UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS

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

T E S I S

QUE PARA OBTENER EL TÍTULO DE:

LICENCIADO EN CIENCIAS DE LA COMPUTACIÓN

PRESENTA:

JUAN CARLOS CORTÉS ORTIZ

DIRECTOR DE TESIS:

DR. FAVIO EZEQUIEL MIRANDA PEREA

2011

Universidad Nacional
Autónoma de México

Biblioteca Central

Dirección General de Bibliotecas de la UNAM

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

1. Datos del alumno	1. Datos del alumno
Apellido paterno	Cortés
Apellido materno	Ortiz
Nombre(s)	Juan Carlos
Teléfono	26 19 06 90
Universidad	Universidad Nacional Autónoma de México
Facultad	Facultad de Ciencias
Carrera	Ciencias de la Computación
Número de cuenta	300181399
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 correo electrónico en el analizador Alloy
Número de páginas	80 p
Año	2011

Índice general

1. Introducción **1**
1.1. Métodos formales . . . 2
1.2. Descripción del trabajo . . . 3

2. Preliminares **5**
2.1. El analizador Alloy . . . 5
2.2. Especificación en lógica relacional . . . 6
2.3. Ejemplos . . . 19
2.3.1. Modelo estático . . . 19
2.3.2. Modelo dinámico . . . 23

3. Estudio de un caso: *Sistema de correo electrónico* **29**
3.1. Modelo básico . . . 29
3.1.1. Usuarios . . . 29
3.1.2. Mensajes . . . 31
3.1.3. Servidor . . . 32
3.2. Trazas . . . 32
3.3. Operaciones sobre los mensajes . . . 33
3.3.1. Redactar . . . 34
3.3.2. Guardar . . . 37
3.3.3. Descartar . . . 40
3.3.4. Enviar . . . 40
3.3.5. Recibir . . . 43
3.3.6. Recibir un mensaje de advertencia . . . 44
3.3.7. Leer . . . 45
3.3.8. Reenviar . . . 46
3.3.9. Recibir un mensaje reenviado . . . 47
3.3.10. Eliminar . . . 48
3.3.11. Eliminar de manera definitiva . . . 49
3.3.12. Reportar como spam . . . 50
3.3.13. Recuperar del spam . . . 51
3.4. Análisis . . . 53

4. Conclusiones **61**

A. Código fuente **63**
A.1. Líneas del metro de la ciudad de Londres . . . 63
A.2. Elección de un proceso líder en un anillo . . . 65
A.3. Estudio del caso: *Sistema de correo electrónico* . . . 67

Bibliografía **77**

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"*, véase [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*, véase [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 especifi-

cación 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, véase [Alloy Community].

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.

1.2. Descripción del trabajo

El objetivo de esta tesis es el de modelar, analizar y verificar un sistema de correo electrónico haciendo uso de los métodos formales ligeros.

Mediante el analizador Alloy, construiremos una especificación que nos permita realizar un *análisis dinámico* en el cual podamos verificar las propiedades de un sistema de correo electrónico. Primero definiremos las entidades básicas junto con sus respectivas propiedades, después construiremos un conjunto de operaciones que modelarán las funciones o características de un sistema real, por último, para comprobar que nuestra especificación es correcta, analizaremos el comportamiento de nuestro modelo bajo diversas circunstancias que nosotros mismos podemos establecer mediante las *aserciones* que nos permite definir el analizador Alloy.

Decimos que realizaremos un análisis dinámico por que utilizaremos una técnica que nos permitirá crear una *traza* para cada instancia generada de nuestra especificación por el analizador Alloy. Una *traza* define un conjunto de pasos ordenados sobre una signatura que representará un conjunto de estados, a partir de un estado inicial podemos realizar una transición de un estado a otro aplicando alguna operación que hayamos definido, en donde cada estado define un cierto comportamiento que tendrá nuestro modelo.

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

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/`.

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

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

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 [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:	$p \& q = \{t \mid t \mid p \wedge t \in q\}$
subconjunto:	p **in** q = $\{(p1 \ldots pn) \in p\} \subseteq \{(q1 \ldots qm) \in q\}$
igualdad:	$(p = q) = \{(p1 \ldots pn) \in p\} = \{(q1 \ldots qn) \in q\}$

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) $\in$ p $\wedge$ (q1 ... qm) $\in$ 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) $\in$ p $\wedge$ (pn,q2 ... qm) $\in$ 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: ˆr = r + r.r + r.r.r + ...
Cerradura reflexiva y transitiva: *r = ˆr + **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)}
ˆnext = {(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: ˆr & **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.

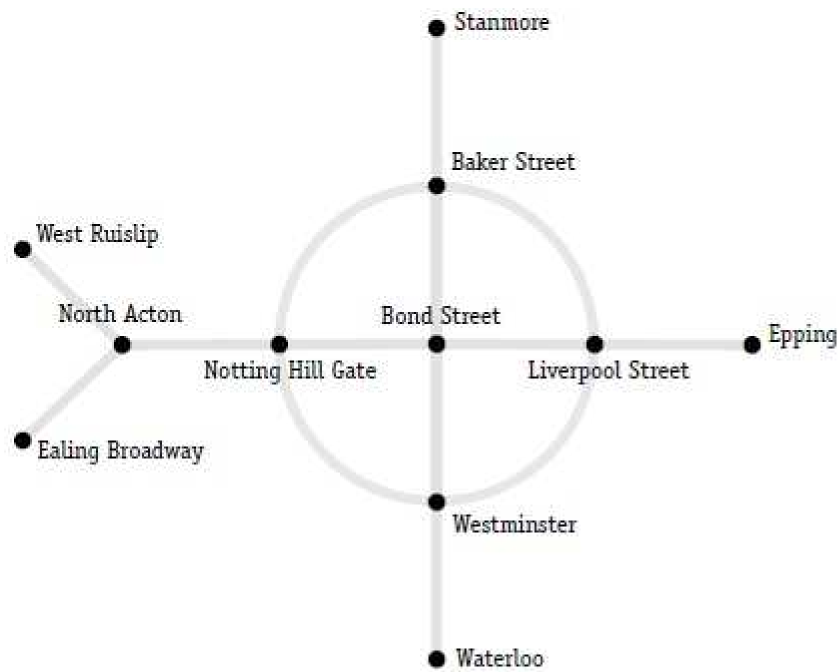

Figura 2.1: Diagrama simplificado del metro de Londres.

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.

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

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{}
```

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

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*.

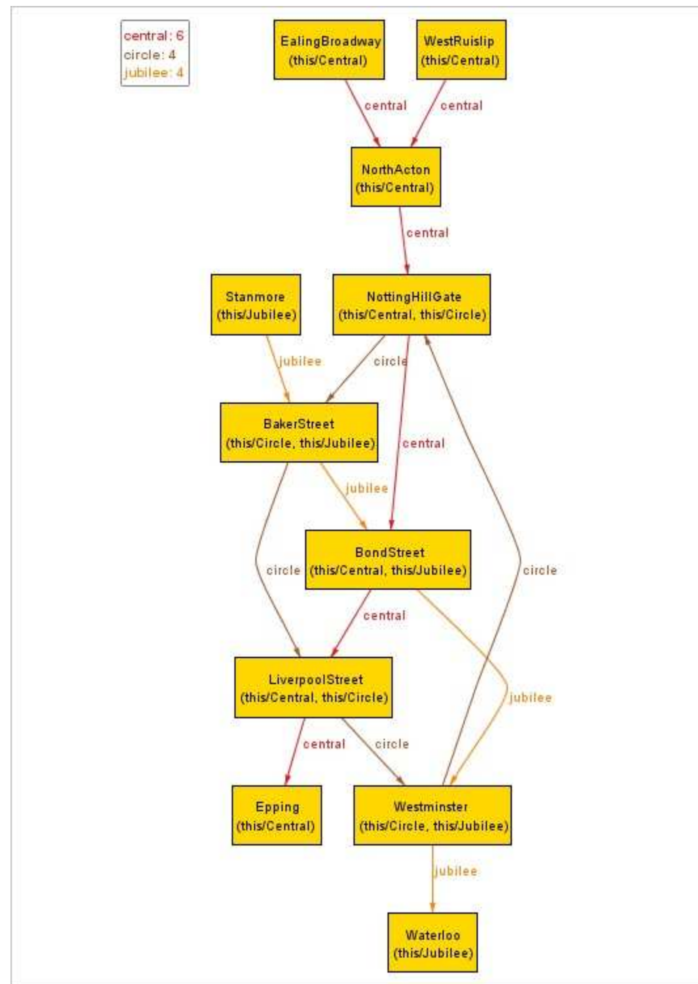

Figura 2.2: Instancia generada por el analizador Alloy correspondiente a la figura 2.1.

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.

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.

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

Time = t0 + t1 + t2 + t3 + t4

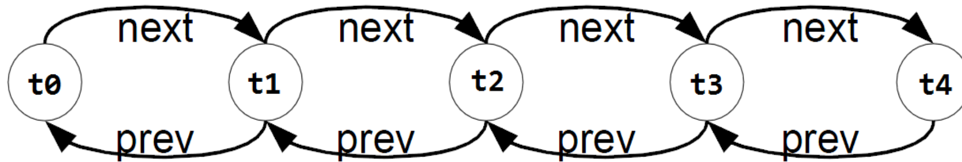

Figura 2.3: Funcionamiento de una traza.

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′
}
```

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.

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

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.

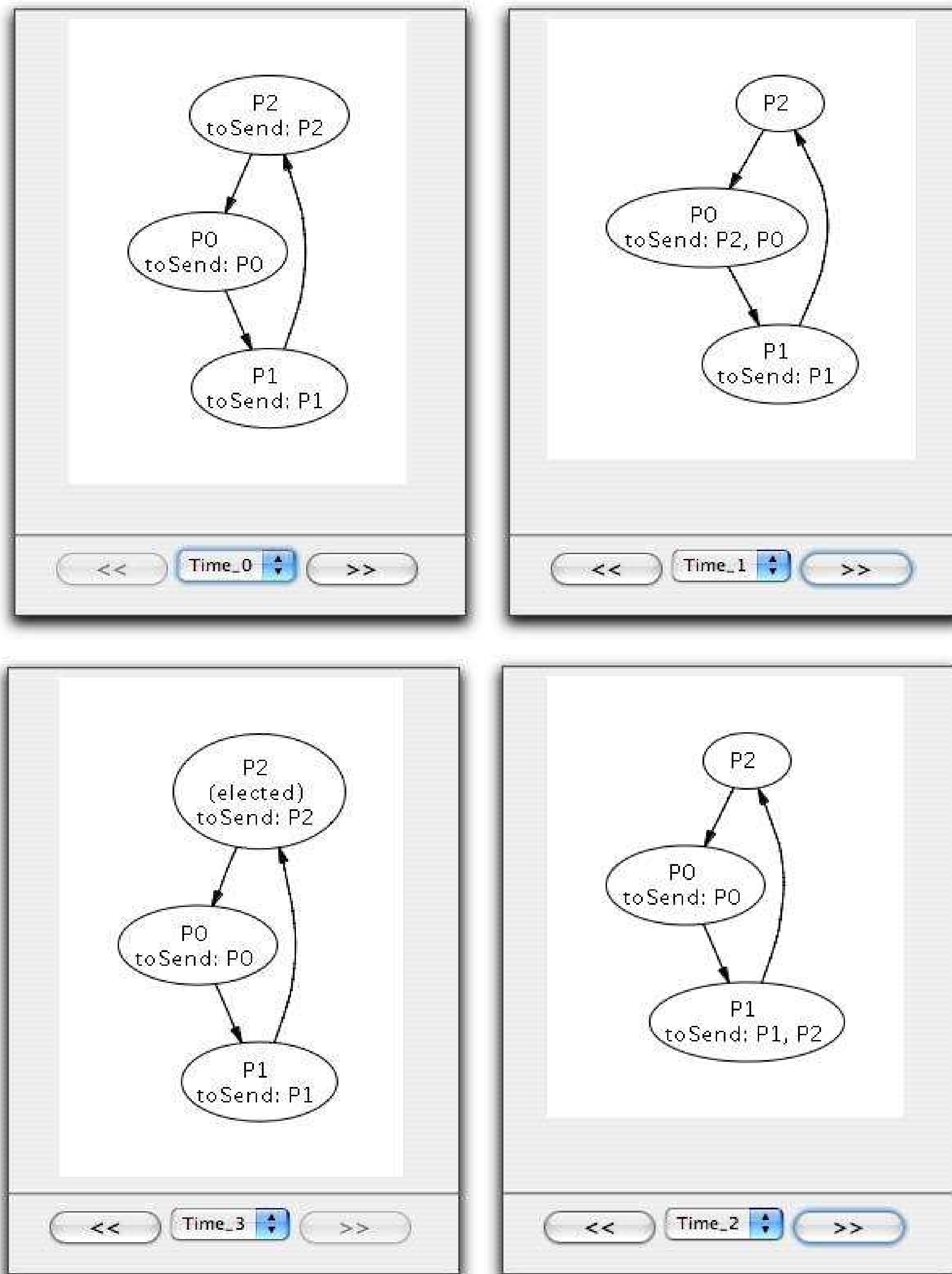

Figura 2.4: Traza generada por el analizador Alloy de la elección de un proceso líder en una topología de anillo.

3: Estudio de un caso: *Sistema de correo electrónico*

En la actualidad existen muchas empresas como Google y Yahoo, que proveen de varios servicios, uno de ellos es el correo electrónico. Este servicio le permite a las personas registrarse en el sistema para tener una cuenta de manera gratuita. Una vez que las personas se registran, son usuarios de este servicio y cuentan con su propio correo electrónico donde pueden realizar diversas operaciones, desde recibir o enviar un *e-mail* hasta comunicarse con sus amigos en una red social. Por ejemplo, este es el caso de Gmail.

A lo largo de este capítulo modelaremos las funciones (*operaciones*) básicas de un sistema de correo electrónico mediante especificaciones, incluiremos una traza a nuestra especificación para analizar de una manera dinámica el comportamiento de nuestro modelo bajo diversas operaciones que un sistema de correo electrónico le permite realizar a los usuarios. La traza estará definida por intervalos de tiempo que representarán a los estados que definen el comportamiento del sistema.

3.1. Modelo básico

Las entidades mas importantes del modelo que construiremos son Usuarios: las personas que se han registrado[1] en algún servicio de correo electrónico, Servidor: el proveedor del servicio de correo electrónico y Mensajes: el texto que contiene un *e-mail*. Pero también debemos definir a la entidad que representará momentos en el tiempo:

```
sig User{}
sig Message{}
one sig Server{}
sig Time{}
```

Al utilizar la palabra reservada *one* en la declaracion de la signatura *Server*, le estamos indicando al analizador Alloy que sólo queremos generar un servidor para cada instancial del modelo.

Las propiedades de las entidades son relaciones que nos van a permitir modelar las operaciones que definiremos más adelante. Comenzaremos definiendo las propiedades de los usuarios.

3.1.1. Usuarios

Lo más básico que puede hacer un usuario es redactar un mensaje, definimos una relación que asocie a los usuarios con los mensajes:

```
write: Message
```

[1] Suponemos que los usuarios se han registrado con el mismo proveedor, el cual cuenta con un solo servidor, por lo tanto, todos los usuarios recibirarán el servicio del mismo servidor.

La propiedad *write* nos muestra todos los mensajes redactados por los usuarios, pero debido a que construiremos un modelo dinámico con respecto al tiempo, tenemos que agregar la signatura *Time* en la declaración:

write: Message -> Time

Ahora sí, la relación *write* nos muestra los mensajes redactados por los usuarios en un cierto tiempo, por lo que si hacemos *u.write.t*, obtenemos todos los mensajes redactados por el usuario *u* en el tiempo *t*.

Un sistema de correo electrónico le permite a los usuarios poder *enviar* o *reenviar* mensajes a otros usuarios. Cuando un usuario envía o reenvía algún mensaje, no le llega de manera inmediata al usuario a quien va dirigido, lo que sucede es que primero se envía el mensaje al servidor y después el servidor se encarga de hacerle llegar el mensaje al destinatario. Las relaciones *send* y *forward* modelan estas operaciones asociando a los usuarios que realizan la acción con los mensajes que se quieren enviar o reenviar junto con el servidor a un determinado momento del tiempo:

send: Message -> Server -> Time
forward: Message -> Server -> Time

Las bandejas en un sistema de correo electrónico son lugares o directorios donde se almacenan los mensajes. Las bandejas que manejaremos son: *Inbox* (correos recibidos), *Sent Mail* (correos enviados), *Drafts* (correos redactados que no se enviaron y se encuentran guardados), *Spam* (correo spam) y *Trash* (correo basura). Estas bandejas son relaciones que asocian a un usuario con mensajes que llegaron a su correo electrónico en algún momento:

inbox: Message -> Time
sentMail: Message -> Time
drafts: Message -> Time
spam: Message -> Time
trash: Message -> Time

Tenemos que hacer que los usuarios sean capaces de poder ver cual es el estado de sus mensajes en el momento que ellos deseen. Necesitamos de una relación que asocie a los mensajes de los usuarios con un estado a un determinado tiempo:

messageStatus: Status -> Message -> Time

El estado de los mensajes al que nos referimos es el de *leído* y *no leído*, por lo que primero debemos declarar la signatura *Status* y definir sus elementos:

abstract sig Status{}
one sig Read, Unread **extends** Status{}

Por último los usuarios necesitan conocer el espacio libre de sus respectivas cuentas:

freeSpace: **Int one** -> Time

La única manera en que podemos medir el espacio de una cuenta en el analizador Alloy es mediante el uso de los números enteros. La expresión *Int one -> Time* quiere decir que asociamos a cada elemento de *Time* con un único entero. Hacemos esto por que no queremos que en un mismo tiempo los usuarios tenga dos relaciones que muestren el espacio de la cuenta e incluso con valores diferentes. En la figura 3.1 se muestra una instancia de como sería la relación *freesSpace* si no la declaráramos como lo hicimos y en la figura 3.2 se muestra lo que realmente queremos.

Figura 3.1: Instancia generada de la relación *freeSpace: **Int** -> Time*.

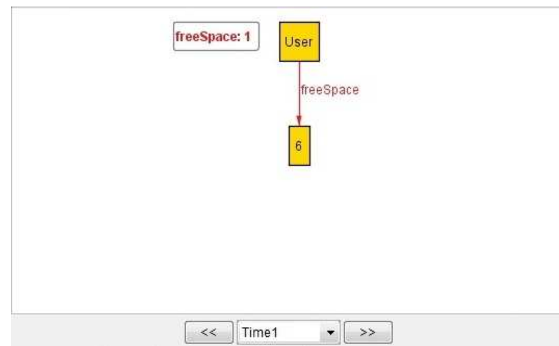

Figura 3.2: Instancia generada de la relación *freeSpace: **Int one** -> Time*.

3.1.2. Mensajes

El *tamaño* de los mensajes es uno de los atributos que nos interesa conocer, pues a través de este atributo las cuentas de los usuarios se irán saturando, para esto, creamos una relación que asocie a los mensajes con un tamaño a un determinado tiempo. Utilizaremos a los números enteros para representar el tamaño de los mensajes:

size: **Int lone** -> Time

La expresión *Int lone -> Time* nos dice que asociamos a cada elemento de *Time* con a lo más un entero. Utilizamos la declaración *lone*, porque en un principio los mensajes no han sido redactados y en ese momento no tienen asociado un peso, pero cuando los mensajes son utilizados por alguna operación en un cierto tiempo, es cuando se define el tamaño de los mensajes.

3.1.3. Servidor

El servidor es la entidad encargada de hacer llegar todos los mensajes envíados a su destino y de notificar a los usuarios que han llegado al límite del espacio de su cuenta. Para que el servidor pueda hacer llegar los mensajes necesitamos el mensaje enviado, el destinatario y el tiempo en el que va a ser recibido:

```
receiver: Message -> User -> Time
```

Para los mensajes reenviados es exactamente la misma idea que la de los mensajes enviados, pero para poder hacer una distinción entre ambos casos crearemos una nueva relación:

```
receiverForward: Message -> User -> Time
```

El servidor debe enviar un mensaje de advertencia a aquellos usuarios que hayan llegado a límite de su espacio en un determinado tiempo:

```
warning: Message -> User -> Time
```

Pero este no es un mensaje cualquiera, es un mensaje que sólo el servidor es capaz de enviar, mas que un mensaje es como una cierta notificación que el servidor le muestra a los usuarios, la cual no tiene definido ningún atributo que muestre su tamaño y tampoco puede ser tomado en cuenta por ninguna operación como se podría hacer con un mensaje cualquiera, por esta razón, definimos a este mensaje especial como sigue:

```
one sig MessageWarning extends Message{} { no size }
```

3.2. Trazas

Como se vió en el ejemplo de la sección 2.3.3, las trazas nos permiten crear de cierta manera un modelo dinámico. En nuestro modelo, lo que queremos es ver como funciona un sistema de correo electrónico con respecto al tiempo. Lo primero que necesitamos es importar la biblioteca *ordering* y aplicarle un orden total a la signatura *Time*:

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

Ahora, necesitamos una regla que nos permita desplazarnos ordenadamente sobre el tiempo. El objetivo es verificar las propiedades (*operaciones*) de nuestro modelo, para eso utilizaremos una *traza*:

```
fact Traces{
    init[T/first[]]
    all t: Time - T/last[] |
        let t' = T/next[t] |
            some u: User, m: Message |
                operación_1[t, t', u, m] or ... or operación_N[t, t', u, m]
}
```

El hecho *Trace* nos muestra la estructura de una *traza*, la cual nos dice que para el primer momento en el tiempo se debe cumplir una condición inicial dada por el predicado *init[]*, después, para todos los tiempos (excepto para el último) definimos una transición como el paso de un tiempo a otro mediante el predicado *next[]*[2], por último, una serie de predicados que serán las operaciones que los usuarios aplicarán a los mensajes. Una vez que tenemos nuestra *traza*, definimos el estado inicial:

```
pred init(t: Time){
    all u: User{
        no u.write.t
        no u.send.t
        no u.forward.t
        no u.inbox.t
        no u.sentMail.t
        no u.drafts.t
        no u.spam.t
        no u.trash.t
        no u.messageStatus.t
        u.freeSpace.t = 7
    }
    all m: Message | no m.size.t
    no Server.receiver.t and no Server.receiverForward.t and no Server.warning.t
}
```

Básicamente lo que queremos es que para el primer momento del tiempo no se haya realizado ninguna operación, por eso todas las propiedades de los usuarios, mensajes y el servidor son vacias, excepto *freeSpace*, la cual nos dice que el espacio libre de la cuenta de los usuarios es de 7[3].

3.3. Operaciones sobre los mensajes

Las operaciones le permiten a nuestro modelo realizar una transición de un estado a otro a través de la traza definiendo al mismo tiempo el comportamiento de cada estado, en donde cada estado es un elemento de la signatura *Time*. Caba aclarar que la traza nos permite realizar una sola transición de un tiempo a otro.

Cada operación, según su definición, altera una o varias relaciones del modelo, este cambio en las relaciones modela el comportamiento de nuestro sistema de correo electrónico y además nos dice qué operación se aplicó. Cuando se realiza una transición no siempre se toman en cuenta todas las relaciones, pero el analizador Alloy puede asignarles un valor arbitrario aunque este no haya sido definido. Esto no es conveniente pues se estarían modelando acciones que no corresponden con la operación en cuestión, lo que necesitamos es que siempre que ocurra una transición se al-

[2] Como los elementos de la signatura *Time* están ordenados, el elemento que se obtiene al aplicar este predicado es el sucesor del tiempo en el que se encuentre el modelo.

[3] Se elige el número 7 por que es el entero positivo más grande que genera el analizador Alloy predeterminadamente.

tere únicamente el estado de las relaciones implicadas y se mantenga el estado del resto de las relaciones. Esto lo conseguimos con el siguiente predicado:

```
pred keepState(t, t′: Time, a, b, c, d, e, f, g, h, i, j, k, l, m, n: Int){
    a = 1 implies write.t′ = write.t
    b = 1 implies send.t′ = send.t
    c = 1 implies forward.t′ = forward.t
    d = 1 implies inbox.t′ = inbox.t
    e = 1 implies sentMail.t′ = sentMail.t
    f = 1 implies drafts.t′ = drafts.t
    g = 1 implies spam.t′ = spam.t
    h = 1 implies trash.t′ = trash.t
    i = 1 implies messageStatus.t′ = messageStatus.t
    j = 1 implies freeSpace.t′ = freeSpace.t
    k = 1 implies size.t′ = size.t
    l = 1 implies receiver.t′ = receiver.t
    m = 1 implies receiverForward.t′ = receiverForward.t
    n = 1 implies warning.t′ = warning.t
}
```

Como todas las relaciones del modelo están asociadas con el tiempo, todas son candidatas a sufrir un cambio aleatorio siempre que se aplique una operación, a menos que especifiquemos qué relaciones queremos que cambien su valor y qué relaciones deben de mantener su valor actual. El predicado *keepState* recibe como parámetros a t y t' elementos de *Time* que representan una transición, y a dieciséis enteros representados con una letra diferente del alfabeto, cada letra asociada a una relación del modelo, donde si su valor es igual a “1” quiere decir que el valor actual de dicha relación se mantiene de t a t', en cualquier otro caso (por convención “0”) no se está obligando a que la relación mantenga su valor actual.

3.3.1. Redactar

Para modelar que un usuario redacta un mensaje, construiremos el predicado *composeMail*. Lo que tenemos que hacer en este predicado es sencillo, simplemente debemos modificar las relaciones *write* y *size*, agregando el mensaje y al usuario que lo está redactando, además definimos en ese mismo instante el peso del mensaje. Al realizar esta transición sólo modificaremos el valor de las relaciones en cuestión y conservaremos el valor actual del resto de las relaciones mediante el predicado *keepState*:

```
1 pred composeMail(t, t′: Time, u: User, m: Message){
2     write.t′ = write.t + u -> m
3     one weight: Int | size.t′ = size.t + m -> weight
4     keepState[t, t′, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1]
5 }
```

El predicado *composeMail* nos dice que de t a t' el usuario u redacta el mensaje m. En la línea 2 decimos que la relación *write* en el tiempo t' es igual a lo que ya había anteriormente (todos los

mensajes redactados con sus respectivos autores) más el usuario *u* asociado al mensaje *m* (*u* -> *m*), esto quiere decir que el usuario *u* redactó el mensaje *m*. Después, en la línea 3 definimos a la variable *weight* como un entero asociado con el mensaje *m* (*m* -> *weight*), entonces el peso del mensaje es el valor de la variable *weight* (*m* -> *weight*) y la agregamos a la relación *size* en el tiempo t'. Por último, en la línea 4 mantenemos el estado de todas las relaciones del tiempo t a t' excepto para *write* y *size*, por lo cual, el valor de las columnas tres y catorce (w*rite* y *size* respectivamente) es distinto de "1".

Una vez definida la primera operación, la agregamos a la traza:

```
fact Traces{
    init[T/first[]]
    all t: Time - T/last[] |
        let t′ = T/next[t] |
            some u: User, m: Message |
                composeMail[t, t′, u, m]
}
```

Ahora, ejecutamos el modelo para ver las instancias o simulaciones generadas por el analizador Alloy:

```
pred show(){}
run show
```

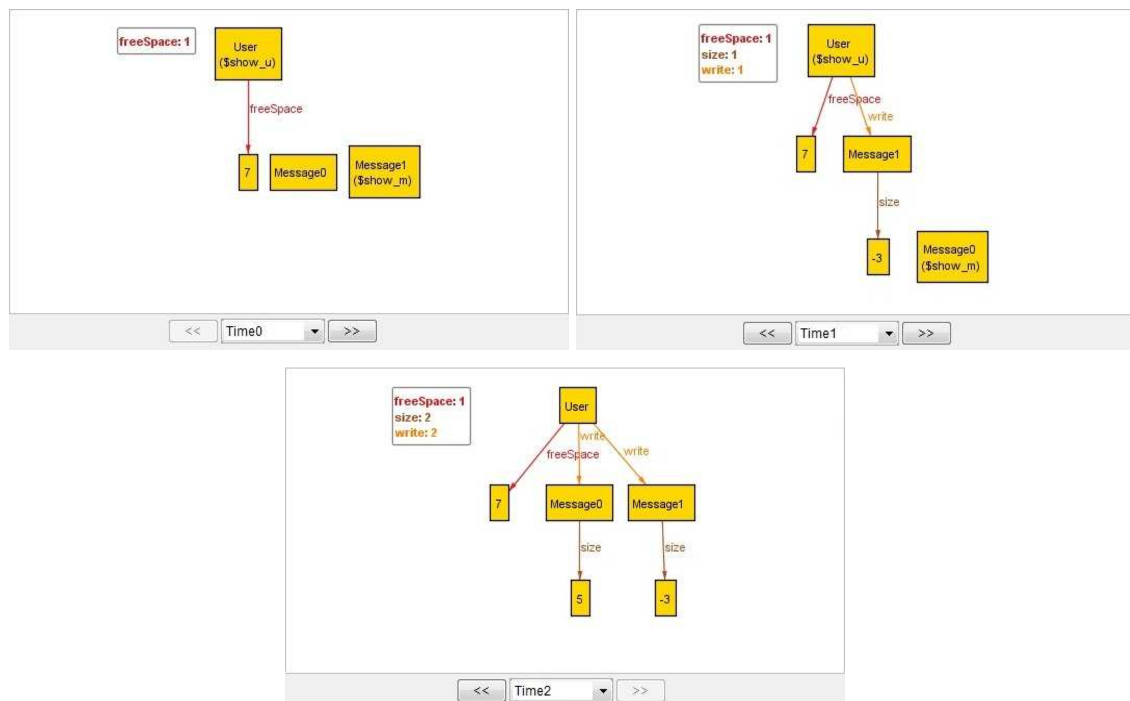

Figura 3.3: Una instancia del modelo mediante la traza: el estado inicial se encuentra en el panel izquierdo de la parte superior, el resto de los paneles muestra las siguientes instancias a través del tiempo .

La figura 3.3 nos muestra el comportamiento de la operación *composeMail*, en el tiempo *t0* el modelo se encuentra en el estado inicial, en el tiempo *t1* se crean las relaciones *User -> Message1* y *Message1 -> -3* lo cual significa que el usuario redacto *Message1* y que su tamaño es -3, y en el tiempo *t2* el usuario redacta *Message0* y su tamaño es 5. Inmediatamente observamos que existen dos errores:

1. Un mensaje no puede tener como tamaño a un número negativo.
2. Cuando un usuario redacta un mensaje en un cierto tiempo, no puede redactar otro mensaje sin antes haber enviado o guardado el mensaje actual, esto es lo que ocurre del tiempo 1 al tiempo 2.

Para evitar que un mensaje tenga un peso negativo, modificamos la regla que define el tamaño del mensaje diciendo que los mensajes tienen un peso mayor o igual a 1:

```
one weight: Int{
    weight >= 1
    size.t' = size.t + m -> weight
}
```

Agregando la siguiente restricción nos aseguramos que los usuarios no se encuentran redactando otro mensaje cuando quieren redactar un nuevo mensaje:

```
u.write.t = none
```

Hay que aclarar que nada está limitando a que los usuarios redacten un mensaje que ya ha redactado otro usuario y que más adelante, cuando los usuarios puedan guardar los mensajes redactados en lugar de enviarlos inmediatamente, estos mensajes sean redactados por otros usuarios. Para evitar que esto pase hay que especificar que el mensaje no ha sido redactado ni guardado en el predicado *composeMail*:

```
m not in User.write.t + User.drafts.t
```

La figura 3.4 nos muestra el comportamiento del modelo con las nuevas restricciones. Podemos **verificar** que el modelo funciona adecuadamente analizando el resultado de las siguientes aserciones:

Los mensajes tienen un peso positivo:

```
assert validateSizeMessage{
    all m: Message, t: Time |
        let weight = m.size.t |
            some weight implies weight >= 1
}check validateSizeMessage for 5
```

Los usuarios no pueden redactar un mensaje al mismo tiempo en el que están redactando otro:

```
assert validateComposeMail{
    all u: User | no m: Message, t: Time | some u.write.t and composeMail[t, t.next, u, m]
```

}**check** validateComposeMail **for** 5

Los mensajes redactados sólo pueden tener un autor:

```
assert validateComposeMail2{
    all m: Message, t: Time |
        let author = write.t.m |
            some author implies #author = 1
}check validateComposeMail2 for 5
```

Los usuarios no pueden redactar mensajes ya redactados, incluso si el autor es el mismo

```
assert validateComposeMail3{
    no u: User, m: Message | one t: Time |
        let author = write.t.m |
            some author and
                (composeMail[t, t.next, author, m] or composeMail[t, t.next, u, m])
}check validateComposeMail3 for 5
```

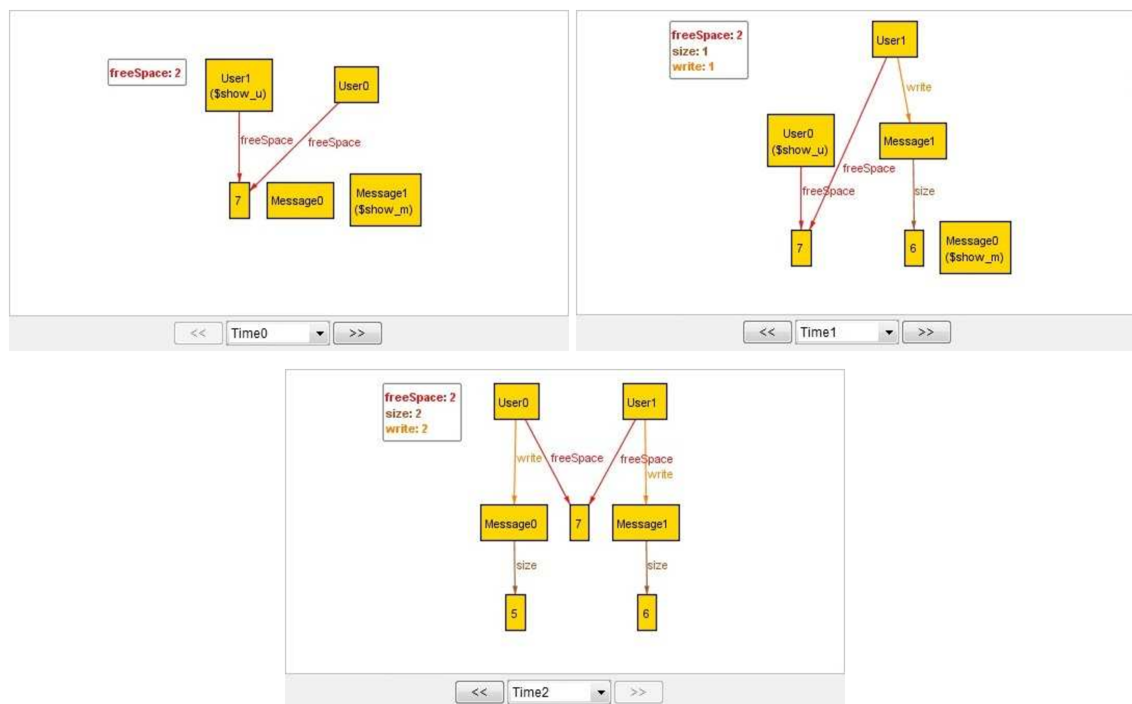

Figura 3.4: La operación de *composeMail* a través del tiempo.

3.3.2. Guardar

Los usuarios sólo pueden guardar los mensajes que han redactado, por esto, es requisito que el usuario y el mensaje se encuentren en la relación *write* en el mismo tiempo en que el usuario desea guardar el mensaje. Cuando los usuarios guardan los mensajes lo que ocurre es que el espacio libre de sus respectivas cuentas va disminuyendo, para conseguir esta acción es necesario declarar un

conjunto de reglas que nos permitan modelarlo. Primero necesitamos importar la biblioteca que nos permite trabajar con enteros:

```
open util/integer
```

Ahora implementamos un predicado que se encargue de manejar el espacio de las cuentas de los usuarios:

```
1 pred accountManagerSpace(t, t′: Time, u: User, m: Message, tipo: Int){
2     tipo = 1 implies{
3         let currentSpace = u.freeSpace.t, newSpace = currentSpace.sub[m.size.t] |
4             freeSpace.t′ = freeSpace.t - u -> currentSpace + u -> newSpace
5     }else{
6         let currentSpace = u.freeSpace.t, newSpace = currentSpace.add[m.size.t] |
7             freeSpace.t′ = freeSpace.t - u -> currentSpace + u -> newSpace
8     }
9 }
```

El predicado *accountManagerSpace* recibe como parámetros: dos momentos de tiempo (que representan la transición de un estado a otro), un usuario, un mensaje que alterará el espacio libre de la cuenta del usuario y un entero que nos dirá que tipo de operación realizar (1). Si el entero recibido es igual a "1" quiere decir que la operación que realizaremos es restar el peso del mensaje al espacio libre de la cuenta (2), guardamos en una variable el espacio actual de la cuenta y en otra variable el nuevo espacio restando el peso del mensaje al peso actual de la cuenta (3), por últmo actualizamos el espacio de la cuenta (4). Si el entero recibido es distinto de "1" (Por convención será el número "2"), entonces la operación que realizaremos será eliminar el peso del mensaje del espacio libre de la cuenta, es decir, liberamos espacio (5). De nuevo guardamos en una variable el espacio actual de la cuenta y en otra variable el nuevo espacio sumando el peso del mensaje al espacio actual de la cuenta (6) y por último, actualizamos el espacio libre de la cuenta (7).

Ahora, declaramos un hecho que evite todo el tiempo que los usuarios sobrepasen el espacio de su cuenta:

```
fact Rules{
    all u: User, t: Time | not accountOverLimit[t, u]
}
```

La función del prdicado *accountOverLimit* es decirnos si algún usuario ha sobrepasado el espacio libre de su cuenta en un determinado tiempo:

```
pred accountOverLimit(t: Time, u: User){
    u.freeSpace.t < 0
}
```

Sólo nos resta modelar como los usuarios guardarán los mensajes:

```
1  pred saveMail(t, t′: Time, u: User, m: Message){
2      u.write.t = m
3      write.t′ = write.t - u -> m
4      drafts.t′ = drafts.t + u -> m
5      accountManagerSpace[t, t′, u, m, 1]
6      keepState[t, t′, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1]
7  }
```

Del tiempo t al tiempo t', el mensaje que queremos guardar tiene que ser el que estamos redactando en el tiempo *t* (1), el usuario ya redactó el mensaje *m* y se guardó en la bandeja de *Drafts* (3 y 4), se actualiza el espacio libre de la cuenta del usuario *u* (5) y se mantiene el estado de todas las relaciones excepto para las relaciones *write* y *drafts* (6).

Siempre que que hayamos construido una nueva operación, debemos agregar esta a la traza:

```
fact Traces{
    ...
        composeMail[t, t′, u, m] or saveMail[t, t′, u, m]
}
```

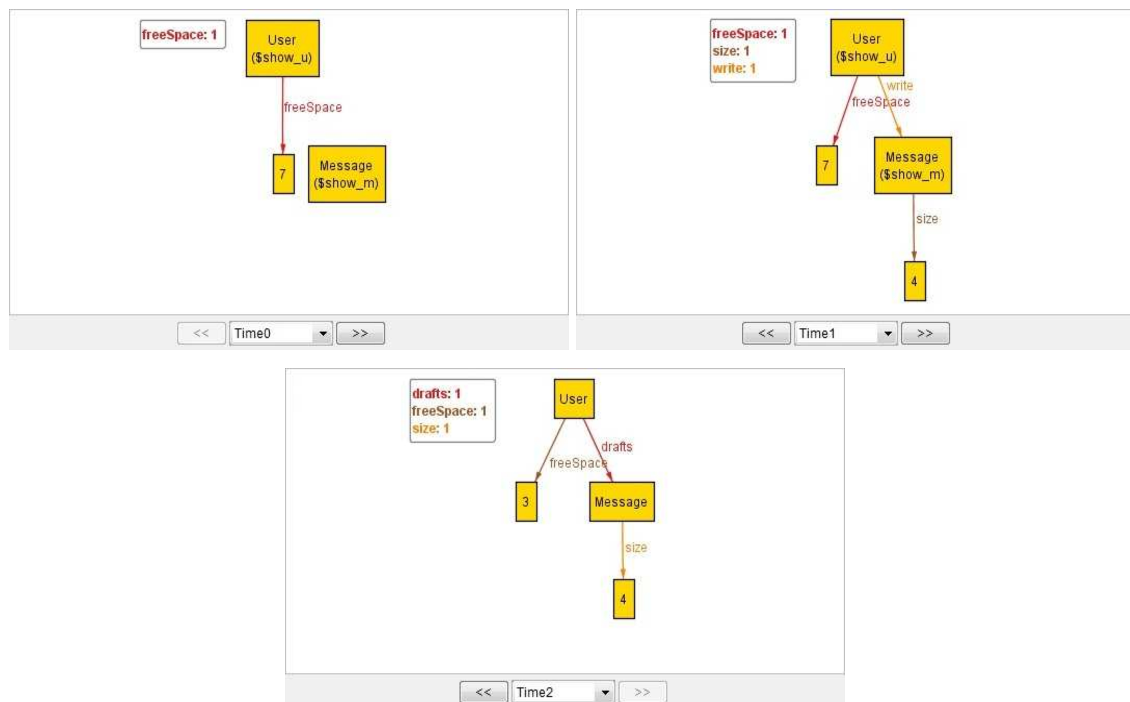

Figura 3.5: La operación *saveMail* a través del tiempo.

La figura 3.5 nos muestra como los usuarios guardan un mensaje: en el tiempo *t0* el usuario no ha realizado ninguna acción, en el tiempo *t1* el usuario redactó un mensaje (*User* -> *Message* se agrega a la relación *write*) y en el tiempo *t2* el usuario guardó el mensaje que había redactado (*User* -> *Message* se agrega a la relación *drafts*).

3.3.3. Descartar

Cuando los usuarios redactan un mensaje, además de guardarlo también podrán descartarlo. El predicado correspondiente es:

```
1 pred discardMail(t, t′: Time, u: User, m: Message){
2     u.write.t = m
3     write.t′ = write.t - u -> m
4     size.t′ = size.t - m -> (m.size.t)
5     keepState[t, t′, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1]
6 }
```

El usuario debe estar redactando el mensaje que quiere descartar (2), se descarta el mensaje (3), se elimina el peso que el mensaje tenía (4) y finalmente se mantiene el estado del resto de las relaciones excepto para las relaciones *write* y *size* (5).

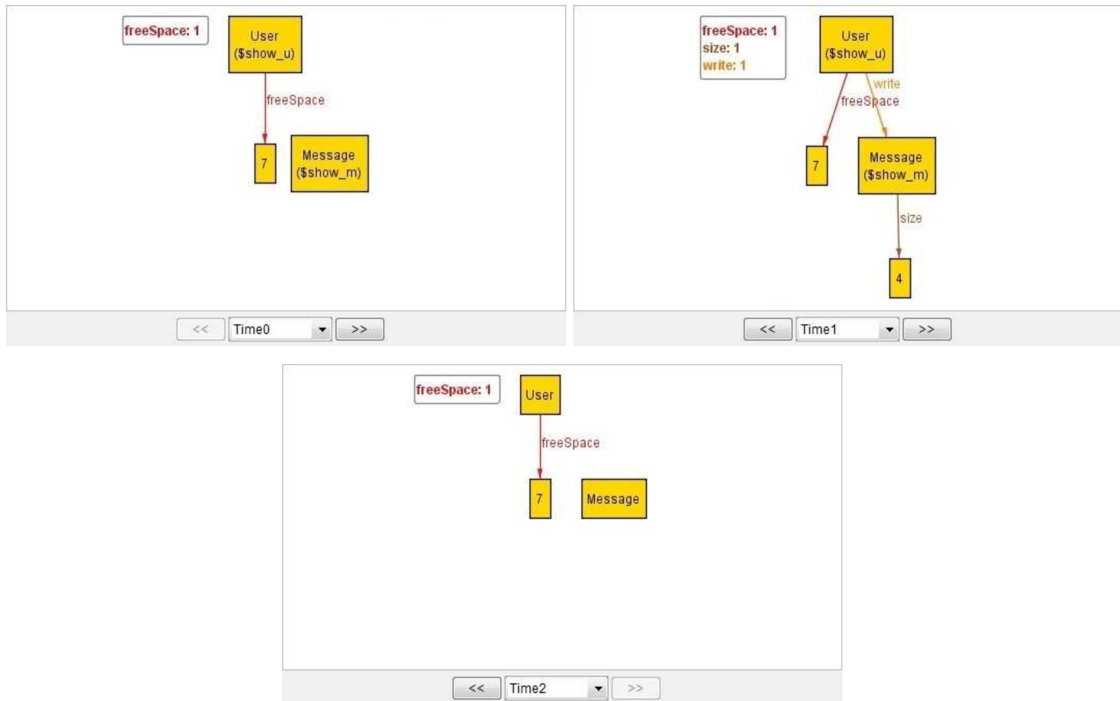

Figura 3.6: La operación *discardMail* a través del tiempo.

En la figura 3.6 se muestra el comportamiento de la operacion *discardMail*. En el tiempo *t0* el usuario no ha realizado ninguna acción, en el tiempo *t1* el usuario redactó un mensaje (*User -> Message* se agrega a la relación *write*) y en el tiempo *t2* el usuario descartó el mensaje que había redactado *(User -> Message* y *Message -> 4* se eliminan de las relaciones *write* y *size* respectivamene).

3.3.4. Enviar

Enviar un correo electrónico es una de las operaciones más importantes en un sistema de correo electrónico, pues es el objetivo de estos sistemas. Lo que vamos a hacer es crear un conjunto de reglas que le permita a los usuarios realizar esta acción:

```
1  pred sendMail(t, t′: Time, u: User, m: Message){
2      m in u.write.t + u.drafts.t
3      m not in User.send.t.Server + Server.receiver.t.User
4      send.t′ = send.t + u -> m -> Server
5      sentMail.t′ = sentMail.t + u -> m
6      messageStatus.t′ = messageStatus.t + u -¿Read -> m
7      m in u.write.t implies{
8          write.t′ = write.t - u -> m
9          accountManagerSpace[t, t′, u, m, 1]
10         keepState[t, t′, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1]
11     }else{
12         u.write.t = none
13         drafts.t′ = drafts.t - u -> m
14         keepState[t, t′, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1]
15     }
16 }
```

Los usuarios sólo pueden enviar mensajes que acaban de redactar o que hayan guardado en algún momento (2), los usuarios no pueden enviar mensajes que ya enviaron[4] (3), el mensaje se envía al servidor (4), el mensaje enviado se mueve a la bandeja de *Sent Mail* del usuario (5), se define el estado del mensaje con respecto al usuario que lo envió, para este caso el mensaje tiene el estado de *leído* (6). Se comprueba el caso en donde el usuario envia el mensaje sin guardarlo en la bandeja de *Drafts* (7), como el mensaje se movió a la bandeja de enviados, el usuario y el mensaje se eliminan de la relación *write* (8), el mensaje nunca se guardó y ahora ocupa un cierto espacio, por lo que se hace ese ajuste a la cuenta del usuario (9), se mantiene el estado de todas las relaciones excepto para las relaciones *send*, *sentMail*, *messageStatus*, *freeSpace* y *write* (10). Ahora se comprueba el caso en donde el usuario envia el mensaje desde su badeja de *Drafts* (11), el usuario no debe estar redactando ningún otro mensaje al mismo tiempo en que desea enviar un mensaje guardado (12), como el mensaje se movió a la bandeja de enviados, el usuario y el mensaje se eliminan de la relación *drafts* (13), finalmente se mantiene el estado de todas las relaciones excepto para las relaciones *send*, *sentMail*, *messageStatus* y *drafts* (14).

En la figura 3.7 se muestra el comportamiento de la operación *sendMail* sin guardar el mensaje en la bandeja de *Drafts*. En el tiempo *t0* el usuario no ha realizado ninguna acción, en el tiempo *t1* el usuario redactó un mensaje (*User -> Message* se agrega a la relación *write*) y en el tiempo *t2* el mensaje redactado se envía al servidor, el mensaje tiene el estado de *leído* y se movió a la bandeja de enviados (*User -> Message -> Server*, *User -> Read -> Message* y *User -> Message* se agregan a las relaciones *send*, *messageStatus* y *sentMail* respectivamente).

En la figura 3.8 se muestra el comportamiento de la operacion *sendMail* con el mensaje guardado en la bandeja de *Drafts*. En el tiempo *t0* el usuario no ha realizado ninguna acción, en el tiempo *t1* el usuario redactó un mensaje (*User -> Message* se agrega a la relación *write*), en el tiempo

[4] Para eso, después crearemos una operación que reenvie mensajes.

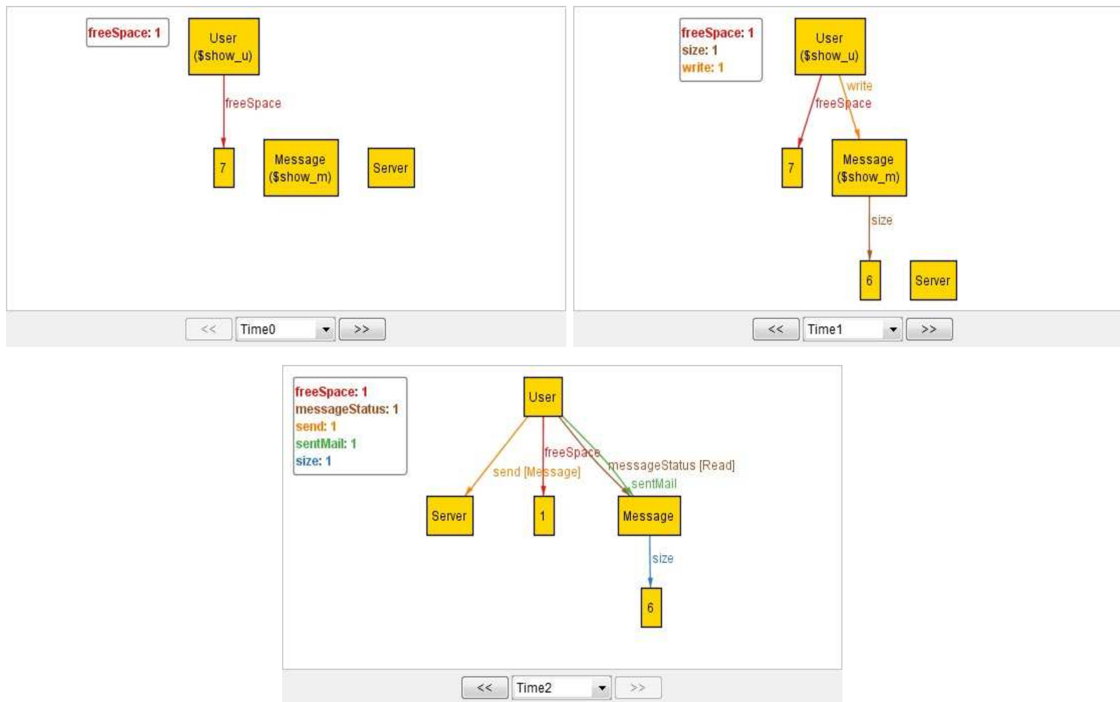

Figura 3.7: La operación *sendMail* sin guardar el mensaje en la bandeja de *Drafts*.

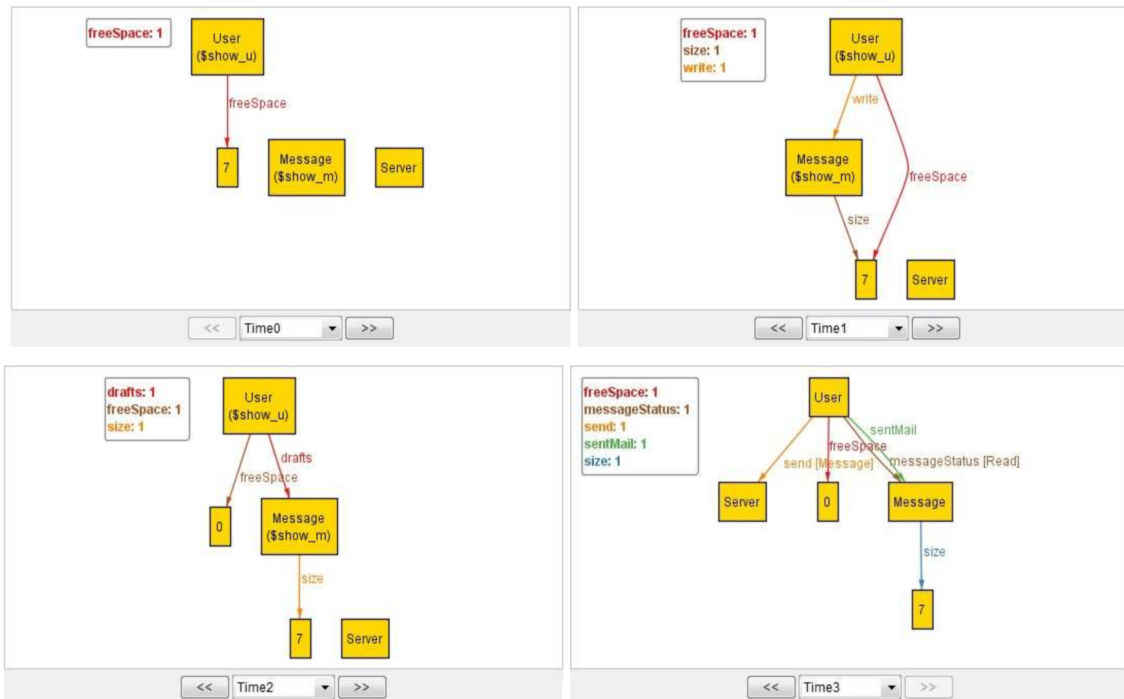

Figura 3.8: La operación *sendMail* con el mensaje guardado en la bandeja de *Drafts*.

t2 el usuario guardó el mensaje redactado en la bandeja de *Drafts* (*User -> Message* se agrega a la relación *drafts* y se elimina de la relación *write*) y en el tiempo *t3* el usuario envió el mensaje guardado hacia el servidor, el mensaje tiene el estado de *leído* y se movió a la bandeja de enviados (*User -> Message -> Server*, *User -> Read -> Message* y *User -> Message* se agregan a las relaciones

send, *messageStatus* y *sentMail* respectivamente).

3.3.5. Recibir

Una vez que los mensajes son enviados, tenemos que hacer que nuestro sistema de correo electrónico le permita a lo usuarios recibir los mensajes enviados:

```
1  pred receiveMail(t, t′: Time, u: User, m: Message){
2      m not in Server.receiver.t.u
3      m in User.send.t.Server
4      receiver.t′ = receiver.t + Server -> m -¿u
5      messageStatus.t′ = messageStatus.t - u -> Status -> m + u -> Unread -> m
6      accountManagerSpace[t, t′, u, m, 1]
7      {inbox.t′ = inbox.t + u -> m
8          keepState[t, t′, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1]}
9      or
10     {spam.t′ = spam.t + u -> m
11         keepState[t, t′, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1]}
12 }
```

Los usuarios no pueden recibir mensajes que ya recibieron[5] (2), para que los usuarios puedan recibir un mensaje, primero tuvo que haber sido enviado por algún usuario (3), el servidor envía el mensaje al destinatario (4), el mensaje enviado llega al destinatario con el estado de *No leído* (5), se realiza un ajuste al espacio libre de la cuenta del usuario en cuestión (6). Cuando un mensaje es recibido por un usuario puede ocurrir alguno de los dos siguientes casos: El mensaje llega a la bandeja de recibidos (7) y se mantiene el estado de todas las relaciones excepto para *receiver*, *messageStatus*, *freeSpace* e *inbox* (8). O el mensaje llega al *spam* (10) y se mantiene el estado de todas las relaciones excepto para *receiver*, *messageStatus*, *freeSpace* y *spam* (11).

En al figura 3.9 se muestra el comportamiento de la operacion *receiveMail*, en donde del tiempo *t0* al tiempo *t1*, *User1* redacta a *Message*, de *t1* a *t2 User1* envia hacia el servido *Message* y de *t2* a *t3 User0* recibe *Message*. El servidor hace llegar el mensaje enviado al destinatario (*Server* -> *Message* -> *User0* se agrega a la relación *receiver*), es decir, el mensaje llega a la bandeja de recibidos del usuario (*User0* -> *Message* se agrega a la relación *inbox*), al llegarle el mensaje, *User0* no ha leído el mensaje (*User0* -> *Unread* -> *Message* se agrega a la relación *messageStatus*), como el peso del mensaje es “7”, el espacio libre de la cuenta del usuario *User0* ha llegado a su límite (*User0* -> *0* se agrega a la relación *freeSpace*).

Hasta este momento, los mensajes ya pueden ser enviados y recibidos, una vez que ocurre esto el modelo no está limitando a los usuarios a no redactar un mensaje que ya fue enviado y hasta quizá recibido, por lo tanto, debemos agregar una regla más al predicado *composeMail* para evitar que esto ocurra:

```
m not in User.send.t.Server + Server.receiver.t.User
```

[5] Para esto se modelará el predicado *receiveForwardMail* que modela como los usuarios reciben mensajes reenviados.

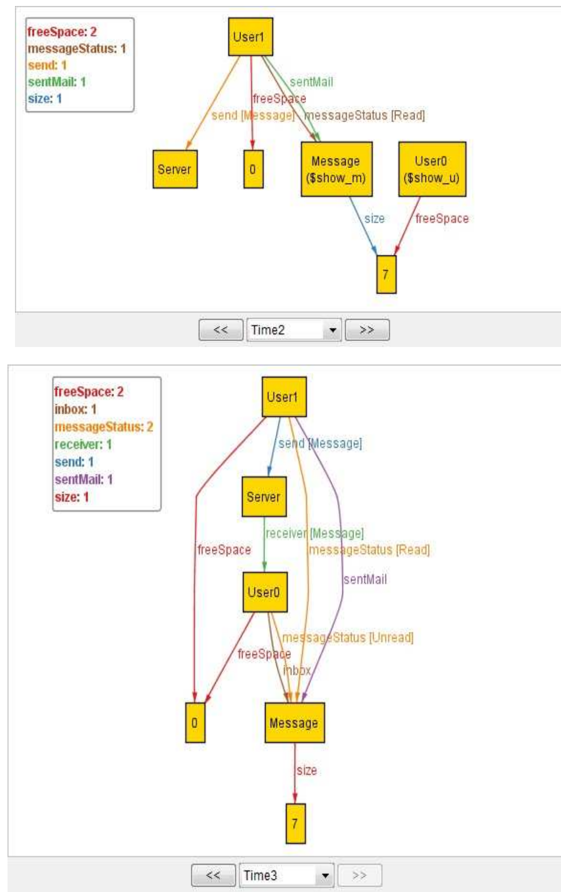

Figura 3.9: La operación *receiveMail*.

Para nuestro modelo, vamos a restringir a los usuarios a que no puedan enviarse mensajes a si mismos pues lo que nos interesa es modelar el comportamiento de estas operaciones entre los usuarios. Esto lo conseguimos agregando la siguiente regla a nuestro hecho *Rules*:

```
all m: Message, t: Time |
    let from = send.t.Server.m, to = m.(Server.receiver.t) |
        some from implies from not in to
```

3.3.6. Recibir un mensaje de advertencia

Ahora que los usuarios son capaces de crear y recibir mensajes, sus cuentas pueden saturarse muy fácilmente. Para mantener al tanto de esta situación a los usuarios, el servidor debe ser capaz de notificarles cuando esto ocurra, por lo cual, primero necesitamos un predicado que nos diga cuando el espacio libre de la cuenta de un usuario llegue a su límite:

```
pred accountLimit(t: Time, u: User){
    u.freeSpace.t = 0
}
```

Después, el predicado que se encargue de enviar el mensaje de advertencia:

```
1  pred receiveMessageWarning(t, t′: Time, u: User, m: Message){
2      accountLimit[t, u]
3      m = MessageWarning
4      let notice = Server -> m -> u{
5          notice not in warning.t
6          warning.t′ = warning.t + notice
7      }
8      inbox.t′ = inbox.t + u -> m
9      keepState[t, t′, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0]
10 }
```

Solamente se puede recibir el mensaje de advertencia si la cuenta de los usuarios llega a su límite (2), el mensaje que se recibe es el mensaje de advertencia (3), no se puede recibir más de una vez el mensaje (5), el servidor envía el mensaje (6) y el usuario lo recibe (8), por último se mantiene el estado de todas las relaciones excepto para las relaciones *inbox* y *warning*.

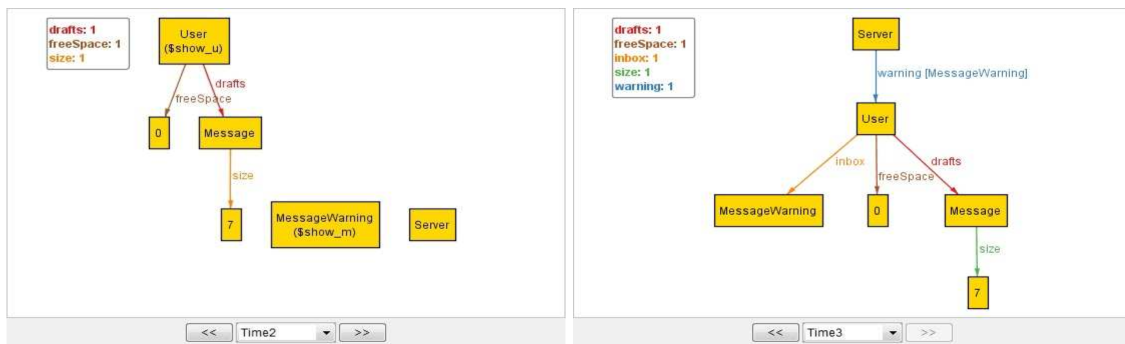

Figura 3.10: La operación *receiveMessageWarning*.

En la figura 3.10 se muestra el comportamiento de la operación *receiveMessageWarning*, en donde del tiempo *t0* al tiempo *t1* el usuario redacta un mensaje, de *t1* a *t2* lo guarda en su bandeja de *Drafts*, y de *t2* a *t3* el servidor le envía un mensaje de advertencia al usuario.

3.3.7. Leer

Ahora que los usuarios ya tienen mensajes en sus cuentas, construiremos algunas operaciones que puedan manipularlos.

Leer[6] un mensaje es una función de suma importancia, pues de lo contrario no tendría sentido un sistema de correo electrónico:

```
1  pred readMail(t, t′: Time, u: User, m: Message){
2      u.write.t = none
3      let status = u -> Unread -> m, newStatus = u -> Read -> m{
4          status in messageStatus.t
```

[6] Sólo modelaremos esta acción con los mensajes no leídos, por cuestiones prácticas no modelaremos cuando un usuario lee un mensaje ya leído.

```
5         messageStatus.t′ = messageStatus.t - status + newStatus
6     }
7     keepState[t, t′, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1]
8 }
```

Los usuarios no pueden leer un mensaje cuando al mismo tiempo el usuario se encuentra redactando otro mensaje (2), se define el estado actual del mensaje y el nuevo estado que tendrá (3), si decimos que el estado actual del mensaje existe en la relación *messageStatus* y además es *No leído*, estamos asegurando que el mensaje se ha recibido y que no ha sido leído (4), se cambia el estado del mensaje de *No leído* a *Leído* (5).

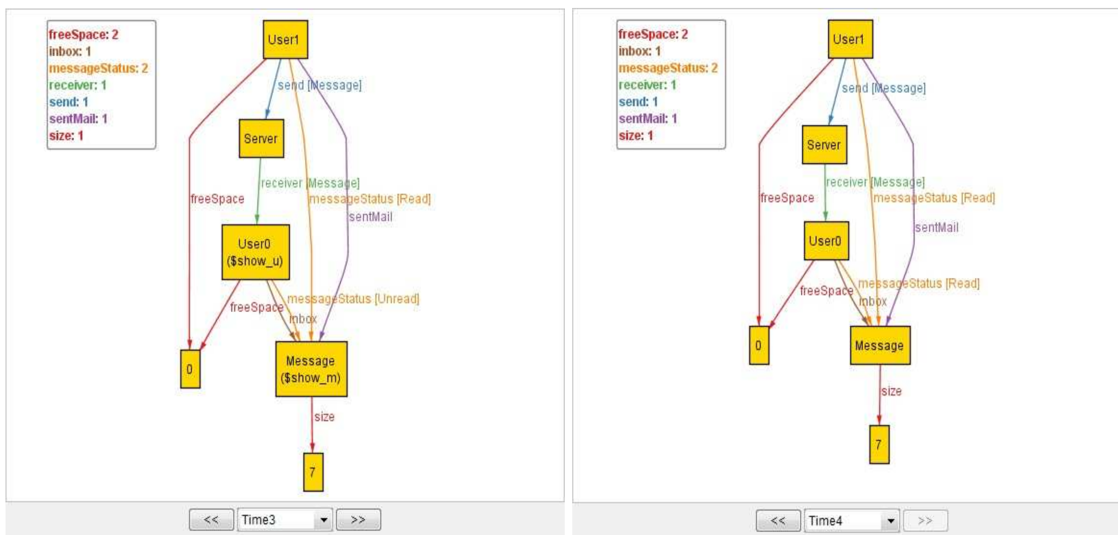

Figura 3.11: La operación *readMail*.

En al figura 3.11 se muestra el comportamiento de la operación *readMail*, en donde del tiempo *t0* al tiempo *t1 User1* redacta a *Message*, de *t1* a *t2 User1* envía hacia el servidor *Message*, de *t2* a *t3 User0* recibe *Message* y de *t3* a *t4 User0* lee *Message*, por lo que *User0 -> Read -> Message* toma el lugar de *User0 -> Unread -> Message* en la relación *messageStatus*.

3.3.8. Reenviar

Como mencionamos anteriormente, los usuarios no pueden enviar un mensaje ya enviado, para que esto ocurra construiremos un predicado que permita enviar mensajes enviados y hasta enviar los mensajes recibidos:

```
1 pred forwardMail(t, t′: Time, u: User, m: Message){
2     u.messageStatus.t.m = Read
3     u.write.t = none
4     forward.t′ = forward.t + u -> m -> Server
5     sentMail.t′ = sentMail.t + u -> m
6     keepState[t, t′, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1]
7 }
```

Para poder reenviar un mensaje primero se debe leer[7] (2), los usuarios no pueden reenviar un mensaje al mismo tiempo en que están redactando otro mensaje (3), el usuario reenvía el mensaje hacia el servidor (4), el mensaje se agrega a la bandeja de enviados del usuario (5), se mantiene el estado de las relaciones excepto para *forward* y *sentMail* (6).

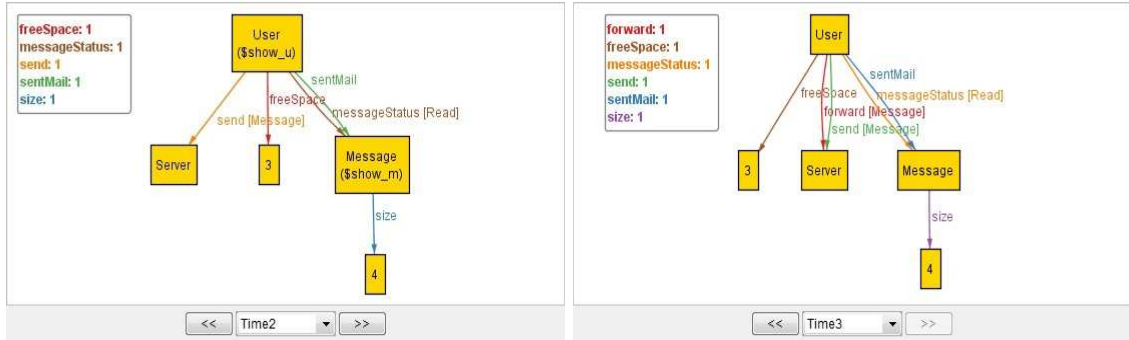

Figura 3.12: La operación *forwardMail*.

En al figura 3.12 se muestra el comportamiento de la operacion *forwardMail*, en donde del tiempo *t0* al tiempo *t1* el usuario redacta un mensaje, *t1* a *t2* el usuario envía el mensaje hacia el servidor y de *t2* a *t3* el usuario reenvía el mensaje hacia el servidor.

Para seguir con el estándar de nuestra especificación, hay que evitar que los usuarios se reenvien mensajes a si mismos. Agregamos la siguiente restricción al hecho *Rules*:

```
all m: Message, t: Time |
    let from = forward.t.Server.m, to = m.(Server.receiverForward.t) |
        some from implies from not in to
```

3.3.9. Recibir un mensaje reenviado

Ahora, los usuarios deben recibir los mensajes reenviados:

```
1  pred receiveForwardMail(t, t′: Time, u: User, m: Message){
2      m in User.forward.t.Server
3      m not in Server.receiverForward.t.u
4      receiverForward.t′ = receiverForward.t + Server -> m -> u
5      messageStatus.t′ = messageStatus.t - u -> Status -> m + u -> Unread -> m
6      accountManagerSpace[t, t′, u, m, 1]
7      {inbox.t′ = inbox.t + u -> m
8          keepState[t, t′, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1]}
9      or
10     {spam.t′ = spam.t + u -> m
11         keepState[t, t′, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1]}
12 }
```

[7] Con esta regla también estamos diciendo que el mensaje fue enviado por alguien y que ha sido recibido.

En donde el mensaje debió haber sido reenviado (2), los usuarios no pueden recibir el mismo mensaje reenviado que ya recibieron por este mismo medio (3), el usuario recibe el mensaje reenviado (4), el estado del mensaje con respecto al usuario que lo recibió es *No leído* (5), se realiza un ajuste al espacio libre de la cuenta del usuario quien recibió el mensaje (6). Cuando se recibe un mensaje, este puede llegar a la bandeja de recibidos (7) o a la bandeja del spam (10).

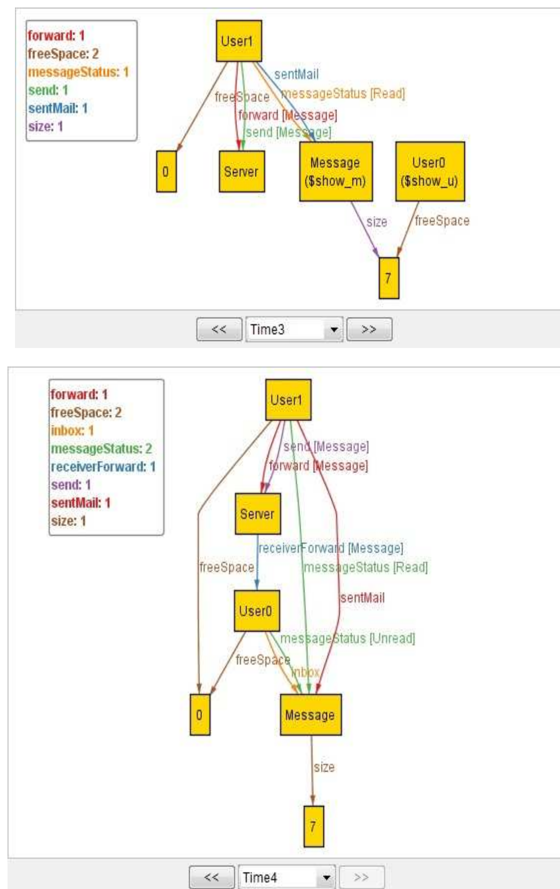

Figura 3.13: La operación *receiveForwardMail*.

En la figura 3.13 se muestra el comportamiento de la operación *receiveForwardMail*, en donde del tiempo *t2* al tiempo *t3 User1* reenvía *Message* y del tiempo *t3* al tiempo *t4 User0* recibe *Message* a su bandeja de *Inbox*.

3.3.10. Eliminar

Es muy común que los usuarios deseen eliminar los mensajes de su correo electrónico. Esta operación no hace otra cosa mas que mover el mensaje que quieren eliminar a la bandeja de basura:

```
1 pred deleteMail(t, t′: Time, u: User, m: Message){
2     m in u.inbox.t + u.sentMail.t
3     m != MessageWarning
4     u.write.t = none
5     inbox.t′ = inbox.t - u -> m
```

```
6       sentMail.t′ = sentMail.t - u -> m
7       spam.t′ = spam.t - u -> m
8       trash.t′ = trash.t + u -> m
9       keepState[t, t′, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1]
10  }
```

Los usuarios mueven a la basura sólo los mensajes que están en las bandejas de recibidos o enviados (2), no tendría sentido mover a la basura los mensajes eliminados o el spam. Los usuarios no pueden mover a la basura el mensaje de advertencia (3), no pueden eliminar un mensaje en el mismo tiempo en que están redactando otro mensaje (4), se elimina el mensaje de las bandejas en las que se encuentre[8] (5 – 7) , el mensaje se mueve a la bandeja de basura (8).

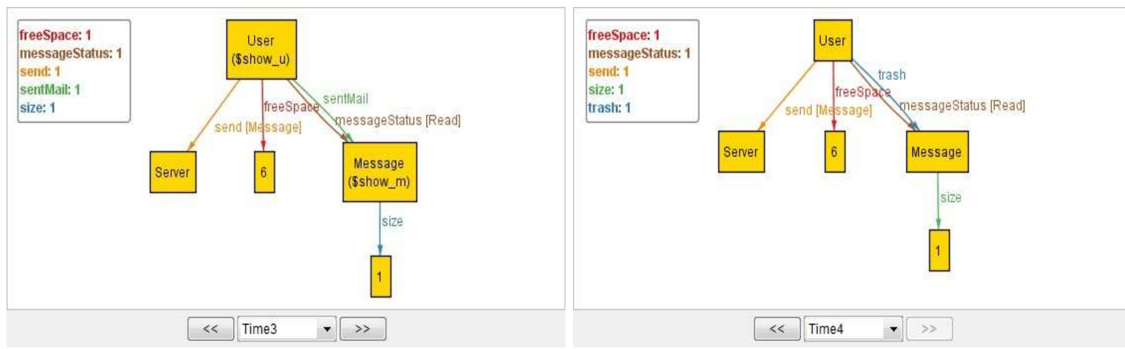

Figura 3.14: La operación *deleteMail*.

En al figura 3.14 se muestra el comportamiento de la operación *deleteMail*, en donde del tiempo *t2* al tiempo *t3* el usuario envía un mensaje y de *t3* a *t4* el usuario elimina (mueve a la basura) el mensaje enviado.

3.3.11. Eliminar de manera definitiva

¿Para qué necesitan los usuarios borrar un mensaje de forma definitiva? Lo más importante es que mediante esta operación pueden liberar espacio en su cuenta de correo electrónico. Los usuarios podrán deshacerse de los mensajes basura (mensajes eliminados), mensajes spam y hasta los mensajes guardados que ya no deseen enviar (drafts):

```
1   pred deleteMailForever(t, t′: Time, u: User, m: Message){
2       m in (u.inbox.t + u.sentMail.t + u.drafts.t + u.spam.t + u.trash.t)
3       u.write.t = none
4       m not in MessageWarning
5       accountManagerSpace[t, t′, u, m, 2]
6       m in u.drafts.t implies{
7           inbox.t′ = inbox.t - u -> MessageWarning
8           drafts.t′ = drafts.t - u -> m
9           size.t′ = size.t - m -> m.size.t
```

[8] Aunque sólo se pueden eliminar los mensajes de *Inbox* y *Sent Mail* se puede dar el caso en donde un usuario quiera reenviar un mensaje del spam, por lo tanto este mensaje estaría también en la bandeja de enviados.

```
10          warning.t′ = warning.t - Server -> MessageWarning -> u
11          keepState[t, t′, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0]
12      }else{
13          inbox.t′ = inbox.t - (u -> m) - (u -> MessageWarning)
14          sentMail.t′ = sentMail.t - u -> m
15          spam.t′ = spam.t - u -> m
16          trash.t′ = trash.t - u -> m
17          messageStatus.t′ = messageStatus.t - u -> Status -> m
18          warning.t′ = warning.t - Server -> MessageWarning -> u
19          keepState[t, t′, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0]
20      }
21  }
```

El usuario debe tener el mensaje en alguna de sus bandejas, con esto también decimos que el mensaje ha sido enviado y/o recibido (2), el usuario no puede eliminar un mensaje si al mismo tiempo se encuentra redactando otro mensaje (3), el usuario no puede eliminar el mensaje de advertencia (4), se realiza un ajuste al espacio libre de la cuenta del usuario[9] (5), comprobamos si el mensaje que se quiere eliminar se encuentra en la bandeja de *Drafts* (6), como se encuentra en la bandeja de *Drafts* del usuario, quiere decir que no ha sido enviado y mucho menos recibido, por lo que sólo nos basta con quitarlo de la relación *drafts* (8) y eliminar el peso del mensaje (9), como el mensaje fue eliminado, debemos quitar el mensaje de advertencia si es que su cuenta esta al límite (7) y del servidor también (10). Ahora comprobamos si el mensaje está en alguna otra bandeja que no sea la de *Drafts* (12), si es que la cuenta del usuario llegó al límite, eliminamos el mensaje de advertencia de la cuenta del usuario (13) y del servidor (18). Como es posible que el mensaje se encuentre en varias bandejas, eliminamos el mensaje de todos los posibles lugares en donde se encuentre (13 – 16) y el estado del mensaje con respecto al usuario que realiza la operación (17).

En al figura 3.15 se muestra el comportamiento de la operación *deleteMailForever*, en donde del tiempo *t2* al tiempo *t3* el usuario recibe un mensaje de advertencia por tener su cuenta en el límite, y de *t3* a *t4* el usuario elimina de manera definitiva un mensaje guardado.

3.3.12. Reportar como spam

Una de las funcionalidades adicionales en un sistema de correo electrónico es darle la capacidad a los usuarios de poder marcar ciertos mensajes como correo spam y moverlos a su respectiva bandeja:

```
1  pred reportSpam(t, t′: Time, u: User, m: Message){
2      m in u.inbox.t + u.sentMail.t + u.trash.t
3      spam.t′ = spam.t + u -> m
4      u.write.t = none
5      m != MessageWarning
6      inbox.t′ = inbox.t - u -> m
```

[9] Al pasarle como parámetro el número “2” al predicado *accountManagerSpace*, estamos diciendo que se liberará espacio de la cuenta.

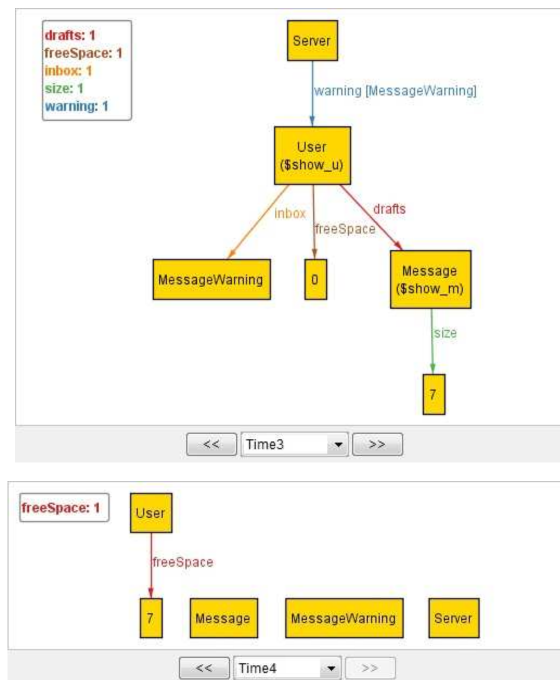

Figura 3.15: La operación *deleteMailForever*.

```
7      sentMail.t′ = sentMail.t - u -> m
8      trash.t′ = trash.t - u -> m
9      keepState[t, t′, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1]
10  }
```

Los usuarios sólo pueden reportar como spam a un mensaje si se encuentra en sus bandejas de recibidos, enviados o basura (2), el mensaje reportado se mueve a la bandeja de *Spam* (3), los usuarios no pueden reportar como spam a un mensaje si se encuentran redactando otro mensaje al mismo tiempo (4), no se puede reportar como spam el mensaje de advertencia (5), el mensaje que se movió a la bandeja de spam, se elimina de las bandejas en las que se encuentre (6 – 8).

En al figura 3.16 se muestra el comportamiento de la operación *reportSpam*, en donde del tiempo *t2* al tiempo *t3 User0* recibe *Message* y de *t3* a *t4 User0* reporta como correo spam a *Message*.

3.3.13. Recuperar del spam

A veces los usuarios quieren aceptar algunos mensajes que llegaron a su bandeja de *Spam*, o por error reportaron como correo spam los mensajes de un contacto y desean revertir este proceso. Para permitir esta acción construimos el siguiente predicado:

```
1  pred notSpam(t, t′: Time, u: User, m: Message){
2      m in u.spam.t
3      u.write.t = none
4      sentMail.t′ = sentMail.t - u -> m
5      spam.t′ = spam.t - u -> m
```

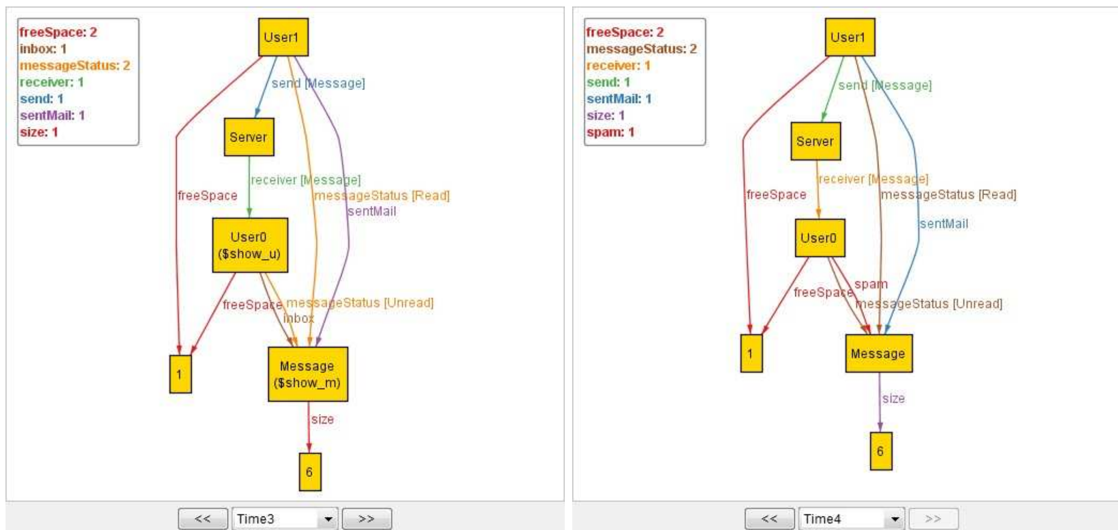

Figura 3.16: La operación *reportSpam*.

```
6       trash.t′ = trash.t - u -> m
7       inbox.t′ = inbox.t + u -> m
8       keepState[t, t′, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1]
9   }
```

El mensaje que se quiere recuperar debe estar en la bandeja de *Spam* (2), los usuarios no pueden reportar como spam a un mensaje si al mismo tiempo se encuentran redactando otro (3), a continuación hacemos lo que todos los sistemas de correo electrónico hacen, mover el mensaje del spam y de todas las bandejas en las que se encuentre hacia la bandeja de *Inbox* (4 – 7).

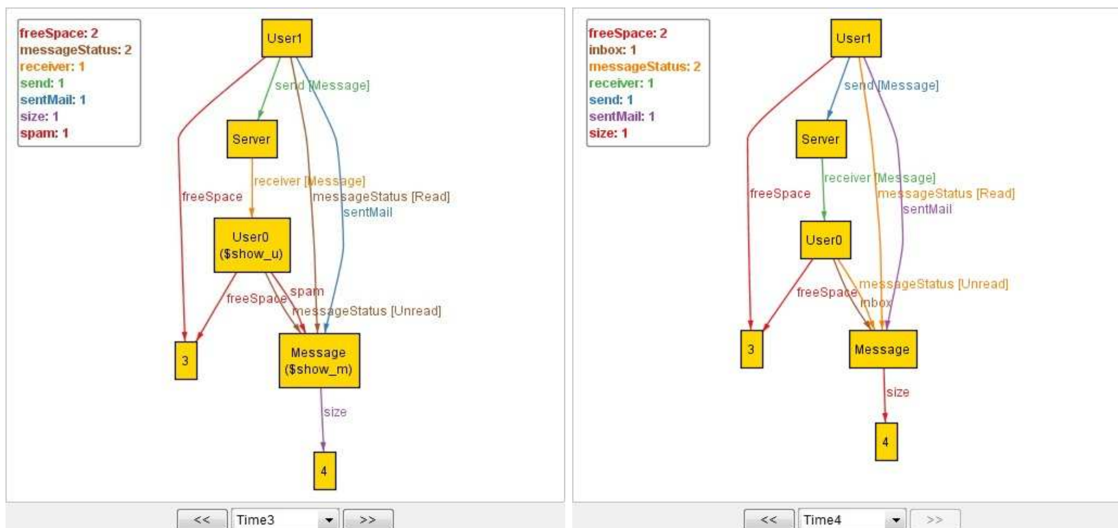

Figura 3.17: La operación *notSpam*.

En la figura 3.17 se muestra el comportamiento de la operacion *notSpam*, en donde del tiempo *t2* al tiempo *t3 User0* recibe *Message* a su bandeja de *Spam*, y de *t3* a *t4 User0* recupera del spam a *Message*.

3.4. Análisis

El análisis nos ayuda a verificar el funcionamiento correcto de nuestra especificación y a comprobar que lo que escribimos es exactamente lo que pensabamos, por ejemplo, realizando una búsqueda exhaustiva para un conjunto de afirmaciones que deseemos verificar dado un cierto *alcance*[10] podemos encontrar pequeños defectos que no habían sido descubiertos o situaciones que no habíamos considerado (como ya fue ejemplificado en la sección 2.3.2).

En este capítulo construiremos una serie de afirmaciones para verificar el comportamiento de nuestro modelo bajo ciertas circunstancias, asegurando de esta manera que nuestra especificación es fiable.

En la sección 3.3.1 de este capítulo se mostró que con la operación *composeMail* nuestro modelo no funcionaba del todo bien pues cuando se redactaba un mensaje, el tamaño de este tomaba un valor negativo e incluso un mismo usuario podía redactar más de un mensaje al mismo tiempo, después agregamos más restricciones al predicado *composeMail* para evitar que estas situaciones y otras no deseadas sucedieran de nuevo. Para verificar que estos errores jamás volverán a ocurrir (al menos para un número finito de elementos), creamos las afirmaciones: *validateSizeMessage*, *validateComposeMail*, *validateComposeMail2* y *validateComposeMail3*. Estas afirmaciones son válidas en el sentido de que el analizador Alloy no encontró contraejemplos.

Primero verificaremos que nuestro modelo funcione adecuadamente, es decir, asegurar que no se tomen en cuenta eventos inadmisibles:

```
assert noComposeSavedMail{
    all u: User, t: Time | no m: Message |
        m in User.drafts.t and composeMail[t, t.next, u, m]
}check noComposeSavedMail for 10
```

Los usuarios no pueden redactar un mensaje que ya ha sido guardado por algún otro usuario (incluso el mismo autor del mensaje) en algún momento. Con la expresión *User.drafts.t* obtenemos todos los mensajes guardados en el tiempo *t* de todos los usuarios. Por lo que la afirmación *noComposeSavedMail* nos dice si para todos los usuarios y en todos los tiempos no existe un mensaje que haya sido guardado y que se pueda radactar.

```
assert noSaveMail{
    all u: User, t: Time | no m: Message |
        some u.write.t and m not in u.write.t and saveMail[t, t.next, u, m]
}check noSaveMail for 10
```

Para que los usuarios puedan guardar los mensajes primero deben redactarlos, pero debemos asegurar que el mensaje que se redactó sea el que se va a guardar y no se tomen en cuenta otros que ni

[10] El *alcance* se refiere al número de elementos de cada signatura del modelo que fueron declarados en el comando *check*. Este número de elementos son los que serán tomados en cuenta por el analizador Alloy para buscar *ejemplos* o *contraejemplos* para una cierta firmación.

siquiera hayan sido redactados. La afirmación *noSaveMail* nos dice que si un usuario se encuentra redactando un mensaje, no existe otro mensaje diferente, tal que sea este el que el usuario vaya a guardar.

```
assert validateDiscardMail{
    all u: User, t: Time | no m: Message |
        m in (User - u).write.t and discardMail[t, t.next, u, m]
}check validateDiscardMail for 10
```

Los usuarios sólo pueden descartar los mensajes que ellos mismos redactaron. La expresión *(User - u).write.t* denota a todos los mensajes redactados por todos los usuarios excepto por el usuario *u* en el tiempo *t* por lo que la afirmación *validateDiscardMail* nos dice que para todos los usuarios en todos los tiempo no existe un mensaje que lo pueda descartar un usuario que no lo redactó.

```
assert validateSendMail{
    all u: User, t: Time | no m: Message |
        m in User.send.t.Server and sendMail[t, t.next, u, m]
}check validateSendMail for 10
```

Los usuarios no pueden enviar un mensaje que ya enviaron, en su lugar podrán reenviar ese mensaje mediante la operación *forward*. La expresión *User.send.t.Server* nos devuelve el conjunto de todos los mensajes enviados por todos los usuarios en el tiempo *t*. La afirmación *validateSendMail* nos dice que para todos los usuarios en todos los tiempo no existe un mensaje que haya sido enviado y que se pueda enviar de nuevo.

```
assert validateReadMail{
    all u: User, t: Time | no m: Message |
        u.messageStatus.t.m = Read and readMail[t, t.next, u, m]
}check validateReadMail for 10
```

Como se mencionó anteriormente, no vamos a permitir que los usuarios lean mensajes ya leídos. La expresión *u.messageStatus.t.m* nos dice el estado del mensaje *m* con respecto al usuario *u* en el tiempo *t*, por lo que si es igual a *Read* entonces los usuarios no pueden leer el mensaje.

```
1  assert validateOperations{
2      all u: User, m: Message, t: Time |
3          some u.write.t implies{
4              (
5                  not composeMail[t, t.next, u, Message] and
6                  not readMail[t, t.next, u, Message] and
7                  not forwardMail[t, t.next, u, Message] and
8                  not deleteMail[t, t.next, u, Message] and
9                  not deleteMailForever[t, t.next, u, Message] and
10                 not reportSpam[t, t.next, u, Message] and
11                 not notSpam[t, t.next, u, Message]
12             )
```

```
13
14              saveMail[t, t.next, u, m] or discardMail[t, t.next, u, m] or
15                  sendMail[t, t.next, u, m] implies
16                      u.write.t = m
17
18              receiveMail[t, t.next, u, Message] or
19                  receiveMessageWarning[t, t.next, u, Message] or
20                      receiveForwardMail[t, t.next, u, Message] implies
21                          u.write.t != m
22          )
23      }
24  }check validateOperations for 10
```

Cuando los usuarios se encuentran redactando un mensaje (3), sólo se les permite aplicar algunas operaciones. En primer lugar; esta prohibido que puedan redactar, leer, reenviar, borrar[11], eliminar de manera definitiva, reportar como spam y recuperar del spam algún mensaje (5 - 11). En segundo lugar; si se guarda, descarta o envía un mensaje *m* (14 y 15), quiere decir que el mensaje que se estaba redactando era *m* (16). Y en tercer lugar; si el usuario recibe un mensaje[12] *m* (18 - 20), entonces el mensaje que se estaba redactando no era *m* (21).

```
assert validateOperations2{
    all u: User, m: Message, t: Time |
        m not in
            Server.receiver.t.u + Server.receiverForward.t.u + u.send.t.Server
        implies
            not readMail[t, t.next, u, m] and not forwardMail[t, t.next, u, m] and
            not deleteMail[t, t.next, u, m] and not reportSpam[t, t.next, u, m] and
            not notSpam[t, t.next, u, m]
}check validateOperations2 for 10
```

Los usuarios no deben realizar ciertas operaciones con los mensajes que aún no tienen[13], como por ejemplo leer un mensaje, reenviar un mensaje, etc. Para saber si los usuarios tienen mensajes, nos basta con decir si han enviado o recibido mensajes en algún momento. Con las expresiones *Server.receiver.t.u* y *Server.receiverForward.t.u* obtenemos el conjunto de todos los mensajes recibidos por el usuario *u* en el tiempo *t* cuando estos fueron enviados y/o reenviados respectivamente, con *u.send.t.Server* el conjunto de todos los mensajes enviados por el usuario *u* en el tiempo *t*. La afirmación anterior nos dice que si los usuarios no tienen mensajes, entonces no pueden leer, reenviar, borrar, reportar como spam y recuperar del spam ningún mensaje.

```
assert validateDeleteMailForever{
    all u: User, m: Message, t: Time |
        m not in
```

[11] Mover un mensaje a la basura.
[12] Este mensaje que se recibe puede ser un mensaje enviado, reenviado o el mensaje de advertencia.
[13] Cuando un usuario tiene un mensaje nos referimos a que ese mensaje se encuentra en alguna de sus bandejas.

```
u.drafts.t + Server.receiver.t.u + Server.receiverForward.t.u + u.send.t.Server
implies
not deleteMailForever[t, t.next, u, m]
}check validateDeleteMailForever for 10
```

A diferencia de las operaciones utilizadas en la afirmación *validateOperations2*, cuando se desea eliminar un mensaje de manera definitiva también se toman en cuenta los mensajes guardados en la bandeja de *Drafts*. La expresión *u.drafts.t* denota a todos los mensajes guardados por el usuario *u* en el tiempo *t*. Por lo tanto, con *validateDeleteMailForever* afirmamos que los usuarios no pueden eliminar de manera definitiva un mensaje que no tienen o que no han guardado.

A continuación, verificaremos que nuestro correo electrónico tiene todas las caracteristicas convencionales que cualquier correo electrónico tiene:

```
assert freeSpaceOverLimit{
    no u: User, t: Time | u.freeSpace.t < 0
}check freeSpaceOverLimit for 10
```

La expresión *u.freeSpace.t* nos dice el espacio libre de la cuenta del usuario *u* en el tiempo *t*. Lo que estamos diciendo es que no existe un usuario y no existe un tiempo en el que el espacio libre de la cuenta de ese usuario sea menor a cero, es decir, aseguramos que los usuarios no pueden sobrepasar el límite de espacio de su cuenta.

```
assert validateForwardMail{
    all u: User, t: Time | no m: Message |
        u.messageStatus.t.m = Unread and forwardMail[t, t.next, u, m]
}check validateForwardMail for 10
```

Los sistemas de correo electrónico le permiten a los usuarios reenviar un mensaje en la misma ventana en donde pueden leerlo, obligandolos de cierta manera a leer un mensaje antes de reenviarlo, nosotros mostraremos que en nuestro sistema también existe esa característica. Con la expresión *u.messageStatus.t.m* obtenemos el estado del mensaje *m* con respecto al usuario *u* en el tiempo *t*, después afirmamos que para todos los usuarios y en cualquier momento no existe un mensaje tal que no han leido y quieran reenviarlo.

```
assert noReceiveMultipleUsers{
    no m: Message, t: Time | #m.(Server.receiver.t) > 1
}check noReceiveMultipleUsers for 10
```

Nuestro modelo le permitirá a los usuarios enviar un mensaje con múltiples destinatarios. Para poder visualizarlo primero afirmaremos lo contrario, después el analizador Alloy deberá refutar nuestra afirmación y con esto sabremos que nuestro sistema cuenta con esta propiedad. Con la expresión *m.(Server.receiver.t)* obtenemos todos los destinatarios del mensaje *m* en el tiempo *t*, ahora sólo decimos que no existe *m* y *t*, tal que el número de destinatarios de *m* en el tiempo *t* sea mayor a uno. La afirmación es falsa[14], por lo tanto, nuestro sistema de correo electrónico permite enviar

[14] Una afirmación es falsa cuando se ejecuta en el analizador Alloy y este encuentra al menos un contraejemplo.

mensajes a múltiples usuarios.

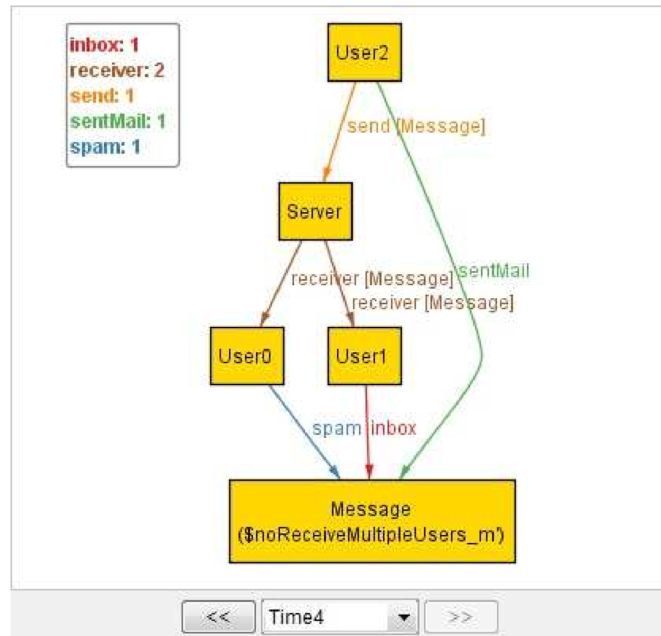

Figura 3.18: Contraejemplo de la afirmación *noReceiveMultipleUsers*.

La figura 3.18 nos muestra como *User0* y *User1* reciben el mismo mensaje que envió *User2* en algún momento.

```
assert noReceiveMultipleUsers2{
    no m: Message, t: Time | #m.(Server.receiverForward.t) > 1
}check noReceiveMultipleUsers2 for 10
```

Debemos mostrar que lo mismo ocurre para los mensajes reenviados usando la misma idea de la afirmación *noReceiveMultipleUsers* y buscando los mensajes reenviados que serán recibidos por algún usuario en la relación *receiverForward*. Con la expresión *m.(Server.receiverForward.t)* obtenemos todos los destinatarios del mensaje *m* en el tiempo *t*, ahora sólo afirmamos que no existe *m* y *t* en donde haya más de un destinatario. Al ejecutar esta afirmación, el analizador nos muestra varios contraejemplos, por lo que la afirmación es falsa.

Al igual que en la afirmación anterior, la figura 3.19 nos muestra como *User0* y *User1* reciben el mismo mensaje que reenvió *User2* en algún momento.

```
assert validateDeleteMail{
    all u: User, t: Time | no m: Message |
        m in (u.trash.t + u.spam.t) and
            m not in (u.inbox.t + u.sentMail.t) and deleteMail[t, t.next, u, m]
}check validateDeleteMail for 10
```

Los usuarios no pueden mover a la basura los mensajes que se encuentran únicamente en la basura o en el spam, decimos únicamente por que en nuestro sistema existe la posibilidad de que en un

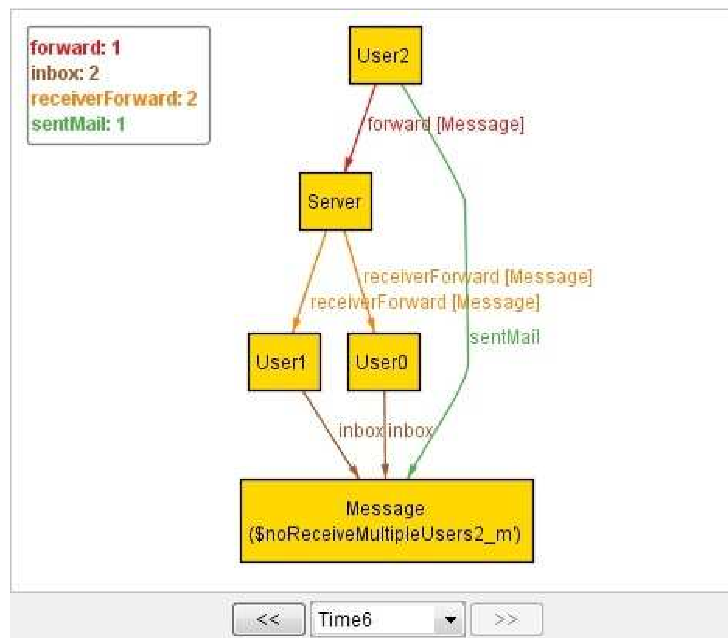

Figura 3.19: Contraejemplo de la afirmación *noReceiveMultipleUsers2*.

mismo tiempo se encuentre en alguna otra bandeja. Por ejemplo, si un usario decide mover al spam un mensaje recibido y después decide reenviar ese mismo mensaje, al siguiente momento del tiempo ese mensaje estaría tanto en el spam como en la bandeja de enviados.

```
1 assert validateReportSpam{
2     all u: User, t: Time | no m: Message{
3         m in u.drafts.t and reportSpam[t, t.next, u, m]
4         m in u.spam.t and
5             m not in (u.inbox.t + u.sentMail.t + u.trash.t) and reportMail[t, t.next, u, m]
6     }
7 }check validateReportSpam for 10
```

Los usuarios pueden mover al spam cualquier mensaje excepto los que están guardados en la bandeja de *Drafts* y los que se encuentran en el spam. Los mensajes en el spam deben estar únicamente en la bandeja de *Spam* para poder aplicar la operación. Primero comprobamos que no exista un mensaje que este guardado en al bandeja de *Drafts* y pueda ser reportado como spam (3). Después decimos que no existe un mensaje que únicamente se encuentre en el spam y pueda ser reportado como spam (4 - 5).

```
1 assert validateNotSpam{
2     all u: User, t: Time | no m: Message{
3         m in u.drafts.t and notSpam[t, t.next, u, m]
4         m not in u.spam.t and notSpam[t, t.next, u, m]
5     }
6 }check validateNotSpam for 10
```

No se puede recuperar del spam un mensaje que no sea spam, lo mismo ocurre con los mensajes en *Drafts*, los cuales son mensajes guardados que aún no han sido enviados. Por eso, primero ve-

rificamos que los usuarios no puedan recuperar los mensajes guardados en *Drafts* (3) y después aseguramos que no puedan recobrar un mensaje que no sea spam (4).

Comenzamos este capítulo definiendo las diferentes entidades y sus respectivos atributos, despúes construímos un conjunto de operaciones que a través de la traza definen el comportamiento de nuestro modelo. Pero no es que desde un principio tuviéramos contempladas todas las reglas y restricciones que se deben cumplir, sino que en la medida en la que agregábamos más aserciones nuestra especificación fue creciendo poco a poco, es por esto que queremos enfatizar la importancia del análisis y la verificación que nos permite realizar el analizador Alloy.

4: Conclusiones

A partir de este trabajo, hemos mostrado que el analizador Alloy nos permite simular las funciones básicas de un sistema de correo electrónico. Tenemos que recalcar que las aserciones nos permiten garantizar la confiabilidad y seguridad de las funciones implementadas en nuestro modelo, ya que debido al uso de estas aserciones durante todo el desarrollo pudimos encontrar ciertos casos que hacían inestable el sistema, casos que nos obligaron a crear más restricciones para evitar situaciones que no ocurrirían en un sistema real.

Hemos decidido dejar fuera de nuestra especificación algunas operaciones que los sistemas modernos poseen, como: crear etiquetas[1], mover mensajes de una etiqueta a otra, etiquetar mensajes, o hasta algo tan importante como adjuntar archivos en un mensaje que será enviado, por que no era el objetivo principal de este trabajo. Sin embargo, es muy sencillo implementar estas operaciones, por ejemplo, si quisiéramos aplicar alguna operación que involucre a las etiquetas nuestra *traza* necesitaría gestionar un parámetro más, el cual sea una entidad que se encargue de representar a las etiquetas, tomando un conjunto de elementos de esta entidad para que junto con los usuarios y los mensajes pueda llevarse a cabo una determinada acción de un tiempo a otro. Lo mismo ocurre si queremos adjuntar archivos a los mensajes, es necesario representar a estos archivos con una entidad para que la *traza* pueda en algún momento relacionarlos con un mensaje.

Lo que queremos decir con esto es que nuestro modelo sirve como base para crear una especificación que contenga más funcionalidades, en otras palabras, lo que ahora sigue es construir un modelo más grande, por ejemplo, un sistema de correo electrónico que cuente con varios servidores en donde se pueda especificar la manera en la que estos se distribuirán para ofrecer el servicio correspondiente a los usuarios, que disponga también de un servicio de mensajería instantánea, una red social e incluso que sea posible modelar las diferentes políticas y estrategias que los sistemas reales tienen para clasificar el correo denominado como *spam*.

Debemos hacer uso de herramientas formales como el analizador Alloy cuando necesitemos asegurar que la abstracción que realizamos sobre la solución de un problema sea correcta, es decir, que nos ofrezca fiabilidad y seguridad. Como ya mencionamos anteriormente, este grado de seguridad se obtiene al realizar un análisis exhaustivo de ciertas propiedades o caracteristicas en particular.

[1] Las etiquetas en Gmail cumplen la misma función que un directorio en otros sistemas de correo electrónico.

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 step[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 del caso: *Sistema de correo electrónico*

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

sig User{
    write: Message -> Time,
    send, forward: Message -> Server -> Time,
    inbox, sentMail, drafts, spam, trash: Message -> Time,
    messageStatus: Status -> Message -> Time,
    freeSpace: Int one -> Time
}

sig Message{
    size: Int lone -> Time,
}

one sig MessageWarning extends Message{} { no size }

one sig Server{
    receiver, receiverForward, warning: Message -> User -> Time
}

abstract sig Status{}

one sig Read, Unread extends Status{}

sig Time{}

fact Rules{
    all u: User, t: Time | not accountOverLimit[t, u]

    all m: Message, t: Time |
        let from = send.t.Server.m, to = m.(Server.receiver.t) |
            some from implies from not in to

    all m: Message, t: Time |
        let from = forward.t.Server.m, to = m.(Server.receiverForward.t) |
            some from implies from not in to
}
```

```
fact Traces{
    init[T/first[]]
    all t: Time - T/last[] |
        let t' = T/next[t] |
            some u: User, m: Message |
                composeMail[t, t', u, m] or saveMail[t, t', u, m] or
                discardMail[t, t', u, m] or sendMail[t, t', u, m] or
                receiveMail[t, t', u, m] or receiveMessageWarning[t, t', u, m] or
                readMail[t, t', u, m] or forwardMail[t, t', u, m] or
                receiveForwardMail[t, t', u, m] or deleteMail[t, t', u, m] or
                deleteMailForever[t, t', u, m] or reportSpam[t, t', u, m] or
                notSpam[t, t', u, m]
}

pred init(t: Time){
    all u: User{
        no u.write.t
        no u.send.t
        no u.forward.t
        no u.inbox.t
        no u.sentMail.t
        no u.drafts.t
        no u.spam.t
        no u.trash.t
        no u.messageStatus.t
        u.freeSpace.t = 7
    }
    all m: Message | no m.size.t
    no Server.receiver.t and no Server.receiverForward.t and no Server.warning.t
}

pred accountLimit(t: Time, u: User){
    u.freeSpace.t = 0
}

pred accountOverLimit(t: Time, u: User){
    u.freeSpace.t < 0
}
```

```
pred keepState(t, t′: Time, a, b, c, d, e, f, g, h, i, j, k, l, m, n: Int){
    a = 1 implies write.t′ = write.t
    b = 1 implies send.t′ = send.t
    c = 1 implies forward.t′ = forward.t
    d = 1 implies inbox.t′ = inbox.t
    e = 1 implies sentMail.t′ = sentMail.t
    f = 1 implies drafts.t′ = drafts.t
    g = 1 implies spam.t′ = spam.t
    h = 1 implies trash.t′ = trash.t
    i = 1 implies messageStatus.t′ = messageStatus.t
    j = 1 implies freeSpace.t′ = freeSpace.t
    k = 1 implies size.t′ = size.t
    l = 1 implies receiver.t′ = receiver.t
    m = 1 implies receiverForward.t′ = receiverForward.t
    n = 1 implies warning.t′ = warning.t
}

pred accountManagerSpace(t, t′: Time, u: User, m: Message, tipo: Int){
    tipo = 1 implies{
        let currentSpace = u.freeSpace.t, newSpace = currentSpace.sub[m.size.t] |
            freeSpace.t′ = freeSpace.t - u -> currentSpace + u -> newSpace
    }else{
        let currentSpace = u.freeSpace.t, newSpace = currentSpace.add[m.size.t] |
            freeSpace.t′ = freeSpace.t - u -> currentSpace + u -> newSpace
    }
}

pred composeMail(t, t′: Time, u: User, m: Message){
    m not in User.write.t + User.drafts.t
    u.write.t = none
    m not in User.send.t.Server + Server.receiver.t.User
    write.t′ = write.t + u -> m
    one weight: Int{
        weight >= 1
        size.t′ = size.t + m -> weight
    }
    keepState[t, t′, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1]
}
```

```
pred saveMail(t, t′: Time, u: User, m: Message){
    u.write.t = m
    write.t′ = write.t - u -> m
    drafts.t′ = drafts.t + u -> m
    accountManagerSpace[t, t′, u, m, 1]
    keepState[t, t′, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1]
}

pred discardMail(t, t′: Time, u: User, m: Message){
    u.write.t = m
    write.t′ = write.t - u -> m
    size.t′ = size.t - m -> (m.size.t)
    keepState[t, t′, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1]
}

pred sendMail(t, t′: Time, u: User, m: Message){
    m in u.write.t + u.drafts.t
    m not in User.send.t.Server + Server.receiver.t.User
    send.t′ = send.t + u -> m -> Server
    sentMail.t′ = sentMail.t + u -> m
    messageStatus.t′ = messageStatus.t + u -> Read -> m
    m in u.write.t implies{
        write.t′ = write.t - u -> m
        accountManagerSpace[t, t′, u, m, 1]
        keepState[t, t′, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1]
    }else{
        u.write.t = none
        drafts.t′ = drafts.t - u -> m
        keepState[t, t′, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1]
    }
}

pred receiveMail(t, t′: Time, u: User, m: Message){
    m not in Server.receiver.t.u
    m in User.send.t.Server
    receiver.t′ = receiver.t + Server -> m -> u
    messageStatus.t′ = messageStatus.t - u -> Status -> m + u -> Unread -> m
    accountManagerSpace[t, t′, u, m, 1]
    {inbox.t′ = inbox.t + u -> m
        keepState[t, t′, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1]}
    or {spam.t′ = spam.t + u -> m
        keepState[t, t′, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1]}
}
```

```
pred receiveMessageWarning(t, t′: Time, u: User, m: Message){
    accountLimit[t, u]
    m = MessageWarning
    let notice = Server -> m -> u{
        notice not in warning.t
        warning.t′ = warning.t + notice
    }
    inbox.t′ = inbox.t + u -> m
    keepState[t, t′, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0]
}

pred readMail(t, t′: Time, u: User, m: Message){
    u.write.t = none
    let status = u -> Unread -> m, newStatus = u -> Read -> m{
        status in messageStatus.t
        messageStatus.t′ = messageStatus.t - status + newStatus
    }
    keepState[t, t′, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1]
}

pred forwardMail(t, t′: Time, u: User, m: Message){
    u.messageStatus.t.m = Read
    u.write.t = none
    forward.t′ = forward.t + u -> m -> Server
    sentMail.t′ = sentMail.t + u -> m
    keepState[t, t′, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1]
}

pred receiveForwardMail(t, t′: Time, u: User, m: Message){
    m in User.forward.t.Server
    m not in Server.receiverForward.t.u
    receiverForward.t′ = receiverForward.t + Server -> m -> u
    messageStatus.t′ = messageStatus.t - u -> Status -> m + u -> Unread -> m
    accountManagerSpace[t, t′, u, m, 1]
    {inbox.t′ = inbox.t + u -> m
        keepState[t, t′, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1]}
    or
    {spam.t′ = spam.t + u -> m
        keepState[t, t′, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1]}
}
```

```
pred deleteMail(t, t′: Time, u: User, m: Message){
    m in u.inbox.t + u.sentMail.t
    m != MessageWarning
    u.write.t = none
    inbox.t′ = inbox.t - u -> m
    sentMail.t′ = sentMail.t - u -> m
    spam.t′ = spam.t - u -> m
    trash.t′ = trash.t + u -> m
    keepState[t, t′, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1]
}

pred deleteMailForever(t, t′: Time, u: User, m: Message){
    m in (u.inbox.t + u.sentMail.t + u.drafts.t + u.spam.t + u.trash.t)
    u.write.t = none
    m not in MessageWarning
    accountManagerSpace[t, t′, u, m, 2]
    m in u.drafts.t implies{
        inbox.t′ = inbox.t - u -> MessageWarning
        drafts.t′ = drafts.t - u -> m
        size.t′ = size.t - m -> m.size.t
        warning.t′ = warning.t - Server -> MessageWarning -> u
        keepState[t, t′, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0]
    }else{
        inbox.t′ = inbox.t - (u -> m) - (u -> MessageWarning)
        sentMail.t′ = sentMail.t - u -> m
        spam.t′ = spam.t - u -> m
        trash.t′ = trash.t - u -> m
        messageStatus.t′ = messageStatus.t - u -> Status -> m
        warning.t′ = warning.t - Server -> MessageWarning -> u
        keepState[t, t′, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0]
    }
}

pred reportSpam(t, t′: Time, u: User, m: Message){
    m in u.inbox.t + u.sentMail.t + u.trash.t
    spam.t′ = spam.t + u -> m
    u.write.t = none
    m != MessageWarning
    inbox.t′ = inbox.t - u -> m
    sentMail.t′ = sentMail.t - u -> m
    trash.t′ = trash.t - u -> m
    keepState[t, t′, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1]
}
```

```
pred notSpam(t, t': Time, u: User, m: Message){
    m in u.spam.t
    u.write.t = none
    sentMail.t' = sentMail.t - u -> m
    spam.t' = spam.t - u -> m
    trash.t' = trash.t - u -> m
    inbox.t' = inbox.t + u -> m
    keepState[t, t', 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1]
}

pred show(){}
run show for 3 but 20 Time

assert validateSizeMessage{
    all m: Message, t: Time |
        let weight = m.size.t |
            some weight implies weight >= 1
}check validateSizeMessage for 10

assert validateComposeMail{
    all u: User | no m: Message, t: Time | some u.write.t and composeMail[t, t.next, u, m]
}check validateComposeMail for 10

assert validateComposeMail2{
    all m: Message, t: Time |
        let author = write.t.m |
            some author implies #author = 1
}check validateComposeMail2 for 10

assert validateComposeMail3{
    no u: User, m: Message | one t: Time |
        let author = write.t.m |
            some author and
                (composeMail[t, t.next, author, m] or composeMail[t, t.next, u, m])
}check validateComposeMail3 for 10

assert noComposeSavedMail{
    all u: User, t: Time | no m: Message |
        m in User.drafts.t and composeMail[t, t.next, u, m]
}check noComposeSavedMail for 10
```

```
assert noSaveMail{
    all u: User, t: Time | no m: Message |
        some u.write.t and m not in u.write.t and saveMail[t, t.next, u, m]
}check noSaveMail for 10

assert validateDiscardMail{
    all u: User, t: Time | no m: Message |
        m in (User - u).write.t and discardMail[t, t.next, u, m]
}check validateDiscardMail for 10

assert validateSendMail{
    all u: User, t: Time | no m: Message |
        m in User.send.t.Server and sendMail[t, t.next, u, m]
}check validateSendMail for 10

assert validateReadMail{
    all u: User, t: Time | no m: Message |
        u.messageStatus.t.m = Read and readMail[t, t.next, u, m]
}check validateReadMail for 10

assert validateOperations{
    all u: User, m: Message, t: Time |
        some u.write.t implies{
            (not composeMail[t, t.next, u, Message] and
                not readMail[t, t.next, u, Message] and
                not forwardMail[t, t.next, u, Message] and
                not deleteMail[t, t.next, u, Message] and
                not deleteMailForever[t, t.next, u, Message] and
                not reportSpam[t, t.next, u, Message] and
                not notSpam[t, t.next, u, Message])

                saveMail[t, t.next, u, m] or discardMail[t, t.next, u, m] or
                    sendMail[t, t.next, u, m] implies
                        u.write.t = m

                receiveMail[t, t.next, u, Message] or
                    receiveMessageWarning[t, t.next, u, Message] or
                        receiveForwardMail[t, t.next, u, Message] implies
                            u.write.t != m
            )
        }
}check validateOperations for 10
```

```
assert validateOperations2{
    all u: User, m: Message, t: Time |
        m not in
            Server.receiver.t.u + Server.receiverForward.t.u + u.send.t.Server
        implies
            not readMail[t, t.next, u, m] and not forwardMail[t, t.next, u, m] and
            not deleteMail[t, t.next, u, m] and not reportSpam[t, t.next, u, m] and
            not notSpam[t, t.next, u, m]
}check validateOperations2 for 10

assert validateDeleteMailForever{
    all u: User, m: Message, t: Time |
        m not in
            u.drafts.t + Server.receiver.t.u + Server.receiverForward.t.u + u.send.t.Server
        implies
            not deleteMailForever[t, t.next, u, m]
}check validateDeleteMailForever for 10

assert freeSpaceOverLimit{
    no u: User, t: Time | u.freeSpace.t < 0
}check freeSpaceOverLimit for 10

assert validateForwardMail{
    all u: User, t: Time | no m: Message |
        u.messageStatus.t.m = Unread and forwardMail[t, t.next, u, m]
}check validateForwardMail for 10

assert noReceiveMultipleUsers{
    no m: Message, t: Time | #m.(Server.receiver.t) > 1
}check noReceiveMultipleUsers for 10

assert noReceiveMultipleUsers2{
    no m: Message, t: Time | #m.(Server.receiverForward.t) > 1
}check noReceiveMultipleUsers2 for 10

assert validateDeleteMail{
    all u: User, t: Time | no m: Message |
        m in (u.trash.t + u.spam.t) and
            m not in (u.inbox.t + u.sentMail.t) and deleteMail[t, t.next, u, m]
}check validateDeleteMail for 10
```

```
assert validateReportSpam{
    all u: User, t: Time | no m: Message{
        m in u.drafts.t and reportSpam[t, t.next, u, m]
        m in u.spam.t and
            m not in (u.inbox.t + u.sentMail.t + u.trash.t) and reportMail[t, t.next, u, m]
    }
}check validateReportSpam for 10

assert validateNotSpam{
    all u: User, t: Time | no m: Message{
        m in u.drafts.t and notSpam[t, t.next, u, m]
        m not in u.spam.t and notSpam[t, t.next, u, m]
    }
}check validateNotSpam for 10
```

Bibliografía

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

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

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

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

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

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

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

[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.

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

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