También incluye los botones de planta, salvo que estos, además, contienen la relación dirección, para indicar que una planta tiene un botón para subir y otro para bajar.

```
sig BotonAscensor{
planta: one Planta
}

sig BotonPlanta {
direccion: Subir + Bajar
}
```

Figura 16: Signaturas Boton Ascensor y Boton Planta en Alloy.

La Figura 17 contiene la signatura Ascensor. Cada ascensor está en un único edificio y está controlado por un único controlador. Además, está compuesto por un conjunto de botones y una única puerta. Un ascensor podría no tener una dirección, si el ascensor está. Por otro lado, en cada instante de tiempo, un ascensor estará en una única planta, y tendrá una única planta de destino asociada.

```
sig Ascensor{
edificio: one Edificio,
planta_actual: Planta one -> Tiempo,
planta_destino: one Planta,
direccionAsc: Direccion lone -> Tiempo,
botones: set BotonAscensor,
puerta: one Puerta,
controlado_por: one Controlador
}{
//No hay dos botones que vayan a la misma planta
all disj  b1, b2: botones |  b1.planta!=b2.planta
//Tiene botones para todas las Plantas
botones.planta=Planta
controlado_por=edificio.controlador

}
```

Figura 17: Signatura ascensor en Alloy.

En la Figura 18, una planta tiene exactamente dos botones, uno con dirección subir y otra bajar. Además, contiene las relaciones psiguiente y panterior, que definen el orden en el que se encuentran las plantas en el edificio. Una puerta tiene un estado en cada instante de tiempo, que puede ser que la puerta está abierta o cerrada, dependiendo si el ascensor está en movimiento o parado.

```
sig Planta {
botones_planta: set BotonPlanta,
psiguiente: lone Planta,
panterior: lone Planta
}{
#botones_planta=2
all disj b1,b2:botones_planta| b1.direccion!=b2.direccion
}

sig Puerta{
estado: Estado_Puerta one -> Tiempo
}
```

Figura 18: Signaturas Planta y Puerta en Alloy.

En la Figura 19 se muestra la signatura Controlador. Un controlador contiene dos conjuntos de ascensores que, en cada instante de tiempo, se encontrarán subiendo o bajando dependiendo de la dirección de su movimiento. Por otro lado, también contiene el conjunto de botones que han sido iluminados, así como las peticiones a planta.

```
sig Controlador{
subiendo,bajando: Ascensor set  ->Tiempo,
iluminados_ascensor: BotonAscensor set -> Tiempo,
iluminados_planta: BotonPlanta set -> Tiempo,
peticiones_ascensor:   BotonAscensor set  -> Tiempo,
peticiones_planta:   BotonPlanta set-> Tiempo
}
```

Figura 19: Signatura Controlador en Alloy.

Las siguientes restricciones que se presentan a continuación (Figuras 20 y 21) nos permiten formalizar con mayor detalle el comportamiento de los objetos y sus relaciones. Llamamos a estas restricciones estáticas porque no dependen de la signatura tiempo.

```
fact{
//Los Ascensores no comparten botones
all disj a1,a2:Ascensor| no a1.botones&a2.botones
all |b:BotonAscensor, a:Ascensor  | b in a.botones

//todas los botones de Planta pertenecen exactamente a una Planta
 all b:BotonPlanta | one botones_planta.b

//Las Plantas no comparten botones
all disj p1,p2:Planta |no p1.botones_planta&p2.botones_planta

//Cada planta no puede ser siguiente ni anterior de ella misma
all p:Planta | p.psiguiente!= p && p.panterior!=p

//Cada planta siguiente es distinta de la planta anterior de una planta
all p:Planta|p.psiguiente!= p.panterior

//El par (comp1,comp2) esta en siguiente si, y solo si, (comp2,comp1) esta en anterior
psiguiente=~panterior
```

Figura 20: Restricciones estáticas sobre el modelo de ascensor 1.

Por ejemplo, la siguiente restricción que aparece en la figura 21 específica que, para todos los ascensores y controladores, cada ascensor esta o bien en el conjunto de ascensores que están bajando o subiendo, y que el controlador corresponde al controlador del edificio al que pertenece el ascensor.

```
all a:Ascensor ,c: Controlador| one c.(bajando+subiendo)[a] && (a.edificio).controlador=c


//Los edificios no compartes ascensores
all disj e1,e2:Edificio| no  e1.ascensores&e2.ascensores

// Cada ascensor esta exactamente en un edificio
all a:Ascensor | one ascensores.a


//Cada planta pertenece solo  a un edificio
all disj p:Planta|one e:Edificio| (p= e.primeraPlanta) || (p = e.ultimaPlanta) ||(p in e.intermedias)

//Los edificios no comparten plantas
all disj e1,e2:Edificio| no  e1.intermedias&e2.intermedias
all e1,e2: Edificio| (e1.primeraPlanta+e1.ultimaPlanta) not in e2.intermedias
all  e:Edificio|no e.primeraPlanta.panterior && no e.ultimaPlanta.psiguiente
all disj e1,e2:Edificio  |(e1.primeraPlanta+e1.ultimaPlanta)!=(e2.primeraPlanta+e2.ultimaPlanta)

//Asociar cada puerta a un solo ascensor
all p: Puerta| one puerta.p

//Las plantas en los edificios se relacionen a través de las relaciones psiguiente y
//panterior de la forma natural
all disj p:Planta| one e:Edificio | plantasEdif[p,e]


//todos los edificiones tienen al menos un ascensor
all e:Edificio| some a:Ascensor|a.edificio=e

//Todos los ascensores estan controlados por un unico controlador

all a:Ascensor | one c:Controlador| a.controlado_por=c

//Cada ascensor esta o bien en el estado bajando, subiendo, o parado y el controlador es el mismo que el del edificio en el que se encuentra
all a:Ascensor ,c: Controlador| one c.(bajando+subiendo)[a] && (a.edificio).controlador=c
```

Figura 21: Restricciones estáticas sobre el modelo de ascensor 2.

El predicado *plantasEdif[p,e]* de la Figura 22 establece la correcta relación entre las plantas. Como se puede observar se específica que cada planta es la primera de un edificio, la última o una de las intermedias. Si la planta es la primera se encontrará en alguna planta anterior a última y, por el contrario, si está en la última o intermedias se encontrará en alguna planta que sigue a la primera planta.

```
pred plantasEdif(p:Planta,e:Edificio){
p in e.primeraPlanta + e.ultimaPlanta+e.intermedias
p = e.primeraPlanta implies p in e.ultimaPlanta.(^panterior)
p in e.intermedias implies  p in e.primeraPlanta.(^psiguiente)
p= e.ultimaPlanta implies  p in e.primeraPlanta.(^psiguiente)
}
```

Figura 22: Predicado relación plantas y edificio del sistema en Alloy.

Restricciones de entorno dinámicas

Las restricciones dinámicas, que dependen del tiempo, suelen especificarse a través de predicados.

Las restricciones que implican relaciones ternarias son algo más complejas como podemos observar en la Figura 23. En las siguientes restricciones se establece cuando un ascensor está subiendo o bajando en un instante de tiempo, dependiendo de su dirección. Dependiendo del estado en el que se encuentre cada uno de los ascensores se especifica si la puerta está abierta o cerrada.

```
pred posicion(t,t':Tiempo, a :Ascensor){
//Si la planta destino esta en alguna de  las siguiente planta estará subiendo
 a.planta_destino in (a.planta_actual.t).psiguiente=>  a in (a.controlado_por).subiendo.t
//Si la planta destino esta en alguna de las plantas anteriores estará bajando
 a.planta_destino in (a.planta_actual.t).panterior =>  a in (a.controlado_por).bajando.t

//Si el estado esta subiendo o bajando se le asignará dicha dirección y las puertas estarán cerradas
 a in  (a.controlado_por).subiendo.t =>  a.direccionAsc.t = Subir && (a.puerta).estado.t = Cerrada

 a in  (a.controlado_por).bajando.t =>   a.direccionAsc.t = Bajar && (a.puerta).estado.t = Cerrada

//Si el ascensor esta parado la puerta estará abierta
 a not in (a.controlado_por).subiendo.t  and a not in (a.controlado_por).bajando.t => (a.puerta).estado.t = Abierta
}
```

Figura 23: Predicado posición del sistema en Alloy.

El predicado de la Figura 24 contiene las restricciones para que un ascensor se mueva de una planta a otra. Tiene varios parámetros de entrada: la planta destino, el ascensor y dos instantes de tiempo consecutivos. Si la planta destino se encuentra en la relación transitiva “psiguiente”, en el instante de tiempo siguiente su planta actual

será la siguiente planta respecto a la que se encuentra, y en la anterior si se encuentra en la relación transitiva “panterior”.

```
pred siguiente_planta(p:Planta,a:Ascensor,t,t':Tiempo){

//El ascensor si esta subiendo va a la siguiente planta si esta bajando a la planta anterior
p in (a.planta_actual.t).^psiguiente => a.planta_actual.t'= (a.planta_actual.t).psiguiente
p in (a.planta_actual.t).^panterior  => a.planta_actual.t'= (a.planta_actual.t).panterior
p in a.planta_actual.t => a.planta_actual.t' =a.planta_actual.t

//Condiciones de marco
todosIgualSalvo[a,t,t']
}
```

Figura 24: Predicado siguiente planta del sistema en Alloy.

Cuando se implementa un predicado de este tipo, es obligatorio especificar las llamadas condiciones de marco. Las condiciones de marco son postcondiciones que deben cumplirse siempre que este predicado sea ejecutado. Las condiciones de marco representan la parte del modelo que no se modifica cuando se ejecuta el predicado. Por ejemplo, en la Figura 25, se muestra la condición de marco que obliga a todos los ascensores, salvo al ascensor a, a quedarse quietos del instante t al t’.

```
pred todosIgualSalvo(a:Ascensor,t,t':Tiempo){
 all a':Ascensor - a| a'.planta_actual.t =a'.planta_actual.t'
}
```

Figura 25: Predicado que define las condiciones de marco del sistema

La política de servicio se define a través del siguiente predicado (en la Figura 26) en el que se establecen dos restricciones básicas para cualquier sistema de ascensores convencional:

1. Un ascensor debe atender cualquier petición realizada desde su interior.

2. Si un ascensor no atiende una petición de planta, ésta será atendida por otro ascensor.

```
pred politicaServicio(a:Ascensor,t,t':Tiempo){

//Un ascensor no puede negar una petición desde dentro del ascensor
no b: ((a.controlado_por).peticiones_ascensor.t + (a.botones&(a.controlado_por).iluminados_ascensor.t)) |  rechazar[t,t',a]

//Si un ascensor deniega una peticion de planta otro lo atenderá
all b: (a.controlado_por).iluminados_planta.t&BotonPlanta- (a.controlado_por).peticiones_planta.t, a1: Ascensor|
rechazar[t,t',a1] =>(some a2: Ascensor|atiende[t,t',a.planta_destino,a] or b in (a.controlado_por).peticiones_planta.t')


}
```

Figura 26: Predicado política de servicio del sistema en Alloy.

Para rechazar o atender peticiones se han definido (en la Figura 27) los predicados rechazar y atiende que establecen si se llama o no al predicado siguiente planta.

```
pred rechazar(t,t':Tiempo,a:Ascensor){
!siguiente_planta[a.planta_destino,a,t,t']
}


pred atiende(t,t':Tiempo,p:Planta,a:Ascensor){
siguiente_planta[p,a,t,t']
}
```

Figura 27: Predicado Rechazar y Atiende del sistema en Alloy.

Para la actualización de botones y peticiones se han definido (Figura 28) los siguientes predicados que especifican que para un instante de tiempo siguiente los botones que permanecerán en los conjuntos, serán los mismo del tiempo actual, salvo los del ascensor que haya alcanzado su planta destino y, además, esté en la misma planta en dos instantes de tiempo consecutivos.

```
pred actualizarIluminacionBotonesAscensor (t,t':Tiempo, a:Ascensor){

(a.controlado_por).iluminados_ascensor.t'=(a.controlado_por).iluminados_ascensor.t -
{b:BotonAscensor|a.planta_actual.t in a.planta_destino && b in a.botones && b.planta in (a.planta_actual.t+a.planta_actual.t')}
}

pred actualizarIuminacionBotonesPlanta(t,t':Tiempo, a:Ascensor){

(a.controlado_por).iluminados_planta.t'=(a.controlado_por).iluminados_planta.t -
{b:BotonPlanta|a.planta_actual.t in a.planta_destino &&   botones_planta.b in (a.planta_actual.t+a.planta_actual.t')}
}

pred actualizarPeticionesAscensor(t,t':Tiempo, a :Ascensor){
(a.controlado_por).peticiones_ascensor.t'=(a.controlado_por).peticiones_ascensor.t -
{b:BotonAscensor|a.planta_actual.t in a.planta_destino && b in a.botones && b.planta in (a.planta_actual.t+a.planta_actual.t')}

}

pred actualizarPeticionesPlanta(t,t':Tiempo, a:Ascensor){
(a.controlado_por).peticiones_planta.t'=(a.controlado_por).peticiones_planta.t -
{b:BotonPlanta|a.planta_actual.t in a.planta_destino &&   botones_planta.b in (a.planta_actual.t+a.planta_actual.t')}
}
```

Figura 28: Predicado actualizar iluminación y peticiones del sistema

El predicado transición de la Figura 29 llama a los predicados que deben satisfacerse en cada instante de tiempo actual y siguiente.

```
pred Transicion (a:Ascensor,t,t':Tiempo) {
-- movimiento y posición validas
posicion[t,t',a]
siguiente_planta[a.planta_destino,a,t,t']

-- la politica del servicio es satisfecha
politicaServicio[a,t,t']

-- los botones son actualizados correctamente
actualizarIluminacionBotonesAscensor[t,t',a]
actualizarIuminacionBotonesPlanta[t,t',a]
actualizarPeticionesAscensor[t,t',a]
actualizarPeticionesPlanta[t,t',a]

}
```

Figura 29: Predicado transición del sistema en Alloy.

La Figura 30 es un ejemplo que muestra el sistema en movimiento se obtiene mediante un predicado **trazas** que itera sobre los distintos instantes de tiempo.

```
pred traza(){

#Planta=4
#Ascensor=1
#Edificio=1

 inicio[first]

  all t:Tiempo-last|
    let t' = next[t] |some a:Ascensor| Transicion[a,t,t']


}

run traza for 11
```

Figura 30: Predicado traza del sistema en Alloy.

3.3 Verificación con Alloy Analyzer

Para que la herramienta Alloy Analyzer encuentre instancias del modelo creado, hay que presionar el botón “execute” señalado en la Figura 31. Evidentemente, el analizador encontrará instancias del modelo, sólo si el predicado es consistente.

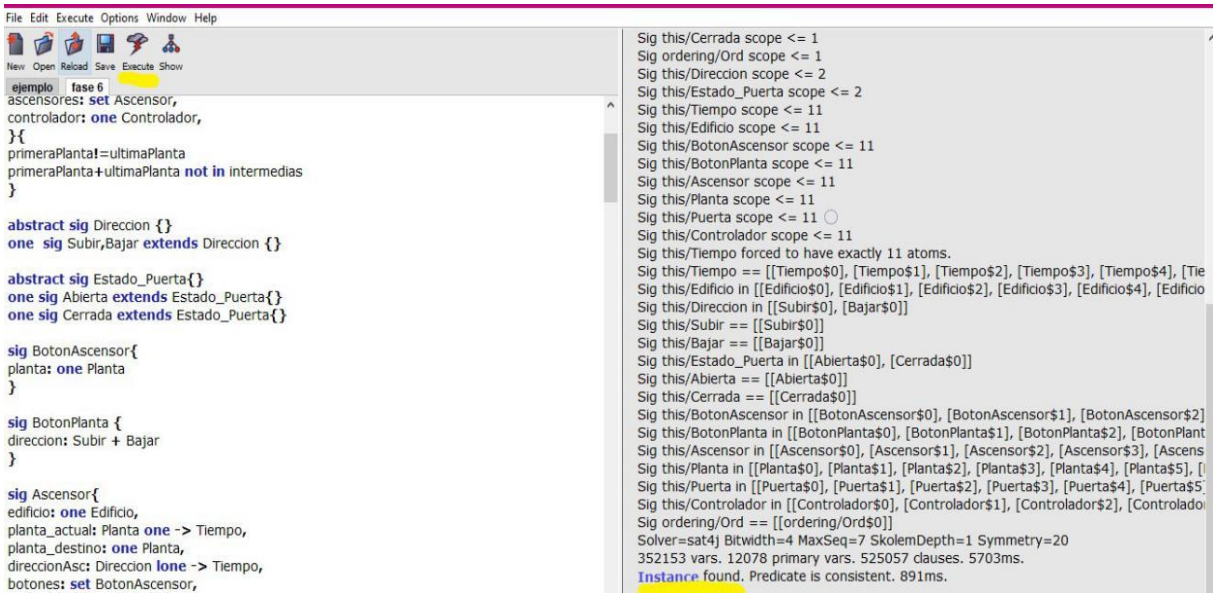

Figura 31: Captura del analizador de Alloy.

A continuación, en las siguientes figuras, se muestra cómo el sistema modelado se mueve correctamente de planta en planta en instantes de tiempos consecutivos, actualizando la iluminación y las peticiones, y llegando a la planta destino correspondiente. El orden de las plantas en la instancia encontrada es el siguiente:

Primera planta: <Planta 2>

Intermedias: < Planta 0, Planta 3 (Planta destino)>

Última planta: <Planta 1>

En la Figura 32, se muestra el estado del sistema en el instante de tiempo 0. El ascensor se encuentra en la planta 2 (primera planta) y su planta destino es la 3 (corresponde a la tercera planta), la puerta se encuentra abierta y sus peticiones de botones e iluminación se corresponden con los botones de la planta 3.

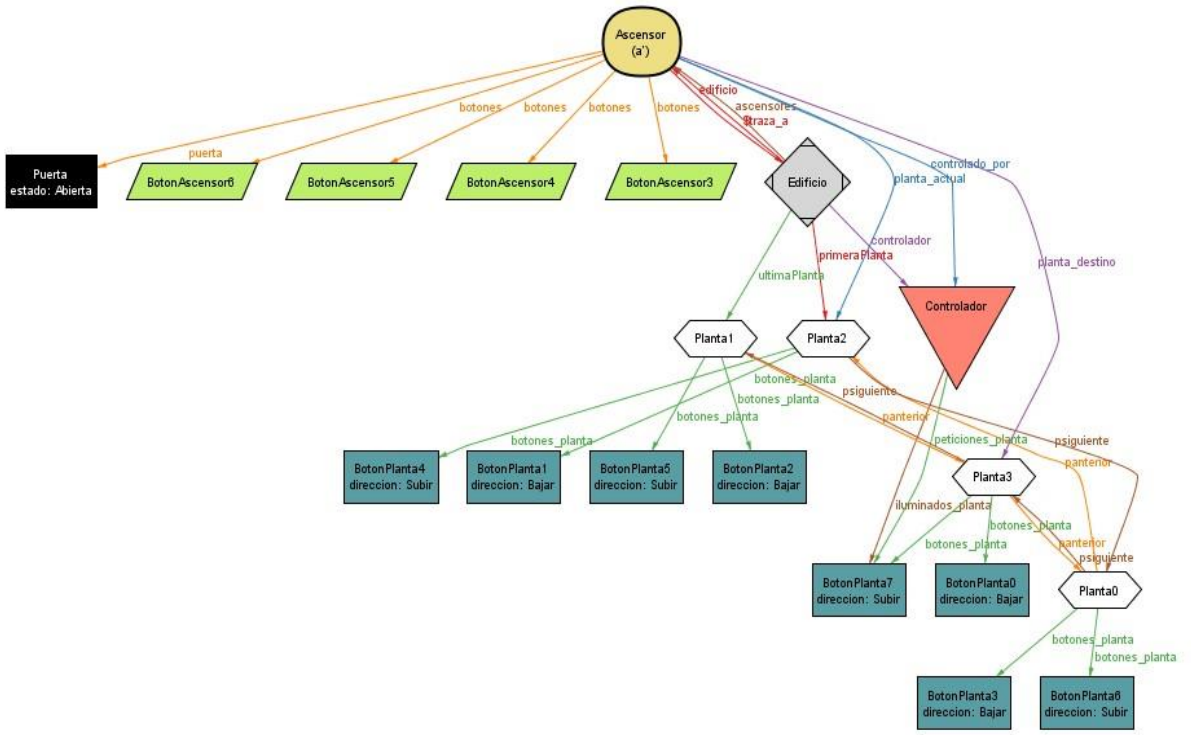

Figura 32: Instancia del modelo generado por el verificador de Alloy (Tiempo 0).

La Figura 33 muestra el sistema en el siguiente tiempo (Tiempo 1). La puerta está en estado Cerrada ya que el ascensor se ha movido a la siguiente planta (Planta 0) y se encuentra en el conjunto de ascensores que están subiendo.

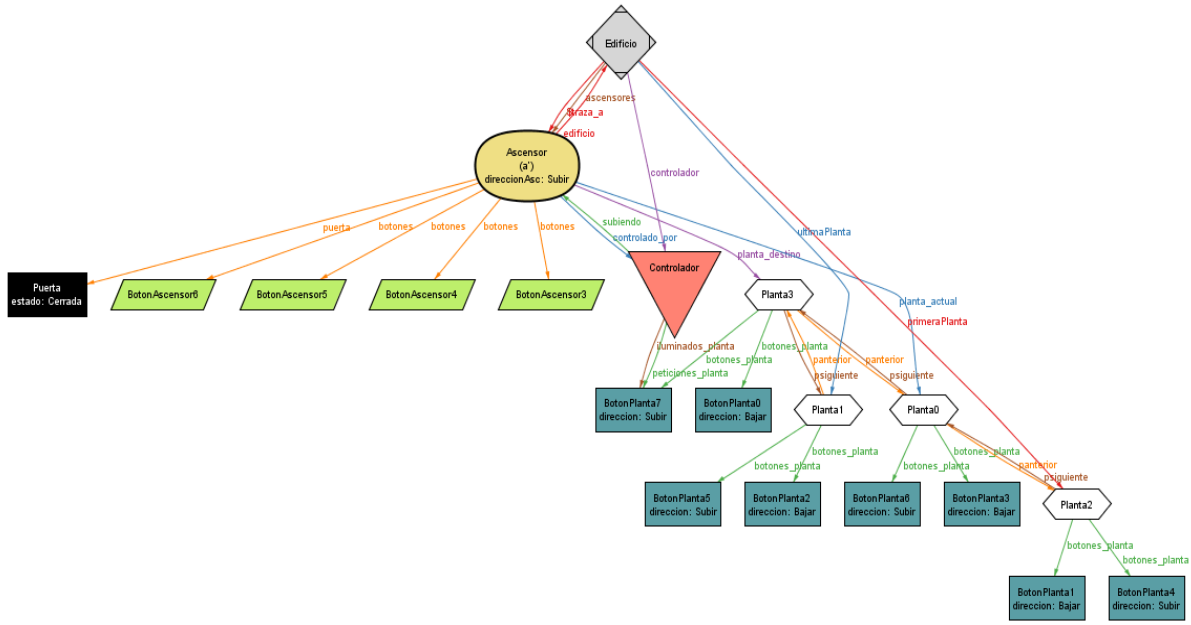

Figura 33: Instancia del modelo generado por el verificador de Alloy (Tiempo 1).

La Figura 34 muestra el sistema en el instante de tiempo siguiente (Tiempo 2). La planta actual es la planta destino (Planta 3), la puerta se ha abierto, el ascensor está parado y se han actualizado correctamente la petición y la iluminación del botón del ascensor que fue pulsado.

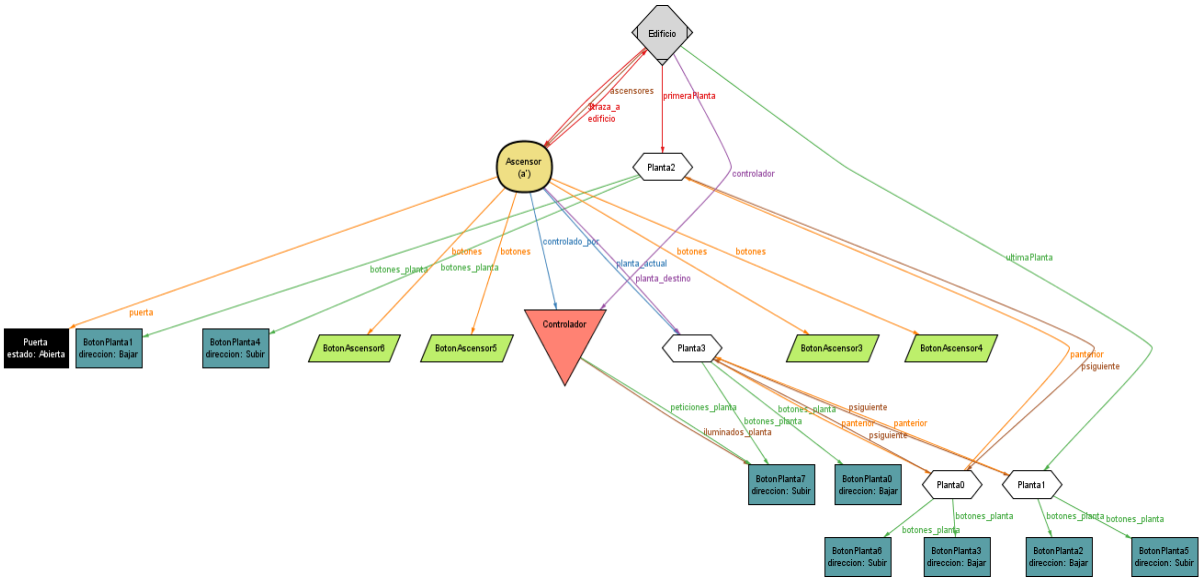

Figura 34: Instancia del modelo generado por el verificador de Alloy (Tiempo 2).

Alloy permite modificar la visualización de relaciones, atributos y otras opciones sobre la vista. Para observar mejor el ejemplo anterior, en la Figura 35, se han ocultado algunas relaciones como, por ejemplo, las plantas a las que pertenecen los botones.

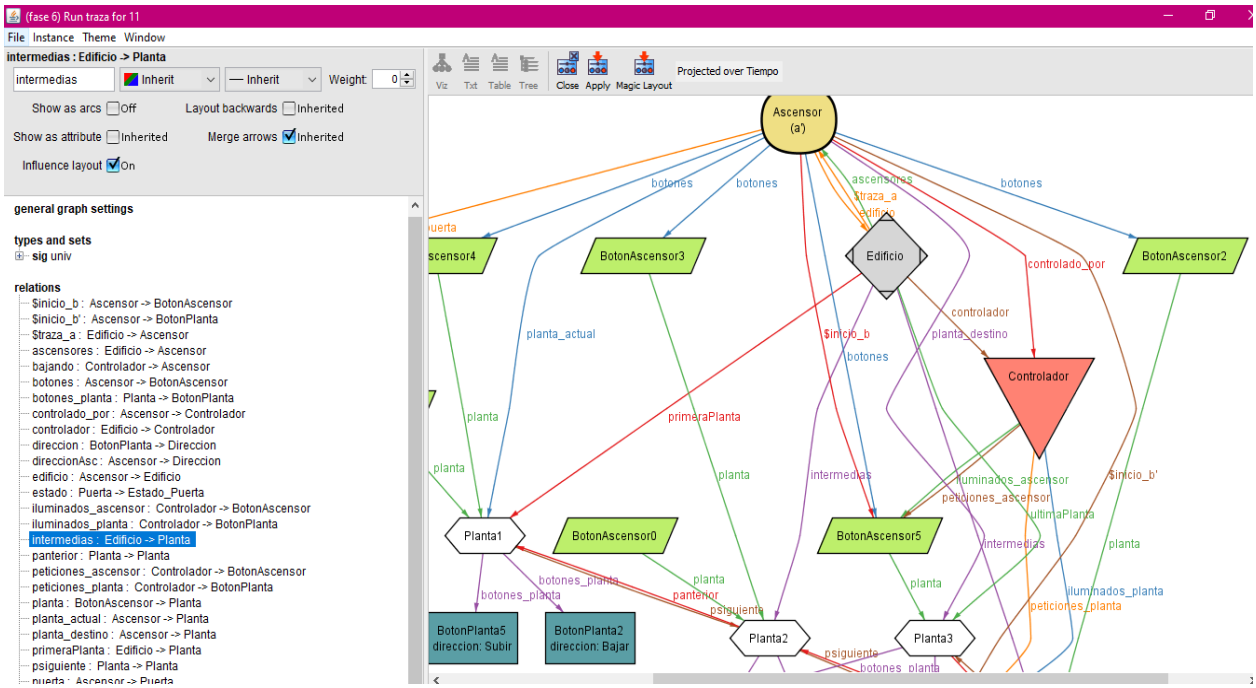

Figura 35: Captura opciones de visualización de Alloy.

VISUALIZANDO UN CONTRAEJEMPLO A UN ASERTO

En esta sección, se muestra cómo comprobar si un aserto es cierto o, por el contrario, es falso en cuyo caso la herramienta encuentra una instancia que no lo satisface (es decir, un contraejemplo). En concreto, se comprobará si es posible que un botón en un instante de tiempo esté iluminado y en el instante siguiente no lo está. El aserto Alloy se muestra en la Figura 36.

```
assert actualizaBotones {
some a:Ascensor,t,t':Tiempo | some b: (a.controlado_por).iluminados_ascensor.t | b not in (a.controlado_por).iluminados_ascensor.t'
}
check  actualizaBotones  for 3 Ascensor, 9 BotonAscensor, 3 Planta, 5 Tiempo,1 Edificio, 6 BotonPlanta, 1 Controlador, 3 Puerta
```

Figura 36: Aserción actualización de botones del sistema en Alloy.

```
Sig this/Puerta in [[Puerta$0], [Puerta$1], [Puerta$2]]
Sig this/Controlador in [[Controlador$0]]
Sig ordering/Ord == [[ordering/Ord$0]]
Solver=sat4j Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20
5037 vars. 453 primary vars. 8736 clauses. 359ms.
Counterexample found. Assertion is invalid. 94ms.
```

Figura 37: Análisis del aserto actualización de botones en Alloy.

La Figura 37 muestra que la herramienta Alloy encuentra un contraejemplo para el aserto descrito. La instancia contraejemplo se encuentra en la Figura 38. Es una instancia en la que un ascensor está en su planta destino, sin ninguna petición más. Por lo que el analizador encuentra una instancia en la que el botón no cumple la condición descrita.

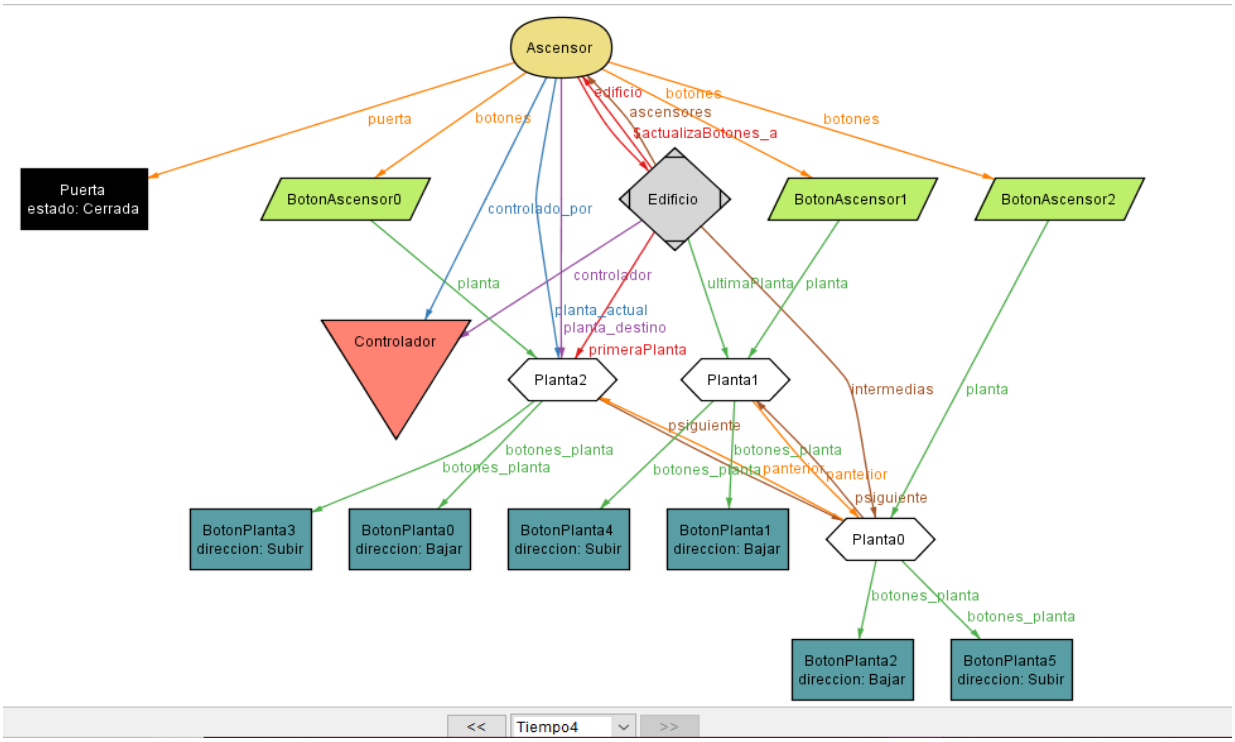

Figura 38: Contraejemplo encontrado por el analizador Alloy.

4 Modelado y verificación con SPIN

Durante los últimos años, la herramienta SPIN ha atraído mucho interés de universitarios que enseñan métodos formales y también por desarrolladores en la industria.

Sus méritos incluyen un lenguaje de diseño simple pero poderoso basado en canales síncronos y asíncronos, así como una lógica expresiva para la verificación de las propiedades. En este documento comparamos la herramienta SPIN existente (sin tiempo) con otra herramienta que dispone de tiempo real, Uppaal, que utiliza especificaciones descritas como autómatas temporizados. En este último caso, se presenta un ejemplo sencillo aplicado a nuestro sistema de ascensores. Las propiedades sin tiempo se analizan en SPIN y las propiedades con tiempo real se verifican usando un modelo de autómatas y Uppaal.

En esta sección, describimos el modelo de sistema de ascensores que se ha realizado utilizando el lenguaje Promela que es el lenguaje de entrada de la herramienta SPIN, el cual no es lenguaje de programación en sí sino más bien un lenguaje para construir modelos de verificación. También se describe la verificación del modelo respecto a algunas propiedades críticas descritas en la lógica temporal lineal (LTL).

Los algoritmos de verificación de modelos se basan en la exploración exhaustiva de todo el espacio de estados generado por un sistema. Esto significa que la complejidad de los algoritmos es proporcional al espacio de estados generado por el sistema [18].

Con SPIN es posible realizar una exploración exhaustiva de todas las posibles ejecuciones del sistema modelado y analizar si una propiedad se cumple en todas ellas. En nuestro caso, hay que tener en cuenta que dependiendo del número de procesos con los que cuente el sistema, el tamaño del espacio de estados puede aumentar considerablemente, pero SPIN se puede llegar a verificar una gran cantidad de estados por lo que es una herramienta de verificación bastante válida. Además, en modo de simulación, SPIN se puede utilizar para obtener una visión de los tipos de comportamiento

4.1 Diseño con el lenguaje promela

Cada entidad es representada en Promela por un proceso. Nuestro sistema consta de n procesos y m canales de capacidad 0, a través de los cuales se comunican los procesos.

Nuestro modelo principal cuenta con los procesos ascensor, planta, controlador y entorno que son instancias de proctypos y se utilizan para definir el comportamiento del SCA.

El cuerpo de un proceso está formado por cero o más declaraciones de datos. La semántica de la ejecución de enunciados es el mecanismo principal para sincronizar a los procesos.

Añadiendo la palabra active al proceso se genera una instancia del mismo, si necesidad de crearlo explícitamente. Cada proceso que se encuentra en ejecución cuenta con un número identificador único y puede dar un paso en la ejecución solo cuando sea ejecutable.

El código de la Figura 39 cuenta define arrays de tamaño 3, siendo éste el número de plantas y ascensores con los que cuenta el sistema. Las variables globales son modificadas por los procesos a lo largo de la ejecución del sistema.

```
#define nPlanta 5
#define nAscen 3

/*****Estados Ascensor*******/

mtype = {parado, moviendose, inactivo}
byte libres=nAscen;

/*******Arrays*******/

bool puertas[nAscen];
bool aocupados[nAscen];
byte boton_ascensor[nAscen];
byte posicion_actual[nAscen];
mtype estado_ascensor[nAscen];

/******Comunicacion entre Ascensor,controlador......*****/

/*Respuesta ascensor*/
mtype = {abrir_ack, time_out, esperando, cerrar_ack};
/*Peticiones ascensor*/
mtype = {estado_peticion, peticion_abrir, peticion_cerrar,llegada_ack};

/*******Arrays Planta*******/
bool peticiones_planta[nPlanta]; //peticiones de planta
bit iluminados_subir[nPlanta];
bit iluminados_bajar[nPlanta];
```

Figura 39: Arrays y canales del sistema de ascensores en promela 1.

Todas las comunicaciones entre los procesos se modelan mediante el envío de mensajes a través de los canales. Se utilizan ocho canales de sincronización (de capacidad cero) cada uno tal y como se muestra en la Figura 40.

```
/****Canales de longitud 0*****/
chan plantaAControl = [0] of {bit,byte}
chan plantaAAscensor[nAscen]= [0] of {mtype}
chan controlAAscensor[nAscen] = [0] of {bit, byte}
chan clientesAPlanta[nPlanta] = [0] of {bit}
chan clientesAAscensor[nAscen] = [0] of {byte}
chan ascensorAControl =[0] of {mtype,byte}
chan controladorAscensor[nAscen]=[0] of {mtype}
chan ascensorAPlanta[nPlanta]= [0] of {byte,bit}
```

Figura 40: Arrays y canales del SCA en promela 2.

La comunicación entre los procesos en el sistema y los mensajes que envía cada canal sigue la lógica definida en el diagrama de la Figura 41.

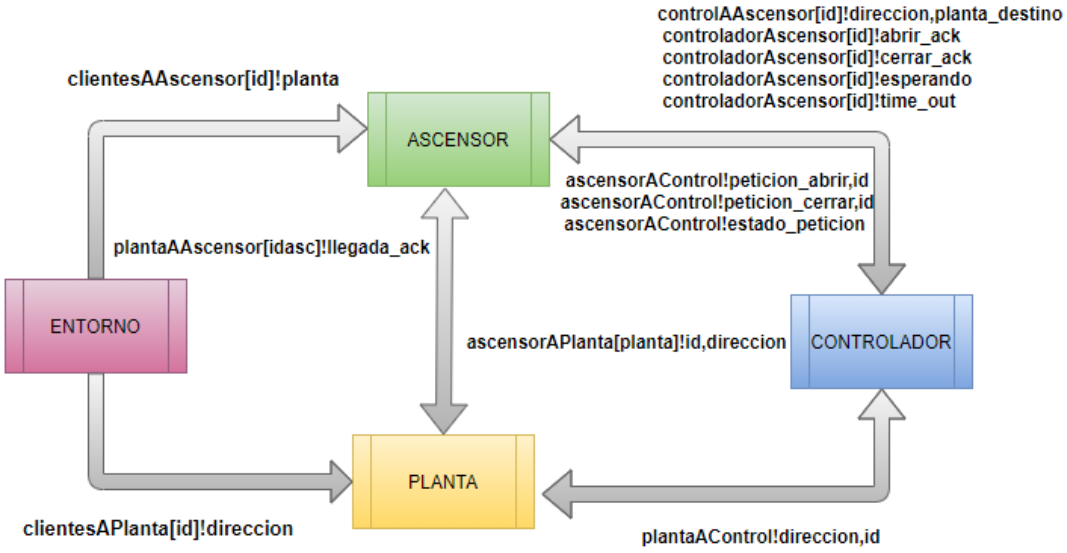

Figura 41: Diagrama de comunicación entre los procesos del sistema de ascensores.

Hay distintos tipos de procesos:

1. El tipo Ascensor, que representa los ascensores del edificio. Todos los ascensores se comportan de la misma forma, solo se diferencian por su identificador.

2. El tipo Controlador, del que hay solo una instancia, y que gestiona el comportamiento correcto de los ascensores.

3. El tipo Planta, que representa cada una de las plantas del edificio, con sus respectivos botones.

4. El entorno representa las llamadas de los usuarios tanto desde planta como desde el interior de los ascensores.

Todas las plantas se comportan del mismo modo, solo se diferencian por su identificador.

Un cliente puede llamar al ascensor de dos formas distintas:

1.Desde dentro de un ascensor (llamada interna): En este caso, se pulsa un botón desde dentro del ascensor y el ascensor responde moviéndose hacia la planta seleccionada.

2.Desde una planta (llamada externa): En este caso el usuario pulsa el botón de subida o de bajada desde una planta, que se ilumina después de ser pulsado. A continuación, la planta se pone en contacto con el controlador, que

busca un ascensor libre y lo envía a la planta. Cuando el ascensor llega se apaga el botón correspondiente.

La siguiente tabla (Tabla 3) muestra el significado de cada mensaje que se intercambia entre los procesos a través de cada canal de sincronización:

clientesAAscensor[id]!planta	Cada canal clientesAAscensor[i] realiza peticiones internas, es decir sirve para comunicar al cliente con el ascensor. El mensaje lleva la planta destino que se ha solicitado.
clientesAPlanta[id]!direccion	Cada canal clientesAPlanta[i] realiza peticiones externas, y el mensaje lleva la dirección deseada por el usuario de subida o de bajada.
ascensorAPlanta[id]!direccion	Cada canal ascensorAPlanta[i] comunica al ascensor con la planta i. El ascensor indica a la planta a través del mensaje su dirección(subir o bajar) para que esta pueda apagar el botón correspondiente.
plantaAControl!direccion,planta	Cuando un cliente realiza una llamada externa el canal plantaAControl envía al controlador un mensaje con el formato (dirección,planta) que indica la dirección (subir o bajar) deseada por el usuario y la planta desde dónde se ha realizado la petición.
controlAAscensor[id]!direccion,planta	Cada canal controlAAscensor[i] comunica al controlador con el ascensor i. El controlador reenvía el mensaje que la planta le envío al ascensor indicando la dirección y la planta destino.
ascensorAControl!peticion_abrir	El canal ascensorAControl le envía un mensaje al controlador indicando que debe abrir las puertas, y el id del ascensor que lo solicita.
ascensorAControl!peticion_cerrar	El canal ascensorAControl le envía un mensaje al controlador indicando que debe cerrar las puertas, y el id del ascensor que lo solicita.
ascensorAControl!estado_peticion	El canal ascensorAControl le envía un mensaje al controlador indicando que quiere recibir el estado del ascensor , y el id del ascensor que lo solicita.

controladorAscensor!abrir_ack	El canal controladorAscensor envía al ascensor la confirmación de que las puertas se han abierto.
controladorAscensor!cerrar_ack	El canal controladorAscensor envía al ascensor la confirmación de que las puertas se han cerrado.
controladorAscensor!esperando	El canal controladorAscensor envía al ascensor el mensaje esperando, el cual indica que se ha producido una petición interna antes de superar el intervalo de tiempo establecido para que el ascensor quede libre.
controladorAscensor!time_out	El canal controladorAscensor envía el mensaje indicando que el intervalo de tiempo establecido ha sido superado, las puertas deben cerrarse y por lo tanto el ascensor queda inactivo.
plantaAAscensor[idasc]!llegada_ack	El canal plantaAAscensor envía el mensaje indicando que el ascensor ha llegado a la planta destino.

Tabla 3: Mensajes de comunicación de los canales del sistema de ascensores en Promela.

A continuación, se describe brevemente la implementación de cada uno de los procesos del sistema.

El Controlador:

El proceso Controlador espera hasta recibir datos y luego envía un mensaje al proceso correspondiente.

Cuando se pulsa un botón de planta, el controlador recibe la petición y busca si hay algún ascensor libre. En este caso, el Controlador envía un mensaje al proceso Ascensor indicándole la dirección y la planta a la que debe dirigirse. Si todos los ascensores están ocupados, se apagan los botones de planta y no se realiza ninguna otra acción.

Por otro lado, el proceso del Controlador (Figura 42) gestiona cuando se abren y se cierran las puertas de los ascensores. Cuando recibe una petición para abrir una puerta, ésta se abre y le envía un mensaje de confirmación al proceso Ascensor. Más tarde recibirá un mensaje del Ascensor con su estado. Si se ha producido una petición desde dentro del ascensor las puertas se cerrarán, y se le enviará un mensaje al Ascensor “esperando”, este mensaje indica que el ascensor ha recibido una petición interna y sigue ocupado, en caso de que no se haya pulsado ningún botón se enviará el mensaje “time out” y dicho Ascensor quedará libre.

Cuando recibe un mensaje solicitando cerrar la puerta, éstas se cierran y se envía al proceso Ascensor un mensaje de confirmación de cierre.

El Ascensor:

El proceso Ascensor (Figura 43), por un lado, recibe un mensaje del controlador con la dirección y la planta a la que tiene que dirigirse, cuando el controlador ha recibido una petición de una planta y el ascensor está libre.

```
proctype Controlador(){
int i =0;
byte id;
bit direccion;
byte planta_destino;
end: do
  ::plantaAControl?direccion,planta_destino->
if
::libres>=1->
   do
    ::i==nAscen->break;
    ::else->
     if
      ::aocupados[i]==false-> peticiones_planta[planta_destino]=true;aocupados[i]=true;libres--;controlAAscensor[i]!direccion,planta_destino;break;
      ::else->skip;
     fi;
      i++;

  od;
::else->peticiones_planta[planta_destino]=false; apagar_botones(id,direccion);
 fi;


::ascensorAControl?peticion_abrir,id->
        puertas[id]=true;
        controladorAscensor[id]!abrir_ack;
        ::ascensorAControl?estado_peticion,id->
          if
           ::clientesAAscensor[id]?planta_destino->
            if
            ::estado_ascensor[id]==parado->puertas[id]=false;  controladorAscensor[id]!esperando;
            ::else->skip;
            fi;

        ::else->controladorAscensor[id]!time_out;
         fi;
::ascensorAControl?peticion_cerrar,id->
          puertas[id]=false;
          controladorAscensor[id]!cerrar_ack;

od;
}
```

Figura 42: Proceso Controlador del sistema de ascensores en Promela.

```
 proctype Ascensor(byte id){
bit direccion;
byte planta;
posicion_actual[id]=0;
end: do           /*El ascensor se mueve hacia la planta solicitada*/
  ::controlAAscensor[id]?direccion,planta->
Inicio:
estado_ascensor[id]=moviendose; goto Moviendose;
Moviendose:
                if
                ::posicion_actual[id]==planta->estado_ascensor[id]=parado; goto Parado;
                ::else->
                 mover(posicion_actual[id],direccion,id);  goto Moviendose;
                fi;
Parado: peticiones_planta[planta]=false;
                ascensorAPlanta[planta]!id,direccion;
                plantaAAscensor[id]?llegada_ack;
                ascensorAControl!peticion_abrir,id;
                controladorAscensor[id]?abrir_ack;

EnviarEstado:     ascensorAControl!estado_peticion,id;
Respuesta:
                  if
    :: controladorAscensor[id]?esperando ->goto Cliente;
    :: controladorAscensor[id]?time_out -> ascensorAControl!peticion_cerrar,id; goto Respuesta;
    :: controladorAscensor[id]?cerrar_ack->estado_ascensor[id]=inactivo;
                    if
                    ::aocupados[id]==true->aocupados[id]=false;libres++;
                    ::else->skip;
                   fi;

 //   goto Inicio;
fi;
```

Figura 43: Petición externa al proceso Ascensor.

El estado del ascensor pasa a ser “Moviéndose” y la posición del ascensor va cambiando en función de código inline que se ejecuta hasta que se llega a la planta destino (Figura 44).

```
inline mover(actual,dir,id){
if
::dir==1 && (actual>=0) && (actual<(nPlanta-1))->posicion_actual[id]=actual+1;
::else->skip;
fi
if
::dir==0 && (actual>0) && (actual<=(nPlanta-1))->posicion_actual[id]=actual-1;
::else->skip;
fi
}
```

Figura 44: Definición de mover del sistema de ascensores.

Los procesos utilizan instrucciones “goto” para saltar de un estado a otro y para salir de los bucles.

Cuando la posición del Ascensor es la planta destino, el estado del mismo pasa a ser “Parado”, y el Ascensor envía un mensaje a la Planta para que se apaguen lo botones de ésta. A continuación, envía un mensaje al Controlador para que las puertas se abran. El ascensor envía un mensaje al Controlador de petición de estado, si el ascensor recibe la repuesta “esperando” por una petición interna, el proceso salta al estado “Cliente” (Figura 45).

Si recibe el mensaje “timeout”, el Ascensor envía una petición para cerrar las puertas al Controlador. Cuando recibe la confirmación de cierre, el estado del Ascensor pasa a ser “inactivo”.

Cuando se produce una petición interna desde el Ascensor, si éste estuviese libre, y el pasajero en el interior del Ascensor, pasaría a estar ocupado. El proceso de movimiento es el mismo explicado en la petición de ascensor a planta.

```
/*Peticion interna ascensor*/
::clientesAAscensor[id]?planta->
Cliente:
boton_ascensor[id]=planta;
if
::(aocupados[id]==false)->aocupados[id]=true;libres--;
::else->skip;
fi;
/*Direccion del ascensor*/
 if
::posicion_actual[id] >planta->direccion=0;
::else->direccion=1;
fi;
if
::(estado_ascensor[id]==inactivo || estado_ascensor[id]==parado || estado_ascensor[id]==0)->
 estado_ascensor[id] = moviendose;
 goto Moviendose;
::else->break;
fi;
od;
}
```

Figura 45: Petición interna del sistema de ascensores.

La Planta:

El proceso Planta (Figura 46) cuando recibe una petición externa de un cliente al pulsar un botón de planta esté envía un mensaje al proceso Controlador con la dirección y la planta solicitada, y se ejecuta un código inline para encender los botones de planta.

Por otro lado, el proceso Planta recibe un mensaje del proceso Ascensor cuando llega a dicha planta para apagar los botones.

```
proctype Planta(byte id){
bit direccion;
byte idasc;
end: do
::clientesAPlanta[id]?direccion-> encender_botones(id,direccion);plantaAControl!direccion,id;
::ascensorAPlanta[id]?idasc,direccion-> apagar_botones(id,direccion);plantaAAscensor[idasc]!llegada_ack;
od;
}
```

Figura 46: Proceso Planta del sistema de ascensores en Promela.

Para encender y apagar los botones se usa el código inline ya mencionado que se muestran en la Figura 47:

```
inline encender_botones(id,dir){
if
::dir==1 -> iluminados_subir[id]=1;
::else->iluminados_bajar[id]=1;
fi;
}
inline apagar_botones(id,dir){
if
::dir==1 -> iluminados_subir[id]=0;
::else->iluminados_bajar[id]=0;
fi;
}
```

Figura 47: Funcioes apagar y encender botones.

El Entorno:

El entorno simula el envío de mensajes de los clientes tanto al proceso Planta como al proceso Ascensor (Figura 48).

```
active   proctype Entorno(){
 if
 ::clientesAPlanta[0]!0 ;
 ::clientesAPlanta[0]!1;

 ::clientesAPlanta[1]!0 ;
 ::clientesAPlanta[1]!1;

 ::clientesAPlanta[2]!0 ;
 ::clientesAPlanta[2]!1;
 ::clientesAPlanta[3]!0 ;
 ::clientesAPlanta[3]!1;
 ::clientesAPlanta[4]!0 ;
 ::clientesAPlanta[4]!1;


 ::clientesAAscensor[0]!0 ;
 ::clientesAAscensor[0]!1 ;
 ::clientesAAscensor[0]!2 ;

::clientesAAscensor[1]!0 ;
::clientesAAscensor[1]!1 ;
::clientesAAscensor[1]!2;

::clientesAAscensor[2]!0 ;
::clientesAAscensor[2]!1 ;
::clientesAAscensor[2]!2;
fi;
}
```

Figura 48: Proceso Entorno del sistema de ascensores en Promela.

4.2 Verificación del modelo

La comprobación de modelos (model checking) es una técnica que consiste en construir un modelo finito de un sistema y comprobar si determinadas propiedades deseadas del sistema se cumplen en el modelo [19].

La herramienta SPIN puede usarse para realizar la verificación de sistemas concurrentes. En esta sección se verificará el SCA con propiedades LTL.

Resultados de la verificación

Se han hecho 3 verificaciones distintas del sistema con un entorno que envía mensajes finitos, incrementando el número de procesos los resultados son los mostrados en la Tabla 4:

Número de procesos	Estados alcanzados	Estados verificados
5 plantas, 3 Ascensores	2857	3269
10 plantas, 3 Ascensores	4917	3269
20 plantas, 5 ascensores	22929	49185

Tabla 4: Comparación de resultados de verificación con distintos números de procesos.

Como se puede observar con un aumento tan pequeño de los procesos de tipo Ascensor o Planta, el aumento del espacio de estados aumenta considerablemente. La Figura 49 muestra la salida del verificador.

```
(Spin Version 6.4.7 -- 16 July 2017)

Full statespace search for:
        never claim         - (not selected)
        assertion violations +
        acceptance   cycles  + (fairness disabled)
        invalid end states +

State-vector 576 byte, depth reached 105, errors: 0
    4917 states, stored
    5749 states, matched
   10666 transitions (= stored+matched)
      13 atomic steps
hash conflicts:         3 (resolved)

Stats on memory usage (in Megabytes):
    2.795   equivalent memory usage for states (stored*(State-vector + overhead))
    1.038   actual memory usage for states (compression: 37.13%)
            state-vector as stored = 201 byte + 20 byte overhead
  128.000 memory used for hash table (-w24)
   53.406  memory used for DFS stack (-m1000000)
  182.382 total actual memory usage
```

Figura 49: Resultados de la verificación con la herramienta SPIN.

Gráfico de la comunicación entre los procesos

En el siguiente diagrama (Figura 50), se puede observar cómo se produce la comunicación entre los procesos.

ESCENARIO:

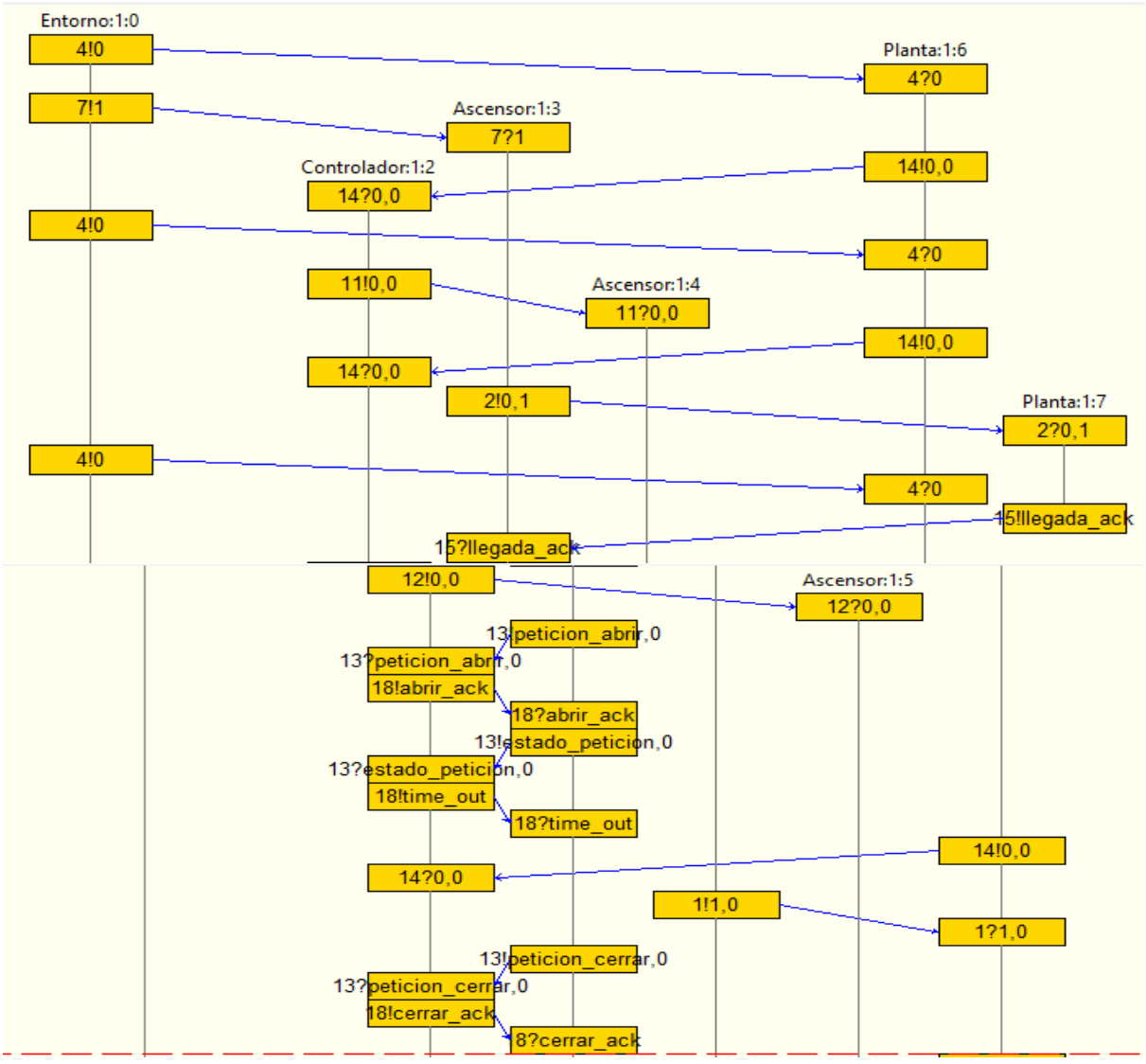

Figura 50: Gráfico de la comunicación de procesos generado por la herramienta Spin.

Mediante la lógica temporal se pueden especificar, expresar y razonar los comportamientos dinámicos del sistema, debido a que Promela permite comprobar si determinadas propiedades expresadas en LTL (Lógica Temporal Lineal) son ciertas.

La correcta especificación de un sistema depende, en gran parte, del éxito o el fracaso en su desarrollo. En SPIN, la verificación es un proceso completamente automático que nos permite comprobar si el sistema modelado satisface las propiedades especificadas.

Esta comprobación se realiza de forma automática y relativamente rápida si se maneja un espacio de estados que sea manejable.

Si la verificación encuentra un contraejemplo, lo devuelve al usuario, lo que da la posibilidad de depurar el sistema encontrando cual ha sido la causa del error. La verificación automática no está centrada en demostrar la corrección de un modelo, sino que busca encontrar errores lo más pronto posible durante el análisis del sistema.

Para verificar las propiedades sobre el sistema se han definido variables globales, definidas con anterioridad, y se han especificado las siguientes propiedades en LTL (la versión LTL aparece más adelante en esta sección):

R1. Siempre que el estado de un ascensor sea en movimiento, estará en movimiento hasta que el ascensor se pare.

R2. Siempre que se haya pulsado un botón del ascensor en algún momento la planta actual del ascensor será la que corresponde al botón pulsado.

R3. Siempre que el ascensor este inactivo, en algún momento siguiente el ascensor estará en movimiento.

R4. Siempre que el controlador reciba una petición, en algún momento llega un ascensor a la planta desde la que se realizó la petición externa.

R5. Si un ascensor se está moviendo, en algún momento llega a su destino. Como todos los ascensores tienen el mismo código, basta con probar la propiedad para un ascensor concreto.

R6. Siempre que las puertas estén abiertas implica o bien que el ascensor está parado o bien que está inactivo

R7. Siempre que el ascensor este en movimiento implica que las puertas están cerradas.

R8. Siempre que haya peticiones a planta en algún momento el ascensor estará en movimiento.

R9. Nunca pueden estar las puertas abiertas y el ascensor en movimiento.

R10. Las luces de las plantas permanecen encendidas desde que se pulsan hasta que un ascensor llega a su planta destino.

A continuación, se presenta una introducción a la sintaxis de la lógica temporal lineal (LTL). Luego, daremos semántica a las propiedades especificadas.

Sintaxis LTL

Las LTL se construyen a través de un conjunto finito de variables proposicionales *AP* , los operadores lógicos ¬ ,∨ ,∧, y los temporales operadores modales X y U . Formalmente, el conjunto de fórmulas LTL sobre *AP* se define de la siguiente manera [20]:

- si p ∈ *AP,* entonces p es una fórmula de LTL;
- si ψ y φ son fórmulas LTL, entonces ¬ψ, φ ∨ ψ, X ψ y φ U ψ son fórmulas LTL.

A partir de la negación y la conjunción, definimos los conectivos booleanos usuales usando las reglas de Morgan:

- ψ ∨ φ ≡ ¬ (¬ψ ∧ ¬φ) (disyunción)
- ψ → φ ≡ ¬ ψ ∨ φ (implicación)

A partir de la sintaxis, definimos lo siguiente:

- True ≡ ψ ∨¬ ψ
- False ≡ ¬True

PROPIEDADES DE VIVEZA

Las propiedades de viveza se verifican marcando la opción ACCEPTANCE CYCLES en la pestaña verificación de la herramienta como se muestra en la Figura 51.

En esta propiedad de viveza, no sé encuentra ninguna traza de error, es decir que la restricción correspondiente se satisface para todas las trazas generadas. La restricción es la siguiente:

```
[]((estado_ascensor[id]==moviendose)-><> (estado_ascensor[id]= = parado))
```

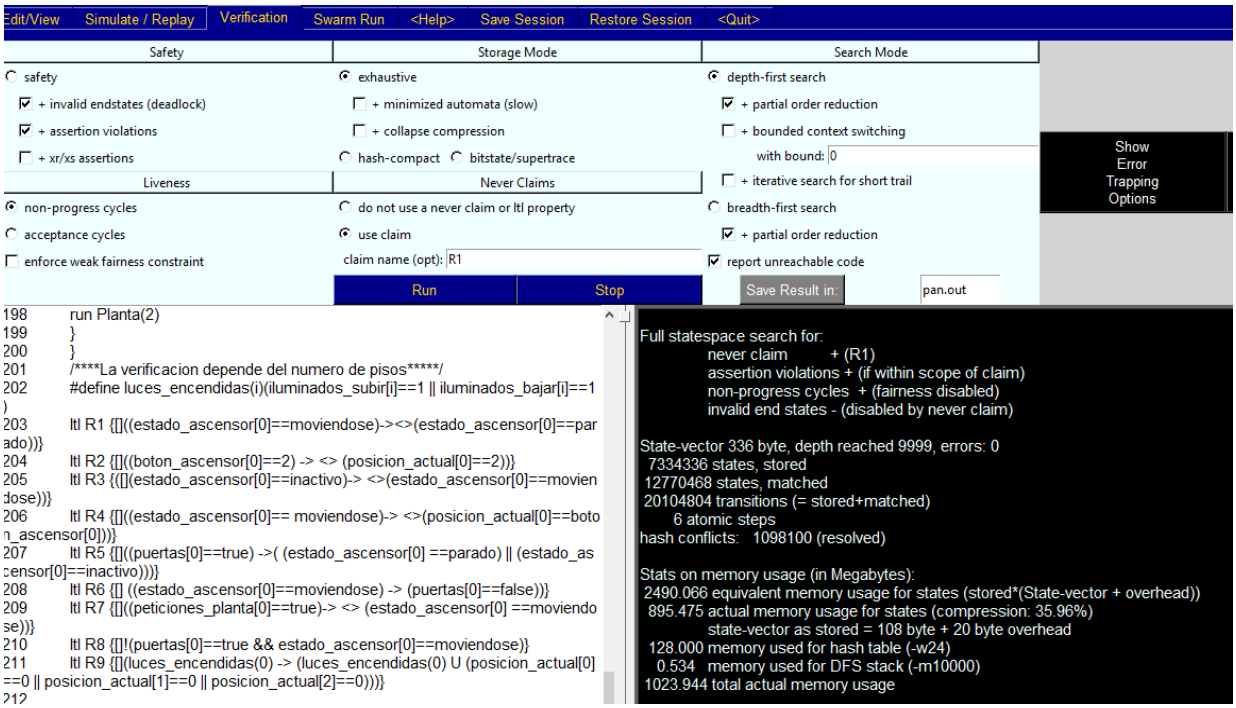

Figura 51: Captura del resultado de la verificación de una propiedad de viveza.

PROPIEDADES DE SEGURIDAD

Las propiedades de seguridad se verifican marcando la opción safety (Figura 52).

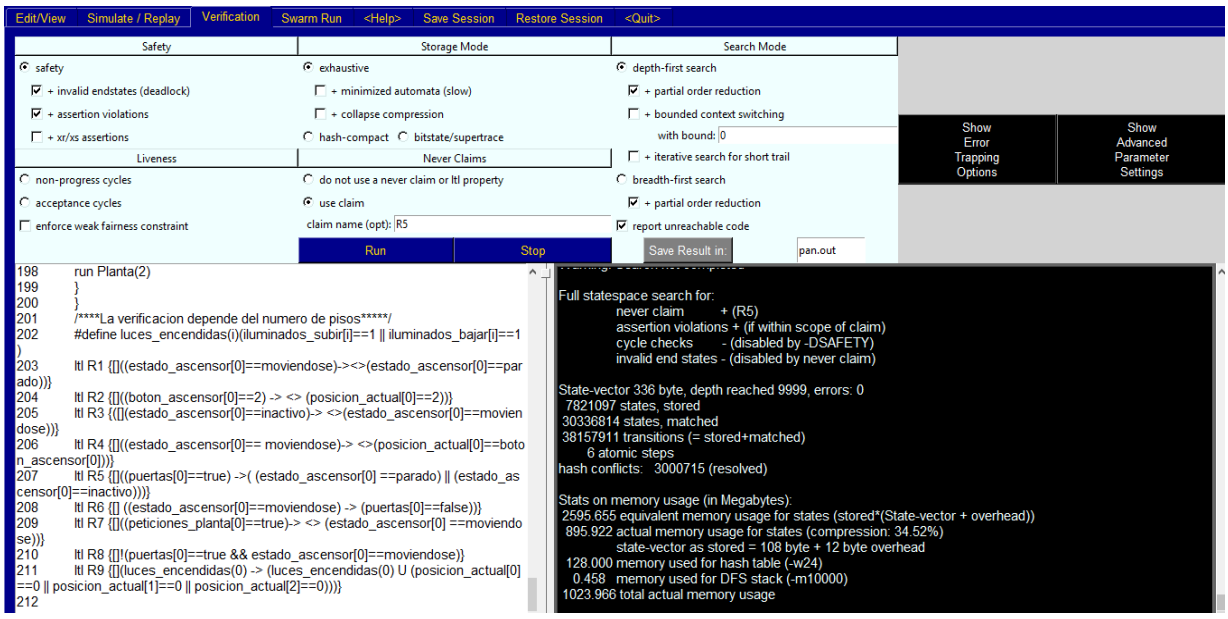

Figura 52: Captura del resultado de la verificación de una propiedad de seguridad.

Todas las propiedades definidas para la verificación de nuestro SCA se muestran en la Tabla 5. En esta tabla se hace referencia al tipo de propiedad, la propiedad escrita en lenguaje de lógica temporal, y si ha sido o no satisfecha.

N	Propiedades	Propiedades LTL	Resultado
1	Viveza	{[](estado_ascensor[0]==moviendose)-><>(estado_ascensor[0]==parado)}	Satisfecha
2	Viveza	{[](boton_ascensor[id]==planta)->[](<> (posicion_actual[id]==planta))}	Satisfecha
3	Viveza	([](estado_ascensor[0]==inactivo)-> <>(estado_ascensor[0]==moviendose))}	Satisfecha
4	Viveza	{[](estado_ascensor[id]==moviendose)-> <>(posicion_actual[id]==boton_ascensor[id])}	Satisfecha
5	Seguridad	{[](puertas[id]==true)->((estado_ascensor[id] ==parado) || (estado_ascensor[id]==inactivo))))}	Satisfecha
6	Seguridad	{[](estado_ascensor[id]==moviendose)-> (puertas[id]==false))}	Satisfecha
7	Viveza	{[](peticiones_planta[id]==true)-><> (estado_ascensor[id] ==moviendose))}	Satisfecha
8	Seguridad	{[]!(puertas[id]==true&& estado_ascensor[id]==moviendose)}	Satisfecha
9	-	{[](luces_encendidas(id) -> (luces_encendidas(id) U (posicion_actual[0]==id || posicion_actual[1]==id || posicion_actual[2]==id))))}	Satisfecha

Tabla 5: Propiedades LTL y el resultado de su verificación.

5 Modelado y verificación con Uppaal

5.1 Introducción a Uppaal

UPPAAL es una herramienta usada para la simulación (mediante autómatas temporizados) y verificación (mediante comprobación automática de modelos) de sistemas en tiempo real. La herramienta ha sido desarrollada en colaboración del grupo de Diseño y Análisis de Sistemas en Tiempo Real de la Universidad de Uppsala, Suecia y el grupo de Investigación Básica en Informática de la Universidad de Aalborg, Dinamarca [21].

La herramienta consta de dos partes principales: la interfaz gráfica de usuario y el motor verificador de modelos.

La idea es modelar un sistema utilizando autómatas temporizados, simularlo y luego verificar las propiedades. Los autómatas temporizados son máquinas de estados finitos extendidos con relojes que marcan el tiempo real. Un sistema consiste en una red de autómatas que se componen de “locations” o estados, y transiciones entre estas localizaciones que definen cómo se comporta el sistema.

La simulación ejecuta el sistema de manera interactiva para comprobar que funciona según lo esperado. Una vez que se ha simulado el sistema, se puede pedir al verificador que analice propiedades de alcanzabilidad, es decir, si un determinado estado es alcanzable o no. También pueden analizarse propiedades descritas en la lógica TCTL (una versión con tiempo de la lógica “computational tree logic” CTL). El motor de verificación trabaja básicamente realizando una búsqueda exhaustiva que cubre todos los comportamientos dinámicos posibles del sistema. Para ello, utiliza abstracciones del tiempo basadas, por ejemplo, en intervalos y poliedros.

Como se ha comentado arriba, Uppaal se basa en autómatas temporizados, es decir, en máquinas de estados finitos con relojes. Los relojes son la forma de manejar el tiempo en Uppaal. El tiempo es continuo y los relojes miden el progreso del tiempo. El tiempo avanzará globalmente al mismo ritmo para todo el sistema. Un sistema en Uppaal se compone de procesos concurrentes, cada uno de ellos modelado como un autómata. El autómata tiene un conjunto de estados (locations).

Las transiciones entre estados tienen guardas. Una guarda es una condición sobre las variables y los relojes que describe cuando la transición está habilitada.

El mecanismo de sincronización en Uppaal se implementa a través de canales síncronos. Por ejemplo, para que dos procesos transiten simultáneamente, uno debe tener una operación de escritura sobre un canal del tipo “canal!” y el otro una operación

de lectura sobre el mismo canal, es decir, “canal?”. Las transiciones permiten realizar dos posibles acciones: asignación de variables y/o reinicio de relojes.

5.2 Verificación del sistema con Uppaal

La ventana principal de Uppaal (Figura 53) tiene dos partes principales: el menú y las pestañas.

Las tres pestañas dan acceso a los tres componentes de Uppaal que son el editor, el simulador y el verificador. La figura 53 muestra la vista del editor. El uso de las plantillas facilita la implementación de sistemas a con varios procesos con comportamiento muy parecido. Las plantillas pueden tener símbolos, variables y constantes como parámetros. Una plantilla también puede tener variables locales y relojes.

Nuestro pequeño ejemplo en Uppaal consta de 4 procesos que se muestran en la Figura 54:

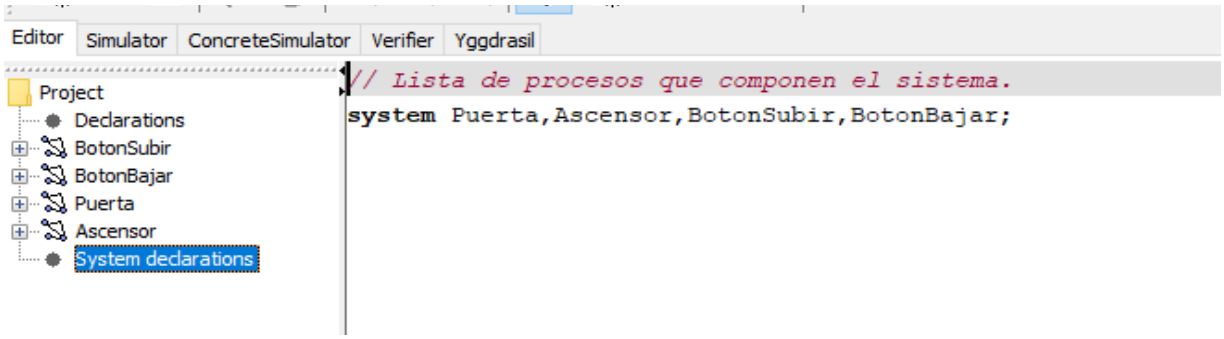

Figura 53: Procesos que componen el sistema de ascensores en Uppaal.

Además, se han declarado las siguientes variables y canales:

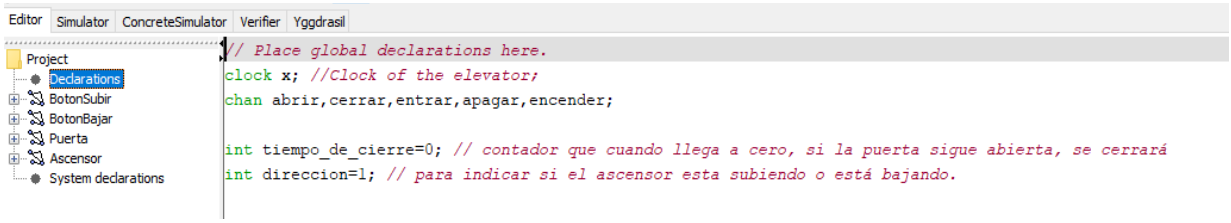

Figura 54: Declaración de variables globales del sistema de ascensores en Uppaal.

En la figura 55 podemos observar que la plantilla del proceso ascensor consta de estado inicial (con un doble círculo) que hace referencia al estado en el que el ascensor llega a una planta. El resto de los “locations” hacen referencia a los estados “Subiendo, Bajando y Parado”.

Las expresiones azules son asignaciones a variables, que se ejecutan cuando se toma la transición correspondiente. Las expresiones en verde son guardas que tienen que ser verdaderas para que la transición correspondiente se habilite.

La variable x es un reloj. La asignación a la variable “x” establece que cuando el ascensor pasa del estado “Planta” al estado “Parado” la variable tomará el valor 2.

Las guardas “x>=2” establecen que se pasará del estado “Bajando” o “Subiendo” al estado planta cuando hayan transcurrido 2 o más segundos.

La guarda “dirección==0” establece que el ascensor pasará al estado “Bajando”. Alternativamente, la guarda “dirección==1” indica que el ascensor debe transitar al estado “Subiendo”.

Los canales que se usan en el proceso ascensor son “abrir!”, que indica cuando el proceso “Puerta” deberá abrir la puerta del ascensor, “cerrar?” que recibe del proceso “Puerta” cuando ésta debe cerrarse y “entrar?” que le envía el proceso “Puerta” siempre que tiempo de cierre aún no ha llegado a 0 y, por lo que, aún pueden entrar pasajeros en el interior del ascensor.

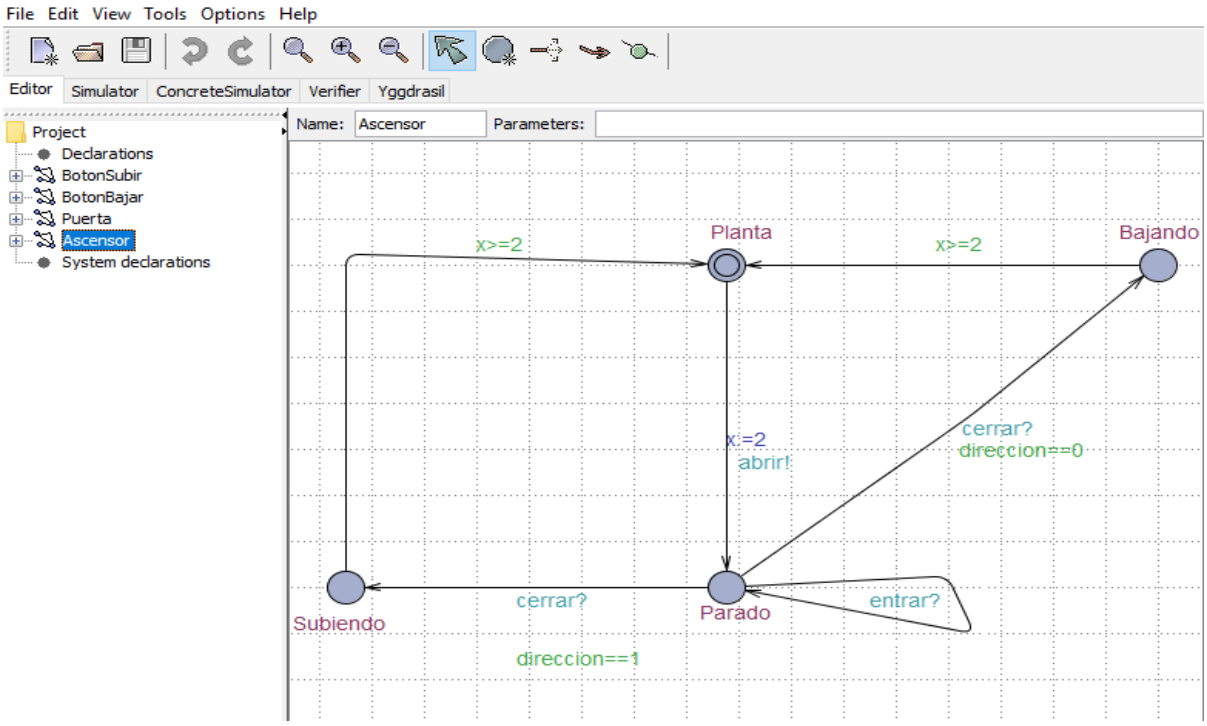

Figura 55: Plantilla Ascensor en Uppaal.

El proceso puerta (Figura 56) consta de 3 estados el inicial es “PuertaCerrada”, el siguiente es “Parado” que indica cuando el ascensor pasa al estado parado, y el siguiente es “PuertaAbierta”. La puerta volverá a este estado si el tiempo de cierre no ha llegado a 0.

En el estado inicial de este proceso, se inicia la variable de tipo entero “tiempo_de_cierre” al valor 2. Si se recibe el mensaje “abrir?” del proceso “Ascensor” y la guarda “x>=2” es cierta se pasará al estado “Parado”.

Mientras se cumpla la guarda “x<=5” (es decir, el tiempo no supere el valor 5) se puede transitar al estado “PuertaAbierta”, en el que si la guarda “tiempo_de_cierre>0” se cumple y se recibe, además, el mensaje “entrar!”, la puerta transitar de nuevo al estado “PuertaAbierta”. El valor de la variable “tiempo_de_cierre se irá disminuyendo en 1.

Las guardas “x>=4 && (tiempo_de_cierre)==0” habilitan la transición del estado “PuertaAbierta” a “PuertaCerrada”. Cuando la transición es posible, la variable “x” tomará el valor 0 y la dirección cambiará.

Hay que tener en cuenta que, en este caso de estudio, se ha definido este cambio de direcciones para simular el comportamiento de un ascensor y dar una visión global de cómo usar esta herramienta.

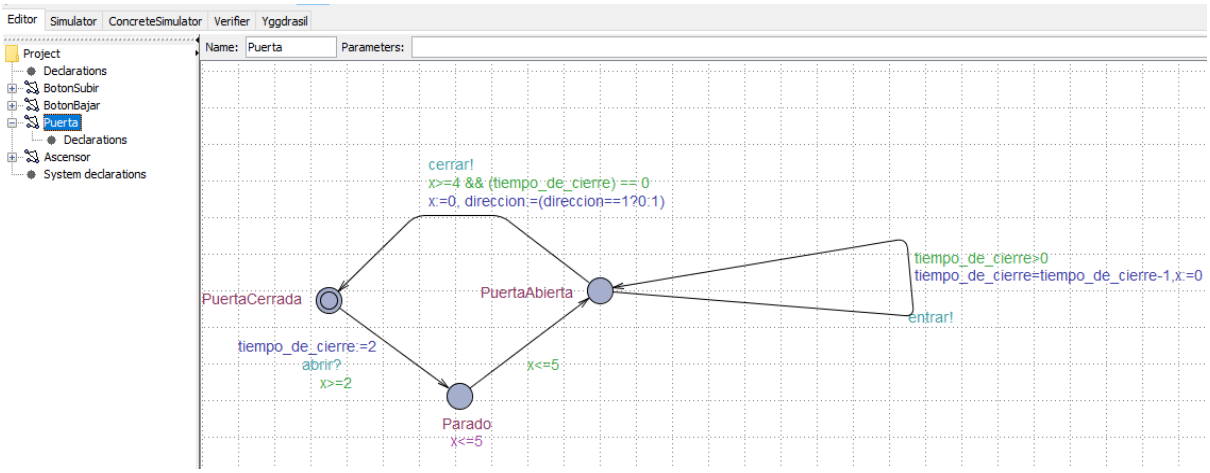

Figura 56: Plantilla Puerta en Uppaal.

Los procesos BotonSubir (Figura 57) y BotonBajar (Figura 58) constan de 2 estados “Encendido” y “Apagado”. La única diferencia entre ambos es son las transiciones para cambiar de estado que, lógicamente, dependen de la dirección que tomen.

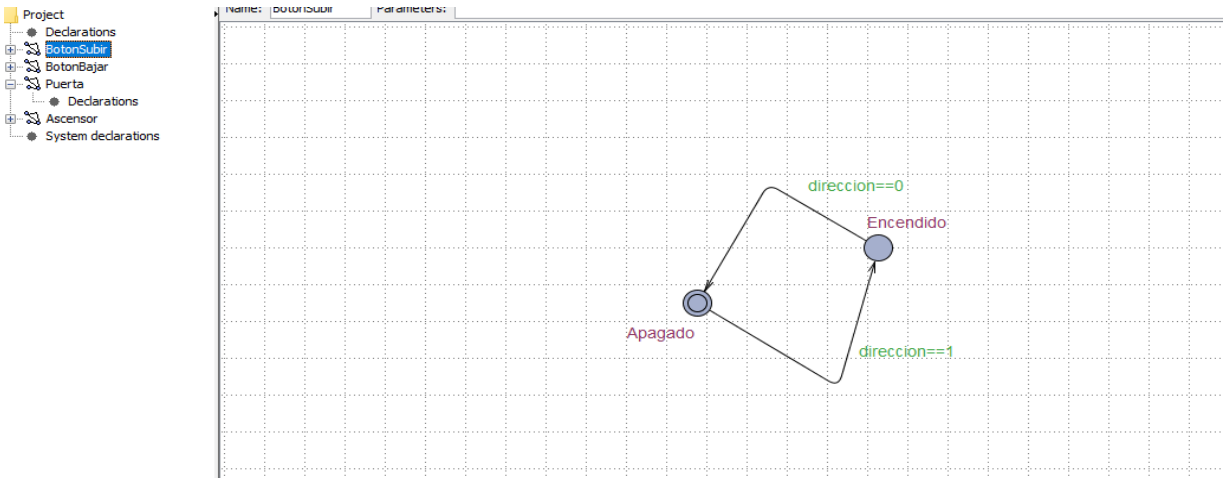

Figura 57: Proceso BotonSubir en Uppaal.

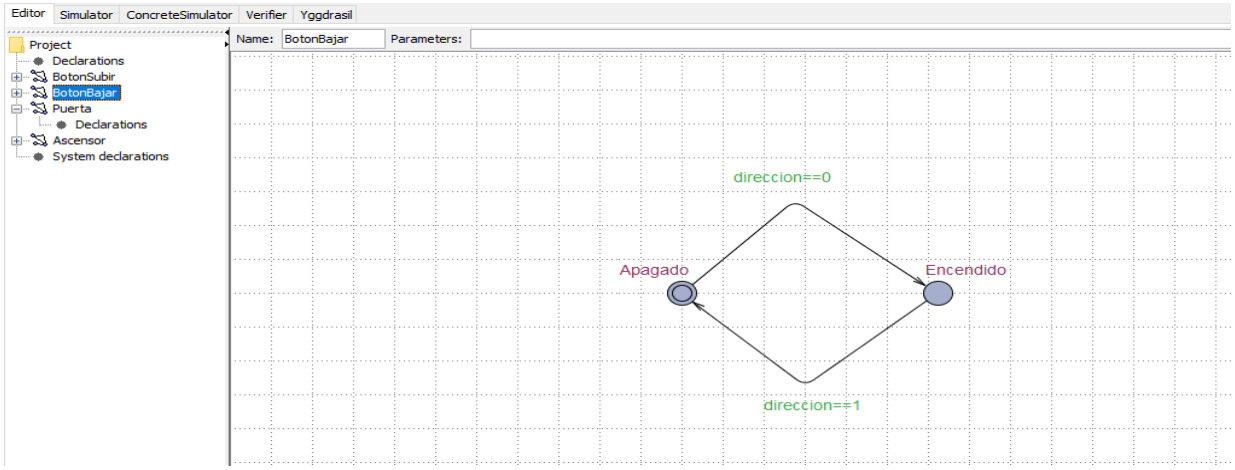

Figura 58: Proceso Boton Bajar en Uppaal.

Una vez modelado el sistema, se puede hacer clic en la pestaña Simulador para iniciar la simulación de nuestro sistema de ascensores.

La Figura 59 muestra una vista del simulador. A la izquierda encontramos la parte de control donde podemos elegir las transiciones (parte superior) y trabaje en una traza existente (parte inferior). En el panel intermedio se encuentran las variables del sistema, y a la derecha se ve el sistema en sí. Debajo del sistema, puede verse que sucede en cada proceso.

Durante la simulación, el estado que se muestra en rojo indica la ubicación actual de cada autómata, la simulación evolucionará cambiando estos estados. Si, en un momento dado, no es posible realizar ninguna transición es porque el sistema está bloqueado. En la figura 60, podemos observar la comunicación entre procesos. Una vez simulado el sistema procederemos a la verificación.

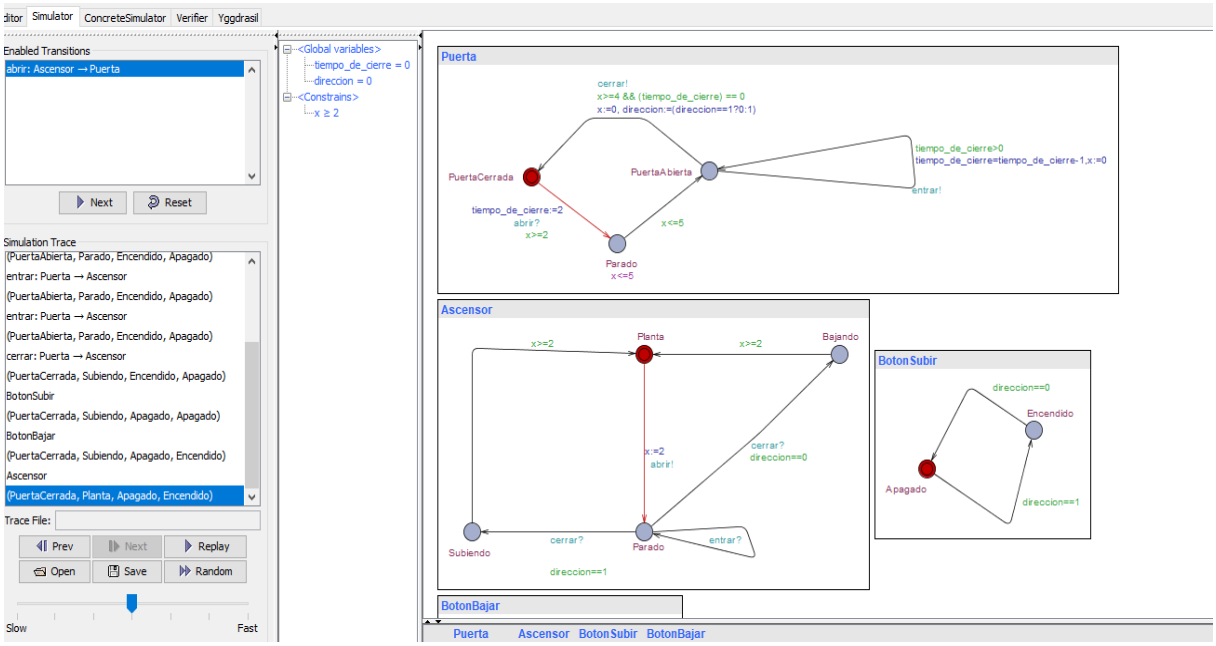

Figura 59: Captura del simulador gráfico de Uppaal.

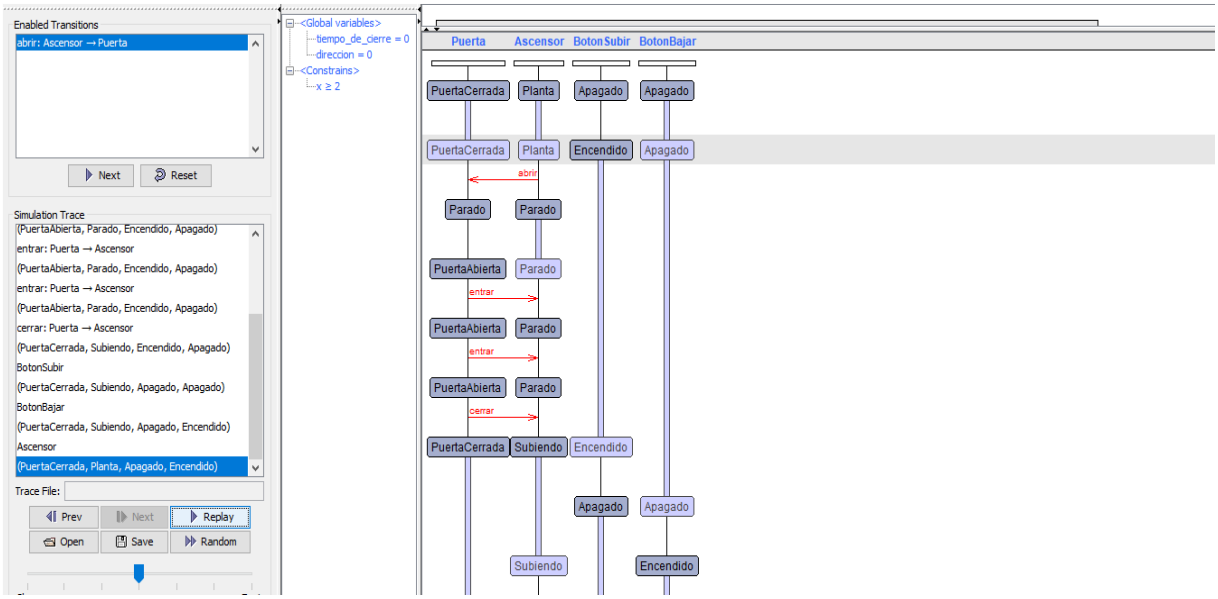

Figura 60: Comunicación entre procesos simulador gráfico de Uppaal.

Para verificar el sistema debemos pulsar la pestaña “Verifier”. Para analizar si el sistema satisface las propiedades deseables, primero debemos escribir en el campo Query la propiedad TCTL correspondiente a la propiedad y pulsar check.

Uppaal utiliza una versión simplificada de la lógica TCTL. Al igual que en TCTL, el lenguaje de consulta consiste en fórmulas de camino y fórmulas de estado. Las fórmulas de estado describen propiedades de estados individuales, mientras que las fórmulas de camino se refieren a trazas o trayectoria posibles del modelo. Las fórmulas de camino se pueden clasificar en alcanzabilidad, seguridad y viveza.

Una fórmula de estado es una expresión que utiliza operadores temporales, por lo que puede evaluarse sobre unos estados específicos. En Uppaal, el interbloqueo se expresa mediante una fórmula de estado especial (aunque esto no es estrictamente una fórmula de estado). La fórmula consiste simplemente en la palabra clave “deadlock y se cumple para todos los estados en los que no es posible ejecutar ninguna transición.

Las propiedades de alcanzabilidad son las más simples. Preguntan si una fórmula de estado dada puede ser satisfecha por algún estado alcanzable. Estas propiedades no garantizan por sí mismas la corrección del modelo, pero sirven para comprobar el comportamiento básico del modelo.

Por ejemplo, para expresar que se puede alcanzar algún estado que satisface ϕ se utiliza fórmula TCLT E <> ϕ.

La propiedad de seguridad sirve para expresar que "algo malo nunca va a ocurrir". Una variación de esta propiedad es que "Algo nunca sucederá". En Uppaal, estas propiedades se formulan positivamente, por ejemplo, algo bueno, es invariablemente cierto. Expresamos que ϕ debe ser verdad en todos los estados alcanzables con las fórmulas de camino A[]ϕ, mientras que E[] ϕ dice que debe existir un camino en el que ϕ es siempre cierta.

Las propiedades de viveza son expresan propiedades de progreso, son del tipo “algo sucederá eventualmente”. Por ejemplo, en un modelo de un protocolo de comunicación, cualquier mensaje que se haya enviado debe ser eventualmente recibido. En su forma simple, la viveza se expresa con la fórmula de ruta A<>ϕ, lo que significa que finalmente se cumple [22].

En la herramienta Uppaal, cuando las propiedades se satisfacen sale un círculo en verde, como se puede ver en la Figura 61. En otro caso, es decir, si no son satisfechas este círculo se mostrará en rojo. Cuando la verificación no es concluyente con la aproximación utilizada las propiedades están marcadas en amarillo.

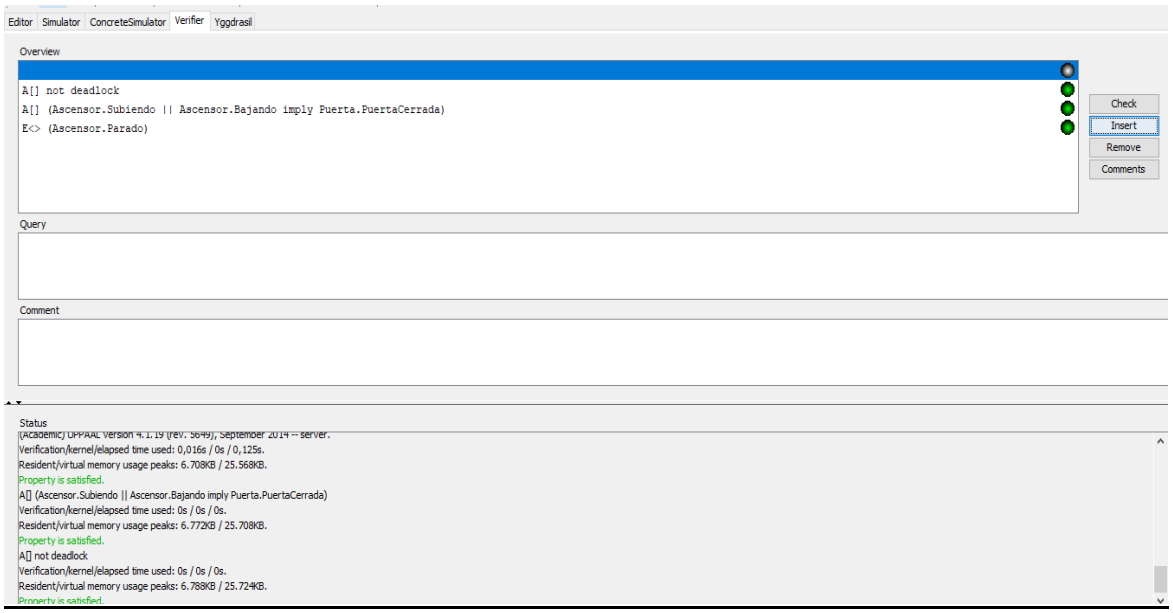

Figura 61: Ejemplo de verificación de propiedades del sistema ascensor.

6 CONCLUSIONES

El principal objetivo de este proyecto es describir algunos métodos formales que permiten el modelado, simulación y verificación de distintos aspectos de los sistemas complejos como, por ejemplo, el modelo de un sistema de ascensores que ha sido el caso de estudio utilizado en el proyecto.

Como se ha podido observar durante todo el desarrollo de ese trabajo, un sistema de ascensores, que a primera vista puede parecer un sistema simple, no lo es en absoluto. Es un sistema con una estructura de clases y objetos que se relacionan de manera no trivial, y con un comportamiento concurrente que puede dar lugar a errores en propiedades básicas de seguridad y viveza.

En el proyecto, se han utilizado distintos métodos formales. Por un lado, se ha demostrado que el uso de OCL permite, sin duda, descripciones más fieles de los modelos UML. Las restricciones sobre el modelo hacen posible una especificación mucho más precisa. Sin embargo, aunque en las herramientas se puede simular la evolución de un sistema UML, en la actualidad, los mecanismos proporcionados son un poco rígidos.

Por otro lado, Alloy, SPIN y Uppaal han permitido el modelado del sistema de ascensores visto como un sistema distribuido. Cada herramienta se ha centrado sobre un aspecto distinto del sistema. Así, el uso conjunto de todos los métodos formales, han permitido analizar con éxito las principales propiedades críticas del sistema, incluyendo los aspectos de tiempo real.

Alloy ha permitido visualizar, proyectando el modelo sobre la signatura Tiempo, el movimiento del ascensor de planta a planta y observar con mejor claridad cómo se actualizan las luces y las peticiones. Además, al poder simular y visualizar fácilmente el comportamiento (incluyendo los contraejemplos que la herramienta nos proporciona), es posible encontrar errores con mucha más facilidad que en las otras herramientas.

Una desventaja de usar Alloy es que, aunque nos proporcione instancias correctas del sistema, no tenemos la certeza absoluta de que el sistema funciona correctamente, ya que la generación de las instancias está acotada por el tamaño. Respecto al diseño una de las características a resaltar de Uppaal que es la posibilidad de localizaciones hacen posible un modelado natural del comportamiento de transmisión necesario en este caso de estudio.

En la verificación con SPIN se puede comparar el aumento de estados generados y alcanzados con un aumento en los procesos y además es útil para la detección de errores en la secuencia de mensajes.

SPIN, Alloy y Uppaal se utilizan para verificar varias propiedades importantes del modelo de ascensores. Sobre la base de la comparación de los resultados, encontramos que SPIN es más adecuado para verificar sistemas distribuidos, mientras que Alloy y USE para analizar propiedades estructurales de los sistemas y Uppaal es más apropiado para verificar sistemas en tiempo real.

7 REFERENCIAS

[1] E. M. Clarke, Jr., O. Grumberg, D. A. Peled, Model Checking, MIT Press, Cambridge, USA, 1999.

[2] C. Baier, J.-P. Katoen, Principles of Model Checking (Representation and Mind Series), The MIT Press, 2008.

[3] E. M. Clarke, O. Grumberg, D. E. Long, Model checking and abstraction, ACM Trans. Program. Lang. Syst. 16 (5) (1994) 15121542.

[4] http://www.escuelapedia.com/historia-del-ascensor/

[5] https://ascensoresdomingo.com/blog/partes-de-un-ascensor

[6]D. VÍCTOR BLANCO BLÁZQUEZ, DRA. MARÍA JESÚS LÓPEZ BOADA
TUTOR SCHINDLER, D. JOSE MARÍA MENÉNDEZ ARTIRIA,
MODERNIZACIÓN DE UNA INSTALACIÓN EXISTENTE DE ASCENSORES

[7] José Ignacio Aguayo Padilla, Dr. Manuel Aguilar Cornejo, Verificación Formal de Sistemas, México D. F. 2006

[8] Grady Booch, James Rumbaugh and Ivar Jacobson. The Unified Modeling Language User Guide.

[9] Marina Egea González, Una semántica formal ejecutable para ocl con aplicaciones al análisis y a la validación de modelos, Madrid 2008

[10] Guillermo González Calderón, ESPECIFICACIÓN FORMAL EN OCL DE REGLAS DE CONSISTENCIA ENTRE EL MODELO DE CLASES Y DE CASOS DE USO DE UML, marzo de 2007

[11] Valeria Becker y Claudia Pons. Semántica de OCL. Reporte interno del proyecto Lifia Eclipse. http://sol.info.unlp.edu.ar/~eclipse

[12] Oriente Cantos, Joaquín, SOPORTE OCL EN MOMENT, UNA HERRAMIENTA DE GESTIÓN DE MODELOS, Septiembre 2006, Valencia

[13]Edward Yue Shung Wong, Michael Herrmann, Omar Tayeb, A Guide To Alloy Second Year Group Project.

[14] Daniel Jackson, Alloy: A Lightweight Object Modeling Notation" ACM Transactions on Software Engineering and Methodology (TOSEM), Volume 11, Issue 2 (April 2002), pp. 256-290.

[15]Daniel Jackson, “Software Abstractions: Logic, Language and Analysis”. MIT Press, 2012.

[16]K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. UML2Alloy: A challenging model transformation. In G. Engels, B. Opdyke, D. C. Schmidt, and F. Weil, editors, Models, volume 4735 of LNCS, pages 436–450. Springer, 2007.

[17]Ma. Laura Cobo, Métodos Formales para Ingeniería de Software, Departamento de Ciencias e Ingeniería de la Computación Universidad Nacional del Sur Argentina.

[18]https://informatica.cv.uma.es/pluginfile.php/260533/mod_resource/content/1/MC-Modelado-uft8.pdf

[19] Renato Neves, Alexandre Madeira, Manuel Martins, Luís Barbosa. “An Institution for Alloy and Its Translation to Second-Order Logic”. In Integration of Reusable Systems.

[20] ARMIN BIERE , KEIJO HELJANKO , TOMMI JUNTTILA , TIMO LATVALA , AND VIKTOR SCHUPPAN, LINEAR ENCODINGS OF BOUNDED LTL MODEL CHECKING.

[21] https://en.wikipedia.org/wiki/Uppaal_Model_Checker

[22] Gerd Behrmann, Alexandre David, and Kim G. Larsen, A Tutorial on Uppaal Updated, Aalborg University, Denmark, 17th November 2004