Universidad Nacional Autónoma de México

Facultad de Ciencias

Métodos formales ligeros: especificación de un sistema de elevadores en el analizador Alloy

T E S I S

QUE PARA OBTENER EL TÍTULO DE:

LICENCIADO EN CIENCIAS DE LA COMPUTACIÓN

PRESENTA:

RODRIGO JIMÉNEZ DEL VALLE

DIRECTOR DE TESIS:

DR. FAVIO EZEQUIEL MIRANDA PEREA

2011

1.Datos del alumno	1. Datos del alumno
Apellido paterno	Jiménez
Apellido materno	Del Valle
Nombre(s)	Rodrigo
Teléfono	53 36 24 69
Universidad	Universidad Nacional Autónoma de México
Facultad	Facultad de Ciencias
Carrera	Ciencias de la Computación
Número de cuenta	300104389
2. Datos del tutor	2. Datos del tutor
Grado	Dr
Nombre(s)	Favio Ezequiel
Apellido paterno	Miranda
Apellido materno	Perea
3. Datos del sinodal 1	3. Datos del sinodal 1
Grado	M en C
Nombre(s)	Miguel
Apellido paterno	Carrillo
Apellido materno	Barajas
4. Datos del sinodal 2	4. Datos del sinodal 2
Grado	M en C
Nombre(s)	Araceli Liliana
Apellido paterno	Reyes
Apellido materno	Cabello
5. Datos del sinodal 3	5. Datos del sinodal 3
Grado	Dr
Nombre(s)	Francisco
Apellido paterno	Hernández
Apellido materno	Quiroz
6. Datos del sinodal 4	6. Datos del sinodal 4
Grado	M en C
Nombre(s)	Lourdes del Carmen
Apellido paterno	González
Apellido materno	Huesca

7.Datos del trabajo escrito.	7. Datos del trabajo escrito
Título	Métodos formales ligeros:especificación de un sistema de elevadores en el analizador Alloy
Número de páginas	122 p
Año	2011

Índice general

Bibliografía **117**

1: Introducción

Programar es una tarea que puede ser fácil o complicada dependiendo de cuál sea la abstracción que se piense construir, si elegimos una abstracción adecuada la programación será una tarea más simple que le permitirá al modelo de interés crecer fácilmente y será más fácil de entender. Si por el contrario, elegimos una abstracción inadecuada estaremos expuestos a sufrir fallas, y el tratar de corregirlas resultará igual o más complicado que el problema que queríamos resolver.

El problema con el que nos enfrentamos a la hora de realizar estas abstracciones (que al hacerlas parecen simples, robustas e intuitivas) es que al implementarlas, el código puede crecer bastante y con ello la dificultad del problema, apareciendo inconsistencias que hacen que nuestro problema no sea tan obvio como esperábamos, entonces, el código usado no es suficiente para realizar una simple abstracción. Partiendo de lo anterior Jackson dice que : *"Una abstracción no es un módulo, una interfaz, una clase o un método, es una estructura pura y simple, una idea reducida a su forma esencial."* (Jackson 2006)

La mayoría de las veces que programamos, partimos de un problema el cual queremos resolver, si tenemos conocimientos de varios lenguajes de programación elegimos el paradigma que creemos más adecuado para resolverlo en base a experiencias previas que hayamos tenido. En general el pseudocódigo que generamos podemos utilizarlo para traducirlo en el lenguaje de programación adecuado, sin embargo, en el proceso de construcción muchas veces nos encontramos con problemas que no consideramos y vamos corrigiendo estos al mismo tiempo que realizamos el programa. Más aún, muchas veces la idea que pensamos originalmente no es correcta o al menos no completamente, por lo que siempre surgen nuevos problemas que no consideramos originalmente y a la hora de tratar de resolverlos no sabemos con certeza la fiabilidad y seguridad de lo que hicimos. Esta incertidumbre nos deja grandes dudas sobre la robustez de la implementación realizada, por lo tanto, necesitamos una manera de probar que lo que hicimos es correcto considerando todos los casos posibles, pero probar esto generalmente es una tarea que se vuelve muy complicada sino es que imposible en algunos casos.

Para resolver este problema Jackson propone el enfoque de *la especificación formal* (Jackson 2006), que es una descripción matemática del software que se puede usar para realizar una implementación, buscando que esta sea una tarea precisa y sin ambigüedades, diciéndonos lo que un sistema tiene que hacer mas no cómo se debe hacer.

La *especificación formal* usa una notación simple y expresiva que posee una serie de conceptos robustos. Nos ofrece una descripción detallada de lo que tenemos que hacer, y una manera de

demostrar la consistencia de una implementación gracias a la seguridad que ofrece la formalidad matemática que tiene. El problema de esto es que, a diferencia de los demostradores de teoremas, sólo se pueden comprobar un número finito de casos, pero aún así consta de un gran número de casos que se pueden analizar. Para conseguirlo no se requieren casos de prueba sino que se da una propiedad la cual será verificada.

1.1. Métodos formales

Los métodos formales son una serie de metodologías que surgen como una respuesta a las inquietudes planteadas, pues su formalidad nos permite especificar un problema mediante lógica y matemáticas, esto hace que el desarrollo sea más confiable permitiéndonos verificar un sistema de software. Se caracterizan por hacer un análisis más preciso, fiable y formal de algún diseño y son usados cuando el grado de seguridad implica un costo muy alto, por ejemplo en una expedición espacial en la que un error de diseño de software podría significar una gran perdida de recursos.

Los métodos formales tienen una base bien cimentada en fundamentos teóricos de las ciencias de la computación, la lógica, la teoría de autómatas, los lenguajes formales, etc. Utilizarlos es una tarea difícil, costosa y muchas veces no se sabe cómo se pueden aplicar a problemas reales de manera adecuada ya que en la mayoría de los casos no hay una interacción suficientemente estrecha entre los investigadores y las empresas. Sin embargo, el uso de los métodos formales propicia la confiabilidad y la seguridad de un sistema al aumentar su comprensión revelando inconsistencias, ambigüedades o carencias que de otra manera pasan inadvertidas.

Muchas veces los métodos formales lo son en exceso y el modelo es muy complejo y abstracto, lo que hace que sólo un puñado de personas sean capaces de desarrollarlos, y en consecuencia, son muy costosos para aquellas empresas o industrias interesadas en ellos. Los métodos formales son requeridos en casos concretos como en un sistema de líneas de ferrocarril, (el Metro), o en un sistema aeronáutico, en el que alguna falla podría ser de consecuencias mortales.

El inicio de los métodos formales se remonta a los años 70, sin embargo en los últimos veinte años su campo de aplicación ha crecido ampliamente, pasando de ser usados sólo en círculos académicos a su aplicación en problemas reales en industrias y empresas tecnológicamente importantes. El crecimiento generalizado de software de calidad implicó que el uso de los métodos formales fuera un asunto de alta prioridad, sin embargo su uso no es generalizado esencialmente por dos razones: es una tarea que pocos pueden desarrollar y por lo tanto, su costo de producción es muy alto debido a la falta de conocimiento acerca de su existencia.

Como ejemplo tenemos a VDM (*Vienna Development Method*), que es uno de los métodos formales más conocidos y reconocidos, el cual incluye un grupo de técnicas y herramientas basadas en

la especificación formal. Este ha sido probado en una gran variedad de aplicaciones en compiladores, plantas industriales, en sistemas utilizados por la NASA y en muchos otros sistemas, véase (VDM).

Una alternativa a la dificultad asociada a estos métodos son los llamados *métodos formales ligeros* como por ejemplo el analizador *Alloy*, gracias a que no se necesita un gran conocimiento previo en métodos formales o fundamentos matemáticos complicados.

Los métodos formales ligeros, en contraste con los que no lo son, ofrecen una alternativa más amistosa ya que no se enfatiza tanto en la formalidad de una especificación, sino que se enfocan en la capacidad expresiva y simple del lenguaje que utilizan, además de que se sirven de una especificación parcial de un modelo. Su uso representa un aumento importante en cuestiones de fiabilidad, a pesar de lo que cuesta desarrollarlos.

El analizador Alloy ha sido usado para modelar y analizar muchos tipos de sistemas como configuraciones de protocolos de red, control de acceso[1], criptografía, entre otros. IBM lo utilizó para verificar métodos de Java contra especificaciones y en muchos otros casos. El analizador Alloy ofrece ventajas sobre los *verificadores de modelos* para analizar sistemas indeterminados o que pueden cambiar drásticamente, además de que puede verificar propiedades que no siempre pueden ser verificadas por un verificador de modelos. Un gran beneficio de utilizar este analizador es que se puede verificar un modelo a partir de pocas líneas de código y hacerlo crecer. (Alloy)

Utilizar el analizador Alloy no nos permitirá probar un sistema, más bien nos permitirá refutarlo encontrando un contraejemplo, esto será de gran ayuda cuando la complejidad del problema es muy grande y queremos verificar ciertas propiedades del modelo para un universo más pequeño, en el cual podremos ver si el modelo será válido.

[1] Por ejemplo, las llaves de habitación de un hotel.

1.2. Descripción del trabajo

El objetivo de este trabajo es realizar una especificación la cual nos muestre el comportamiento de un elevador a través del tiempo, así como la interacción de este con varios usuarios que realizan distintas peticiones, esto quiere decir que el sistema es variable con respecto al número de usuarios, pisos y el tiempo en que le toma al elevador terminar una trayectoria, permitiéndonos así ver distintos ejemplos del funcionamiento del elevador.

Para realizar esta especificación, utilizamos una serie de reglas u operaciones declarativas que nos dicen cómo se moverá el elevador con los usuarios a través de los pisos, para realizarlo se utilizó el enfoque de los métodos formales ligeros, en particular el analizador Alloy. Con el cual construiremos una especificación que simule el sistema para varios escenarios.

A partir de la especificación podremos ver una serie de instancias que genera el analizador y mediante estas haremos el análisis correspondiente que nos permita darnos cuenta de la calidad del sistema realizado. Es pertinente señalar que, hasta donde conocemos, los resultados aquí presentados son originales.

2: Preliminares

2.1. El analizador Alloy

El analizador Alloy es una herramienta que genera micromodelos de software que son utilizados para analizar especificaciones propuestas por el usuario, describiendo las estructuras con nociones matemáticas mínimas y generando instancias de un modelo, pudiendo así verificar parcialmente las propiedades que el usuario proporcione, ya que sólo se puede mostrar la validez del modelo para un número pequeño de objetos que intervienen en el modelo, partiendo de la hipótesis del alcance o ámbito pequeño[1] que nos dice que, un gran número de errores pueden encontrarse probando el programa en un ámbito pequeño es decir, en un modelo con un número pequeño de elementos básicos.

El analizador Alloy está fuertemente influenciado por la notación del modelado de objetos, un usuario parte de una idea, la abstracción de algún problema que generalmente esta ligado con la realidad, como el modelado de una agenda electrónica, el manejo de llaves electrónicas en un hotel o la selección de procesos líderes en un sistema operativo.

Un modelo se construye traduciendo restricciones escritas en el analizador Alloy expresadas con lógica relacional a lógica booleana que serán resueltas por un *solucionador del problema SAT*[2], los pasos a seguir son: construir un modelo con el analizador, simularlo y verificarlo; de tal forma que los modelos se desarrollan de manera incremental permitiendo que el usuario agregue más restricciones o que las quite por no considerarlas importantes o que sean contradictorias dándose cuenta así de los errores de la especificación propuesta.

El analizador consta de una herramienta gráfica que permite visualizar una instancia o *simulación* del modelo propuesto, dándole al usuario no sólo la opción de visualizar su modelo sino también le permite, durante el proceso, incorporar nuevas restricciones que vaya considerando necesarias para enriquecer más el modelo.

El analizador Alloy fue desarrollado por Daniel Jackson y su equipo del *Massachusetts Institute of Technology*, la versión actual es la 4 que puede descargarse: `http://alloy.mit.edu/alloy4/`.

[1] Small scope.
[2] *SAT Solver*.

El analizador consta de 3 elementos clave que son:

Lógica: Toda estructura está representada por una relación y sus propiedades se expresan con una serie de operadores, la ejecución de estos operadores está determinada por las restricciones que decide el usuario, permitiendo al modelo crecer al agregar o quitar más restricciones.

Lenguaje: Consta de una sintaxis flexible y fácil de usar, que nos permite hacer una descripción de un modelo, así como reutilizar las estructuras que se realicen.

Análisis: Consiste en encontrar una serie de instancias que satisfagan una propiedad dada, en caso contrario, el analizador nos mostrará contraejemplos implicando que la propiedad es falsa, esto permite hacer un mejor análisis a las restricciones de nuestro modelo.

Toda búsqueda de instancias tiene asociada un alcance que está ligada al número de objetos que el usuario especifique. Este análisis se puede facilitar al visualizar gráficamente las instancias o simulaciones de salida para un modelo dado gracias a la herramienta gráfica con la que cuenta el analizador.

2.2. Especificación en lógica relacional

La especificación en lógica relacional es un núcleo que nos provee de conceptos elementales que permiten una interacción simple y expresiva con la abstracción que queremos realizar. Este núcleo combina la lógica de primer orden con el cálculo relacional.

En esta sección explicaremos como vamos a utilizar estos conceptos para realizar una abstracción, traduciéndolos al lenguaje del analizador Alloy. También explicaremos la sintaxis, los componentes y la manera en la que trabaja el analizador.

Para explicar todo esto haremos uso del ejemplo de una agenda electrónica, véase [Daniel Jackson, 2006], en el cual se mapean nombres de personas o alias a sus respectivas direcciones. Este ejemplo consta de tres tipos de signaturas *Name*, *Addr* y *Book* las cuales definen los elementos que habrá en la agenda, estos elementos los podemos asociar por medio de relaciones entre los objetos, como por ejemplo las relaciones *names* que asocia nombres con libros(agendas) y *addrs* que asocia libros con nombres y direcciones. A lo largo de esta sección se explicará la interacción de estos elementos por medio de ejemplos, se irán agregando relaciones nuevas y veremos como funcionan los operadores existentes en el analizador Alloy con los elementos de la agenda, permitiendonos no sólo ver la interacción de los componentes sino también el crecimento de un modelo pequeño.

Lógica

Una relación es una estructura que relaciona átomos, entidades indivisibles, inmutables y sin interpretación. Todo es una relación ya sean escalares y conjuntos.

Conjuntos: Son relaciones unarias

Name = {(N0), (N1), (N2), (N3)}

Escalares : Son conjuntos con un solo elemento

Name = {(N0)}
Book = {(B1)}

Relaciones binarias:

names = {(B0, N0), (B0, N1), (B1, N2)}

Relaciones ternarias:

addrs = {(B0, N0, A0), (B0, N1, A1), (B1, N1, A2), (B1, N2, A2)}

Constantes:

none - conjunto vacío
univ - conjunto universal
iden – relación identidad

Dado un modelo con *Name* y *Addr* como sigue:

Name = {(N0), (N1), (N2)}
Addr = {(A0), (A1)}

tenemos que:

none = {}
univ = {(N0), (N1), (N2),(A0), (A1)}
iden = {(N0, N0), (N1, N1), (N2, N2), (A0, A0), (A1, A1)}

Operadores de conjuntos

unión: $p + q = \{t \mid t \in p \vee t \in q\}$
diferencia: $p - q = \{t \mid t \in p \wedge t \notin q\}$

intersección:	p & q = {t | t | p ∧ t ∈ q}
subconjunto:	p **in** q = {(p1 ... pn) ∈ p} ⊆ {(q1 ... qm) ∈ q}
igualdad:	(p = q) = {(p1 ... pn) ∈ p} = {(q1 ... qn) ∈ q}

Ejemplos:

Juan = {(N0)}
Pedro = {(N1)}
Juan + Pedro = {(N0), (N1)}

Juan = Pedro = **false**
Pedro **in** none = **false**

Name = {(N0), (N1), (N2)}
Alias = {(N1), (N2)}
Group = {(N0)}
RecentlyUsed = {(N0), (N2)}
Alias + Group = {(N0), (N1), (N2)}

Alias & RecentlyUsed = {(N2)}
Name – RecentlyUsed = {(N1)}
RecentlyUsed **in** Alias = **false**
RecentlyUsed **in** Nombre = **true**

Operadores relacionales

Operador flecha: p -> q = {(p1 ... pn, q1 ... qm) | (p1 ... pn) ∈ p ∧ (q1 ... qm) ∈ q}

El operador flecha te permite obtener la relación que toma cada combinación de una tupla de *p* y una de *q* y las concatena. Este operador trabaja de manera similar al producto cartesiano en teoría de conjuntos, la diferencia radica no sólo en su utilización con conjuntos, sino también con átomos y escalares.

p -> q	cuando *p* y *q* son conjuntos *p* -> *q* es un producto cartesiano.
r: p -> q	nos dice que *r* mapea átomos que están en *p* con átomos en *q*.
p -> q	es una tupla cuando *p* y *q* son escalares.

Tomando en cuenta el siguiente modelo:

Name = {(N0), (N1)}
Addr = {(A0), (A1)}
Book = {(B0)}

tenemos que:

Name -> Addr = {(N0, A0), (N0, A1), (N1, A0), (N1, A1)}
Book -> Name -> Addr = {(B0, N0, A0), (B0, N0, A1), (B0, N1, A0), (B0, N1, A1)}

b = {(B0)}
b′ = {(B1)}
address = {(N0, A0), (N1, A1)}
address′ = {(N2, D2)}
b -> b′ = {(B0, B1)}
b -> address + b′ -> address′ = {(B0, N0, A0), (B0, N1,A1), (B1, N2, A2)}

Operador punto o join: p.q {(p1 ... pn-1, q2 ... qm) | (p1 ... pn) ∈ p ∧ (pn,q2 ... qm) ∈ q}

El operador *join* o de composición nos permite combinar tuplas, tomamos el último átomo de una tupla y lo comparamos con el primero de otra tupla, si estos coinciden la tupla resultante de su composición sera aquella que comience con los átomos de la primera tupla y termine con los átomos de las segunda omitiendo el átomo que coincida.

Tomando el siguiente modelo:
Book = {B0, B1} Name = {N0, N1, N2} host = {(A0, H0), (A1, H1), (A2, H1)}
Addr = {A0, A1, A2} Host = {H0, H1}
addr = {(B0, N0, A0), (B0, N1, A0), (B1, N2, A2), (B1, N1, A1)}

tenemos que:

Book.addr = {(N0, A0), (N1, A0), (N2, A2), (N1, A1)}
B0.addr = {(N0, A0), (N1, A0)}
Addr.host = {H0, H1}
A1.host = {H1}
addr.host = {(B0, N0, H0), (B0, N1, H0), (B1, N2, H1), (B1, N1, H1)}

Otros operadores relacionales de importancia son:

Inversa: ~r
Cerradura transitiva: ^r = r + r.r + r.r.r + ...
Cerradura reflexiva y transitiva: *r = ^r + **iden**
Constantes: **univ**, **none**, **iden**

Ejemplos:

Node = {(N0), (N1), (N2), (N3)}
next = {(N0, N1), (N1, N2), (N2, N3)}
~next = {(N1, N0), (N2, N1), (N3, N2)}
^next = {(N0, N1), (N0, N2), (N0, N3), (N1, N2), (N1, N3), (N2, N3)}
*next = {(N0, N0), (N0, N1), (N0, N2), (N0, N3), (N1,N1), (N1, N2), (N1, N3), (N2, N2),
(N2, N3), (N3, N3)}

Utilizando los operadores relacionales y los unarios podemos definir una serie de propiedades básicas que son:

r es una relación reflexiva: **iden in** r
r es una relación simétrica: r **in** r
r es una relación transitiva: r.r **in** r
r es antirreflexiva: **iden** & r = **none**
r es funcional r.r **in iden**
r es acíclica: ^r & **iden** = **none**

Operadores lógicos

Existen dos maneras de escribir los operadores lógicos:

Negación:	**not**	!
Conjunción:	**and**	&&
Disyunción:	**or**	||
Implicación:	**implies**	=>
Alternativa:	**else**	,
Equivalencia:	**iff**	<=>

Cuantificadores

Un cuantificador toma la forma *C x: e | F* en donde *F* es una restricción que contiene a la variable *x*, *e* es una expresión que liga a *x* en F y *C* es un cuantificador. Los cuantificadores se escriben de la siguiente manera:

Universal: **all** x: e|F
Existencial: **some** x: e|F, **some** e, #e > 0
Negación de existencial: **no** x: e|F, **no** e, #e = 0
Existencia de a lo más uno: **lone** x: e|F, **lone** e, #e =< 1
Existencia de un único: **one** x: e|F, **one** e, #e = 1

Ejemplos:

some n: Name, a: Address | a **in** n.address
Algunos nombres se mapean a algunas direcciones.

no n: Name | n **in** n.^address
No existen ciclos en una agenda.

all n: Name | **lone** d: Address | d **in** n.address
Todos los nombre se mapean al menos a una dirección.

all n: Name | **no disj** d, d′: Address | d + d′ **in** n.address
Para todo nombre no existe un par distinto de direcciones que esten asociadas a él.

Relación de multiplicidad

Si la expresión ligada es construida con el operador flecha, la multiplicidad puede escribirse de la siguiente manera:

r: A m -> n B

En donde A, B son conjuntos y m, n son palabras reservadas que representan la multiplicidad, r es la relación que mapea cada miembro de A a cada miembro de B.

Ejemplos:

r: A -> **one** B
Una función cuyo dominio es A.

r: A **one** -> B
Una relación inyectiva cuyo rango es B.

r: A -> **lone** B
Una función parcial sobre un dominio A.

r: A **one** -> **one** B
Una función inyectiva cuyo dominio es A y su rango es B, también llamada biyectiva.

r: A **some** -> **some** B
Una relación con dominio A y rango B.

Multiplicidad

La declaración *x: e* tiene casi el mismo significado que *x in e*, pero puede contener restricciones de multiplicidad, por lo tanto se escribirá *x: m e*, en donde m es el tamaño del conjunto.

Para las multiplicidades utilizamos las palabras reservadas:

set = cualquier número
one = exactamente uno
lone = cero o uno
some = uno o más

Ejemplos:

RecentlyUsed: **set** Name
RecentlyUsed es un subconjunto del conjunto *Name*.

senderAddress: Addr
senderAddress es un único escalar en el conjunto *Addr*.

senderName: **lone** Name
senderName es un subconjunto que es vacío o sólo tiene un escalar de *Name*.

receiverAddresses: **some** Addr
receiverAddresses es un subconjunto no vacío de *Addr*.

Declaración de relaciones

workAddress: Alias -> **lone** Addr
Cada alias referencia a lo más a una dirección (work address)

homeAddress: Alias -> **one** Addr
Cada alias referencia a exactamente una dirección (home address)

members: Alias **lone** -> **some** Addr
Cada alias esta asociado con a lo más una direccion y cada dirección tiene asociada al menos un alias.

Expresiones cuantificadas

some e, e tiene al menos una tupla.
no e, e no tiene tuplas.
lone e, e tiene a lo mas una tupla.
one e, e tiene exactamente una tupla.

some Name - el conjunto de nombres es no vacío.
no (address.Addr - Name) - nada se puede mapear a una dirección excepto nombres.
all n: Name | **lone** n.address - todo nombre se mapea a lo mas a una dirección.

Expresion Let

La expresión *let* nos permite definir expresiones complicadas o que se repiten de manera regular:

let x = e | A

En donde la variable x reemplaza a la expresión e en el cuerpo de A.

La siguiente expresión nos dice que para todos los nombre en la agenda, si al menos un nombre está ligado con la dirección de un trabajo, entonces ese nombre esta ligado a una dirección que es alguna de un trabajo y en caso de que no sea una dirección de trabajo será una dirección de casa.

all n: Name |
(**some** n.workAddress **implies** n.address = n.workAddress
else n.address = n.homeAddress)

Como vemos en la expresión anterior tenemos elementos que se repiten, al agregar una expresión *let* podremos factorizar esta expresión a una equivalente que puede escribirse de distintas maneras pero con el mismo significado, estas son:

all n:Name |
let w = n.workAddress, a = n.address |
(**some** w **implies** a = w **else** a = n.homeAddress)

all n: Name |
let w = n.workAddress | n.address = (**some** w **implies** w **else** n.homeAddress)

all n: Name |
n.address = (**let** w = n.workAddress | (**some** w **implies** w **else** n.homeAddress))

Cardinales

Existe el operador # que aplicado a una relación nos dice el número de tuplas que contiene esta. Existe también una serie de operadores para comparar a los números enteros, los cuales son:

+	más.
-	menos.
=	igual.
<	menor que.
>	mayor que.
<=	menor o igual a.
>=	mayor o igual a.

Lenguaje

El lenguaje nos permite describir los elementos que interactúan en la construcción de una abstracción; construyendo modelos distintos que pueden ir creciendo y ser usados más de una vez.Los elementos del lenguaje de manera resumida son:

Signaturas y campos

- Introduce un conjunto de átomos y relaciones.
- Clasificación jerárquica y de subtipos.

Restricciones

- **fact**: Hechos que se asume son siempre ciertos.
- **pred**: Predicados que son restricciones encapsuladas.
- **fun**: Funciones que devuelven un resultado.
- **assert**: Verificación de afirmaciones.

Comandos

- **run**: Genera instancias de un predicado.
- **check**: Verifica contraejemplos de afirmaciones.

Signaturas

sig A{}
Conjunto de átomos A

sig A{}
sig B{}
Conjuntos separados A y B (**no** A & B)
sig A, B{}
Conjunto de átomos A y B

sig B **extends** A{}
El conjunto B es un subconjunto de A (B **in** A)

sig B extends A{}
sig C extends A{}
B y C son subconjuntos separados de A ((B **in** A) && (C **in** A) && (**no** B & C))

abstract sig A{}

Una signatura abstracta no tiene elementos excepto por aquellos que extienden de esta.

sig B **extends** A{}
sig C **extends** A{}

A es un conjunto partido en conjuntos separados B y C (**no**(B & C) && A = (B + C))

sig B **in** A{}

B es un subconjunto de A, no necesariamente separado de cualquier otro conjunto

sig C **in** A + B{}

C es un subconjunto de la unión de A y B

one sig A{}: A es un conjunto con un único elemento.
lone sig B{}: B es un conjunto con un único elemento o con ningún elemento.
some sig C{}: C es un conjunto no vacío.

Campos

sig A{f: e}

Las relaciones son declaradas como campos de signaturas en donde f es una relación binaria con dominio A y un rango dado por la expresión e, f está obligada a ser una función (*f: A -> **one** e*) ó (***all** a: A | a.f: e*).

Ejemplos:

```
sig Book{
    names: set Name,
    addrs: names -> Addr
}
```

Todo libro o agenda tiene un conjunto de nombres los cuales tiene asociadas direcciones para cada nombre.

```
abstract sig Person{
    father: lone Man,
    mother: lone Woman
}
```

Las personas tienen a lo más un padre y una madre.

```
sig Man extends Person{
    wife: lone Woman
}
```

Las esposas son mujeres y todo hombre tiene a lo más una.

```
sig Woman extends Person{
    husband: lone Man
}
```

Los esposos son hombres y toda mujer tiene al menos uno.

Hechos

Son restricciones que se asume que siempre se cumplen.

```
fact{ F }
fact f{ F }
```

Ejemplos:

```
fact{all x: Link | x.from != x.to}
```

No existe una liga que vaya de un *host* hacia él mismo.

```
fact{
        no p: Person |
        p in p.^(mother + father)
        wife =  husband
}
```

Ninguna personas es su propio ancestro y la esposa de un hombre tiene como esposo a ese hombre al igual que el esposo de una mujer tiene como esposa a esa mujer.

Funciones

Las funciones son expresiones las cuales tienen un nombre, en ellas declaramos parámetros de entrada al igual que declaramos una expresión que será el resultado, este lo obtenemos de la expresión que invoca los parámetros de entrada y se encuentra dentro de la función.

```
fun f[x1: e1, ..., xn: en]: e{ E }
```

Ejemplos:

```
fun lookup[b: Book, n: Name]: set Addr{
    b.addr[n]
}

fun grandpas[p: Person]: set Person{
    p.(mother + father).father
}
```

La función *lookup* recibe dos parámetros un libro y un nombre, y está nos dice que el resultado de una búsqueda es cualquier conjunto de direcciones en la que el nombre n se mapea a una addr que mapea el libro b.

La función *grandpas* recibe un atributo persona p y el resultado es el conjunto de personas que son abuelos de p al mapear este con los padres de el padre de p, que son los abuelos de p.

Predicados

Son fórmulas que tiene un nombre y en las cuales declaramos parámetros de entrada, y representan una restricción.

```
pred p[x1: e1, ..., xn: en]{ F }
```

Ejemplos:

```
pred contains(b:Book, n:Name, d:Addr){
    n -> d in b.addr
}

pred ownGrandpa(p: Person){
    p in grandpas[p]
}
```

El predicado *contains* nos dice que dado un libro b, un nombre n y una dirección d, si n esta asociado a una dirección está estará en el conjunto que mapea al libro b con las direcciones.

El predicado *ownGrandpa* nos dice que dada una persona p, está sera o no su propio abuelo si p esta en el conjunto de abuelos.

Aserciones

Son restricciones que intencionalmente deben seguirse de los hechos del modelo. El analizador verifica estas aserciones, si una aserción no se sigue de los hechos puede haber un error, por ejemplo si al realizar un modelo encontramos algún error, por medio de las aserciones podemos ver que este puede ser detectado. Las aserciones son una parte muy importante de Alloy, ya que con estas podemos darnos cuenta de errores en nuestro modelo permitiendonos hacer un mejor análisis de lo que estamos haciendo.

assert A{ F } En donde A es el nombre de la aserción y F es la expresión a verificar.

Algunos ejemplos tomando en cuenta el siguiente modelo:

```
sig Node{
    children: set Node
}

one sig Root extends Node{}

fact{
    Node in Root.*children
}
```

aserción inválida:

```
assert someParent{
    all n: Node | some children.n
    }
}
```

Es inválida porque estamos diciendo que todo los nodos son hijos de alguien, y esto no es cierto por el hecho, que dice que existe uno que es *Root* y es padre de todo nodo.

aserción valida:

```
assert someParent{
    all n: Node - Root | some children.n
}
```

Es valida porque decimos que todo nodo menos el nodo *Root* es hijo de alguien.

Análisis y verificación

El comando *check* nos permite verificar una aserción en un alcance determinado por el usuario. Se le pide al analizador que busque algún contraejemplo a la aserción propuesta.

assert a{ F }
check a *alcance*

Si un modelo tiene H hechos y H es la conjunción de los hechos del modelo, el analizador intentará encontrar la solución a la expresión a *H && !F*.
Ejemplos:

abstract sig Person{}
sig Man **extends** Person{}
sig Woman **extends** Person{}
sig Grandpa **extends** Man{}

check a
check a **for** 4
check a **for** 4 **but** 3 Woman
check a **for** 4 **but** 3 Man, 5 Woman
check a **for** 4 Person
check a **for** 4 Person, 3 Woman
check a **for** 3 Man, 4 Woman
check a **for** 3 Man, 4 Woman, 2 Grandpa

Si no escribimos algún alcance, por defecto el analizador verifica las aserciones a lo más para 3 elementos de cada signatura que exista en el modelo. En los ejemplos anteriores podemos ver como se utilizan los comandos *check* y *for*, donde se especifica que se verificará para x o menos elementos. El comando *but* verifica exactamente para x elementos, mientras que *run* busca una instancia de un predicado dentro de un cierto alcance.

pred p(x: X, y: Y, ...){ F }
run p *alcance*

Si un modelo tiene H hechos y H es la conjunción de los hechos del modelo, el analizador intentará encontrar una solución a la expresión *H && (**some** x: X, y: Y, ... | F)*.

Ejemplo:

```
fun grandpas(p: Person): set Person{
    p.(mother + father).father
}

pred ownGrandpa(p: Person){
    p in grandpas[p]
```

```
}

run ownGrandpa for 4 Person
```

En el ejemplo anterior verificamos si para un alcance de 4 personas existirá una instancia en la que una persona sea su propio abuelo.

2.3. Ejemplos

Esta sección contiene dos ejemplos, cada uno de ellos ilustra un tipo de aplicación diferente:

- El primero consta de una representación estática basada en un conjunto de reglas aplicadas a las líneas del metro de la ciudad de Londres, véase [Jackson, 2006].
- El segundo trata sobre la elección de un proceso líder bajo una topología de anillo. Para este ejemplo se introduce la técnica de la *traza* para construir un modelo dinámico, véase [Jackson, 2006]

Para poder explicar mejor los predicados largos o complejos, los numeraremos y examinaremos línea por línea todo el bloque de código.

2.3.1. Modelo estático

En este ejemplo vamos a construir algunas reglas sobre las líneas de ferrocarril, y después aplicar estas reglas al metro de la ciudad de Londres.

En la figura 2.1 se muestra el mapa simplificado de la ciudad de Londres[3]. Para este modelo sólo se tomaron en cuenta tres lineas: Jubilee que corre de norte a sur desde Stanmore hasta Waterloo; Central que corre del oeste al este West Ruislip y Ealing Brodway hasta Epping; y Circle que corre con sentido a la derecha a través de Baker Street.

Vamos a modelar todas las estaciones de una línea de trenes como un conjunto de estaciones y con una relación binaria sobre éstas el recorrido de una a otra:

```
abstract sig Station{
    jubilee, central, circle: set Station
}
```

Declaramos las signaturas correspondientes a las lineas del metro:

```
sig Jubilee, Central, Circle in Station{}
```

[3] Se puede encontrar el real en *http://tube.tfl.gov.uk/*.

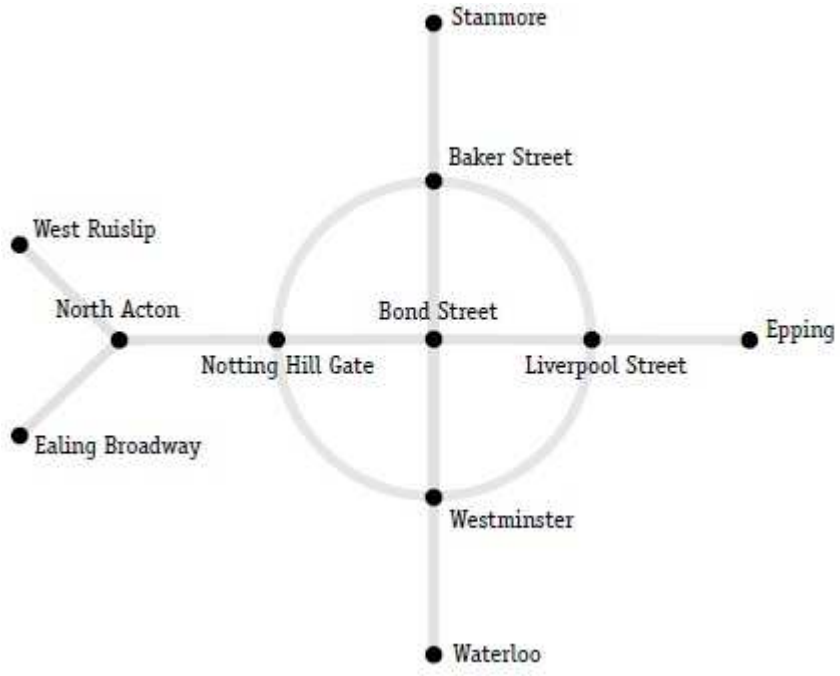

Figura 2.1: Diagrama simplificado del metro de Londres.

Y las estaciones de estas líneas:

```
one sig
    Stanmore, BakerStreet, BondStreet, Westminster, Waterloo,
    WestRuislip, EalingBroadway, NorthActon, NottingHillgate,
        LiverpoolStreet, Epping
    extends Station{}
```

A continuación especificamos a que línea o líneas pertenece cada estación:

```
fact Init{
    Jubilee = Stanmore + BakerStreet + BondStreet + Westminster + Waterloo
    Central = WestRuislip + EalingBroadway + NorthActon + NottingHillGate + BondStreet +
        LiverpoolStreet + Epping
    Circle = BakerStreet + NottingHillGate + Westminster + LiverpoolStreet
}
```

Ahora vamos a declarar las reglas para construir las líneas del metro. La línea *Circle* debe formar de un círculo:

```
1  pred getCircle(){
2
```

```
3       circle in Circle -> one Circle
4
5       all x: Circle | one y: Circle{
6           x -> y in circle
7           y -> x not in circle
8           one circle.x
9       }
10
11  }
```

En la línea 3 (de ahora en adelante nos referiremos al número de la línea como: *(número)*) decimos que todas la estaciones de *Circle* están relacionadas con una única estación de *Circle*. El tren viaja de la estación S a la estación S' (6) pero no de S' a S (7). Todas las estaciones tienen un único destino (8).
La línea *Jubilee* forma una línea recta:

```
1   pred getStraightLine(){
2       jubilee in Jubilee -> lone Jubilee
3       one x: Jubilee | no x.jubilee and x = Waterloo
4       one x: Jubilee | Jubilee in x.*jubilee and x = Stanmore
5   }
```

Todas las estaciones de *Jubilee* están asociadas a lo más con una estación de *Jubilee* (2). Existe una estación que no tiene asociado un destino, y esa estación es Waterloo (3). Existe una estación a la que no se puede llegar desde otra estación y que a partir de ella se puede llegar al resto de las estaciones de Jubilee y esa estación es Stanmore (4).

La línea *Central* forma una línea recta hasta que se bifurca en dos rectas en la estación S:

```
1   pred getStraightLineBranches(){
2
3       central in Central -> lone Central
4
5       one x: Central | no x.central and x = Epping
6
7       one x: Central | all y: Central - central.x{
8           #central.x = 2
9           y in x.*central
10          central.x = EalingBroadway + WestRuislip
11      }
12
```

```
13 }
```

Todas las estaciones de *Central* están asociadas a lo más con una estación de *Central* (3). Existe una estación que no tiene asociado un destino, y esa estación es Epping (5). Existe una estación S en *Central* en donde se puede llegar desde dos estaciones diferentes (8). Todas las estaciones son accesibles desde la estación S (9), excepto EalingBroadway y WestRuislip quienes son las ramificaciones en la estación S (10).

Ahora definimos un hecho que obligue al analizador Alloy a generar todas las instancias de nuestra especificación a partir de nuestros tres predicados:

```
fact Structure{
    getCircle[]
    getStraightLine[]
    getStraightLineBranches[]
}
```

Para que las instancias de nuestro modelo queden lo más parecido al diagrama de la figura 2.1, vamos a acotar la línea *Circle* agregando las siguientes reglas a nuestro hecho *Structure*:
Limitamos a *Circle* por el norte de *Jubilee*:

```
one x: Jubilee | no jubilee.x and one x.jubilee & Circle
```

Limitamos a *Circle* por el sur de *Jubilee*:

```
one x: Jubilee | no x.jubilee and one jubilee.x & Circle
```

Limitamos a *Circle* por el este de *Central*:

```
one x: Central | no x.central and one central.x & Circle
```

Limitamos a *Circle* por el oeste de *Central*:

```
one x: Central | #central.x = 2 and one x.central & Circle
```

Definimos a la estación que se encuentra en el centro de *Circle*:

```
one x: Central & Jubilee |
    one jubilee.x & Circle and one x.jubilee & Circle and one central.x & Circle and
        one x.central & Circle
```

Definimos el orden en el que se intersectan *Central* y *Jubilee* con *Circle*:

```
all x: Circle | x in Jubilee implies x.circle in Central else x.circle in Jubilee
```

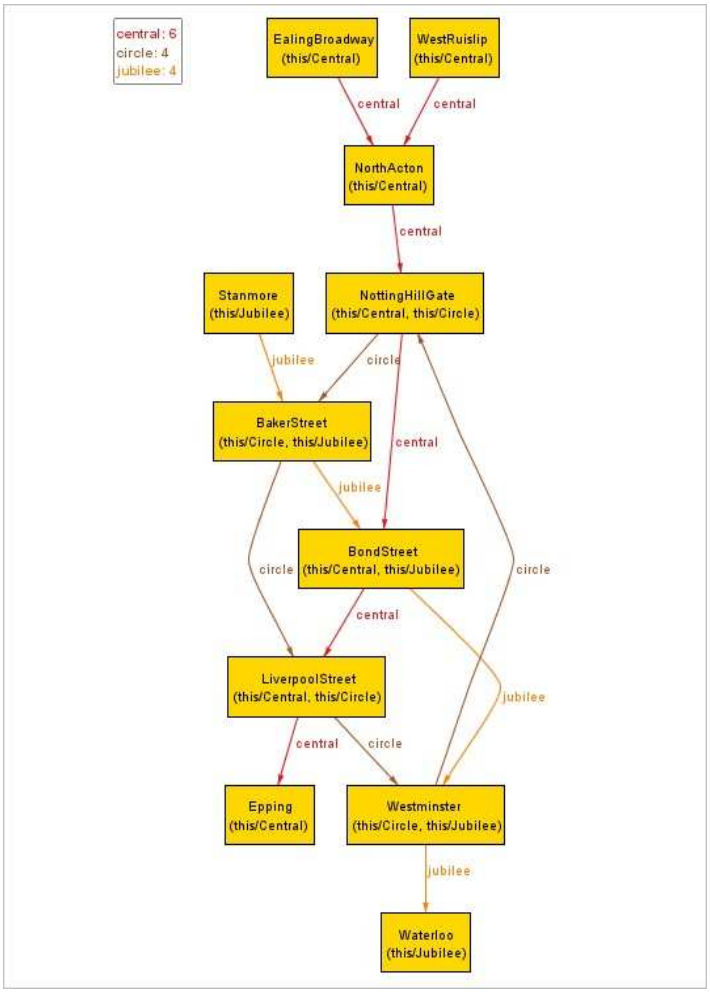

Figura 2.2: Instancia generada por el analizador Alloy correspondiente a la figura 2.1.

Por último, ejecutamos nuestro modelo para obtener una simulación:

pred show()
run show

En la figura 2.2 se muestra una simulación del diagrama simplificado del metro de Londres. Pero con este tipo de especificación no nos es suficiente para modelar todas las caracteristcas deseadas para un problema en particular, a veces necesitaremos definir un conjunto de estados ordenados y relacionados entre si en donde podamos analizar el comportamiento de cada estado al realizar una transición. El siguiente ejemplo se enfoca en este caso en particular, en donde para cada instancia generada de un modelo se definen un conjunto de estados denominado como *traza*.

2.3.2. Modelo dinámico

En este ejemplo se hace uso de una técnica que permite construir un modelo en el cual se pueda analizar *dinámicamente* todas las situaciones que puedan surgir. A esta técnica se le conoce como *traza*.

Una traza nos permite ordenar los elementos de una signatura y desplazarnos ordenadamente entre estos. La figura 2.3 nos muestra el comportamiento de una traza, en donde las relaciones *next* y *prev* estan dadas por las operaciones *T/next[tn]* ó *tn.next* y *T/prev[tn]* ó *tn.prev* respectivamente.

```
open util/ordering[Time] as T
```

```
Time = t0 + t1 + t2 + t3 + t4
```

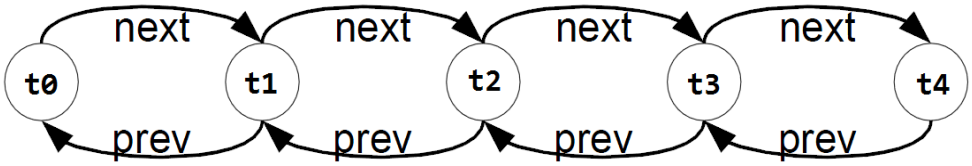

Figura 2.3: Funcionamiento de una traza.

El objetivo de una traza no es el de considerar el resultado de una instancia a otra de manera individual, sino que consiste en analizar toda la traza de cada una de las instancias generadas por el analizador Alloy. Cada traza consta de un conjunto de pasos de un estado a otro relacionados entre si a partir de un *estado inicial*, en donde cada estado define un compotamiento del modelo. Para comenzar a utilizar una traza, se modelará el problema de la elección de un proceso líder bajo una topología de anillo, véase [Jackson, 2006]. Para este ejemplo se considera el caso en donde un conjunto de procesos forman un anillo, todo lo que se necesita es ordenar los elementos de una entidad, limitando a que el primer elemento de esa entidad satisface una serie de condiciones iniciales, y el resto de los elementos ordenados se van relacionando por una operación, que no necesariamente tiene que ser la misma.

Primero hay que importar la biblioteca que permite implementar un orden total aplicándolo a las signaturas que representarán a los intervalos de tiempo y a los procesos en el anillo respectivamente:

open util/ordering[Time] as TO
open util/ordering[Process] as PO

Declaramos la signatura que representará a los estado del modelo como momentos en el tiempo:

```
sig Time{}
```

Y por último, la signatura que representa a los procesos junto con sus propiedades:

```
1  sig Process{
2      succ: Process,
3      toSend: Ptocess -> Time,
4      elected: set Time
5  }
```

Cada proceso tiene un proceso sucesor (su vecino dentro del anillo) (2), un conjunto de identificadores de proceso para ser enviados a lo largo de todo el anillo (3), y una serie de tiempos en la que se considera elegido a un proceso como el líder (4).

La relación *succ* asegura que cada proceso tiene exactamente un sucesor y como se esta trabajando en una topología de anillo se debe agregar una regla que permita que los procesos puedan ser accesibles desde cualquier otro siguiendo *succ* varias veces:

```
fact Ring{all p: Process | Process in p.^succ}
```

Ahora se necesitan definir las operaciones con las cuales se relacionarán los elementos de la traza. Primero se establece la condición inicial, donde se especifica que todos los procesos sólo pueden enviar su propio identificador:

```
pred init(t: Time){
    all p: Process | p.toSend.t = p
}
```

Segundo, se describe la transición de un estado a otro. En un tiempo dado se elige un identificador arbitrario (*id*) del conjunto de identificadores asociados a procesos (*from*) y se mueve al conjunto asociado con su respectivo sucesor (*to*):

```
pred step(t, t': Time, p: Process){
    let from = p.toSend, to = p.succ.toSend |
        some id: from.t{
            from.t' = from.t - id
            to.t' = to.t + (id - PO/prevs[p.succ])
        }
}
```

La expresión *id – P0/prevs[p.succ]* elimina del conjunto que contiene el identificador *id* al conjunto de todos los identificadores que preceden a *p.succ*.

Tercero, se describe la designación del proceso elegido. En el primer momento en el tiempo, ningún proceso ha sido elegido, para los otros momentos en el tiempo el conjunto de procesos elegidos es el conjunto de procesos que acaban de recibir sus propios identificadores:

```
fact DefineElected{
    no elected.TO/first[]
    all t: Time – TO/first[] |
        elected.t = p: Process | p in p.toSend.t - p.toSend.(TO/prev[t])
}
```

Ejecución de una traza

Existen dos propiedades que se desean verificar: que a lo más un proceso es elegido como el líder y que eventualmente se elegirá a un líder. Agregando un hecho que permita el desplazamiento de un estado a otro mediante un conjunto de operaciones, de las cuales, sólo una es elegida **arbitrariamente** para realizar la transición, se esta creando una traza para cada instancia del modelo generada por el analizador Alloy. Con este hecho se pueden formular varias aserciones para comprobar el resultado de las propiedades que se quieren verificar. Si una aserción es inválida, el analizador Alloy genera un contraejemplo con al menos una traza que muestra como se violan las reglas.
Ahora, se crea la traza:

```
fact Traces{
    init [TO/first []]
    all t: Time - TO/last[] | let t' = TO/next [t] |
        all p: Process |
            step[t, t', p] or step[t, t', succ.p] or skip[t, t', p]
}
```

La traza nos dice que el estado inicial se satisface para el primer momento en el tiempo (*TO/first []*) y que para cualquier momento en el tiempo posterior cada proceso p da un paso[4] o su predecesor da un paso, o bien, este proceso no hace nada. Donde *no hacer nada* se modela con el predicado *skip*:

```
pred skip (t, t': Time, p: Process){
    p.toSend.t = p.toSend.t'
}
```

[4] Dar un paso se refiere a la operación *step*.

Análisis dinámico

Se comienza generando una instancia muy simple del modelo, en donde se pregunta por una ejecución en donde se ha elegido algún proceso:

```
pred show(){
    some elected
}
run show for 3 Process, 4 Time
```

El alcance del comando *run* considera 3 procesos y 4 instantes de tiempo para generar instancias o simulaciones del modelo. La traza generada por el analizador Alloy se muestra en la figura 2.4, en donde el identificador del proceso *P2* recorre todo el camino, antes de que cualquier otro identificador haya sido enviado.

Ahora que se ha establecido que el modelo es consistente, es momento de comprobar algunas propiedades. Como se menciona anteriormente, se verificará que se elige exactamente un líder:

```
assert AtMostOneElected{
    lone elected.Time
}
check AtMostOneElected for 3 Process but 7 Time
```

El alcance de esta aserción limita al análisis a un anillo de 3 procesos y 7 instantes de tiempo. La afirmación *AtMostOneElected* es válida y no se encuentran contraejemplos.
Ahora, para verificar que eventualmente se elegirá a un líder, parece bastante obvia la siguiente pregunta:

```
assert AtLeastOneElected{
    some t: Time | some elected.t
}
check AtLeastOneElected for 3 but 7 Time
```

La afirmación *AtLeastOneElected* es inválida, el analizador Alloy nos muestra un contraejemplo en el que no pasa nada. El problema consiste en incluir la operación *skip* que le permite a los procesos *no hacer nada*.

Para solucionar este problema, se puede forzar el progreso haciendo que siempre que algún proceso no vaya a hacer nada, entonces algún proceso (no necesariamente el mismo) debe hacer un movimiento. Se escribe esto como un predicado:

```
pred progress(){
    all t: Time - TO/last[] |
        let t′ = TO/next [t] |
            some Process.toSend.t =>
                some p: Process | not skip [t, t′, p]
}
```

Ahora, haciendo uso del predicado *progress*, la afirmación se redefine como sigue:

```
assert AtLeastOneElected{
    progress[] => some elected.Time
}
check AtLeastOneElected for 3 Process, 7 Time
```

El alcance de la afirmación de 7 instantes de tiempo es en realidad el más pequeño que garantiza encontrar un líder. Para encontrar este alcance, simplemente se comenzó con uno menor y se fue aumentando hasta que no se encontró ningún contraejemplo.

En la figura 2.4 se muestra una traza generada por el analizador Alloy de la elección de un proceso líder en una topología de anillo. En el panel de arriba a la izquierda el modelo se encuentra en el estado inicial y la ejecución continua a través del resto de los paneles en sentido del reloj.

En este capítulo comprendimos que es la especificación en lógica relacional asi como el lenguaje y la sintaxis del analizador Alloy. En base a las ejemplificaciones que hicimos, podemos generar nuestras propias especificaciones para modelar y analizar algún problema en particular.

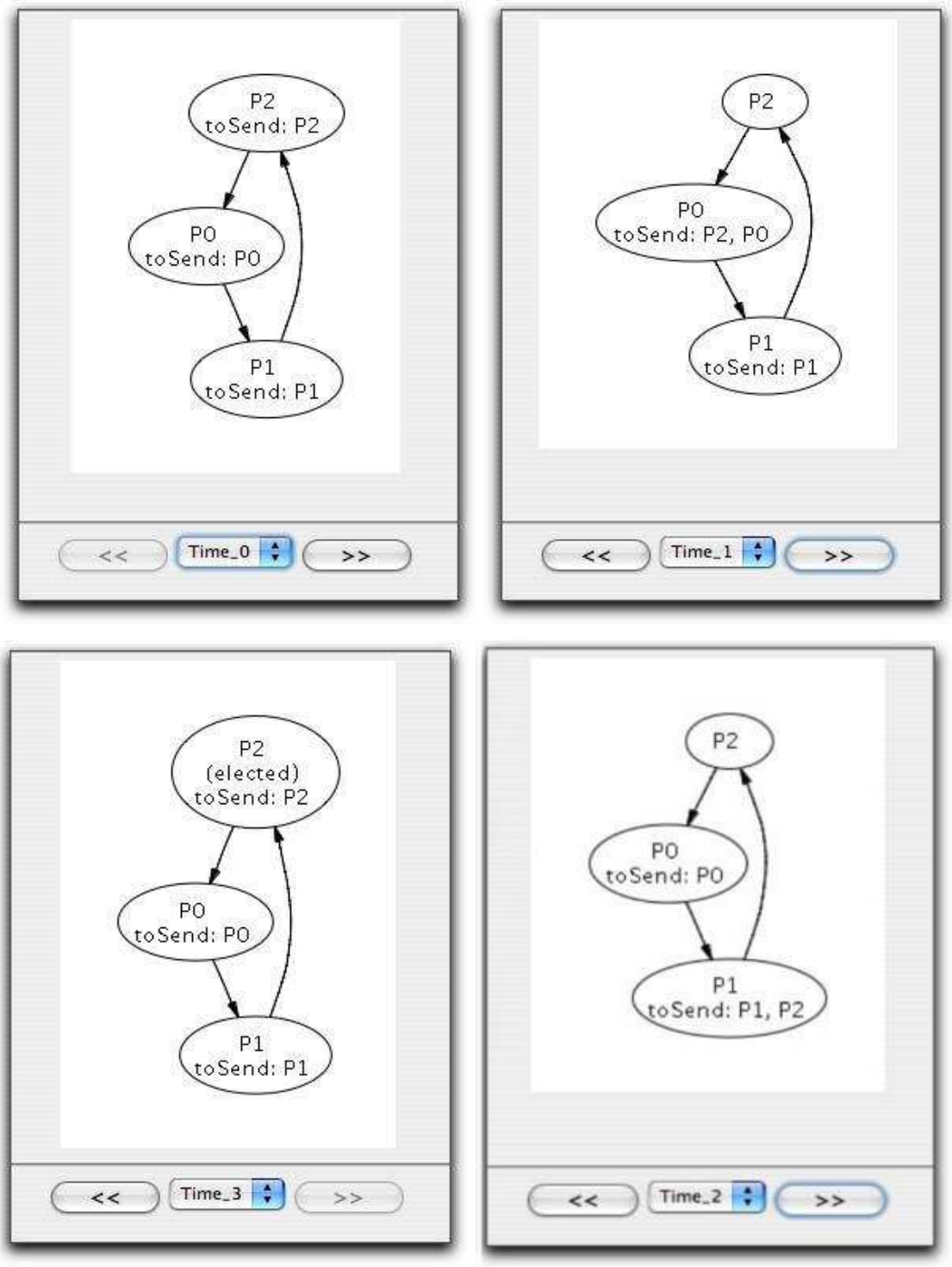

Figura 2.4: Traza generada por el analizador Alloy de la elección de un proceso líder en una topología de anillo.

3: Estudio de caso: *Problema de usuarios y pisos múltiples para un elevador*

Es conocido el problema que tienen los elevadores para servir a un gran número de peticiones que los usuarios realizan, muchas veces el elevador no se comporta de la manera deseada, tarda mucho en responder o, en algunos casos, puede no atender a algún usuario. En el estudio de caso se encontró una manera parcial de resolver este problema.

Existen algunas especificaciones de este problema que permiten resolverlo pero de manera muy limitada, solo para un número reducido de usuarios y de pisos, ó solo se da una descripción de como tienen que ser las reglas lógicas (Strobol-Wisspeintner 1999), otros resuelven el problema con lógica temporal (Wood 1989), la diferencia de estos y lo hecho en esta investigación radica en el uso de métodos formales, además gracias a la herramienta gráfica del analizador *Alloy* podemos ver ejemplos particulares o instancias de como funcionaría el modelo; generalmente este problema se resuelve con lógica difusa (Khiang-Khalid-Yusof 2007), pero esta no asegura que nuestro sistema sea completamente fiable. El objetivo de este trabajo es ver como a partir de ciertos elementos escritos en lenguaje de *Alloy* podemos ver una serie de instancias que nos muestren el funcionamiento de un elevador asegurándonos lo que debe de hacer este sistema. El estudio de este caso se realizó pensando en generalizar el problema de tal forma que para todo usuario en cualquier piso, el elevador sea capaz de atender su petición en un número finito de pasos .

Se modela una política en la que la prioridad más alta en orden descendente esta dada por: la dirección que obtiene el elevador para responder a la primera petición que se hizo, el usuario que va al piso mas cercano deseado en la dirección del movimiento, el usuario que hizo la primera petición con la dirección del elevador opuesta a la prefijada anteriormente. Se comenzó a desarrollar la especificación para un solo usuario con el número de pisos variables y una vez entendido el problema, se extendió para un número de usuarios variable de tal forma que se muestre la evolución del trabajo.

3.1. Primera aproximación

El primer paso es entender cómo se comporta un elevador. En general los elevadores se comportan de la siguiente manera: un usuario está en algún piso y realiza una petición al elevador oprimiendo un botón, generalmente en la parte de arriba de los elevadores se muestra en un *display* la dirección actual que tiene pudiendo no ser la deseada por el usuario. Eventualmente el elevador llega al piso

en donde está el usuario y éste sube sabiendo que el elevador no cambiará de dirección sino que tiene la dirección que el usuario quiere seguir. En el camino que sigue el elevador puede haber otras peticiones de otros usuarios que están en otros pisos, el elevador, al seguir su trayectoria, va encontrando en el camino a estos usuarios que tienen el deseo de ir a alguno de los pisos que se encuentran en la dirección que lleva este. Si el elevador llega a un piso en donde está algún otro usuario que quiere seguir la misma trayectoria, el elevador abre su puerta y el o los usuarios se introducen seleccionando el piso al que quieren ir, este dejará a algunos usuarios en el piso deseado; siempre considerando satisfacer a todas las peticiones en el orden en el que van apareciendo los pisos. Por supuesto la prioridad la tendrá el usuario que va al piso mas cercano deseado en la dirección de movimiento.

3.1.1. Limitaciones

La manera en que funciona un elevador se modeló de una manera muy similar a la realidad, con algunas limitaciones, en el presente trabajo no se modela el uso de puertas o de botones que no sean los que representan algún piso; esto quiere decir que no existe el botón de emergencia para detener el elevador, el botón de alarma, los botones de piso dentro del elevador o cualquier otro de este tipo. Tampoco se modelan las luces que representan, por ejemplo, en donde está un elevador o la representación que muestra que la luz sigue prendida hasta que el elevador llegue a donde está el usuario. Las condiciones de falla de un elevador tampoco son consideradas en la especificación y tampoco la capacidad de un elevador.

3.1.2. Modelo básico

Existe un número de pisos representados por la signatura:

sig Piso {}

Existe solo un elevador representado por la signatura:

one sig Elevador {}

Existe un número de usuarios representados por la signatura:

sig Usuario {}

Existen tres tipos de direcciones representados por la signatura:

sig Direccion {}

Existe una representación de pasos en el tiempo, representados por la signatura:

sig Tiempo {}

Hay ciertos elementos del modelo que nos interesan ordenar, por ejemplo los pisos, queremos que estén ordenados Piso0, Piso1,...,PisoN, también es deseable que los usuarios estén ordenados ya que nos interesa que las peticiones de los usuarios sean resueltas en orden Usuario0, Usuario1,...,UsuarioN; otro elemento importante que debemos ordenar es el tiempo, que nos va a servir para ver como se mueve el elevador, este se moverá de posición de un tiempo a otro, Tiempo0, Tiempo1,...,TiempoN en donde el tiempo final será aquel que haya cumplido con todas las peticiones. Para hacer estos ordenamientos al igual que para usar ciertas operaciones sobre conjuntos ordenados se importará la biblioteca ordering que ya está implementada y puede ser usada como una ayuda para realizar la especificación.

open util/ordering [Tiempo] as TO
open util/ordering [Piso] as PO
open util/ordering [Usuario] as U

Las signaturas Usuario y Elevador representan relaciones cuyo número de argumentos es cuatro.

sig Usuario{
    peticion: (Piso -> Piso) **lone** -> Tiempo
}

sig Elevador{
    posicion:(Direccion -> Piso) **one** -> Tiempo
}

La relación múltiple petición que se refiere a las peticiones que hacen los usuarios, (Usuario -> Piso -> Piso -> Tiempo) nos dice por ejemplo que un usuario “U” quiere ir a un piso “P1” estando en otro piso “P5” en un tiempo “T8” dado.

U-> P1-> P5-> T8

Utilizamos el cuantificador *lone* que nos dice que una petición hecha por un usuario está relacionada a lo más con un piso en un tiempo dado, esto quiere decir que no puede pasar que un usuario esté en dos pisos al mismo tiempo.

Se hizo una relación múltiple cuyo número de argumentos es cuatro con el objeto de tener una bandera que mantuviera el estado del piso al que quiere ir el usuario, es la manera en la que nos podemos percatar a donde quiere ir el usuario.

En el caso anterior "P1" será la bandera que tendrá que revisar el elevador para saber si recoge o no al usuario "U".

La relación múltiple que refiere la posición del elevador (Elevador -> Direccion -> Piso -> Tiempo) es similar a la de usuario salvo que difiere en el tipo de bandera, que en este caso será la "Direccion" que el elevador seguirá; la dirección solo podrá ser de 3 tipos arriba, abajo y en reposo que serán representada por la relación abstracta "Direccion":

abstract sig Direccion{
}

Por ser una signatura abstracta, definimos cuáles serán los únicos elementos que habrá, estos serán aquellos pertenecientes a sus extensiones. Entonces dirección será un conjunto con tres elementos:

one sig arriba, abajo, reposo **extends** Direccion{
}

Luego entonces la relación posición describirá por ejemplo un elevador "E" que lleva una dirección "arriba" y está parado en el piso "P2" en el tiempo "T3"

E -> arriba -> P2 -> T3

En las signaturas de "Elevador" y "Usuario" el último parámetro de la relación es el tiempo, este parámetro es el que permite ver el funcionamiento dinámico del modelo, ya que podremos realizar una proyección sobre el tiempo para cada instancia que obtengamos por medio del analizador, esto permitirá que visualicemos un estado de nuestro modelo en un tiempo dado y al seguir estos tiempos podremos ver como el elevador realiza su trayectoria para cumplir con todas las peticiones hechas en cada instancia que obtengamos. Este parámetro se acostumbra utilizar al final por comodidad en el manejo de las relaciones y porque suele ponerse al final en los lenguajes de modelado.

3.2. Primer modelo

El primer modelo se hizo pensando en un solo usuario para ver cuál era su comportamiento, de tal manera que al entender esto podríamos extender el modelo a más usuarios. Al existir solo un usuario utilizaremos la biblioteca *ordering* solo para ordenar los tiempos y los pisos.

```
open util/ordering [Tiempo] as TO
open util/ordering [Piso] as PO
```

La especificación tendrá que cumplir una serie de restricciones que se dan por hecho y se cumplirán en todo momento en el modelo. En el momento en el que pensamos cuáles son los hechos necesarios para nuestro modelo podemos ver que tienen ciertas características en común, las cuales agruparemos en cuatro grupos: las condiciones iniciales, condiciones finales, condiciones de movimiento y condiciones para los usuarios.

Restricción: Sólo hay un usuario y un elevador en todo momento(esta restricción esta dada por la signatura elevador).

```
# Usuario = 1
```

Restricción: No debe de haber una instancia en la que el elevador vaya hacia arriba o hacia abajo y no exista alguna petición, es decir que siempre que el elevador tenga una dirección distinta a la de reposo es porque existe una petición.

```
fact cond_inicial {
    all t:Tiempo - last | ((Elevador.posicion).t).Piso != reposo implies
        peticion != none->none->none->none
    }
```

Restricción: No debe existir una instancia en la que el elevador esté en el último piso o en el piso de inicio y el elevador quiera ir hacia arriba o hacia abajo respectivamente.

```
fact cond_movimiento {
    no t:Tiempo | ( (posicion.t).PO/last).arriba != none
    no t:Tiempo | ( (posicion.t).PO/first).abajo != none
}
```

Restricción: En el último tiempo no existen peticiones ya que en el último tiempo se habrán resuelto todas las peticiones que realizaron los usuarios y el elevador se encontrará en reposo.

```
fact cond_final {
    peticion.TO/last = none -> none -> none
}
```

Restricción: En el último tiempo la dirección del elevador siempre esta en reposo.

```
fact cond_final {
    reposo.((Elevador.(posicion)).TO/last) != none
}
```

Restricción: En el primer tiempo siempre hay peticiones. Esto lo definimos de esta manera porque queremos ver la evolución de un ejemplo desde el primer tiempo hasta que termine de servir todas las peticiones.

```
fact cond_inicial {
    peticion.TO/first != none -> none -> none
}
```

Restricción: Siempre existen peticiones menos en el último tiempo.

```
fact cond_final {
    (peticion.(Tiempo-last ) != none -> none -> none )
}
```

Restricción: No pasa que un usuario quiera ir a un piso en el que ya está. Hay que tomar en cuenta que esta regla solo sirve para un usuario.

```
fact cond_inicial {
    # Usuario.(peticion.Tiempo.Piso) = 1
}
```

Entonces a partir de las restricciones anteriores tenemos el siguiente código:

```
open util/ordering [Tiempo] as TO
open util/ordering [Piso] as PO


sig Tiempo{
    }

sig Piso{
    }
```

abstract sig Direccion{
}

one sig arriba, abajo, reposo **extends** Direccion{
}

–Un usuario hace una petición en un piso hacia otro piso en un tiempo dado.

sig Usuario{
peticion: (Piso -> Piso) **lone** -> Tiempo
}

–Un elevador esta en un piso con una dirección en un tiempo dado.

one sig Elevador{
posicion:(Direccion -> Piso) **one** -> Tiempo
}

fact cond_iniciales {

–Solo hay un usuario.
Usuario = 1

–No pasa que en el primer tiempo no haya peticiones.
peticion.TO/**first** != **none** -> **none** -> **none**

–La primera posición del elevador tiene que estar asociada a un piso.
posicion.**first** != **none** -> **none** -> **none**

}

fact cond_finales {

–Para que en el último tiempo el usuario no aparezca.
peticion.TO/**last** = **none** -> **none** -> **none**

–En todos los tiempos a excepción del último hay peticiones.
(peticion.(Tiempo-**last**) != **none** -> **none** -> **none**)

–En el último tiempo siempre está en reposo el elevador.
reposo.((Elevador.(posicion)).TO/**last**) != **none**

}

fact cond_usuarios {

–El piso al que el usuario quiere ir siempre es el mismo.
Usuario.(peticion.Tiempo.Piso) = 1

}

fact cond_movimiento {

–No pasa que en el último piso el elevador quiera ir hacia arriba.
no t:Tiempo | ((posicion.t).PO/**last**).arriba != **none**

–No pasa que en el primer piso el elevador quiera ir hacia abajo.
no t:Tiempo | ((posicion.t).PO/**first**).abajo != **none**

–No debe de haber una instancia en la que el elevador vaya hacia arriba o hacia abajo.
–y no haya ninguna petición de algún usuario, es decir si la dirección no es en reposo entonces existe una petición.

all t:Tiempo - **last** | ((Elevador.posicion).t).Piso != reposo **implies**
peticion != **none**->**none**->**none**->**none**
}

}

3.2.1. Análisis

En esta parte de la especificación es importante hacer el análisis con respecto a las restricciones propuestas, es deseable ver si estos hechos funcionan correctamente y hacen realmente lo que deseamos. Para hacerlo utilizaremos los *asserts* o aserciones, de tal manera que el analizador verifique si hay un contraejemplo al hecho que estamos diciendo que no puede pasar y así podemos verificar si la especificación de los hechos son o no correctas.

En cada aserción verificamos que pasaría sino no pusiéramos esa restricción como hecho.

Aserción: Aseguramos que en el último tiempo no existan peticiones.

assert peticion_ultimoTiempo {
not (peticion.TO/**last** != **none** -> **none** -> **none**)
}

En la figura 3.1 vemos una instancia de la aserción en la cual nos damos cuenta que en el último tiempo todavía hay peticiones por servir, lo cual es incorrecto.

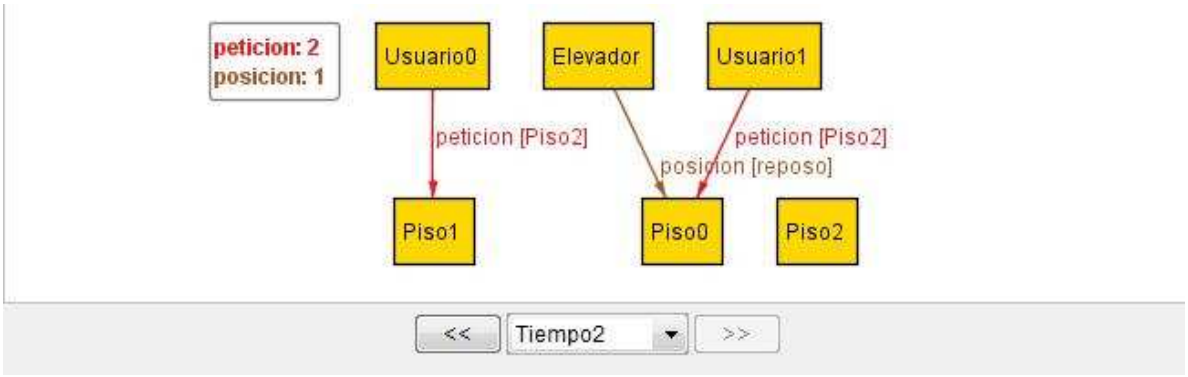

Figura 3.1: Instancia generada por el assert peticion_ultimoTiempo.

Aserción: En el primer tiempo deberá haber peticiones.

assert peticion_primerTiempo {
not (peticion.TO/**first** = **none** -> **none** -> **none**)
}

En la figura 3.2 vemos una instancia de la aserción en la cual no existen peticiones lo cual no tiene ningún sentido.

Figura 3.2: Instancia generada por el assert peticion_primerTiempo.

Aserción: La primera posición del elevador tiene que estar asociada a un piso.

assert posicion_primerTiempo {
not (posicion.TO/**first** = **none** -> **none** -> **none**)
}

En la figura 3.3 vemos una instancia de la aserción donde el elevador no aparece lo cual no es deseable.

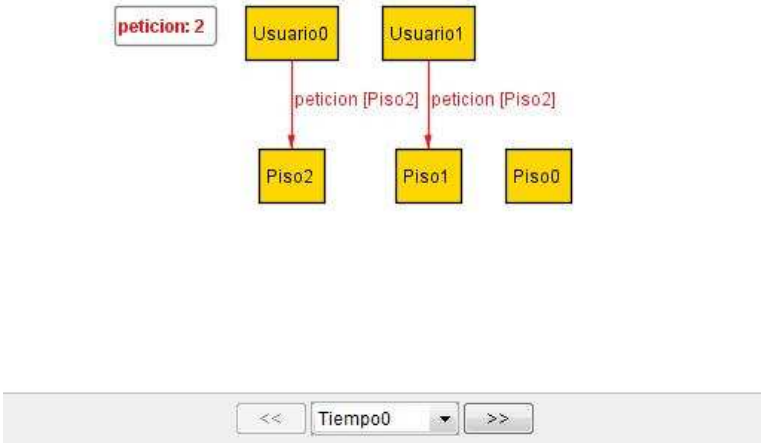

Figura 3.3: Instancia generada por el assert posicion_primerTiempo.

Aserción: No puede pasar que un usuario en el primer tiempo quiera ir al piso en el que ya está.

assert estar_EnPisoQueQuieres {
not (# Usuario.(peticion.Tiempo.Piso) > 1 and # Usuario = 1)
}

En la figura 3.4 vemos una instancia de la aserción que nos muestra un caso que no tiene sentido.

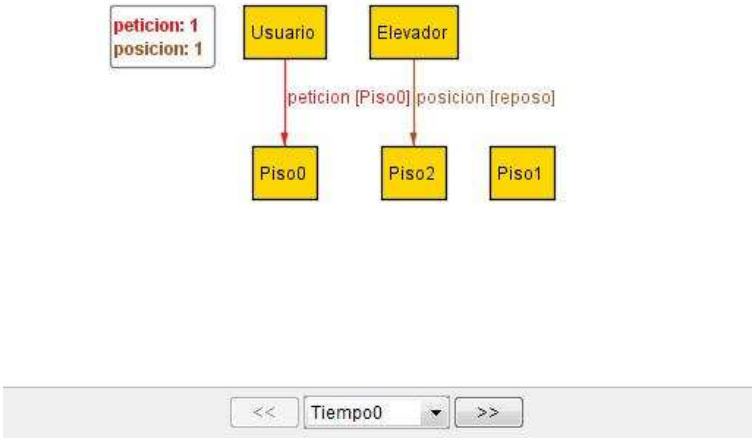

Figura 3.4: Instancia generada por el assert estar_EnPisoQueQuieres.

Aserción: No hay una instancia en la que el elevador esté en el último piso y va hacia arriba.

```
assert ultimoPiso_HaciaArriba {
    not( one t:Tiempo | (Direccion.(Elevador.posicion)).t & PO/last != none
            and ((Elevador.posicion).t).Piso = arriba )
    }
```

En la figura 3.5 vemos una instancia de la aserción que no es correcta.

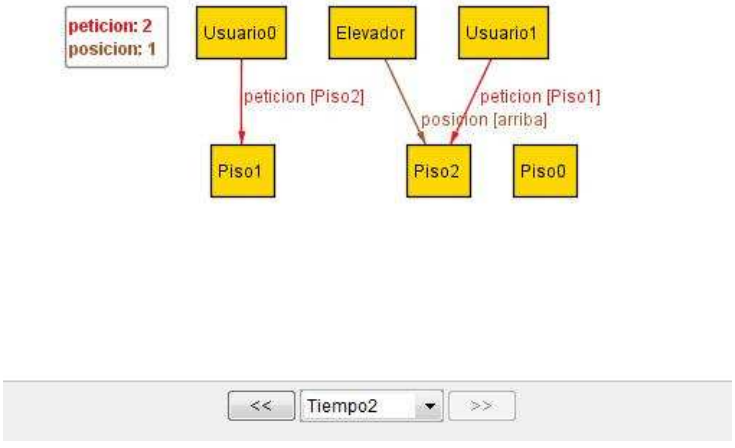

Figura 3.5: Instancia generada por el assert ultimoPiso_HaciaArriba.

Aserción: No hay una instancia en la que el elevador esté en el primer piso y va hacia abajo.

```
assert primerPiso_HaciaAbajo {
    not( one t:Tiempo | (Direccion.(Elevador.posicion)).t & PO/first != none
```

and ((Elevador.posicion).t).Piso = abajo)
}

En la figura 3.6 vemos una instancia de la aserción que nos muestra este caso que no es deseable.

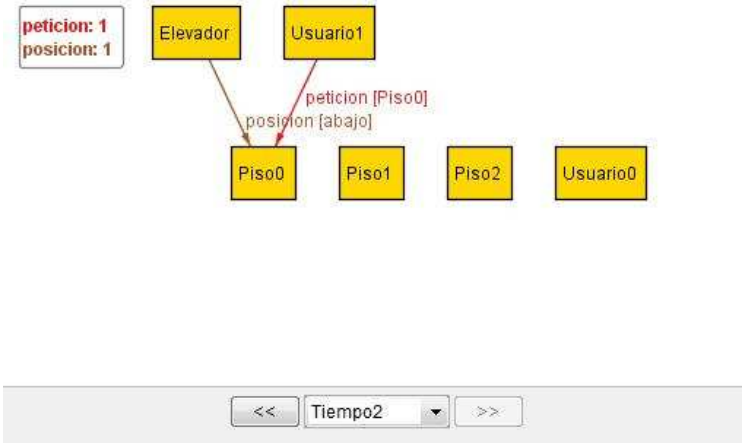

Figura 3.6: Instancia generada por el assert primerPiso_HaciaAbajo.

Aserción: En el último tiempo siempre está en reposo el elevador.

assert ElevReposo_UltimoTiempo {
reposo.((Elevador.(posicion)).TO/**last**) != **none**
}

En la figura 3.7 vemos una instancia de la aserción que nos muestra que en el último tiempo el elevador no esta en reposo lo cual es incorrecto.

Aserción: No debe de haber una instancia en la que el elevador vaya hacia arriba o hacia abajo y no haya ninguna petición de algún usuario.Es decir si la dirección no es en reposo entonces existe una petición.

assert noReposo_ ent_Petición {
all t:Tiempo - **last** | ((Elevador.posicion).t).Piso != reposo **implies**
peticion != **none**->**none**->**none**->**none**
}

En la figura 3.8 vemos una instancia de la aserción en la que vemos que el elevador tiene una dirección hacia arriba, pero no hay petición por la cual responder, lo pertinente sería que estuviera en reposo.

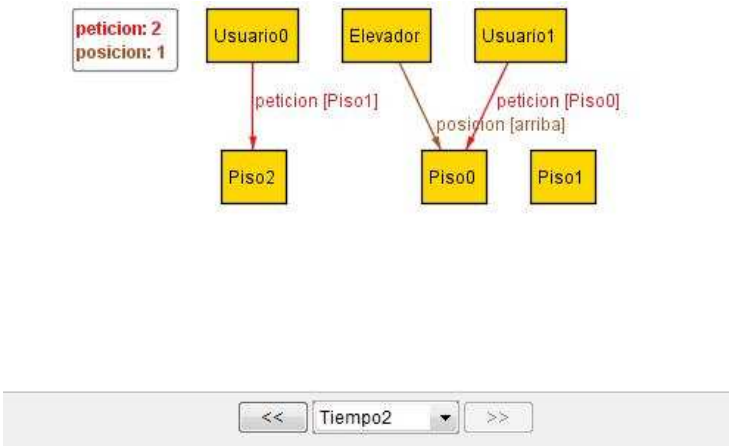

Figura 3.7: Instancia generada por el assert ElevReposo_UltimoTiempo.

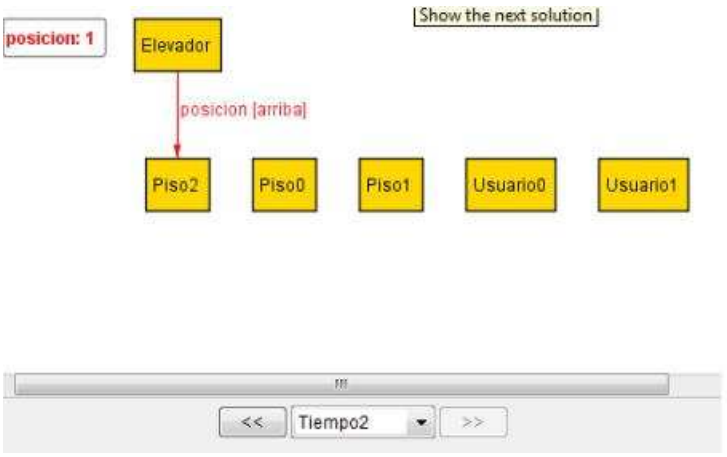

Figura 3.8: Instancia generada por el assert noReposo_ ent_Petición .

A partir de estas aserciones podemos darnos cuenta qué hechos son necesarios en nuestro modelo ya que sin ellos caemos en errores; por lo tanto son restricciones necesarias.

3.2.2. Definición de una traza

Una vez que se definieron los hechos queremos verificar las propiedades del movimiento del elevador por partes y una vez hecho esto podemos generar una secuencia con este conjunto de propiedades, simulando transiciones para ver el movimiento del elevador desde un estado inicial hasta un estado final. Cada operación representará la transición de un estado en un tiempo dado a otro estado en el tiempo siguiente. Para realizar esta representación dinámica del modelo se utilizará un traza, creando un orden lineal sobre los estados, que permite hacer esta conexión entre estados sucesivos por medio de las operaciones, de tal forma que todos estos son alcanzables.

Representación general de cualquier traza:

```
open util/ordering[State]

pred init (s: State) ...
pred op1 (s, s': State) ...
pred opN (s, s': State) ...

    fact Traces{
        init[S/first[]]
        all s: State - S/last[] |
            let t' = s.next | op1[s, s'] or ... or opN[s, s']
    }
```

En el ejemplo anterior podemos ver la estructura de una traza que utiliza el módulo *ordering* para ordenar los átomos de la signatura *State*, un predicado *init* que da la condición inicial de la transición, una serie de predicados que serán las operaciones que deberá de cumplir la traza. Esta será un hecho el cual dirá que alguna de las operaciones siempre es cierta para algún estado, esto permitirá la conexión entre cada cada uno de estos.

3.2.3. Reglas de movimiento para el primer modelo

En esta sección explicaremos el funcionamiento de las reglas para el movimiento de un elevador con un solo usuario.

Existen ciertos elementos del modelo que siempre tenemos que tomar en cuenta para construir nuestras reglas de movimiento, estos son:

1. El piso donde esta el elevador:	x = (Direccion.(Elevador.posicion)).Tiempo
2. El piso a donde quiere ir el usuario:	y = ((Usuario.peticion).Tiempo).Piso
3. El piso Donde esta el usuario:	z = Pisos.((Usuario.(peticion.Tiempo)))
4. Dirección que tiene el elevador:	d = ((Elevador.posicion).Tiempo).Piso

Las siguientes reglas dicen en donde está el elevador, a donde quiere ir el usuario y en donde está el usuario, de un tiempo a otro.

Regla 0:

La regla 0 describe el movimiento de un elevador que está en el mismo piso que el usuario, quiere ir hacia abajo y el piso al que quiere ir el usuario no es el siguiente inmediato. El predicado tiene 5 atributos: el tiempo t que es el antecesor del tiempo t', el piso "x" que es el piso donde está el elevador, "y" el piso al que el usuario quiere ir y "z" es el piso en el que está el usuario, todos en el tiempo t. Utilizaremos x', y' y z' para representar en donde estarán en el siguiente tiempo t' y d para representar la dirección que tiene el elevador.

```
pred r0 (t: Tiempo , t': Tiempo, x, y, z:Piso) {
    let x=(Direccion.(Elevador.posicion)).t, y=((Usuario.peticion).t).Piso,
        z=Piso.((Usuario.(peticion.t))) ,
        x' = (Direccion.(Elevador.posicion)).t', y' = ((Usuario.peticion).t').Piso,
        z'= Piso.((Usuario.(peticion.t'))),
        d= ((Elevador.posicion).t).Piso , d'= ((Elevador.posicion).t').Piso |
            x = z and y != x and (y = min[x + y])
            and x != none and y != none and z != none
            and d = abajo
            and y != x.prev and y != z.prev
            and x' = x.prev and z' = z.prev and x' = z' and y' = y
        }
```

x = z; *El usuario está en el mismo piso que el elevador.*
y != x; *El piso donde está el elevador, es distinto al piso al que quiere ir el usuario.*
(y = min[x + y]); *El piso al que quiere ir el usuario, esta en un piso más abajo que en el que esta el elevador.*
x != none and y != none and z != none; *Los pisos representados por* x, y, y z, *para* z y y *están asociados a un usuario, y para* x *a un elevador.*
d = abajo; *La dirección del elevador es abajo.*
y != x.prev and y != z.prev; *El piso al que quiere ir el usuario, no puede ser el inmediato siguiente en donde deberán estar el elevador con el usuario.*
x' = x.prev and z' = z.prev; *Tanto el elevador como el usuario, estarán en el piso anterior al piso en el que están en el tiempo siguiente.*
x' = z'; *El usuario esta en el mismo piso que el elevador en el tiempo siguiente.*
y' = y; *El piso al que quiere ir el usuario, seguirá siendo el mismo en el tiempo siguiente.*

Nota: Se utilizaron varios métodos del módulo ordering, como prev que nos dice cual es el elemento previo de un conjunto ordenado, también se utilizo min[a+b], que nos dice cual es el elemento mínimo de un conjunto ordenado. Se utilizarán otros métodos de este módulo en reglas posteriores.

En la figura 3.9 vemos un ejemplo del funcionamiento de la regla 0, en la que un usuario baja un piso.

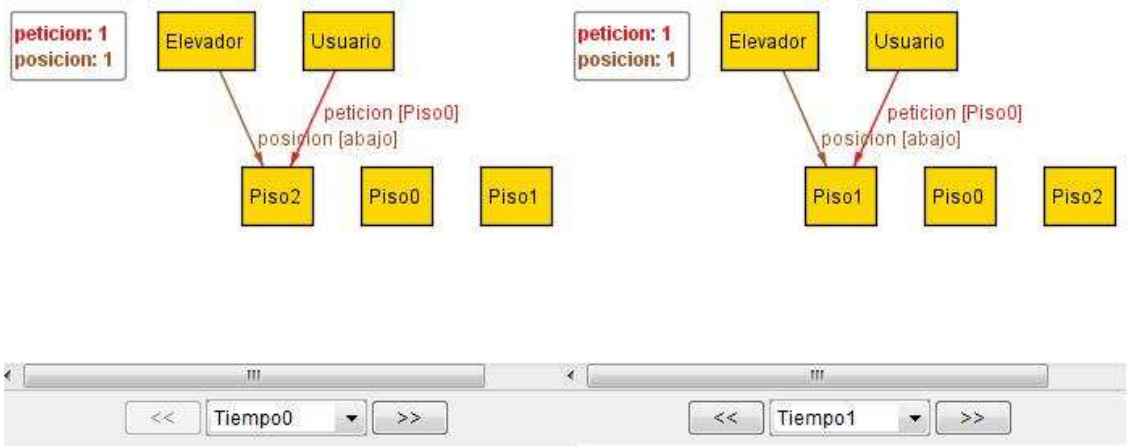

Figura 3.9: Movimiento de la regla 0.

Regla 1:

La regla 1 describe el movimiento de un elevador, que está en el mismo piso que el usuario, quiere ir hacia arriba y el piso al que quiere ir el usuario no es el siguiente inmediato. Es muy similar a la regla 0, la diferencia es la dirección del elevador que es hacia arriba, esto quiere decir que el elevador se moverá con el usuario al piso siguiente que este arriba.

```
pred r1(t: Tiempo , t': Tiempo, x, y, z:Piso) {
    let x=(Direccion.(Elevador.posicion)).t, y=((Usuario.peticion).t).Piso,
        z=Piso.((Usuario.(peticion.t))) ,
        x' = (Direccion.(Elevador.posicion)).t', y' = ((Usuario.peticion).t').Piso,
        z'= Piso.((Usuario.(peticion.t'))),
        d= ((Elevador.posicion).t).Piso , d'= ((Elevador.posicion).t').Piso |
            x = z and y != x and (y = max[x + y])
            and x != none and y != none and z != none
            and d = arriba
            and y != x.next and y != z.next
            and x' = x.next and z' = z.next and x' = z' and y' = y
    }
```

En la figura 3.10 vemos un ejemplo del funcionamiento de la regla 1, en la que un usuario sube un piso.

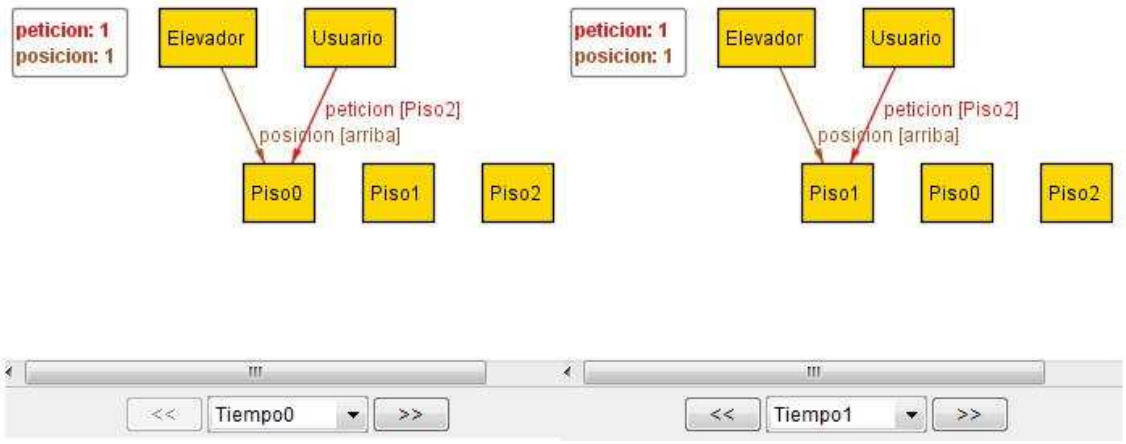

Figura 3.10: Movimiento de la regla 1.

Regla 2:

La regla 2 describe el movimiento de un elevador vacío, o sea, que no está en el mismo piso que el usuario y quiere ir hacia abajo.

```
pred r2(t: Tiempo , t': Tiempo, x, y, z:Piso) {
    let x=(Direccion.(Elevador.posicion)).t, y=((Usuario.peticion).t).Piso,
        z=Piso.((Usuario.(peticion.t))) ,
        x' = (Direccion.(Elevador.posicion)).t', y' = ((Usuario.peticion).t').Piso,
        z'= Piso.((Usuario.(peticion.t'))) ,
        d= ((Elevador.posicion).t).Piso , d'= ((Elevador.posicion).t').Piso |
            ( (x != z)
            and (z = min[x + z])
            and x != none and y != none and z != none
            and d = abajo
            and (x.prev =x' ) and (y' = y) and (z'= z) )
    }
```

((x != z); *El usuario no se encuentra en el mismo piso en el que está el elevador.*
(z = min[x + z]); *El usuario está en cualquier piso anterior a donde se encuentra* actualmente el elevador.
(x.prev =x'); *El elevador se moverá al piso inmediato que se encuentre abajo.*

En la figura 3.11 vemos un ejemplo del funcionamiento de la regla 2, en la que el elevador baja un piso.

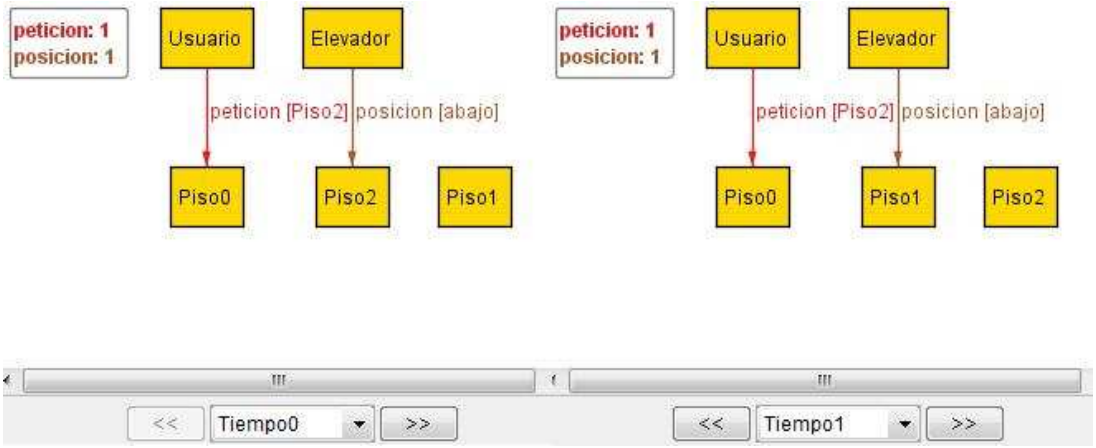

Figura 3.11: Movimiento de la regla 2.

Regla 3:

La regla 3 describe el movimiento de un elevador vacío, o sea, que no está en el mismo piso que el usuario y quiere ir hacia arriba. Es muy similar a la regla anterior, la diferencia es la dirección del elevador que es hacia arriba, esto quiere decir que el elevador se moverá al piso siguiente que este arriba.

```
pred r3(t: Tiempo , t': Tiempo, x, y, z:Piso) {
    let x=(Direccion.(Elevador.posicion)).t, y=((Usuario.peticion).t).Piso,
        z=Piso.((Usuario.(peticion.t))) ,
        x' = (Direccion.(Elevador.posicion)).t', y' = ((Usuario.peticion).t').Piso,
        z'= Piso.((Usuario.(peticion.t'))),
        d= ((Elevador.posicion).t).Piso , d'= ((Elevador.posicion).t').Piso |
        ( (x != z)
            and (x = min[x + z])
            and x != none and y != none and z != none
            and d = arriba
            and (x.next =x' ) and (y' = y) and (z'= z) )
    }
```

En la figura 3.12 vemos un ejemplo del funcionamiento de la regla 3, en la que un elevador sube un piso .

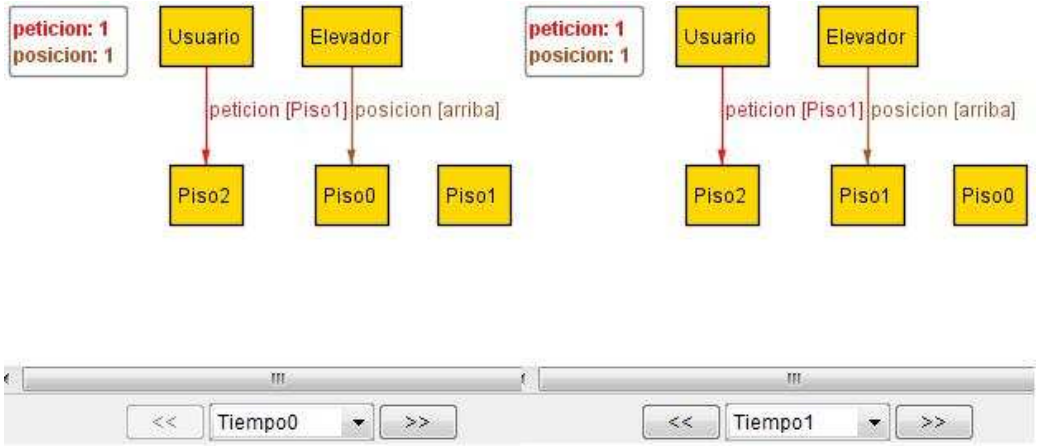

Figura 3.12: Movimiento de la regla 3.

Regla 4:

La regla 4 describe el movimiento de un elevador con el usuario, quiere ir hacia abajo y el piso al que quiere ir el usuario es el siguiente inmediato. Esta regla es parecida a la regla 0 con algunas diferencias importantes.

```
pred r4 (t: Tiempo , t': Tiempo, x, y, z:Piso) {
    let x=(Direccion.(Elevador.posicion)).t, y=((Usuario.peticion).t).Piso,
        z=Piso.((Usuario.(peticion.t))) ,
        x' = (Direccion.(Elevador.posicion)).t', y' = ((Usuario.peticion).t').Piso,
        z'= Piso.((Usuario.(peticion.t'))),
        d= ((Elevador.posicion).t).Piso , d'= ((Elevador.posicion).t').Piso |
            x = z and y != x
            and y = x.prev and y = z.prev
            and x != none and y != none and z != none
            and d = abajo
            and x' = x.prev and z'= z.prev and y' = y
            and d' = reposo
    }
```

y = x.prev and y = z.prev; *El piso al que quiere ir el usuario, es el inmediato siguiente al piso en el que se encuentra.*
d' = reposo; *La dirección del elevador en el tiempo t' será la de reposo ya que el elevador habrá llegado al destino deseado.*

En la figura 3.13 vemos un ejemplo del funcionamiento de la regla 4, en la que un usuario baja al piso al que quiere ir.

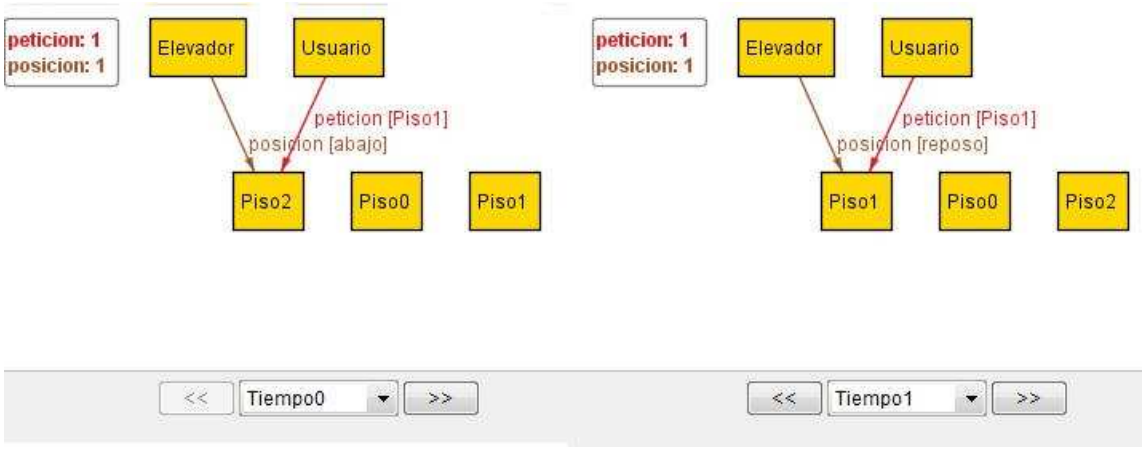

Figura 3.13: Movimiento de la regla 4.

Regla 5:

La regla 5 describe el movimiento de un elevador con el usuario, quiere ir hacia arriba y el piso al que quiere ir el usuario es el siguiente inmediato. Es muy similar a la regla anterior con la diferencia de la dirección del elevador que es hacia arriba esto quiere decir que el elevador se moverá con el usuario al piso siguiente que este arriba y al dejarlo permanecerá en reposo.

```
pred r5(t: Tiempo , t': Tiempo, x, y, z:Piso){
    let x=(Direccion.(Elevador.posicion)).t, y=((Usuario.peticion).t).Piso,
        z=Piso.((Usuario.(peticion.t))) ,
        x' = (Direccion.(Elevador.posicion)).t', y' = ((Usuario.peticion).t').Piso,
        z'= Piso.((Usuario.(peticion.t'))),
        d= ((Elevador.posicion).t).Piso , d'= ((Elevador.posicion).t').Piso |
        x = z and y != x and
            y= x.next and y = z.next
            and x != none and y != none and z != none
            and d = arriba
            and x' = x.next and z' = z.next and y'= y
            and d' = reposo
}
```

En la figura 3.14 vemos un ejemplo del funcionamiento de la regla 5, en la que un usuario sube al piso al que quiere ir.

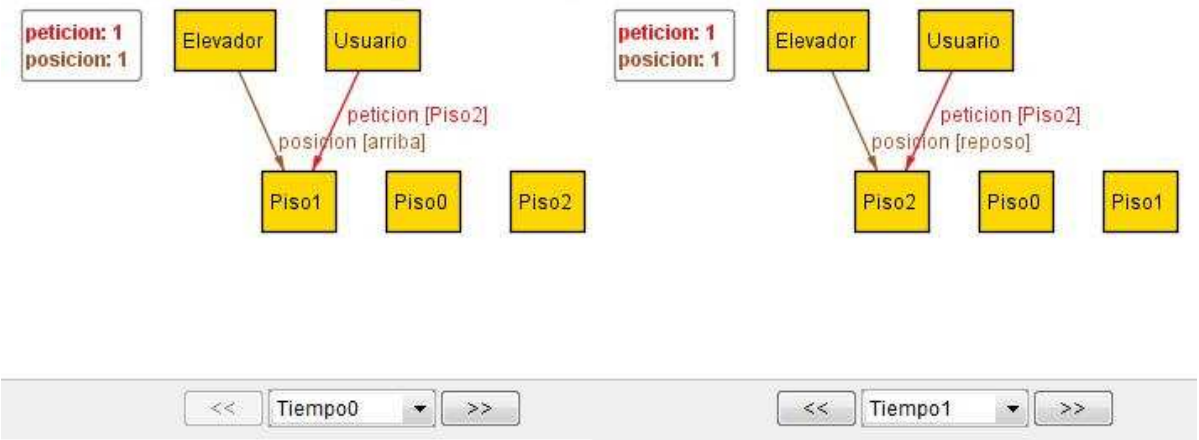

Figura 3.14: Movimiento de la regla 5.

Regla 6:

Esta regla describe el estado de reposo que tendrá el elevador cuando deje al usuario en el piso al que quería ir.

```
pred r6(t: Tiempo , t': Tiempo, x, y, z:Piso){
    let x=(Direccion.(Elevador.posicion)).t, y=((Usuario.peticion).t).Piso,
        z=Piso.((Usuario.(peticion.t))) ,
        x' = (Direccion.(Elevador.posicion)).t', y' = ((Usuario.peticion).t').Piso,
        z'= Piso.((Usuario.(peticion.t'))),
        d= ((Elevador.posicion).t).Piso, d'= ((Elevador.posicion).t').Piso |
            x = y and z= y
            and x' !=none
            and d = reposo
            and x' = x and z' = none and y'= none
            and d' = reposo
}
```

En la figura 3.15 vemos un ejemplo del funcionamiento de la regla 6, en la que el usuario termina su trayectoria y el elevador permanece en reposo.

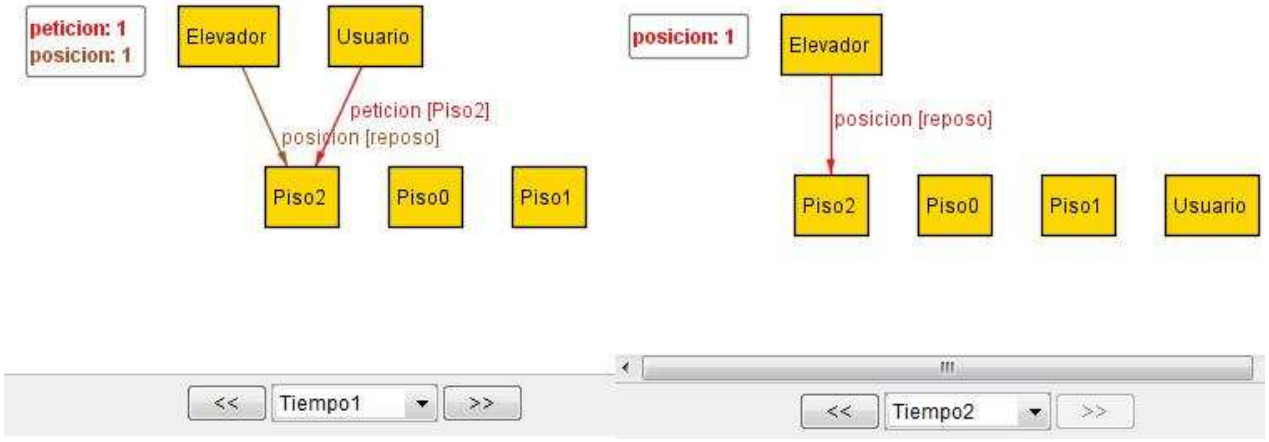

Figura 3.15: Movimiento de la regla 6.

Extrapolando la definición de traza que se dijo en la sección anterior y la serie de reglas descritas, tendremos lo siguiente:

La condición inicial será:

En el primer tiempo nunca pasa que el usuario quiere ir al piso en el que está.

```
pred init [t:Tiempo] {
        ((Usuario.peticion).t).Piso != Piso.((Usuario.(peticion.t)))
    }
```

¿Por qué es esta la condición inicial?

Esto es porque no tendría sentido.

En la figura 3.16 vemos una instancia incorrecta de la condición inicial.

Las reglas descritas nos dicen cómo van a ser las transiciones que el modelo tendrá que seguir en la traza la cual quedará de la siguiente manera:

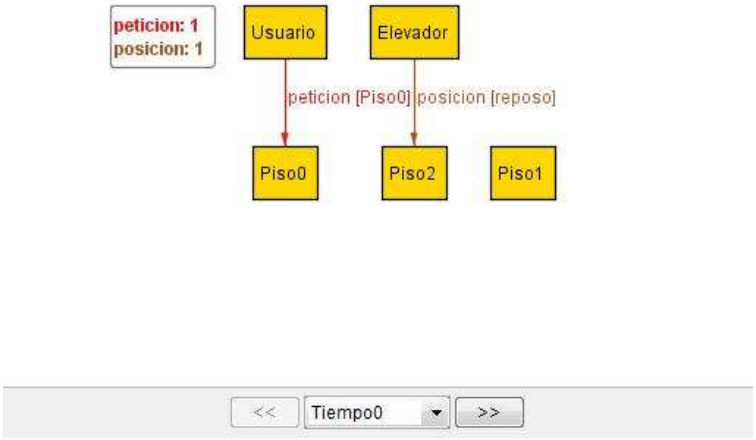

Figura 3.16: Predicado de inicio incorrecto.

```
fact trace {
    init[TO/first]
        all t:Tiempo - last | let t'= t.next |
            all x,y,z:Piso |
                r0[t,t',x,y,z] or r1[t,t',x,y,z] or r2[t,t',x,y,z] or r3[t,t',x,y,z]
                    or r4[t,t',x,y,z] or r5[t,t',x,y,z] or r6[t,t',x,y,z]
    }
```

Pero como podemos ver, nuestro modelo esta incompleto, ya que el funcionamiento para un solo usuario no es suficiente para entender el comportamiento del elevador, es deseable ver como funciona para un número mayor de usuarios haciendo mas real el modelo. En la sección siguiente veremos como hicimos crecer el modelo.

En las siguientes figuras visualizamos una instancia completa, la cual nos muestra un modelo en el que hay seis pisos y un usuario, este usuario hace una petición al piso0 estando en el piso5, el elevador está en el Piso4 y responderá a la petición del usuario para dejarlo en el piso deseado.

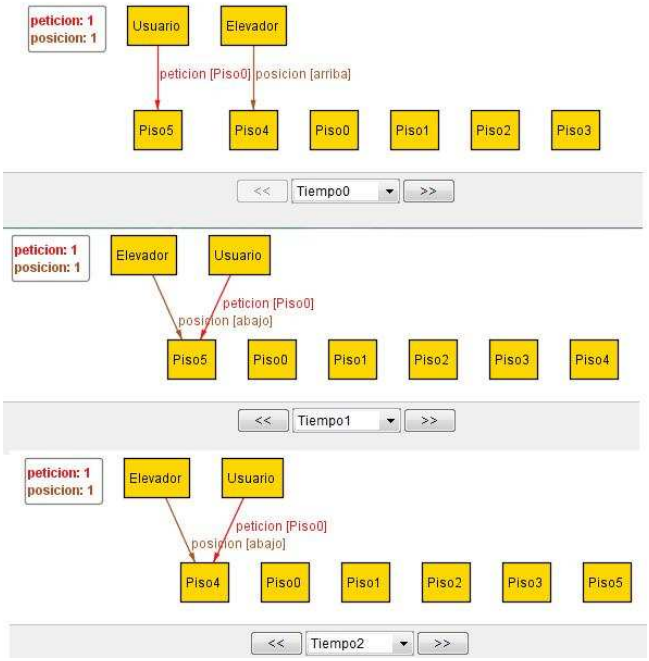

Figura 3.17: Ejemplo de un traza completa para un solo usuario parte 1.

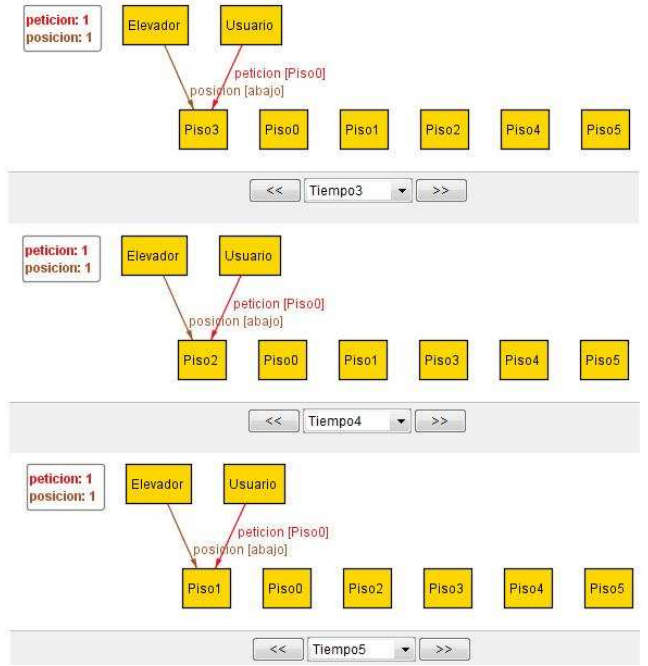

Figura 3.18: Ejemplo de un traza completa para un solo usuario parte 2.

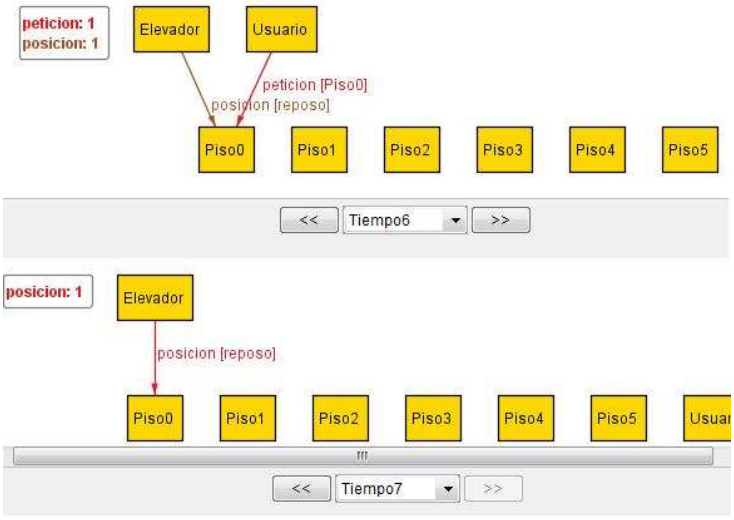

Figura 3.19: Ejemplo de un traza completa para un solo usuario parte 3.

3.3. Segundo Modelo

Una vez que ya se tiene una idea del comportamiento del elevador para un solo usuario, de las reglas básicas de movimiento que éste sigue, y de lo que no debe realizar, podemos extender el modelo para varios usuarios.

Hay ciertas cosas que no deben suceder:

1. No podrá pasar, que un elevador esté en el último piso o en el piso de inicio, y quiera ir hacia arriba o hacia abajo respectivamente.
2. No existirá ningún modelo en el que no haya ninguna petición.
3. No existirá ningún modelo en el que no se resuelvan todas las peticiones.
4. Un usuario no puede realizar una petición al mismo piso en el que está.
5. El mismo usuario no podrá estar en dos pisos diferentes al mismo tiempo
6. El elevador, no podrá pasar un piso sin recoger a algún usuario, que haya hecho una petición hacia algún piso por el cual éste va a pasar.
7. El elevador, no recogerá a un usuario que quiere ir hacia la dirección contraria que lleva este.

En base a lo anterior tenemos que extender nuestro modelo agregando nuevas restricciones que deben cumplirse para múltiples usuarios.

Restricción: El piso al que el usuario quiere ir siempre es el mismo.

```
fact cond_inicial {
    all disj u,v:Usuario | # u.peticion.(Tiempo - last).Piso= 1
                and # v.peticion.(Tiempo - last).Piso= 1
}
```

Esta restricción nos dice que el número de pisos al que quiere ir un usuario es uno, y esto le dirá al modelo que siempre es el mismo piso que eligió el usuario para cualquier tiempo menos el último tiempo, ya que en este se habrán terminado todas las peticiones, esto pasará para todos los usuarios.

Restricción: En el último tiempo el elevador siempre está en reposo. Tomemos en cuenta que el siguiente hecho por si solo no asegura esto, ya que hay que considerar que la signatura *elevador*, en todo momento solo permite que éste tenga asociada una dirección.

```
fact cond_final {
    reposo.((Elevador.(posicion)).TO/last) != none
}
```

Nótese aquí que hacemos uso de las funciones que ofrece la librería ordering "TO/last".

Restricción: No pasa que en el tiempo uno el elevador esté en reposo.

```
fact cond_inicial {
    ((Elevador.posicion).TO/first).Piso & reposo = none
}
```

Esta restricción implica que toda instancia dada por el analizador en el primer tiempo, no permitirá que el elevador esté en reposo sin responder a alguna petición, sino que el elevador estará activo con una dirección ya sea hacia arriba o hacia abajo.

Restricción:

```
fact cond_movimiento {
    no t:Tiempo | ( (posicion.t).PO/last).arriba != none
    no t:Tiempo | ( (posicion.t).PO/first).abajo != none
}
```

El hecho anterior nos dice que no existe un tiempo tal que el elevador esté en el último piso y tenga una dirección hacia arriba, de la misma manera no existe algún tiempo tal que el elevador esté en el piso de inicio y el elevador tenga una dirección hacia abajo.

Restricción: En el penúltimo tiempo siempre está en reposo.

```
fact cond_final {
    reposo.((Elevador.(posicion)).(TO/last.prev)) != none
}
```

Aquí se quiere que en el penúltimo tiempo el elevador siempre esté en reposo porque no tendrá alguna otra señal de alguna petición que todavía falte por cumplir, es decir que al no quedar más peticiones el elevador permanecerá en reposo indicando cual fue el último usuario en dejar.

Las restricciones básicas descritas en el primer modelo se pueden extender para el segundo modelo.

3.3.1. Funciones

Para el primer modelo hablamos de la importancia de tres elementos básicos en el movimiento del elevador, los cuales se refieren a la posición de los elementos que interactúan en este; estos son, el piso en donde se encuentra el elevador, los pisos en donde se encuentran los usuarios y los pisos a donde quieren ir los usuarios. Estos elementos se utilizarán de manera regular a lo largo de toda la especificación, por lo tanto es deseable que obtengamos estos elementos por medio de una función que nos devuelva la posición de alguno en un tiempo dado.

FUNCIONES QUE NOS DEVUELVEN PISOS

Para t:Tiempo, piso_del_elevador[t]= piso donde esta el elevador en el tiempo t.

```
fun piso_del_elevador (t:Tiempo):Piso {
    (Direccion.(Elevador.posicion)).t
}
```

Para t:Tiempo, piso_solicitado[t]= piso a donde quiere ir el usuario en el tiempo t.

```
fun piso_solicitado (u:Usuario,t:Tiempo):Piso {
    ((u.peticion).t).Piso
}
```

Para t:Tiempo, piso_del_usuario[t] = piso en donde esta el usuario en el tiempo t.

```
fun piso_del_usuario (u:Usuario,t:Tiempo):Piso{
    Piso.((u.(peticion.t)))
}
```

Estas funciones serán de gran ayuda al extender la especificación. El objetivo del sistema para modelar el comportamiento de un elevador es recoger a todos los usuarios en determinado momento sin dejar a alguno sin atender, en el tiempo más corto posible. Los usuarios que interactúan en el sistema no comparten las mismas características a lo largo de todo el recorrido del elevador, ya que unos usuarios serán atendidos primero y otros no serán recogidos pues desean seguir otra dirección, esto implica que, en cierto sentido los usuarios tienen una dirección dada. Así podemos considerar a los usuarios como un conjunto formado por diferentes subconjuntos que representan usuarios con características comunes.

A lo largo de la especificación hacemos esta diferencia entre tipos de usuarios para facilitar la interacción entre las reglas que rigen su movimiento; las funciones describen el manejo de los diferentes tipos de usuarios de una manera más sencilla, esto permite la simplificación de los enunciados escritos en lenguaje de alloy. Las funciones propuestas son un medio para obtener los distintos tipos de usuario en un tiempo dado, los cuales son:

1. Usuarios que están en el mismo piso que el elevador y se van a subir a este con dirección hacia abajo.
2. Usuarios que se va a quedar en el mismo piso en el siguiente tiempo, cuando el elevador tiene la dirección hacia abajo.
3. Usuarios que están en el mismo piso que el elevador y se van a subir a este con dirección hacia arriba.
4. Usuarios que se va a quedar en el mismo piso en el siguiente tiempo, cuando el elevador tiene la dirección hacia arriba.
5. Todo usuario que está en el elevador en el tiempo siguiente.
6. Todo usuario que no está en el elevador en el tiempo siguiente.
7. Usuarios que estén en el piso al que querían ir, representa el paso antes de dejar a esos usuarios.
8. Usuarios que no estén en el piso al que quieren ir, en el momento en el que elevador vaya a dejar a algún usuario.
9. Usuarios que no han realizado una petición en un tiempo dado.
10. Todo usuario que está realizado una petición en un tiempo dado.

FUNCIONES QUE NOS DICEN LOS DISTINTOS TIPOS DE USUARIOS

Para t:Tiempo, usuario_en_elevadorAbajo[t] = usuarios que están en el mismo piso que el elevador y se van a mover en el tiempo t (elevador con dirección hacia abajo).

```
fun usuario_en_elevadorAbajo(t:Tiempo):Usuario {
    (peticion.t.(Direccion.(Elevador.posicion)).t).(((Direccion.(Elevador.posicion)).t).prevs )
}
```

Para t:Tiempo, usuario_NoEn_elevadorAbajo[t] = usuarios que se van a quedar en el mismo piso en el siguiente tiempo t (elevador con dirección hacia abajo).

```
fun usuario_NoEn_elevadorAbajo(t:Tiempo):Usuario {
    Usuario - usuario_en_elevadorAbajo(t)
}
```

Para t:Tiempo, usuario_en_elevadorArriba[t] = usuarios que están en el mismo piso que el elevador y se van a mover en el tiempo t (elevador con dirección hacia arriba).

```
fun usuario_en_elevadorArriba(t:Tiempo):Usuario {
    (peticion.t.(Direccion.(Elevador.posicion)).t).(((Direccion.(Elevador.posicion)).t).nexts)
}
```

Para t:Tiempo, usuario_NoEn_elevadorArriba[t] = usuarios que se van a quedar en el mismo piso en el siguiente tiempo t (elevador con dirección hacia arriba).

```
fun usuario_NoEn_elevadorArriba(t:Tiempo):Usuario {
    Usuario - usuario_en_elevadorArriba(t)
}
```

Para t:Tiempo, usuario_en_elevadorSiguenteTiempo[t'] = usuarios que están en el elevador en el siguiente tiempo t'.

```
fun usuario_en_elevadorSiguenteTiempo(t':Tiempo):Usuario {
    (((peticion.t').(Direccion.(Elevador.posicion)).t').Piso)
```

```
}
```

Para t:Tiempo, usuario_NoEn_elevadorSiguenteTiempo[t'] = usuarios que no están en el elevador en el siguiente tiempo t'.

```
fun usuario_NoEn_elevadorSiguenteTiempo(t':Tiempo):Usuario {
    Usuario - usuario_en_elevadorSiguenteTiempo(t')
}
```

Para t:Tiempo, usuario_En_pisoDesado[t] = usuarios tal que estén en el piso al que quieren ir en el tiempo t (Paso antes de dejar a ese usuario).

```
fun usuario_En_pisoDesado(t:Tiempo):Usuario {
    ((peticion.t).(((Usuario.(peticion.t)) & iden).Piso)).((((Usuario.(peticion.t)) & iden).Piso))
}
```

Para t:Tiempo, usuario_NoEn_pisoDesado[t] = usuarios tal que no estén en el piso al que quiere ir en el tiempo t.

```
fun usuario_NoEn_pisoDesado(t:Tiempo):Usuario {
    Usuario - usuario_En_pisoDesado(t)
}
```

Para t:Tiempo, usuario_con_peticion[t] = usuarios que están realizado una petición en un tiempo t.

```
fun usuario_con_peticion(t:Tiempo):Usuario {
    ((peticion.t).Piso).Piso
}
```

Para t:Tiempo, usuario_sin_peticion[t] = usuarios que no han realizado una petición en un tiempo t.

```
fun usuario_sin_peticion(t:Tiempo):Usuario {
    Usuario - usuario_con_peticion(t)
}
```

A lo largo de la especificación se ha mencionado la utilización del módulo ordering como una ayuda para el ordenamiento lineal de átomos que pertenecen a una signatura dada.

Por ejemplo:

open util/ordering [Piso] as PO

En la figura 3.20 vemos un diagrama de los pisos y como están ordenados. A partir de esto, las funciones del módulo *ordering* aplicadas al diagrama nos dan como resultado:

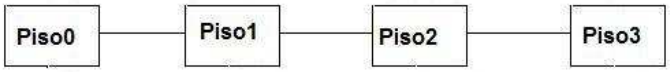

Figura 3.20: Operaciones básicas de ordering.

first = Piso0	last = Piso3
Piso1.next = Piso2	Piso1.prev = Piso0
Piso1.nexts = Piso2 + Piso3	Piso2.prevs = Piso0 + Piso1
lt[Piso2,Piso3] = true	lte[Piso0,Piso0] = true
gt[Piso2,Piso3] = false	gte[Piso2,Piso0] = true
lt[Piso3,Piso3] = false	gte[Piso1,Piso1] = true

Consideremos las abreviaturas para las funciones lt = menor que y lte = menor o igual a, gt = mayor que y gte = mayor o igual a. Las operaciones descritas serán utilizadas en las reglas que describen el movimiento del elevador.

3.3.2. Reglas de movimiento para el segundo modelo

Como para el primer modelo tenemos que definir las reglas que rigen el sistema de elevador para varios usuarios, por ser para varios, su definición es más complicada ya que tenemos que considerar más escenarios y posibles errores que aparezcan. Una vez definidas las funciones que utilizaremos para facilitar la definición de las reglas, al igual que las funciones que ofrece el módulo *ordering*, podemos empezar a definir nuestras reglas. Por la longitud de las reglas descritas el código está numerado para un descripción mas clara.

Regla 0:

La regla 0 describe el movimiento de un elevador, que está en el mismo piso que uno o varios usuarios que quieren ir hacia abajo y teniendo en cuenta que el piso al que quieren ir los usuarios no es el siguiente inmediato. El predicado tiene 7 atributos, el tiempo t que es el antecesor del tiempo t', el piso "x" que es el piso donde está el elevador, "y" el piso al que el usuario quiere ir y "z" es el piso en el que está el usuario todos en el tiempo t. Utilizaremos x', y' y z' para representar en donde estarán en el siguiente tiempo t' , u serán todos aquellos usuarios que va a recoger el elevador y u' serán todos los usuarios que permanecerán en su sitio en el tiempo t', " b " y "c " nos dicen a donde quieren ir los usuarios y en donde están los usuarios respectivamente en el tiempo t pero estos usuarios son aquellos que no están en el elevador en el tiempo t, de igual forma b' y c' nos dicen lo mismo pero de los usuarios que no están en el elevador en el tiempo t'.

Dentro del predicado haremos uso de la expresión *let* que utilizaremos para facilitar el manejo de los tipos de usuarios y de los tipos de pisos.

```
1. pred r0 (t: Tiempo , t': Tiempo, x, y, z:Piso,u,u':Usuario){
2.
3.   let u= usuario_en_elevadorAbajo[t], u'=usuario_NoEn_elevadorAbajo[t],
4.       v=usuario_en_elevadorSiguenteTiempo[t'], v'=usuario_NoEn_elevadorSiguenteTiempo[t'],
5.       k=usuario_sin_peticion[t], r = min[usuario_con_peticion[t]],
6.       x=piso_del_elevador[t], y=piso_solicitado[u,t], z=piso_del_usuario [u,t],
7.       x'=piso_del_elevador[t'], y'=piso_solicitado[v,t'], z'=piso_del_usuario [v,t'],
8.       b=piso_solicitado[u',t], c=piso_del_usuario [u',t],
9.       b'=piso_solicitado[v',t'], c'=piso_del_usuario [v',t'],
10.      d= ((Elevador.posicion).t).Piso, d'= ((Elevador.posicion).t').Piso |
11.
12.          estado[1] and x = z and x not in y
13.          and x.prev not in y and z.prev not in y
14.          and d = abajo
```

15. **and** x != **none and** y != **none and** z != **none and** u != **none**
16. **and** (x = PO/**first.next or** (x != PO/**first.next and** d' = abajo))
17. **and** (v = u + (((peticion.t).(x.**prev**)).Piso) **or** v = u + (((peticion.t).(x.**prev**)).Piso) + k)
18. **and** (((v'.(peticion.t')) = ((u'.(peticion.t)) - (((((peticion.t).(x.**prev**)).Piso)).(peticion.t))))
19. **or** (v'.(peticion.t')) = (((u'.(peticion.t)) - ((((((peticion.t).(x.**prev**)).Piso)).(peticion.t)))) +
20. k.peticion.t') **or** ((v'.(peticion.t')) = (u'.(peticion.t))) **or** ((v'.(peticion.t')) = **none** ->**none**))
21. **and** # (peticion.t)<= # (peticion.t')
22. **and** (peticion.t) - (u->y->x) **in** (peticion.t')
23. **and** x'= x.**prev and** z'=z.**prev**
24. **and** x' = z' **and** y **in** y'
25. **and**((b' **in** b **and** c' **in** c) **or** (b **in** b' **and** c **in** c') **or** (b = b' **and** c = c'))
26. **and** (**lt**[piso_del_usuario [r,t] , piso_del_elevador[t]] **or**
27. (piso_del_usuario [r,t] = piso_del_elevador[t] and **lt** [piso_solicitado[r,t] , piso_del_elevador[t]]))
28. }

Funcionamiento

Es importante mencionar que el objetivo de las reglas es decirnos cual es un posible escenario en un tiempo dado y las consecuencias de este escenario serán representadas en el siguiente tiempo.

En esta primera regla tenemos que asegurarnos de tres cosas importantes: que los usuarios estén en el mismo piso que el elevador (12), que el piso al que quieren ir los usuarios no es el siguiente inmediato (13) y que quieran ir hacia abajo (14). Es importante asegurar que ciertos elementos siempre están presentes y no pueda pasar que no se cumplan (15) o sea que ninguno de los elementos descritos sea vacío. Si tenemos una situación en la que el elevador no esté en primer piso, "Piso1", implica que la dirección en el siguiente tiempo del elevador será hacia abajo si no lo es, será claro que el elevador esta en el primer piso y la dirección en el siguiente tiempo no podrá ser hacia abajo, ya que existe un hecho que impide este tipo de situaciones (16). Ya que sabemos cuáles son los usuarios " u " que están en el elevador, tendremos que asegurar que éstos estén en el siguiente tiempo con tal vez algunos otros que estén esperando en el piso de llegada y serán los " v " (17), ó bien podrá pasar que en el tiempo siguiente llegue un nuevo usuario " z " que realiza una petición (17), ocurrirá lo mismo con los usuarios que no estén en el elevador " u' ", tendrán que mantenerse igual en el tiempo siguiente y serán los " v' ". Si hablamos de usuarios, tenemos que hablar de las peticiones que estos hacen las cuales deben de permanecer iguales, no podrán desaparecer sólo podrán hacerse nuevas peticiones (18,19 y 20) o en el caso de que no existan permanecerán así (20), por lo tanto estas peticiones en el tiempo t serán iguales o menores a las de t' (21), sin embargo habrá algunas peticiones que no permanecerán iguales que serán todas aquellas que va a resolver el elevador, ya que al moverse no existirán mas en el mismo piso (22).

En el siguiente tiempo t' lo único que se habrá movido será el elevador al piso correspondiente que esté abajo, junto con los usuarios que haya recogido (23), manteniéndose el estado de los usuarios hacia donde quieren ir, tanto de los que están en el elevador (24) como de los que no lo están (25). Por último tenemos que decir de alguna manera que el usuario con el número más pequeño " r " será aquel que haya presionado el botón hacia algún piso antes que los otros usuarios (26 y 27), usando ciertos métodos del módulo ordering como ayuda.

En la figura 3.21 vemos un ejemplo del funcionamiento de la regla 0 para el modelo 2. En la que el elevador tiene una dirección hacia abajo, y baja con un usuario un piso que no es al que quiere ir.

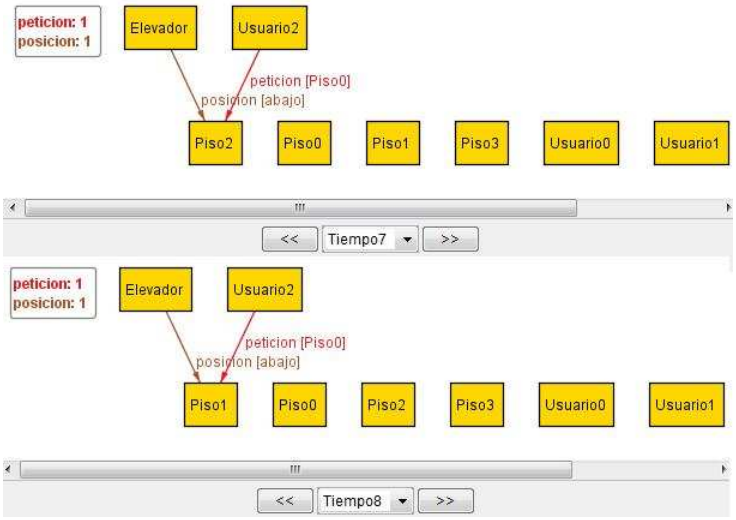

Figura 3.21: Movimiento de la regla 0 para el modelo 2.

Regla 1:

La regla 1 describe el movimiento, de un elevador que está en el mismo piso que uno o varios usuarios, y algún usuario quiere ir hacia arriba y el piso al que quiere ir no es el siguiente inmediato.

1. **pred** r1(t: Tiempo , t': Tiempo, x, y, z:Piso,u,u':Usuario){
2.
3. **let** u= usuario_en_elevadorArriba[t], u'=usuario_NoEn_elevadorArriba[t],
4. v=usuario_en_elevadorSiguenteTiempo[t'], v'=usuario_NoEn_elevadorSiguenteTiempo[t'],
5. k=usuario_sin_peticion[t],r = **min**[usuario_con_peticion[t]],
6. x=piso_del_elevador[t], y=piso_solicitado[u,t], z=piso_del_usuario [u,t],
7. x'=piso_del_elevador[t'],y'=piso_solicitado[v,t'],z'=piso_del_usuario [v,t'],

```
8.      b=piso_solicitado[u',t],c=piso_del_usuario [u',t],
9.      b'=piso_solicitado[v',t'],c'=piso_del_usuario [v',t'],
10.     d= ((Elevador.posicion).t).Piso, d'= ((Elevador.posicion).t').Piso |
11.
12.     estado[1] and x = z and x not in y
13.     and x.next not in y and z.next not in y
14.     and d = arriba
15.     and x != none and y != none and z != none and u != none
16.     and ( x = PO/last.prev or (x != PO/last.prev and d' = arriba) )
17.     and (v =u+(((peticion.t).(x.next)).Piso) or v = u+(((peticion.t).(x.next)).Piso) + k)
18.     and (((v'.(peticion.t'))=((u'.(peticion.t))-(((((peticion.t).(x.next)).Piso)).(peticion.t))))
19.     or (v'.(peticion.t')) = (((u'.(peticion.t)) - ((((((peticion.t).(x.next)).Piso)).(peticion.t))))
20.     + k.peticion.t' ) or ( (v'.(peticion.t')) = (u'.(peticion.t )) )
21.     or ( (v'.(peticion.t')) = none->none ) )
22.     and # (peticion.t )<= #(peticion.t')
23.     and (peticion.t ) - (u->y->x) in (peticion.t')
24.     and x'= x.next and z'=z.next
25.     and x' = z' and y in y'
26.     and(( b' in b and c' in c) or (b in b' and c in c') or (b = b' and c = c') )
27.         and( gt[ piso_del_usuario [r,t] , piso_del_elevador[t] ] or
28.         (piso_del_usuario [r,t]=piso_del_elevador[t] and
29.         gt[piso_solicitado[r,t],piso_del_elevador[t]]))
30. }
```

Funcionamiento

Esta regla es muy similar a la regla 0 con algunas diferencias, la más importante es que la dirección del elevador es hacia arriba. Básicamente es cambiar el método " prev " por " next " (13,17,18,19 y 24), en el caso del renglón (16) se cambia el " first.next " por " last.prev " esto nos dice que si el elevador está en el piso previo al último la dirección en el siguiente tiempo del elevador será hacia abajo ya que no tendría sentido que fuera hacia arriba. Otro cambio es " lt " por " gt" (27 y 28) que una vez más, son métodos del módulo *ordering*, que nos ayudan a seleccionar la prioridad correcta que seguirá el elevador.

En la figura 3.22 vemos un ejemplo del funcionamiento de la regla 1 para el modelo 2. En la que el elevador tiene una dirección hacia arriba y sube con un usuario, un piso que no es al que quiere ir.

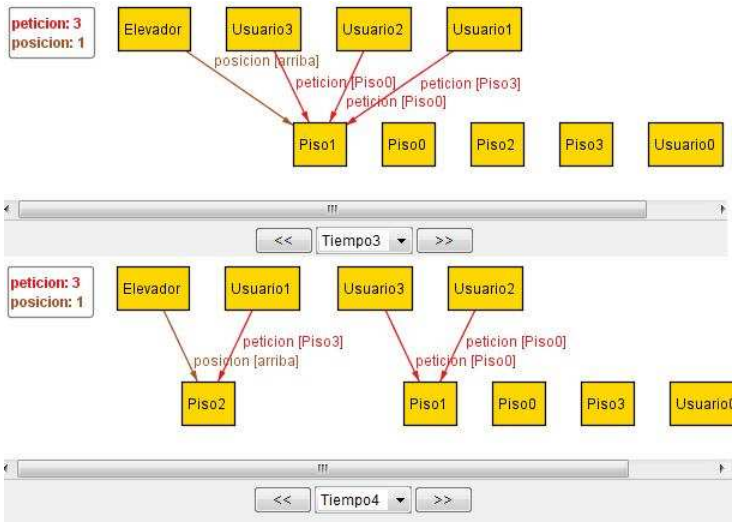

Figura 3.22: Movimiento de la regla 1 para el modelo 2.

Regla 2:

La regla 2 describe el movimiento de un elevador que no está en el mismo piso que un usuario y quiere ir hacia abajo, consta de los mismos atributos y tiene una serie de expresiones *let* al igual que las reglas anteriores.

```
1. pred r2(t: Tiempo , t': Tiempo, x, y, z:Piso,u,u':Usuario){
2.
3.   let u = min[usuario_con_peticion[t]],
4.       u'=usuario_NoEn_elevadorAbajo[t], v =usuario_en_elevadorAbajo[t],
5.       x=piso_del_elevador[t], y=piso_solicitado[u,t], z=piso_del_usuario [u,t],
6.       x'=piso_del_elevador[t'], y'=piso_solicitado[u,t'], z'=piso_del_usuario [u,t'],
7.       b=piso_solicitado[u',t], c=piso_del_usuario [u',t],
8.       b'=piso_solicitado[u',t],c'=piso_del_usuario [u',t],
9.       d= ((Elevador.posicion).t).Piso |
10.
11.          estado[1] and x != z
12.          and (x.prev =x' ) and (y in y') and (z'= z)
13.          and d = abajo
14.          and v= none
15.          and u != none
16.          and (peticion.t ) in (peticion.t')
17.          and # (peticion.t )<= #(peticion.t')
```

```
18.         and (z = min[x + z])
19.         and x != none and y != none and z != none
20.         and b in b' and c in c'
21.
22.     }
```

Funcionamiento

En esta regla el elevador esta vacío (11) , tendrá que ir hacia abajo (13) y mantener el estado del o los usuarios por los que se dirige, es decir el piso en el que está y al que quiere ir (12), aseguramos que el elevador no va a tener ningún usuario (14) al igual que aseguramos que hay un usuario que hizo la petición primero (15), las peticiones que aparecen el tiempo t deberán estar en el t' (16), el número de peticiones en el tiempo t será menor o igual al número en el tiempo t' por la aparición de alguna otra petición (17), el piso en el que está el usuario que va a recoger el elevador, es menor a el piso en donde está el elevador (18) , aseguramos que todo permanezca igual y sin errores (19) y (20) de la misma forma que en las reglas anteriores.

En la figura 3.23 vemos un ejemplo del funcionamiento de la regla 2 para el modelo 2. En la que el elevador está vacío, tiene una dirección hacia abajo y baja un piso.

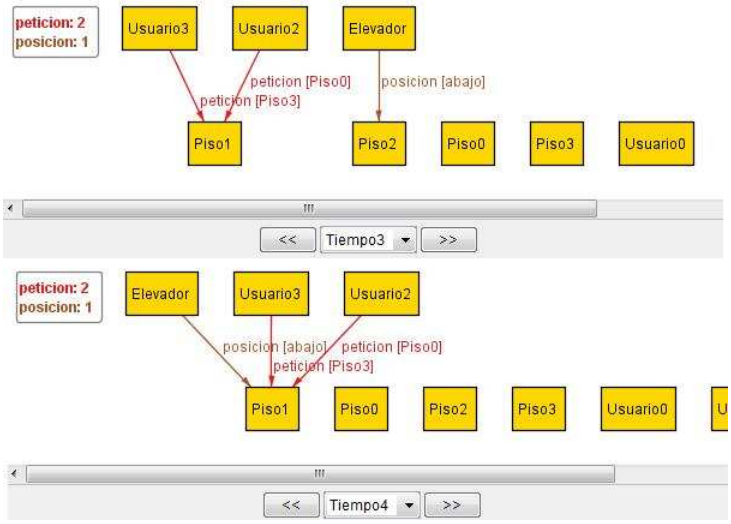

Figura 3.23: Movimiento de la regla 2 para el modelo 2.

Regla 3:

La regla 3 describe el movimiento de un elevador, que no está en el mismo piso que el usuario y quiere ir hacia arriba, consta de los mismos atributos y tiene algunas expresiones *let* al igual que las reglas anteriores.

```
1. pred r3(t: Tiempo , t': Tiempo, x, y, z:Piso,u,u':Usuario){
2.
3.   let u = min[usuario_con_peticion[t]], u'=usuario_NoEn_elevadorAbajo[t],
4.       v =usuario_en_elevadorAbajo[t],
5.       x=piso_del_elevador[t], y=piso_solicitado[u,t], z=piso_del_usuario [u,t],
6.       x'=piso_del_elevador[t'], y'=piso_solicitado[u,t'], z'=piso_del_usuario [u,t'],
7.       b=piso_solicitado[u',t],c=piso_del_usuario [u',t],
8.       b'=piso_solicitado[u',t],c'=piso_del_usuario [u',t],
9.       d= ((Elevador.posicion).t).Piso |
10.
11.          estado[1] and x != z
12.          and (x.next =x' ) and (y in y') and (z'= z)
13.          and d = arriba
14.          and v= none
15.          and u != none
16.          and (peticion.t ) in (peticion.t')
17.          and # (peticion.t ) <= #(peticion.t')
18.          and (x = min[x + z])
19.          and x != none and y != none and z != none
20.          and b in b' and c in c'
21.
22.      }
```

Funcionamiento

Es similar al de la regla 2 solo que el elevador se moverá hacia arriba y por lo tanto la dirección del elevador es hacia arriba (12) y (13), además el elevador estará en un piso menor al que se encuentra el usuario por el que va (18).

En la figura 3.24 vemos un ejemplo del funcionamiento de la regla 3 para el modelo 2. En la que el elevador esta vacío, tiene una dirección hacia arriba y sube un piso.

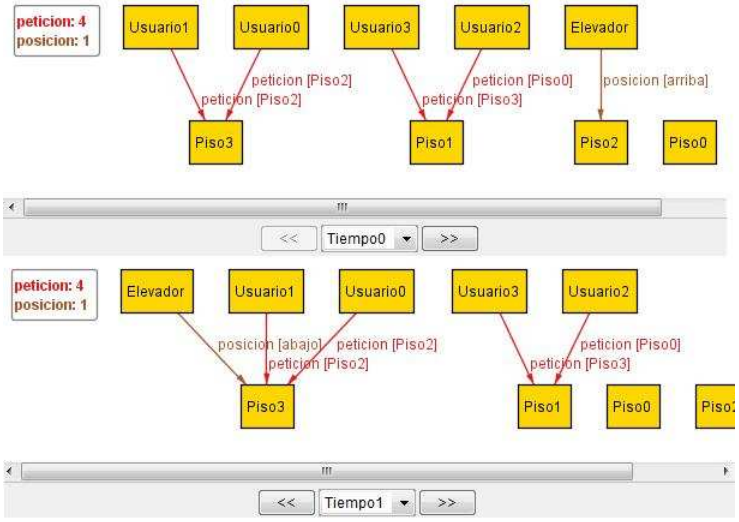

Figura 3.24: Movimiento de la regla 3 para el modelo 2.

Regla 4:

Esta regla describe el último movimiento en el transcurso de la trayectoria del elevador, es el paso antes de dejar al último o a los últimos usuarios, estos permanecerán en reposo por que el elevador los habrá dejado en el tiempo t y en el tiempo t' solo quedará el elevador en reposo.

```
1. pred r4(t: Tiempo , t': Tiempo, x, y, z:Piso,u,u':Usuario){
2.
3.    let u= usuario_con_peticion[t],
4.        x=piso_del_elevador[t], y=piso_solicitado[u,t], z=piso_del_usuario [u,t],
5.        x'=piso_del_elevador[t'], y'=piso_solicitado[u,t'], z'=piso_del_usuario [u,t'],
6.        d= ((Elevador.posicion).t).Piso, d'= ((Elevador.posicion).t').Piso |
7.
8.        estado[1] and x = z
9.        and x in y
10.       and x != none and y != none and z != none
11.       and d = reposo
12.       and x' = x and z'= none and y' = none
13.       and d' = reposo
14.
15.       }
```

Funcionamiento

Esta regla es muy sencilla solo nos dice que el elevador está dejando a los usuarios en el piso que querían (8) y (9), aseguramos que el piso en donde están los usuarios y el elevador exista (10), el elevador tiene una dirección en reposo en el tiempo t (11) que mantiene en el tiempo t' (13), por último el elevador tendrá que estar en el piso sin ningún usuario realizando una petición (12).

En la figura 3.25 vemos un ejemplo del funcionamiento de la regla 4 para el modelo 2. En la que el elevador deja a el último usuario por servir.

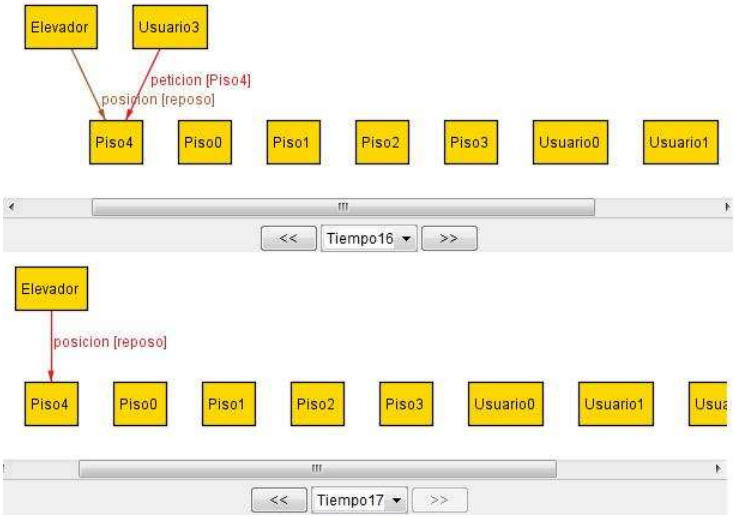

Figura 3.25: Movimiento de la regla 4 para el modelo 2.

Regla 5A:

Esta regla representa al penúltimo tiempo cuando todos las peticiones de los usuarios son atendidas. Es la representación del elevador cuando deja a los últimos usuarios y su posición será en reposo en el tiempo siguiente porque el elevador no tendrá más peticiones por servir.

1. **pred** r5A(t: Tiempo , t': Tiempo, x, y, z:Piso,u,u':Usuario){
2.
3. **let** u= usuario_en_elevadorAbajo[t], u'=usuario_NoEn_elevadorAbajo[t],
4. v=usuario_en_elevadorSiguenteTiempo[t'], v'=usuario_NoEn_elevadorSiguenteTiempo[t'],
5. x=piso_del_elevador[t], y=piso_solicitado[u,t], z=piso_del_usuario [u,t],
6. x'=piso_del_elevador[t'], y'=piso_solicitado[v,t'], z'=piso_del_usuario [v,t'],
7. b=piso_solicitado[u',t], c=piso_del_usuario [u',t],

```
8.      b'=piso_solicitado[v',t'],c'=piso_del_usuario [v',t'],
9.      d= ((Elevador.posicion).t).Piso, d'= ((Elevador.posicion).t').Piso |
11.
13.         estado[1] and u != none and v != none and u = v
14.         and u'= none and v'=none and b= none and c = none
15.         and b = b' and c =c'
16.         and x = z and x not in y
17.         and x.prev = y and z.prev = y
18.         and x != none and y != none and z != none
19.         and d = abajo
20.         and x' = x.prev
21.         and x' != none and y' != none and z' != none
22.         and d' = reposo
23.
24.     }
```

Funcionamiento

Los últimos usuarios que estén por ser atendidos en el tiempo t tendrán que ser los mismos que aparezcan en el tiempo t' y nos aseguramos que estén en el elevador (13); al igual que no debe de haber ningún otro usuario que falte por servir ni en el tiempo t ni en el t' (14). El elevador se encontrará en el piso previo al que quiere ir el usuario en el tiempo t (16) y (17) con la dirección del elevador hacia abajo (19). El elevador en el tiempo t' se encontrará dejando al usuario en el piso deseado (20) con la posición en reposo (22). Las líneas no comentados son iguales a las explicadas anteriormente.

En la figura 3.26 vemos un ejemplo del funcionamiento de la regla 5A para el modelo 2. En la que el elevador baja y está por dejar un usuario al piso que desea.

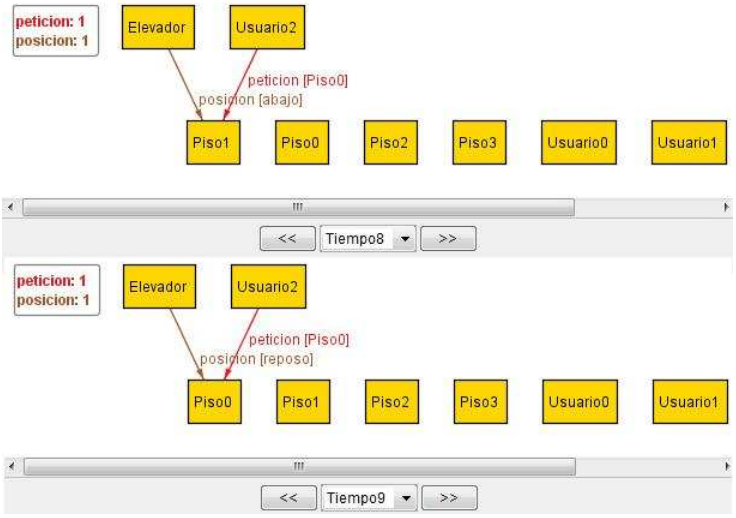

Figura 3.26: Movimiento de la regla 5A para el modelo 2.

Regla 5B:

La regla describe el penúltimo tiempo antes de dejar a los últimos usuarios. Es muy similar a la regla anterior tiene los mismos atributos y expresiones *let* solo cambia la dirección a donde quiere el usuario.

```
1. pred r5B(t: Tiempo , t': Tiempo, x, y, z:Piso,u,u':Usuario){
2.
3.    let u= usuario_en_elevadorAbajo[t], u'=usuario_NoEn_elevadorAbajo[t],
4.        v=usuario_en_elevadorSiguenteTiempo[t'], v'=usuario_NoEn_elevadorSiguenteTiempo[t'],
5.        x=piso_del_elevador[t], y=piso_solicitado[u,t], z=piso_del_usuario [u,t],
6.        x'=piso_del_elevador[t'], y'=piso_solicitado[v,t'], z'=piso_del_usuario [v,t'],
7.        b=piso_solicitado[u',t], c=piso_del_usuario [u',t],
8.        b'=piso_solicitado[v',t'], c'=piso_del_usuario [v',t'],
9.        d= ((Elevador.posicion).t).Piso, d'= ((Elevador.posicion).t').Piso |
10.
11.           estado[1] and u != none and v != none and u = v
12.           and u'= none and v'=none and b= none and c = none
13.           and b = b' and c =c'
14.           and x = z and x not in y
15.           and x.next = y and z.next = y
16.           and x != none and y != none and z != none
17.           and d = arriba
```

```
18.         and x' = x.next
19.         and x' != none and y' != none and z' != none
20.         and d' = reposo
21.
22.     }
```

Funcionamiento

La regla 5B es muy similar a la 5A solo cambia la dirección del elevador en el tiempo t, que es hacia arriba, también consta del mismo tipo de usuarios.

En la figura 3.27 vemos un ejemplo del funcionamiento de la regla 5B para el modelo 2. En la que el elevador sube y está por dejar un usuario al piso que desea.

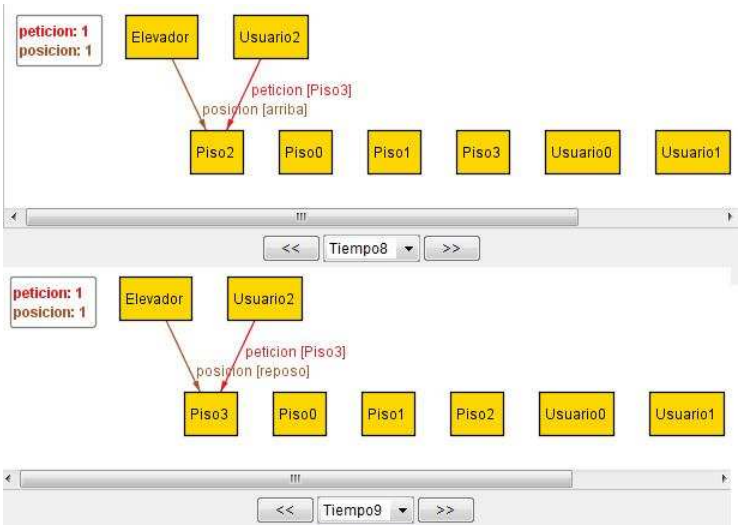

Figura 3.27: Movimiento de la regla 5B para el modelo 2.

Regla 6:

La regla 6 describe el movimiento de un elevador, que está en el mismo piso que uno o varios usuarios, que quieren ir al siguiente piso con dirección hacia abajo, los tipos de usuarios son los mismos a los de la regla 0.

```
1. pred r6 (t: Tiempo , t': Tiempo, x, y, z:Piso,u,u':Usuario){
2.
```

```
3.  let u= usuario_en_elevadorAbajo[t], u'=usuario_NoEn_elevadorAbajo[t],
4.      v=usuario_en_elevadorSiguenteTiempo[t'], v'=usuario_NoEn_elevadorSiguenteTiempo[t'],
5.      w=usuario_en_elevadorSiguenteTiempo[t], k=usuario_sin_peticion[t],
6.      r=min[usuario_con_peticion[t]], x=piso_del_elevador[t], y=piso_solicitado[u,t],
7.      z=piso_del_usuario [u,t], x'=piso_del_elevador[t'], y'=piso_solicitado[v,t'],
8.      z'=piso_del_usuario [v,t'], b=piso_solicitado[u',t], c=piso_del_usuario [u',t],
9.      b'=piso_solicitado[v',t'], c'=piso_del_usuario [v',t'],
10.     d= ((Elevador.posicion).t).Piso, d'= ((Elevador.posicion).t').Piso |
11.
12.     estado[1] and x = z and x not in y
13.     and x != none and y != none and z != none and u != none
14.     and (x = PO/first.next or (x != PO/first.next and d' = abajo) )
15.     and ( v = u + (((peticion.t).(x.prev)).Piso)
16.     or v = u +(((peticion.t).(x.prev)).Piso) + k)
17.     and (((v'.(peticion.t'))=((u'.(peticion.t))-(((((peticion.t).(x.prev)).Piso)).(peticion.t))))
18.     or (v'.(peticion.t'))=(((u'.(peticion.t))-((((((peticion.t).(x.prev)).Piso)).(peticion.t))))
19.     or ((v'.(peticion.t'))=(u'.(peticion.t))) or ((v'.(peticion.t'))=none -> none))
20.     and # (peticion.t )¡= #(peticion.t')
21.     and (peticion.t ) - (u -> y -> x) in (peticion.t')
22.     and x.prev in y and z.prev in y
23.     and d = abajo
24.     and x'= x.prev and z'=z.prev
25.     and x' = z' and y in y'
26.     and b in b' and c in c'
27.     and( lt[ piso_del_usuario [r,t] , piso_del_elevador[t] ] or
28.     (piso_del_usuario[r,t]=piso_del_elevador[t]and lt [piso_solicitado[r,t],piso_del_elevador[t]]))
29.
30.     }
```

Funcionamiento

Es muy similar al de la regla 0 sólo que los usuarios quieren ir al piso inmediato siguiente que está abajo (22). Este tipo de reglas aunque son muy similares (regla 0 y regla 6) se separaron con el objeto de hacer un mejor análisis de lo que ocurre, ya que al realizar la especificación se dividió el problema de manera modular para un mejor entendimiento del movimiento de las reglas que rigen la trayectoria del elevador.

En la figura 3.28 vemos un ejemplo del funcionamiento de la regla 6 para el modelo 2. En ésta vemos al elevador bajar un piso, el cual es al que quiere ir algún usuario que esta en el elevador.

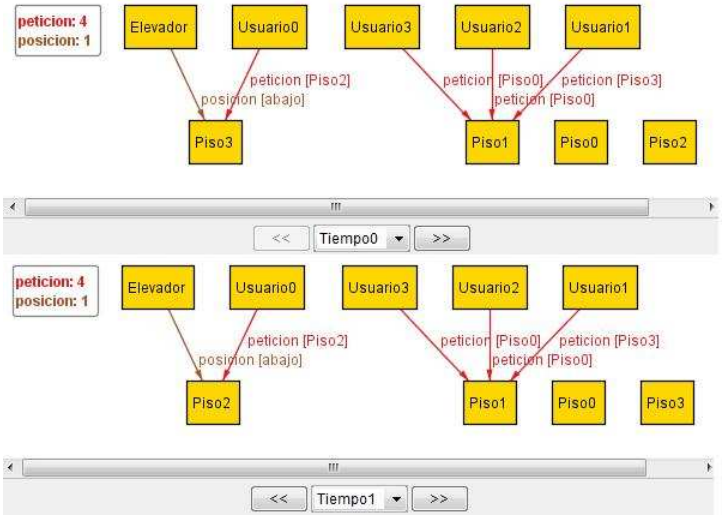

Figura 3.28: Movimiento de la regla 6 para el modelo 2.

Regla 7:

La regla 7 describe el movimiento de un elevador que está en el mismo piso que uno o varios usuarios, que quieren ir hacia arriba y el piso al que quieren ir es el siguiente inmediato. Tiene el mismo tipo de usuarios que la regla 1.

1. **pred** r7 (t: Tiempo , t': Tiempo, x, y, z:Piso,u,u':Usuario){
2.
3. **let** u= usuario_en_elevadorAbajo[t], u'=usuario_NoEn_elevadorAbajo[t],
4. v=usuario_en_elevadorSiguenteTiempo[t'], v'=usuario_NoEn_elevadorSiguenteTiempo[t'],
5. w=usuario_en_elevadorSiguenteTiempo[t], k=usuario_sin_peticion[t],
6. r =**min**[usuario_con_peticion[t]], x=piso_del_elevador[t], y=piso_solicitado[u,t],
7. z=piso_del_usuario [u,t], x'=piso_del_elevador[t'],y'=piso_solicitado[v,t'],
8. z'=piso_del_usuario [v,t'], b=piso_solicitado[u',t],c=piso_del_usuario [u',t],
9. b'=piso_solicitado[v',t'], c'=piso_del_usuario [v',t'],
10. d= ((Elevador.posicion).t).Piso, d'= ((Elevador.posicion).t').Piso |
11.
12. estado[1] **and** x = z and x **not in** y
13. **and** x != **none and** y != **none and** z != **none and** u != **none**
14. **and** (x = PO/**last.prev or** (x != PO/**last.prev and** d' = arriba))
15. **and** (v = u + (((peticion.t).(x.**next**)).Piso)
16. **or** v = u +(((peticion.t).(x.**next**)).Piso) + k)
17. **and** (((v'.(peticion.t'))=((u'.(peticion.t))-(((((peticion.t).(x.**next**)).Piso)).(peticion.t))))

```
18.  or (v'.(peticion.t'))=(((u'.(peticion.t ))-((((((peticion.t).(x.next)).Piso)).(peticion.t))))
19.  or ((v'.(peticion.t'))=(u'.(peticion.t))) or ((v'.(peticion.t')) = none -> none))
20.  and # (peticion.t )¡= #(peticion.t')
21.  and (peticion.t ) - (u -> y -> x) in (peticion.t')
22.  and x.next in y and z.next in y
23.  and d = arriba
24.  and x'= x.next and z'=z.next
25.  and x' = z' and y in y'
26.  and b in b' and c in c'
27.  and( gt[ piso_del_usuario [r,t] , piso_del_elevador[t] ] or
28.  (piso_del_usuario [r,t]=piso_del_elevador[t]
29.      and gt[piso_solicitado[r,t],piso_del_elevador[t]]))
30.  }
```

Funcionamiento

Análogo al de la regla 1 sólo que los usuarios quieren ir al piso inmediato siguiente que este arriba (20).

En la figura 3.29 vemos un ejemplo del funcionamiento de la regla 7 para el modelo 2. En esta vemos al elevador subir un piso.

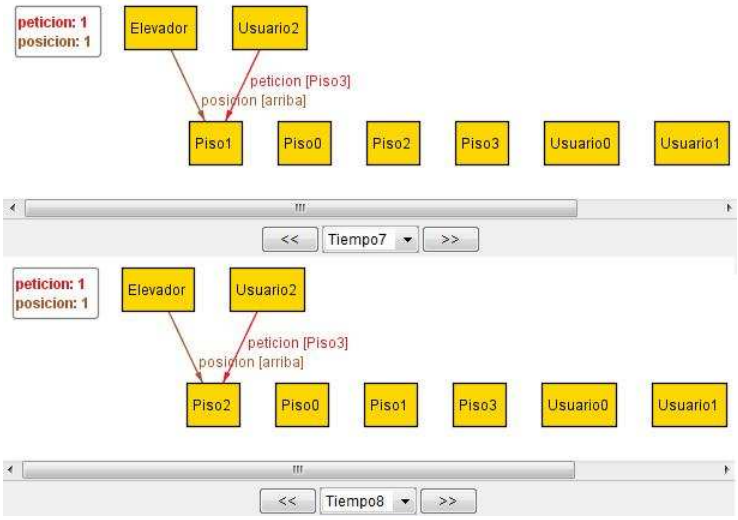

Figura 3.29: Movimiento de la regla 7 para el modelo 2.

Regla 8:

La regla 8 describe el movimiento del elevador, cuando va a dejar a algún usuario en el piso deseado con la dirección del elevador hacia abajo.

```
1. pred r8(t: Tiempo , t': Tiempo, x, y, z:Piso,u,u':Usuario){
2.
3.   let u=usuario_En_pisoDesado[t],u'=usuario_NoEn_pisoDesado[t],
4.       w=usuario_En_pisoDesado[t'],
5.       v'=usuario_NoEn_pisoDesado[t'], v=usuario_en_elevadorSiguenteTiempo[t'],
6.       x=piso_del_elevador[t], y=piso_solicitado[u,t], z=piso_del_usuario [u,t],
7.       x'=piso_del_elevador[t'], y'=piso_solicitado[w,t'], z'=piso_del_usuario [w,t'],
8.       b=piso_solicitado[u',t], c=piso_del_usuario [u',t],
9.       b'=piso_solicitado[v',t'], c'=piso_del_usuario [v',t'],
10.      d= ((Elevador.posicion).t).Piso, d'= ((Elevador.posicion).t').Piso |
11.
12.          estado[1] and x = z
13.          and x in y
14.          and x != none and y != none and z != none
15.          and u != none
16.          and (peticion.t ) - (u -> y ->x) in (peticion.t')
17.          and # ((peticion.t ) - (u ->y ->x)) <= #(peticion.t')
18.          and u.peticion.t' = none -> none
19.          and v = ( (((peticion.t').(x)).Piso) )- u
20.          and # (u'.(peticion.t )) <= # (v'.(peticion.t'))
21.          and d = abajo
22.          and x'= x and z' = none and y' =none
23.          and b in b' and c in c'
24.          and d' = abajo
25.
26.      }
```

Funcionamiento

Al dejar a algún usuario, el número de peticiones en el tiempo t será diferente al número de peticiones en t' ya que al menos una de las peticiones se habrá servido (17), por lo tanto, las peticiones de los usuarios que están en el elevador en el tiempo t no estarán en el tiempo t' (18); " v "será todo usuario que está en el elevador en el tiempo t' menos " u ", todo usuario tal que esté en el piso al que quiere ir (19). El número de peticiones en el tiempo t' será mayor o igual a las

que existían originalmente en t pero con la opción de que pueden haber nuevas peticiones de otros usuarios nuevos (20). El elevador tendrá una dirección hacia abajo en el tiempo t y la mantendrá en el tiempo t' ya que sólo se verá la representación de dejar a los usuarios (21) y (22). Las líneas no comentadas son iguales a las ya comentadas en otras reglas.

En la figura 3.30 vemos un ejemplo del funcionamiento de la regla 8 para el modelo 2. En la que el elevador baja y deja un usuario en el piso que quiere, pero todavía hay peticiones por servir.

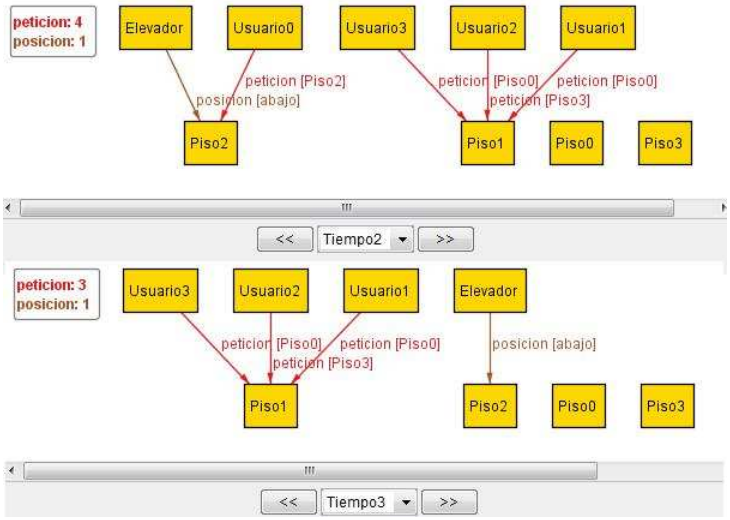

Figura 3.30: Movimiento de la regla 8 para el modelo 2.

Regla 9:

La regla 9 describe el movimiento del elevador, cuando va a dejar a algún usuario en el piso deseado con dirección hacia arriba.

```
1. pred r9(t: Tiempo , t': Tiempo, x, y, z:Piso,u,u':Usuario){
2.
3.   let u=usuario_En_pisoDesado[t], u'=usuario_NoEn_pisoDesado[t],
4.       w=usuario_En_pisoDesado[t'],
5.       v'=usuario_NoEn_pisoDesado[t'], v=usuario_en_elevadorSiguenteTiempo[t'],
6.       x=piso_del_elevador[t], y=piso_solicitado[u,t], z=piso_del_usuario [u,t],
7.       x'=piso_del_elevador[t'], y'=piso_solicitado[w,t'], z'=piso_del_usuario [w,t'],
8.       b=piso_solicitado[u',t], c=piso_del_usuario [u',t],
9.       b'=piso_solicitado[v',t'], c'=piso_del_usuario [v',t'],
10.      d= ((Elevador.posicion).t).Piso, d'= ((Elevador.posicion).t').Piso |
11.
12.          estado[1] and x = z
13.          and x in y
14.          and x != none and y != none and z != none
15.          and u != none
16.          and (peticion.t ) - (u -> y ->x) in (peticion.t')
17.          and # ((peticion.t ) - (u -> y- > x)) <= #(peticion.t')
18.          and u.peticion.t' = none -> none
19.          and v = ( (((peticion.t').(x)).Piso) )- u
20.          and # (u'.(peticion.t )) <= # (v'.(peticion.t'))
21.          and d = arriba
22.          and x'= x and z' = none and y' =none
23.          and b in b' and c in c'
24.          and d' = arriba
25.
26.      }
```

Funcionamiento

Es análogo al de la regla 8 salvo que la dirección es hacia arriba.

En la figura 3.31 vemos un ejemplo del funcionamiento de la regla 9 para el modelo 2. En la que el elevador sube y deja un usuario en el piso que quiere, pero todavía hay peticiones por servir.

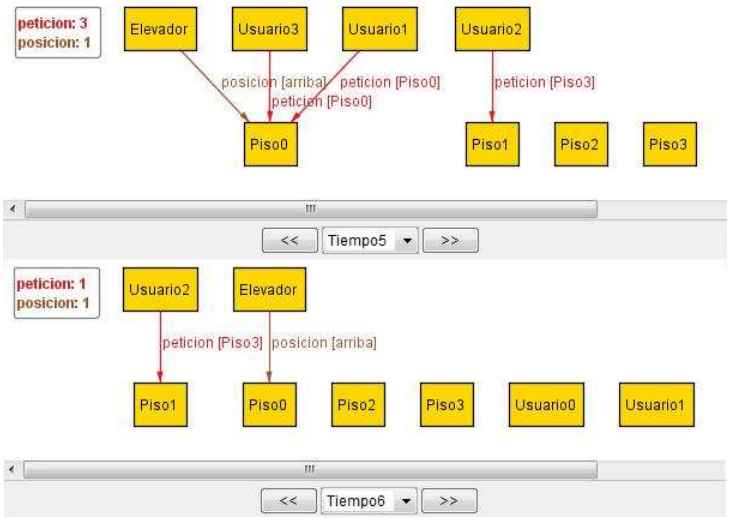

Figura 3.31: Movimiento de la regla 9 para el modelo 2.

3.3.3. Traza para el segundo modelo

Como se dijo para el primer modelo y por la definición de una traza tenemos que escribir un predicado de inicio, en el segundo modelo tenemos que dar dos condiciones de inicio, una será con respecto al tiempo y otra con respecto a los usuarios, habrá dos porque no solo tenemos que ordenar el tiempo sino también la acción de los usuarios de presionar primero un botón.

Predicados de inicio para la traza

```
1.  pred init [t:Tiempo] {
2.      all u:Usuario - usuario_sin_peticion[t] |
3.          ((u.peticion).t).Piso != Piso.((u.(peticion.t)))
4.
5.      }
```

La condición inicial con respecto a los usuarios es que para todo usuario que no pertenezca al conjunto de usuarios que todavía no realizan una petición (2), en el primer tiempo nunca pasa que quieran ir al piso en el que están (3).

Predicado de inicio con respecto a los usuarios.

```
1.  pred init [u:Usuario] {
2.      u=U/first and u!= none
3.          and u in usuario_con_peticion[TO/first]
4.
5.      }
```

La condición inicial con respecto a los usuarios es que el primer usuario (Usuario0) siempre aparece en el primer tiempo de la instancia mostrada (2) y además de existir, tiene que estar realizando una petición en el primer tiempo (3).

A partir de las condiciones de inicio y de la regla general para hacer una traza se hace una que une todos los predicados que rigen el movimiento de un elevador para varios usuarios, con una particularidad, no sólo es con respecto del tiempo sino también con respecto a los usuarios, formando así una de doble traza.

Considero importante resaltar que la propuesta de una doble traza que se presenta arriba en este trabajo, hasta donde se, es original. En los trabajos consultados la atención se centra en ordenar solo un conjunto, mientras que mi propuesta ordena mas de uno, generalmente lo que suele hacerse es elegir el conjunto que nos interesa ordenar, este será con el cual se hará la proyección y nos permitirá simular el dinamismo de nuestro modelo. Con ayuda del modulo *ordering* y por medio de la definición general de una traza tomamos este conjunto y lo utilizamos para realizar una traza, utilizando este concepto nos interesa no solo ordenar al tiempo bajo el cual se hace la proyección sino también ordenaremos al conjunto de usuarios por medio de la política ya descrita.

```
fact trace {
    init[TO/first]
    init1[U/first]
        all t:Tiempo - last | let t'= t.next |
            all x,y,z:Piso |
                all u:Usuario | let u'= u.next |
                    r0[t,t',x,y,z,u,u'] or r1[t,t',x,y,z,u,u'] or r2[t,t',x,y,z,u,u'] or r3[t,t',x,y,z,u,u']
                        or r4[t,t',x,y,z,u,u'] or r5A[t,t',x,y,z,u,u'] or r5B[t,t',x,y,z,u,u']
                            or r6[t,t',x,y,z,u,u'] or r7[t,t',x,y,z,u,u']
                                or r8[t,t',x,y,z,u,u'] or r9[t,t',x,y,z,u,u']
        }
```

En las siguientes figuras visualizamos una instancia completa, la cual nos muestra un modelo en el que hay cuatro pisos y cuatro usuarios, estos usuarios hacen peticiones a tres distintos pisos, el elevador responde a todas utilizando una política no tonta, esto quiere decir que el elevador responderá a las peticiones en orden, donde el usuario mas pequeño y mas cercano será aquel que tenga la prioridad mas alta.

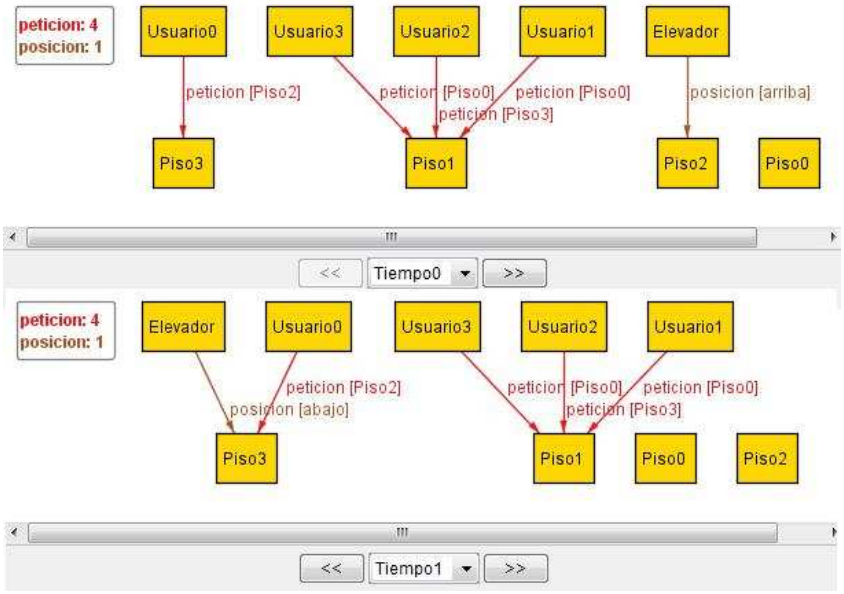

Figura 3.32: Ejemplo de un traza completa del elevador parte 1.

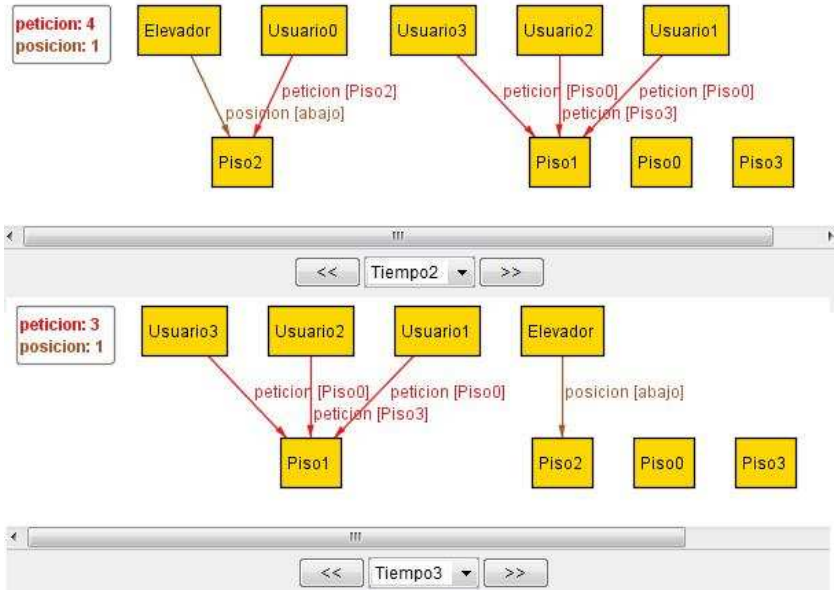

Figura 3.33: Ejemplo de un traza completa del elevador parte 2.

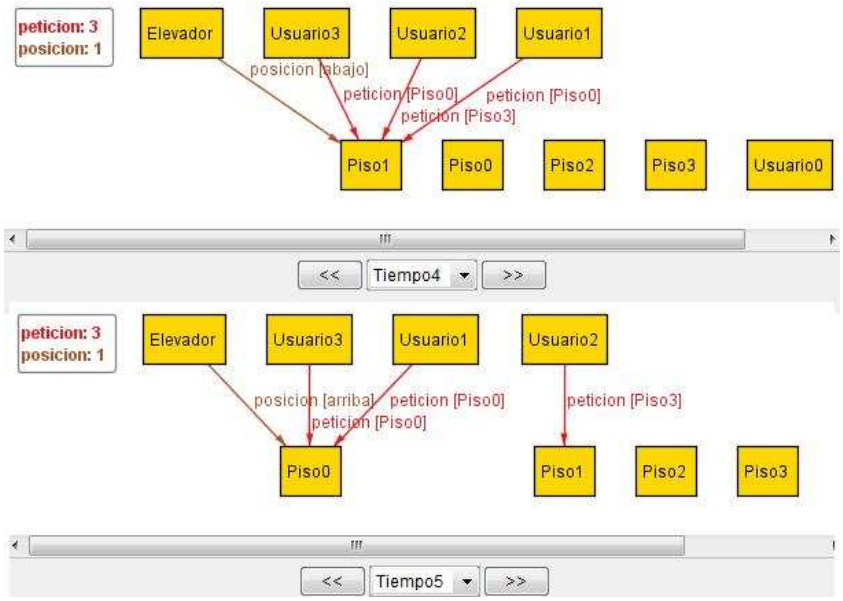

Figura 3.34: Ejemplo de un traza completa del elevador parte 3.

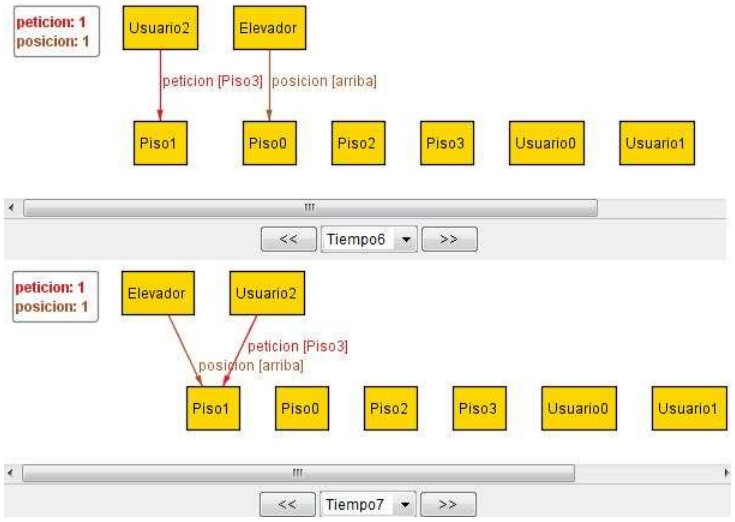

Figura 3.35: Ejemplo de un traza completa del elevador parte 4.

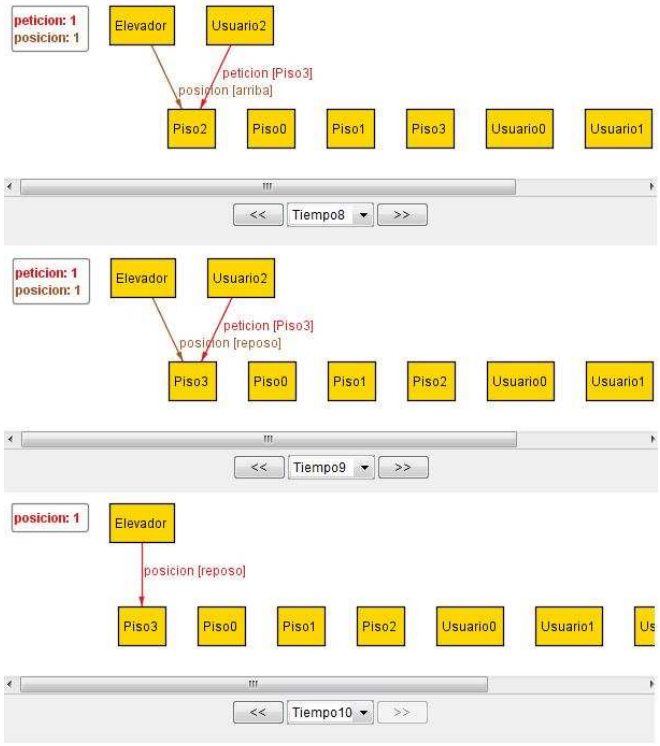

Figura 3.36: Ejemplo de un traza completa del elevador parte 5.

3.3.4. Resultado de la traza

Esta traza aunque es correcta está incompleta ya que únicamente funciona para un número de usuarios mayor a uno, esto es por la definición que se dio para cada predicado, la cual contiene dos atributos con respecto a los usuarios, que nos dice los tipos de usuarios que hay; pero al pedirle al analizador que solo haya un usuario la traza no encuentra ninguna instancia y por lo tanto el modelo está incompleto.

En el primer modelo se implementa el caso para un usuario, por lo que tenemos dos trazas una para un usuario y otra para dos o más, pareciera entonces que simplemente tenemos que agregar las reglas del primer modelo al segundo, sin embargo, al hacerlo el analizador no funciona de manera correcta, por lo que hay que buscar la manera de resolver este problema para fusionar los modelos y obtener una especificación más completa.

Para resolver este problema se creó un predicado auxiliar, el cual le permite a cada predicado encargado del funcionamiento del elevador saber cual es el estado que tiene que tener para que funcione de manera correcta, esto requiere el uso de alguna bandera en cada predicado que indique si debe funcionar para uno o más usuarios.

```
1.  pred estado(r:Int) {
2.      r =1 implies #Usuario >1
3.      r =0 implies #Usuario =1
4.
5.      }
```

El predicado tiene un atributo “r” que es un entero que solo puede ser 0 ó 1 en donde uno implica que el número de usuarios es mayor a uno y a cero indica que hay un único usuario.

Al poner este valor en cada predicado de ambos modelos diferenciándolos ya sea por *estado[0]* ó *estado[1]* en la traza se podrán agregar las reglas del primer modelo para un usuario, la cual quedará de la siguiente forma.

```
fact trace {
    init[TO/first]
    init1[U/first]
        all t:Tiempo - last | let t'= t.next |
            all x,y,z:Piso |

–Reglas para dos o más usuarios.
                all u:Usuario | let u'= u.next |
                    r0[t,t',x,y,z,u,u'] or r1[t,t',x,y,z,u,u'] or r2[t,t',x,y,z,u,u'] or r3[t,t',x,y,z,u,u']
                        or r4[t,t',x,y,z,u,u'] or r5A[t,t',x,y,z,u,u'] or r5B[t,t',x,y,z,u,u']
                            or r6[t,t',x,y,z,u,u'] or r7[t,t',x,y,z,u,u']
                                or r8[t,t',x,y,z,u,u'] or r9[t,t',x,y,z,u,u']
–Reglas para un usuario.
                    ru0[t,t',x,y,z,u,u'] or ru1[t,t',x,y,z,u,u'] or ru2[t,t',x,y,z,u,u'] or ru3[t,t',x,y,z,u,u']
                        or ru4[t,t',x,y,z,u,u'] or ru5[t,t',x,y,z,u,u']
                            or ru6[t,t',x,y,z,u,u']
        }
```

Los predicados para el primer modelo son análogos a los del segundo modelo sólo que tiene dos atributos más “ u ” y “ u' ” en donde “ u ” es el único usuario y “ u' ” es un usuario que no existe, pero ¿ por qué tenemos que poner este usuario que no existe ? Este es resultado de la definición de la regla general de una traza que nos dice que siempre existirán los mismos atributos en todos los predicados.

A diferencia del primer modelo en el que especificamos en cada *let* los pisos “ x, y y z ” , utilizamos las funciones que nos dan el piso correspondiente para cada “ x, y y z ”.

```
pred ru0 (t: Tiempo , t': Tiempo, x, y, z:Piso,u,u':Usuario){
    let u=Usuario ,u'= none,
        x=piso_del_elevador[t], y=piso_solicitado[u,t], z=piso_del_usuario [u,t],
        x'=piso_del_elevador[t'], y'=piso_solicitado[u,t'], z'=piso_del_usuario [u,t'],
        d= ((Elevador.posicion).t).Piso |

            estado[0]
            and x = z and y != x
            and (y = min[x + y])
            and x != none and y != none and z != none
            and d = abajo
            and y != x.prev and y != z.prev
            and x'.next = x and z'.next = z
            and x' = z' and y' = y
            and u'= none
            }
```

Las reglas completas para un usuario tomadas del primer modelo y arregladas para el segundo modelo se muestran en el apéndice.

3.3.5. Restricciones para el segundo modelo

Para el segundo modelo existen una serie de restricciones nuevas que debemos implementar:

Restricción: En el penúltimo tiempo las peticiones que faltan por atender deber ser al mismo piso.

```
fact cond_final {
    #(((((peticion.(TO/last.prev)).
        (piso_del_elevador[TO/last.prev])).Piso).peticion).(TO/last.prev)).Piso=1
}
```

Restricción: En los tiempos posteriores al primero al haber peticiones dinámicas, aseguramos que ningún usuario realice una petición al mismo piso en el que se encuentra.

```
fact cond_usuario {
    all t:Tiempo | all k:usuario_sin_peticion[t] |
        k.peticion.(t.next) != none -> none implies
```

```
        (k.peticion.(t.next) & iden )= none -> none
}
```

Restricción: Un usuario que ha sido servido por el elevador no podrá subir al elevador otra vez en la misma instancia. (Esto evita que el elevador nunca termine).

```
fact cond_usuario {
    all t:Tiempo | all u:Usuario |
        piso_del_elevador[t] = piso_solicitado[u,t] and piso_del_elevador[t] = piso_del_usuario [u,t]
        implies piso_solicitado[u,t.nexts] = none and piso_del_usuario [u,t.nexts] = none
}
```

3.3.6. Análisis

Como se hizo para el primer modelo verificamos la siguientes afirmaciones y verificaremos que pasaría si no estuvieran.

Aserción: En los tiempos posteriores al primero al haber peticiones dinámicas, aseguramos que ningún usuario realice una petición al mismo piso en el que se encuentra.

```
assert pet_dinámica_correcta {
all t:Tiempo | all k:usuario_sin_peticion[t] |
    k.peticion.(t.next) != none -> none implies
    (k.peticion.(t.next) & iden )= none -> none
        }
```

En la figura 3.37 vemos un ejemplo de la aserción pet_dinámica_correcta para el modelo 2. La cual nos muestra que pasaría sino incluyéramos este, podemos ver como la petición del Usuario2 en el tiempo Tiempo2 es hacia el piso en el que ya está.

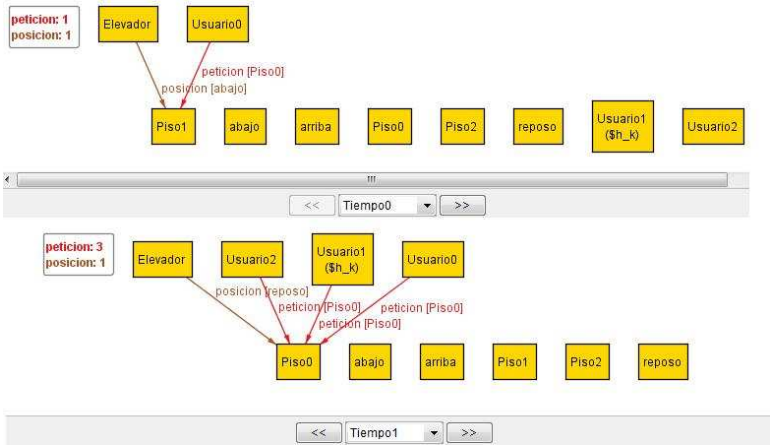

Figura 3.37: Ejemplo de la aserción.

Preguntas al analizador.

¿Pasará que exista algún usuario que nunca es servido al piso que quiere ir?

```
assert no_servido {
    not(all u:Usuario | all t:Tiempo |
        (u.peticion.t) =none -> none
```

}

El analizador contesta que no encuentra algún contraejemplo por lo tanto esto no sucede.

¿ Existirá una instancia en la que el número de peticiones a pisos distintos que hagan los usuarios sea igual a cuatro, tomando en cuenta el alcance descrito en el comando *check* ?

assert peticion_distinta_igual_cuatro {
not(#(((((peticion.Tiempo).
(Direccion.(Elevador.posicion)).Tiempo).Piso).peticion).Tiempo).Piso=4
and # (peticion.((TO/**first**))).Piso=4)
}

check d **for** 5 Piso, 18 Tiempo, 4 Usuario, 1 Elevador

En la figura 3.38 vemos un ejemplo de la aserción peticion_distinta_igual_cuatro para el modelo 2.

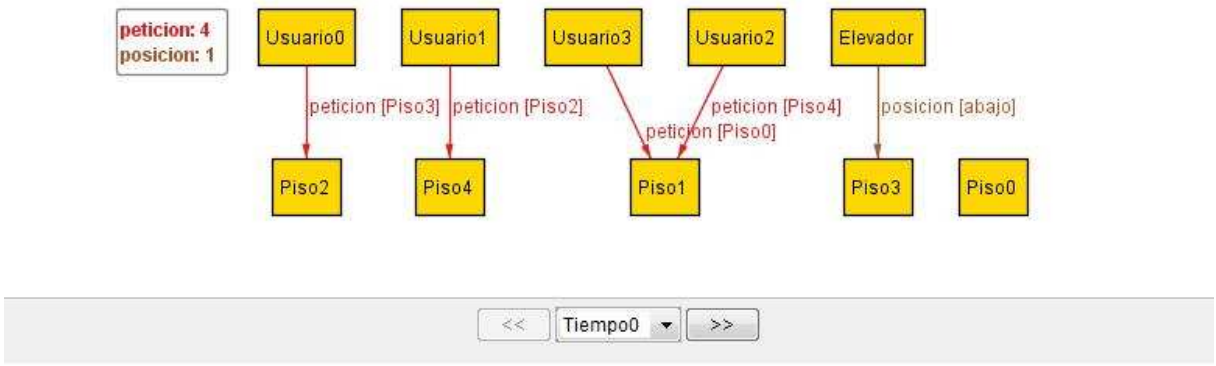

Figura 3.38: Ejemplo de la aserción.

La instancia anterior nos muestra un contraejemplo a la aserción, es decir, si existe una instancia que cumple esas características.

¿ Pasará que en el tiempo t en el que el elevador esté por dejar a un usuario al piso al que quiere ir, en el siguiente tiempo t' no lo deje (paso antes de hacerlo) ?

assert no_dejar_usuario {
not(**all** t:Tiempo | **all** u:Usuario |
piso_del_elevador[t] = piso_solicitado[u,t] and piso_del_elevador[t] = piso_del_usuario [u,t]

implies
piso_solicitado[u,t.**next**] != **none and** piso_del_usuario [u,t.**next**] != **none**)
}

check g

El analizador contesta que no encuentra algún contraejemplo por lo tanto no puede pasar esto.

¿ Pasará que el número de peticiones en pisos distintos es igual a cuatro cuando solo se tienen cuatro pisos ?

assert pisos_distintos_igual_numero_pisos {
not(# Piso.((Usuario.(peticion.TO/**first**))) =4 **and** # Usuario > 1)
}

check k **for** 4 Piso, 11 Tiempo, 4 Usuario, 1 Elevador

En la figura 3.39 vemos un ejemplo de la aserción pisos_distintos_igual_numero_pisos para el modelo 2.

Figura 3.39: Ejemplo de la aserción.

La instancia anterior nos muestra un contraejemplo a la aserción, es decir, si existe una instancia que cumple esas características.

¿Podrá subir un usuario que ha sido servido en algún tiempo, en la misma instancia?

```
assert subir_usuario_servido {
    not(all t:Tiempo | all u:Usuario |
        piso_del_elevador[t] = piso_solicitado[u,t] and piso_del_elevador[t] = piso_del_usuario [u,t]
        implies
        piso_solicitado[u,t.nexts] != none and piso_del_usuario [u,t.nexts] != none)
    }
```

Dice que no existe para todo tiempo y para todo usuario que ha sido servido en algún tiempo, un momento en el que ese usuario vuelva a subir al elevador en la misma instancia, ya que recordemos que esto se prohíbe como hecho.

¿Podrá existir algún tiempo en el que cualquier usuario no es llevado al piso al que quiere ir?

```
assert usuario_no_va_piso {
    not(all u:Usuario | some t:Tiempo |
            (u.peticion.t) = none -> none and t != TO/last )
    }
```

El analizador nos dice que no, lo cual significa que todo usuario es dejado en algún momento.

¿ Si algún usuario desea ir hacia abajo, está en el elevador y el paso correcto siguiente es bajar, el elevador podrá ir para arriba ?

```
assert elevador_dir_incorrecta {
    not(one t:Tiempo,u:Usuario |
            lt[piso_solicitado[u,t],piso_del_usuario [u,t] ] and
                Elevador.(((posicion.TO/first).Piso)) = arriba)
    }
```

El analizador nos dice que no existe un contraejemplo, por lo tanto no pasa esto.

¿Existirá un usuario que nunca suba al elevador?

```
assert usuario_no_sube {
    not(one u:Usuario |
            usuario_sin_peticion[TO/first] != none and u != none and
```

```
            u in usuario_sin_peticion[TO/first]
                implies u in usuario_sin_peticion[(TO/first).nexts] )
    }
```

El analizador nos dice que no existe un contraejemplo, por lo tanto no pasa esto.

¿ Qué pasaría si no pusiéramos como hecho la siguiente aserción?

En el penúltimo tiempo, el número de peticiones a pisos distintos es mayor a uno, lo cual no es correcto y por lo tanto es necesario poner esto como hecho para que no pase lo contrario.

```
assert numero_peticiones_ultimoTiempo {
    #(((((peticion.(TO/last.prev)).
        (piso_del_elevador[TO/last.prev])).Piso).peticion).(TO/last.prev)).Piso=1
    }
```

En la figura 3.40 vemos un ejemplo de que pasaría si no pusiéramos la aserción numero_peticiones_ultimoTiempo para el modelo 2.

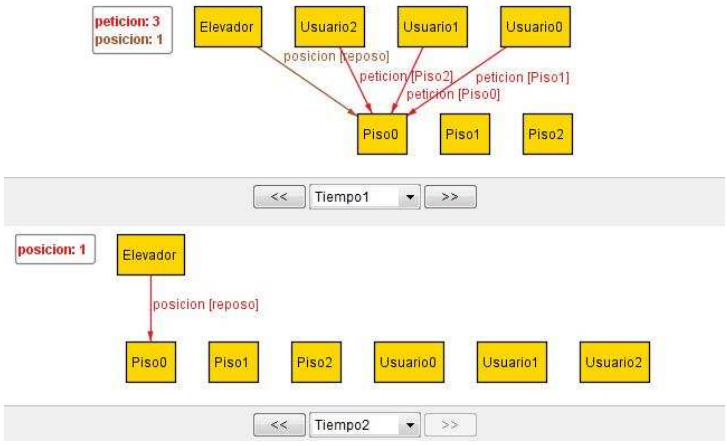

Figura 3.40: Ejemplo de la aserción.

Por lo tanto es un hecho necesario para el segundo modelo.

¿ Existirá un usuario que suba al elevador y nunca baje ?

Esto no puede pasar y puede verse en el *assert* g, que nos expone un ejemplo en el que hay un usuario en el elevador y al preguntarle al analizador si podrá o no dejarlo, o sea que nunca baje del elevador veremos que nos contesta que esto no es posible.

¿ Si hay usuarios en medio del trayecto subirán o no ?

Depende, puede o no hacerlo ver *assert* d for 5 Piso, 18 Tiempo, 4 Usuario, para ver un ejemplo el cual muestra que pueden subir o no subir dependiendo de la dirección que el elevador lleve.

¿ El elevador permanecerá en reposo cuando hay usuarios con petición ?

Esto no pasa porque se forzó a a la especificación a no hacerlo por simple convención, ya que también no tendrá sentido que un elevador permanezca en reposo cuando hay usuarios por servir.

¿ El elevador se moverá sin subir usuarios ?

Depende, puede o no hacerlo ver *assert* d for 5 Piso, 18 Tiempo, 4 Usuario, para ver un ejemplo el cual muestra en que caso si subirán o no a los usuarios, en ningún caso el elevador podrá moverse sin recoger nunca usuarios para alguna instancia.

¿Puede haber peticiones en el último tiempo ?

Esto no pasa porque se forzó a la especificación a no hacerlo, ya que se pensó que para toda instancia queremos ver un ejemplo de un caso particular el cual queremos ver que termine.

Como podemos ver el análisis es una parte fundamental de cualquier especificación en Alloy, este nos permite encontrar algún error en nuestro modelo para un alcance dado, permitiéndonos corregirlo o mostrando que lo que escribimos es correcto y dice o hace lo que pensamos. Gracias a esto, podemos hacer crecer un modelo progresivamente.

4: Conclusiones

El problema del elevador en la presente trabajo, se resuelve, gracias a una serie de reglas declarativas que nos permiten saber cual es el funcionamiento correcto de un elevador. En ninguno de los artículos que aparecen en la bibliografía, y resuelven este problema, se utilizan reglas declarativas, en cambio usan otros enfoques como lógica temporal (Wood 1989), o lógica difusa (Khiang-Khalid-Yusof 2007); en este último artículo se plantea que existen dos formas de resolver el problema de los elevadores, una es con lógica difusa que es la presentada por el artículo y la otra solo mencionada, es por medio de reglas declarativas, pero esté nos dice lo complicado que resulta esto.

¿Por qué seguimos aquí el procedimiento de reglas declarativas ? Usamos reglas declarativas por la forma en la que funciona *Alloy* ya que para usar un algoritmo para lógica difusa, como lo propone el artículo, tendríamos que definir reglas dándoles a cada una de estas un peso determinado, que tiene que ver con la prioridad de que suceda cierto escenario, y en vista del uso de números no enteros, entramos en conflicto con el analizador. Entonces es natural preguntarse porque hacerlo de esta manera; la respuesta es que al usar *Alloy* estamos asegurándonos que la abstracción que hicimos es correcta para un alcance dado, esto es de gran ayuda ya que nos permite darnos cuenta de la fortaleza de la idea propuesta; además el hecho de usar un método formal ligero permite resolver el problema de una manera mas rápida, ya que se fueron encontrando resultados al mismo tiempo que se iba realizando la especificación, pudiendo observar tanto el crecimiento como los defectos de esta, permitiéndonos conocer la solidez de nuestro modelo por el grado de seguridad que nos ofrece *Alloy*.

Una vez que sabemos la veracidad del modelo para un alcance dado, en este caso el sistema de elevador, podemos empezar a construirlo para un caso concreto considerando la alta fiabilidad de la idea.

Gracias al analizador se puede observar una instancia generada para un alcance dado la cual nos permiten ver un caso particular del funcionamiento del elevador, este sigue una trayectoria dada por la serie de reglas que se dieron, pudiendo solo seguir alguna de estas reglas de un tiempo dado al siguiente, gracias a la proyección sobre el tiempo que hacemos; esto nos permite ver el dinamismo de la especificación.

Por medio de un traza unimos todas las reglas que son necesarias para el funcionamiento de un elevador con múltiples usuarios y pisos, esta traza la usamos para el ordenamiento de estos elementos con respecto al tiempo, pero también la usamos para el ordenamiento de los usuarios haciendo que esta traza sea en realidad dos trazas escritas como una, gracias a un predicado auxiliar que nos lo permite. En ningún otro modelo investigado se encontró una traza como está, que es más compleja

que las descritas para otros modelos.

Originalmente quise resolver el problema para varios elevadores, pero esto se volvió una tarea muy difícil para realizarla solo con reglas declarativas, esto no quiere decir que el problema para varios elevadores no pueda realizarse por medio de reglas declarativas, sino que la dificultad del problema crece, ya que no solo hay que decirle a la traza como ordenar a los usuarios y al tiempo sino también como ordenar las peticiones a las que tendrían que responder los elevadores. Esto quiere decir que tendríamos que redefinir las reglas con una política adecuada para varios elevadores capaz de resolver el problema de la manera más adecuada y cercana a la realidad, esta linea podría seguirse en un trabajo futuro, sin embargo el estudio de caso funciona para el número de usuarios y pisos que permite el analizador, ya que esté es solo un micromodelo, para casos en los que haya muchos usuarios o pisos el analizador tardará mucho o no podrá enfrentar el problema; sin embargo, nos permitirá saber con certeza que lo que estamos haciendo esta bien. Por todo esto considero que la solución presentada en el presente trabajo es original.

Apéndice A: Código fuente

A.1. Líneas del metro de la ciudad de Londres

```
abstract sig Station{
    jubilee, central, circle: set Station
}

sig Jubilee, Central, Circle in Station{}

one sig
    Stanmore, BakerStreet, BondStreet, Westminster, Waterloo,
    WestRuislip, EalingBroadway, NorthActon, NottingHillGate,
        LiverpoolStreet, Epping
    extends Station{}

fact Init{
    Jubilee = Stanmore + BakerStreet + BondStreet + Westminster + Waterloo
    Central = WestRuislip + EalingBroadway + NorthActon + NottingHillGate + BondStreet +
        LiverpoolStreet + Epping
    Circle = BakerStreet + NottingHillGate + Westminster + LiverpoolStreet
}

fact Structure{

    getCircle[]
    getStraightLine[]
    getStraightLineBranches[]

    one x: Jubilee | no jubilee.x and one x.jubilee & Circle
    one x: Jubilee | no x.jubilee and one jubilee.x & Circle
    one x: Central | no x.central and one central.x & Circle
    one x: Central | #central.x = 2 and one x.central & Circle
    one x: Central & Jubilee |
        one jubilee.x & Circle and one x.jubilee & Circle and one central.x & Circle and
            one x.central & Circle
    all x: Circle | x in Jubilee implies x.circle in Central else x.circle in Jubilee
```

```
}

pred getCircle(){

    circle in Circle -> one Circle

    all x: Circle | one y: Circle{
        x -> y in circle
        y -> x not in circle
        one circle.x
    }

}

pred getStraightLine(){
    jubilee in Jubilee -> lone Jubilee
    one x: Jubilee | no x.jubilee and x = Waterloo
    one x: Jubilee | Jubilee in x.*jubilee and x = Stanmore
}

pred getStraightLineBranches(){

    central in Central -> lone Central

    one x: Central | no x.central and x = Epping

    one x: Central | all y: Central - central.x{
        #central.x = 2
        y in x.*central
        central.x = EalingBroadway + WestRuislip
    }

}

pred show(){}
run show
```

A.2. Elección de un proceso líder en un anillo

```
open util/ordering[Time] as TO
open util/ordering[Process] as PO

sig Time{}

sig Process{
    succ: Process,
    toSend: Process -> Time,
    elected: set Time
}

fact Ring{all p: Process | Process in p.^succ}

pred init(t: Time){all p: Process | p.toSend.t = p}

pred step(t, t': Time, p: Process){
    let from = p.toSend, to = p.succ.toSend |
        some id: from.t{
            from.t' = from.t - id
            to.t' = to.t + (id - PO/prevs[p.succ])
        }
}

pred skip(t, t': Time, p: Process){p.toSend.t = p.toSend.t'}

fact Traces{
    init[TO/first []]
    all t: Time - TO/last[] | let t' = TO/next[t] |
        all p: Process |
            step[t, t', p] or st[t, t', succ.p] or skip[t, t', p]
}

fact DefineElected{
    no elected.TO/first[]
    all t: Time – TO/first[] |
        elected.t =
            p: Process | p in p.toSend.t - p.toSend.(TO/prev[t])
```

```
}

assert AtMostOneElected{lone elected.Time}

check AtMostOneElected for 3 Process, 7 Time

pred progress(){
    all t: Time - TO/last[] | let t' = TO/next[t] |
        some Process.toSend.t =>
            some p: Process | not skip[t, t', p]
}

assert AtLeastOneElected{
    progress[] => some elected.Time
}

pred show(){}
run show for 3 Process, 7 Time
```

A.3. Estudio de caso para el elevador

```
open util/ordering [Tiempo] as TO
open util/ordering [Piso] as PO
open util/ordering [Usuario] as U


sig Tiempo{
    }

sig Piso{
    }

abstract sig Direccion{
        }

one sig arriba, abajo, reposo extends Direccion{
        }
```

–Un usuario hace una petición en un piso hacia otro piso en un tiempo dado.

```
sig Usuario{
    peticion: (Piso -> Piso ) lone -> Tiempo
        }


–Un elevador está en un piso con una dirección en un tiempo dado.

one sig Elevador{
    posicion:(Direccion -> Piso) one -> Tiempo
        }


fact cond_iniciales {
–No pasa que en el tiempo uno el elevador este en reposo.
((Elevador.posicion).TO/first).Piso & reposo = none

–La primera posición del elevador tiene que estar asociada a un piso.
posicion.first != none -> none -> none
}

fact cond_movimiento {
–No pasa que en el último piso el elevador quiera ir hacia arriba.
no t:Tiempo |( (posicion.t).PO/last).arriba != none

–No pasa que en el primer piso el elevador quiera ir hacia abajo.
no t:Tiempo |( (posicion.t).PO/first).abajo != none

–En el penúltimo tiempo siempre se está en reposo.
estado[1] implies reposo.((Elevador.(posicion)).(TO/last.prev)) != none
estado[0] implies reposo.((Elevador.(posicion)).(TO/last.prev)) != none
}

fact cond_finales {
–En el último tiempo el elevador siempre se esta en reposo.
reposo.((Elevador.(posicion)).TO/last) != none

–En el último tiempo no aparesca ningun usuario.
peticion.TO/last = none -> none -> none

–El número de peticiones de personas en el elevador en el tiempo previo al final siempre es uno.
```

#(((((peticion.(TO/**last.prev**)).
(piso_del_elevador[TO/**last.prev**])).Piso).peticion).(TO/**last.prev**)).Piso=1
}

fact cond_usuario {
–Se impide que un usuario que realize una petición en un tiempo que no es el primero, realicé una petición hacia un piso al cual quiere ir. Esto es que al hacer una petición dinámica algún usuario no podrá querer ir al piso en el que ya esta.
all t:Tiempo | **all** k:usuario_sin_peticion[t] | k.peticion.(t.**next**) != **none** -> **none**
implies (k.peticion.(t.**next**) & **iden**)= **none** -> **none**

–Restricción que nos dice que un usuario que es dejado o servido por el elevador no podrá subir al elevador otra vez en la misma instancia.
all t:Tiempo | **all** u:Usuario | piso_del_elevador[t] = piso_solicitado[u,t] **and** piso_del_elevador[t] = piso_del_usuario [u,t] **implies**
piso_solicitado[u,t.**nexts**] = **none and** piso_del_usuario [u,t.**nexts**] = **none**

–El piso al que el usuario quiere ir siempre es el mismo.
all disj u,v:Usuario | #u.peticion.(Tiempo-**last**).Piso= 1 **and** #v.peticion.(Tiempo-**last**).Piso= 1
}

```
    pred estado(r:Int) {
        r =1 implies #Usuario >1
        r =0 implies #Usuario =1

        }

    pred init [t:Tiempo] {
        all u:Usuario - usuario_sin_peticion[t] |
            ((u.peticion).t).Piso != Piso.((u.(peticion.t)))

        }

    pred init [u:Usuario] {
        u=U/first and u!= none
            and u in usuario_con_peticion[TO/first]

        }
```

fact trace {

```
    init[TO/first]
    init1[U/first]
        all t:Tiempo - last | let t'= t.next |
            all x,y,z:Piso |

–Reglas para dos o más usuarios.
                all u:Usuario | let u'= u.next |
                    r0[t,t',x,y,z,u,u'] or r1[t,t',x,y,z,u,u'] or r2[t,t',x,y,z,u,u'] or r3[t,t',x,y,z,u,u']
                        or r4[t,t',x,y,z,u,u'] or r5A[t,t',x,y,z,u,u'] or r5B[t,t',x,y,z,u,u']
                            or r6[t,t',x,y,z,u,u'] or r7[t,t',x,y,z,u,u']
                                or r8[t,t',x,y,z,u,u'] or r9[t,t',x,y,z,u,u']
–Reglas para dos o más usuarios.
                    ru0[t,t',x,y,z,u,u'] or ru1[t,t',x,y,z,u,u'] or ru2[t,t',x,y,z,u,u'] or ru3[t,t',x,y,z,u,u']
                        or ru4[t,t',x,y,z,u,u'] or ru5[t,t',x,y,z,u,u']
                            or ru6[t,t',x,y,z,u,u']
        }


pred r0 (t: Tiempo , t': Tiempo, x, y, z:Piso,u,u':Usuario){

    let u= usuario_en_elevadorAbajo[t], u'=usuario_NoEn_elevadorAbajo[t],
        v=usuario_en_elevadorSiguenteTiempo[t'], v'=usuario_NoEn_elevadorSiguenteTiempo[t'],
        k=usuario_sin_peticion[t], r = min[usuario_con_peticion[t]],
        x=piso_del_elevador[t],y=piso_solicitado[u,t],z=piso_del_usuario [u,t],
        x'=piso_del_elevador[t'],y'=piso_solicitado[v,t'],z'=piso_del_usuario [v,t'],
        b=piso_solicitado[u',t],c=piso_del_usuario [u',t],
        b'=piso_solicitado[v',t'],c'=piso_del_usuario [v',t'],
        d=((Elevador.posicion).t).Piso, d'= ((Elevador.posicion).t').Piso |

            estado[1] and x = z and x not in y
            and x.prev not in y and z.prev not in y
            and d = abajo
            and x != none and y != none and z != none and u != none
            and (x = PO/first.next or (x != PO/first.next and d' = abajo) )
            and ( v = u + (((peticion.t).(x.prev)).Piso)
            or v = u + (((peticion.t).(x.prev)).Piso) + k)
            and ( ( (v'.(peticion.t')) = ( (u'.(peticion.t )) -
            (((((peticion.t).(x.prev)).Piso) ).(peticion.t)) ) )
            or (v'.(peticion.t')) = (( (u'.(peticion.t )) -
            ((((((peticion.t).(x.prev)).Piso) ).(peticion.t))) ) +
```

```
        k.peticion.t' ) or ( (v'.(peticion.t')) = (u'.(peticion.t )) )
        or ( (v'.(peticion.t')) = none ->none ) )
        and # (peticion.t )<= # (peticion.t')
        and (peticion.t ) - (u->y->x) in (peticion.t')
        and x'= x.prev and z'=z.prev
        and x' = z' and y in y'
        and(( b' in b and c' in c) or
        (b in b' and c in c') or (b = b' and c = c') )
        and ( lt[ piso_del_usuario [r,t] , piso_del_elevador[t] ] or
        ( piso_del_usuario [r,t] = piso_del_elevador[t] and
        lt [ piso_solicitado[r,t] , piso_del_elevador[t] ]) )
        }

pred r1(t: Tiempo , t': Tiempo, x, y, z:Piso,u,u':Usuario){

    let u= usuario_en_elevadorArriba[t], u'=usuario_NoEn_elevadorArriba[t],
        v=usuario_en_elevadorSiguenteTiempo[t'], v'=usuario_NoEn_elevadorSiguenteTiempo[t'],
        k=usuario_sin_peticion[t],r = min[usuario_con_peticion[t]],
        x=piso_del_elevador[t],y=piso_solicitado[u,t],z=piso_del_usuario [u,t],
        x'=piso_del_elevador[t'],y'=piso_solicitado[v,t'],z'=piso_del_usuario [v,t'],
        b=piso_solicitado[u',t],c=piso_del_usuario [u',t],
        b'=piso_solicitado[v',t'],c'=piso_del_usuario [v',t'],
        d= ((Elevador.posicion).t).Piso, d'= ((Elevador.posicion).t').Piso |

            estado[1] and x = z and x not in y
            and x.next not in y and z.next not in y
            and d = arriba
            and x != none and y != none and z != none and u != none
            and ( x = PO/last.prev or (x != PO/last.prev and d' = arriba) )
            and ( v = u + (((peticion.t).(x.next)).Piso)
            or v = u + (((peticion.t).(x.next)).Piso) + k)
            and ( ( (v'.(peticion.t')) = ( (u'.(peticion.t )) -
            (((((peticion.t).(x.next)).Piso) ).(peticion.t)) ) )
            or (v'.(peticion.t')) = (( (u'.(peticion.t )) -
            ((((((peticion.t).(x.next)).Piso) ).(peticion.t))) )
            + k.peticion.t' ) or ( (v'.(peticion.t')) = (u'.(peticion.t )) )
            or ( (v'.(peticion.t')) = none->none ) )
            and # (peticion.t )<= #(peticion.t')
            and (peticion.t ) - (u->y->x) in (peticion.t')
```

```
        and x'= x.next and z'=z.next
        and x' = z' and y in y'
        and(( b' in b and c' in c) or
        (b in b' and c in c') or (b = b' and c = c') )
        and( gt[ piso_del_usuario [r,t] , piso_del_elevador[t] ] or
        (piso_del_usuario [r,t] = piso_del_elevador[t]
        and gt [ piso_solicitado[r,t] , piso_del_elevador[t] ]) )

}


pred r2(t: Tiempo , t': Tiempo, x, y, z:Piso,u,u':Usuario){

    let u = min[usuario_con_peticion[t]], u'=usuario_NoEn_elevadorAbajo[t],
        v =usuario_en_elevadorAbajo[t],
        x=piso_del_elevador[t], y=piso_solicitado[u,t], z=piso_del_usuario [u,t],
        x'=piso_del_elevador[t'], y'=piso_solicitado[u,t'], z'=piso_del_usuario [u,t'],
        b=piso_solicitado[u',t], c=piso_del_usuario [u',t],
        b'=piso_solicitado[u',t], c'=piso_del_usuario [u',t],
        d= ((Elevador.posicion).t).Piso |

            estado[1] and x != z
            and (x.prev =x' ) and (y in y') and (z'= z)
            and d = abajo
            and v= none
            and u != none
            and (peticion.t ) in (peticion.t')
            and # (peticion.t )<= #(peticion.t')
            and (z = min[x + z])
            and x != none and y != none and z != none
            and b in b' and c in c'

        }


pred r3(t: Tiempo , t': Tiempo, x, y, z:Piso,u,u':Usuario){

    let u = min[usuario_con_peticion[t]], u'=usuario_NoEn_elevadorAbajo[t],
        v =usuario_en_elevadorAbajo[t],
        x=piso_del_elevador[t], y=piso_solicitado[u,t], z=piso_del_usuario [u,t],
```

```
    x'=piso_del_elevador[t'],y'=piso_solicitado[u,t'],z'=piso_del_usuario [u,t'],
    b=piso_solicitado[u',t], c=piso_del_usuario [u',t],
    b'=piso_solicitado[u',t],c'=piso_del_usuario [u',t],
    d= ((Elevador.posicion).t).Piso |

        estado[1] and x != z
        and (x.next =x' ) and (y in y') and (z'= z)
        and d = arriba
        and v= none
        and u != none
        and (peticion.t ) in (peticion.t')
        and # (peticion.t ) <= #(peticion.t')
        and (x = min[x + z])
        and x != none and y != none and z != none
        and b in b' and c in c'

    }
```

Regla 4:

```
pred r4(t: Tiempo , t': Tiempo, x, y, z:Piso,u,u':Usuario){

    let u= x=piso_del_elevador[t], y=piso_solicitado[u,t], z=piso_del_usuario [u,t],
        x'=piso_del_elevador[t'], y'=piso_solicitado[u,t'], z'=piso_del_usuario [u,t'],
        d= ((Elevador.posicion).t).Piso, d'= ((Elevador.posicion).t').Piso |

        estado[1] and x = z
        and x in y
        and x != none and y != none and z != none
        and d = reposo
        and x' = x and z'= none and y' = none
        and d' = reposo

        }
```

Regla 5A:

```
pred r5A(t: Tiempo , t': Tiempo, x, y, z:Piso,u,u':Usuario){
```

let u= usuario_en_elevadorAbajo[t], u'=usuario_NoEn_elevadorAbajo[t],
v=usuario_en_elevadorSiguenteTiempo[t'], v'=usuario_NoEn_elevadorSiguenteTiempo[t'],
x=piso_del_elevador[t], y=piso_solicitado[u,t], z=piso_del_usuario [u,t],
x'=piso_del_elevador[t'], y'=piso_solicitado[v,t'], z'=piso_del_usuario [v,t'],
b=piso_solicitado[u',t], c=piso_del_usuario [u',t],
b'=piso_solicitado[v',t'], c'=piso_del_usuario [v',t'],
d= ((Elevador.posicion).t).Piso, d'= ((Elevador.posicion).t').Piso |

estado[1] **and** u != **none and** v != **none and** u = v
and u'= **none and** v'=**none and** b= **none and** c = **none**
and b = b' **and** c =c'
and x = z **and** x **not in** y
and x.**prev** = y and z.**prev** = y
and x != **none and** y != **none and** z != **none**
and d = abajo
and x' = x.**prev**
and x' != **none and** y' != **none and** z' != **none**
and d' = reposo

}

Regla 5B:

pred r5B(t: Tiempo , t': Tiempo, x, y, z:Piso,u,u':Usuario){

let u= usuario_en_elevadorAbajo[t], u'=usuario_NoEn_elevadorAbajo[t],
v=usuario_en_elevadorSiguenteTiempo[t'], v'=usuario_NoEn_elevadorSiguenteTiempo[t'],
x=piso_del_elevador[t], y=piso_solicitado[u,t], z=piso_del_usuario [u,t],
x'=piso_del_elevador[t'], y'=piso_solicitado[v,t'], z'=piso_del_usuario [v,t'],
b=piso_solicitado[u',t], c=piso_del_usuario [u',t],
b'=piso_solicitado[v',t'], c'=piso_del_usuario [v',t'],
d= ((Elevador.posicion).t).Piso, d'= ((Elevador.posicion).t').Piso |

estado[1] **and** u != **none and** v != **none and** u = v
and u'= **none and** v'=**none and** b= **none and** c = **none**
and b = b' **and** c =c'
and x = z **and** x **not in** y
and x.**next** = y and z.**next** = y
and x != **none and** y != **none and** z != **none**

```
        and d = arriba
        and x' = x.next
        and x' != none and y' != none and z' != none
        and d' = reposo

    }
```

Regla 6:

```
pred r6 (t: Tiempo , t': Tiempo, x, y, z:Piso,u,u':Usuario){

    let u= usuario_en_elevadorAbajo[t], u'=usuario_NoEn_elevadorAbajo[t],
        v=usuario_en_elevadorSiguenteTiempo[t'], v'=usuario_NoEn_elevadorSiguenteTiempo[t'],
        w=usuario_en_elevadorSiguenteTiempo[t], k=usuario_sin_peticion[t],
        r =min[usuario_con_peticion[t]],
        x=piso_del_elevador[t], y=piso_solicitado[u,t], z=piso_del_usuario [u,t],
        x'=piso_del_elevador[t'], y'=piso_solicitado[v,t'], z'=piso_del_usuario [v,t'],
        b=piso_solicitado[u',t], c=piso_del_usuario [u',t],
        b'=piso_solicitado[v',t'], c'=piso_del_usuario [v',t'],
        d= ((Elevador.posicion).t).Piso, d'= ((Elevador.posicion).t').Piso |

            estado[1] and x = z and x not in y
            and x != none and y != none and z != none and u != none
            and (x = PO/first.next or (x != PO/first.next and d' = abajo) )
            and ( v = u + (((peticion.t).(x.prev)).Piso)
            or v = u +(((peticion.t).(x.prev)).Piso) + k)
            and ( ( (v'.(peticion.t')) = ( (u'.(peticion.t )) -
            (((((peticion.t).(x.prev)).Piso) ).(peticion.t)) ) )
            or (v'.(peticion.t')) = (( (u'.(peticion.t )) -
            ((((((peticion.t).(x.prev)).Piso) ).(peticion.t))) )
            or ( (v'.(peticion.t')) = (u'.(peticion.t )) ) or
            ( (v'.(peticion.t')) = none -> none ) )
            and # (peticion.t )¡= #(peticion.t')
            and (peticion.t ) - (u -> y -> x) in (peticion.t')
            and x.prev in y and z.prev in y
            and d = abajo
            and x'= x.prev and z'=z.prev
            and x' = z' and y in y'
            and b in b' and c in c'
```

```
        and( lt[ piso_del_usuario [r,t] , piso_del_elevador[t] ] or
        ( piso_del_usuario [r,t] = piso_del_elevador[t]
        and lt [ piso_solicitado[r,t] , piso_del_elevador[t] ]) )

    }
```

Regla 7:

```
pred r7 (t: Tiempo , t': Tiempo, x, y, z:Piso,u,u':Usuario){

    let u= usuario_en_elevadorAbajo[t], u'=usuario_NoEn_elevadorAbajo[t],
        v=usuario_en_elevadorSiguenteTiempo[t'], v'=usuario_NoEn_elevadorSiguenteTiempo[t'],
        w = usuario_en_elevadorSiguenteTiempo[t], k=usuario_sin_peticion[t],
        r =min[usuario_con_peticion[t]],
        x=piso_del_elevador[t], y=piso_solicitado[u,t], z=piso_del_usuario [u,t],
        x'=piso_del_elevador[t'], y'=piso_solicitado[v,t'], z'=piso_del_usuario [v,t'],
        b=piso_solicitado[u',t], c=piso_del_usuario [u',t],
        b'=piso_solicitado[v',t'], c'=piso_del_usuario [v',t'],
        d= ((Elevador.posicion).t).Piso, d'= ((Elevador.posicion).t').Piso |

            estado[1] and x = z and x not in y
            and x != none and y != none and z != none and u != none
            and (x = PO/last.prev or (x != PO/last.prev and d' = arriba) )
            and ( v = u + (((peticion.t).(x.next)).Piso)
            or v = u +(((peticion.t).(x.next)).Piso) + k)
            and ( ( (v'.(peticion.t')) = ( (u'.(peticion.t )) -
            (((((peticion.t).(x.next)).Piso) ).(peticion.t)) ) )
            or (v'.(peticion.t')) = (( (u'.(peticion.t )) -
            ((((((peticion.t).(x.next)).Piso) ).(peticion.t))) )
            or ( (v'.(peticion.t')) = (u'.(peticion.t )) ) or
            ( (v'.(peticion.t')) = none -> none ) )
            and # (peticion.t )¡= #(peticion.t')
            and (peticion.t ) - (u -> y -> x) in (peticion.t')
            and x.next in y and z.next in y
            and d = arriba
            and x'= x.next and z'=z.next
            and x' = z' and y in y'
            and b in b' and c in c'
            and( gt[ piso_del_usuario [r,t] , piso_del_elevador[t] ] or
```

```
        ( piso_del_usuario [r,t] = piso_del_elevador[t]
        and gt [ piso_solicitado[r,t] , piso_del_elevador[t] ]) )

    }
```

Regla 8:

```
pred r8(t: Tiempo , t': Tiempo, x, y, z:Piso,u,u':Usuario){

    let u=usuario_En_pisoDesado[t],u'=usuario_NoEn_pisoDesado[t],
        w=usuario_En_pisoDesado[t'],
        v'=usuario_NoEn_pisoDesado[t'],v=usuario_en_elevadorSiguenteTiempo[t'],
        x=piso_del_elevador[t], y=piso_solicitado[u,t], z=piso_del_usuario [u,t],
        x'=piso_del_elevador[t'], y'=piso_solicitado[w,t'], z'=piso_del_usuario [w,t'],
        b=piso_solicitado[u',t], c=piso_del_usuario [u',t],
        b'=piso_solicitado[v',t'], c'=piso_del_usuario [v',t'],
        d= ((Elevador.posicion).t).Piso, d'= ((Elevador.posicion).t').Piso |

            estado[1] and x = z
            and x in y
            and x != none and y != none and z != none
            and u != none
            and (peticion.t ) - (u -> y ->x) in (peticion.t')
            and # ((peticion.t ) - (u ->y ->x)) <= #(peticion.t')
            and u.peticion.t' = none -> none
            and v = ( (((peticion.t').(x)).Piso) )- u
            and # (u'.(peticion.t )) <= # (v'.(peticion.t'))
            and d = abajo
            and x'= x and z' = none and y' =none
            and b in b' and c in c'
            and d' = abajo

    }
```

Regla 9:

```
pred r9(t: Tiempo , t': Tiempo, x, y, z:Piso,u,u':Usuario){

    let u=usuario_En_pisoDesado[t], u'=usuario_NoEn_pisoDesado[t],
```

```
        w=usuario_En_pisoDesado[t'],
        v'=usuario_NoEn_pisoDesado[t'], v=usuario_en_elevadorSiguenteTiempo[t'],
        x=piso_del_elevador[t], y=piso_solicitado[u,t], z=piso_del_usuario [u,t],
        x'=piso_del_elevador[t'], y'=piso_solicitado[w,t'], z'=piso_del_usuario [w,t'],
        b=piso_solicitado[u',t], c=piso_del_usuario [u',t],
        b'=piso_solicitado[v',t'], c'=piso_del_usuario [v',t'],
        d= ((Elevador.posicion).t).Piso, d'= ((Elevador.posicion).t').Piso |

            estado[1] and x = z
            and x in y
            and x != none and y != none and z != none
            and u != none
            and (peticion.t ) - (u -> y ->x) in (peticion.t')
            and # ((peticion.t ) - (u -> y- > x)) <= #(peticion.t')
            and u.peticion.t' = none -> none
            and v = ( (((peticion.t').(x)).Piso) )- u
            and # (u'.(peticion.t )) <= # (v'.(peticion.t'))
            and d = arriba
            and x'= x and z' = none and y' =none
            and b in b' and c in c'
            and d' = arriba

        }


pred ru0 (t: Tiempo , t': Tiempo, x, y, z:Piso,u,u':Usuario){
    let u=Usuario ,u'= none,
        x=piso_del_elevador[t], y=piso_solicitado[u,t], z=piso_del_usuario [u,t],
        x'=piso_del_elevador[t'], y'=piso_solicitado[u,t'], z'=piso_del_usuario [u,t'],
        d= ((Elevador.posicion).t).Piso, |

            estado[0]
            and x = z and y != x
            and (y = min[x + y])
            and x != none and y != none and z != none
            and d = abajo
            and y != x.prev and y != z.prev
            and x'.next = x and z'.next = z
            and x' = z' and y' = y
            and u'= none
```

```
        }

pred ru1 (t: Tiempo , t': Tiempo, x, y, z:Piso,u,u':Usuario){
    let u=Usuario ,u'= none,
        x=piso_del_elevador[t],y=piso_solicitado[u,t],z=piso_del_usuario [u,t],
        x'=piso_del_elevador[t'], y'=piso_solicitado[u,t'], z'=piso_del_usuario [u,t'],
        d= ((Elevador.posicion).t).Piso |

            estado[0]
            and x = z and y != x
            and (y = max[x + y])
            and x != none and y != none and z != none
            and d = arriba
            and y != x.next and y != z.next
            and x'.prev = x and z'.prev = z
            and x' = z' and y' = y
            and u'= none
            }

pred ru2 (t: Tiempo , t': Tiempo, x, y, z:Piso,u,u':Usuario){
    let u=Usuario ,u'= none,
        x=piso_del_elevador[t], y=piso_solicitado[u,t], z=piso_del_usuario [u,t],
        x'=piso_del_elevador[t'], y'=piso_solicitado[u,t'], z'=piso_del_usuario [u,t'],
        d= ((Elevador.posicion).t).Piso |

            estado[0]
            and x != z
            and (z = min[x + z])
            and x != none and y != none and z != none
            and d = abajo
            and x.prev = x'
            and y' = y' and z' = z
            and u'= none
            }

pred ru3 (t: Tiempo , t': Tiempo, x, y, z:Piso,u,u':Usuario){
    let u=Usuario ,u'= none,
        x=piso_del_elevador[t], y=piso_solicitado[u,t], z=piso_del_usuario [u,t],
```

```
        x'=piso_del_elevador[t'], y'=piso_solicitado[u,t'], z'=piso_del_usuario [u,t'],
        d= ((Elevador.posicion).t).Piso |

            estado[0]
            and x != z
            and (x = min[x + z])
            and x != none and y != none and z != none
            and d = arriba
            and x.next = x'
            and y' = y' and z' = z
            and u'= none
            }


pred ru4 (t: Tiempo , t': Tiempo, x, y, z:Piso,u,u':Usuario){
    let u=Usuario ,u'= none,
        x=piso_del_elevador[t], y=piso_solicitado[u,t], z=piso_del_usuario [u,t],
        x'=piso_del_elevador[t'], y'=piso_solicitado[u,t'], z'=piso_del_usuario [u,t'],
        d= ((Elevador.posicion).t).Piso, d'= ((Elevador.posicion).t').Piso |

            estado[0]
            and x = z and y != x
            and y = x.prev and y = z.prev
            and x != none and y != none and z != none
            and d = abajo
            and x' = x.prev and y'= y and z'=z.prev none
            and d' = reposo
            and u'= none
            }


pred ru5 (t: Tiempo , t': Tiempo, x, y, z:Piso,u,u':Usuario){
    let u=Usuario ,u'= none,
        x=piso_del_elevador[t], y=piso_solicitado[u,t], z=piso_del_usuario [u,t],
        x'=piso_del_elevador[t'], y'=piso_solicitado[u,t'], z'=piso_del_usuario [u,t'],
        d= ((Elevador.posicion).t).Piso, d'= ((Elevador.posicion).t').Piso |

            estado[0]
            and x = z and y != x
            and y = x.next and y = z.next
```

```
        and x != none and y != none and z != none
        and d = arriba
        and x' = x.next and y'= y and z'= z.next none
        and d' = reposo
        and u'= none
        }


pred ru6 (t: Tiempo , t': Tiempo, x, y, z:Piso,u,u':Usuario){
    let u=Usuario ,u'= none,
        x=piso_del_elevador[t], y=piso_solicitado[u,t], z=piso_del_usuario [u,t],
        x'=piso_del_elevador[t'], y'=piso_solicitado[u,t'], z'=piso_del_usuario [u,t'],
        d= ((Elevador.posicion).t).Piso, d'= ((Elevador.posicion).t').Piso |

        estado[0]
        and x = y
        and z = y and x != none and
        and d = reposo
        and x' = x and y' = none and z' = none
        and d = reposo
        and u'= none
        }
```

Bibliografía

[Alloy comunity] Alloy Community, Tutorial for Alloy Analyzer 4.0: *http://alloy.mit.edu/alloy4/tutorial4/*.

[Alloy] Alloy Community, *http://alloy.mit.edu/faq.php*.

[Cunningham 1994] H. Conrad Cunningham, Viren R. Shah, and Shu Shen, *Devising a Formal Specification for an Elevator Controller*, 1994.

[Dennis 2008] Greg Dennis and Rob Seater, *Alloy Analyzer 4 Tutorial Software Design Group, MIT*, 2008.

[Huth,Ryan 2004] Michael Huth and Mark Ryan, Logic in Computer Science: *Modelling and Reasoning about Systems*, 2004.

[Jackson 2006] Daniel Jackson, Software Abstractions: *Logic, Languaje, and Analysis*, 2006.

[Jackson 2002] Daniel Jackson, *Micromodels of Software, Lecture 4:Case Study*, 2002.

[Jackson 2005] Daniel Jackson, *Alloy in 90 minutes*, 2005.

[Khiang-Khalid-Yusof 2007] Khiang Tan Kok Khiang, Marzuki Khalid, Rubiyah Yusof, *Intelligent Elevator Control By Ordinal Structure Fuzzy Logic Algorithm*, 2007.

[Miranda 2008] Favio E. Miranda Perea, Micromodelos de Software, XLI Congreso Nacional SMM Valle de Bravo, Edo. de México, 2008.

[Morgan 1998] Carroll Morgan, Programing from Specifications, 1998.

[Strobol-Wisspeintner 1999] Frank Strobol and Alexander Wisspeintner, *Specification of an Elevator Control System*, 1999.

[VDM] *http://en.wikipedia.org/wiki/Vienna_Development_Method*, Enero 2011.

[Wood 1989] William G. Wood *Temporal Logic Case Study*, 1989.