Sistemas Informáticos

Curso 2007-2008

HOTL: Hypothesis and Observations Testing Logic

Autores:

Jesús Caño Núñez
Ignacio Ramos Ruiz
Enrique Álvarez Fernández

Dirigido por:

Don Ismael Rodríguez Laguna

Resumen

El proyecto realizado consiste en la implementación de una herramienta capaz de simular el comportamiento descrito en una lógica desarrollada (HOTL: Hypotheses and observations testing logic), en un excelente trabajo de investigación, por parte de nuestro profesor director de proyecto D. Ismael Rodríguez Laguna y sus dos compañeros D. Manuel Núñez y Dña. Mercedes G. Merayo.

La implementación ha sido integrada en una interfaz gráfica que nos permitirá ejecutar una seria de reglas definidas en la lógica dado un modelo, y dadas una especificación concreta, unas observaciones y unas hipótesis y la aplicación de las reglas sobre estos elementos, nos diga si la implementación es conforme o no respecto de la especificación. Todo esto mostrándose de manera gráfica para que el usuario sea consciente de lo ocurrido en cada momento, como por ejemplo; la carga de la especificación, de las observaciones e hipótesis, los modelos generados a partir de las observaciones, la generación de modelos que implica la ejecución de las reglas una a una, etc.

La herramienta ha sido implementada sobre una plataforma Java y su diseño se ha realizado llevando a cabo técnicas y métodos empleados en ingeniería del software, tales como un arquitectura clara de la aplicación (que facilitará la ampliación futura de la herramienta), uso de patrones de diseño, división del trabajo en iteraciones, etc., además del uso de conocimientos de metodología y tecnología de la programación para el desarrollo de ciertos algoritmos necesarios para la implementación de la lógica.

Summary

The made project consists of the implementation of a tool able to simulate the behavior described in a developed logic (HOTL: Hypotheses and observations testing logic), in an excellent work of investigation, on the part of our professor director of project D. Ismael Rodriguez Laguna and their two companions D. Manuel Núñez and Dña. Mercedes G. Merayo.

The implementation has been integrated in a graphical interface that will allow us to execute serious of rules defined in the given logic a model, and given a concrete specification, observations and a hypothesis and the application of the rules on these elements, says to us if the implementation is in agreement or nonrespect to the specification. All this being of graphical way so that the user is conscious of the happened thing at every moment, like for example; the load of the specification, the observations and hypothesis, the models generated from the observations, the generation of models that the execution of rules one to one implies, etc.

The tool has been implemented on a Java platform and its technical design has been made carrying out and used methods in engineering of the software, such as a clear architecture of the application (that it will facilitate the future extension of the tool), use of design patterns, division of the work in iterations, etc., in addition to the use of knowledge of methodology and technology of the programming for the development of certain necessary algorithms for the implementation of the logic.

Palabras claves

- Hotl
- Fsm
- Modelo
- Observación
- Hipótesis
- Regla
- Testing
- Lógica
- Conformidad
- CountElim

Autorización

Autorizamos a la facultad de Informática de la Universidad Complutense de Madrid, así como al resto de los centros adscritos a la Universidad Complutense de Madrid a la utilización y modificación de todos los componentes que forman este proyecto. Siempre y cuando no se derive en una utilización comercial de cualquiera de los componentes o de parte de ellos.

Firmado: ……………… ………………… ………………

Enrique Álvarez *Jesús Caño* *Ignacio Ramos*

Índice

1. Introducción 7
 1.1. Objetivos generales 7
 1.2. Breve introducción a HOTL 10
  1.2.1. Introducción a la lógica HOTL 10
  1.2.2. Explicación del funcionamiento de HOTL 11
  1.2.3. Conceptos y definiciones de HOTL 12
  1.2.4. Observaciones 14
  1.2.5. Hipótesis 15
  1.2.6. Reglas 18
  1.2.7. Algoritmo de ejecución (Máquina de estados) 26
 1.3. Resultados esperados 28
  1.3.1. Limitaciones teóricas 28
  1.3.2. Limitaciones prácticas 29
 1.4. Metodología de trabajo 30
  1.4.1. Introducción 30
  1.4.2. Análisis de riesgo 30
  1.4.3. Planificación temporal 31
  1.4.4. Modelo de proceso 31
2. Desarrollo de la aplicación 33
 2.1. Tecnologías utilizadas 33
 2.2. Organización interna de la aplicación 34
  2.2.1. Arquitectura 34
  2.2.2. Patrones de diseño 35
  2.2.3. Reusabilidad y modularidad del código 37
 2.3. Algoritmos específicos utilizados 38
  2.3.1. ModelElim y CountElim 38
  2.3.2. AllCorrect y Conformance 41
  2.3.3. Subconjuntos de un vector 48
  2.3.4. Mapeado 49
 2.4. Persistencia de datos 50
3. Aplicación final 51
 3.1. Comprobación de objetivos y resultados obtenidos 51
 3.2. Manual de usuario 52
 3.3. Ejemplos de uso 64
 3.4. Posibles extensiones 74
  3.4.1. Extensiones de la lógica 74
  3.4.2. Extensiones de la aplicación 74
4. Conclusiones generales 76
5. Agradecimientos 78
6. Bibliografía 79

1. INTRODUCCIÓN

1.1. Objetivos generales

A día de hoy y en los tiempos que transcurren los sistemas, en general, han adquirido una gran complejidad. Llegando estos a estar construidos con múltiples componentes e incluso desarrollados por diferentes programadores.

Como resultado de esto, ninguno de los miembros del equipo de desarrollo conoce la implementación tan afondo como para saber, sin usar ninguna consideración adicional, que el sistema que estamos tratando es correcto en todos sus aspectos. Estos acontecimientos han precipitado o han hecho necesario la aplicación de técnicas o reglas para chequear la corrección del sistema.

Actualmente existen muchas y diferentes alternativas para hacer todo esto. Una de ellas es el "testeado lógico mediante hipótesis y observaciones", sobre la cual nuestro equipo de proyecto ha trabajado para intentar hacer de esta alternativa de chequeo de sistemas una herramienta potente e intuitiva, siempre dentro de los términos y las posibilidades que nos permite el testeado lógico de sistemas.

El propósito de nuestra herramienta es trabajar con un modelo abstracto o especificación que muestre el comportamiento deseable del sistema. Así, definimos la IUT (implementation under test) en términos de la comparación con la especificación. Podemos decir que la IUT es "conforme" a la especificación si muestra un comportamiento similar a la del modelo.

Para poder chequear la conformidad de la implementación con respecto a la especificación, debemos usar técnicas formales de testeado para la extracción de tests desde la especificación., representando cada test un comportamiento deseable que la IUT debe satisfacer o realizar.

Es obvio que cuanto más tiempo empleemos testeando un iut, más seguro estamos de su corrección. Sin embargo en la gran mayoría de proyectos de software las fechas de entrega son bastante restrictivas por lo que el tiempo y esfuerzo dedicado al testeado de dicho proyecto es limitado.

Realmente, ya que muchos sistemas exhiben posiblemente infinitos comportamientos, les llevaría un tiempo infinito el asegurarse la validación de todos y cada uno de los diferentes comportamientos. Para poder superar este problema, testadores han asumido algunas suposiciones sobre la IUT con respecto al conocimiento sobre esta construcción.

Por ejemplo, un testador puede asumir que la implementación puede ser representada por una máquina finita de estados, que tiene al menos n estados, etc.

En esta línea, un amplio rango de metodologías de testeado han sido propuestas, las cuáles, para un conjunto inicial de hipótesis específico garantiza que un test extraído de la especificación es correcto y completo para chequear la conformidad de la IUT con respecto a la especificación. Sin embargo, un marco de hipótesis establecido en desarrollo es muy estricto y limita la aplicabilidad de una metodología de testado específica. Por ejemplo, podría ser deseable que, en un contexto concreto, el testador asumiera que el comportamiento para cuatro estados concretos de la implementación son deterministas y que dos de ellos representaran una equivalencia en dicha implementación. Además, el testador podría hacer otras consideraciones más complejas tales como "estados no-deterministas de la implementación no puede mostrar salidas que la máquina no haya mostrado que ese estado haya sido probado 100 veces". En un escenario diferente el testador podría no creer esta consideración pero podría pensar que "si ella observa dos secuencias de longitud 200 y todas sus entradas y salidas coinciden entonces ellos recorren los mismos estados de IUT".

Notar que si un testador asume la validación de un conjunto de hipótesis para probar una IUT dada, entonces un test específico podría ser apropiado, pero usando otro tipo de hipótesis podría no llegar a serlo. Sería entonces deseable proporcionarle al testador de una herramienta que le permitiese analizar el impacto de consideración de un conjunto de hipótesis dado en proceso de testado, tal y como las consecuencias de adición/eliminación de hipótesis del conjunto. El objetivo de esta metodología podría ser el establecer si dado un conjunto finito de observaciones extraídas por un conjunto de pruebas es completo en el caso de sostener las hipótesis consideradas, estos es, asumimos si obteniendo estas observaciones desde la IUT implica que la IUT es conforme a la especificación si la hipótesis se sostienen. Llegados a este punto se presenta una lógica llamada HOTL (Hypotheses and Observations Testing logic) y que será el centro teórico para la implementación de nuestra herramienta de testeado lógico.

Una vez vistos los anteriores puntos, nuestro principal objetivo es diseñar una herramienta que implemente la lógica "HOTL". Nuestra implementación está pensada más hacia el diseño de una herramienta didáctica, que sirva para comprender visualmente y de una manera más sencilla el funcionamiento de dicha lógica para pequeños ejemplos, mostrando gráficamente los resultados obtenidos por la lógica, de manera que un testador se evite tener que realizarlo "a mano".

La herramienta se acerca más a un prototipo sobre el que se pudiera seguir trabajando y perfeccionando ésta para un mejor acercamiento al usuario, que de una herramienta totalmente terminada.

1.2. Breve introducción a HOTL

1.2.1. Introducción a la lógica de HOTL

El objetivo de HOTL (Hypotheses and Observations Testing logic) es asumir si dado un conjunto de observaciones implica la corrección de la IUT bajo la suposición de un conjunto de hipótesis.

Para permitir al testador la composición de un conjunto de hipótesis, la lógica proporciona un conjunto de hipótesis, incluyendo hipótesis que aparecen en metodologías conocidas de testado. El objetivo final de la lógica es facilitar al menos el seguimiento de tres tareas:

- Primero, un testador puede usarla para personalizar el proceso de testado para su entorno especificado. Por el uso de lógica, ella puede inferir no sola las consecuencias de añadir nuevos estados, sino que también las consecuencias de añadir nuevas hipótesis. De esta manera, el testador tiene el control sobre un amplio rango de variables de testado. En particular, la construcción de un conjunto de pruebas o test para extraer las observaciones y la definición de hipótesis pueden influir la una en la otra.

  Esto proporciona un escenario dinámico de pruebas donde, dependiendo de la especificación y del conocimiento del testador de la IUT, diferentes conjuntos de pruebas y de hipótesis pueden ser consideradas.

- Segundo, una lógica así permite al testador evaluar la “calidad” de un conjunto de pruebas para descubrir errores en una implementación: Si las observaciones que puedan ser extraídas por un conjunto de pruebas requiere (para su completitud) un conjunto de hipótesis que difícilmente son aceptadas entonces estas son requeridas por otro conjunto, entonces el último conjunto podría ser preferido.

  Esto es porque este conjunto podría permitir al testador alcanzar diagnósticos en un entorno menos restrictivo.

- Finalmente, denotar que una lógica debe proveer de un puente conceptual entre diferentes aproximaciones de testado. En particular, debemos usar esto para representar los conjuntos de hipótesis considerados por diferentes aproximaciones. Entonces, considerando un las observaciones de un conjunto de pruebas podemos obtener, un conjunto de pruebas que es completo en una aproximación (e.g., bajo algunas hipótesis) podría transformarse en un nuevo conjunto que es completo en otro (e.g., bajo la suposición de otro conjunto de hipótesis).

De la misma manera, podemos analizar como el tamaño del conjunto de pruebas se ve afectado por las hipótesis. Además, podemos usar la lógica para crear aproximaciones intermedias donde el conjunto de hipótesis está apropiadamente mezclado.

Denotar también que comparar dos metodologías puede requerir la extensión de la lógica: Si una hipótesis no puede ser expresada con el repertorio actual de hipótesis, entonces la lógica puede ser extendida para permitir dicha representación. Afortunadamente, la modularidad de lógica hará que esta sea una tarea que se pueda llevar a cabo con suma facilidad.

Así podremos ver, que es muy agradable que un nuevo predicado de hipótesis y una nueva regla de deducción, manteniendo esta hipótesis, podría ser añadida.

1.2.2. Funcionamiento de HOTL

El funcionamiento o metodología de la lógica se pueden englobar en dos fases que consisten en:

FASE 1

La primera fase consiste en la clásica aplicación de pruebas para una IUT. Usando cualquiera de los métodos citados en puntos anteriores, un conjunto de pruebas será derivado de una especificación. Si la aplicación de este conjunto de pruebas encuentra un resultado inesperado entonces el proceso de testado se parará y podremos decir que la IUT no es conforme a la especificación. Sin embargo, si un mal comportamiento no es detectado el testador no podrá llegar a saber si le IUT es correcta. En caso de producirse esto último se dará comienzo a la segunda fase de la que se compone nuestra metodología.

FASE 2

Una vez nos encontramos en esta fase el testador aplica la lógica descrita para deducir si pasando estas pruebas implica que la IUT es correcta si un conjunto de hipótesis es asumido. Si esto es así la lógica asumida es considerada como correcta; de otra manera, el testador tiene o debe estar interesado en la aplicación de más pruebas o incluso en la consideración de más hipótesis y en la aplicación de la lógica hasta que la corrección de la IUT esté total y efectivamente garantizada.

Para poder aplicar apropiadamente la lógica, el comportamiento de la IUT observado durante la aplicación de las diferentes pruebas debe ser apropiadamente representado.

Para la aplicación de cada una de las prueba para la IUT, construimos una observación, esto es, una secuencia de entradas y salidas denotando la prueba y la repuesta producida por la IUT, respectivamente. Ambos, las observaciones y las hipótesis asumidas serán representadas por predicados de la lógica. Entonces, la deducción de reglas de la lógica nos permitirá deducir si podemos reivindicar que la IUT sea conforme a la especificación. En efecto, la lógica se usará para comprobar si todas las implementaciones que pueden producir estas observaciones y satisfacer los requisitos de las hipótesis conforme a la especificación.

1.2.3. Conceptos y definiciones de HOTL

En este apartado presentaremos algunos conceptos que son básicos para una mejor comprensión del trabajo realizado. Por ejemplo, conceptos como maquina finita de estados (FSM), que es una relación de conformidad, así como otros conceptos de importancia relevante.

Máquina finita de estados (FSM):

Una FSM es una tupla de 5 elementos F = (S, inputs, outputs, I, T) donde S es un conjunto de estados, imputs es un conjunto de entradas activas, outputs un conjunto de salidas activas, I es un subconjunto de S y es el conjunto de estados iniciales, y T es el conjunto de transiciones.

Transición:

Un transición es una tupla de 4 elementos (s, i, o, s'), donde s y s' pertenecen al conjunto de estados S y son el estado inicial y final de la transición respectivamente, i pertenece al conjunto de entradas imputs y es la entrada activa en la transición, y por último o pertenece al conjunto de salidas outputs producido en respuesta.

isReachable (F, s1, s2):

Nos indica si un estado s2 es alcanzable desde un estado s1 para una Fsm F.

reachableStates(F, s):

Contiene todos los estados s'

isDet(F, s):

Nos dice si el estado s es determinista en la Fsm F.

Relación Conformance:

A continuación se explicará que es un modelo y una serie de conceptos concernientes a este. Un modelo será construido según las hipótesis y observaciones que consideremos. En particular, estos inducir un grafo consistente con las observaciones y las hipótesis anteriormente consideradas.

Modelo (m):

Denotaremos un modelo mediante una tupla de la forma m = (S, T, I, A, E, D, O). El significado de los diferentes elementos de la tupla es el siguiente:

- S (Estados): conjunto de estados que aparecen en el grafo del modelo.
- T (Transiciones): conjunto de transiciones que aparecen en el grafo del modelo.
- I (Estados iniciales): conjunto de estados iniciales en el modelo. En I pueden aparecer dos símbolos adicionales, estos son:
  - α : india que cualquier estado en S puede ser un estado inicial
  - β: indica que no solo los estados pertenecientes a S pueden ser estados iniciales (estos también implican α), sino que también estados no representados explícitamente en S podrían ser iniciales.
- A (Accounting): conjunto de registros contabilizadores.
- E (relaciones de igualdad): conjunto de igualdades que relacionan estados en S. una igualdad tendrá la siguiente forma: s is q, donde s es un estado de S y q pertenece a Q es un nombre identificador de estado.
- D (estados deterministas): conjunto de estados que son deterministas (de acuerdo a las hipótesis consideradas antes).
- (Observaciones usadas): conjunto de observaciones que fueron anteriormente usadas para la construcción de este modelo.

Model (correct (m)):

Indica que un modelo es correcto con respecto a una especificación.

Model (consistent (m)):

Significa que el modelo no incluye ninguna inconsistencia

ModelSubset (M’):

Este predicado se usa cuando queremos indicar que M’ es un subconjunto de modelos que puede escribir la implementación.

WorstCase:

El caso peor de un modelo m con respecto a una especificación dada se representa mediante una FSM y refleja el peor comportamiento que podría tener el modelo en aquellas partes donde no se sabe con certeza la respuesta que producirá.

$$\left(\begin{array}{l} \mathcal{S} \cup \{\bot\}, \mathbf{inputs}_{spec}, \mathbf{outputs}_{spec} \cup \{error\}, \\ \mathcal{T} \cup \left\{ s \xrightarrow{i/error} \bot \;\middle|\; \begin{array}{l} s \in \mathcal{S} \cup \{\bot\} \wedge i \in \mathbf{inputs}_{spec} \wedge \\ \nexists\, outs, f : (s, i, outs, f, \top) \in \mathcal{A} \wedge \\ (s \notin \mathcal{D} \vee \nexists s', o : s \xrightarrow{i/o} s' \in \mathcal{T}) \end{array} \right\}, \\ \mathcal{I}' \end{array}\right)$$

where $\mathcal{I}'$ is defined as

$$\mathcal{I}' = \begin{cases} \mathcal{I} & \text{if } \mathcal{I} \cap \{\alpha, \beta\} = \emptyset \\ \mathcal{S} & \text{if } \alpha \in \mathcal{I} \\ \mathcal{S} \cup \{\bot\} & \text{otherwise} \end{cases}$$

ModelElim:

La definición de la función *modelElim* se construye en dos pasos. Cuando eliminamos un estado del modelo y transferimos sus responsabilidades a otro modelo, tenemos que modificar todos los componentes definidos por el modelo.

Uno de los componentes a modificar es el “Accounting”. Esto se hace mediante la función *countElim*, esta función muestra como un accounting A es actualizado cuando un estado s2 es modificado por que es igual a un estado s1. Básicamente, moveremos toda la información del estado s2 al s1.

1.2.4. Observaciones

Las observaciones son unos de los predicados que formarán parte de HOTL. Estas las obtendremos de la IUT durante la fase preliminar clásica de testado. Las observaciones indicarán que, en respuesta a una secuencia de entradas dadas, la IUT produjo una secuencia de salidas dadas. Nuestra noción de observación incluirá algunas suposiciones sobre la IUT además del comportamiento observado.

Remarcar que si una de las secuencias muestra un comportamiento que es prohibido por la especificación, entonces la IUT no es conforme a la especificación y ya no será necesario un análisis, es decir, no hay necesidad de aplicar la lógica.

Las observaciones tendrán la siguiente forma: ob = $a_1,i_1/o_1,a_2,\ldots,a_n,i_n/o_n,a_{n+1}$, siendo las a_i los posibles atributos de cada estados, i_i la secuencia de entradas y o_i las secuencias de salida.

Los atributos denotan la información que asumimos acerca del estado y pueden ser:

- Vacío: No asumimos nada sobre el estado
- Det: Asumimos el determinismo del estado
- Imp (*q*): denota que el estado referenciado está asociado al estado *q* de la especificación.
- Spec (*s*): Solo aplicable al último estado de la observación e indica que el estado en el que nos encontramos es tal que el subgrafo que puede ser alcanzado desde el es isomorfo al subgrafo que puede ser alcanzado desde el estado *s* de la especificación, es decir, el subgrafo de la especificación que se obtiene desde el estado *s* se copia y pega al estado de la observación en el que nos encontramos.

1.2.5. Hipótesis

En el repertorio predefinido, las hipótesis se dividen en 2 clases: Hipótesis acerca de partes específicas de la IUT e hipótesis que conciernen a toda la IUT.

Para hacer referencia sin ambigüedad de los estados alcanzados por la primera, será adjuntada por las observaciones correspondientes que alcanzaron estos estados. Por ejemplo, si la IUT estuvo mostrando una secuencia de salidas o1, o2,…,on como respuesta a la secuencia de entradas i1, i2,…,in , el testador debe pensar que el estado alcanzado depuse de llevar a cabo i1/o1 es determinista o que el estado alcanzado después de llevar a cabo toda la secuencia i1/o1, i2/o2, …, in/on.

Remarcar que estas son las hipótesis que el testador está asumiendo. De esta manera, esta puede ser errónea y alcanzar una conclusión errónea. Sin embargo, esto es similar al caso cuando un testador supone que la implementación es determinista o que al menos tiene n estados y, en realidad, este no es el caso.

Además usando hipótesis asociadas a las observaciones, el testador puede considerar las hipótesis globales que conciernen a toda la IUT. Estas son suposiciones como una de las mencionadas antes: Suponiendo que la IUT es determinista, que tiene al menos n estados, que tiene un único estado inicial, etc. A fin de indicar la suposición de esta clase de hipótesis, predicados lógicos específicos serán usados.

Hacer notar que hay varios papeles donde las hipótesis probadas o testadas son utilizadas para desarrollar procesos de testado. Por ejemplo, podemos considerar que la implementación es determinista, que son probadas varias parejas de componentes suponiendo que todas ellas son correctas o que al menos una de ellas es incorrecta, etc.

Nuestra metodología proporciona una generalización de estos marcos porque permiten decidir las hipótesis específicas que consideraremos.

En esta línea podemos comparar la idoneidad de diferentes conjuntos de pruebas o un criterio de prueba en términos de las hipótesis que son consideradas; algunos conjuntos de prueba para comparar conjuntos de pruebas han sido definidos.

De nuestra lógica se proporciona un mecanismo para comparar de forma efectiva conjuntos de hipótesis, esto puede ayudar a computar las relaciones definidas en estos términos. Aunque trabajemos con reglas y propiedades, el trabajo no está relacionado con el “Model Cheking”, nosotros no vamos a comprobar la validez de propiedades: Suponemos que estas se sostienen y deduciremos resultados sobre la conformidad de la IUT usando estas suposiciones.

Tabla de Hipótesis

Hipótesis	**Fuente**	**Regla**	**Significado**
Det	Obs	obser	El estado correspondiente a IUT es determinista
imp(q)	Obs	obser	
spec(s)	Obs	obser	Desde el estado correspondiente a la IUT, el comportamiento coincide por uno dado por la especificación desde el estado s.
singleInit	Hyp	singleInit	Hay un único estado inicial en la IUT
allDet	Hyp	allDet	Todos los estados de IUT son deterministas
allTranHappenWith(n)	Hyp	allTran	Para cada estado y entrada, todas las transiciones saliendo del estado etiquetado por esta entrada son observadas después de ofrecer la entrada n veces.
upperBoundOfState(n)	Hyp	upper	La IUT tiene al menos n estados.
longSequencesSamePath(n)	Hyp	long	Todas las veces que una secuencia de longitud n es efectuada, los mismos estados de la IUT son atravesados.
uniqueOrigin(i, o)	Hyp	origin	Todas las transiciones etiquetadas con i/o salen desde el mismo estado.
uniqueDestination(i, o)	Hyp	destination	Todas las transiciones etiquetadas con i/o producen el mismo estado.

1.2.6. Reglas

Las reglas descritas en HOTL seguirán el formato premisas/conclusiones. El objetivo de las reglas de la lógica es deducir la "conformidad" de un conjunto de observaciones 'Obs' y de hipótesis 'Hyp', esto es, si todas las maquinas de estas finitas (FSMs) que conozcan estas condiciones son conformes a la especificación. Puesto que pueden aparecer modelos inconsistentes, la conformidad será admitida si existe al menos un modelo consistente que conozca estas premisas.

Para una mejor compresión del funcionamiento de las reglas y de su definición formal es aconsejable para el lector tener en cuenta los apartados anteriores de conceptos, definiciones, observaciones e hipótesis.

REGLA1 (obser):

Esta regla indica cómo construir un modelo desde una simple observación. Dando un predicado que indique que una observación fue tomada, la regla deducirá algunos detalles sobre el comportamiento de la implementación. Estos detalles son codificados por el significado de un modelo que muestre este comportamiento. Básicamente, los nuevos estados y las transiciones serán creadas en el modelo por lo que esto puede producir la observación. Aunque algunos estados del modelo podrían realmente coincidir, nosotros no consideramos estos hechos aún. De esta manera, tomaremos estados frescos para nombrar a todos ellos. Además, las hipótesis indicaran por los atributos de la observación que influirá en la información asociada a los estados del modelo correspondiente.

En particular, si el testador supone que el último estado de la observación es isomórfico a un estado de la especificación entonces los conjuntos de estados, transiciones, registros de contabilidad ("accounting"), y estados deterministas serán extendidas con algún elemento extra desde la especificación e indicado por S', T', A', y D', respectivamente. Los nuevos estados y transiciones S' y T', respectivamente, copiará la estructura existente entre los estados que pueden ser alcanzados desde s en la especificación., el nuevo accounting, A', el conocimiento concerniente a los nuevos estados es "cerrado" para todas las entradas, esto es, las únicas transiciones salientes desde estos estados son esos que copiamos desde la especificación y ninguna otra transición será añadida en el futuro. Finalmente, esos estados del modelo que corresponden a estados deterministas de la especificación serán incluidos en el conjunto D' de estados deterministas del modelo.

$$(obser)\frac{ob = (a_1, i_1/o_1, a_2, \ldots, a_n, i_n/o_n, a_{n+1}) \in \texttt{Obs} \ \wedge \ s_1, \ldots, s_{n+1} \text{ are fresh states}}{\texttt{model}\begin{pmatrix} \{s_1, \ldots, s_{n+1}\} \cup \mathcal{S}', \\ \{s_1 \xrightarrow{i_1/o_1} s_2, \ldots, s_n \xrightarrow{i_n/o_n} s_{n+1}\} \cup \mathcal{T}', \{s_1, \beta\}, \\ \{(s_j, i_j, \{o_j\}, f_{s_j}, 1) \mid 1 \leq j \leq n\} \cup \mathcal{A}', \\ \{s_j \texttt{ is } q_j | 1 \leq j \leq n+1 \ \wedge \ \texttt{imp}(q_j) \in a_j\}, \\ \{s_j \mid 1 \leq j \leq n+1 \ \wedge \ \texttt{det} \in a_j\} \cup \mathcal{D}', \{ob\} \end{pmatrix}}$$

where $f_{s_j}(t) = 1$ if $t = s_j \xrightarrow{i_j/o_j} s_{j+1}$ and $f_{s_j}(t) = 0$ otherwise.

En el caso de que se añada como último atributo la hipótesis de que el estado es igual al estado s de la especificación, es decir, aparece *spec(s)* como atributo, entonces el modelo viene determinado además por:

$$U = \{u_j \mid u_j \text{ is a fresh state } \wedge \ 1 \leq j < |\texttt{reachableStates}(spec, s)|\}$$

and a bijective function $g : \texttt{reachableStates}(spec, s) \longrightarrow U \cup \{s_{n+1}\}$ such that $g(s) = s_{n+1}$. Then, $(\mathcal{S}', \mathcal{T}', \mathcal{A}', \mathcal{D}')$ is equal to

$$\begin{pmatrix} U, \\ \{g(s') \xrightarrow{i/o} g(s'') \mid s' \xrightarrow{i/o} s'' \in \mathcal{T}_{spec} \ \wedge \ \texttt{isReachable}(spec, s, s')\}, \\ \left\{ (u, i, \texttt{outs}(spec, g^{-1}(u), i), f_u^i, \top) \middle| \begin{array}{l} u \in U \cup \{s_{n+1}\} \ \wedge \ i \in \texttt{inputs}_{spec} \ \wedge \\ \exists u' \in U, o \in \texttt{outputs}_{spec} : \ u \xrightarrow{i/o} u' \in \mathcal{T}' \end{array} \right\}, \\ \{g(s') | \texttt{isReachable}(spec, s, s') \ \wedge \ \texttt{isDet}(spec, s')\} \end{pmatrix}$$

where $f_u^i(t) = 1$ for all t such that there exists o', u' with $t = u \xrightarrow{i/o'} u' \in \mathcal{T}'$ and $f_u^i(t) = 0$ otherwise.

REGLA2 (fusion):

La regla 2 o de fusión nos permitirá unificar diferentes modelos construidos desde diferentes observaciones en un único modelo. Estos componentes del nuevo modelo serán la unión de los componentes de cada modelo.

$$(fusion)\frac{\begin{array}{c}\texttt{model}\,(\mathcal{S}_1,\mathcal{T}_1,\mathcal{I}_1,\mathcal{A}_1,\mathcal{E}_1,\mathcal{D}_1,\mathcal{O}_1)\ \wedge\\ \texttt{model}\,(\mathcal{S}_2,\mathcal{T}_2,\mathcal{I}_2,\mathcal{A}_2,\mathcal{E}_2,\mathcal{D}_2,\mathcal{O}_2)\ \wedge\ \mathcal{O}_1\cap\mathcal{O}_2=\emptyset\end{array}}{\texttt{model}\left(\mathcal{S}_1\cup\mathcal{S}_2,\mathcal{T}_1\cup\mathcal{T}_2,\mathcal{I}_1\cup\mathcal{I}_2,\mathcal{A}_1\cup\mathcal{A}_2,\mathcal{E}_1\cup\mathcal{E}_2,\mathcal{D}_1\cup\mathcal{D}_2,\mathcal{O}_1\cup\mathcal{O}_2\right)}$$

Indicar que si aplicamos las dos primeras reglas de forma iterativa, podremos obtener finalmente un modelo donde O incluye todas las observaciones pertenecientes al conjunto Obs.

REGLA3 (set):

Durante esta nueva fase, necesitaremos varios modelos para representar todas las FSMs que son compatibles con un conjunto de observaciones e hipótesis. Esta regla nos permitirá representar un único modelo por medio de un conjunto que contiene un único elemento. Las reglas siguientes serán concernientes a las segunda fase y en todos las casos tendremos O=Obs.

$$(set)\frac{\texttt{model}\,(\mathcal{S},\mathcal{T},\mathcal{I},\mathcal{A},\mathcal{E},\mathcal{D},\texttt{Obs})}{\texttt{models}\,(\{(\mathcal{S},\mathcal{T},\mathcal{I},\mathcal{A},\mathcal{E},\mathcal{D},\texttt{Obs})\})}$$

REGLA 4 (propagation):

Esta regla permite poder reflejar como una regla que aplica un único modelo afecta al conjunto que incluye este modelo. ϕ denota cualquier predicado lógico y m un modelo.

$$(propagation)\frac{\texttt{models}\,(\mathcal{M}\cup\{m\})\wedge\varphi\wedge((\texttt{model}\,(m)\wedge\varphi)\vdash\texttt{modelsSubset}\,(\mathcal{M}'))}{\texttt{models}\,(\mathcal{M}\cup\mathcal{M}')}$$

Usando esta regla, podremos usar otras reglas para aplicar a un único modelo y entonces propagar los cambios de un conjunto donde el modelo es incluido.

REGLA 5(equality):

La regla 5 o 'equality' nos presentar como unir dos estados si el conjunto de igualdades (equalities) permite deducir que ambos coinciden.

$$(equality)\frac{\texttt{model}\,(\mathcal{S},\mathcal{T},\mathcal{I},\mathcal{A},\mathcal{E},\mathcal{D},\texttt{Obs})\ \wedge\ \ s_1,s_2\in\mathcal{S}\ \wedge\ \{s_1\ \texttt{is}\ q, s_2\ \texttt{is}\ q\}\subseteq\mathcal{E}}{\texttt{modelsSubset}\,(\texttt{modelElim}((\mathcal{S},\mathcal{T},\mathcal{I},\mathcal{A},\mathcal{E},\mathcal{D},\texttt{Obs}),s_1,s_2))}$$

REGLA6 (Determ):

Otra situación donde dos estados pueden ser fusionados aparece cuando un estado determinista muestra dos transiciones etiquetadas con la misma entrada, desde que un estado es determinista, estos puede ser también etiquetados con la misma salida. El determinismo de estados implica que ambos destinos son el mismo estado. Por lo tanto, estos dos estados alcanzados pueden ser fusionados. Subrayar que si ambas salidas son diferentes en el modelo es inconsistente, porque el determinismo del estado no se ha preservado. En este caso, un conjunto vacío de modelos producidos.

$$(determ)\frac{\begin{array}{c}\texttt{model}\,(m)\ \wedge\ m=(\mathcal{S},\mathcal{T},\mathcal{I},\mathcal{A},\mathcal{E},\mathcal{D},\texttt{Obs})\ \wedge\\ s,s_1,s_2\in\mathcal{S}\ \wedge\ s\in\mathcal{D}\ \wedge\ \{s\xrightarrow{i/o_1}s_1, s\xrightarrow{i/o_2}s_2\}\subseteq\mathcal{T}\end{array}}{\texttt{modelsSubset}\,(\mathcal{M}')}$$

where $\mathcal{M}'=\texttt{modelElim}(m,s_1,s_2)$ if $o_1=o_2$ and $\mathcal{M}'=\emptyset$ otherwise.

REGLA 7 (SingleInit):

'SingleInit' es la primera regla que está relacionada con una hipótesis que no está dada implícitamente por una observación. De este modo, consideraremos un elemento perteneciente al conjunto Hyp. Esta hipótesis permite asumir que el estado inicial de la implementación es único. En caso de ocurrir esta situación todos los estados iniciales serán fusionados. Además, cualquier símbolo en I puede indicar que cualquier otro estado puede ser inicial, esto es, α y β, serán eliminados.

$$(singleInit)\frac{\begin{array}{c}\mathtt{model}\,(\mathcal{S},\mathcal{T},\mathcal{I},\mathcal{A},\mathcal{E},\mathcal{D},\mathtt{Obs})\ \wedge\ \mathcal{I}\cap\mathcal{S}=\{s_1,\ldots,s_n\}\ \wedge\\ \mathtt{singleInit}\in\mathtt{Hyp}\ \wedge\ m'=(\mathcal{S},\mathcal{T},\mathcal{I}\backslash\{\alpha,\beta\},\mathcal{A},\mathcal{E},\mathcal{D},\mathtt{Obs})\end{array}}{\mathtt{modelsSubset}\,(\mathtt{modelElim}(m',s_1,\{s_2,\ldots,s_n\}))}$$

REGLA 8 (allDet):

Si el testador añade la hipótesis de que todos los estados son deterministas entonces el conjunto completo de estados S coincide con el conjunto de estados deterministas D.

$$(allDet)\frac{\mathtt{model}\,(\mathcal{S},\mathcal{T},\mathcal{I},\mathcal{A},\mathcal{E},\mathcal{D},\mathtt{Obs})\ \wedge\ \mathtt{allDet}\in\mathtt{Hyp}}{\mathtt{modelsSubset}\,(\{(\mathcal{S},\mathcal{T},\mathcal{I},\mathcal{A},\mathcal{E},\mathcal{S},\mathtt{Obs})\})}$$

REGLA 9 (consistent):

La siguiente regla detecta que un modelo es consistente. Esto requiere que ninguna otra regla de las que las hipótesis manejan pueda modificar el modelo. En concreto, estas reglas son todas las reglas pertenecientes a R que hemos visto hasta el momento.

$$(consistent)\frac{\begin{array}{c}m=(\mathcal{S},\mathcal{T},\mathcal{I},\mathcal{A},\mathcal{E},\mathcal{D},\mathtt{Obs})\ \wedge\ \mathtt{model}\,(m)\ \wedge\\ \mathtt{unable}(m,\mathcal{R}\backslash\{consistent,correct\})\end{array}}{\mathtt{modelsSubset}\,(\{consistent(m)\})}$$

REGLA 10 (correct):

Desde que un modelo es una representación (probablemente incompleta) de la IUT, para poder comprobar si un modelo es conforme a la especificación se tienen que tener en cuenta dos aspectos:

- Sólo la conformidad de modelos consistentes será considerada.

- Dado un modelo consistente, probaremos que este es conforme con respecto a la especificación considerando la peor instancia del modelo, esto es, si la instancia es conforme a la especificación entonces cualquier otra instancia extraída del modelo también lo será.

AQUÍ AHORA HABLA DE CÓMO SE FORMAEL WORST CASE, NO SE SI METERLO AQUÍ O EN DEFINICIONES O Q HACER CON EL.

Así la regla que indica la corrección de un modelo es:

$$(correct)\frac{m = (\mathcal{S},\mathcal{T},\mathcal{I},\mathcal{A},\mathcal{E},\mathcal{D},\texttt{Obs}) \wedge \\ \texttt{model}\,(consistent(m)) \;\wedge\; \texttt{worstCase}(m)\;\texttt{conf}\;spec}{\texttt{modelsSubset}\,(\{correct(m)\})}$$

REGLA11 (allCorrect):

Ahora podemos considerar la conformidad de un conjunto de modelos. Un conjunto es conforme a la especificación si todos los elementos lo son y el conjunto contiene al menos un elemento.

Apuntar que un conjunto vacío de modelos indica que todos los modelos eran inconsistentes. Por lo tanto, admitir la conformidad de un conjunto vacío implicaría la aceptación de modelos que no representa ninguna implementación.

$$(allCorrect)\frac{\texttt{models}\,(\mathcal{M}) \;\wedge\; \mathcal{M} \neq \emptyset \;\wedge\; \mathcal{M} = \{correct(m_1),\ldots,correct(m_n)\}}{\texttt{allModelsCorrect}}$$

REGLA 12 (allTran):

Para poder indicar la funcionalidad de esta regla hay es necesario saber primero que indica la hipótesis 'allTranHappenWith(n)' (incluida dentro de la tabla de hipótesis que se encuentra en el apartado de estas). Lo que esta hipótesis suponía era que si una entrada es producida n veces dando un estado, entonces podremos observar todas las salidas que pueden ser producidas por este estado en respuesta a estas salidas y trasladar todos los estados que podamos mover desde este estado con esta entrada.

En particular, suponemos que todas las transiciones que salen de este estado en esta entrada son observadas. Denotaremos que podremos suponer esta hipótesis en un modelo que realmente no la cumple.

$$(allTran)\frac{\begin{array}{l}\texttt{model}\,(\mathcal{S},\mathcal{T},\mathcal{I},\mathcal{A},\mathcal{E},\mathcal{D},\texttt{Obs})\ \wedge \\ \mathcal{M}'=\{(\mathcal{S},\mathcal{T},\mathcal{I},\mathcal{A}',\mathcal{E}\cup\mathcal{E}',\mathcal{D},\mathcal{O})\mid \mathcal{E}'\in K\}\ \wedge \\ n\in\mathbb{N}\ \wedge\ \texttt{allTranHappenWith}(n)\in\texttt{Hyp}\end{array}}{\texttt{modelsSubset}\,(\mathcal{M}')}$$

Donde:

$$\mathcal{A}' = \{(s,i,outs,f,n')|(s,i,outs,f,n')\in\mathcal{A}, n'<n\}$$
$$\bigcup$$
$$\{(s,i,outs,f,\top)|(s,i,outs,f,n')\in\mathcal{A}, n'\geq n\}$$

Y el conjunto k se define como sigue:

(a) Let $(s,i,outs,f,n')\in\mathcal{A}$ such that $|T_s^i|>n$. Let us suppose that there exist $T'\subseteq T_s^i$, with $\sum\{f(t)|t\in T'\}\geq n$, and a transition $s\xrightarrow{i/o}s'\in T_s^i$ such that for all $s\xrightarrow{i/o'}s''\in T'$ we have $o\neq o'$. Then, $K=\emptyset$ (i.e., $\mathcal{M}'=\emptyset$).

(b) Otherwise, K is the set containing all the sets $\mathcal{E}'$ fulfilling:

- For all $(s,i,outs,f,n')\in\mathcal{A}$ such that $|T_s^i|>n$ and set $T'\subseteq T_s^i$, with $T'=\{s\xrightarrow{i/o_1}s_1,\ldots,s\xrightarrow{i/o_q}s_q\}$ such that $\sum\{f(t)|t\in T'\}\geq n$, let us consider the set $\{u_i\mid 1\leq i\leq q'\}=\{u|\exists\ o: s\xrightarrow{i/o}u\in T_s^i\backslash T'\}$. Then, for some set of q fresh state identifier names $\{w_i\mid 1\leq i\leq q\}$ we have $\{s_i\ \texttt{is}\ w_i\mid 1\leq i\leq q\}\subseteq\mathcal{E}'$ and $\{u_i\ \texttt{is}\ w_{r_i}\mid 1\leq i\leq q'\}\subseteq\mathcal{E}'$, where for all $1\leq k\leq q'$ there exists o_k such that $s\xrightarrow{i/o_k}u_k\in T_s^i\backslash T'$ and $s\xrightarrow{i/o_k}s_{r_k}\in T'$, with $1\leq r_k\leq q$.
- There are not more elements belonging to $\mathcal{E}'$.

If there does not exist such a set $\mathcal{E}'$ fulfilling the previous properties then we consider $K=\{\emptyset\}$ (i.e., $\mathcal{M}'=\{(\mathcal{S},\mathcal{T},\mathcal{I},\mathcal{A}',\mathcal{E},\mathcal{D},\texttt{Obs})\}$).

REGLA 13 (upper):

La hipótesis 'upperBoundOfStates(n)' nos indicaba que podemos suponer un límite superior en el número de estados. Así en fusiones de estados próximas produciremos un modelo donde al menos n estados serán usados.

En particular, cada estado le será asignado un nombre identificador de estado tomado de un conjunto de n nombre. Cada manera de asignar estos nombres producirá un modelo diferente que es incluido en el conjunto de modelos deducidos.

$$(upper)\frac{\begin{array}{l}\texttt{model}\,(\mathcal{S},\mathcal{T},\mathcal{I},\mathcal{A},\mathcal{E},\mathcal{D},\texttt{Obs})\ \wedge\\ \mathcal{M}'=\{(\mathcal{S},\mathcal{T},\mathcal{I}',\mathcal{A},\mathcal{E}\cup\mathcal{E}',\mathcal{D},\texttt{Obs})\mid \mathcal{E}'\in K\}\ \wedge\\ n\in\mathbb{N}\ \wedge\ \texttt{upperBoundOfStates}(n)\in\texttt{Hyp}\end{array}}{\texttt{modelsSubset}\,(\mathcal{M}')}$$

where $\mathcal{I}'=(\mathcal{I}\backslash\{\beta\})\cup\{\alpha\}$ if $\beta\in\mathcal{I}$ and $\mathcal{I}'=\mathcal{I}$ otherwise. In order to define the set K, we have $K=\{\emptyset\}$ if $|\mathcal{S}|\leq n$; otherwise, let $\mathcal{S}=\{s_1,\ldots,s_q\}$ and $\{w_1,\ldots,w_n\}$ be a set of n fresh state identifier names. Then, K is the set of all the sets $\mathcal{E}'$ fulfilling $\{s_i\ \texttt{is}\ w_{r_i}\mid 1\leq i\leq q\}\subseteq\mathcal{E}'$, where for all $1\leq i\leq q$ we have $1\leq r_i\leq n$, and $\mathcal{E}'$ does not contain other elements.

REGLA 14 (long):

Esta regla está basada en la hipótesis 'longSequencesSamePath' que supone que todas las secuencias o trazas que tienen al menos n transiciones y producen la misma secuencia de entradas y de salidas realmente atraviesan la misma implementación de estados. En este caso, COBINAREMOS (match)? Estos estados que son los mismos puntos de cada una de estas secuencias. Para poder hacer esto, uniremos los estados que se corresponde con puntos equivalentes por adición de nuevas igualdades en el conjunto de igualdades.

$$(long)\frac{\texttt{model}\,(\mathcal{S},\mathcal{T},\mathcal{I},\mathcal{A},\mathcal{E},\mathcal{D},\texttt{Obs})\ \wedge\ n\in\mathbb{N}\ \wedge\ \texttt{longSequencesSamePath}(n)\in\texttt{Hyp}}{\texttt{modelsSubset}\,(\{(\mathcal{S},\mathcal{T},\mathcal{I},\mathcal{A},\mathcal{E}\cup\mathcal{E}',\mathcal{D},\texttt{Obs})\})}$$

where

$$\mathcal{E}'=\left\{\begin{array}{c|l} s_i\ \texttt{is}\ w_i, & s_1\xrightarrow{i_1/o_1}s_2\in\mathcal{T},\ldots,s_n\xrightarrow{i_n/o_n}s_{n+1}\in\mathcal{T}\ \wedge\\ & s'_1\xrightarrow{i_1/o_1}s'_2\in\mathcal{T},\ldots,s'_n\xrightarrow{i_n/o_n}s'_{n+1}\in\mathcal{T}\ \wedge\\ s'_i\ \texttt{is}\ w_i & 1\leq i\leq n+1\ \wedge\ s_i\neq s'_i\ \wedge\ w_i\ \text{fresh state identifier name}\end{array}\right\}$$

REGLA 15 (Origin):

'uniqueOrigin (i, o)' es la hipótesis en la que se basará esta regla. Dicha hipótesis nos permite suponer que la salida de un estado coincide para todas las transiciones de la IUT etiquetadas por el par i/o. identificaremos estas transiciones en la IUT y podremos COMBINAR (MATCH)? Todos los estados salientes asignándoles el mismo nombre identificar de estado.

$$(origin)\frac{\texttt{model}\,(\mathcal{S},\mathcal{T},\mathcal{I},\mathcal{A},\mathcal{E},\mathcal{D},\texttt{Obs})\ \wedge\ \texttt{uniqueOrigin}(i,o)\in\texttt{Hyp}}{\texttt{modelsSubset}\,(\{(\mathcal{S},\mathcal{T},\mathcal{I},\mathcal{A},\mathcal{E}\cup\mathcal{E}',\mathcal{D},\texttt{Obs})\})}$$

where in order to define $\mathcal{E}'$, let us consider a fresh state identifier w and let $Y = \{s \texttt{ is } w \mid \exists\, s' : s \xrightarrow{i/o} s' \in \mathcal{T}\}$. Then, $\mathcal{E}' = \emptyset$ if $|Y| < 2$ and $\mathcal{E}' = Y$ otherwise.

REGLA 16 (Destination):

De la misma manera la hipótesis 'uniqueDestination (i, o)' es también la base de esta última regla. En este caso supondremos que todas las transiciones de la IUT etiquetadas por el par i/o producen el mismo estado.

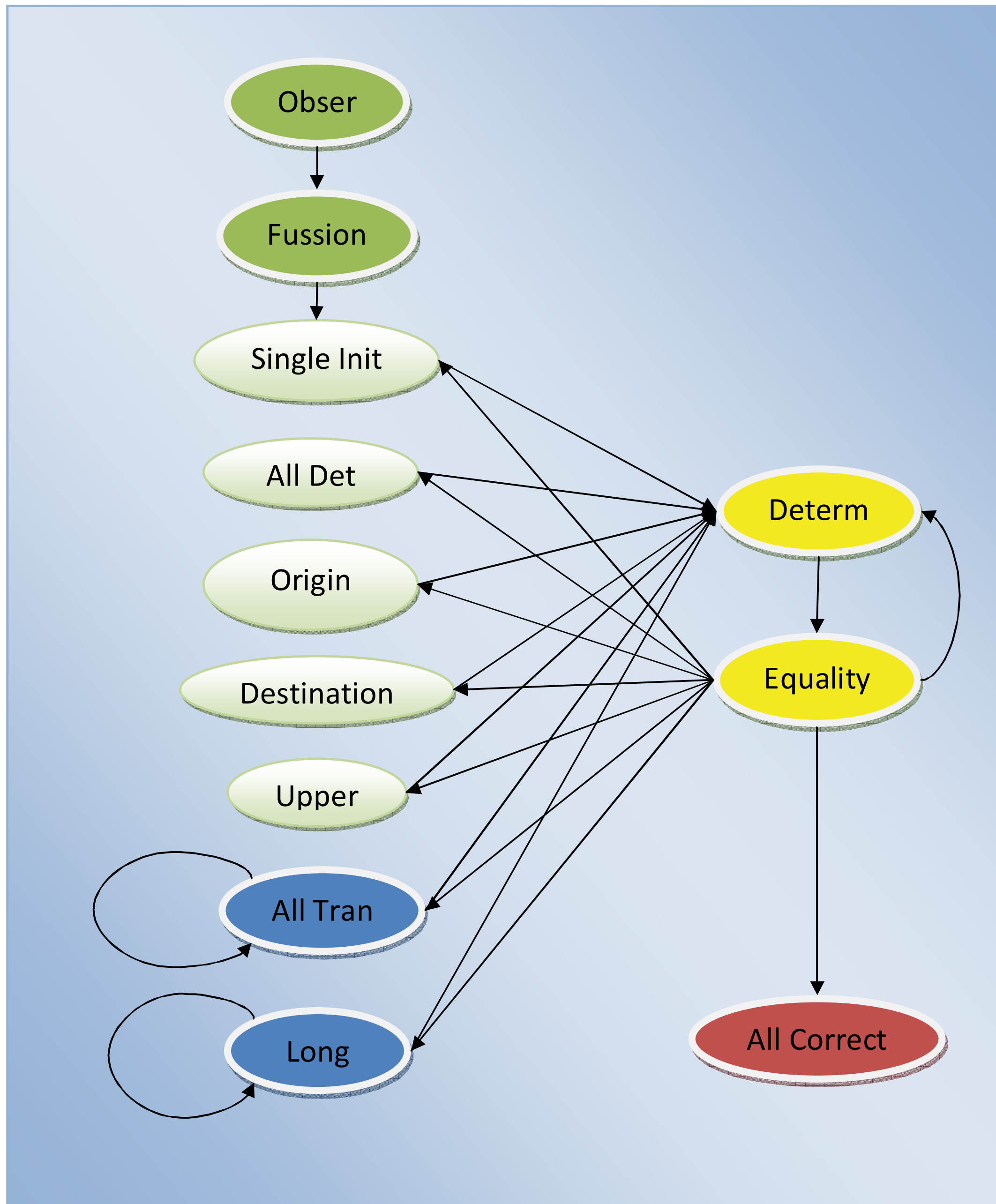
$$(destination)\frac{\texttt{model}\,(\mathcal{S},\mathcal{T},\mathcal{I},\mathcal{A},\mathcal{E},\mathcal{D},\texttt{Obs})\ \wedge\ \texttt{uniqueOrigin}(i,o)\in\texttt{Hyp}}{\texttt{modelsSubset}\,(\{(\mathcal{S},\mathcal{T},\mathcal{I},\mathcal{A},\mathcal{E}\cup\mathcal{E}',\mathcal{D},\texttt{Obs})\})}$$

where in order to define $\mathcal{E}'$, let us consider a fresh state identifier w and let $Y = \{s \texttt{ is } w \mid \exists\, s' : s' \xrightarrow{i/o} s \in \mathcal{T}\}$. Then, $\mathcal{E}' = \emptyset$ if $|Y| < 2$ and $\mathcal{E}' = Y$ otherwise.

1.2.7. Algoritmo de ejecución (Máquina de estados)

Para garantizar la finalización de la aplicación de las reglas a un conjunto de observaciones, hipótesis y FSM en un número finito de pasos es necesario seguir un orden determinado. Dicho orden garantiza la terminación pero es posible que otro orden de aplicación pueda terminar.

Máquina de estados de HOTL



1.3. Resultados esperados

Una vez se nos había presentado el proyecto y teníamos claro de qué trataba, cuál era su dificultad y qué era lo que se espera de él, estaba claro que nuestro objetivo, dentro del tiempo disponible, era conseguir una aplicación, gráfica o no, capaz de ejecutar todas las reglas de la lógica dado un modelo y en última instancia si, dadas una especificación concreta, unas observaciones, unas hipótesis y la aplicación de las reglas sobre estos elementos nos diga si la implementación es conforme o no.

Durante el desarrollo del proyecto sabíamos que nos encontraríamos con una serie de limitaciones que iban a contribuir a que el desarrollo del mismo no fuera ni muchos menos sencillo, todo lo contrario, nos iba a exigir una gran carga de trabajo y, un elemento que consideramos importantísimo, que iba a ser la unión y el trabajo en equipo de cada uno de los miembros del grupo.

Estas limitaciones hemos considerado conveniente englobarlas en dos grandes grupos, limitaciones teóricas y prácticas, con elementos en ambas que influyen en el grupo contrario.

1.3.1. Limitaciones teóricas

Para el desarrollo del proyecto era fundamental entender muy bien el trabajo de investigación sobre el que nos basábamos. La compresión de este nos llevó un tiempo considerable, sobre todo por la enorme carga teórica de dicho trabajo de investigación. Otro aspecto que nos dificultaba un poco el trabajo era que la documentación disponible, y en la que nos basábamos, era en un inglés técnico en el que, muchos de los términos y expresiones no eran fáciles para nosotros teniendo en cuenta el nivel de inglés de cada uno de los miembros del grupo.

Otro dificultad más era la forma de aplicar algoritmos a la forma de implementación. La aplicación de las reglas en varios casos conllevaría que algunos de estos fueran exponenciales. Así, tendríamos un número considerable de recorridos exponenciales de modelos almacenados en cada paso.

A parte de tener recorridos exponenciales, las reglas pueden dar como resultado un número muy elevado de modelos y, ser a su vez estos de un tamaño considerable, provocando un aumento del tiempo de procesamiento.

1.3.2. Limitaciones prácticas

En la práctica, uno de nuestros objetivo era que fuésemos capaces de llevar a cabo de forma eficiente y práctica una implementación de modelos, observaciones, hipótesis, etc.

Una dificultad que hay que remarcar es la implantación de ciertos algoritmos para ciertas reglas, pues, una cosa es saber que una regla es exponencial y otra es que la implementación de código no sólo no haga empeorar ya este altísimo coste, sino que se intente mejorar para que sea menor dentro de esta exponencialidad. Así el objetivo marcado aquí era intentar mantener o mejorar el coste dentro de lo esperado y que la aplicación fuese capaz de realizar los ejemplos del artículo en el que nos basábamos o ejemplos pequeños.

Por último, sabíamos que la interfaz iba a ser complicada sobre todo a la hora de mostrar de forma clara al usuario los modelos resultantes, pues, dependiendo de las observaciones e hipótesis elegidas, estos modelos pueden ser muchos y a su vez muy complejos.

1.4. Metodología de trabajo

1.4.1. Introducción

La aplicación de técnicas de ingeniería del software al desarrollo de esta aplicación nos proporciona un enfoque sistemático, disciplinado y cuantificable tanto del proceso de desarrollo en sí, como de la operación y mantenimiento de la misma. Con esto conseguimos ciertas garantías de que desarrollaremos un software de calidad en un espacio de tiempo y con unos costes razonables.

En el presente punto pretendemos hacer un pequeño resumen de la información referente a la gestión del proyecto analizando diversos aspectos como son los riesgos, el modelo de proceso, los requisitos, y la planificación temporal entre otros.

1.4.2. Análisis del riesgo

En este apartado definiremos lo que entendemos por riesgo, identificaremos aquellos que tienen mayor relevancia en nuestro proyecto y los priorizaremos. Gracias a esto no solo seremos conscientes de los riesgos que pueden afectar negativamente a la resolución de nuestro proyecto, sino que adoptaremos medidas para minimizar la probabilidad de que se conviertan en problemas y poder actuar a tiempo contra ellos.

Entendemos por riesgo todo aquello que pueda afectar negativamente a nuestro proyecto software. Obviamente si nos ajustamos a esta definición, la lista de riesgos que pueden entorpecer, e incluso impedir, la finalización del proyecto es interminable. Pero el sentido común nos dice que hay riesgos potenciales con mucha mayor probabilidad de materializarse que otros.

Existen diversas técnicas para la identificación de riesgos. Una de ellas son las listas de comprobación. Son listas de los riesgos más comunes en la realización de proyectos software, los cuales podemos comprobar si son considerables para el nuestro. Emplearemos los Top 10 Software Risk ítems de Boehm:

A cada uno de ellos le asignaremos la probabilidad de que se convierta en un problema (improbable, remoto, ocasional, o probable) y el impacto que esto tendría en nuestro proyecto (menor, serio, crítico, o catastrófico). Así nos resulta la siguiente tabla:

Tabla De Análisis De Riesgos

Elemento de Riesgo	Probabilidad	Impacto
Deficiencia de personal	Remoto	Catastrófico
Planificaciones poco realistas	Probable	Crítico
Desarrollo de propiedades y funciones erróneas	Ocasional	Serio
Desarrollo erróneo de la interfaz de usuario	Ocasional	Serio
Deficiencias en componentes proporcionados externamente	Remoto	Crítico
No ajustarnos a la planificación temporal	Probable	Crítico
Uso de tecnologías desconocidas	Probable	Crítico
No disponibilidad de ordenadores para el desarrollo de la aplicación	Ocasional	Menor
Incompatibilidades entre las versiones del software utilizado	Probable	Menor

1.4.3. Planificación temporal

Este es uno de los puntos clave del plan de proyecto. En el asignaremos las tareas concretas en las que hemos divido el desarrollo del proyecto a cada miembro con una estimación del tiempo que debería emplear en llevarla a cabo. Gracias a esta planificación cada miembro del equipo sabe con suficiente antelación qué se espera que haga y cuándo. Además será la base principal sobre la que juzgaremos si el estado de desarrollo del proyecto en un momento dado es satisfactorio.

Es necesario revisar y actualizar la planificación periódicamente si queremos que cumpla su principal objetivo, que no es otro que el de evitar retrasos en el desarrollo.

1.4.4. Modelo de proceso

La combinación de las múltiples demandas que la IS provoca en el desarrollo de un proyecto, y de la inexperiencia de los miembros del equipo en este tipo de

actividades, realza todavía más la importancia de acogernos a un modelo de proceso que imponga racionalidad en la organización de nuestro trabajo.

Para el desarrollo de esta aplicación hemos tenido desde el primer momento claro que la forma en que se debía actuar era incremental. Es decir, partir de unos pequeños requisitos ir construyendo el esqueleto y poco a poco evolucionarlo para añadir mayores funcionalidades.

En nuestro caso la especificación es clara y los requisitos están fijados desde el principio y son, en principio, inmutables. Sin embargo, creemos que este tipo de modelos de proceso son idóneos para minimizar el impacto que supone la inexperiencia de los miembros del equipo en el desarrollo de este tipo de proyectos.

Tras comparar las ventajas y desventajas de varios modelos (espiral de Boehm, espiral de Boston y Proceso Unificado de Desarrollo) nos decimos por el modelo Espiral de Boston. Nos pareció suficientemente estricto como para dar un soporte sólido a nuestro trabajo pero suficientemente flexible como para adaptarse a nuestras necesidades.

A lo largo del desarrollo de nuestro producto software realizaremos varias iteraciones (es decir, vueltas a la citada espiral), pasando en cada una de ellas por las siguientes seis actividades estructurales:

- Comunicación con el tutor: Consideramos aquí la actividad de reuniones con el tutor para comentar aspectos de la aplicación, nivel de análisis, diseño y sobre todo inicialmente garantizar la correcta comprensión de la base teórica del proyecto.
- Planificación: Tarea encargada de la definición de recursos y temporalidad del proyecto.
- Gestión del riesgo: Actividad encargada de la valoración del riesgo tanto técnico como de gestión del proyecto.
- Ingeniería: Actividad encargada de la construcción de una representación de la aplicación. Esta actividad consta de dos subactividades, Análisis y Diseño.
- Construcción y Adaptación: Tareas dedicadas a la generación de código de los diferentes módulos de la aplicación (codificación), su prueba y el ensamblado de estos módulos.

Evaluación por el tutor: Revisión por parte del tutor de las distintas versiones parciales del proyecto.

2. DESARROLLO DE LA APLICACIÓN

2.1. Tecnologías utilizadas

Para el desarrollo de la aplicación han sido necesarios los siguientes componentes:

- *Eclipse 3.2*: Eclipse es un entorno de desarrollo integrado de código abierto independiente de una plataforma para desarrollar lo que el proyecto llama "Aplicaciones de Cliente Enriquecido", opuesto a las aplicaciones "Cliente-liviano" basadas en navegadores. Esta plataforma, típicamente ha sido usada para desarrollar entornos de desarrollo integrados (del inglés IDE), como el IDE de Java llamado *Java Development Toolkit* (JDT) y el compilador (ECJ) que se entrega como parte de Eclipse (y que son usados también para desarrollar el mismo Eclipse).
- *API Java*: El API Java es una Interfaz de Programación de Aplicaciones (API: por sus siglas en ingles) provista por los creadores del lenguaje Java, y que da a los programadores los medios para desarrollar aplicaciones Java. Como el lenguaje Java es un Lenguaje Orientado a Objetos, la API de Java provee de un conjunto de clases utilitarias para efectuar toda clase de tareas necesarias dentro de un programa. La API Java está organizada en paquetes lógicos, donde cada paquete contiene un conjunto de clases relacionadas semánticamente.
- *JDK 1.6*: Java Development Kit o (JDK), es un software que provee herramientas de desarrollo para la creación de programas en java. Puede instalarse en una computadora local o en una unidad de red. En la unidad de red se puede tener la aplicación distribuida en varias computadoras y trabajar como una sola aplicación. En especial *JDK 1.6* posee la librería *Vector* la cual ha sido de gran ayuda a la hora de la implementación de las FSM y de los modelos.
- *JGraph:* Es una de los más poderosos y sencillos componentes de Java para la representación de objetos. Es de código abierto, desarrollado completamente en Java y está integrado en la jerarquía de Swing. Está basado en los patrones de Swing MVC.

2.2. Organización interna de la aplicación

2.2.1. Arquitectura

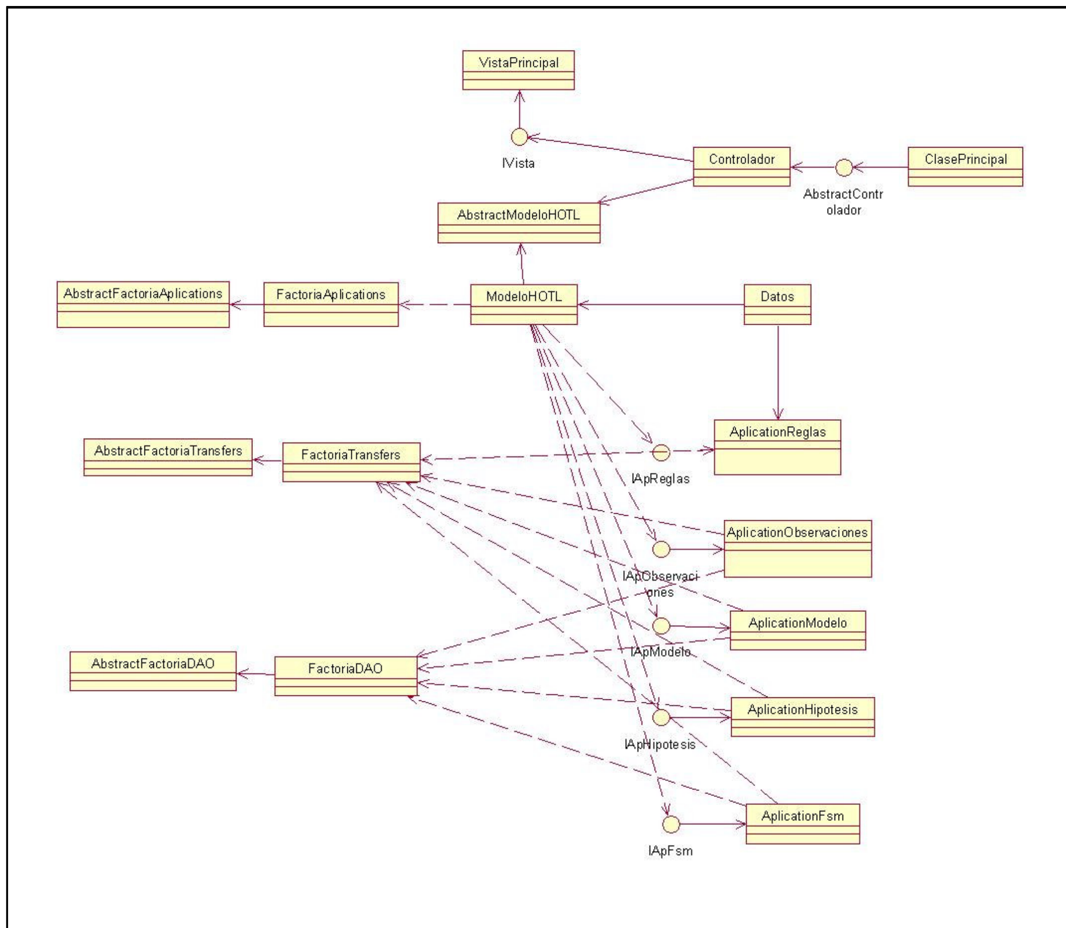
Una de los principales objetivos en el desarrollo de la aplicación de HOTL ha sido el mantener una programación adecuadamente estructurada y en la que se apliquen técnicas de IS.

Debido a esto hemos optado por una arquitectura multicapa, que a pesar de ser la más compleja de las distintas arquitecturas es la que mayores ganancias ofrece, en la que se diferencian claramente tres capas, la capa de integración, de negocio y de presentación.

- *La capa Presentación*, encargada de la interfaz, en ella se usaran principalmente objetos de la clase Javax.Swing (la última versión del JDK 1.6) ya que la aplicación está orientada hacia un uso en ordenadores personales u otro tipo de maquinas que puedan ejecutar la maquina virtual de Java con salida por pantalla.
- *La capa Negocio*, proporciona la funcionalidad y ha sido a su vez dividida en paquetes atendiendo a la naturaleza de las operaciones que implementa.
- *La capa Integración*, garantiza la persistencia de los datos. En este caso se ha usado como soporte para los datos de la aplicación ficheros de texto, que son fácilmente editables y por ello nos han permitido una gran fluidez a la hora de realizar las distintas pruebas de funcionamiento.

La estructura básica de la aplicación vista mediante un diagrama de clases en el cual se han relajado en gran medida el formalismo y detalle queda de la siguiente forma:

Diagrama de clases



2.2.2. Patrones de diseño

Inherentemente a la arquitectura Multicapa existen una serie de patrones a implementar, como son;

El patrón Transferencia (en adelante Transfer) usado en la capa de Negocio, cuyo interés es el de encapsular objetos de datos para que pasen de una capa a otra, se crean mediante una factoría abstracta en la que se engloban todos los posibles transfers que se pueden crear y ésta es extendida por una factoría para conformar los objetos transfers en función de la información que en base a cada paquete nos ha parecido razonable enviar.

El Servicio de Aplicación (o Application Service) usado también en la capa de negocio que nos sirve para centralizar la lógica de la aplicación y reduce el código duplicado al encontrarse los métodos en un mismo sitio. Las clases que lo componen son Clientes.java, Coches.java, Pedidos.java Proveedores.java y Ventas.java. En cada

una de ellas se implementan los métodos necesarios para la funcionalidad de la aplicación y son llamadas a través de sus respectivos interfaces.

Dentro del Application Service nos pareció adecuado insertar el patrón Fachada, que queda implementado en la clase ModeloHOTL, establece el vínculo entre la capa de Negocio y la capa Presentación y proporciona una interfaz unificada para poder comunicarse con el resto de las interfaces de las Application Service. La utilidad de éste patrón se hace patente a la hora de codificar el proyecto, puesto que le da una mayor modularidad al sistema con lo que resulta más sencillo de repartir y facilita la localización de posibles errores.

El patrón Data Access Object (en adelante DAO) en la capa de Integración cuyo objetivo es el de separar el acceso a datos con la manipulación de los mismos. La capa de Integración está exclusivamente formada por Daos, el relativo a los clientes (DaoClientes.java), los referentes a coches (DaoCoches.java DaoMarcas.java), el relativo a los pedidos (DaoPedidos.java), DaoProveedores.java y DaoVinculacion.java para los proveedores y por último el DaoFacturas.java que se encarga de las factura. Éstos tienen también sus respectivas interfaces para realizar las llamadas a las operaciones básicas de los Daos: create, update, read, readAll así como el almacenaje y recuperación de números de id en los transfer que así lo necesiten.

Además de éstos patrones hemos usado otros disponibles en el libro “Core J2EE patterns” con el objeto de hacer nuestra aplicación más sólida, estructurada, inteligible y escalable. El más importante de los patrones (si es que puede establecer un orden de importancia) por ser el que organiza la estructura de la capa de presentación de la aplicación que es el Modelo Vista Controlador. Nos hemos decantado por una versión pasiva de éste, en la que no se pueden realizar llamadas desde la vista a capas inferiores, puesto que nos hace más independiente la capa de presentación (la vista. Con respecto al controlador la única peculiaridad es que hemos implementado uno sólo que controla tanto los eventos asociados a la interfaz como los de negocio agrupándolos según su naturaleza mediante un sistema simple de numeración de eventos por rangos.

Cabe destacar que se han usado diferentes estrategias en distintas zonas de la arquitectura de la aplicación, para así poder probar un mayor rango de patrones.

Para el controlador hemos usado el patrón Singleton así como en la capa Presentación, con lo que no solo garantizamos que únicamente existe una instancia de la clase singleton que invoquemos sino que además garantizamos el acceso a ellos desde otras clases evitando sobre escrituras de datos y obtención de datos desfasados.

En los Application Services sin embargo así como en los Daos hemos usado Factorías Abstracta.

2.2.3. Reusabilidad y modularidad de código

Existen una serie de principios fundamentales para comprender cómo se modeliza la realidad al crear un programa bajo el paradigma de la orientación a objetos de entre los que cabe destacar el principio de modularidad.

Mediante la modularidad se propone al programador dividir su aplicación en varios módulos diferentes (ya sea en forma de clases, paquetes o bibliotecas), cada uno de ellos con un sentido propio.

Esta fragmentación disminuye el grado de dificultad del problema al que da respuesta el programa, pues se afronta el problema como un conjunto de problemas de menor dificultad, además de facilitar la comprensión del programa.

Estos módulos que se puedan compilar por separado, pero que tienen conexiones con otros módulos. Al igual que la encapsulación, los lenguajes soportan la Modularidad de diversas formas.

En base a este principio hemos desarrollado la aplicación utilizando una programación estructurada que facilite en todo momento el desarrollo modular de esta misma. Vease por ejemplo la aplicación de la tecnología multicapa que nos permite independizar la interfaz de la lógica y de la capa de persistencia de datos.

2.3. Algoritmos Específicos utilizados

2.3.1. ModelElim y CountElim

Introducción:

En la mayoría de los casos la aplicación de una regla no modifica en sí los modelos del conjunto de modelos sino que genera igualdades en cada uno de ellos respetando las características de cada regla, de forma que la próxima vez que se aplique la regla de “Equality” se comprueben estas igualdades y se eliminen los estados indicados. Estas reglas junto con las que realizan una eliminación directa de estados llevan a cabo esta acción mediante el algoritmo de “ModelElim”.

La acción de “modelElim (m, s1, s2)” elimina del modelo m el estado s2 copiando toda su información en el estado s1 debido a que se ha deducido que los estados s1 y s2 son iguales.

En concreto este algoritmo se lleva a cabo en dos fases, la primera en la que modificamos los accountings del modelo, y la segunda en la que modificamos el resto de componentes del modelo.

Cabe resaltar que en ocasiones la eliminación de un estado del modelo y transferencia de sus responsabilidades a otro estado podría producir como resultado un modelo inconsistente con lo que se sepa de él hasta el momento, como por ejemplo, al obtener un estado que sabíamos que era determinista con dos transiciones de salida con la misma entrada y estados de llegada distintos.

Primera parte (CountElim (a, s1, s2)):

Esta función muestra como los registros de accountings del modelo se actualizan. Básicamente movemos toda la información de los accountings cuyo estado sea s2 a s1.

Para realizar esta función el nuevo conjunto de accountings se construye uniendo dos conjuntos.

El primero de ellos, encerrado en la imagen bajo el primer conjunto de corchetes, representa los accountings cuyos estados sean distintos a s1 y s2. El segundo conjunto representa al resto de accountings

A continuación mostramos formalmente la definición de esta función:

$$\texttt{countElim}(\mathcal{A}, s_1, s_2) =$$

$$\left\{ (s, i, outs, f', n) \;\middle|\; \begin{array}{l} s \notin \{s_1, s_2\} \wedge (s, i, outs, f, n) \in \mathcal{A} \wedge \\ f'(s \xrightarrow{i/o} s') = \begin{cases} f(s \xrightarrow{i/o} s') & \texttt{if } s' \neq s_1, s_2 \\ f(s \xrightarrow{i/o} s_1) + f(s \xrightarrow{i/o} s_2) & \texttt{if } s' = s_1 \\ 0 & \texttt{if } s' = s_2 \end{cases} \end{array} \right\}$$

$$\cup$$

$$\left\{ \begin{pmatrix} s_1, i, \\ outs_1 \cup outs_2, \\ f', n \end{pmatrix} \;\middle|\; \begin{array}{l} \exists\, p, q, g, h : \\ \quad ((s_1, i, outs_1, g, p) \in \mathcal{A} \vee (s_2, i, outs_2, h, q) \in \mathcal{A}) \wedge \\ n = \sum\{m \mid (s, i, outs, f, m) \in \mathcal{A}, s \in \{s_1, s_2\}\} \wedge \\ f'(t) = \sum\{f(t) | (s, i, outs, f, m) \in \mathcal{A}, s \in \{s_1, s_2\}\} \end{array} \right\}$$

<u>Implementación de CountElim:</u>

Para la implementación del algoritmo es necesario recorrer todos los registros de accountings, y para cada uno de ellos hacer tres posibles tratamientos:

- En caso de que el estado del accounting sea distinto de s1 y s2 es necesario observar las transiciones que parten del estado del accounting y fijarnos en los casos en que el estado de llegada sea s1 o s2. En caso de que sea s1 deberemos asegurarnos de que transmitimos la información del registro, y si es s2 deberemos crear una nueva transición si no existía del estado actual a s1 y eliminar la transición que llegaba a s2.
- Si el estado del registro es s2 entonces buscamos el accounting con estado s1 y entrada igual que el accounting que estamos tratando y trabajamos con él. SI no existía le cambiamos el nombre al de s2. Para cada transición referente al registro del accounting modificamos la transición para que salga de s1, si no existía ya. Si la transición existía actualizamos las transiciones pertinentes del accounting de s1, además de borrar la que no hemos modificado. Si al final modificamos creamos una transición nueva
- En caso de que el estado sea s1 recorreremos todas las transiciones del registro y si obtenemos alguna cuyo estado de llegada sea s2 intentaremos generar una transición nueva si no existía previamente. En caso de existir simplemente actualizaremos el número de veces que ha sido observada sumando las que ya tenía con las de la transición que estábamos tratando cuya llegada era s2.

<u>Segunda parte (ModelElim (m, s1, s2)):</u>

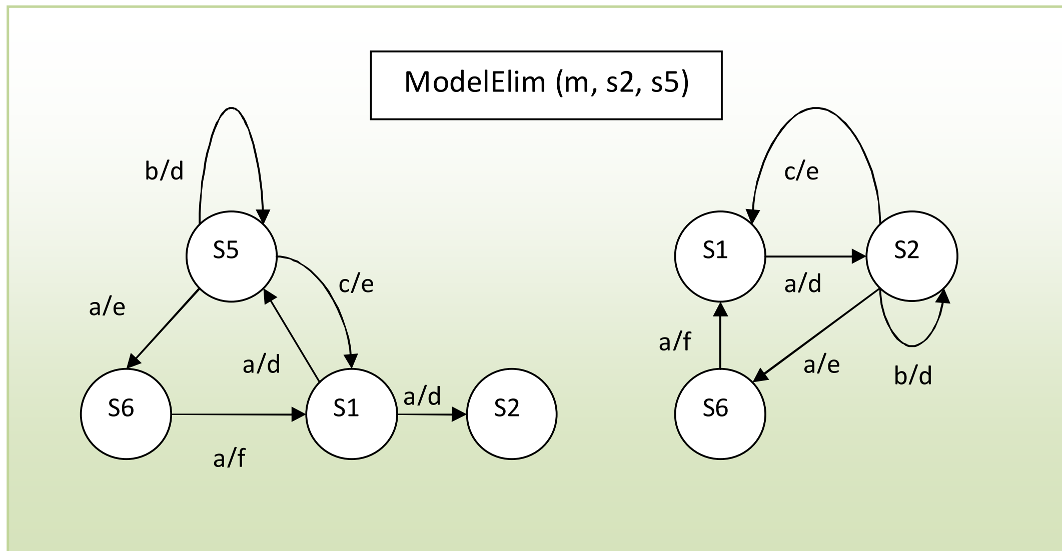
La función previa es una función es usada por “ModelElim” para poder eliminar un estado s que hemos descubierto que es igual a otro dentro de un modelo determinado. Como hemos dicho antes transferiremos todas las responsabilidades de un estado a otro.

De la aplicación de esta función podemos obtener un modelo inconsistente por lo que se devolverá un modelo vacío, indicado en el apartado (a) de la imagen. El nuevo modelo que obtenemos es el resultado de sustituir todas las ocurrencias del estado a eliminar por el estado que se mantendrá, indicado por el apartado (b) en la imagen posterior.

(a) If there exist $i, outs, f, o, s$, and $j \in \{1,2\}$ such that $s_{3-j} \xrightarrow{i/o} s \in \mathcal{T}$, $(s_j, i, outs, f, \top) \in \mathcal{A}$, and $o \notin outs$, then $\mathcal{M} = \emptyset$.

(b) Otherwise, $\mathcal{M} = \left\{ \left(\begin{array}{c} \mathcal{S}\backslash\{s_2\}, \mathcal{T}[s_2/s_1], \mathcal{I}[s_2/s_1], \\ \texttt{countElim}(\mathcal{A}, s_1, s_2), \mathcal{E}[s_2/s_1], \\ \mathcal{D}[s_2/s_1], \mathcal{O} \end{array} \right) \right\}$

A continuación mostramos un pequeño ejemplo de la aplicación de “ModelElim” sobre un modelo:



2.3.2. AllCorrect y Conformance

Introducción:

Una vez aplicadas todas las reglas posibles hasta no poder aplicar ninguna más (En el caso de ejecución por pasos) o en el momento que el usuario decida (En el caso de ejecución manual) es necesario conocer si el conjunto de modelos es correcto con respecto a la especificación, es decir, si todos los modelos generados por la sucesiva aplicación de las reglas son consistentes y conformes a la especificación. En éste caso podremos decir que nuestra máquina implementada, con sus observaciones y las hipótesis tomadas es correcta con respecto a la especificación que se deseaba implementar.

Señalar que en el caso de ejecución por pasos, al seguir la ejecución de las reglas un orden establecido que asegura la terminación, cuando aplicamos la regla "All Correct" tras haber aplicado todas las correspondientes reglas, el resultado obtenido (correcto o incorrecto) es definitivo. Sin embargo en la ejecución manual, la regla "All Correct" puede aplicarse cuando el usuario decida, dando un resultado, por ejemplo incorrecto, que de seguir aplicando más reglas podría cambiar. Por lo tanto la llamada a "All Correct" debería hacerse cuando no se pueden aplicar ya más reglas.

Implementación:

Como hemos dicho antes, "All Correct" se basa en mirar si todos los modelos son consistentes y conformes a la especificación. La implementación del algoritmo para ver si un modelo es consistente fue sencilla siguiendo el artículo de "HOTL". Para la implementación de "All Correct" tuvimos que investigar en diversos artículos para encontrar un algoritmo "conform" que nos diera los mejores resultados tanto en tiempo de ejecución como en uso de memoria.

Debido a que nuestras máquinas de estados pueden ser no deterministas, nuestra primera intención era evitar convertirlas en deterministas, debido a que al convertir un autómata no determinista en uno determinista el número de estados generados para el nuevo autómata puede ser 2^n, siendo n el número de estados del autómata no determinista. Para ello intentamos una modificación de algoritmo de minimización de autómatas deterministas, de forma que sirviera para autómatas no deterministas, tras buscar diversas formas de hacerlo y estudiar con el tutor del proyecto la forma de llevarlo a cabo desestimamos este método.

Finalmente decidimos utilizar un algoritmo que recibiendo el modelo y la especificación realizaba los siguientes 4 pasos:

1. Trasformación del modelo y la especificación en máquinas con un solo estado inicial.
2. Transformación de la especificación y el modelo en máquinas deterministas.
3. Construcción de la máquina-p con el modelo y la especificación trasformadas.
4. Búsqueda de algún camino en la máquina-p desde el estado inicial a error.

Trasformación del modelo y la especificación en máquinas con un solo estado inicial:

El primer paso del algoritmo es reducir a un solo estado inicial tanto la especificación como el modelo que queremos ver si es conforme o no.

Para ello usamos el siguiente algoritmo:

Entrada: Fsm MN = (Q, E, S, δ, Q_0)

Salida: Fsm MT = (Q', E', S', δ', Q'_0) tal que L(MT) = L(MN)

Siendo:

- *Q, Q' -> Conjuntos de estados*
- *E, E' -> Conjuntos de entradas*
- *S. S' -> Conjuntos de salidas*
- *δ, δ' -> Conjuntos de transiciones*
- *Q_0, Q'_0 -> Conjuntos de transiciones*

```
MT = MN;
inicial = q0;
for all δi ∈ δ'
        for all qj ∈ Q'0
                if qj== δi.salida then
                        δi.salida = inicial;
                end if;
                if qj == δi.llegada then
                        δi.llegada = inicial;
                end if;
        end for;
end for;
for all qj ∈ Q'0
```

remove(Q', q_j);

remove(Q'_0,q_j);

end for;

añadir(Q', incial);

añadir(Q'_0, inicial);

Transformación de la especificación y el modelo en máquinas deterministas:

El siguiente paso es trasformar tanto la especificación como el modelo en máquinas deterministas, ya que el algoritmo de la máquina-p solo es aplicable a máquinas deterministas.

Para ello usamos el siguiente algoritmo, que dada una máquina no determinista, devuelve una máquina determinista equivalente:

Entrada: Un AFND MN (Q, E, S, δ, q_0)

Salida: Un AFD MD = (Q', E', S', δ', q'_0) tal que L(MT) = L(MN)

E' = E;

S' = S;

$Q' = \{\{q_0\}\}$;

$q'_0 = \{q_0\}$;

$\delta' = \{\}$;

for all $a_i \in E$

for $s_i \in S$

$\delta'(q_0,a_i,s_i) = \delta(q_0,a_i,s_i)$;

añadir(Q', $\delta'(q_0,a_i,s_i)$); //Solo se añade si es nuevo

end for

marcar (Q' , q'_0)); // para saber que se han calculado sus transiciones

while haya estados no marcados en Q'

sea Sj un estado no marcado de Q'

for all $a_i \in E$

for $s_i \in S$

$\delta'(S_j,a_i,s_i) = \cup_{q \in sj} (\delta(q,a_i,s_i))$;

añadir(Q', $\delta(S_j,a_i,s_i)$); //Solo se añade si es nuevo

añadir(δ, (S_j, a_i, s_i, $\delta(S_j,a_i,s_i)$))

end for

end for

marcar (Q´ , S_j);

end while

Construcción de la máquina-p con el modelo y la especificación trasformadas:

El siguiente paso es construir la máquina-p a partir de la especificación y el modelo trasformados siguiendo los dos pasos anteriores.

La máquina-p es una nueva máquina de estados determinista construida a partir de otras dos máquinas de estados, donde:

- Sus estados son el producto cartesiano de los estados de estas dos, siendo el primer estado del par perteneciente a la especificación y el segundo al modelo, más un estado de error. Por decirlo de alguna manera, cada par de estados significa estados que deben ser equivalentes entre la especificación y el modelo. Los estados los iremos construyendo bajo demanda, es decir, dado un estado de la máquina-p analizaremos sus transiciones y en el caso de necesitar la creación de un nuevo estado lo haremos, no crearemos todos desde el principio ya que muchos de ellos serán estados sueltos hacia los que no irá ninguna transición y por lo tanto inútiles.
- Sus salidas son las salidas de las dos máquinas.
- Sus entradas son las entradas de las dos máquinas.
- Sus transiciones serán: Dado un estado (s1,s1') de la máquina-p y la transición (s1,a,s,s2) perteneciente a la especificación. Para todas la transiciones (s1',a',s',s2') pertenecientes al modelo.

    - Si a = a' y s = s', creamos una nueva transición para la máquina-p de la siguiente forma ((s1,s1'),a,s,(s2,s2')) debido a que el modelo realiza lo que dice la especificación para ese estado.

    - Si a = a' y s /= s' y para todas las transiciones del modelo no se ha encontrado ninguna q cumpla el anterior punto, creamos una nueva transición para la máquina-p de la siguiente forma ((s1,s1'),a,s,error) debido a que el modelo no realiza lo que dice la especificación para ese estado, puesto que la salida de la transición es diferente en el modelo que en la especificación.

- En el caso de que a /= a' y s /= s' para todas las transiciones del modelo creamos una nueva transición para la máquina-p de la siguiente forma ((s1,s1'),a,s,error) debido a que el modelo no realiza lo que dice la especificación para ese estado, ya que no existe en el modelo la transición que existe en la especificación.

Teniendo en cuenta todo lo anterior, el algoritmo para la construcción de la máquina-p sería el siguiente:

Entrada: Fsm spec (Q, E, S, δ, q_0); Fsm modelo (Q', E, S, δ', q'_0);

Salida: máquina-p M = (Q_p, E_p, S_p, δ_p, q_{0p})

E_p = E;

S_p = S;

inicial = (q_0, q'_0);

q_{0p} = inicial;

añade (Q_p, q_{0p});

añade (Q_p,error);

while haya estados no marcados en Q_p

sea (S,M) un estado no marcado de Q_p;

sea tranSpec las transiciones en la especificación cuyo estado de salida es S;

sea tranModelo las transiciones en el modelo cuyo estado de salida es M;

boolean encontrada;

for all $a_i \in$ tranSpec

encontrada = false;

for $s_i \in$ tranModelo

if(a_i.input == s_i.input && (a_i.output == s_i.output) then

encontrada = true;

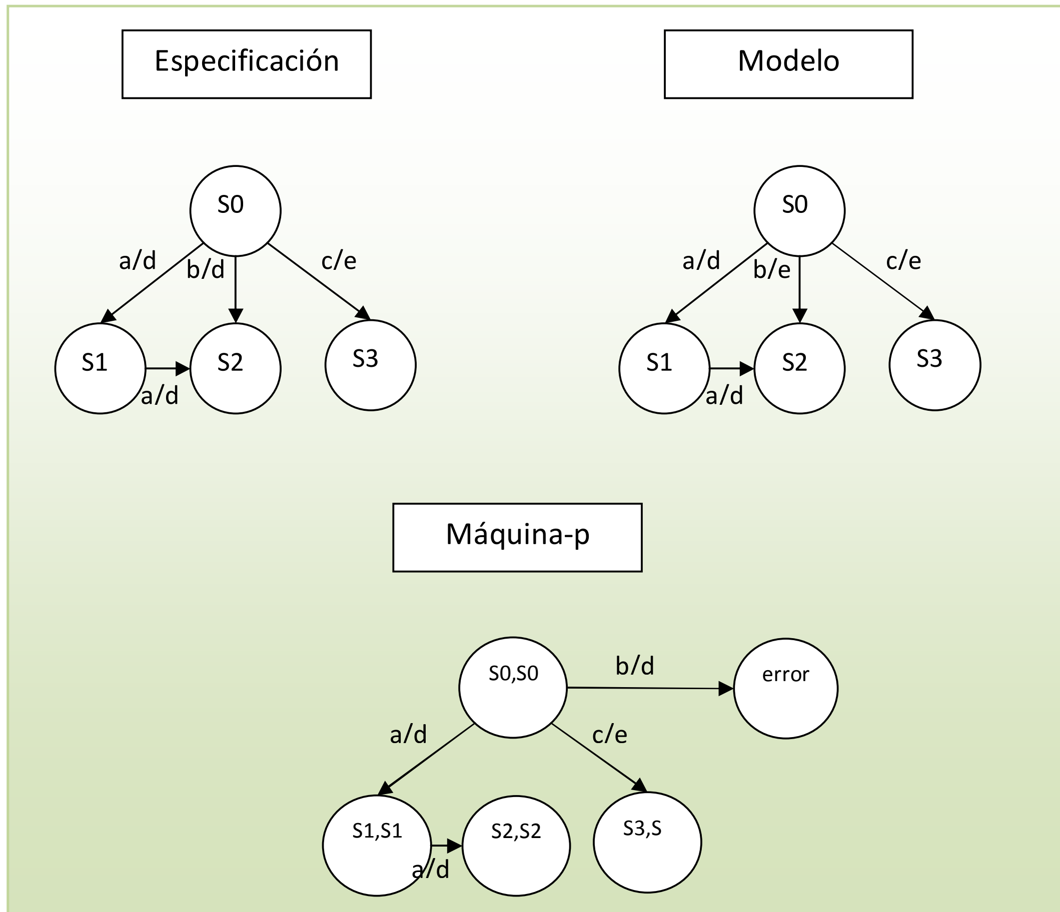
añade(Q_p, (a_i.llegada, a_j.llegada)); //Solo lo añade si es nuevo

añade(δ_p, ((S,M), a_i.input, a_i.output,(a_i.llegada, a_j.llegada));

```
                    else if(ai.input == si.input && (ai.output != si.output) then
          Y                   encontrada = true;
                              añade(δp, ((S,M), ai.input, ai.output,error);
                    end if
          end for
          if (!encontrada) then
                    añade(δp, ((S,M), ai.input, ai.output,error);
          end if
     end for
     marcar (Qp, (S,M)); // Ya han sido tratadas sus transiciones
end while
```

Una vez tenemos nuestra máquina-p, en el caso de haber un camino desde el estado inicial hasta estado de error el modelo no es conforme a la especificación y sí lo es en caso contrario.

Veamos un ejemplo de construcción de una máquina-p dada una especificación y un modelo:



Para este sencillo ejemplo se puede ver claramente como el modelo no es conforme a la especificación. La especificación requiere que estando en el estado S0, cuando entre una b, la salida sea d y se pase al estado S2. Sin embargo en el modelo entra una b en el estado S0 (estado equivalente al S0 de la especificación) y sale una e y únicamente existe esta transición, por lo tanto no cumple lo que la especificación requiere, creándose una transición al estado de error. Una vez creada la máquina-p se observa que existe un camino desde el estado inicial a error, por lo tanto no es conforme.

2.3.3. Subconjuntos de un vector

Introducción:

Este algoritmo está usado en la implementación de la regla número 12 (AllTranHappen) en la que se asumía que si una transición se había observado un número determinado de veces podíamos asumir que su comportamiento es cerrado, y por lo tanto no responderá de otra forma a entradas futuras más que de la forma en la que ya lo había hecho hasta el momento.

Recordamos que para hacer esto es necesario comprobar todas las formas posibles de agrupar las transiciones de un registro accounting, lo que en definitiva se traduce en obtener todos los subconjuntos posibles de un conjunto dado de elementos.

Implementación:

El algoritmo es el resultante:

```
Boolean [] subset = new boolean [tam];
while (true) {
        // Realizar acción
        // Add 1
        int cont=0;
        do {
                subset [cont] =!subset[cont];
                ++cont;
        } while (cont<tam && !subset [cont-1]);
        If (cont>=tam && !subset [cont-1]) break;
}
```

2.3.4. Mapeado

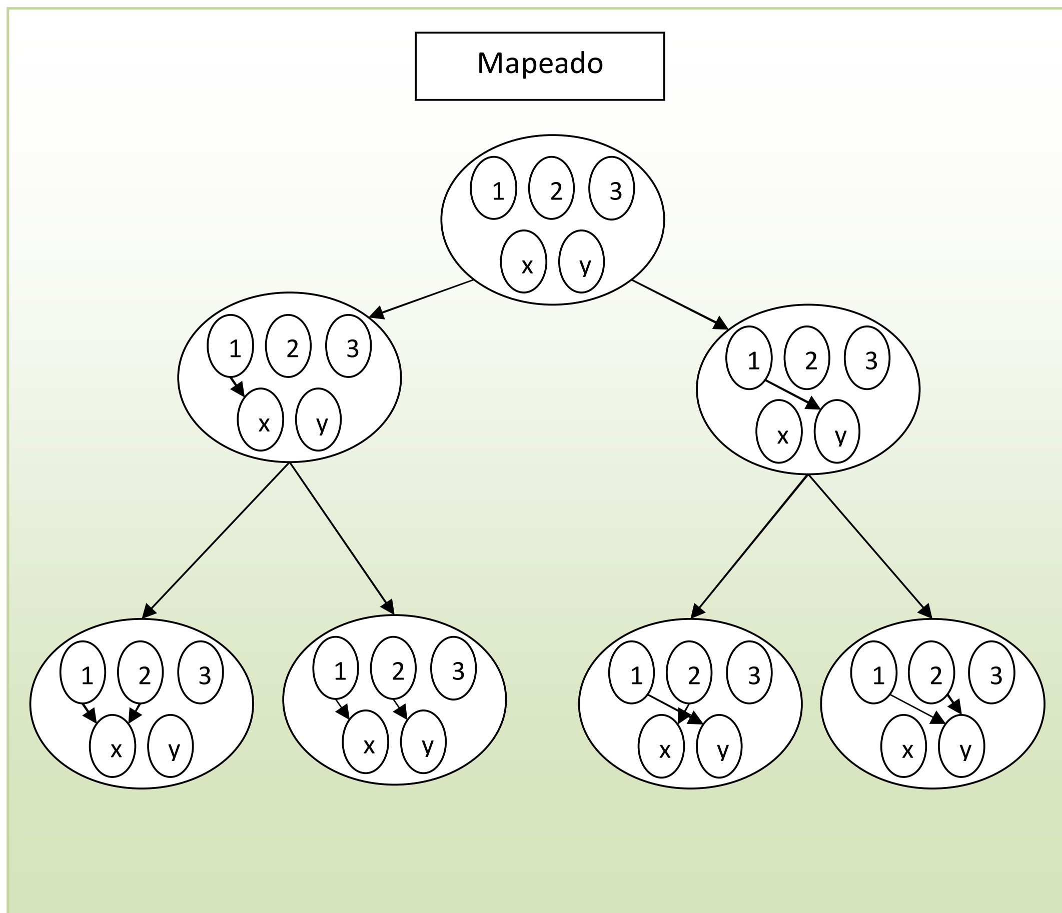
Introducción:

El mapeado es un algoritmo necesario para la implementación de las reglas de “AllTranHappen”, “Long” y “Upper”.

Dados un conjunto de elementos *a* y otro *b* el mapeado consiste en emparejar de todas las formas posibles cada elemento de a con cada elemento de b.

Implementación:

Se genera una estructura en forma de árbol en el cual cada nodo representa el estado de las uniones. El nivel iesimo representa todas las posibles formas de unir el elemento iesimo de a con cada elemento de b de forma que el número de niveles del árbol viene dado por los elementos de a, y el número de hijos de cada nodo vendrá dado por el número de elementos de b.



2.4. Persistencia de datos

La persistencia de datos de la aplicación de HOTL tiene lugar, como se ha indicado en el apartado 2.2.1. (Arquitectura de HOTL) en la capa de integración. Es en esta capa donde se almacenan y obtienen los datos que se desea que sean persistentes para futuros usos de la aplicación.

Diremos que la capa de integración es una capa intermedia en nuestra aplicación que ofrece servicios de persistencia y recuperación de información a las capas superiores.

En particular hemos optado por la utilización de ficheros de texto para llevar a cabo esta persistencia debido principalmente a la gran facilidad de su edición. Esta característica nos resulta de gran importancia ya que para realizar las comprobaciones del funcionamiento de la herramienta es necesario contar con una gran batería de pruebas que contemplen todos los casos posibles, y es aquí donde otros formatos como XML no pueden ser editados de forma tan directa.

En el siguiente apartado de manual de usuario se añade una descripción detallada del formato de los ficheros de texto para cada tipo fundamental de HOTL, es decir, para las FSM, observaciones, hipótesis y modelos.

3. APLICACIÓN FINAL

3.1. Comprobación de objetivos y resultados obtenidos

Como hemos indicado antes nuestro objetivo principal fue implementar una herramienta que fuera capaz dadas una hipótesis, unas observaciones y una especificación, llegar a un diagnóstico que dijera si la máquina sobre la que se han realizado las pruebas es conforme a la especificación o no utilizando las reglas de la lógica HOTL. Dado el carácter didáctico de nuestra herramienta otro objetivo fue ir mostrando paso a paso como van evolucionando los modelos tras la aplicación de las distintas reglas, mostrándolos por pantalla y no limitándonos sólo a dar un diagnóstico. Desde un principio se pensó también en dotar a la herramienta de una interfaz gráfica agradable, pero dejando esto en un segundo plano y empezando con ella una vez que la lógica del programa estuviera completa. Dichos objetivos han sido cumplidos en su totalidad.

A medida que el proyecto avanzaba se fueron proponiendo más ideas para añadir al programa, como por ejemplo que el orden de la ejecución de las reglas no se basase sólo en el algoritmo que garantiza terminación, sino que también pudiesen ser aplicadas en el orden que el usuario quisiese, dotando a la herramienta de la posibilidad de ejecutar las reglas manualmente, mejoras en la interfaz gráfica como mostrar en cada paso los modelos antiguos y los nuevos modelos que han surgido a partir de ellos tras la aplicación de una regla, etc. Estos objetivos fueron igualmente cumplidos.

Según nos fuimos acercando a la fecha de entrega fueron surgiendo más ideas para añadir, sobre todo por la intención de los profesores de usar la herramienta en alguna de sus conferencias acerca de HOTL, ideas como por ejemplo la posibilidad de introducir nuevas observaciones e hipótesis en mitad de una ejecución manual, mejoras en la interfaz gráfica para hacerla más amena, intuitiva y profesional. Lamentablemente algunas estas nuevas ideas no pudieron llevarse a cabo debido a la falta de tiempo, pero otras muchas sí, mejorando la herramienta.

Por lo tanto a nivel global podría decirse que la mayoría de objetivos han sido cumplidos, habiéndose cumplidos todos los objetivos iniciales y dejando sin cumplir sólo algunos que surgieron en el tramo final del proyecto, de todos modos debido a la reusabilidad de nuestro código sería fácilmente ampliable en un futuro con las mejoras que quedaron pendientes y con otras que pudieran surgir.

3.2. Manual de usuario

El siguiente texto tratará de acercar y enseñar al lector como manejar la aplicación. Realmente su uso es muy sencillo una vez que se conoce como funciona la lógica y de lo que trata la aplicación, pero para alguien que no conozca la materia puede resultar algo complicado.

Señalar que este manual, es una manual sobre la herramienta que introduce mínimamente la lógica de "HOTL". Para una mejor comprensión de la lógica y uso adecuado de la herramienta sería conveniente la lectura de la investigación sobre la que se ha desarrollado. *HOTL: Hypotheses and observation testing logic* .

Lo primero indicar que no se puede empezar la ejecución de las reglas sin haber cargado antes la especificación, las hipótesis y las observaciones.

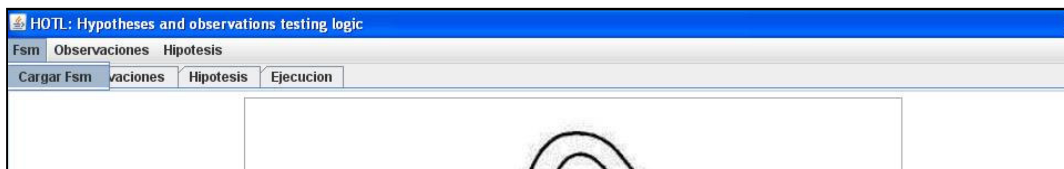
Veremos ahora como poder hacer las anteriores cosas, además de cómo poder trabajar con la herramienta una vez cargados los datos.

Carga de la especificación:

Lo primero que necesita nuestra aplicación es la especificación con la cual se va a trabajar. La especificación se carga en el programa a través de un archivo de texto el cual debe seguir un formato específico.

Para cargar la FSM se accederá al menú "Fsm" y se seleccionará la opción "Cargar Fsm", una vez cargada, la especificación aparecerá en la pestaña con nombre "Fsm".

Menú para cargar la especificación



Como bien sabemos Una FSM es una tupla de 5 elementos F = (S, inputs, outputs, I, T) donde:

- S es un conjunto de estados
- inputs es un conjunto de entradas activas

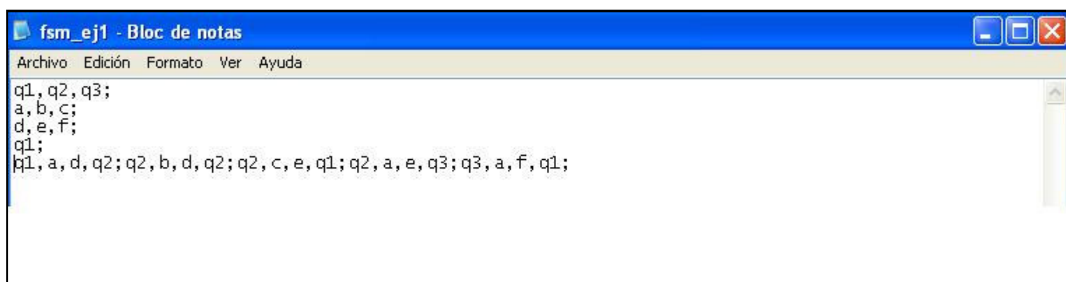
- outputs un conjunto de salidas activas
- I es un subconjunto de S y es el conjunto de estados iniciales
- T es el conjunto de transiciones.

El formato del archivo de texto para la FSM de la especificación debe ser el siguiente:

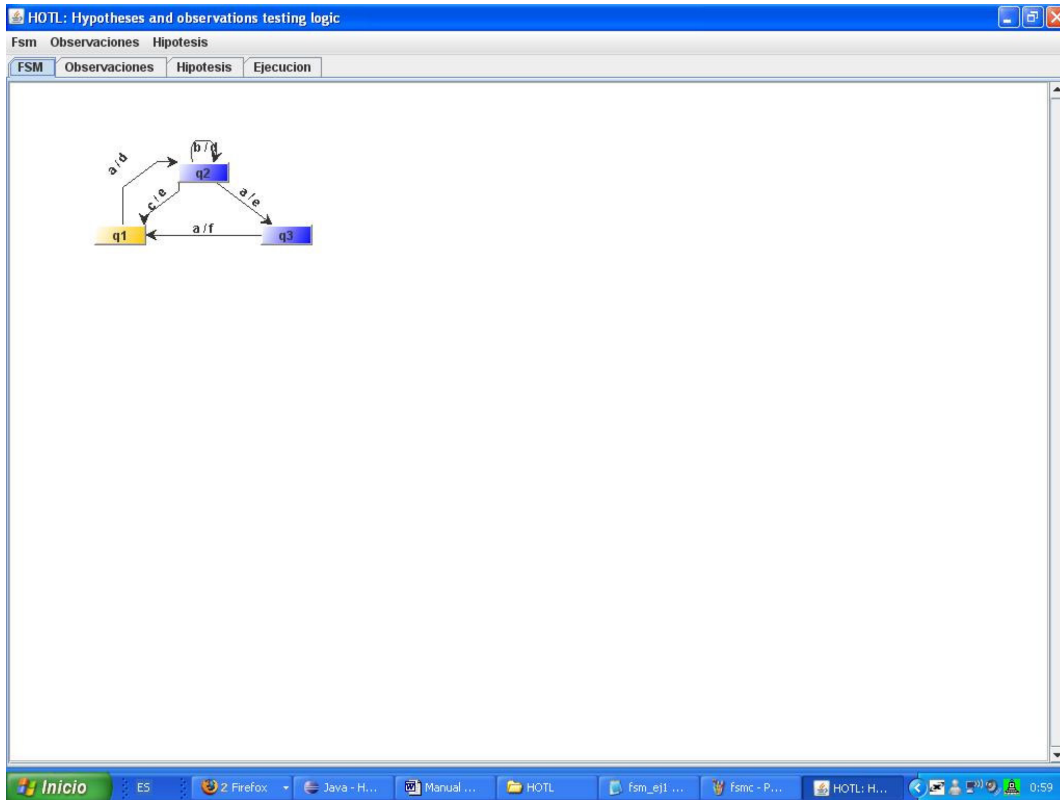
- En la primera línea debe ir el conjunto de estados, separados por "," cada uno de ellos y terminados en ";".
- En la segunda línea debe ir el conjunto de entradas, separadas por "," cada una de ellas y terminadas en ";".
- En la tercera línea debe ir el conjunto de salidas, separadas por "," cada una de ellas y terminadas en ";".
- En la cuarta línea debe ir el conjunto de estados iniciales, separados por "," cada uno de ellos y terminadas en ";".
- En la quinta línea debe ir el conjunto de transiciones. Cada transición irá separada por ";" y el formato de cada transición será estado de salida, input, output y estado de llegada separados cada elemento por ",".

Un ejemplo de un archivo de texto que representa la FSM:

- S = {q1,q2,q3}
- Inputs = {a,b,c}
- Outputs = {d,e,f}
- I = {q1}
- T = {{q1,a,d,q2}, {q2,b,d,q2}, {q2,c,e,q1}, {q2,a,e,q3}, {q3,a,f,q1}}



Fsm cargada visible en la pestaña "FSM"

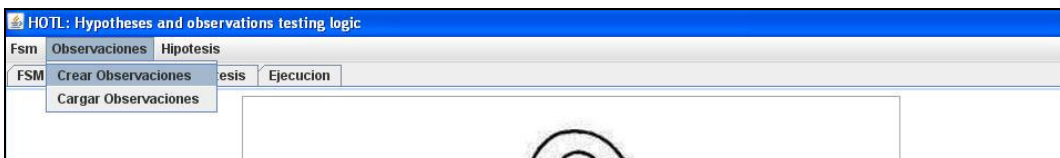


Carga de las observaciones:

Las observaciones se cargan en el programa, al igual que la especificación a través de un archivo de texto el cual debe seguir un formato específico.

Para cargar las observaciones se accederá al menú "Observaciones" y se seleccionará la opción "Cargar Observaciones", una vez cargadas, las observaciones aparecerán en la pestaña con nombre "Observaciones".

Menú para cargar la especificación:



Las observaciones tendrán la siguiente forma: ob = $a_1,i_1/o_1,a_2,\ldots,a_n,i_n/o_n,a_{n+1}$, Siendo:

- a_i los posibles atributos de cada estados, que puede ser uno de los siguientes:
  - Det: Asumimos el determinismo del estado.
  - Imp(q) : Asumimos que denota que el estado referenciado está asociado al estado *q* de la especificación. q debe ser un estado de la especificación.
  - Spec(*q*): Solo aplicable al último estado de la observación e indica que el estado en el que nos encontramos es tal que el subgrafo que puede ser alcanzado desde el es isomorfo al subgrafo que puede ser alcanzado desde el estado *q* de la especificación, es decir, el subgrafo de la especificación que se obtiene desde el estado *s* se copia y pega al estado de la observación en el que nos encontramos. q debe ser un estado de la especificación.
  - 0: Denota el conjunto vacío de atributos.
- i_i la secuencia de entradas.
- o_i las secuencias de salida.

El formato del archivo de texto para las observaciones debe ser el siguiente:

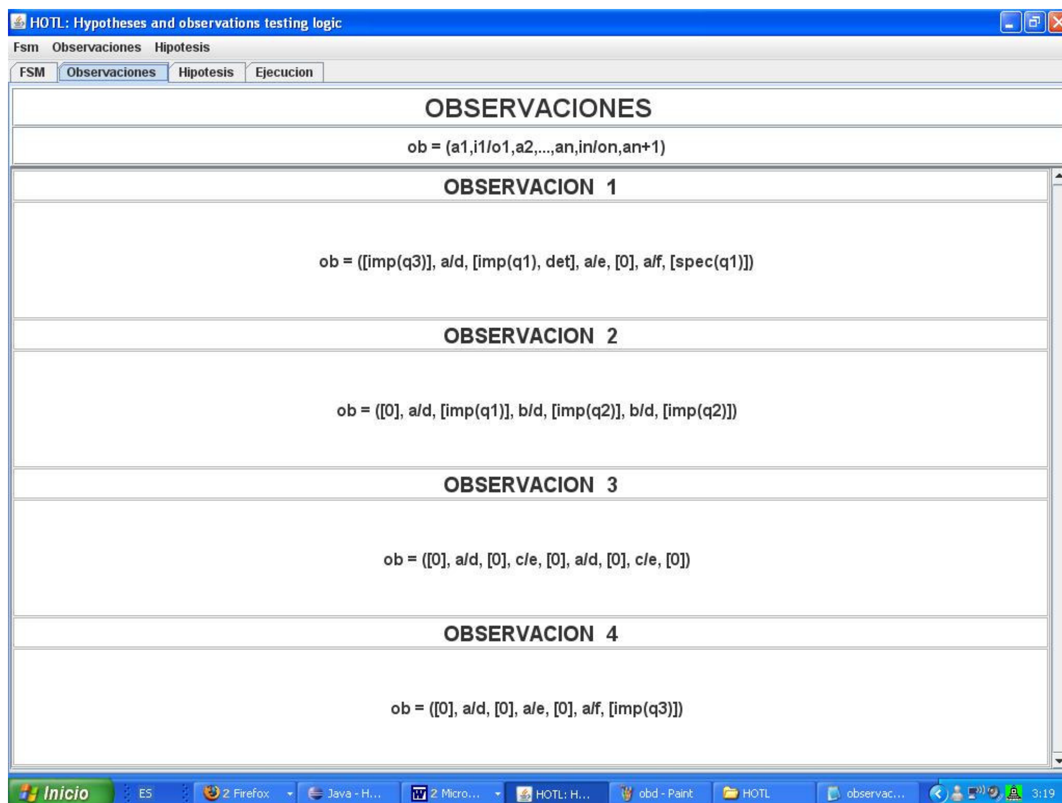
- Cada observación ocupará tres líneas, separando cada observación por ":"
- En la primera línea de cada observación debe ir el conjunto de atributos de cada estado, separado cada conjunto por "," y terminando en ";". Dentro del cada conjunto de atributos de un estado, separaremos estos por "-".
- En la segunda línea de cada observación debe ir el conjunto de entradas separando cada una por "," y terminando en ";".
- En la tercera línea de cada observación debe ir el conjunto de salidas separando cada una por "," y terminando en ":" ya que es el fin de la observación.

Un ejemplo de un archivo de texto que representa las siguientes cuatro observaciones:

- ob = ([imp(q3)], a/d, [imp(q1),det], a/e, [0], a/f, [spec(q1)])
- ([0], a/d, [imp(q1)], b/d, [imp(q2)], b/d, [imp(q1)])
- ([0], a/d, [0], c/e, [0], a/d, [0], c/e, [0])
- ([0], a/d, [0], a/e, [0], a/f, [imp(q3)])

```
observaciones_ej1_1 - Bloc de notas
Archivo  Edición  Formato  Ver  Ayuda
imp(q3),imp(q1)-det,0,spec(q1);
a,a,a;
d,e,f:
0,imp(q1),imp(q2),imp(q2);
a,b,b;
d,d,d:
0,0,0,0,0;
a,c,a,c;
d,e,d,e:
0,0,0,imp(q3);
a,a,a;
d,e,f:
```

Observaciones cargadas visibles en la pestaña “Observaciones”



<u>Carga y creación de las hipótesis:</u>

Las hipótesis se pueden cargar en el programa bien cargándolas desde archivo como en los dos casos anteriores o creándolas a través de la interfaz gráfica.

Para cargar las hipótesis se accederá al menú “Hipótesis” y se seleccionará la opción “Cargar Hipótesis”, una vez cargada, las hipótesis aparecerán en la pestaña con nombre “Hipótesis”.

Menú para cargar y crear las hipótesis

Como bien sabemos las hipótesis pueden ser las siguientes (Consultar el apartado 1.2.3 del manual para más información acerca de las hipótesis) :

- SingleInit
- AllDet
- AllTranHappenWith
- UpperBoundOfStates
- LongSequencesSamePath
- UniqueOrigin
- UniqueDestination

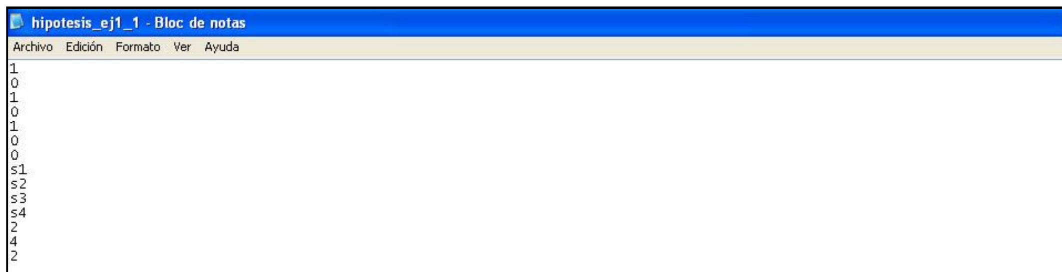
El formato del archivo de texto para las hipótesis debe ser el siguiente:

- Las 7 primeras líneas serán 1 ó 0 dependiendo si tomamos la hipótesis como cierta o no siguiendo el orden de la relación anterior (Primero SingleInit, después AllDet,…).
- La siguiente línea serán las entradas de las hipótesis UniqueOrigin (Ya que puede haber más de una para diferentes entradas y salidas) separadas por "," y terminadas en ";". En el caso de que la hipótesis UniqueOrigin no sea cierta, el contenido de dicha línea es irrelevante.
- La siguiente línea serán las salidas de las hipótesis UniqueOrigin (Ya que puede haber más de una para diferentes entradas y salidas) separadas por "," y terminadas en ";". En el caso de que la hipótesis UniqueOrigin no sea cierta, el contenido de dicha línea es irrelevante.
- La siguiente línea serán las entradas de las hipótesis UniqueDestination (Ya que puede haber más de una para diferentes entradas y salidas) separadas por "," y terminadas en ";". En el caso de que la hipótesis UniqueDestination no sea cierta, el contenido de dicha línea es irrelevante.
- La siguiente línea serán las salidas de las hipótesis UniqueDestination (Ya que puede haber más de una para diferentes entradas y salidas) separadas por "," y terminadas en ";". En el caso de que la hipótesis UniqueDestination no sea cierta, el contenido de dicha línea es irrelevante.

- La siguiente línea será el parámetro n de la hipótesis AllTranHappenWith. En el caso de que la hipótesis AllTranHappenWith no sea cierta, el contenido de dicha línea es irrelevante.
- La siguiente línea será el parámetro n de la hipótesis UpperBoundOfSatates. En el caso de que la hipótesis UpperBoundOfSatates no sea cierta, el contenido de dicha línea es irrelevante.
- La siguiente línea será el parámetro n de la hipótesis LongSequenceSamePath. En el caso de que la hipótesis LongSequenceSamePath no sea cierta, el contenido de dicha línea es irrelevante.

Un ejemplo de un archivo de texto que representa las siguientes hipótesis:

- SingleInit: TRUE
- AllDet: FALSE
- AllTranHappenWith: TRUE, n=2
- UpperBoundOfStates: FALSE
- LongSequencesSamePath: TRUE, n = 2
- UniqueOrigin: FALSE
- UniqueDestination: FALSE



Apuntar que en el ejemplo anterior al ser UpperBoundOfStates, UniqueOrigin y UniqueDestination FALSE el contenido de las líneas 8,9,10,11 y 13 es irrelevante.

Además de poder cargar las hipótesis desde un archivo de texto podemos crearlas a través de la interfaz gráfica. Para ello se accederá al menú “Hipótesis” y se seleccionará la opción “Crear Hipótesis”, una vez cargadas, las hipótesis aparecerán en la pestaña con nombre “Hipótesis”.

Pantalla para la creación de las hipótesis

Para crear la hipótesis se ponen a true aquellas que van a usarse, añadiendo además el parámetro n para el caso de AllTranHappenWith, UpperBoundOfStates y LongSequencesSamePath. Para añadir las hipótesis UniqueOrigin y UniqueDestination se pulsa sobre el botón para dicho fin, se desplegará entonces un panel nuevo donde habrá que introducir la input y la output para dichas hipótesis, pulsando en el botón “aceptar” de dicho panel la hipótesis será añadida al repertorio seleccionado, se pulsará el botón “cancelar” en el caso de querer anular la introducción de la hipótesis.

Una vez se tienen la hipótesis seleccionadas se puede elegir entre pulsar el botón “aceptar” en cuyo caso las hipótesis se cargarán al programa o el botón “guardar” en cuyo caso las hipótesis seleccionadas además de cargarse en el programa, se podrán grabar en un fichero de texto seleccionado por el usuario para su posterior uso si así lo necesitase.

Hipótesis cargadas o creadas visibles en la pestaña "Hipótesis"

<u>Ejecución de las reglas:</u>

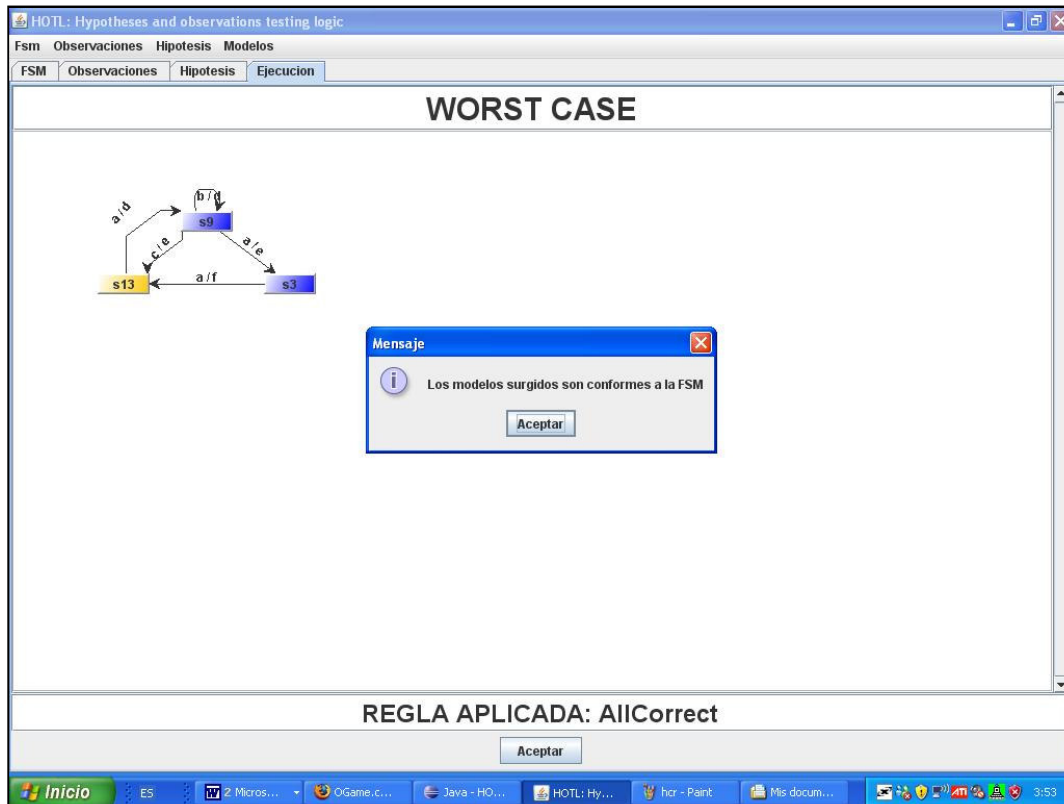
Una vez tenemos la especificación, las observaciones y las hipótesis cargadas en nuestra aplicación podemos comenzar a aplicar las reglas de la lógica "HOTL" para trabajar con dichos datos.

El programa tiene tres modos de ejecución diferentes: "Ejecución total", "Ejecución por pasos" y "Ejecución manual". Para seleccionar cada uno de ellos se accederá al menú "Ejecución" y se seleccionará una de las tres opciones disponibles correspondientes a los tres tipos diferentes de ejecución, los distintos modelos que van surgiendo durante la ejecución de las reglas serán visibles en la pestaña "Ejecución".

Si se selecciona la "Ejecución total" en la parte de abajo de la pestaña "Ejecución" aparecerá un botón etiquetado como "Ejecutar", al pulsar dicho botón se ejecutarán todas las reglas posibles siguiendo el algoritmo de ejecución que garantiza terminación explicado en el (apartado 1.2.7 Maquina de estados). Mostrando finalmente si los modelos surgidos son conformes a la especificación o no. También se

mostrará por pantalla los worst-case de los modelos finales surgidos (apartado 1.2.3. Conceptos y definiciones de HOTL) sobre los que se ha estudiado su conformidad.

Ejemplo de ejecución total:

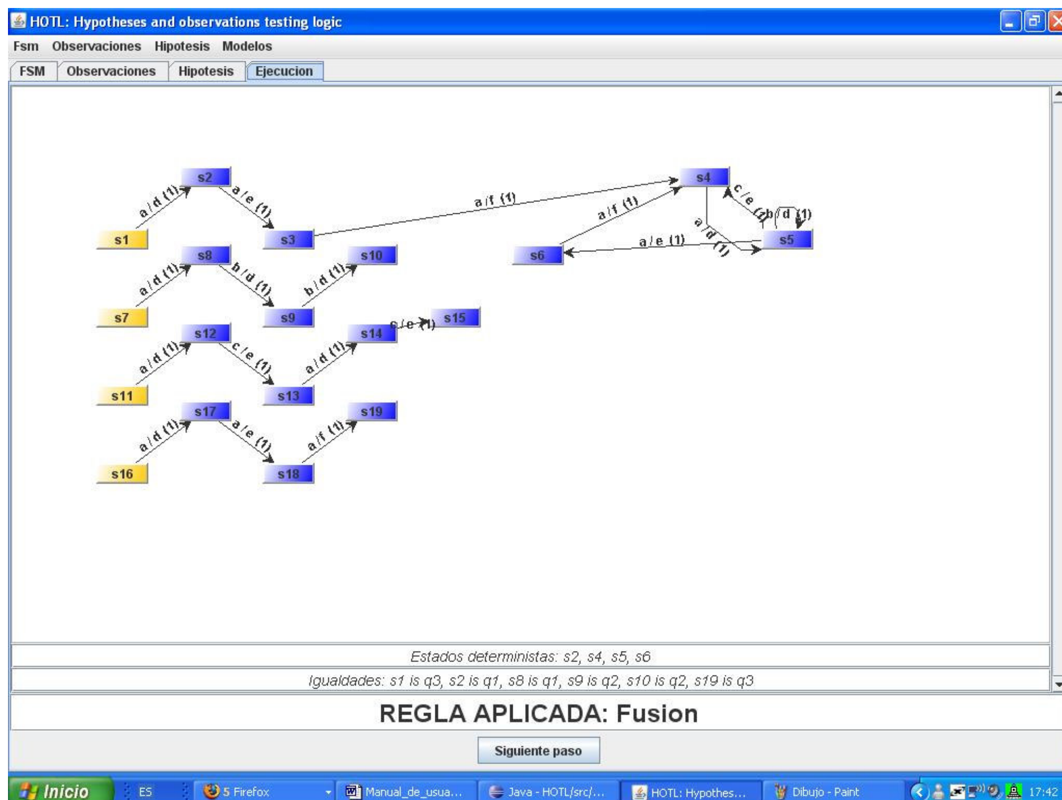


La "Ejecución por pasos" es igual que la "Ejecución total" solo que va mostrando cada uno de los pasos tras la aplicación de alguna regla. En la parte de abajo de la pestaña "Ejecución" aparecerá un botón etiquetado como "Siguiente paso", al pulsar dicho botón se aplicará la siguiente regla siguiendo el algoritmo de ejecución que garantiza terminación. En la pantalla se mostrará la regla aplicada además de los modelos anteriores (old models) y los nuevos modelos (new models) surgidos tras la aplicación de la regla. De cada modelo se mostrará además las veces que ha sido observada cada transición añadiendo al lado de la transición en número entre paréntesis, también se mostrará en la parte de abajo del modelo sus igualdades y los estados deterministas. Además los modelos aparecerán ordenados por grupos, agrupándose de forma que en el mismo grupo aparezcan los modelos antiguos y los nuevos modelos que han surgido de la aplicación de las reglas a dichos modelos, es

decir, si para un modelo antiguo A, tras la aplicación de una regla sobre él, han surgido los modelos B, C y D, los 4 modelos serán agrupados en un mismo grupo.

Mientras sigan pudiéndose aplicar reglas, se podrá seguir pulsando el botón "Siguiente paso" hasta que llegue el momento en que no sea posible la aplicación de más reglas en cuyo caso se mostrará por pantalla si los modelos finales surgidos son conformes a la especificación o no. Al igual que para el modo de ejecución anterior también se mostrará el worst-case de los modelos finales surgidos de los cuales se ha estudiado su conformidad.

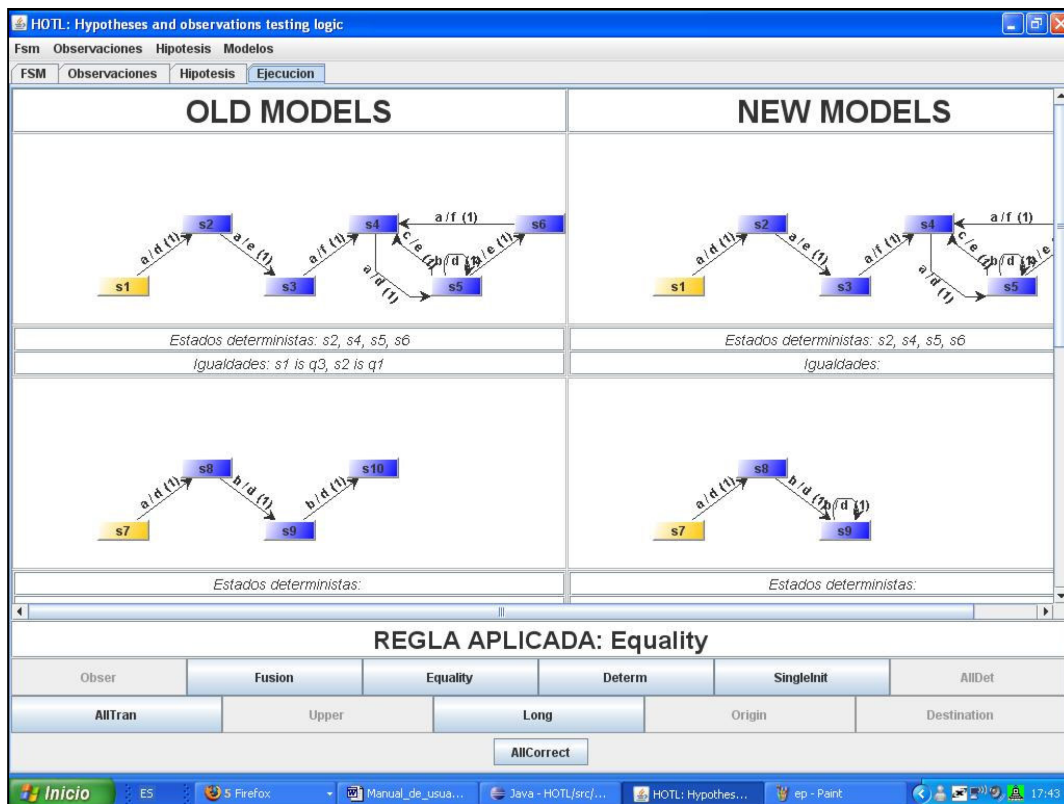
Ejemplo de ejecución por pasos:



El último modo de ejecución es la ejecución manual. En él se puede elegir manualmente la siguiente regla a aplicar, por lo tanto la aplicación de las reglas no sigue ningún algoritmo como en los dos modos de ejecución anteriores. En la parte de abajo de la pantalla se encontrarán los botones con las posibles reglas a aplicar, en gris claro se encontrarán las reglas que se pueden aplicar y en gris oscuro las que no pueden ser aplicadas porque las hipótesis necesarias para su aplicación no han sido tomadas o porque son reglas que ya han sido aplicadas y sólo deben ser aplicadas una

vez. La información sobre los modelos que se muestra en la pantalla tras la aplicación de una regla es la misma que para el modo de ejecución anterior. Al igual que en los casos anteriores cuando el usuario aplique la regla "AllCorrect", en este caso manualmente cuando él decida, se mostrará por pantalla los worst-case de los modelos finales surgidos sobre los que se ha estudiado su conformidad.

Ejemplo de ejecución manual:

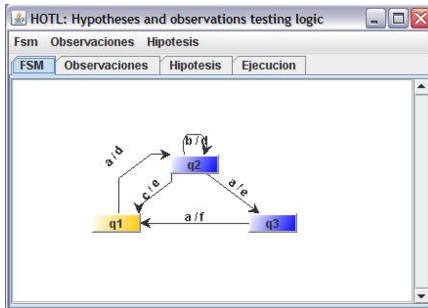


Una vez ha sido aplicada la regla "AllCorrect" para cualquier modo de ejecución (Automáticamente para los dos primeros o cuando el usuario decida para el tercero), se mostrará si es conforme o no y dicha ejecución terminará. Para continuar con una nueva ejecución habrá que pulsar el botón "Aceptar" en la parte inferior de la pantalla y seleccionar la nueva ejecución que se desee. En el caso de que para la nueva ejecución se decida cambiar las hipótesis, observaciones o especificación, habrá que cargar las nuevas que se deseen antes de comenzarla.

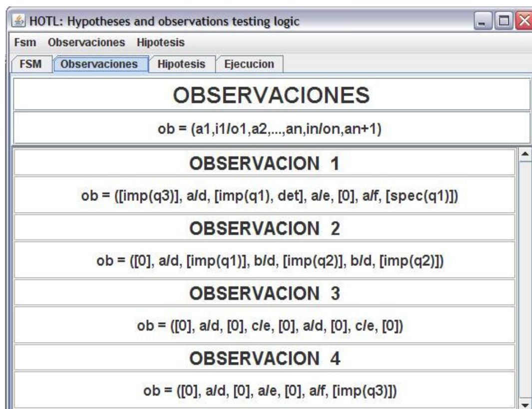
3.3. Ejemplos de uso

A continuación mostraremos la ejecución de un ejemplo positivo, en el cual, tras cargar la especificación, observaciones, hipótesis y ejecutar las reglas obtenemos que la especificación es conforme con su implementación.

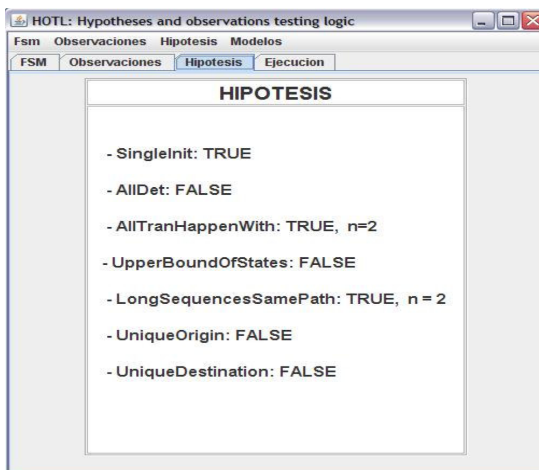
Especificación del ejemplo:



Observaciones del ejemplo:

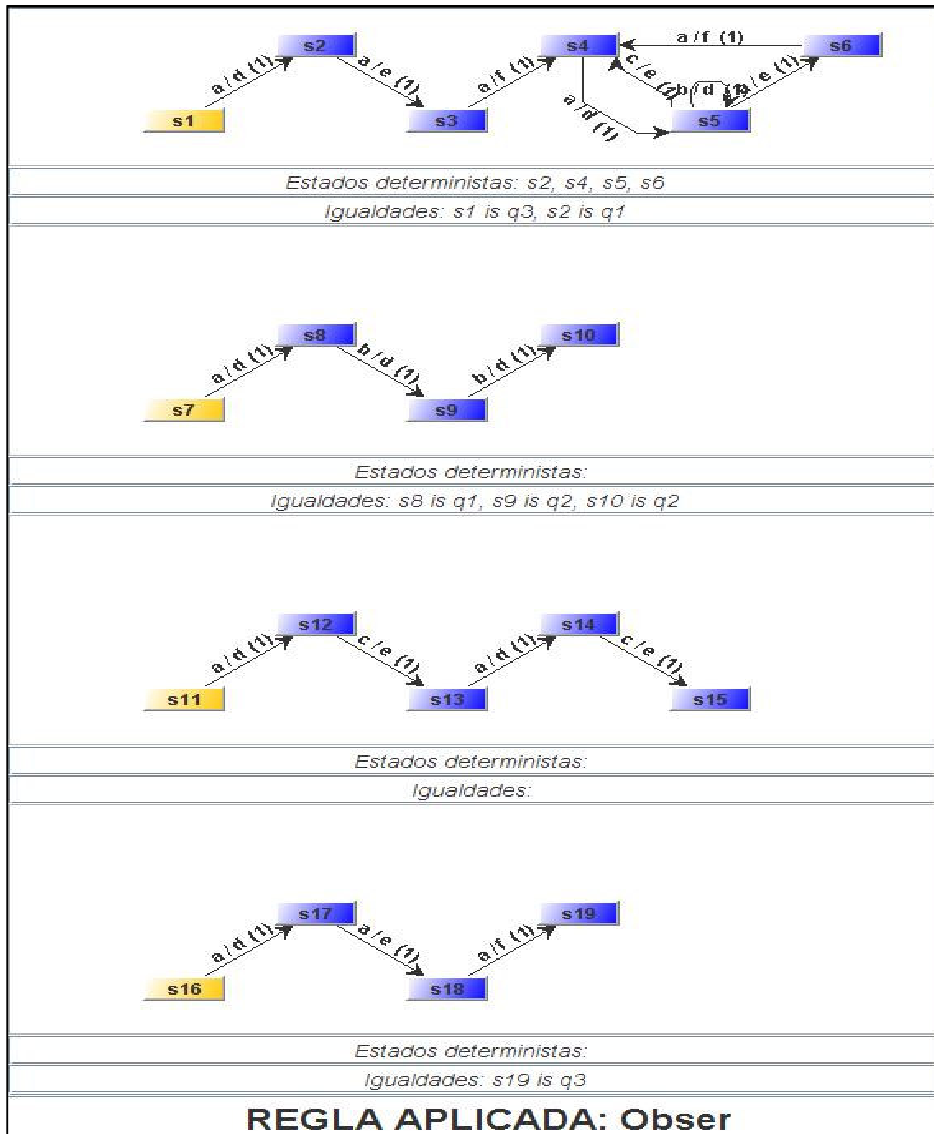


Hipótesis del ejemplo:



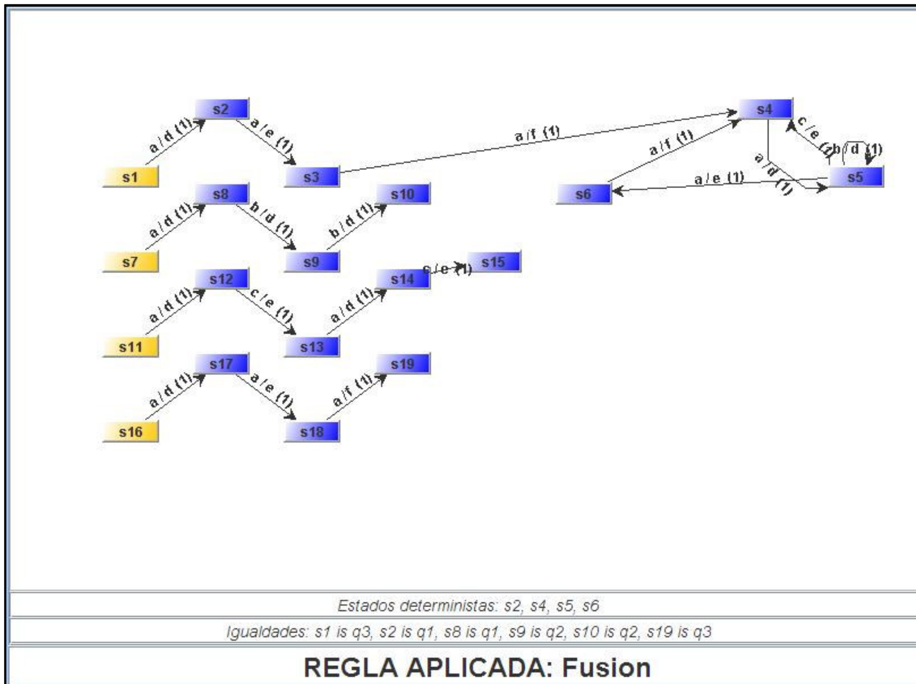
Comienzo de la ejecución:

Aplicamos Obser



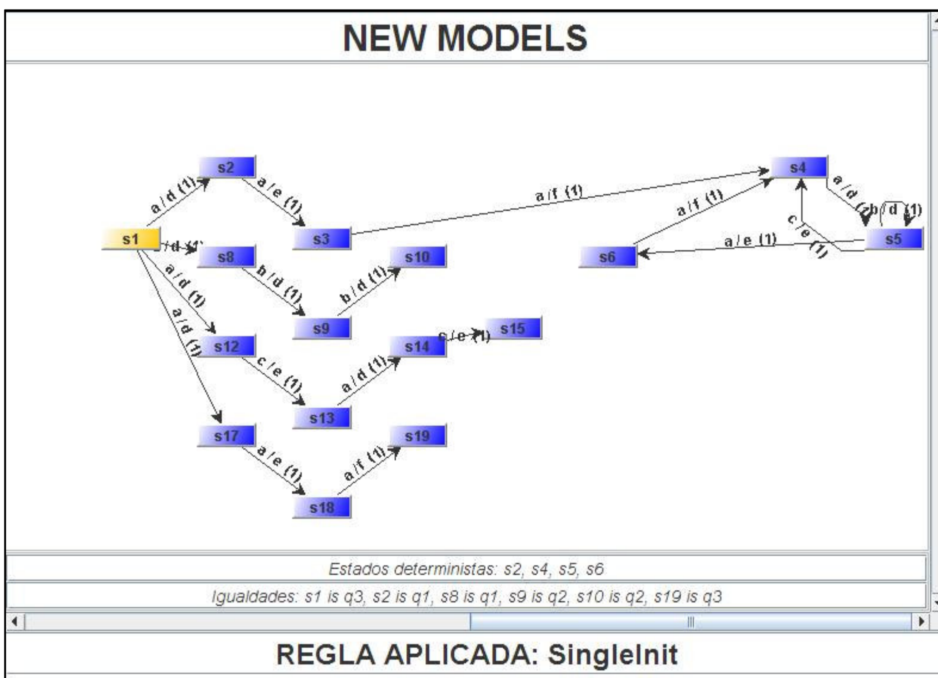
Tras aplicar la regla "Obser" obtenemos tantos modelos como observaciones hayamos hecho de la implementación. En nuestro caso el fichero de observaciones tenía cuatro.

Aplicamos Fusion:



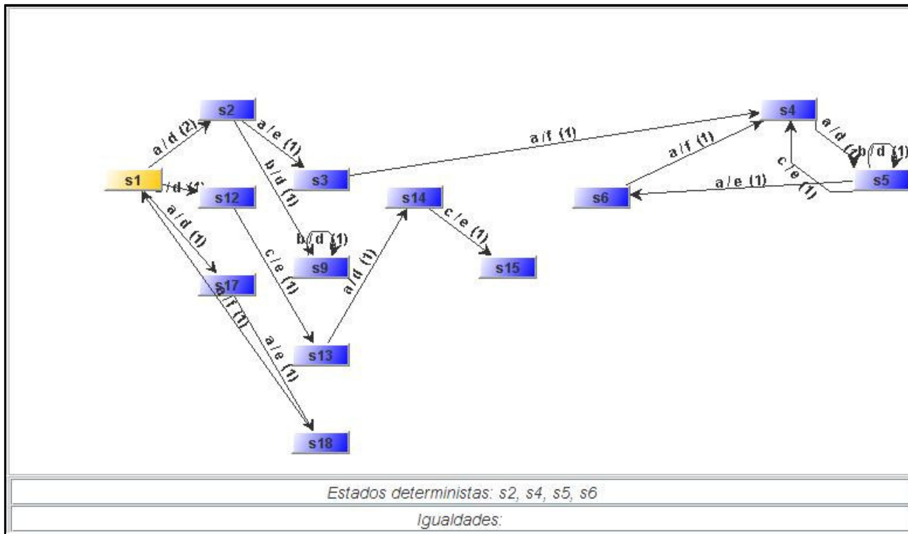
Obtenemos un único modelo uniendo los cuatro anteriores.

Aplicamos SingleInit:



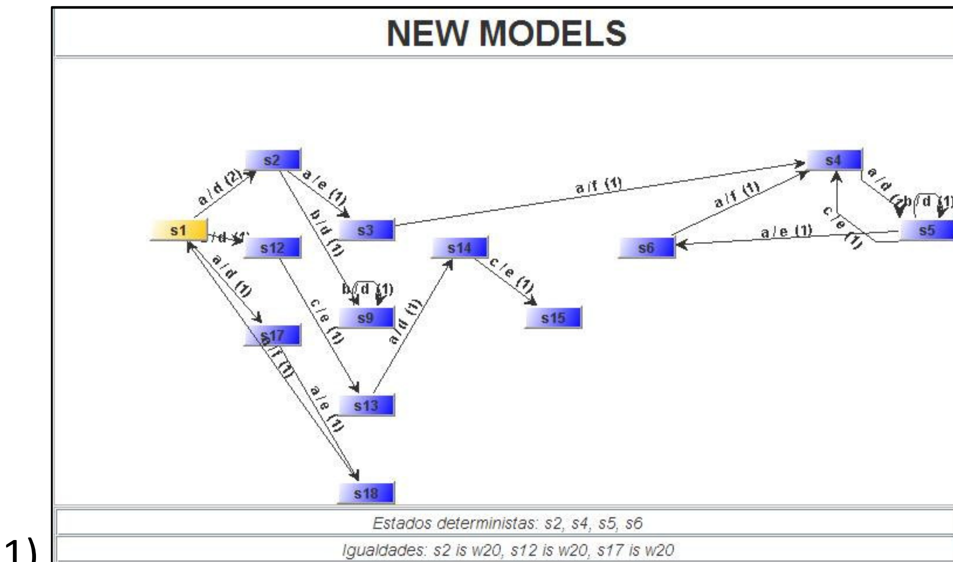
Todos los estados iniciales se fusionan en uno solo.

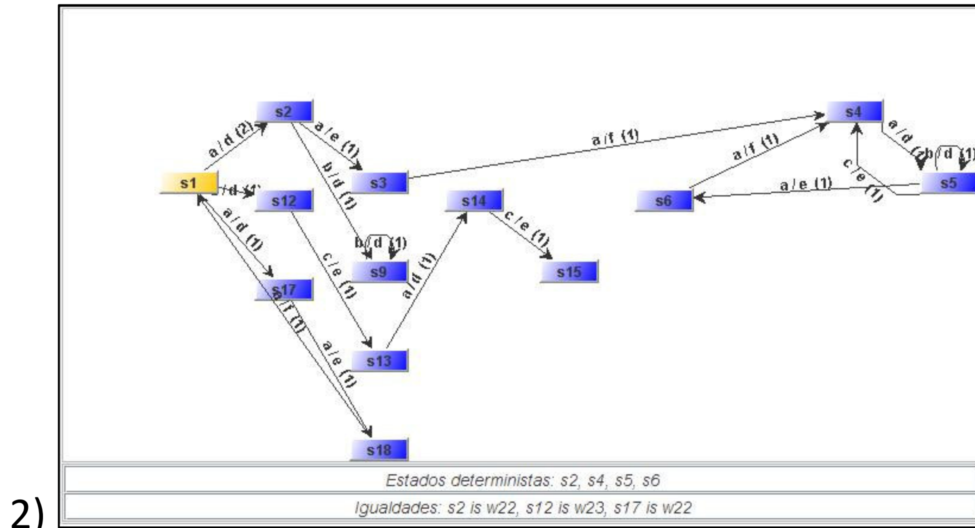
Aplicamos equality:



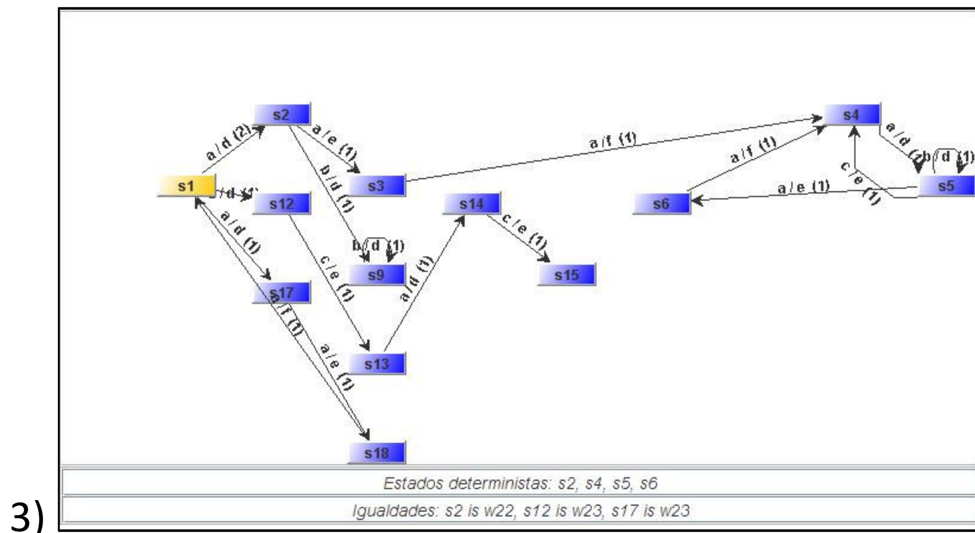
Se eliminan los estados correspondientes a las igualdades almacenadas en el modelo anterior.

Aplicamos AllTran:

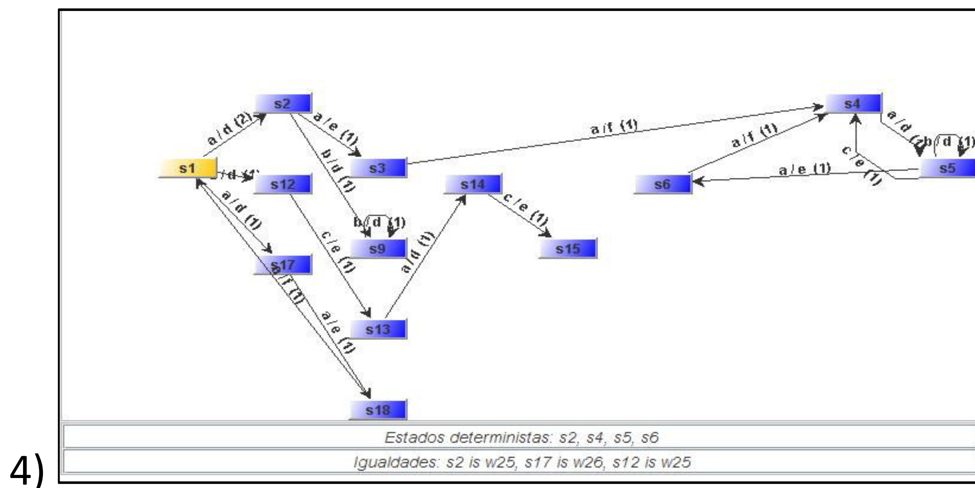
1) 



2)



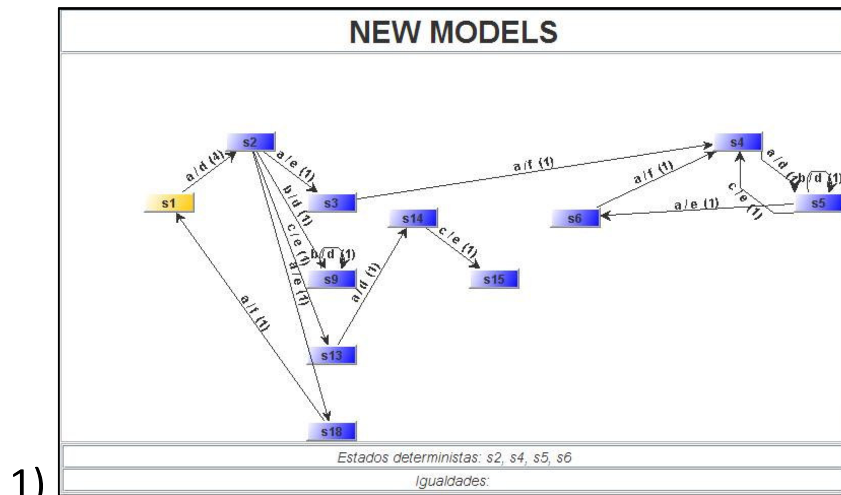
3)



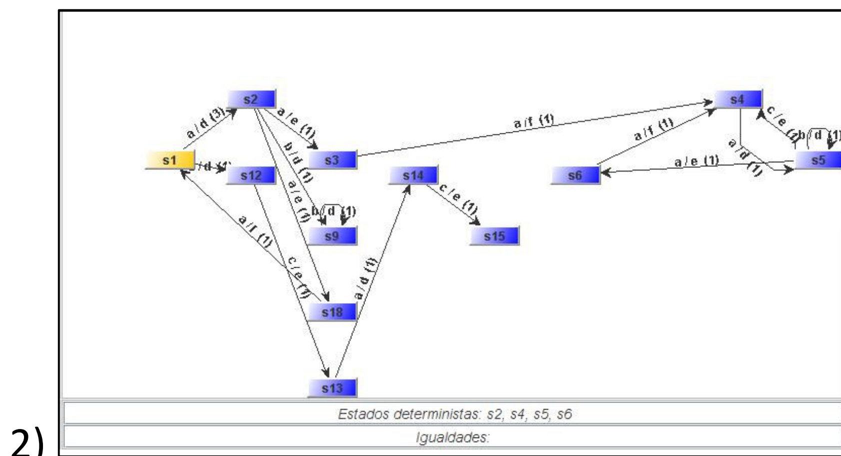
4)

Obtenemos a partir del modelo anterior cuatro nuevos modelos iguales salvo por el conjunto de igualdades de cada uno.

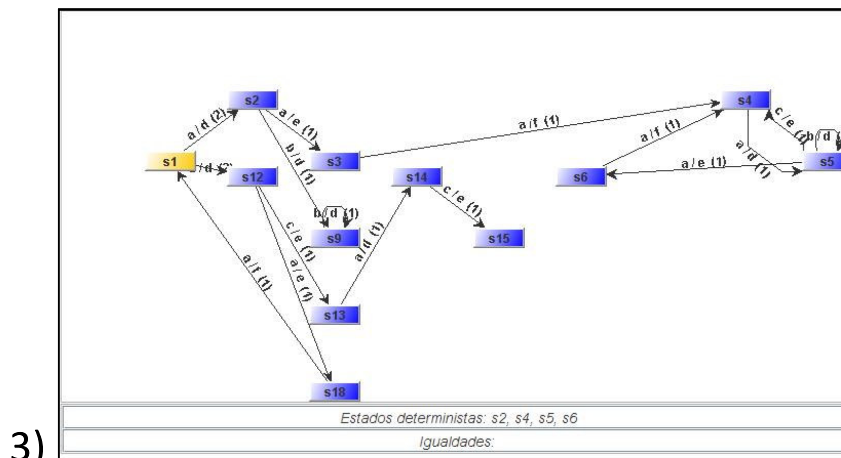
Aplicamos Equality:



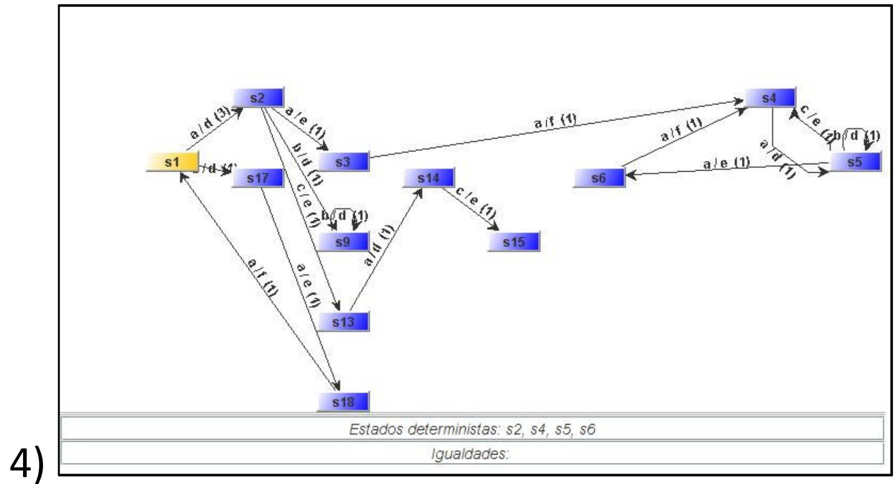
1)



2)



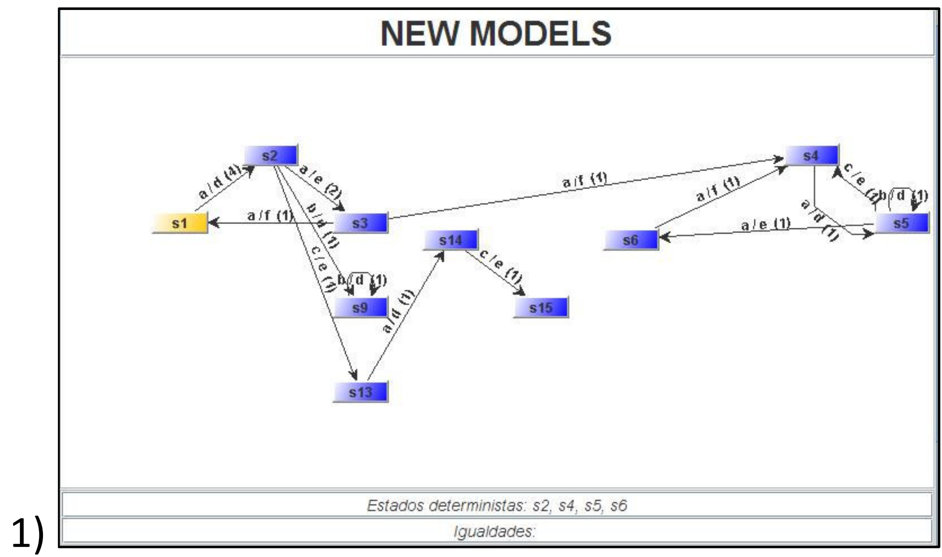
3)



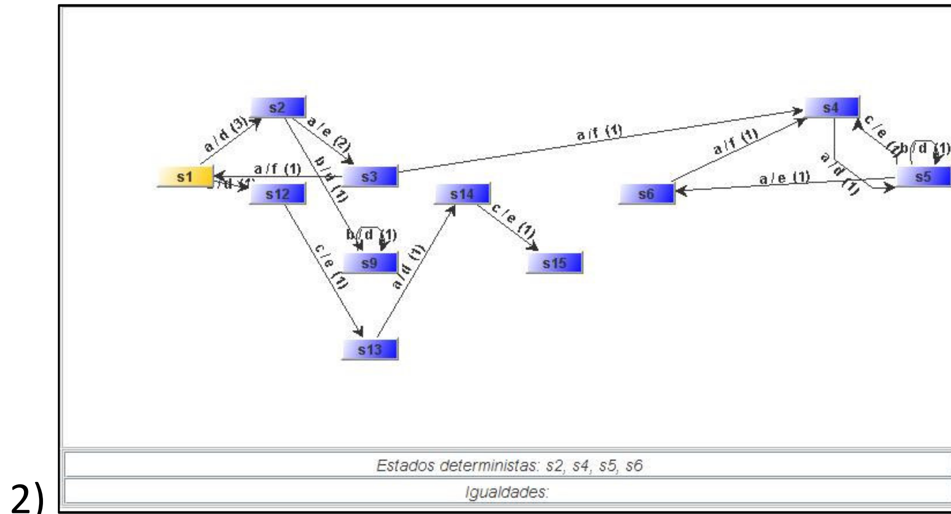
4)

Tras la aplicación de Equality obtenemos modelos distintos y con sus igualdades vacías.

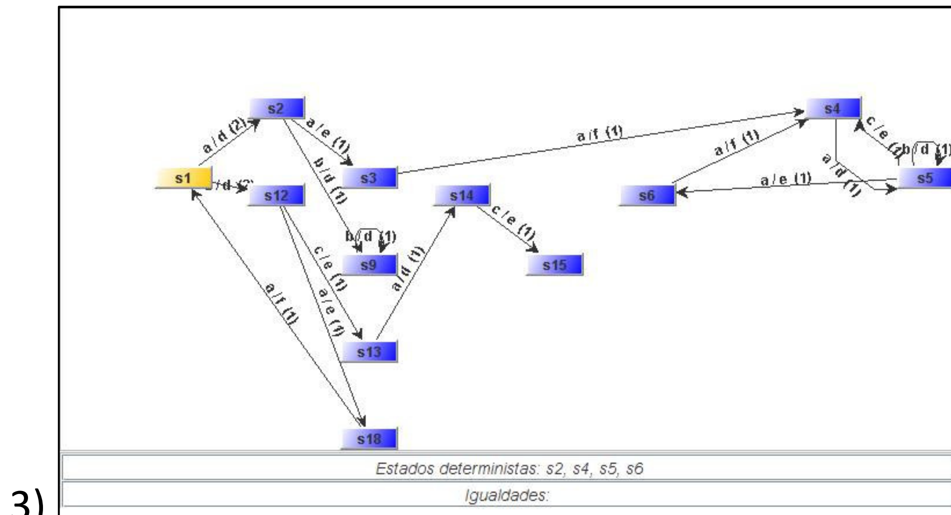
Aplicamos Determ:



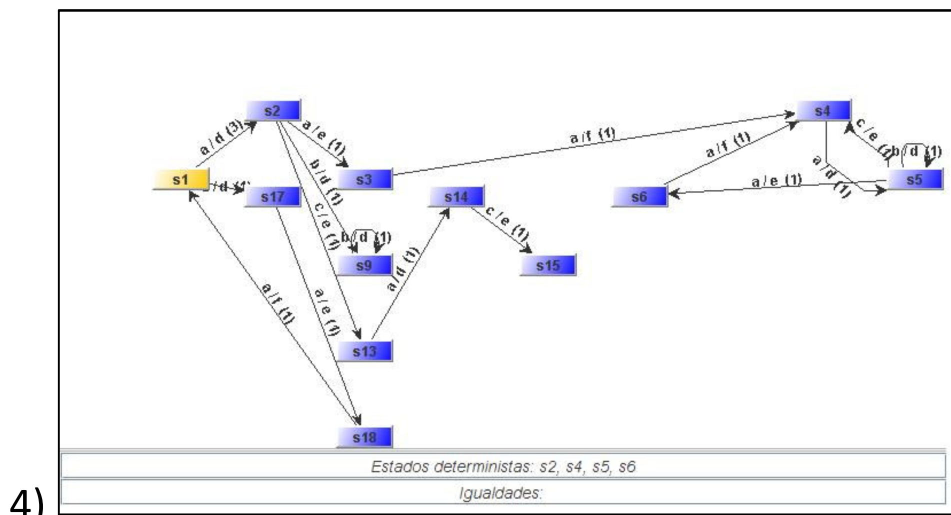
1)



2)

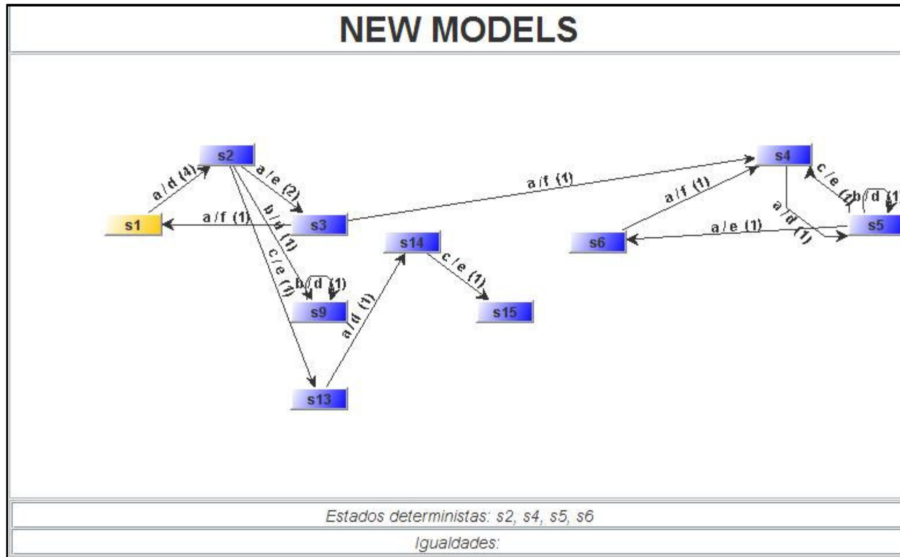


3)

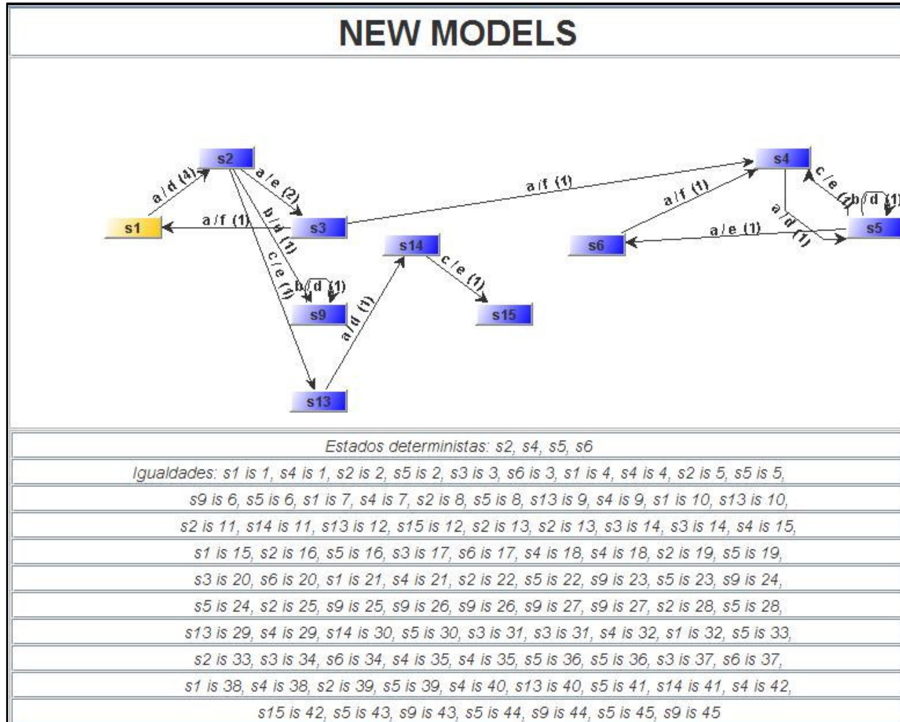


4)

Por el bien de la claridad del ejemplo hemos omitido la iteración que se produce en la ejecución de las reglas en referencia al bucle que comprende a “AllTran” y a “Equality” y que como última instancia produce cuatro modelos iguales de los cuales se eliminan tres por ser redundantes. Como resultado tenemos:

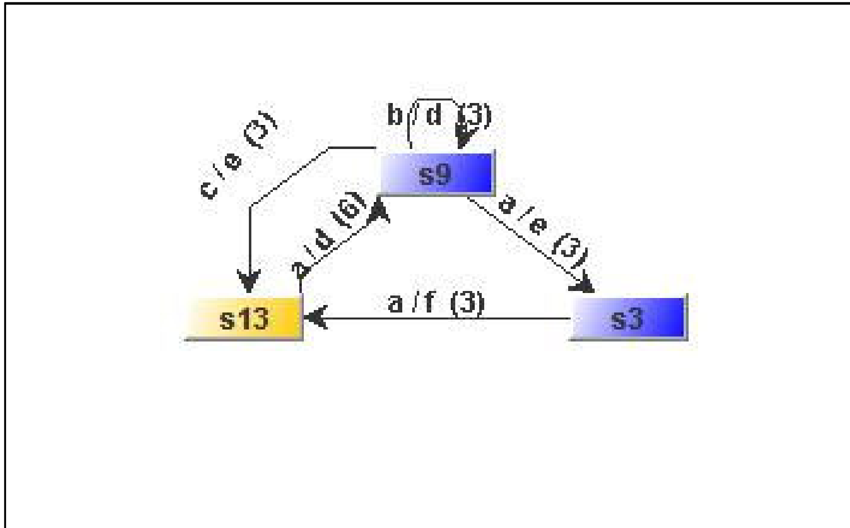


Aplicamos Long:

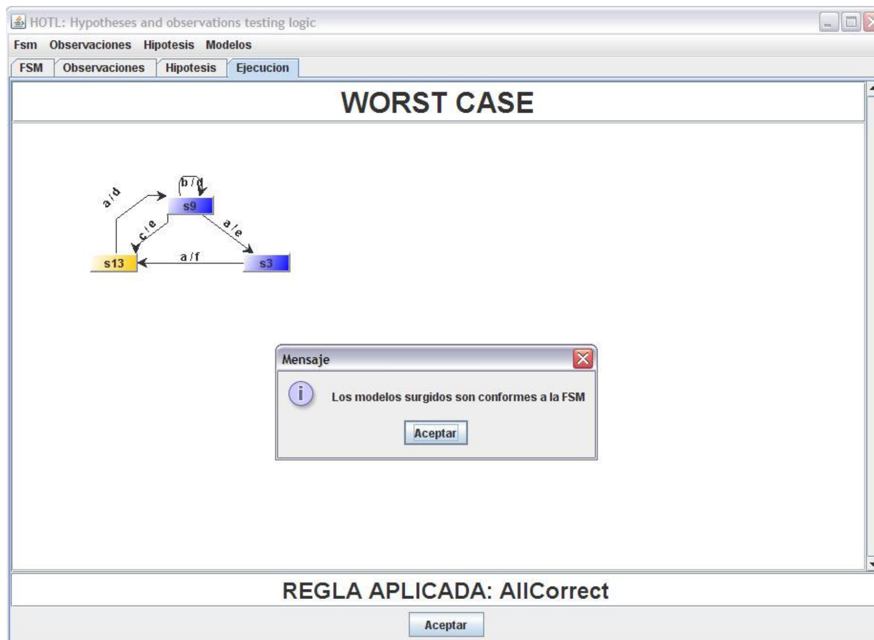


Esta regla, como la gran mayoría, es exponencial, lo cual se puede observar al fijarse en el conjunto de igualdades que se ha generado.

Aplicamos equality:



Aplicamos AllCorrect:



En este punto obtenemos el caso peor del modelo que ha resultado al aplicar las reglas de HOTL. En este caso el caso peor es idéntico al modelo del que se ha generado ya que el comportamiento de los estados está cerrado y no se pueden crear transiciones maliciosas.

En última instancia aplicamos AllCorrect ya que el conjunto de modelo era consistente, no se les podía aplicar ninguna regla más, y obtenemos como resultado la indicación de que la implementación es conforme con la especificación.

3.4. Posibles extensiones

El trabajo realizado se puede seguir ampliando, tanto la parte de la lógica HOTL como la parte de la herramienta implementada.

3.4.1. Extensiones de la lógica

Relativo a la lógica HOTL podría extender el repertorio de hipótesis permitiendo formalismos más expresivos para la representación de especificaciones e implementaciones, por ejemplo integrando en HOTL algoritmos de *testing* conocidos que eviten el crecimiento exponencial de modelos tras la aplicación de algunas reglas, etc.

También sería deseable probar la lógica con alguna aplicación real más compleja como podrían ser los protocolos de Internet.

También se podría añadir una puntuación de "viabilidad" para cada una de las reglas. Por ejemplo para una determinada prueba asumir que todos los estados son deterministas es "más fuerte" que asumir que la implementación tiene menos de 50 estados. En este caso, una menor puntuación de "viabilidad" será asignada a la primera hipótesis. Teniendo en cuenta la "viabilidad" de todas las hipótesis que han sido añadidas antes de asegurarnos la conformidad, obtendremos una medida de lo apropiadas que han sido las observaciones consideradas e indirectamente, de las pruebas que hemos realizado para obtenerlas. Por lo tanto la lógica puede ayudar a un testador a elegir sus pruebas de manera que los resultados obtenidos sean más creíbles.

Todas estas ampliaciones les correspondería realizarlas a los autores de la investigación en la cual hemos basado la aplicación.

3.4.2. Extensiones de la aplicación

Dentro de la aplicación, que ha sido nuestra área de trabajo, las posibles ampliaciones pueden ser muy numerosas, más que ampliaciones, serían ideas que han ido surgiendo durante el desarrollo de la aplicación o consejos dados por los autores de la lógica que no han podido llevarse a cabo por falta de tiempo.

A nivel de funcionalidades de la práctica se podría añadir el poder introducir observaciones e hipótesis en cualquier momento durante la ejecución manual de las reglas de manera que las pruebas fueran más dinámicas. Así el testador podría ir añadiendo más observaciones e hipótesis en función de sus necesidades. Tal y como está ahora la herramienta, la aplicación de las reglas se hará con unas hipótesis y unas

observaciones cargadas o creadas de antemano no pudiendo modificarlas durante la aplicación de las reglas, de manera que si se quiere añadir una nueva observación o hipótesis sea necesario cargar de nuevo éstas junto con las que se quieren añadir y comenzar de nuevo. Éste sería un punto muy interesante a añadir.

La mayoría de las mejoras que se pueden añadir serían en la parte gráfica de la aplicación, por ejemplo:

- La parte de la interfaz para crear las observaciones podría mejorarse bastante, de manera que sea más intuitiva para el usuario, por ejemplo que los atributos y las entradas y salidas se puedan seleccionar desde cuadros de selección y arrastrarlas y vaya mostrando la observación a medida que la vas creando. Debido a que la idea inicial era simplemente cargarlas desde un fichero ya que lo lógico de una aplicación que realiza pruebas es que las observaciones las devuelva en un fichero. Al final se pidió que las observaciones también se pudieran crear manualmente y por falta de tiempo no se pudo hacer todo lo intuitivo que se deseaba.
- La especificación, se carga también desde fichero. Una ampliación podría ser poder pintarla manualmente añadiendo los estados y las transiciones con el mismo formato que se pinta una vez cargada, es decir, añadir un editor para crear las especificaciones. Esto proporcionaría a la herramienta una mayor simplicidad a la hora de crear la especificación ya que evitaría tener que crear el archivo de texto y además le daría un toque más profesional.
- Mejora de la forma de pintar los modelos
- Mejora de los botones, paneles,… (colores de fondo, bordes, fuente, etc.).

Debido a la modularidad y reusabilidad del código escrito, ya que se han usado un gran número de patrones de diseño y se ha dividido la aplicación en diversos componentes para facilitar la comprensión del código escrito, la aplicación sería fácilmente mejorable y ampliable en un futuro si así se necesitase (Cambios en la lógica “HOTL”, añadir nuevas funcionalidades…).

4. CONCLUSIONES GENERALES

En *primer lugar* hacemos mención a las conclusiones teóricas en relación al trabajo de investigación en que se ha basado nuestra herramienta:

Nuestra aplicación se basa en una lógica para deducir si una colección de observaciones obtenidas testando una IUT junto con un conjunto de hipótesis que permiten deducir que la IUT es conforme a la especificación.

Un repertorio heterogéneo de hipótesis produciendo un testador con expresividad para indicar que un amplio rango de escenarios de pruebas han sido presentados. Considerando esas observaciones e hipótesis que encajan mejor en sus necesidades, el testador puede obtener resultados diagnósticos sobre la conformidad de una IUT en un rango flexible de situaciones. Además la lógica permite añadir iterativamente observaciones y/o hipótesis hasta completar el conjunto de predicados garantizando la conformidad. En este sentido la lógica puede ser usada para guiar dinámicamente de una metodología de testing.

En *segundo lugar* mencionamos las conclusiones referentes al trabajo realizado por el grupo de proyecto a lo largo del periodo de realización del mismo:

Una vez terminado el proyecto el grupo de trabajo se encuentra satisfecho por los resultados obtenidos, pues los principales objetivos planteados, en el momento en que se nos presento cual deberían de ser nuestro cometido, se han llevado acabado con éxito.

Entre los objetivos mencionados cabe destacar:

- Estudio y entendimiento de la lógica "HOTL" desarrollado por nuestro profesor director y sus dos compañeros de investigación.
- Implementación y correcto funcionamiento de dicha lógica sobre una plataforma de diseño consistente y eficaz, en nuestro caso, Java.
- Esta implementación nos ha copado la gran mayoría del tiempo disponible para la realización de dicho proyecto.
- Diseño de una arquitectura clara que permita la ampliación futura de la aplicación.
- La implementación de una interfaz gráfica capaz de mostrar al usuario, en todo momento, los resultados parciales y finales de la ejecución. Así como la

carga a través de fichero tanto de una especificación como de observaciones e hipótesis y la creación de estos últimos.

- Y por último, como ya ha sido mencionado en puntos anteriores, este proyecto está abierto a ampliaciones y mejoras futuras.

Consideramos ante todo que la aplicación desarrollada en este proyecto tiene como principales características una gran capacidad didáctica ya que es capaz de mostrar de una forma clara y concisa la aplicación de la lógica en pequeños ejemplos de muestra. Es por ello una herramienta de gran potencial para la introducción en el mundo del testing lógico y la lógica HOTL en concreto.

5. AGRADECIMIENTOS

Especial agradecimiento al director del proyecto y profesor de la Facultad de Informática de la Universidad Complutense de Madrid, Don Ismael Rodríguez Laguna por aceptarnos en su proyecto para la asignatura de Sistemas Informáticos y por la atención y ayuda ofrecida para la realización de este.

Agradecimientos a, Don Manuel Núñez y Doña Mercedes G. Merayo profesores de la Facultad de Informática de la Universidad Complutense de Madrid y co-autores del artículo de investigación sobre el que se basa nuestro proyecto, por su ayuda, espíritu crítico y disponibilidad en todo momento.

6. BIBLIOGRAFÍA

- I.Rodriguez et al., HOTL: Hipotheses and observations testing logic, J. Logic Algebr. Progr. (2007), doi: 10.1016/j.jlap.2007.03.002.
- María Teresa Hortalá, Javier Leach y Mario Rodríguez, *Matemática discreta y lógica matemática. 2ª Edición.* Editorial complutense 2001.
- JacobsonI.,Booch G., Rumbaugh J., *El proceso unificado de desarrollo del software*. Addison-Wesley.
- Pressman R.S., *Ingeniería del software. Un enfoque práctico. 6ª Edición.* McGraw, 2005.
- Alur, D., Crupi, J., Malks, D. *Core J2EE Patterns: Best Practices and Design Strategies, 2 Edition.* Prentice-Hall PTR, 2003
- http://www.corej2eepatterns.com
- http://es.wikipedia.org/wiki/Modularidad
- http://pisuerga.inf.ubu.es/lsi/Invest/Java/Tuto/I_1.htm
- http://csdl2.computer.org/persagen/DLAbsToc.jsp?resourcePath=/dl/mags/dt/&toc=comp/mags/dt/1997/01/d1toc.xml&DOI=10.1109/54.573365
- http://es.wikipedia.org