UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE CIENCIAS MATEMÁTICAS

Departamento de Sistemas Informáticos y Programación

TÉCNICAS DE ESPECIFICACIÓN FORMAL DE SISTEMAS ORIENTADOS A OBJETOS BASADOS EN LÓGICA DE REESCRITURA

MEMORIA PRESENTADA PARA OPTAR AL GRADO DE DOCTOR POR

María Isabel Pita Andreu

Bajo la dirección del Doctor:

Narciso Martí Oliet

Madrid, 2003

ISBN: 84-669-1804-3

Técnicas de especificación formal de sistemas orientados a objetos basadas en lógica de reescritura

UCM
UNIVERSIDAD COMPLUTENSE MADRID

TESIS DOCTORAL

María Isabel Pita Andreu

Departamento de Sistemas Informáticos y Programación
Facultad de Ciencias Matemáticas
Universidad Complutense de Madrid
Enero 2003

Técnicas de especificación formal de sistemas orientados a objetos basadas en lógica de reescritura

Memoria presentada para obtener el grado de
Doctor en Matemáticas

María Isabel Pita Andreu

Dirigida por el profesor

Narciso Martí Oliet

Departamento de Sistemas Informáticos y Programación
Facultad de Ciencias Matemáticas
Universidad Complutense de Madrid

Enero 2003

A mis padres,
a mi marido y
a mis hijas

Resumen

Las técnicas de especificación formal de sistemas concurrentes pueden agruparse en general en dos niveles; en el primero se incluyen las técnicas consistentes en el desarrollo de modelos formales del sistema y en el segundo las técnicas que realizan la especificación del sistema mediante la definición de propiedades abstractas del mismo.

El objetivo de esta tesis es proponer una metodología de especificación de sistemas que cubra ambos niveles de especificación mediante el uso de un marco matemático uniforme, proporcionado por la lógica de reescritura y su implementación vía el metalenguaje Maude. La especificación en el primer nivel se realizará directamente en el propio lenguaje Maude, mientras que para realizar la especificación de segundo nivel definiremos una lógica modal para probar propiedades de sistemas especificados en Maude, en la cual las transiciones definidas por las reglas de reescritura se capturan como acciones en la lógica. La lógica definida puede utilizarse además mediante la definición de la interfaz apropiada para probar propiedades especificadas en otras lógicas temporales o modales.

En la tesis se estudian en primer lugar las especificaciones en el lenguaje Maude. Mediante el desarrollo de una especificación de un modelo orientado a objetos para redes de telecomunicación de banda ancha se muestra el poder del lenguaje para especificar este tipo de sistemas y en particular la relación de herencia, la relación de contenido y las relaciones explícitas de grupo (ser-miembro-de, cliente-servidor, ...). Se estudia el uso de la reflexión en el control de un proceso de modificación de características de la red. En este sentido se combinan ideas del campo de la reflexión lógica con ideas provenientes del campo de la reflexión orientada a objetos mediante el uso de un mediador, un metaobjeto que vive en el metanivel y que tiene acceso a la configuración de la red para su gestión.

En segundo lugar se procede a la definición de la lógica modal *Verification Logic for Rewriting Logic* (VLRL). La principal característica de esta lógica es que proporciona dos modalidades, una de ellas una modalidad de acción que permite capturar las reglas de reescritura como acciones de la lógica, y la otra una modalidad espacial que permite definir propiedades sobre partes del sistema y relacionarlas con propiedades del sistema completo así como definir propiedades sobre acciones realizadas en partes del sistema. La lógica VLRL permite además probar propiedades definidas en otras lógicas modales o temporales mediante la definición de la interfaz apropiada. Se muestra el uso de la lógica en la prueba de propiedades de seguridad de varios sistemas orientados a objetos: un protocolo de exclusión mutua, el sistema del mundo de los bloques y el sistema Mobile Maude como modelo de movilidad de objetos entre procesos.

Por último se muestra otro medio de probar propiedades de sistemas especificados en lógica de reescritura mediante un ejemplo en el que se realiza una prueba semi-formal por inducción de propiedades de seguridad y vivacidad del protocolo para la elección de líder del bus en serie multimedia IEEE 1394.

Agradecimientos

Mi primer agradecimiento debe ser para mi director de tesis, Narciso Martí Oliet, por toda la paciencia que ha tenido durante estos años y por su constante confianza en mi trabajo, además de sus explicaciones a todas mis dudas y sus múltiples y detallados comentarios a todos los artículos que hemos publicado juntos. En segundo lugar debo agradecer su apoyo a todos los miembros del grupo de lógica de reescritura del departamento, en particular a Alberto Verdejo que siempre tiene un momento para ayudar y a quien le debo no solo la especificación del sistema IEEE 1394, sino también muchísimas aclaraciones sobre cómo ejecutar especificaciones en Maude, el sistema Mobile Maude y el editor Latex; a Miguel Palomino por sus interesantes comentarios a varios de los trabajos que figuran en la tesis; y a Manuel García Clavel que, además de llegar al departamento en el momento más apropiado para facilitar el empujón final a la tesis, me ha proporcionado interesantes comentarios a la versión preliminar. Deseo agradecerles a todos ellos el ambiente de trabajo que han sabido crear y el compañerismo que demuestran a diario.

No puedo olvidar, aunque se encuentre más lejos, a José Meseguer como pionero en los estudios sobre lógica de reescritura. Quiero agradecerle no solo su trabajo en esta área, sino también el apoyo que ha mostrado siempre a nuestro grupo y en particular el cariño que siempre ha demostrado hacia mí. Debo mencionar también a Francisco Durán, que desde Málaga siempre resuelve las dudas que tenemos del lenguaje Maude, y a José Luiz Fiadeiro, por todo su trabajo inicial en la lógica VLRL que forma la base de esta tesis, su apoyo en la acción integrada que tuvimos juntos y en los artículos que hemos publicado.

También debo agradecer su ayuda y apoyo a muchos miembros del departamento: a David de Frutos, por su apoyo al incorporarme al ámbito universitario y el haberme permitido colaborar en todos los proyectos que ha dirigido desde entonces; a Mario, por sus comentarios a versiones preliminares de esta tesis, y a los demás miembros del proyecto TREND: Antonio, Joaquín, Paco, Puri, Ana, Susana, etc.; a María Inés, Yolanda y Margarita, en quienes siempre he podido confiar; a Ricardo, que me proporcionó mi primer odenador en la universidad y siempre confió en que terminaría la tesis; a Luis Llana, por la paciencia que ha tenido conmigo cuando el ordenador no me funciona; a todos los miembros más jovenes del departamento, a los que siempre se puede acudir cuando se tiene una duda: Clara, Fernando, Natalia, Alberto E...; y a aquellos que han facilitado mi trabajo en la universidad, como Teresa, siempre dispuesta a encontrar una solución a un problema con las clases o los laboratorios.

Por último agradezco a los miembros del tribunal su rápida disposición a participar en el mismo y el trabajo dedicado a evaluar la tesis.

Índice general

1. Introducción **1**

1.1. Contexto . . . 2

1.1.1. Sistemas orientados a objetos . . . 2

1.1.2. La lógica de reescritura y el lenguaje Maude . . . 3

1.1.3. Reflexión . . . 3

1.1.4. Lógicas modales . . . 4

1.1.5. Operadores espaciales . . . 6

1.1.6. Lógicas temporales . . . 6

1.1.7. Clasificación de propiedades . . . 8

1.2. Decisiones de diseño de la lógica modal . . . 9

1.3. Aportaciones de la tesis . . . 10

2. La lógica de reescritura y el lenguaje Maude **13**

2.1. La lógica de reescritura . . . 13

2.1.1. Reglas de inferencia . . . 13

2.1.2. Modelos de la lógica de reescritura . . . 15

2.1.3. Características reflexivas de la lógica de reescritura . . . 17

2.1.4. Especificaciones de sistemas de tiempo real en lógica de reescritura . 17

2.2. El lenguaje Maude . . . 19

2.2.1. Módulos funcionales . . . 19

2.2.2. Módulos de sistema . . . 21

2.2.3. Módulos orientados a objetos . . . 22

2.2.4. Módulos parametrizados . . . 27

2.2.5. Reflexión y lenguajes de estrategias . . . 28

2.2.6. El sistema Mobile Maude . . . 31

3. Especificación de un modelo de red de telecomunicaciones **35**

3.1. Descripción del modelo . . . 35

3.1.1. Objetos de la red . . . 36

3.1.2. Relaciones entre los objetos de la red . . . 37
3.2. La especificación de la red . . . 38
3.2.1. Las clases orientadas a objetos . . . 38
3.2.2. Implementación de las relaciones entre objetos . . . 39
3.2.3. Mensajes de acceso a la información . . . 44
3.2.4. Mensajes de modificación . . . 46
3.3. Especificación de la red usando reflexión . . . 51
3.3.1. Estrategias y reglas para el proceso de modificación utilizando reflexión 51
3.3.2. Comparación de las dos especificaciones . . . 55
3.3.3. Mejorando el control al cambiar las estrategias . . . 56
3.4. Conclusiones . . . 58

4. Propiedades modales y temporales **61**
4.1. Programas, especificaciones y verificación . . . 61
4.2. La lógica de verificación VLRL . . . 63
4.2.1. Signaturas de verificación . . . 63
4.2.2. Ejemplo: una máquina expendedora . . . 64
4.2.3. El lenguaje de acciones . . . 65
4.2.4. Modelos . . . 67
4.2.5. El lenguaje modal . . . 68
4.2.6. La relación de satisfacción . . . 69
4.3. Un procedimiento de decisión para la lógica VLRL . . . 70
4.3.1. Definición del modelo filtrado . . . 71
4.3.2. Procedimiento de decisión . . . 73
4.3.3. Corrección y complejidad del algoritmo . . . 74
4.3.4. Ejemplo de aplicación del algoritmo de decisión . . . 76
4.4. Teoría de demostración para la lógica VLRL . . . 82
4.4.1. Modalidad de acción . . . 82
4.4.2. Modalidad espacial . . . 85
4.4.3. Algunas consideraciones sobre la completitud de la lógica . . . 86
4.4.4. Propiedades espaciales básicas . . . 88
4.5. Transiciones *en contexto* . . . 91
4.6. Especificación de propiedades temporales . . . 94
4.6.1. Lógica temporal sobre VLRL . . . 95
4.6.2. Completitud de las reglas de inferencia de la interfaz de la lógica temporal con VLRL . . . 97
4.6.3. Ejemplo: Propiedades temporales de la máquina expendedora . . . 99
4.7. Conclusiones . . . 104

5. Especificación de sistemas orientados a objetos 105
5.1. Un ejemplo sencillo de exclusión mutua . . . 106
5.1.1. Probando la exclusión mutua . . . 107
5.1.2. Probando un invariante . . . 108
5.2. Un mundo de bloques . . . 109
5.3. Observando el número de bloques . . . 110
5.3.1. Definición de las observaciones . . . 110
5.3.2. Ejemplos de modalidad espacial y modalidad de acción . . . 112
5.3.3. Propiedades básicas sobre el número de bloques . . . 113
5.3.4. Una propiedad temporal sobre el número de bloques . . . 114
5.4. Observando la posición de los bloques . . . 116
5.4.1. Definición de las observaciones . . . 116
5.4.2. Propiedades básicas sobre la posición de los bloques . . . 117
5.4.3. Una propiedad temporal sobre la posición de los bloques . . . 119
5.4.4. Otra propiedad temporal sobre la posición de los bloques . . . 120
5.5. El sistema Mobile Maude . . . 122
5.5.1. Definición de las observaciones . . . 122
5.5.2. Propiedades espaciales básicas de Mobile Maude . . . 123
5.5.3. Combinando acciones con propiedades espaciales . . . 124
5.5.4. Uso de propiedades espaciales para probar propiedades temporales . 125
5.6. Conclusiones . . . 126

6. Verificación semi-formal del protocolo IEEE 1394 127
6.1. Descripción informal del protocolo . . . 127
6.2. Primera especificación del protocolo (con comunicación síncrona) . . . 128
6.3. Especificación con comunicación asíncrona y con tiempo . . . 129
6.4. Tercera descripción del protocolo . . . 133
6.5. Verificación de propiedades . . . 135
6.5.1. Verificación de la especificación síncrona . . . 135
6.5.2. Verificación de la segunda especificación . . . 136
6.5.3. Verificación de la tercera especificación . . . 143
6.6. Conclusiones . . . 145

7. Temas de trabajo futuro 147

A. Recopilación de las reglas de inferencia de la lógica VLRL 151

Capítulo 1

Introducción

La especificación formal de los sistemas reporta numerosas ventajas al desarrollo del software, como es el forzar a los diseñadores de programas a realizar decisiones precisas sobre las principales funcionalidades del programa y a eliminar las ambigüedades de la descripción del comportamiento. Los primeros métodos de especificación formal fueron métodos basados en aserciones, con las que se definía la relación existente entre la entrada y la salida del programa. Estos métodos estaban diseñados para especificar propiedades de programas secuenciales que producen una serie de valores al finalizar su ejecución, pero no pueden ser aplicados a sistemas que no terminan en los que, por esi mismo, no existe un estado final.

El desarrollo de sistemas concurrentes que producen resultados al interaccionar con el entorno pero que carecen de estado final ha dado lugar al desarrollo de distintas técnicas de especificación formal. Estas técnicas pertenecen, en general, a dos niveles de especificación; en el primero se incluyen las técnicas consistentes en el desarrollo de modelos formales del sistema y en el segundo las técnicas que realizan la especificación del sistema mediante la definición de propiedades abstractas del mismo. Las especificaciones del primer nivel pueden estar basadas en diversos formalismos, como por ejemplo, las máquinas de Turing, las redes de Petri o los modelos algebraicos, y se caracterizan porque son ejecutables. Por otro lado la especificación de propiedades abstractas de un sistema se realiza utilizando un formalismo lógico, que puede ser bien una lógica sencilla como la lógica proposicional, o bien pueden utilizarse lógicas más complejas pero que permitan expresar propiedades sobre el avance en el cómputo de un sistema o sobre partes del sistema, como las lógicas modales y temporales. Ambos niveles de especificación se complementan y pueden ser utilizados conjuntamente como ocurre por ejemplo con la especificación de álgebras de procesos con CCS y la lógica de Hennessy-Milner [HM80, HM85, Sti01] o con el modelo de computación de UNITY y su lógica de especificación [CM88].

Nuestro objetivo en esta tesis es proponer una metodología de especificación de sistemas que cubra ambos niveles de especificación mediante el uso de un marco matemático uniforme y que resulte adecuada para la especificación de sistemas concurrentes y en particular sistemas con configuración orientada a objetos dado el amplio uso dado a estos sistemas actualmente. El marco utilizado será el proporcionado por la lógica de reescri-

tura y su implementación vía el metalenguaje Maude. En esta línea la especificación de primer nivel se realizará directamente en el propio lenguaje Maude, mientras que para realizar la especificación de segundo nivel definiremos una lógica modal adecuada para probar propiedades de sistemas especificados en Maude, en la cual las transiciones definidas por las reglas de reescritura se capturan como acciones en la lógica. La lógica definida puede utilizarse además mediante la definición de la interfaz apropiada para demostrar propiedades especificadas en otras lógicas temporales o modales.

1.1. Contexto

1.1.1. Sistemas orientados a objetos

La orientación a objetos está ampliamente aceptada en el desarrollo del software: existen lenguajes de programación orientaos a objetos, bases de datos orientadas a objetos y métodos de desarrollo de programas. Entre estos últimos existen multitud de métodos de análisis, modelado y diseño orientados a objetos, pero la mayor parte de ellos, aunque útiles, son informales, o como mucho semi-formales.

Existen, además, varias aproximaciones a la especificación formal de sistemas orientados a objetos. Desde un punto de vista orientado a objetos, los sistemas computacionales son colecciones dinámicas de objetos autónomos que interaccionan unos con otros por medio de mensajes. Los objetos son autónomos en el sentido de que cada objeto encapsula todas las características necesarias para actuar como un agente de cómputo independiente. Existen, por lo tanto, dos niveles de especificación en los sistemas orientados a objetos que deben ser abordados: por una parte, la especificación de cada objeto individual y por otra parte la especificación de sistemas completos de objetos. Fiadeiro y Maibaum [FM91c, FM91a] proponen especificar los objetos como teorías en una lógica modal, donde los operadores modales se utilizan para describir los efectos de los eventos en los atributos de los objetos, mientras que las restricciones y requisitos para la ocurrencia de los eventos se describen mediante el uso de dos predicados *deónticos*. La lógica deóntica es adecuada para describir el comportamiento de entidades dinámicas y permite probar propiedades de seguridad y de vivacidad de los objetos. Por otra parte, la especificación del sistema global se consigue componiendo la especificación de sus componentes mediante morfismos entre teorías.

En una línea semejante se encuentra el trabajo realizado por Ehrich, A. Sernadas y C. Sernadas, entre otros, para dotar de fundamentos semánticos a la familia de lenguajes Oblog y a los lenguajes académicos relacionados con ella como Troll [Ehr99]. Para la especificación de los objetos utilizan una lógica temporal lineal que les permite capturar los aspectos dinámicos inherentes a los objetos. Sin embargo para especificar los sistemas globales extienden esta lógica a una lógica temporal distribuida.

Existen otras aproximaciones a la especificación formal de objetos. Cabe citar, por su relación con los métodos de Ehrich, A. Sernadas y C. Sernadas el lenguaje Foops [GM87] basado en OBJ3, y por lo tanto en lógica ecuacional, y el lenguaje Maude basado en lógica de reescritura por ser el formalismo utilizado en esta tesis.

1.1.2. La lógica de reescritura y el lenguaje Maude

La lógica de reescritura fue propuesta por Meseguer en 1990 [Mes90a, Mes92] como marco de unificación de modelos de computación concurrentes. Desde entonces investigadores de todo el mundo han llevado a cabo una gran cantidad de trabajo que ha contribuido al desarrollo de diversos aspectos de la lógica y a sus aplicaciones en distintas áreas de las ciencias de la computación [MOM02, Mes98b, Mes96, KK98, Fut00].

La lógica de reescritura es una lógica para representar sistemas concurrentes que tienen estados y evolucionan por medio de transiciones. Es una lógica de cambio en la cual los estados distribuidos del sistema se definen como estructuras de datos axiomatizadas algebraicamente, y los cambios locales elementales que pueden ocurrir concurrentemente en un sistema se axiomatizan como reglas de reescritura. Las reglas de reescritura corresponden a patrones locales, los cuales, cuando están presentes en el estado de un sistema, pueden convertirse en otros patrones.

La lógica de reescritura puede usarse directamente como un lenguaje de amplio espectro que soporta la especificación, el prototipado rápido y la programación de sistemas concurrentes por medio del lenguaje de especificación Maude. Este lenguaje fue desarrollado a finales de los años noventa en el Computer Science Laboratory de SRI International (EE.UU.) por un grupo de investigadores dirigidos por el propio Meseguer [CDE+99a, CDE+00a]. El lenguaje está fuertemente influenciado por el lenguaje ecuacional OBJ3 [GWM+00], el cual puede considerarse como un sublenguaje funcional de Maude. Sin embargo, Maude utiliza a nivel ecuacional la lógica ecuacional de pertenencia [Mes98a], que extiende la lógica ecuacional de tipos ordenados de OBJ3. Además, Maude integra la programación ecuacional y orientada a objetos de una forma satisfactoria. Su base lógica facilita una definición clara de la semántica orientada a objetos y hace de él una buena elección para la especificación formal de sistemas orientados a objetos.

Existen otros dos lenguajes de especificación basados en la lógica de reescritura: ELAN [BKKM02], desarrollado en Francia, y CafeOBJ [DF02], desarrollado en Japón. Una característica importante de Maude frente a estos dos lenguajes, especialmente interesante para nuestro trabajo, es la forma en que explota el hecho de que la lógica de reescritura es reflexiva, realizando una implementación muy eficiente de la reflexión. Esta implementación facilita su uso sistemático, y permite además la definición de lenguajes de estrategias para controlar el proceso de reescritura, en el mismo lenguaje.

1.1.3. Reflexión

La reflexión entendida como la capacidad de un sistema de representarse a sí mismo; es una propiedad muy útil y potente utilizada dentro de las ciencias de la computación en temas como los lenguajes de programación, demostradores de teoremas, compilación, entornos de programación, sistemas operativos y bases de datos.

La lógica de reescritura es reflexiva, pero lo que realmente diferencia al lenguaje Maude de otros formalismos reflexivos es que la reflexividad se puede utilizar de manera conveniente, gracias a la implementación realizada (ver Sección 2.2.5). Prácticamente desde que se comenzó a desarrollar el lenguaje Maude una parte de la investigación se ha centra-

do en el estudio de propiedades reflexivas de la lógica de reescritura y sus aplicaciones [Cla00, CDE+98]. Puesto que la reflexión permite a un sistema acceder a su propio metanivel, en el caso de la lógica de reescritura esta proporciona un mecanismo potente para controlar el proceso de reescritura. En este sentido, se han propuesto algunos *lenguajes de estrategias* generales en Maude [CM96b, CELM96, Cla00] para definir estrategias adecuadas para controlar la reescritura. La cuestión importante es que, gracias a la reflexión, estos lenguajes están basados en la reescritura y su semántica e implementación están descritos en la misma lógica. De esta forma, el control no es un añadido extra-lógico del lenguaje sino que permanece dentro de la lógica.

La reflexión incrementa la modularidad de las especificaciones, permitiendo formular especificaciones elegantes de los sistemas. En esta línea Clavel muestra en [Cla00] cómo utilizar las propiedades reflexivas de Maude para definir y ejecutar representaciones de una lógica determinada en el marco lógico de la lógica de reescritura; cómo definir y ejecutar diferentes lenguajes y modelos de computación dentro del marco semántico de la lógica de reescritura utilizando estrategias; y un método general de construcción de demostradores de teoremas.

1.1.4. Lógicas modales

La lógica modal es la lógica de la posibilidad y de la necesidad. Esta lógica es especialmente adecuada para razonar en situaciones en las que el valor de verdad de una afirmación no es fijo, sino que depende del estado en el que se evalúa.

Sin embargo, lo que se entiende por lógica modal no es una única lógica sino un conjunto de ellas. Esta clase de lógicas fue desarrollada originariamente por filósofos para estudiar diferentes modos de verdad; por ejemplo, una cierta aserción puede ser falsa en este mundo pero puede ser cierta en un mundo diferente. Pero fueron los trabajos de S. Kripke, A. Prior y J. Hintikka, entre otros, los que a principios de los años 60 dieron fama a esta clase de lógicas al aplicarlas a la computación. Una buena referencia histórica de estas lógicas se puede encontrar en [LS77, BS84].

Aplicada a las ciencias de la computación, la lógica modal se ocupa de la relación entre el estado presente y otros estados posibles del cómputo. Por lo tanto el universo de una lógica modal está formado por un conjunto de estados y una relación de accesibilidad entre ellos. Esta relación puede ser una relación temporal *antes/después*, lo que permite hablar de estados pasados y futuros en relación a la ejecución de las instrucciones del programa, o bien puede ser una relación espacial *ser parte de*, lo que nos permite hablar de estados y partes de estados.

Hay dos formas de razonar en una lógica: por un lado, el denominado *nivel no interpretado* que permite expresar propiedades en la lógica que son independientes del dominio de interpretación, esto es, no se realiza ninguna suposición sobre las propiedades de la estructura del sistema; y por otro lado, el denominado *nivel interpretado* que utiliza la lógica para razonar acerca de los cómputos sobre un dominio particular. En general, en este caso el conjunto de estados está dado por el conjunto de posibles ejecuciones del sistema, y la relación de accesibilidad proporciona los estados accesibles desde uno dado al ejecutar una instrucción (o conjunto de instrucciones) del programa o al descomponer un

estado en partes.

La lógica modal más básica es la lógica modal proposicional, la cual extiende la lógica proposicional con dos operadores: el operador de necesidad, $\Box$, y el operador de posibilidad, $\Diamond$. De esta forma, si φ es una propiedad de un sistema, $\Box\varphi$ es la propiedad que afirma que φ *se cumple en todos los estados posibles*. El significado dado a los estados posibles depende de la definición de la relación de accesibilidad; así si la relación es temporal $\Box\varphi$ significa en todos los estados futuros o bien en todos los estados pasados, mientras que si la relación es espacial, la propiedad se leerá como en todas las partes del sistema.

Si en lugar de aumentar la lógica proposicional con una modalidad, le añadimos un conjunto de modalidades indizado sobre un conjunto A obtenemos la lógica multimodal [Gol92, HKT00]. El operador de necesidad se expresa entonces como $[a]$ y el operador de posibilidad como $\langle a \rangle$ con $a \in A$.

Las lógicas multimodales parecieron en un principio demasiado generales para ser aplicadas a la especificación formal de programas, por lo que se desarrolló la *lógica dinámica* como una extensión de la lógica multimodal en la cual los operadores modales están parametrizados por los programas [HKT00]. Entonces, si α es un programa, la propiedad $[\alpha]\varphi$ afirma que si el programa α termina, lo hará en un estado que satisfará la propiedad φ, mientras que la propiedad $\langle\alpha\rangle\varphi$ afirma que es posible ejecutar α y que termine en un estado que satisfaga φ. Sin embargo, esta lógica solo es apropiada para la especificación de programas terminantes, al igual que ocurría con las especificaciones basadas en aserciones, aunque esta lógica es más flexible y expresiva que aquellas.

Buscando una lógica más potente que la lógica dinámica, se derivó de ésta el *μ-cálculo*, que utiliza además de los operadores típicos $[_]$ y $\langle_\rangle$, un operador de punto fijo sobre fórmulas que captura las nociones de iteración y recursión [Koz83].

Otra lógica multimodal, pero aplicable en este caso a programas que no terminan, y por ello más cercana a nuestro trabajo, es la lógica de Hennessy-Milner que se utiliza para definir capacidades locales de procesos en CCS [HM80, HM85, Sti96]. En esta lógica el conjunto de modalidades está indizado por el conjunto de acciones de los procesos.

Algunas lógicas modales/temporales más recientes se aplican a la especificación de sistemas desarrollados en entornos distribuidos [ECSD98]. Destacan los trabajos sobre lógicas de especificación para sistemas concurrentes orientados a objetos realizados por Fiadeiro y Maibaum [FM91b, FM91a, FM91c, FM92] que cubren tanto el ámbito local al objeto como el ámbito global del sistema. En lo concerniente al ámbito local al objeto describen el efecto de los eventos en los atributos y las restricciones y requisitos para la ocurrencia de los eventos. En cuanto al ámbito global lo consiguen componiendo las especificaciones locales. En el marco de la lógica de reescritura, este enfoque ha sido aplicado en un trabajo de Denker [Den98] para la especificación de sistemas distribuidos orientados a objetos. En este mismo contexto de sistemas orientados a objetos especificados en Maude, los trabajos de Lechner [Lec97, Lec98] utilizan una variante del μ-cálculo.

1.1.5. Operadores espaciales

La primera interpretación dada al operador modal al aplicar la lógica modal a la especificación de sistemas computacionales fue la de una relación de tiempo, esto es, cómo se modifica el sistema al avanzar el cómputo. Sin embargo, al aplicar la lógica modal a la definición de propiedades de sistemas distribuidos, aparece una nueva interpretación del operador modal, relacionada con el espacio, esto es, el operador modal se utiliza para relacionar las propiedades del sistema con las propiedades de sus partes.

Una de las lógicas modales más conocidas que utilizan este tipo de operador modal más conocida es la desarrollada por Cardelli y Gordon [CG98, CG00]. El modelo que utilizan es el cálculo de ambientes, un cálculo de procesos en el que los procesos residen en una jerarquía de localidades y pueden modificarlas. La estructura espacial de los estados es fija, y está representada por árboles no ordenados con arcos etiquetados, en los que las etiquetas de los arcos se corresponden con nombres de sublocalidades y los subárboles se corresponden con sublocalidades.

Por el contrario, en la lógica que se define en este trabajo, la configuración del sistema no está fija, sino que se define mediante las operaciones que especifican la estructura del sistema y que son dadas por el usuario. Además, en la lógica de Cardelli y Gordon el conjunto de operadores lógicos está fijado (como en la lógica de primer orden), mientras que en nuestra lógica los operadores son definidos por el usuario (como en la lógica ecuacional o en la propia lógica de reescritura).

La lógica de ambientes proporciona tres operadores: $n[\varphi]$, que representa un paso en el espacio, lo que permite hablar sobre un lugar dentro de n; $\diamond\varphi$, que permite hablar sobre un número arbitrario de pasos en el espacio, y su operador dual $\Box\varphi$, que permite hablar sobre cualquier sublocalidad. Estos operadores son semejantes a los de la lógica temporal lineal que se explica a continuación. Un aspecto central en esta lógica modal es el nombre de los ambientes. Cada ambiente tiene un nombre que se utiliza para controlar el acceso (entrada de procesos, salida, comunicaciones, etc.).

El trabajo anterior ha sido adaptado por Caires y Cardelli [CC01, CC02] con el fin de considerar propiedades espaciales que cubren si un sistema está compuesto de dos o más subsistemas identificables y sobre si un sistema restringe el uso de ciertos recursos a ciertos subsistemas. En este caso no utilizan los operadores espaciales básicos sino que definen un operador de punto fijo máximo, semejante al utilizado en el μ-cálculo y simplifican el cálculo de procesos básico y utilizan el π-cálculo asíncrono.

1.1.6. Lógicas temporales

Algunos autores, como Fisher y Barringer derivan las lógicas temporales de la lógica de tiempo [FB86], desarrollada como su nombre indica para modelar el uso de tiempos en el lenguaje natural. Otros, como Emerson [Eme90], derivan la lógica temporal a partir de la lógica modal, siendo los operadores temporales un tipo de modalidad que nos permite razonar sobre cómo se modifican los valores de verdad de ciertas aserciones a lo largo del tiempo.

En ambos casos, los operadores temporales básicos son: *alguna vez*, para expresar que

una aserción es cierta si existe algún momento futuro en el cual es cierta; *siempre*, para expresar que una aserción es cierta si lo es en todos los momentos futuros; y *siguiente*, para razonar sobre el momento siguiente en la secuencia de estados. Esta lógica es apropiada para razonar acerca de la ejecución detallada de un programa, aunque no es apropiada para razonar sobre múltiples programas, como la lógica dinámica.

Como ocurre con las lógicas modales, existen también muchas lógicas temporales. Estas pueden clasificarse siguiendo varios criterios [Eme90]: lógica proposicional o de primer orden, dependiendo de la lógica sobre la que se está definiendo la lógica temporal; lógica global, en la que los operadores se interpretan en un único universo, o lógica composicional, que permite expresar propiedades de diversos universos; lógica de tiempo ramificado, en la que cada momento de tiempo puede estar seguido de distintos momentos que representan diferentes futuros posibles, y lógica de tiempo lineal, en la que cada momento tiene un único momento siguiente; lógica de puntos, en la que los operadores temporales se evalúan en algún punto del tiempo, y lógica de intervalos, en la que los operadores se evalúan en intervalos de tiempo; lógica de tiempo discreto, en la que el momento presente se corresponde con el estado actual del programa y el momento siguiente se corresponde con el estado sucesor del programa, y la lógica de tiempo continuo, en la que la estructura de tiempo es densa; y por último, la lógica de tiempo pasado frente a la lógica de tiempo futuro. Añadir a la lógica operadores de tiempo pasado no incrementa la capacidad expresiva de la lógica en aplicaciones informáticas ya que la ejecución de los programas siempre tiene un punto de comienzo; sin embargo, el uso de estos operadores puede resultar muy útil para la formulación de propiedades de forma más natural.

Si se consideran cuantificadores sobre los operadores temporales, entonces la lógica se denomina *lógica temporal cuantificada* [KP95]. Este tipo de lógicas son más expresivas que las lógicas temporales sin cuantificar, esto es, permiten expresar propiedades que no se pueden expresar en las lógicas temporales sin cuantificar. Sin embargo la lógica no es completa, aunque pueden imponerse restricciones en la interacción de los cuantificadores con los operadores temporales de forma que la lógica resultante sí lo sea.

Los tipos de lógicas temporales que han sido más estudiados son las lógicas discretas lineales y las lógicas discretas de tiempo ramificado. Entre las primeras destacan los trabajos de Manna y Pnueli [MP92, MP95] sobre lógicas de tiempo futuro y pasado. Esta lógica considera básicamente cuatro operadores de tiempo futuro: $\bigcirc$, $\Diamond$, $\Box$, y $\mathcal{U}$, con el significado de *siguiente*, *alguna vez*, *siempre* y *hasta que*, y los correspondiente operadores de tiempo pasado.

Por otro lado, en las lógicas discretas de tiempo ramificado cabe citar los trabajos de Emerson y Clarke sobre la lógica CTL (*Computation Tree Logic)* [CE81] que extiende la lógica UB (*Unified Branching Time Logic*) de Ben-Ari, Manna y Pnueli [BAPM83] añadiéndole un operador *hasta que*, y la extension de CTL por Emerson y Halpern dando lugar a la lógica CTL* [EH86, EH85]. En este tipo de lógicas se consideran además de los operadores básicos de las lógicas temporales lineales, denominados en este caso X, G y F, unos operadores de cuantificación sobre las ramas del cómputo: A y E. De esta forma AGφ expresa que la propiedad φ se cumple en todas las ramas del cómputo (operador A) y en todos los estados (operador G), mientras que EGφ expresa que la propiedad se cumple en una rama del cómputo (operador E) y en todos los estados de esta rama.

Los demás operadores utilizados por estas lógicas son:

- AX : en todas las ramas en el siguiente estado,
- EX : en una rama en el siguiente estado,
- AF : en todas las ramas en algún estado,
- EF : en alguna rama en algún estado,
- A_U_ : en todas las ramas se cumple una propiedad hasta que se cumple otra,
- E_U_ : en alguna rama se cumple una propiedad hasta que se cumple otra.

Se utiliza también una versión débil de los dos últimos operadores, A_W_ y E_W_, en la que no se pide que la segunda propiedad llegue necesariamente a cumplirse. Equivalencias e implicaciones entre los diversos operadores, tanto de la lógica temporal lineal como de la ramificada pueden encontrarse en [Eme90].

Ambas lógicas temporales no son comparables desde el punto de vista de la expresividad de la lógica ya que utilizan diferentes modelos temporales. Cada una tiene sus ventajas y desventajas y el debate sobre sus respectivos méritos continua abierto en la actualidad [Var01]. La selección entre una lógica de tiempo lineal o una lógica de tiempo ramificado debe hacerse por lo tanto en base al tipo de propiedades que se desean probar, siendo la segunda más apropiada para sistemas no deterministas.

1.1.7. Clasificación de propiedades

Las propiedades que pueden expresarse en una lógica temporal se clasifican generalmente en propiedades de *seguridad*, formuladas con el operador temporal $\Box$, y propiedades de *vivacidad*, formuladas con el operador temporal $\Diamond$. Las primeras se utilizan principalmente para expresar propiedades invariantes sobre todos los estados del cómputo, esto es, *algo malo no ocurre*, mientras que las segundas se utilizan para asegurar que algún evento ocurrirá, esto es, *algo bueno ha de ocurrir*.

Esta clasificación ha sido refinada y formalizada por Manna y Pnueli en [MP92] para la lógica temporal lineal atendiendo a la interacción de varios operadores temporales. Cada clase de propiedades considerada se caracteriza por un esquema. Las dos clases básicas, propiedades de seguridad y propiedades de garantía, se corresponden aproximadamente con las propiedades de seguridad y vivacidad expresadas anteriormente, y se caracterizan por los esquemas $\Box\varphi$ y $\Diamond\varphi$ respectivamente. Ambas clases son disjuntas salvo por la fórmula trivial *true*.

Algunas propiedades, sin embargo, no pueden ser expresadas utilizando únicamente alguno de los dos esquemas anteriores, es necesario por lo tanto utilizar una combinación lógica de ellos. Se definen entonces las siguientes clases:

- propiedades de *obligación*, utilizadas para expresar que o bien la propiedad φ se cumple en todos los estados del cómputo, o bien la propiedad ψ se cumple en algún

estado. Estas propiedades están representadas por el esquema $\bigwedge_{i=1}^{n}(\Box\varphi_i \vee \Diamond\psi_i)$. Aunque quizá resulta más intuitivo el esquema equivalente $\bigwedge_{i=1}^{n}(\Diamond\varphi_i \rightarrow \Diamond\psi_i)$, esto es, si en algún momento se satisface una propiedad entonces en algún momento posterior se satisfará otra. Esta clase incluye a la clase de propiedades de seguridad y de garantía.

- propiedades de *respuesta*, utilizadas para expresar que existen infinitas posiciones en el cómputo que verifican la propiedad φ. Estas propiedades están representadas por el esquema $\Box\Diamond\varphi$. Una propiedad muy importante perteneciente a esta clase es la propiedad de *justicia*, la cual, referida, por ejemplo, a las transiciones de un sistema expresa que o bien la transición no puede tener lugar un número infinito de veces o bien tendrá lugar un número infinito de veces. Esta clase incluye a la clase de propiedades de obligación.

- propiedades *persistentes*, utilizadas para expresar que todos los estados a partir de un momento dado cumplirán una determinada propiedad φ. Estas propiedades están representadas por el esquema $\Diamond\Box\varphi$. Esta clase incluye a la clase de propiedades de obligación y es una clase dual a la clase de propiedades de respuesta.

- propiedades *reactivas*, utilizadas para expresar que si la propiedad φ se satisface en infinitos estados del cómputo, entonces la propiedad ψ debe satisfacerse también en infinitos estados. Estas propiedades están representadas mediante el esquema $\bigwedge_{i=1}^{n}(\Box\Diamond\varphi_i \vee \Diamond\Box\psi_i)$. Un tipo importante de propiedades que pertenecen a esta clase son las propiedades que expresan *compasión*, esto es, si una transición puede realizarse en infinitos estados, entonces se realizará infinitas veces. Esta clase incluye a las clases de propiedades de respuesta y de propiedades persistentes.

Dentro de esta clasificación están consideradas todas las posibles propiedades temporales que pueden expresarse sin utilizar cuantificadores. En este sentido Manna y Pnueli prueban en [MP92, página 296] que toda fórmula temporal libre de cuantificadores es equivalente a una fórmula reactiva.

Esta clasificación de propiedades, aunque ha sido realizada para una lógica temporal lineal, podría aplicarse a una lógica de tiempo ramificado, considerándola bien en una única rama del cómputo o bien en todas las ramas.

1.2. Decisiones de diseño de la lógica modal

Como ya se ha mencionado, uno de nuestros objetivos es definir mecanismos lógicos que soporten el desarrollo de sistemas software usando la lógica de reescritura. La idea es que estos mecanismos lógicos deben permitir definir y probar propiedades sobre los cómputos generados por una teoría de reescritura. Al elegir nuestra lógica tuvimos dos opciones principales:

1. Elegir una clase específica de configuraciones de estado dadas por la signatura de una teoría de reescritura, y desarrollar una lógica para estas configuraciones. Por ejem-

plo, podríamos haber elegido restringir nuestra lógica a configuraciones de estado orientadas a objetos formadas por multiconjuntos de objetos y mensajes.

2. Intentar ser lo más generales posible, es decir, realizar los menos compromisos posibles sobre propiedades específicas del estado como sea posible.

Una elección natural sobre la primera opción habrían sido las configuraciones multiconjunto orientadas a objetos, utilizadas por Grit Denker en sus trabajos sobre una lógica modal distribuida localmente [Den98]. Existe también el trabajo previo de Ulrike Lechner sobre el uso del μ-cálculo para especificaciones orientadas a objetos en Maude [Lec97]. En ambos casos, lógicas ya existentes se adaptan a las configuraciones orientadas a objetos de Maude, estudiando las relaciones de refinamiento entre las especificaciones a ambos niveles.

Nosotros hemos favorecido la segunda opción, motivados por el deseo de investigar cuánto puede decirse a nivel de especificación sobre los diferentes modelos de concurrencia de las teorías de reescritura. Una vez tomada esta decisión, decidimos desarrollar y explorar una nueva lógica modal de acciones en la cual solo las reglas de reescritura son consideradas como acciones, y definir otra lógica para los cómputos y las correspondientes interfaces con ella. Una situación similar aparece en la lógica temporal de Stirling definida sobre los cómputos generados por un sistema de transiciones [Sti92], o en la relación entre la lógica temporal y la lógica deóntica desarrollada por Fiadeiro y Maibaum [FM91b].

El principal beneficio de nuestro método es que no nos restringimos a un tipo especial de configuración de estado como se hace en [Den98, Lec97]. Más aún, comparado con la lógica modal distribuida localmente del trabajo de Denker, estamos interesados en propiedades globales del sistema, orientado a objetos o no, en lugar de propiedades vistas por un objeto local; y comparado con el trabajo de Lechner, nuestras fórmulas son más generales que simples proposiciones afirmando la presencia de mensajes y/o objetos. Esta es la principal razón que motiva la introducción de las observaciones para expresar propiedades más interesantes del sistema.

1.3. Aportaciones de la tesis

El trabajo mostrado en esta tesis proporciona un marco matemático de especificación de sistemas concurrentes basado en lógica de reescritura. Este marco de especificación es especialmente adecuado para la especificación de sistemas orientados a objetos al permitir realizar la especificación del modelo en el lenguaje Maude y la especificación de propiedades abstractas en una lógica modal o temporal capturando de esta forma las propiedades dinámicas del sistema.

Los Capítulos 2 y 3, están dedicados a las especificaciones en el lenguaje Maude. En el primero de ellos se presentan los aspectos básicos de la lógica de reescritura y del lenguaje Maude que serán utilizados en la tesis. Cabe destacar el uso de las facilidades proporcionadas por el lenguaje para la especificación de sistemas orientados a objetos y de las técnicas de reflexión. En el segundo capítulo se desarrolla una especificación en Maude de un modelo orientado a objetos para redes de telecomunicación de banda ancha que

muestra el poder del lenguaje para especificar este tipo de sistemas, y en particular la conocida relación de herencia y otras relaciones entre objetos como la relación de contenido o las relaciones explícitas de grupo (ser-miembro-de, cliente-servidor, etc.). Nos hemos centrado en el impacto de estas relaciones en los procesos de creación y borrado de objetos, proporcionando un método general para su especificación. En particular, proponemos utilizar subobjetos contenidos en otros objetos [Mes93a] para especificar la relación de contenido y comparamos esta aproximación con el uso del identificador del objeto para representar al objeto contenido. La aplicación elegida de modelar redes de telecomunicaciones de banda ancha es una buena elección para mostrar las relaciones de contenido y las relaciones explícitas de grupo porque estas aparecen de forma natural entre los objetos de las diferentes capas del modelo de red y entre los objetos de la misma capa.

La reflexión se utiliza en la especificación de la red de telecomunicaciones para explotar mejor los recursos en los diferentes niveles. Combinamos ideas del campo de la reflexión lógica con ideas provenientes del campo de la reflexión orientada a objetos [Mes93a] mediante el uso de un *mediador*, un metaobjeto que vive en el metanivel y que tiene acceso a la configuración de la red, para su gestión. La aplicación de las propiedades reflexivas de la lógica de reescritura se ilustra mediante un proceso que modifica la demanda de un servicio entre dos nodos en una red. El lenguaje de estrategias usado para controlar el proceso está basado en el lenguaje presentado en [CELM96], adaptando la sintaxis a la última versión del lenguaje disponible en el momento de escribir esta tesis [CDE+00b]. A continuación se compara la especificación obtenida sin utilizar reflexión con la especificación utilizando reflexión y se muestra que la segunda especificación es más sencilla que la primera.

En los Capítulos 4 y 5 se tratan las especificaciones, como expresión de las propiedades que debe cumplir el sistema, formuladas en lógicas modales y temporales. Para ello se desarrolla una lógica modal, llamada *Verification Logic for Rewriting logic* (VLRL), que captura las reglas de reescritura como acciones. Esta lógica proporciona además de la modalidad temporal una modalidad espacial[1] que permite definir propiedades sobre partes del sistema y relacionarlas con propiedades del sistema completo así como definir propiedades sobre acciones realizadas en partes del sistema.

En esta tesis nos centramos en las propiedades de seguridad aplicadas sobre una rama o sobre todas las ramas posibles del cómputo, ya que esta clase de propiedades incluye muchas de las propiedades fundamentales de los sistemas concurrentes como la ausencia de bloqueo o la exclusión mutua en secciones críticas, además de propiedades invariantes que garantizan la corrección parcial del sistema. Para formular este tipo de propiedades definimos, por encima de la lógica modal, una lógica temporal de tiempo ramificado con tres operadores básicos: AX, que expresa que una propiedad se cumple en todos los estados sucesores posibles, A(_W_), que expresa que una propiedad se cumplirá hasta que se cumpla otra o bien se cumplirá siempre, y EX, que expresa que una propiedad se cumple en algún estado sucesor. Para cada uno de estos operadores se proporciona la regla de inferencia

[1] En trabajos previos a esta tesis se denominó a estos operadores *topológicos* atendiendo a que se utilizaban para definir propiedades sobre la topología del sistema. Sin embargo este término puede resultar confuso y por ello se ha preferido pasar a denominar a estos operadores *espaciales*, ya que este es el término que se está utilizando recientemente para ellos en la literatura.

que nos permiten derivar las propiedades temporales a partir de propiedades en VLRL, probándose que son correctas y completas. El uso de una lógica temporal nos permite especificar, en particular, las propiedades dinámicas de los sistemas orientados a objetos.

Los sistemas seleccionados en el Capítulo 5 muestran la aplicación de la lógica VLRL y de la lógica temporal seleccionada en sistemas con configuración orientada a objetos. La configuración orientada a objetos es especialmente adecuada para las especificaciones en VLRL ya que la noción de estado viene dada por la propia configuración del sistema o por una restricción de ella, y los elementos a observar son características de los objetos, pudiendo ser una característica de un grupo de ellos o bien una característica de un objeto en particular.

La lógica VLRL nos permite también definir de forma natural propiedades espaciales, permitiendo de esta forma estudiar el uso de las modalidades espaciales y de acción, en particular cómo derivar propiedades espaciales básicas a partir de la sintaxis de los programas y cómo utilizar a continuación estas propiedades en los pasos de verificación de las propiedades temporales. El caso de la derivación de propiedades espaciales básicas se ve también beneficiado por el uso de configuraciones orientadas a objetos ya que varios de los métodos definidos tratan observaciones sobre elementos únicos del sistema, y estos quedan bien definidos en las configuraciones orientadas a objetos.

La lógica VLRL permite asimismo la utilización de variables en la formulación de propiedades, así como el uso de familias de observaciones, esto es, observaciones realizadas sobre un conjunto de elementos del sistema.

Por último en el Capítulo 6 se muestra un mecanismo de verificación de propiedades de sistemas escritos en lógica de reescritura basado en técnicas inductivas. Las propiedades seleccionadas en este caso son tanto propiedades de seguridad como propiedades de vivacidad. Se ilustran estas ideas por medio de la especificación del protocolo de elección de líder del bus en serie multimedia IEEE 1394 y la verificación de su corrección: el protocolo termina y se elige un único líder.

Capítulo 2

La lógica de reescritura y el lenguaje Maude

La lógica de reescritura es una lógica para especificar sistemas concurrentes que tienen estados y evolucionan por medio de transiciones. Es una lógica *de cambio* que puede ser utilizada directamente como un lenguaje que soporta la especificación, el prototipado rápido y la programación de sistemas concurrentes [LMOM94]. Las partes dinámicas del sistema se especifican por medio de reglas de reescritura condicionales etiquetadas, mientras que la estructura y las partes estáticas se especifican por medio de ecuaciones.

La especificación se expresa en el lenguaje multiparadigma Maude, basado en dicha lógica, el cual integra la programación ecuacional y orientada a objetos de una forma satisfactoria. Su base lógica facilita una definición clara de la semántica orientada a objetos y hace de él una buena elección para la especificación formal de sistemas concurrentes orientados a objetos. El lenguaje Maude está completamente descrito en el manual [CDE+99a] y pueden encontrarse ejemplos de su utilización en el tutorial [CDE+00a].

2.1. La lógica de reescritura

En esta sección se resumen los conceptos principales de la lógica de reescritura. Para más información se puede consultar [Mes92, Mes93a].

2.1.1. Reglas de inferencia

Una *teoría de reescritura* $\mathcal{R}$ se define como una 4-tupla $\mathcal{R} = (\Sigma, E, L, R)$ donde (Σ, E) es una teoría ecuacional, L es un conjunto de etiquetas, y R es un conjunto de *reglas de reescritura* de la forma[1] $l : [t]_E \longrightarrow [t']_E$, donde $l \in L$, t y t' son Σ-términos posiblemente

[1] Para simplificar la exposición mostramos las reglas de la lógica solo para el caso de reglas de reescritura no condicionales. Sin embargo, todos los conceptos que se presentan aquí han sido extendidos en [Mes92] al caso de reglas condicionales de la forma

$$l : [t] \rightarrow [t'] \ \mathit{if} \ [u_1] \rightarrow [v_1] \wedge \ldots \wedge [u_k] \rightarrow [v_k].$$

con algunas variables, y $[t]_E$ denota la clase de equivalencia de términos t módulo las ecuaciones E.

Intuitivamente, la teoría (Σ, E) de una teoría de reescritura describe una estructura particular para los estados del sistema, y las reglas de reescritura describen qué transiciones elementales son posibles en el estado distribuido mediante transformaciones locales concurrentes. Alternativamente, se puede adoptar un punto de vista lógico y considerar las reglas de reescritura como reglas de deducción en un sistema lógico y cada paso de reescritura como una derivación lógica en un sistema formal.

Estas dos interpretaciones se pueden resumir en el siguiente diagrama:

Estado	$\leftrightarrow$	*Término*	$\leftrightarrow$	*Proposición*
Transición	$\leftrightarrow$	*Reescritura*	$\leftrightarrow$	*Deducción*
Estruc. Distribuida	$\leftrightarrow$	*Estruc. Algebraica*	$\leftrightarrow$	*Estruc. Proposicional*

Informalmente, la última línea de equivalencias expresa el hecho de que un estado puede transformarse de forma concurrente solamente en caso de que esté compuesto por partes más pequeñas que puedan transformarse de forma independiante. En la lógica de reescritura, el hecho de que un estado esté compuesto por varias partes se especifica mediante las operaciones de la signatura Σ de la teoría de reescritura $\mathcal{R}$ que axiomatiza el sistema. Desde el punto de vista lógico, estas operaciones se pueden considerar como conectivas proposicionales definibles por el usuario que definen la estructura particular de un estado.

Dada una teoría de reescritura $\mathcal{R}$, decimos que de $\mathcal{R}$ se deriva un secuente $[t] \rightarrow [t']$ y lo escribimos como $\mathcal{R} \vdash [t] \rightarrow [t']$ si y solo si $[t] \rightarrow [t']$ se puede obtener mediante la aplicación finita de las siguientes reglas de deducción donde suponemos que todos los términos están bien formados y $t(\overline{w}/\overline{x})$ denota la sustitución simultánea de w_i en el lugar de x_i en t:

1. *Reflexividad*: para cada $[t] \in T_{\Sigma,E}(X)$, $\dfrac{}{[t] \longrightarrow [t]}$,

2. *Congruencia*: para cada $f \in \Sigma_n$, $n \in \mathbb{N}$,

$$\frac{[t_1] \longrightarrow [t'_1] \quad \ldots \quad [t_n] \longrightarrow [t'_n]}{[f(t_1, \ldots, t_n)] \longrightarrow [f(t'_1, \ldots, t'_n)]} \ ,$$

3. *Reemplazamiento*[2]: para cada regla de reescritura $r : [t(\overline{x})] \longrightarrow [t'(\overline{x})]$ en R,

$$\frac{[w_1] \longrightarrow [w'_1] \quad \ldots \quad [w_n] \longrightarrow [w'_n]}{[t(\overline{w}/\overline{x})] \longrightarrow [t'(\overline{w'}/\overline{x})]} \ ,$$

4. *Transitividad*:

$$\frac{[t_1] \longrightarrow [t_2] \quad [t_2] \longrightarrow [t_3]}{[t_1] \rightarrow [t_3]} \ .$$

[2] Aquí y en el resto de la tesis, utilizamos una línea superior para abreviar secuencias de expresiones; por ejemplo, $f(\overline{x})$ denota $f(x_1, \ldots, x_n)$.

Un secuente $[t] \to [t']$ en $\mathcal{R}$ se denomina una *reescritura en un paso* si y solo si se puede derivar de $\mathcal{R}$ mediante la aplicación de las reglas (1)-(3) un número finito de veces, y aplicando al menos una vez la regla (3). Si la regla (3) se aplica exactamente una vez entonces decimos que el secuente es una *reescritura secuencial en un paso*.

2.1.2. Modelos de la lógica de reescritura

A continuación se esboza la construcción de los modelos inicial y libre para una teoría de reescritura $\mathcal{R}$ [Mes92]. Estos modelos capturan la idea intuitiva de un sistema de reescritura, en el sentido de que son sistemas cuyos estados son clases de equivalencia de términos, y cuyas transiciones son reescrituras concurrentes que utilizan las reglas de $\mathcal{R}$. La estructura de estos sistemas es una categoría en la que los objetos se corresponden con los estados, los morfismos con las transiciones, y la composición de morfismos con la composición secuencial.

Un modelo para una teoría de reescritura $\mathcal{R} = (\Sigma, E, L, R)$, en la que suponemos que las reglas diferentes tiene etiquetas diferentes, es una categoría $\mathcal{T}_{\mathcal{R}}(X)$ cuyos objetos son clases de equivalencia de términos $[t] \in T_{\Sigma,E}(X)$ y cuyos morfismos son clases de equivalencia de términos de prueba que representan reescrituras concurrentes. Las reglas para generar estos términos de prueba, con la especificación de sus respectivos dominios y codominios es la siguiente[3]:

1. *Identidades*: para cada $[t] \in T_{\Sigma,E}(X)$, $\dfrac{}{[t] : [t] \longrightarrow [t]}$,

2. Σ-*Estructura*: para cada $f \in \Sigma_n$, $n \in \mathbb{N}$,

$$\frac{\alpha_1 : [t_1] \longrightarrow [t'_1] \quad \ldots \quad \alpha_n : [t_n] \longrightarrow [t'_n]}{f(\alpha_1, \ldots, \alpha_n) : [f(t_1, \ldots, t_n)] \longrightarrow [f(t'_1, \ldots, t'_n)]} \ ,$$

3. *Reemplazamiento*: para cada regla de reescritura $r : [t(\overline{x})] \longrightarrow [t'(\overline{x})]$ en R,

$$\frac{\alpha_1 : [w_1] \longrightarrow [w'_1] \quad \ldots \quad \alpha_n : [w_n] \longrightarrow [w'_n]}{r(\alpha_1, \ldots, \alpha_n) : [t(\overline{w}/\overline{x})] \longrightarrow [t'(\overline{w}/\overline{x})]} \ ,$$

4. *Composición*:

$$\frac{\alpha : [t_1] \longrightarrow [t_2] \quad \beta : [t_2] \longrightarrow [t_3]}{\alpha ; \beta : [t_1] \to [t_3]} \ .$$

Las reglas coinciden con las reglas de la lógica de reescritura, salvo en el uso de los términos de prueba. Cada una de ellas define una operación diferente que toma como argumentos una serie de términos de prueba y devuelve el término de prueba resultante. Esto es, los términos de prueba forman una estructura algebraica $\mathcal{P}_{\mathcal{R}}(X)$ consistente en un grafo con nodos $T_{\Sigma,E}(X)$, con flechas identidad y con operaciones f (para cada $f \in \Sigma$), r (para cada regla de reescritura), y $_;_$ (para componer flechas). El modelo $\mathcal{T}_{\mathcal{R}}(X)$ es el cociente de $\mathcal{P}_{\mathcal{R}}(X)$ módulo las siguientes ecuaciones:

[3]En las reglas siguientes, $\alpha; \beta$ significa siempre la composición de α seguida de β.

1. *Categoría*

   a) *Asociatividad.* Para todos α, β, γ, $(\alpha; \beta); \gamma = \alpha; (\beta; \gamma)$.

   b) *Identidades.* Para cada $\alpha : [t] \to [t']$, $\alpha; [t'] = \alpha$ y $[t]; \alpha = \alpha$.

2. *Funtorialidad de la estructura algebraica.* Para cada $f \in \Sigma_n$,

   a) *Composición.* Para todos $\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_n$

   $$f(\alpha_1; \beta_1, \ldots, \alpha_n; \beta_n) = f(\alpha_1, \ldots, \alpha_n); f(\beta_1, \ldots, \beta_n).$$

   b) *Identidades.* $f([t_1], \ldots, [t_n]) = [f(t_1, \ldots, t_n)]$.

3. *Axiomas en E.* Para cada axioma $t(x_1, \ldots, x_n) = t'(x_1, \ldots, x_n)$ en E, para todos $\alpha_1, \ldots, \alpha_n$, $t(\alpha_1, \ldots, \alpha_n) = t'(\alpha_1, \ldots, \alpha_n)$.

4. *Intercambio.* Para cada regla de reescritura $r : [t(\overline{x})] \to [t'(\overline{x})]$ en R

   $$\frac{\alpha : [w_1] \to [w'_1] \ldots \alpha_n : [w_n] \to [w'_n]}{r(\overline{\alpha}) = r([\overline{w}]); t'(\overline{\alpha}) = t(\overline{\alpha}); r([\overline{w'}])}.$$

En particular, las ecuaciones 1 hacen que $\mathcal{T}_{\mathcal{R}}(X)$ sea una categoría, las ecuaciones 2 hacen que cada $f \in \Sigma$ sea un funtor, y la ecuación 3 fuerza los axiomas de E. Por último, la ecuación 4 afirma que la reescritura simultánea de un término y de sus subtérminos es equivalente a la reescritura del término seguida secuencialmente por la reescritura de sus subtérminos, así como a la reescritura de los subtérminos seguida por la reescritura del término.

Los modelos $\mathcal{T}_{\mathcal{R}} = \mathcal{T}_{\mathcal{R}}(\emptyset)$ y $\mathcal{T}_{\mathcal{R}}(X)$ son, respectivamente, el modelo inicial y el modelo libre sobre el conjunto de generadores X de una categoría general de modelos, $\mathcal{R}$**-Sys**, que puede definirse de la siguiente forma, donde $*$ denota la composición horizontal de transformaciones naturales:

Dada una teoría de reescritura $\mathcal{R} = (\Sigma, E, L, R)$, un *R-sistema* $\mathcal{S}$ es una categoría $\mathcal{S}$ junto con:

- una estructura de (Σ, E)-álgebra, dada por una familia de funtores

  $$\{f_{\mathcal{S}} : \mathcal{S}^n \to \mathcal{S} \mid f \in \Sigma_n, n \in \mathbb{N}\}$$

  que satisfacen las ecuaciones E, es decir, para todo $t(x_1, \ldots, x_n) = t'(x_1, \ldots, x_n)$ en E tenemos una identidad de funtores $t_{\mathcal{S}} = t'_{\mathcal{S}}$, donde el funtor $t_{\mathcal{S}}$ se define de forma inductiva a partir de los funtores $f_{\mathcal{S}}$ de forma obvia.

- Para cada regla de reescritura $r : [t(\overline{x})] \to [t'(\overline{x})]$ en R, una transformación natural $r_{\mathcal{S}} : t_{\mathcal{S}} \Rightarrow t'_{\mathcal{S}}$.

Un *R-homomorfismo* $F : \mathcal{S} \to \mathcal{S}'$ entre dos $\mathcal{R}$-sistemas es un funtor $F : \mathcal{S} \to \mathcal{S}'$ tal que F es un homomorfismo de Σ-álgebras, es decir, $f_{\mathcal{S}} * F = F^n * f_{\mathcal{S}'}$, para cada f en Σ_n, $n \in \mathbb{N}$, y tal que F "preserva" R, es decir, para cada regla de reescritura $r : [t(\overline{x})] \to [t'(\overline{x})]$ en R tenemos la identidad de transformaciones naturales $r_{\mathcal{S}} * F = F^n * r_{\mathcal{S}'}$, donde n es el número de variables que aparecen en la regla. Esto define una categoría $\mathcal{R}$-**Sys** de forma obvia.

La existencia de modelos iniciales y libres para el caso más general de teorías de reescritura condicionales aparece en [Mes90b], donde también se prueba que la lógica de reescritura es correcta y completa para estos modelos.

2.1.3. Características reflexivas de la lógica de reescritura

Una característica muy importante de la lógica de reescritura es que es reflexiva [Cla00, CMP02]. La reflexión permite obtener soluciones elegantes de problemas típicos de lógica y computación, como por ejemplo las aplicaciones que se presentan en [Cla00] sobre la representación de una lógica dada en el marco general proporcionado por la lógica de reescritura, la definición y ejecución de CCS en el marco general de la lógica de reescritura y la construcción de demostradores de teoremas. En esta tesis se utilizan las características reflexivas de la lógica para definir un lenguaje de estrategias que permite controlar el proceso de reescritura del sistema que se presenta en el Capítulo 3. La aplicación concreta de estrategias en Maude se explica en la Sección 2.2.5.

Al ser la lógica de reescritura *reflexiva*, existe una teoría de reescritura universal $\mathcal{U}$ con un número finito de operaciones, ecuaciones y reglas que puede simular cualquier otra teoría de reescritura $\mathcal{R}$ representable de forma finita en el sentido siguiente: dados dos términos cualquiera t, t' en $\mathcal{R}$, existen términos correspondientes $\langle \overline{\mathcal{R}}, \overline{t} \rangle$ y $\langle \overline{\mathcal{R}}, \overline{t'} \rangle$ en $\mathcal{U}$ tales que

$$\mathcal{R} \vdash t \longrightarrow t' \iff \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \longrightarrow \langle \overline{\mathcal{R}}, \overline{t'} \rangle.$$

Dado que $\mathcal{U}$ es a su vez representable obtenemos una *torre reflexiva*, con un número arbitrario de niveles de reflexión

$$\mathcal{R} \vdash t \to t' \;\Leftrightarrow\; \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \to \langle \overline{\mathcal{R}}, \overline{t'} \rangle \;\Leftrightarrow\; \mathcal{U} \vdash \langle \overline{\mathcal{U}}, \overline{\langle \overline{\mathcal{R}}, \overline{t} \rangle} \rangle \to \langle \overline{U}, \overline{\langle \overline{\mathcal{R}}, \overline{t'} \rangle} \rangle \ldots$$

Denotaremos la representación de un término del nivel objeto *ot* en el metanivel por $\overline{ot}$ (ver [CM96a, Cla00] para los detalles de la correspondiente definición).

2.1.4. Especificaciones de sistemas de tiempo real en lógica de reescritura

Los sistemas de tiempo real pueden ser especificados de una forma natural en lógica de reescritura. Una primera aproximación a este tipo de especificaciones se debe a Kosiuczenko y Wirsing [KW97] con su trabajo sobre *lógica de reescritura con tiempo*.

La segunda aproximación se debe a Olveczky y Meseguer [Ölv00], con sus trabajos sobre *teorías de reescritura de tiempo real* y es la que se aplicará en la especificación de los aspectos temporales del protocolo IEEE 1394 del Capítulo 6.

Una teoría de reescritura de tiempo real es una tupla $(\mathcal{R}, \phi, \tau)$, donde $\mathcal{R} = (\Sigma, E, L, R)$ es una teoría de reescritura tal que:

- ϕ es un morfismo de teorías ecuacionales $\phi : \mathit{TIME} \rightarrow (\Sigma, E)$, donde TIME es una teoría en la que se define el tipo *Time* como un monoide conmutativo $(\mathit{Time}, +, 0)$, con las operaciones adicionales de $\leq, <$ y la diferencia,

- El dominio de tiempo es funcional, esto es, si $\alpha : r \rightarrow r'$ es una prueba de reescritura en $\mathcal{R}$ y r es un término de tipo $\phi(\mathit{Time})$, entonces $r = r'$ y α es equivalente a la prueba identidad r,

- (Σ, E) contiene un tipo designado *State* y un tipo específico *System*, sin subtipos o supertipos y con una sola operación $\{_\} : \mathit{State} \rightarrow \mathit{System}$ que no satisface ninguna ecuación no trivial; más aún, para cualquier operación $f : s_1, \ldots, s_n \rightarrow s$ en Σ el tipo *System* no aparece entre los $s_1, \ldots, s_n$,

- τ es una asignación de un término $\tau_l(x_1, \ldots, x_n)$ de tipo $\phi(\mathit{Time})$ a cada regla de reescritura en $\mathcal{R}$ de la forma

$$[l] : u(x_1, \ldots, x_n) \rightarrow u'(x_1, \ldots, x_n) \text{ } \mathit{if} \text{ } C(x_1, \ldots, x_n) \tag{2.1}$$

  donde u y u' son términos de tipo *System*.

Las reglas de la forma (2.1) se denominan *reglas globales* y pueden ser *reglas tick* si su duración $\tau_l(x_1, \ldots, x_n)$ es diferente de $\phi(0)$ para alguna instanciación de sus variables, y *reglas instantáneas* en otro caso. Las reglas que no son de la forma (2.1) se denominan *reglas locales* ya que no actúan en el sistema completo sino solo en alguna de sus componentes y se consideran siempre como reglas instantáneas.

Las teorías de reescritura de tiempo real se pueden transformar en teorías de reescritura ordinarias añadiendo un reloj al estado. Un estado en este sistema ampliado tendrá la forma $\langle t, r \rangle$, donde t es el estado global de tipo *System* y r es un valor de tipo $\phi(\mathit{Time})$, que denota el tiempo de cómputo si en el estado inicial el reloj tenía el valor $\phi(0)$. La formalización de esta transformación puede encontrarse en [Ölv00] y no es relevante para el trabajo desarrollado en esta tesis.

Las reglas *tick* serán normalmente de la forma

$$[\mathit{tick}] : \{t\} \rightarrow^{x_r} \{t'(x_r)\} \text{ if } x_r \leq \mathit{mte}(t) \text{ and } C(t)$$

donde x_r es una variable que denota el tiempo avanzado por la regla, $\mathit{mte}(t)$ calcula el espacio de tiempo máximo permitido para asegurar las acciones críticas (acciones que deben ocurrir en un especio de tiempo determinado), y la condición $x_r \leq \mathit{mte}(t)$ asegura que el espacio de tiempo transcurrido puede temporalmente ser interrumpido para una posible aplicación de una regla no crítica, esto es una regla que modela una acción que puede ocurrir en algún momento arbitrario del tiempo.

2.2. El lenguaje Maude

El lenguaje de especificación Maude fue desarrollado a finales de los años noventa en el Computer Science Laboratory de SRI International por un grupo de investigadores dirigidos por Meseguer [CDE+02, CDE+99a, CDE+00a]. Se trata de un lenguaje lógico basado en la lógica de reescritura. El lenguaje está fuertemente influenciado por el lenguaje ecuacional OBJ3 [GKK+88, GWM+00], el cual puede considerarse como un sublenguaje funcional de Maude. Con todo, Maude utiliza a nivel ecuacional la lógica ecuacional de pertenencia [Mes98a], que extiende la lógica ecuacional de tipos ordenados de OBJ3.

Los sistemas en Maude se construyen a partir de elementos básicos llamados *módulos*.

2.2.1. Módulos funcionales

Los *módulos funcionales* se utilizan para la definición de tipos de datos algebraicos, y tienen la forma `fmod` (Σ, E) `endfm`, donde $E = E' \cup A$, siendo A un conjunto de axiomas ecuacionales de ciertos operadores de la signatura Σ y E' un conjunto de ecuaciones que deben ser Church-Rosser y terminantes módulo los axiomas de A.

Un modelo matemático de los datos y las funciones viene dado por el álgebra inicial definida por la teoría, cuyos elementos consisten en clases de equivalencia de términos básicos módulo las ecuaciones.

Estructura de un módulo funcional. El siguiente módulo se utiliza para definir los números naturales.

```
fmod BASIC-NAT is
  sort Nat .
  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .
  op _+_ : Nat Nat -> Nat .
  vars N M : Nat .
  eq 0 + N = N .
  eq s(M) + N = s(M + N) .
endfm
```

Un módulo semejante se utiliza en la especificación del modelo de red de comunicaciones desarrollado en el Capítulo 3.

Los módulos funcionales comienzan con la palabra reservada `fmod`, a continuación aparece el identificador del módulo, en este caso `BASIC-NAT`, y la palabra reservada `is`.

Después de la cabecera, se declaran los módulos importados. Por ejemplo el módulo siguiente importa el módulo en que se definen los números naturales:

```
fmod NAT+ is
  protecting BASIC-NAT .
  op _*_ : Nat Nat -> Nat .
  vars N M : Nat .
```

```
  eq 0 * N = 0 .
  eq s(M) * N = (M * N) + N .
endfm
```

Maude proporciona dos formas de importación de módulos. La forma más restrictiva va precedida de la palabra reservada `protecting` o bien su forma abreviada `pr`, seguida del identificador del módulo importado. Esta forma de importación afirma que los tipos del módulo importado no se modifican, en el sentido de que no se añaden nuevos elementos al tipo, ni las nuevas ecuaciones identifican elementos ya existentes. Los módulos pueden también ser importados utilizando la palabra reservada `including` o su abreviatura `inc`, con la que no se imponen las restricciones anteriores. Sí se imponen, sin embargo, ciertas restricciones, en el sentido de que si la teoría importada importa a su vez en modo protegido otra teoría, esta se mantiene protegida.

La declaración de tipos comienza con la palabra reservada `sort` en el caso de tratarse de un solo tipo o `sorts` en el caso de tratarse de varios tipos, seguida del identificador del tipo, en nuestro ejemplo `Nat`. Las relaciones entre los tipos se indican mediante la palabra reservada `subsort` y deben entenderse como que los datos del tipo que aparece en el lado izquierdo del símbolo de operación relacional `<` están incluidos en el tipo que se menciona a la derecha de dicho símbolo. Se permite la definición de varios subtipos en una sola declaración.

Las operaciones sobre los tipos del módulo se declaran precedidas de la palabra reservada `op` seguida del identificador de la operación y de los tipos de sus argumentos y del resultado. Los operadores pueden declararse en forma *prefija* si se declaran como una cadena de caracteres en la que no aparece el símbolo de subrayado, o bien en forma *mixfija* en la cual la posición de los argumentos en la cadena de caracteres se indica mediante un símbolo subrayado. Entre corchetes se indican los axiomas ecuacionales de los operadores que pueden ser: asociativo, mediante la palabra reservada `assoc`, conmutativo, mediante la palabra reservada `comm`, y con elemento neutro, mediante la palabra reservada `id` seguida del elemento neutro que se considere. La palabra reservada `ctor` indica que la operación es una constructora del tipo.

En la primera versión de Maude, las ecuaciones que definen las operaciones iban precedidas de una declaración de variables, dada por la palabra reservada `var` o `vars` seguida del identificador de la (o las) variables y su tipo. En la nueva versión de Maude, Maude 2.0 [CDE+00b], las variables aparecen directamente en los términos, como el identificador seguido de su tipo. Se mantiene, sin embargo, la posibilidad de hacer una declaración de variables, que se entiende en este caso como un alias, que permite utilizar en los términos únicamente el identificador de la variable, sin ser necesario especificar su tipo.

Las ecuaciones van precedidas de la palabra reservada `eq` o `ceq` si la ecuación es condicional. Todas las variables utilizadas en el lado derecho de una ecuación deben aparecer en el lado izquierdo.

Debido a que la lógica utilizada por Maude es la lógica ecuacional de pertenencia, pueden definirse en los módulos axiomas de pertenencia, introducidos por la palabra reservada `mb` o `cmb` si el axioma es condicional, en los cuales se afirma que un término tiene un cierto tipo si se satisface una condición.

Tanto en el caso de ecuaciones condicionales como en el caso de los axiomas de pertenencia condicionales, las condiciones se forman como la conjunción de ecuaciones y axiomas de pertenencia. En ambos casos las variables que aparecen en la parte derecha de la condición deben aparecer también en la parte izquierda.

En Maude 2.0 se permiten un tipo especial de ecuaciones, denominadas *ecuaciones de encaje de patrones*, en las que se permiten variables nuevas en el lado izquierdo de la ecuación. Estas ecuaciones se interpretan matemáticamente como ecuaciones ordinarias; sin embargo, en su ejecución las nuevas variables se instancian encajando el patrón del lado izquierdo de la condición con el lado derecho.

Módulos funcionales predefinidos. Maude 2.0 proporciona módulos funcionales predefinidos para la definición de los números naturales, enteros, racionales y reales en los módulos `NAT`, `INT`, `RAT` y `FLOAT`, respectivamente. La implementación de las operaciones en estos módulos se realiza de forma muy eficiente.

Se proporcionan también módulos para el tratamiento de los valores booleanos, `BOOL`, los identificadores con comilla, `QID`, y las cadenas de caracteres, `STRING`.

Por último, el módulo `META-LEVEL` proporciona las operaciones básicas para manejar la reflexión (ver Sección 2.2.5).

2.2.2. Módulos de sistema

El tipo de reescritura de los módulos funcionales es un reemplazamiento de iguales por iguales hasta que se alcanza un único valor final, ya que estos módulos deben ser terminantes y confluentes. Sin embargo, en general, el conjunto de reglas de reescritura no tiene por qué ser ni terminante ni confluente. Para especificar este tipo de sistemas se utilizan los *módulos de sistema*, que son los módulos más generales que se permiten en Maude. En estos módulos, los términos t dejan de ser expresiones funcionales para ser *estados* del sistema, y las reglas de reescritura $t \to t'$ no se interpretan como ecuaciones, sino como transiciones locales de estado, las cuales afirman que si una parte del estado del sistema se corresponde con un cierto patrón dado por el término t entonces esta parte del sistema puede cambiar a t'. Más aún, las transiciones pueden ocurrir concurrentemente, siempre y cuando la parte del sistema que tratan no se solape con otra transición.

Estos módulos especifican el modelo inicial $\mathcal{T}_{\mathcal{R}}$ de una teoría de reescritura $\mathcal{R}$. Este modelo inicial es un sistema de transiciones cuyos estados son clases de equivalencia $[t]$ de términos básicos módulo las ecuaciones E de $\mathcal{R}$, y cuyas transiciones son pruebas $\alpha : [t] \to [t']$ en lógica de reescritura, esto es, cómputos concurrentes del sistema (véase la Sección 2.1.2).

Estructura de un módulo de sistema. Estos módulos se declaran como `mod` $\mathcal{R}$ `endm` siendo $\mathcal{R}$ una teoría de reescritura, cuyas ecuaciones E deben cumplir las mismas restricciones que en el caso de los módulos funcionales.

Por ejemplo, el siguiente módulo especifica una máquina expendedora que permite comprar bizcochos y manzanas [MOM99]. Un bizcocho vale un dólar y una manzana tres

cuartos. La máquina solo acepta dólares y devuelve un cuarto cuando el usuario compra una manzana. También ofrece la posibilidad de cambiar cuatro cuartos en un dólar. Este módulo se utiliza en la Sección 4.2.2 para la prueba de propiedades modales y temporales con la lógica VLRL.

```
mod VENDING-MACHINE is
  sort State .
  ops $ q a c : -> State .
  op __ : State State -> State [assoc comm] .
  rl [buy-c] : $ => c .
  rl [buy-a] : $ => a q .
  rl [change] : q q q q => $ .
endm
```

La declaración de los módulos importados, los tipos y subtipos, y las operaciones se realiza igual que en los módulos funcionales. La declaración de las reglas de reescritura, se introduce por la palabra reservada `rl`, o `crl` en el caso de reglas condicinales, seguida de una etiqueta que identifica la regla entre corchetes y a continuación los términos t y t' de la regla de reescritura separados por el símbolo `=>`, en último lugar se escriben las condiciones de la regla cuando esta es condicional.

Las reglas de reescritura pueden tener variables en su parte derecha que no aparezcan en la parte izquierda. Como en el caso de las ecuaciones, las condiciones pueden tener ecuaciones, que pueden ser de encaje de patrones, y axiomas de pertenencia. En Maude 2.0 se permite además que las condiciones contengan reglas de reescritura [CDE+00b].

2.2.3. Módulos orientados a objetos

Los *módulos orientados a objetos* son un tipo especial de módulos de sistema, que se utilizan para la definición de clases orientadas a objetos, facilitando una notación conveniente para ello.

Estructura de un módulo orientado a objetos. Un módulo orientado a objetos consiste en una lista de importación de módulos, una declaración de clases, una declaración de mensajes, y unas reglas de reescritura. Opcionalmente puede incluir declaraciones de tipos y de operaciones de forma similar a las declaraciones de los módulos de sistema.

Por ejemplo, el siguiente módulo especifica una versión orientada a objetos del mundo de los bloques [CDE+00a, Sección 9.5]. Un bloque se representa por medio de un objeto con dos atributos, `under`, que indica si el bloque está debajo de otro bloque o si está libre, y `on`, que indica si el bloque está sobre otro bloque o si está sobre la mesa. Un robot se representa por medio de otro objeto con un atributo `hold`, que indica si el robot está vacío o si sostiene un bloque. Las acciones se representan por medio de mensajes. El sistema se utilizará en la Sección 5.2 para probar propiedades modales y temporales de sistemas orientados a objetos en VLRL.

```
(omod OO-BLOCKSWORLD is
  protecting QID .
  sorts BlockId RobotId Up Down Hold .
  subsorts Qid < BlockId RobotId < Oid .
  subsorts BlockId < Up Down Hold .
  op clear : -> Up .      *** block is clear
  op catchup : -> Up .    *** block is caught by the robot arm
  op table : -> Down .    *** block is on the table
  op catchd : -> Down .   *** block is caught by the robot arm
  op empty : -> Hold .    *** robot arm is empty
  class Block | under : Up, on : Down .
  class Robot | hold : Hold .
  msgs pickup putdown : RobotId BlockId -> Msg .
  msgs unstack stack : RobotId BlockId BlockId -> Msg .
  vars X Y : BlockId .
  var R : RobotId .

  rl [pickup] : pickup(R,X) < R : Robot | hold : empty >
                < X : Block | under : clear, on : table >
             => < R : Robot | hold : X >
                < X : Block | under : catchup, on : catchd > .
  rl [putdown] : putdown(R,X) < R : Robot | hold : X >
                 < X : Block | under : catchup, on : catchd >
              => < R : Robot | hold : empty >
                 < X : Block | under : clear, on : table > .
  rl [unstack] : unstack(R,X,Y) < R : Robot | hold : empty >
                 < X : Block | under : clear, on : Y >
                 < Y : Block | under : X >
              => < R : Robot | hold : X >
                 < X : Block | under : catchup, on : catchd >
                 < Y : Block | under : clear > .
  rl [stack] : stack(R,X,Y) < R : Robot | hold : X >
                 < X : Block | under : catchup, on : catchd >
                 < Y : Block | under : clear >
              => < R : Robot | hold : empty >
                 < X : Block | under : clear, on : Y >
                 < Y : Block | under : X > .
endom)
```

Una declaración de clases se introduce con la palabra reservada `class` e incluye el identificador de la clase, en nuestro caso `Block` y `Robot`; los identificadores de los atributos, `under` y `on` en la clase `Block`, y `hold` en la clase `Robot`; y el tipo de cada atributo, `Up`, `Down` y `Hold` respectivamente. El tipo puede ser un tipo de datos algebraico definido en un módulo funcional, o una configuración completa definida en otro módulo orientado a objetos o en un módulo de sistema como ocurre por ejemplo con el atributo `EqConm` en la Sección 3.2.2.

La declaración de mensajes se introduce con la palabra reservada `msg`, o `msgs` en caso de tratarse de una declaración de varios mensajes, seguida de los identificadores de los mensajes y de sus tipos. Como en el caso de las operaciones, los mensajes pueden definirse

utilizando una notación prefija o una notación mixfija.

Representación de los objetos y métodos. Un objeto se representa como un término $< O : C \mid a_1 : v_1, \ldots, a_n : v_n >$, donde O es el nombre del objeto perteneciente a un conjunto `Oid` de identificadores de objetos, C es el identificador de la clase, a_i son los nombres de los atributos del objeto y v_i son sus valores respectivos, que deben pertenecer al tipo asociado con el atributo. Por ejemplo, en la primera regla de reescritura del módulo `OO-BLOCKSWORLD` se utilizan las variables `R` y `X` de clase `Robot` y `Block` respectivamente. El atributo `hold` del robot `R` tiene valor `empty`, el atributo `under` del bloque `X` tiene valor `clear` y el atributo `on` tiene valor `table`.

Las reglas de reescritura representan el código real del módulo, esto es, la implementación del método asociado a un mensaje recibido por un objeto. En general una regla de reescritura tiene la forma

$$\begin{array}{ll} \texttt{crl}\ [l]: & M_1 \ldots M_n < O_1 : C_1 \mid atts_1 > \ldots < O_m : C_m \mid atts_m > \\ \Longrightarrow & < O_{i_1} : C'_{i_1} \mid atts'_{i_1} > \ldots < O_{i_k} : C'_{i_k} \mid atts'_{i_k} > \\ & < Q_1 : D_1 \mid atts''_1 > \ldots < Q_p : D_p \mid atts''_p > \\ & M'_1 \ldots M'_q \\ \texttt{if}\ C & \end{array}$$

donde $k, p, q, n, m \geq 0$, los M_s son expresiones de mensajes, $i_1, \ldots, i_k$ son números diferentes entre los originales $1, \ldots, m$, C es la condición de la regla y l es una etiqueta. El resultado de aplicar una regla de reescritura como esta es que

- los mensajes $M_1, \ldots, M_n$ desaparecen;
- el estado y posiblemente la clase de los objetos $O_{i_1}, \ldots, O_{i_k}$ cambia;
- todos los demás objetos O_j desaparecen;
- se crean nuevos objetos $Q_1, \ldots, Q_p$;
- se envían nuevos mensajes $M'_1, \ldots, M'_q$.

Por ejemplo, al ejecutar la primera regla de reescritura del módulo `OO-BLOCKSWORLD` desaparece el mensaje `pickup(R,X)` y cambia el valor de los atributos de los objetos. En este caso no se crea ni destruye ningún objeto, ni tampoco se envía ningún mensaje nuevo.

Debido a que la regla anterior involucra a varios objetos y mensajes en su lado izquierdo, decimos que es una regla *síncrona*. Todas las reglas del módulo `OO-BLOCKSWORLD` son síncronas Es importante desde el punto de vista conceptual distinguir el caso especial de las reglas que involucran como mucho un objeto y un mensaje en su lado izquierdo. Estas

reglas se llaman *asíncronas* y tienen la forma

$$
\begin{array}{ll}
\texttt{crl}\ [l]: & M \ \ < O : C \mid atts > \\
\Longrightarrow & < O : C' \mid atts' > \\
& < Q_1 : D_1 \mid atts''_1 > \ldots < Q_p : D_p \mid atts''_p > \\
& M'_1 \ldots M'_q \\
\texttt{if}\ \ C &
\end{array}
$$

Por ejemplo, en la implementación del sistema Mobile Maude (Sección 2.2.6), se utiliza la regla de reescritura asíncrona

```
rl [message-out-move] :
   < M : MO | s : '_&_[T,'go[T']], mode : active >
=> go(downPid(T'),
   < M : MO | s : ''_&_[T,'none'MsgSet], mode : idle >) .
```

Por convenio, los únicos atributos de objeto que se hacen explícitos en una regla son aquellos relevantes para esa regla. En particular, los atributos mencionados solo en la parte izquierda de la regla no cambian, los valores originales de los atributos mencionados solamente en el lado derecho de la regla no importan, y todos los atributos no mencionados explícitamente no se modifican.

Los módulos orientados a objetos como módulos de sistema. Todo módulo orientado a objetos puede ser traducido a un módulo de sistema haciendo explícita la estructura completa del módulo que queda oculta bajo las convenciones sintácticas adoptadas [CDE+99a]; esta estructura completa se utiliza en el proceso de verificación de algunas propiedades temporales (ver, por ejemplo, la Sección 5.1.2). En el proceso de traducción, la estructura más básica compartida por todos los módulos orientados a objetos se hace explícita en el módulo funcional CONFIGURATION

```
fmod CONFIGURATION is
  sort Oid Cid Attribute AttributeSet
       Object Message Configuration .
  subsort Object Message < Configuration .
  subsort Attribute < AttributeSet .

  op none : -> AttributeSet .
  op _,_ : AttributeSet AttributeSet -> AttributeSet [assoc comm id: none] .

  op <_:_| > : Oid Cid -> Object .
  op <_:_|_> : Oid Cid AttributeSet -> Object .

  op  none : -> Configuration .
  op __ : Configuration Configuration -> Configuration [assoc comm id: none] .
endfm
```

La traducción de un determinado módulo extiende esta estructura con las clases, mensajes y reglas del módulo [Mes93a, Dur99]:

- Para cada declaración de clase `class` C `|` a_1`:` S_1`,` ... `,`a_n`:` S_n, se introduce un subtipo C del tipo `Cid`, una constante C de tipo C, y una operación (a_i `:` _) `:` S_i `->Attribute` para cada atributo a_i,

- Para cada declaración de mensaje `msg` M`:`$S_1 \ldots S_n$ `->Msg .` se introduce una operación `op` M`:`$S_1 \ldots S_n$ `->Msg .`,

- Para cada relación de subclase $C < C'$ se introduce una declaración de subtipo `subsort` $C < C'$ `.` y el conjunto de atributos de los objetos de la clase C se completan con los de la clase C' y,

- Las reglas de reescritura se modifican para hacerlas aplicables a todos los objetos de una clase y de sus subclases. También se hacen explícitos todos los atributos de los objetos que aparecen en las reglas.

Relación de herencia. Con respecto a la relación de herencia, en Maude se distinguen dos tipos de herencia: *herencia de clases* y *herencia de módulos*.

La herencia de clases está soportada directamente por la estructura de tipos ordenados de Maude. El efecto de una declaración de subclase es que los atributos, mensajes y reglas de todas las superclases contribuyen a la estructura y comportamiento de los objetos en la subclase, y no pueden ser modificados en la subclase; además, la subclase puede tener nuevos atributos y mensajes.

Por ejemplo, podemos refinar el mundo de los bloques definido antes en `OO-BLOCKSWORLD` de forma que los bloques puedan tener color. Sin embargo, las reglas que manipulan los bloques deben permanecer exactamente como en el módulo anterior, lo que se consigue utilizando herencia de clases.

```
(omod OO-BLOCKS-WORLD+COLOR is
  including OO-BLOCKSWORLD .

  sort Color .
  ops red blue yellow : -> Color [ctor] .
  class ColoredBlock | color : Color .
  subclass ColoredBlock < Block .
endom)
```

Por otra parte, la herencia de módulos se utiliza para reutilizar código, permitiendo la modificación del código original de diversas formas. Esta herencia se implementa por medio de la importación de módulos explicada en la Sección 2.2.1. Si en la relación de herencia se quieren añadir nuevos objetos a la clase o identificar objetos de la clase ya existente entonces debe utilizarse una importación no protegida mediante la palabra reservada `inc`; en caso contrario puede utilizarse la importación protegida utilizando la palabra reservada `pr`.

2.2.4. Módulos parametrizados

Los módulos pueden estar parametrizados por *teorías*, en las que se declaran los requisitos de la interfaz, esto es, la estructura y propiedades que se requieren para el parámetro real. La instanciación del parámetro formal de un módulo parametrizado con el módulo del parámetro real requiere una *vista* desde la teoría de la interfaz formal al correspondiente módulo actual. Esto es, las vistas proporcionan la interpretación de los parámetros reales.

Definición de las teorías. Las teorías son teorías de la lógica de reescritura con una interpretación laxa.

Maude soporta tres tipos de teorías: teorías funcionales, teorías de sistema y teorías orientadas a objetos. La estructura de cada una de ellas es la misma que la estructura de los diferentes tipos de módulos, pudiendo importar otras teorías o módulos y contener declaraciones de tipos, de relaciones de subtipo entre tipos, operaciones, ecuaciones, reglas de reescritura, declaraciones de clases, etc. Las teorías funcionales se declaran con las palabras reservadas `fth ... endfth`, las teorías de sistema con las palabras reservadas `th ... endth`, y las teorías orientadas a objetos con `oth ... endoth`.

Por ejemplo, la definición de una teoría `TRIV` que solo requiere la definición de un tipo es la siguiente:

```
fth TRIV is
  sort Elt .
endfth
```

Definición de los módulos parametrizados. Los módulos pueden estar parametrizados por una o más teorías, pudiendo existir varios parámetros declarados con la misma teoría. Por ejemplo, el siguiente módulo se utiliza para definir conjuntos de elementos.

```
fmod SET[X :: TRIV] is
  sort Set[X] .
  subsort Elt.X < Set[X] .

  op empty-set : -> Set[X] [ctor] .
  op __ : Set[X] Set[X] -> Set[X] [ctor assoc comm id: empty-set] .

  vars N N' : Elt.X .
  vars S S' : Set[X] .

  eq N N = N .
endfm
```

La implementación del módulo se ha tomado de [CDE+00a, página 48], donde se definen también operaciones para comprobar la pertenencia de un elemento al conjunto, eliminar un elemento, calcular el cardinal, y obtener el conjunto diferencia.

Todas las teorías que aparezcan en la interfaz deben estar etiquetadas, de forma que sus tipos se identifiquen de forma única. En el código del módulo todos los tipos definidos en las

teorías deben cualificarse con su etiqueta, aunque no exista ambigüedad en su identificador. Por ejemplo, en el módulo `SET` el tipo `Elt` de la teoría se ha cualificado con la etiqueta `X` en la declaración `subsort Elt.X <Set[X]`. Debido a problemas de ambigüedad, los tipos deben ser desambiguados con el nombre de la vista (o vistas) usado en la instanciación del módulo parametrizado. Así, por ejemplo, en el módulo anterior aparece la declaración de tipos `sort Set[X]`, la cual se convierte en `Set[V]` cuando el módulo es instanciado con una vista `V`.

Definición de las vistas. Se utilizan las vistas para indicar cómo un módulo o teoría satisface los requisitos de una teoría dada como parámetro de un módulo parametrizado. En la versión actual de Maude, todas las vistas deben ser definidas y todas ellas deben de tener un nombre.

La definición de una vista comienza con la palabra reservada `view` seguida del identificador de la vista; a continuación la palabra reservada `from` seguida del identificador de la teoría que define el parámetro, seguida de la palabra reservada `to` y del identificador del módulo o teoría con el que se quiere instanciar el parámetro, seguido de la palabra reservada `is`. A continuación de la cabecera debe declararse la correspondencia existente entre cada tipo, operación, clase y mensaje de la teoría que define el parámetro. Esta comienza con la palabra reservada `sort` (`op`, `class`, o `msg`) seguida del identificador del tipo (operación, clase, o mensaje) en la teoría que define el parámetro, seguida de la palabra reservada `to` y del identificador del tipo (operación, clase o mensaje) en el módulo o teoría con el que queremos instanciar el parámetro. Aunque es necesario dar la correspondencia para todos los tipos, aun en el caso de identidades, no es necesario dar la correspondencia de operaciones, clases y mensajes cuando se trata de identidades.

Por ejemplo, la siguiente vista instancia el parámetro del módulo `SET` con el módulo que proporciona el sistema para la definición de los números enteros.

```
view Nat from TRIV to NAT is
  sort Elt to Nat .
endv
```

La instanciación es el proceso por el cual se asignan parámetros actuales a los parámetros de un módulo parametrizado, creando como resultado un nuevo módulo. La instanciación requiere una vista desde cada parámetro formal al correspondiente parámetro actual. La instanciación de un módulo parametrizado tiene que hacerse con vistas definidas explícitamente con anterioridad. Por ejemplo, podemos obtener los conjuntos de enteros con la expresión de módulo `SET[Nat]`.

2.2.5. Reflexión y lenguajes de estrategias

Maude explota de forma eficiente la funcionalidad que le proporciona el hecho de que la lógica de reescritura sea reflexiva, soportando además un número arbitrario de niveles de reflexión. En este sentido, el módulo predefinido `META-LEVEL` [CDE+99a, Apéndice D] proporciona la funcionalidad básica en la teoría universal $\mathcal{U}$ de la lógica de reescritura,

esto es, la capacidad de representar teorías como datos y la capacidad de reducir términos de estas teorías y aplicar sus reglas de reescritura.

Definición del módulo `META-LEVEL`. Los términos de Maude son elementos del tipo de datos `Term` y los módulos son términos del tipo `Module`.

El proceso de reducir un término a su forma normal en un módulo funcional se lleva a cabo por medio de la función:

```
metaReduce : Module Term -> [ResultPair] .
```

la cual toma como primer argumento la meta-representación de un módulo R y como segundo argumento la meta-representación de un término t en R, y devuelve la meta-representación del término t reducido a forma normal usando las ecuaciones de R junto con su tipo.

La operación

```
metaRewrite : Module Term Bound -> [ResultPair] .
```

es análoga a la anterior, pero en lugar de utilizar únicamente la parte ecuacional del módulo utiliza también las reglas de reescritura para reescribir el término utilizando la estrategia por defecto de Maude.

El proceso de aplicar una regla de un módulo de sistema a un término se realiza con la operación

```
metaApply : Module Term Qid Substitution Nat -> [ResultTriple] .
```

Los cuatro primeros argumentos de esta operación deben ser las meta-representaciones de un módulo R, un término t en R, una etiqueta l de algunas reglas[4] en R y una sustitución σ para las variables de estas reglas. El último argumento es un número natural n. La operación `metaApply` se evalúa de la forma siguiente:

1. Se reduce el término t completamente utilizando las ecuaciones en R.
2. El término resultado se intenta encajar con los lados izquierdos de todas las reglas con etiqueta l en el módulo, parcialmente instanciadas con la sustitución σ, descartando aquellos encajes que no satisfagan la condición de la regla.
3. Se descartan los n primeros encajes que hayan tenido éxito; si existe un encaje $n+1$ se aplica su regla usando este encaje realizándose a continuación los pasos siguientes. En otro caso se devuelve `failure`.
4. El término resultado de aplicar la regla dada con el $(n+1)$ encaje, se reduce utilizando las ecuaciones de R.
5. Se devuelve una terna cuyo primer elemento es la representación del término resultante reducido, el segundo elemento es el tipo de este término, y cuyo tercer elemento es la representación del encaje utilizado en el proceso de reducción.

[4]Nótese que en general varias reglas de reescritura pueden tener la misma etiqueta.

Lenguajes de estrategias. Las estrategias se utilizan para controlar el proceso de reescritura, que en principio puede dirigirse en cualquier dirección. Debido a las propiedades reflexivas de la lógica de reescritura, es posible definir el lenguaje de estrategias dentro de la lógica, por medio de reglas de reescritura. De hecho, hay una gran libertad para definir diferentes lenguajes de estrategias dentro de Maude [CM96b, Cla00, CELM96]. Como metodología general cada usuario puede definir su propio lenguaje de estrategias extendiendo las operaciones básicas `metaReduce` y `metaApply` del módulo `META-LEVEL` para definir operaciones más complejas.

En la especificación del modelo de red de telecomunicaciones que se presenta en el Capítulo 3 se utiliza y amplía el lenguaje de estrategias `STRAT` definido por Clavel y otros en [CELM96]. El lenguaje `STRAT` define los tipos `Strategy` y `StrategyExp` para las estrategias de reescritura y las expresiones de estrategias, respectivamente, y extiende el módulo `META-LEVEL` con operaciones para componer estrategias. En la aplicación del Capítulo 3, sin embargo, no usaremos su estructura de árbol de soluciones, y hemos adaptado la sintaxis de las operaciones y ecuaciones a la última versión disponible del lenguaje en este momento [CDE+00b]. En particular, las operaciones que usaremos en nuestra aplicación son:

- operaciones que definen estrategias básicas:

```
op idle : -> Strategy .
op apply : Label -> Strategy .
op rew_in_with_ : Term Module Strategy -> StrategyExp .
op failure : -> StrategyExp .
```

- operaciones que componen estrategias:

```
op _;_ : Strategy Strategy -> Strategy .
op _;;_orelse_  : Strategy Strategy Strategy -> Strategy .
op _andthen_ : StrategyExp Strategy -> StrategyExp .
```

Estas operaciones satisfacen las siguientes ecuaciones:

```
eq rew T in M with (S ; S') = (rew T in M with S) andthen S' .
eq (rew T in M with idle) andthen S = rew T in M with S .
eq failure andthen S = failure .
ceq rew T in M with apply(L) =
    if metaApply(M,T,L,none,0)) :: ResultTriple then
      rew getTerm(metaApply(M,T,L,none,0)) in M with idle
    else failure .
ceq rew T in M with (S ;; S'' orelse S') =
    if rew T in M with S == failure then rew T in M with S'
    else (rew T in M with S) andthen S'' .
```

Una expresión de estrategia tiene inicialmente la forma `rew T in M with S`, lo que significa que tenemos que aplicar la estrategia `S` al meta-término `T` en el módulo `M`. Esta

expresión de estrategia se reduce entonces por medio de las ecuaciones del módulo STRAT a una expresión de la forma rew T' in M with S', donde T' es el término calculado al aplicar la estrategia S al meta-término T y S' es la estrategia restante. Si el proceso de reducción tiene éxito, se alcanza una expresión de estrategia de la forma rew T'' in M with idle, donde T'' es un término que representa la solución obtenida; en otro caso, se genera la expresión de estrategia failure.

Controlar el orden en el que una transacción consistente en una secuencia de pasos de reescritura tendrá lugar en el nivel objeto requiere el uso de la operación de concatenación de estrategias y de la operación que aplica una regla en el nivel objeto. La idea es que la estrategia concatene las reglas de reescritura en el orden en que deben ser usadas en el proceso por medio de la operación _;_. Obsérvese que las reglas se aplican en el nivel objeto según van siendo requeridas por la estrategia, y que la estrategia controla las situaciones de fallo cuando no existe ninguna regla que aplicar por medio de la ecuación que define la operación apply usando metaApply.

Añadimos una nueva operación sobre estrategias al lenguaje STRAT de [CELM96] con el fin de aplicar una estrategia sobre una lista de objetos:

```
op Iterate : Strategy -> Strategy .
eq Iterate(S) = S ;; Iterate(S) orelse idle .
```

Intuitivamente, la estrategia Iterate(S) es muy general y simplemente aplica varias veces una estrategia S, finalizando cuando esta estrategia no puede aplicarse más. Siguiendo la definición original de las operaciones en el lenguaje de estrategias STRAT [CELM96], hemos definido la operación Iterate por medio de una ecuación; sin embargo, dado que las ecuaciones deben ser terminantes el usuario debe asegurar que para su aplicación concreta el proceso de reducción finaliza. En el ejemplo del Capítulo 3, queremos aplicar una regla dada a una lista de objetos, aplicando la misma estrategia pero a un objeto diferente cada vez. Esto es, queremos iterar sobre la lista dada. Si la estrategia no puede aplicarse más de una vez a un objeto (por ejemplo, cuando uno de sus efectos es eliminar al objeto de la lista) no hay que considerar nada más. Sin embargo, si este no es el caso, debemos asegurarnos de que cada objeto de la lista se trata exactamente una vez.

Una posibilidad diferente es llevar el control de la lista de objetos en el nivel del lenguaje de estrategias, en lugar de hacerlo en el nivel objeto. Esto puede conseguirse, por ejemplo, en el lenguaje de estrategias que Clavel y Meseguer describen en [Cla00, CM96b], donde usan sustituciones y ligaduras para pasar información de un nivel a otro. Nosotros hemos decidido no hacerlo con el fin de mantener el lenguaje de estrategias lo más simple posible, y reutilizar en lo posible las reglas de la aplicación no reflexiva.

2.2.6. El sistema Mobile Maude

El sistema Mobile Maude desarrollado por Durán y otros en [DELM00] proporciona una especificación en Maude de un modelo de movilidad de objetos entre procesos. En Mobile Maude hay dos entidades principales: los *procesos* y los *objetos móviles* que pueden moverse de un proceso a otro e intercambian mensajes con otros objetos, en el mismo proceso o en otro diferente.

La clase `P` de procesos en Mobile Maude se declara como:

```
class P |
  cnt : Nat,   *** contador para generar nuevos identificadores
                     *** de objetos móviles
  cf : Configuration, *** configuración del proceso
  guests : Set[Mid],  *** objetos en el proceso
  forward : PFun[Nat, Tuple[Pid, Nat]] .
            *** información sobre la localización de los objetos del proceso
```

donde los identificadores de los procesos son de tipo `Pid` y los identificadores de los objetos móviles son de tipo `Mid` y tienen la forma `o(PI,N)` donde `PI` es el nombre del proceso en el que el objeto fue creado y `N` es un número. `PFun` es una tabla que almacena información sobre donde se encuentran los objetos móviles que fueron creados en el proceso.

Los objetos móviles se especifican como objetos de clase `MO`:

```
class MO |
  mod : Module,       *** reglas de reescritura del objeto
  s : Term,           *** estado actual
  p : Pid,            *** localización actual
  gas : Nat,          *** límite de recursos
  hops : Nat,         *** número de saltos
  mode : Mode .       *** modo del objeto (ocioso/activo)
```

donde el módulo del objeto móvil debe ser orientado a objetos y el estado debe ser la meta-representación de un par de configuraciones de la forma `C & C'` con `C'` un multiconjunto de *mensajes de salida* que deben ser enviados y `C` un multiconjunto de *mensajes de entrada* todavía no procesados y un objeto que debe tener el mismo identificador que el objeto móvil que lo contiene. El modo será `active` cuando el objeto esté dentro de un proceso e `idle` cuando se esté moviendo.

Un objeto comienza a moverse a partir del objeto interno al objeto móvil, que pone la meta-representación `'go[T']` como segunda componente (es decir, como mensaje de salida) del estado. El término `T'` es la meta-representación del nombre del proceso donde el objeto quiere ir. Esto permite que se aplique la regla

```
rl [message-out-move] :
   < M : MO | s : '_&_[T,'go[T']], mode : active >
=> go(downPid(T'),
   < M : MO | s : ''_&_[T,'none'MsgSet], mode : idle >) .
```

que inicia el movimiento de un objeto cambiando el modo del objeto de `active` a `idle` y enviando un mensaje `go`.

El proceso de movilidad continúa con la regla

```
rl [go-proc] : < PI : P | cf : C go(PI', < M : MO | >), guests : M . SMO >
=>  if PI =/= PI' then
       < PI : P | cf : C, guests : SMO > go(PI', < M : MO >)
    else < PI : P | cf : C < M : MO | p : PI, mode : active > >
    fi .
```

que manda el mensaje go fuera del proceso en el que estaba el objeto móvil.

Por último, la regla

```
rl [arrive-proc] : go(PI, < o(PI', N) : MO | hops : N' >)
   < PI : P | cf : C, guests : SMO, forward : F >
=> if PI == PI' then
      < PI : P | cf : C < o(PI', N) : MO | p : PI, hops : N' + 1, mode : active >,
      guests : o(PI', N) . SMO, forward : F [ N -> (PI, N' + 1) ] >
   else
      < PI : P | cf : C < o(PI', N) : MO | p : PI, hops : N' + 1, mode : active >,
                 guests : o(PI', N) . SMO >
      to PI' @ (PI, N' + 1)  N
  fi .
```

introduce el objeto móvil en el proceso de destino.

Los mensajes en la configuración de un proceso o en el módulo de un objeto pueden ser de cualquier forma, pero los mensajes entre procesos deben ser de la forma to_:_, go o newo. Para enviar un mensaje entre dos procesos se emplean las reglas: message-out-to, msg-send, msg-arrive-to-proc y msg-in.

La especificación completa del sistema se encuentra en [DELM00].

Capítulo 3

Especificación de un modelo de red de telecomunicaciones

En este capítulo se muestran muchas de las facilidades que proporciona el lenguaje Maude para la especificación de sistemas complejos. Para ello se presenta una especificación de un modelo orientado a objetos de una red de telecomunicaciones de banda ancha. Este trabajo fue la primera aplicación de la lógica de reescritura y Maude a sistemas reales, ya que el modelo puede utilizarse tanto para la gestión de la red como para su planificación [PMO02]. La especificación en Maude puede entonces utilizarse para simular y analizar los protocolos y aplicaciones de estos sistemas.

En el desarrollo de la especificación se comparan diversas opciones que proporciona el lenguaje, como por ejemplo el uso de identificadores de objetos como valor de un atributo o bien el uso del objeto en sí mismo, comentando los aspectos positivos y negativos de cada una de ellas en la especificación de este modelo concreto. Se muestra también cómo el uso de la reflexión permite simplificar la especificación de los sistemas, al mismo tiempo que se resaltan algunas facilidades que sería interesante introducir en el lenguaje.

La especificación desarrollada en este capítulo se presentó en primer lugar en la conferencia *First International Workshop on Rewriting Logic and its Applications (Asilomar, California, EE.UU.)* publicándose en [PMO96]. Posteriormente se amplió el trabajo con la especificación de la red usando reflexión y se presentó en las conferencias *APPIA-GULP-PRODE'97, Joint Conference on Declarative Programming (Grado, Italia)* y *Recent Trends in Algebraic Development Techniques, WADT'98 (Lisboa, Portugal)* publicándose en [PMO97, PMO99]. Un compendio de todo el trabajo se ha publicado en la revista *Theoretical Computer Science* [PMO02].

3.1. Descripción del modelo

El modelo seleccionado se basa en la estructura de capas de las redes de banda ancha con modo de transferencia asíncrono (ATM), que dividen la red en tres capas lógicas: capa *física*, capa de *caminos virtuales* (VP), y capa de *canales virtuales* (VC), cada una de ellas

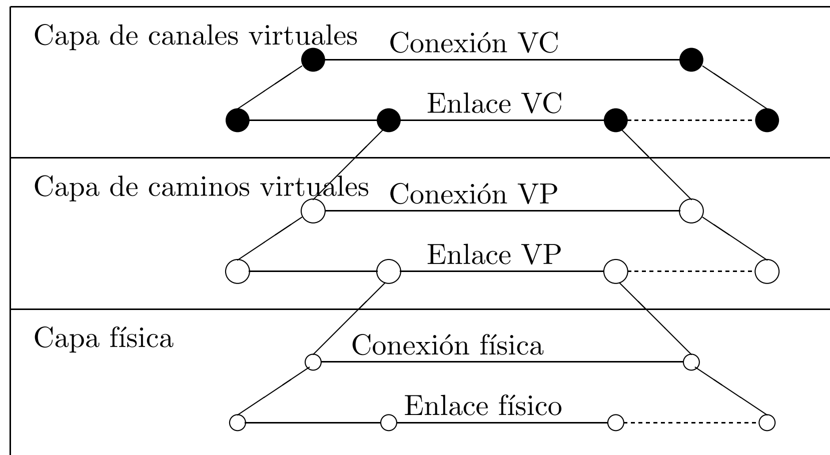

Figura 3.1: Estructura de la red

relacionada con funciones diferentes de la red (Figura 3.1).

3.1.1. Objetos de la red

Los objetos básicos del modelo son los *nodos*, los *enlaces*, y las *conexiones*, cada uno de los cuales aparece en cada una de las tres capas de la red. Los nodos representan los puntos de la red donde se manejan las señales de comunicación. Se distinguen dos clases de nodos:

- *Nodos de transmisión*, que tienen funciones físicas y de transmisión y están definidos en el nivel físico.
- *Nodos de conmutación*, que tienen funciones de conexión y conmutación y están definidos en las capas superiores.

Los *enlaces* se definen entre los nodos de una misma capa como entidades manejables que transportan información que puede ser accedida en sus dos extremos.

Las *conexiones* se definen como secuencias configurables de enlaces de una misma capa. Se utilizan para soportar los servicios de comunicaciones entre una pareja de usuarios, estando cada usuario relacionado con un único nodo. Los enlaces de las capas superiores están soportados por las conexiones de la capa inferior, y los enlaces físicos están soportados por el medio de transmisión.

Además de los tres objetos básicos utilizados para definir la estructura topológica de la red, existen otros componentes en la red:

- Los *nodos* están formados por piezas de equipo, entre las que cabe destacar los puertos, que representan las características físicas (número de puertos en una unidad

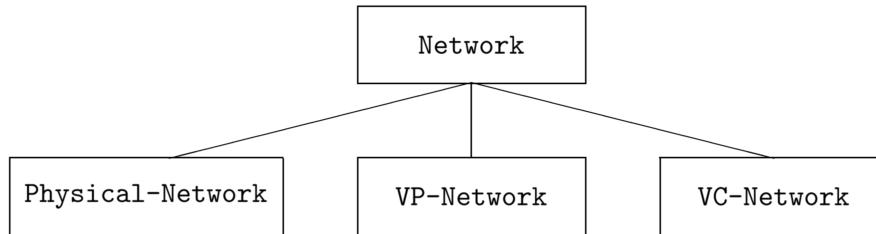

Figura 3.2: Relación de herencia entre clases de la red

de terminación, capacidad del puerto, ...) y el coste de los elementos definidos en el nodo. El equipo de un nodo de la capa física se denomina *equipo de transmisión*, mientras que el equipo de un nodo de las capas VP y CV se denomina *equipo de conmutación*.

- Los *servicios* representan las características en cuanto a ancho de banda demandada de los servicios de comunicación (fax, teléfono, correo electrónico, etc.) proporcionados por la red.

Una *red* está formada por un conjunto de enlaces junto con el conjunto de nodos definidos como sus extremos y las correspondientes conexiones entre los nodos. Se define una red diferente para cada capa del modelo: *red física*, *red VP*, y *red VC*. Los enlaces definidos en cada capa se denominan *enlaces físicos*, *enlaces VP*, y *enlaces VC*, respectivamente, y análogamente para los nodos y las conexiones.

3.1.2. Relaciones entre los objetos de la red

En la recomendación X.720 de la Unión Internacional de Telecomunicaciones (ITU) se definen tres tipos de relaciones entre los objetos de una red: relación de herencia, relación de contenido, y relaciones explícitas de grupo [ISO90, ZP92].

La relación de herencia captura las propiedades comunes de un conjunto de clases definiendo un orden taxonómico ente ellas. La herencia aparece en el modelo entre las clases genéricas y las respectivas clases especializadas de cada capa. Por ejemplo, la Figura 3.2 muestra una clase genérica `Network` que tiene tres subclases: `Physical-Network`, `VP-Network` y `VC-Network`. Se definen relaciones similares entre las clases genéricas correspondientes a los nodos, enlaces y conexiones, y las clases especializadas respectivas de cada capa.

La relación de contenido, también llamada *composición de objetos*, captura la semántica asociada con la relación "es-parte-de" entre objetos. En su sentido más fuerte, la composición de objetos implica que una parte no puede existir sin su propietario, ni puede ser compartida con otro propietario. Esto es, las partes deben ser destruidas cuando el objeto propietario es destruido, y solo deben ser creadas como parte del proceso de creación del objeto propietario.

En las redes de telecomunicaciones es una práctica común aplicar la relación de contenido al proceso de construcción del identificador del objeto. Los identificadores de los objetos se forman como la concatenación del identificador del objeto propietario con un identificador único del objeto miembro. La unicidad de los identificadores de los objetos para objetos miembros se aplica solo dentro del dominio del objeto propietario. Nosotros aplicaremos esta relación a los nodos con las piezas de equipo que lo componen. También puede utilizarse esta relación para modelar la relación entre la red y sus elementos (nodos, enlaces, y conexiones), y entre objetos de diferentes capas, pero, debido a las fuertes restricciones que impone en el modelo, hemos preferido utilizar relaciones explícitas de grupo para modelarlas.

Las relaciones explícitas de grupo son relaciones entre objetos que no implican la existencia de ningún vínculo ni asociación permanente, esto es, no involucran a las operaciones de creado y borrado, al tiempo que permiten que un objeto tenga muchos propietarios. Estas relaciones pueden ser de diferentes tipos: cliente-servidor, ser-miembro-de, copia-de-seguridad, etc. La relación cliente-servidor se utiliza para representar la relación entre los objetos de las distintas capas. La relación ser-miembro-de se define de forma natural entre los enlaces y las conexiones de una misma capa, y entre la red y el conjunto de nodos, enlaces y conexiones que la forman. Las relaciones de copia-de-seguridad se definen entre los enlaces de una misma capa y entre conexiones de una misma capa.

3.2. La especificación de la red

El modelo del sistema lo forman los objetos de la red y las relaciones entre ellos. El sistema evoluciona debido a las peticiones de los usuarios (preguntas, modificaciones, creaciones y eliminaciones) que producen una cadena de mensajes entre los objetos de la red hasta que se alcanza una nueva configuración estable correspondiente a la petición.

3.2.1. Las clases orientadas a objetos

En general, cada objeto de la red descrito en la Sección 3.1.1 da lugar a una clase en la especificación orientada a objetos, y de hecho cada clase se especifica en un módulo orientado a objetos de Maude. Además, necesitamos varios módulos funcionales paramétricos para especificar los tipos de datos algebraicos tales como listas, conjuntos, tuplas, etc., los cuales son posteriormente instanciados, por ejemplo, como conjuntos de identificadores de nodos.

La clase principal del modelo es la clase `C-Network` que toma el papel de un meta-objeto tal como se describe en [Mes93a], cuya función es difundir los mensajes de usuario que llegan al sistema, ya que esta clase tiene información sobre todos los objetos actuales del sistema. La clase `C-Network` actúa como la clase raíz del modelo, esto es, todos los accesos al sistema se realizan a través de esta clase o de sus subclases. La clase se especializa para cada una de las capas de la red en subclases `C-Physical-Network`, `C-VP-Network` y `C-VC-Network`, como se ha explicado anteriormente.

Cada objeto de la clase `C-Network` se caracteriza por todos los nodos, enlaces y conexiones que pertenecen a la red.

```
class C-Network | NodeSet : NodeOidSet,
                  LinkSet : LinkOidSet,
                  ConnectionSet : ConnectionOidSet .
```

Debido a que el orden de los elementos no es importante, los tipos de los atributos que los definen son *conjuntos* de identificadores de objetos en lugar de listas de estos identificadores. Declarar conjuntos de identificadores de objetos es suficiente en el modelo propuesto, ya que la relación entre estos elementos y la red es una relación ser-miembro-de. Si hubiéramos definido una relación de contenido, hubiese sido mejor representar estos elementos como conjuntos de objetos y definirlos como subconfiguraciones [Mes93a, LLNW96], tal como se discute en la Sección 3.2.2.

Los mensajes asociados a la clase `C-Network` representan las preguntas, modificaciones, creaciones y eliminaciones que los usuarios pueden hacer en el sistema.

3.2.2. Implementación de las relaciones entre objetos

Se consideran tres relaciones en el modelo: la relación de herencia, la relación de contenido y las relaciones explícitas de grupo. La relación de herencia está soportada directamente por la herencia de clases en Maude (ver Sección 2.2.3); por el contrario, la relación de contenido y las relaciones explícitas de grupo no están directamente soportadas por el lenguaje, aunque pueden ser especificadas de forma natural en él.

Relación de herencia.

La herencia de clases se define entre las clases genéricas `C-Network`, `C-Node`, `C-Link` y `C-Connection`, y las clases especializadas para cada capa.

En el caso de los nodos, se define una nueva clase `C-ATM-Node` para capturar el comportamiento común de las clases `C-VP-Node` y `C-VC-Node`. Básicamente, esta clase implementa los mensajes relacionados con el coste del equipo del nodo, el cual se calcula de forma diferente en la capa física y en las otras dos capas debido a las diferencias existentes entre el equipo de transmisión y el equipo de conmutación. La relación de herencia entre estas clases está representada en la Figura 3.3.

Como los mensajes relacionados con el coste de los nodos deben ser implementados también en el módulo correspondiente a la clase `C-Physical-Node`, podríamos utilizar la herencia de módulos proporcionada por Maude para redefinirlos a partir del módulo que define la clase `C-ATM-Node`. Sin embargo, hemos preferido no hacerlo, ya que casi todos los mensajes originales deben ser redefinidos, contraviniendo el objetivo de la reutilización del código. Lo que hacemos es, simplemente, proporcionar las nuevas reglas para los correspondientes mensajes en el módulo que define la clase `C-Physical-Node`.

A partir de nuestra experiencia en esta aplicación, concluimos con respecto al diseño del lenguaje Maude que sería conveniente tener la posibilidad de definir operaciones *abstractas*

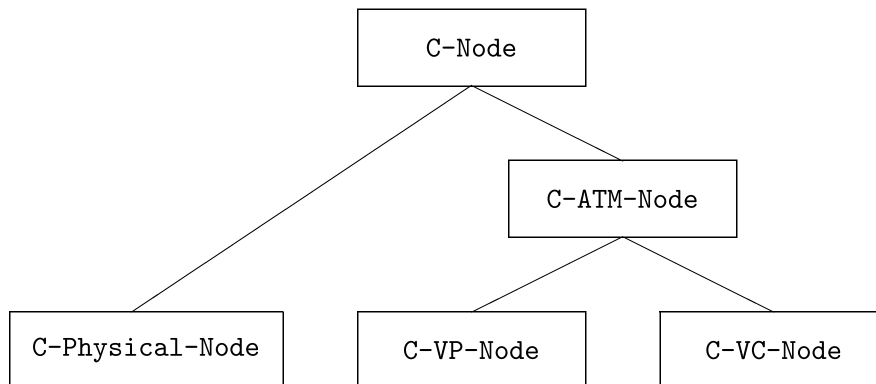

Figura 3.3: Relación de herencia entre clases relacionadas con los nodos

en las superclases de forma que se forzase la implementación de estas operaciones en las subclases.

Relación de contenido.

Comparamos dos enfoques para la implementación de la relación de contenido. El primero considera los identificadores de objeto de los objetos miembro como atributos del objeto propietario. El segundo define un atributo del objeto propietario como una subconfiguración de los objetos miembro [Mes93a, LLNW96]. En ambos casos, la relación de contenido impacta en las reglas de creación y borrado de objetos en Maude. La creación de objetos debe no solo asegurar la unicidad de la identidad de los objetos, sino que debe además asegurar las dos propiedades de la relación de contenido:

- El objeto miembro debe tener un único objeto propietario.
- El objeto miembro no puede existir si su objeto propietario no existe.

La creación de objetos se define en [Mes93a] por medio del mensaje `new(C | Atts)` donde `C` es el identificador de la clase y `Atts` son los atributos. Para conseguir la unicidad de la identidad de los objetos, se asume que en la configuración inicial tenemos una colección de objetos diferentes de clase `ProtoObject` la cual tiene un atributo `counter`, cuyo valor es un número natural que se utiliza como parte del nuevo nombre del objeto. Las regla para crear objetos es la siguiente:

```
rl [New] : new(C | Atts) < O : ProtoObject | counter : N >
=> < O : ProtoObject | counter : N+1 > < O.N : C | Atts > .
```

Este esquema ha sido ligeramente modificado en [Mes93b] para forzar que algunos atributos del objeto tengan siempre un cierto valor inicial fijo. Esto se consigue introduciendo

una instrucción `initially` en la definición de la clase, la cual declara los atributos que tienen un valor inicial y su correspondiente valor. Por ejemplo, podemos tener la siguiente declaración de clase:

```
class C | a1 : t1, a2 : t2, a3 : t3, a4 : t4 .
initially a1 : v1, a2 : v2 .
```

donde `ai` son los identificadores de los atributos, `ti` son sus respectivos tipos, y `vi` sus valores. Las reglas para crear objetos tiene ahora la forma:

```
rl [New-ival1] : new(C | a3 : v3, a4 : v4)
   < O : ProtoObject | counter : N >
=> < O : ProtoObject | counter : N+1 >
   < O.N : C | a1 : v1, a2 : v2, a3 : v3, a4 : v4 > .
```

El primer enfoque para la implementación de la relación de contenido declara los identificadores de los objetos miembro como atributos del objeto propietario y, siguiendo el esquema previo, podemos forzar que tengan un valor inicial en el proceso de creación. Por ejemplo, las reglas de creación para una clase `X` con clases miembro `Y1` y `Y2` podrían ser las siguientes:

```
class Y1 | b1 : t1, b2 : t2 .
initially b1 : v1, b2 : v2 .

class Y2 | c1 : t1', c2 : t2' .
initially c1 : w1, c2 : w2 .

class X | a1 : Oid, a2 : Oid, a3 : t3, a4 : t4 .
initially a1 : Y1, a2 : Y2 .

rl [New-ival2] : new(X | a3 : v3, a4 : v4)
   < O : ProtoObject | counter : N >
=> < O : ProtoObject | counter : N+1 >
   < O.N : X | a1 : O.N.1, a2 : O.N.2, a3 : v3, a4 : v4 >
   < O.N.1 : Y1 | b1 : v1, b2 : v2 >
   < O.N.2 : Y2 | c1 : w1, c2 : w2 > .
```

El resultado de aplicar esta regla es que el mensaje `new` desaparece, el atributo `counter` del objeto `ProtoObject` se incrementa, y se crean tres nuevos objetos en el sistema: un objeto de la clase declarada en el mensaje `new`, cuyo identificador es único en el sistema ya que se forma de manera automática con el identificador del objeto `ProtoObject` y el valor del atributo `counter`, y con dos atributos que son identificadores de objetos. Estos dos identificadores de objetos son también únicos en el sistema ya que están formados con el identificador del objeto propietario. Los otros dos objetos corresponden a los objetos miembro. Este tipo de regla garantiza que los objetos miembro son creados con el objeto propietario, y el proceso de construcción del identificador del objeto es el usado normalmente en las redes de telecomunicaciones.

Las principales diferencias introducidas en este proceso de creación de objetos con respecto al esquema presentado por Meseguer [Mes93b] y explicado anteriormente son:

- La instrucción `initially` define como valor inicial del atributo el identificador de la clase de los objetos miembro. El valor asignado al atributo es el identificador del objeto proporcionado automáticamente por la regla de creación mediante la combinación del identificador del objeto propietario y un identificador dado a cada objeto miembro por el objeto propietario (en el ejemplo anterior, un número asociado con el nombre del atributo).

- La regla para el mensaje `new` crea el nuevo objeto así como los objetos miembro. Crear los objetos miembro directamente en lugar de mandar nuevos mensajes `new` nos permite nombrar a los objetos miembro usando el nombre del objeto propietario como es usual en las redes de telecomunicaciones. Mandando nuevos mensajes `new`, el objeto miembro sería creado usando la clase `ProtoObject` y tendría por lo tanto un identificador general.

- No se proporcionan mensajes `new` para los objetos miembro, porque no pueden ser creados fuera del proceso de creación de sus objetos propietarios.

El enfoque que se presenta en [Mes93a] al borrado de objetos utiliza un mensaje `delete(A)`, con una regla

```
rl [Delete] : delete(A) < A : X | Atts > => null .
```

Para mantener la relación de contenido, los objetos miembro deben ser borrados al mismo tiempo que el objeto propietario. Esto se consigue simplemente enviando mensajes de borrado a todos los objetos miembro en la regla de borrado del objeto propietario.

El segundo enfoque, que trata los subobjetos como parte del estado del objeto propietario, es más simple. La regla de creación anterior se define en este caso como:

```
class Y1 | b1 : t1, b2 : t2 .
initially b1 : v1, b2 : v2 .

class Y2 | c1 : t1', c2 : t2' .
initially c1 : w1, c2 : w2 .

class X | conf : Subconfiguration of Y1 Y2, a3 : t3, a4 : t4 .

rl [New-subconfiguration] : new(X | a3 : v3, a4 : v4)
   < O : ProtoObject | counter : N >
=> < O : ProtoObject | counter : N+1 >
   < O.N : X | conf : < O.N.1 : Y1 | b1 : v1, b2 : v2 >
                      < O.N.2 : Y2 | c1 : w1, c2 : w2 >,
               a3 : v3, a4 : v4 > .
```

Este enfoque enfatiza el hecho de que un objeto es parte de otro. Sin embargo, el primero puede resultar más elegante cuando se consideran cadenas de relaciones de contenido entre los objetos de la red ya que el segundo puede dar lugar a una implementación confusa. La elección entre ambas estará basada principalmente en las características del lenguaje. En nuestro modelo, hemos seleccionado la segunda, tal como fue propuesta en [Mes93a]. Como se ha mencionado anteriormente, la clase `C-ATM-Node` define una relación de contenido entre el nodo y su equipo:

```
class C-ATM-Node | EqConm : C-Eq-Conm, UtilL : EQCUtilList .
```

donde `C-Eq-Conm` es una clase que define el equipo de conmutación del nodo.

Relaciones explícitas de grupo.

Las relaciones explícitas de grupo se modelan directamente definiendo un conjunto de identificadores de objetos como el valor de un atributo. La única restricción que se impone en el proceso de creación del objeto es que los objetos que aparecen como valor del atributo deben existir en el sistema. Siguiendo esta idea, un objeto de red, relacionado mediante una relación de es-miembro-de con el conjunto de nodos, enlaces y conexiones que lo forman, se crea inicialmente con un conjunto vacío de estos elementos. Entonces, los nodos, enlaces y conexiones se añaden a la red utilizando mensajes `AddNodeTo_`, `AddLinkTo_Between_and_` y `AddConnectionTo_Between_and_`. Las reglas que implementan estos mensajes crean los objetos al mismo tiempo que se introducen como elementos del objeto red.

```
rl [Add-node] : AddNodeTo N
=> (new C-Node ack N req X) AddNodeToNetwork(N,X) .

rl [Add-node] : (to N:X is No) AddNodeToNetwork(N,X)
  < N : C-Network | NodeSet : Ns >
=> < N : C-Network | NodeSet : No Ns > .
```

El objeto nodo se crea utilizando el mensaje `new`, porque el identificador del nodo lo proporciona el objeto de clase `ProtoObject`, ya que las relaciones explícitas de grupo no crean un espacio de construcción de identificadores de objetos propio como ocurre con las relaciones de contenido. Utilizamos el mensaje `new` con acuse de recibo [Mes93a] que devuelve el mensaje `(to N:X is No)` cuando es atendido, donde `X` es un identificador de mensaje que se genera cuando se envía el mensaje y `No` es el identificador de objeto asignado al nuevo objeto. Una vez que el objeto nodo se ha creado en el sistema, la segunda regla se utiliza para introducirlo en la red correspondiente.

En el caso de los enlaces y las conexiones, que tienen relaciones de grupo adicionales declaradas con otros objetos, las reglas deben comprobar también que se satisfacen estos requisitos. Por ejemplo, las reglas que implementan el mensaje `(AddLinkTo N Between No1 and No2)` que añade un enlace a una red física debe comprobar que los dos nodos asociados ya han sido añadidos anteriormente a la red.

```
rl [Add-link] : (AddLinkTo N Between No1 and No2)
   < N : C-Physical-Network | NodeSet : No1 No2 Ns >
=> < N : C-Physical-Network | > (new C-Physical-Link ack N req X)
   AddLinkToNetwork(N,X,No1,No2) .

rl [Add-link] : (to N:X is L) AddLinkToNetwork(N,X,No1,No2)
   < N : C-Physical-Network | LinkSet : Ls >
   < L : C-Physical-Link | >
=> < N : C-Physical-Network | LinkSet : L Ls >
   < L : C-Physical-Link | Nodes : << No1;No2 >> > .
```

Usando este proceso de creación de objetos está garantizado que la topología de red resultante es consistente, y no existirán valores de atributos relacionados con objetos que no existan. La consistencia debe mantenerse en el proceso de borrado imponiendo que los objetos relacionados con otros objetos no pueden ser eliminados sin eliminar previamente el valor del atributo en el objeto relacionado. Obsérvese que en las relaciones explícitas de grupo es suficiente con eliminar el valor de un atributo y no es necesario eliminar el objeto completo como en la relación de contenido.

3.2.3. Mensajes de acceso a la información

Se muestran dos mensajes para acceder a la información del sistema: el primero obtiene el valor de un atributo que está definido en el modelo, y el segundo calcula el valor requerido a partir de valores existentes de los atributos. En ambos casos, la pregunta es enviada a un objeto de clase `C-Network`, como raíz del sistema, que envía mensajes a los componentes apropiados de la red para obtener el valor requerido. A continuación, el nodo raíz recapitula la información contenida en los mensajes de retorno con el fin de enviar un único mensaje al objeto externo que realizó la pregunta original.

El mensaje `LinkLoad?(O,N,L)` obtiene la carga de un enlace seleccionado `L` en una red `N` y responde al objeto externo `O`. Se utilizan dos reglas para implementar este mensaje:

```
rl [Link-load] : LinkLoad?(O,N,L) < N : C-Network | LinkSet : L Ls >
=> < N : C-Network | > (L.Load reply to N and O) .

rl [Link-load] : (to N and O, L.Load is Ld)
   < N : C-Network | LinkSet : L Ls >
=> < N : C-Network | > (To O LinkLoad L is Ld in N) .
```

donde las variables usadas se declaran como

```
var O : Oid .
var L : LinkOid .
var Ld : Nat .
var Ls : LinkOidSet .
var N : NetworkOid .
```

La primera regla envía la pregunta al objeto de clase `C-Link` apropiado. Esta regla solo se aplica cuando el enlace `L` aparece en el conjunto de enlaces de la red, dado por el atributo `LinkSet`, tal como se requiere en el lado izquierdo de la regla al declarar el valor del atributo `LinkSet` como el conjunto `L Ls`, donde `L` es el enlace definido en el mensaje y `Ls` es un conjunto de identificadores de objetos de la clase `C-Link`. La segunda regla recibe los acuses de recibo de los enlaces indicando el valor `Ld` del atributo `Load`, y envía el correspondiente acuse de recibo al objeto externo que realizó la pregunta original. De esta forma los únicos objetos del sistema que interaccionan con los objetos externos son los objetos de clase `C-Network`.

Los mensajes `(L.Load reply to N and O)` y `(to N and O, L.Load is Ld)` obtienen el valor de un atributo definido en una clase y que no está oculto —en este caso el valor del atributo `Load` de un enlace `L`— y lo envían a la red `N` de la forma descrita en [Mes93a], aunque todavía no implementada en Maude. El objeto externo `O` es necesario en el mensaje para identificar la dirección de retorno apropiada.

El mensaje `NodeLinks?(O,N,No)` obtiene todos los enlaces de la red `N` que tienen un nodo dado `No` como origen o como destino. Para obtener esta información debe investigarse el sistema completo. Se difunde un mensaje a todos los enlaces de la red `N`, se recopilan aquellos enlaces que tienen un extremo en el nodo `No` y se envían al objeto externo `O`. Las reglas correspondientes son

```
vars No M1 M2 : NodeOid .
var Ns : NodeOidSet .
var Ls1 : LinkOidSet .

rl [Node-links] : NodeLinks?(O,N,No)
   < N : C-Network | NodeSet : No Ns, LinkSet : Ls >
=> FindLink(O,N,No,Ls,null) < N : C-Network | > .

crl [Find-link] : FindLink(O,N,No,L Ls,Ls1)
   < L : C-Link | Nodes : << M1;M2 >> >
=> < L : C-Link | > FindLink(O,N,No,Ls,L Ls1)
if (M1 == No) or (M2 == No) .

crl [Find-link] : FindLink(O,N,No,L Ls,Ls1)
   < L : C-Link | Nodes : << M1;M2 >> >
=> < L : C-Link | > FindLink(O,N,No,Ls,Ls1)
if (M1 =/= No) and (M2 =/= No) .

rl [Find-link] : FindLink(O,N,No,null,Ls)
=> (To N and O NodeLinks No are Ls) .

rl [Node-links] : (To N and O NodeLinks No are Ls) < N : C-Network | >
=> < N : C-Network | > (To O in N Node No Links Ls) .
```

La primera y la última regla están implementadas en el módulo orientado a objetos que define la clase `C-Network`. Se utilizan dos atributos de la clase `C-Network` en la primera regla: `NodeSet`, que declara el conjunto de nodos que pertenecen a la red y cuyo valor para

la red `N` incluye el identificador `No` del objeto de clase `C-Node` dado en el mensaje y un conjunto de identificadores de objetos de clase `C-Node Ns`; y `LinkSet`, que declara el conjunto de enlaces que pertenecen a la red y está definido como el conjunto de identificadores de objetos de clase `C-Link Ls`.

Las tres reglas que implementan el mensaje `FindLink` pertenecen al módulo orientado a objetos que define la clase `C-Link`. Una vez que la red `N` difunde la pregunta a todos sus enlaces, las reglas `FindLink` recopilan los enlaces que tienen el nodo dado como uno de sus extremos añadiéndolos de uno en uno al conjunto de identificadores de enlaces definido en el último parámetro del mensaje `FindLink(O,N,No,Ls,Ls1)`, y envían un nuevo mensaje a los restantes enlaces en el conjunto `Ls` hasta que no hay más enlaces en este conjunto. Cuando se han tratado todos los enlaces de la red, se envía un mensaje `(To N and O NodeLinks No are Ls)` con el conjunto `Ls` de todos los enlaces que cumplen la condición dada al objeto de red `N`, que a continuación responde al objeto externo `O`.

3.2.4. Mensajes de modificación

El mensaje `ChDemand(O,N,No1,No2,« S;D »)` cambia la demanda de ancho de banda de la conexión entre los nodos `No1` y `No2` en la red `N`, añadiendo el ancho de banda `D` del servicio `S` (expresado mediante el par `« S;D »`) a la demanda existente en la conexión. Si la modificación se realiza, se envía el mensaje `(To O AckChDemand No1 and No2 in N)` al objeto externo; en otro caso, se envía un mensaje indicando la razón por la que el cambio no ha tenido lugar.

Asumiendo que

- la topología de la red es correcta, esto es, los nodos definidos como puntos de terminación de los enlaces y los enlaces definidos en una conexión forman parte de la configuración de la red, y que la aplicación de las reglas de reescritura en el nivel objeto garantiza la corrección de la topología de la red, y que
- no existe ninguna restricción en añadir puertos a un nodo si la capacidad del puerto está soportada por el nodo,

solo se utilizan dos mensajes de error: `(To O NoConnectionBetween No1 and No2 in N)` para el caso en que no exista una conexión definida entre los nodos dados, y `(To O ServiceCapacityNoSupported)` para el caso en que no existe un puerto de la capacidad requerida en alguno de los nodos por los que pasa la conexión. Por simplicidad asumimos que si hay puertos de una cierta capacidad en un nodo de una capa superior, entonces hay puertos de esta capacidad o de capacidad superior en los nodos de las capas inferiores.

La Figura 3.4 muestra el protocolo que sigue el proceso de modificación en una capa. Primero, se envían mensajes a la conexión, enlaces y nodos relacionados con el cambio requerido, para verificar si la modificación es posible. Si el mensaje de retorno indica que la modificación puede en efecto ser realizada, entonces el proceso de modificación comienza cambiando la demanda de servicio en la conexión, el ancho de banda requerido en los enlaces que soportan la conexión, y el número de puertos en los nodos que se atraviesan; en otro caso, se envía al objeto externo `O` a través del objeto de red el correspondiente

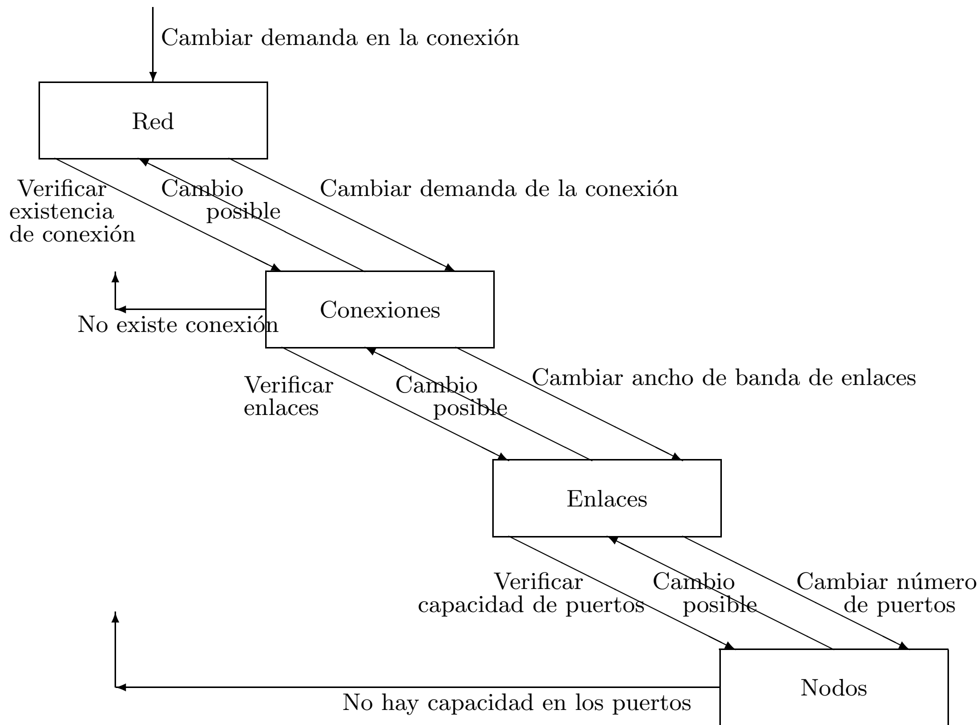

Figura 3.4: Proceso de modificación

mensaje de error. En esta versión del proceso de modificación, las operaciones `ChDemand` se ejecutan únicamente en serie, esto es, el objeto de red no acepta una nueva petición hasta que la última haya sido contestada.

El objeto de red `N` envía la pregunta a su conjunto de conexiones, usando el mensaje `ComMod(O,N,No1,No2,Cs,« S;D »)`, por medio de la regla:

```
var D : Nat .
var S : ServiceOid .
vars No1 No2 : NodeOid .
var Cs : ConnectionOidSet .

rl [Ch-demand] : ChDemand(O,N,No1,No2,<< S;D >>)
   < N : C-Network | NodeSet : No1 No2 Ns, ConnectionSet : Cs >
=> < N : C-Network | > ComMod(O,N,No1,No2,Cs,<< S;D >>) .
```

El conjunto de conexiones se explora conexión a conexión. Si se encuentra una conexión entre los nodos requeridos `No1` y `No2`, entonces se envía un nuevo mensaje a la lista de enlaces que forman la conexión para verificar si la modificación es posible; en otro caso, se envía un mensaje de error al objeto de red `N`, el cual lo reenvía al objeto externo `O`. Una versión diferente y más compleja sería crear la conexión cuando no exista. Las correspondientes reglas son las siguientes:

```
vars M1 M2 : NodeOid .
var Ll : LinkOidList .

crl [Com-mod1] : ComMod(O,N,No1,No2,C Cs,<< S;D >>)
   < C : C-Connection | Nodes : << M1;M2 >>, LinkList : Ll >
=> < C : C-Connection | > LinkListLoadReq(O,N,C,<< S;D >>,Ll )
if ((M1 == No1) and (M2 == No2)) or ((M2 == No1) and (M1 == No2)) .

crl [Com-mod1] : ComMod(O,N,No1,No2,C Cs,<< S;D >>)
   < C : C-Connection | Nodes: << M1;M2 >> >
=> < C : C-Connection | > ComMod(O,N,No1,No2,Cs,<< S;D >>)
if ((M1 =/= No1) or (M2 =/= No2)) and ((M2 =/= No1) or (M1 =/= No2)) .

rl [Com-mod1] : ComMod(O,N,No1,No2,null,<< S;D >>)
=> (To N and O NoConnectionBetween No1 to No2) .

rl [error] : (To N and O NoConnectionBetween No1 to No2)
   < N : C-Network | >
=> < N : C-Network | > (To O NoConnectionBetween No1 to No2 in N) .
```

Si se encuentra la conexión entre los nodos requeridos en la red, el proceso de verificación continúa con el mensaje `LinkListLoadReq(O,N,C,« S;D »,Ll)`, que recorre la lista de enlaces que forman la conexión, enviando mensajes a los nodos que forman sus puntos de terminación para verificar la capacidad de sus puertos. Las reglas son como sigue:

```
vars L L1 : LinkOid .

rl [Link-list-req] : LinkListLoadReq(O,N,C,<< S;D >>,L Ll)
   < L : C-Link | Nodes : << No1;No2 >> >
=> < L : C-Link | > PortNodeReq(O,N,C,L,<< S;D >>,No1)
   PortNodeReq(O,N,C,L,<< S;D >>,No2)
   LinkListLoadReq2(O,N,C,<< S;D >>,Ll,L,No1,No2) .

rl [Link-list-req] : LinkListLoadReq2(O,N,C,<< S;D >>,L Ll,L1,No1,No2)
   (To L1 and O and N and C and << S;D >> PortInNode No1)
   (To L1 and O and N and C and << S;D >> PortInNode No2)
   < L : C-Link | Nodes : << M1;M2 >> >
=> < L : C-Link | > PortNodeReq(O,N,C,L,<< S;D >>,M1)
   PortNodeReq(O,N,C,L,<< S;D >>,M2)
   LinkListLoadReq2(O,N,C,<< S;D >>,Ll,L,M1,M2) .

rl [Link-list-req] : LinkListLoadReq2(O,N,C,<< S;D >>,nil,L,No1,No2)
   (To L and O and N and C and << S;D >> PortInNode No1)
   (To L and O and N and C and << S;D >> PortInNode No2)
=> (To C and O and N and << S;D >> LinksVerified) .
```

Se utiliza un nuevo mensaje `LinkListLoadReq2` para recorrer la lista y recopilar los mensajes de respuesta de los nodos. Este mensaje auxiliar se utiliza únicamente para la implementación de este proceso y no será invocado por ningún otro objeto. En este caso,

sería de gran utilidad un mecanismo de encapsulamiento, ya que evitaría un posible uso erróneo de este mensaje por cualquier otro objeto. Es posible construir un iterador *en el meta-nivel*, como hacemos posteriormente en la Sección 3.3, que recorre listas genéricas y simplifica las reglas en el nivel objeto, pero el lenguaje no proporciona esta clase de operador en el nivel objeto.

Finalmente, el mensaje `PortNodeReq(O,N,C,L,« S;D »,No)` se envía al nodo `No` para verificar si tiene puertos para transportar por lo menos el ancho de banda requerido por el servicio `S`, y de este modo terminar el proceso de verificación. La implementación de esta regla en la capa física es diferente de la implementación en las otras dos capas, debido a que el equipo de los nodos de transmisión es diferente del equipo de los nodos de conmutación. Las reglas para los nodos de la capa física son:

```
vars Cap Cp : Nat .
var E : EquipmentOid .

crl [Port-node-req] : PortNodeReq(O,N,C,L,<< S;D >>,No)
   < No : C-Physical-Node | EqTrans : < E : C-Eq-Trans | Capacity: Cp > >
   < S : C-Service | Capacity : Cap >
=> < No : C-Physical-Node | > < S : C-Service | >
   (To L and O and N and C and << S;D >> PortInNode No)
if (Cap <= Cp) .

crl [Port-node-req] : PortNodeReq(O,N,C,L,<< S;D >>,No)
   < No : C-Physical-Node | EqTrans: < E : C-Eq-Trans | Capacity: Cp > >
   < S : C-Service | Capacity : Cap >
=> < No : C-Physical-Node | > < S : C-Service | >
   (To N and O NoPortCapacity Cp Node No)
if (Cap > Cp) .
```

Las reglas comprueban que la capacidad del puerto `Cp` es suficiente para transportar una comunicación. Obsérvese que el valor del atributo `EqTrans` es una configuración que consta de un objeto de clase `C-Eq-Trans` que define el equipo de transmisión en lugar de un identificador de un objeto de esta clase. La razón es que se ha declarado una relación de contenido entre los nodos y el equipo asociado, y esta relación se implementa usando configuraciones como el valor de los correspondientes atributos como se explicó en la Sección 3.2.2.

Una vez que se ha verificado que todos los enlaces y nodos de la conexión pueden soportar el cambio de demanda, el objeto de clase `C-Connection` comienza el proceso de modificación por medio del mensaje `ComMod2(O,N,C,« S;D »,Dl)`, como indica la siguiente regla

```
rl [Links-verified] : (To C and O and N and << S;D >> LinksVerified)
   < C : C-Connection | DemandList : Dl >
=> < C : C-Connection | > ComMod2(C,<< S;D >>,Dl)
   (To N and O ConnectionModified C) .
```

donde `Dl` denota una lista de tuplas, cada una formada por un identificador del objeto

correspondiente al servicio que se desea modificar y el número de comunicaciones de este servicio que deben añadirse a la conexión.

El mensaje `ComMod2` recorre la lista de demanda `Dl` buscando una tupla cuyo identificador de servicio sea el utilizado en el mensaje. Si este identificador existe se modifica la demanda; en otro caso, se añade el servicio a la lista de demanda. En ambos casos se envía un mensaje para modificar los enlaces.

```
vars S S1 S2 : ServiceOid .
vars D1 D2 : Nat .
vars Dl1 Dl2 : DemandList .

rl [Com-mod2] : ComMod2(C,<< S;D1 >>,<< S;D2 >> Dl)
   < C : C-Connection | DemandList: Dl1 << S;D2 >> Dl2, LinkList: Ll >
   < S : C-Service | Capacity : Cp >
=> < C : C-Connection | DemandList : Dl1 << S;D1 + D2 >> Dl2 >
   < S : C-Service | > ChLinkListLoad(Ll,D1 * Cp) .

crl [Com-mod2] : ComMod2(C,<< S1;D1 >>,<< S2;D2 >> Dl)
   < C : C-Connection | >
=> < C : C-Connection | > ComMod2(C,<< S1;D1 >>,Dl)
if (S1 =/= S2) .

rl [Com-mod2] : ComMod2(C,<< S;D1 >>,nil)
   < C : C-Connection | DemandList : Dl, LinkList : Ll >
   < S : C-Service | Capacity : Cp >
=> < C : C-Connection | DemandList : Dl << S;D1 >> >
   < S : C-Service | > ChLinkListLoad(Ll,D1 * Cp) .
```

El mensaje `ChLinkListLoad(Ll,R)` modifica la carga de los enlaces de la lista `Ll` añadiéndoles el ancho de banda `R`. Como en el caso del mensaje `PortNodeReq`, las reglas que implementan este mensaje son diferentes en la capa física y en las otras dos capas. Las reglas correspondientes a la capa física son:

```
vars Ld R : Nat .

rl [ChLinkListLoad] : ChLinkListLoad(L Ll,R)
   < L : C-Physical-Link | Nodes : << No1;No2 >>, Load : Ld >
=> < L : C-Physical-Link | Load : Ld + R >
   ChPortNode(No1,R) ChPortNode(No2,R) ChLinkListLoad(Ll,R) .

rl [ChLinkListLoad] : ChLinkListLoad(nil,R) => nil .
```

Finalmente, el mensaje `ChPortNode(No,R)` modifica el número de puertos que se utilizan en el nodo `No` de acuerdo con el nuevo ancho de banda `R`. Una vez más, las reglas para la capa física son diferentes de las reglas para las otras dos capas.

```
rl [ChPortNode] : ChPortNode(No,R) < No : C-Physical-Node |
   EqTrans : < E : C-Eq-Trans | Capacity : Cp >, Used : U >
=> < No : C-Physical-Node | Used : U + R div Cp > .
```

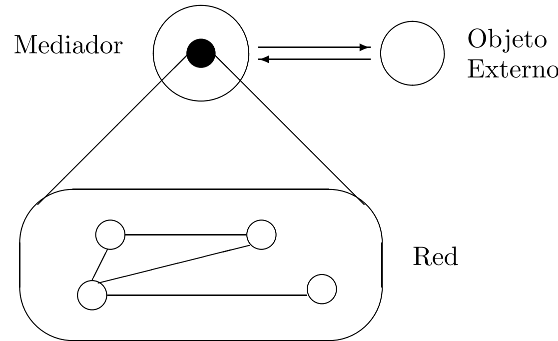

Figura 3.5: Mediador del meta-nivel para una red

3.3. Especificación de la red usando reflexión

En esta sección usamos reflexión para especificar el proceso de modificación esbozado en la Figura 3.4. El proceso de reescritura se controla mediante el lenguaje de estrategias presentado en la Sección 2.2.5. Las operaciones `apply` e `Iterate` nos permiten definir la secuencia de acciones que tiene lugar en el proceso de modificación.

Todos los objetos que constituyen una red forman una configuración en el sentido que se describe en [Mes93a], esto es, forman un estado distribuido de un sistema concurrente orientado a objetos en el nivel objeto. Para controlar una red, definimos *en el meta-nivel* una clase de *meta-objetos* denominados *mediadores*:

```
class Mediator | Config : Network-Configuration .
```

donde `Network-Configuration` es un subtipo del tipo predefinido de Maude `Term`, esto es, el valor del atributo `Config` es un meta-término representando *en el meta-nivel* el multiconjunto de objetos que constituyen la red.

Un *mediador* tiene como valor de su atributo `Config` un meta-término de la forma $\overline{\texttt{C}}$, donde `C` denota la configuración de la red gestionada por el mediador. Asumimos que todo el control de la red se realiza a través de su mediador, que toma el papel de la red en el nivel objeto. Esto es, los mensajes dirigidos a la red desde el objeto externo son procesados por el mediador, que también es el encargado de devolver la correspondiente respuesta. Esta situación se muestra en la Figura 3.5, y constituye una aplicación novedosa de las ideas de la reflexión lógica a la descripción de la reflexión de grupo orientada a objetos esbozada ya por Meseguer en [Mes93a].

3.3.1. Estrategias y reglas para el proceso de modificación utilizando reflexión

Un primer enfoque para controlar el proceso de modificación a través del mediador sigue el protocolo descrito en la Figura 3.4. Definimos dos reglas adicionales en el meta-nivel. La primera se aplica cuando la modificación es posible, mientras que la otra trata los casos de error. La idea es reducir la expresión de estrategia que modifica la demanda

en la configuración de red tanto como sea posible, encajando la expresión de estrategia resultante con el lado izquierdo de la ecuación de encaje de patrones de las reglas siguientes. Las ecuaciones de encaje de patrones se interpretan matemáticamente como ecuaciones ordinarias; sin embargo, pueden tener nuevas variables que no aparecen en el lado izquierdo de la correspondiente regla (ver Sección 2.2.1). En la ejecución de una ecuación de encaje de patrones, estas nuevas variables se instancian encajando el lado izquierdo de la ecuación de encaje de patrones con el lado derecho. La configuración se cambia entonces de acuerdo con el encaje dado o se envía un mensaje de error al objeto externo.

```
crl [ChDemand1] : ChDemand(O,N,No1,No2,<< S;D >>)
   < N : Mediator | Config : T >
=> < N : Mediator | Config : T' > (To O AckChDemand No1 and No2 in N)
if rew T' in M-NET with idle := rew Tc in M-NET with stratChDemand1 .

crl [ChDemand1] : ChDemand(O,N,No1,No2,<< S;D >>)
   < N : Mediator | Config : T >
=> < N : Mediator | > (To O NoChDemand No1 and No2 in N)
if rew Tc in M-NET with stratChDemand1 == failure .
```

donde el meta-término `T` representa un estado correcto de la red, `Tc` denota la configuración `__[T,ConnectionReq(O,N,No1,No2,« S;D »)`, `M-NET` es el módulo que define la red, y `stratChDemand1` denota la estrategia

```
apply(ConnectionReq); apply(LinkListLoadReq);
Iterate(apply(PortNodeReq); apply(PortNodeReq); apply(LinkListLoadReq));
apply(ChConnection); Iterate(apply(ChConnection));
Iterate(apply(ChLinkListLoad); apply(ChPortNode); apply(ChPortNode)) .
```

Las constantes `ConnectionReq`, `LinkListLoadReq`, `ChConnection`, `ChLinkListLoad`, `PortNodeReq`, y `ChPortNode` son las etiquetas de las reglas definidas en el nivel objeto, que se describen a continuación para el caso reflexivo.

Cuando el mediador de la red recibe un mensaje `ChDemand` se evalúa la condición de la regla, para poder aplicar una de las reglas `ChDemand1` anteriores. Esto requiere reducir la expresión de estrategia

```
rew Tc in M-NET with stratChDemand1 .                                    (1)
```

usando las ecuaciones del lenguaje de estrategias. Primero, se intenta aplicar la regla `ConnectionReq` en el nivel objeto, la cual verifica la existencia de una conexión entre los nodos `No1` y `No2` en la red `N`.

```
crl [ConnectionReq] : ConnectionReq(O,N,No1,No2,<< S;D >>)
   < C : C-Connection | Nodes : << M1;M2 >>, LinkList : Ll >             (2)
=> < C : C-Connection | > LinkListLoadReq(O,N,C,<< S;D >> ,Ll)
if ((M1 == No1) and (M2 == No2)) or ((M2 == No1) and (M1 == No2)) .
```

Si la conexión existe entonces la expresión de estrategia inicial (1) se reduce a

```
rew getTerm(metaApply(M-NET,Tc,ConnectionReq,none,0)) in M-NET
with apply(LinkListLoadReq);
Iterate(apply(PortNodeReq); apply(PortNodeReq);
        apply(LinkListLoadReq));                                    (3)
apply(ChConnection); Iterate(apply(ChConnection));
Iterate(apply(ChLinkListLoad); apply(ChPortNode);apply(ChPortNode)) .
```

En otro caso, la expresión (1) se reduce a `failure`, porque la regla `ConnectionReq` no puede aplicarse, y el fallo se propaga a través del resto de la expresión. En este caso, se satisface la condición de la segunda regla y se envía el mensaje `(To O NoChDemand No1 and No2 in N)` al objeto externo `O`.

Si no hay error, la expresión de estrategia (3) se reduce a[1]:

```
rew __[T,↑ LinkListLoadReq(O,N,C,<< S;D >>,Ll)] in M-NET
with apply(LinkListLoadReq);
Iterate(apply(PortNodeReq); apply(PortNodeReq);
        apply(LinkListLoadReq));
apply(ChConnection); Iterate(apply(ChConnection));
Iterate(apply(ChLinkListLoad); apply(ChPortNode); apply(ChPortNode)) .
```

Las reglas `LinkListLoadReq` envían mensajes a los nodos del enlace para verificar si soportan la capacidad pedida por medio de la regla `PortNodeReq`. Con el fin de tratar todos los enlaces, se utiliza la operación `Iterate` definida en la Sección 2.2.5. Sin embargo, dado que las reglas `LinkListLoadReq` puede aplicarse más de una vez a un objeto, hay que asegurarse de que cada enlace de la lista se trata exactamente una vez. Para conseguirlo, usamos un parámetro de tipo lista en las reglas de reescritura `LinkListLoadReq` que controla la lista de enlaces que todavía no ha sido tratada. Cuando todos los enlaces se han encajado con la regla y la lista de enlaces no tratados se encuentra vacía, la estrategia `Iterate(apply(PortNodeReq); apply(PortNodeReq);apply(LinkListLoadReq))` fallará al aplicarse y la operación `_;;_orelse_` finalizará con la estrategia `idle`.

```
rl [LinkListLoadReq] : LinkListLoadReq(O,N,C,<< S;D >>,L Ll)
   < L : C-Link | Nodes : << No1;No2 >> >
=> < L : C-Link | > PortNodeReq(S,No1)                              (4)
   PortNodeReq(S,No2) LinkListLoadReq(O,N,C,<< S;D >>,Ll) .

rl [LinkListLoadReq] : LinkListLoadReq(O,N,C,<< S;D >>,nil)
   < C : C-Connection | DemandList : Dl >
=> < C : C-Connection | > ChConnection(C,<< S;D >>,Dl) .

crl [PortNodeReq] : PortNodeReq(S,No)
    < No : C-Node | Eq : < Et : C-Eq-Trans | Capacity : Cap > >
    < S : C-Service | Capacity : Cp >
=> < No : C-Node | > < S : C-Service | >
if (Cp <= Cap) .
```

[1] `↑LinkListLoadReq(O,N,C,« S;D »,Ll)` denota el término $\overline{\texttt{LinkListLoadReq(O,N,C,« S;D »,Ll)}}$.

La aplicación de la ecuación recursiva `Iterate` continúa hasta qué bien algún nodo produce un error o bien se hayan tratado todos los enlaces de la conexión.

En el primer caso, la expresión de estrategia se reduce a `failure` ya que la operación `Iterate` termina sin enviar un mensaje `ChConnection` en el nivel objeto y entonces la regla `ChConnection` no puede aplicarse. En otro caso, se termina la parte de modificación del proceso utilizando el resto de la estrategia

```
apply(ChConnection);Iterate(apply(ChConnection));
Iterate(apply(ChLinkListLoad); apply(ChPortNode); apply(ChPortNode))
```

que no puede fallar, porque la parte de verificación garantiza que la modificación es posible.

El mensaje `ChConnection` recorre la lista de demanda buscando una tupla cuyo identificador del objeto de clase `C-Service` es el usado en el mensaje. La operación `Iterate` se utiliza hasta que se encuentra el servicio en la lista de demanda o la lista de demanda es `nil`. La operación `Iterate` se reduce en este caso a `idle` y la regla `ChLinkListLoad` se puede aplicar. Es equivalente al mensaje `ComMod2` del caso no reflexivo.

```
rl [ChConnection] : ChConnection(C,<< S;D1 >>,<< S;D2 >> Dl)
   < C : C-Connection | DemandList : Dl1 << S;D2 >> Dl2,
                        LinkList : Ll >
   < S : C-Service | Capacity : Cp >
=> < C : C-Connection | DemandList : Dl1 << S;D1 + D2 >> Dl2 >
   < S : C-Service | > ChLinkListLoad(Ll,D1 * Cp) .

crl [ChConnection] : ChConnection(C,<< S1;D1 >>,<< S2;D2 >> Dl)
   < C : C-Connection | >
=> < C : C-Connection | > ChConnection(C,<< S1;D1 >>,Dl)
if (S1 =/= S2) .

rl [ChConnection] : ChConnection(C,<< S;D1 >>,nil)
   < C : C-Connection | DemandList : Dl, LinkList : Ll >
   < S : C-Service | Capacity : Cp >
=> < C : C-Connection | DemandList : Dl << S;D1 >> >
   < S : C-Service | > ChLinkListLoad(Ll,D1 * Cp) .
```

La regla `ChLinkListLoad` se modifica ligeramente respecto al caso no reflexivo. Ahora el mensaje `ChLinkListLoad` no se genera para la lista vacía.

```
crl [ChLinkListLoad] : ChLinkListLoad(L Ll,R)
   < L : C-Physical-Link | Nodes : << No1;No2 >>, Load : Ld >
=> < L : C-Physical-Link | Load : Ld + R > ChPortNode(No1,R)
   ChPortNode(No2,R) ChLinkListLoad(Ll,R)
if (Ll =/= nil) .

crl [ChLinkListLoad] : ChLinkListLoad(L Ll,R)
   < L : C-Physical-Link | Nodes : << No1;No2 >>, Load : Ld >
=> < L : C-Physical-Link | Load : Ld + R >
   ChPortNode(No1,R) ChPortNode(No2,R)
if (Ll == nil) .
```

Finalmente la regla `ChPortNode` no cambia respecto al caso no reflexivo y queda como se describe al final de la Sección 3.2.4.

Las reglas se aplican por medio de la operación `Iterate` hasta que la lista de enlaces está vacía y la operación finaliza con `idle`. La expresión de estrategia se reduce así completamente produciendo un meta-término `T'` que representa la configuración de la red con la demanda modificada. Se satisface entonces la condición de la primera regla y la configuración de la red se cambia por `T'`.

3.3.2. Comparación de las dos especificaciones

El uso de la reflexión simplifica las reglas en el nivel objeto, principalmente debido al control ejercido por el meta-nivel sobre las situaciones de error y el orden de aplicación de las reglas.

La regla `ConnectionReq` (2), usada para verificar la existencia de conexión entre dos nodos, sustituye a las tres reglas `Com-mod1` definidas en la Sección 3.2.4.

Estas reglas necesitan un parámetro extra, el conjunto de conexiones de la red, que se utiliza para recorrer las conexiones de la red hasta que se encuentra la conexión entre los dos nodos dados, o se verifica que la conexión no existe. La primera regla trata la existencia de una conexión entre los nodos dados, la segunda regla se utiliza para recorrer el conjunto de conexiones obviando aquellas no definidas entre los nodos dados, y la tercera regla se utiliza para enviar el mensaje de error una vez que todas las conexiones han sido comprobadas y no se ha encontrado ninguna entre los nodos dados.

Por otra parte, usando la reflexión una única regla es suficiente para tratar la existencia de una conexión. Si la condición de esta regla no encaja con los nodos dados, la estrategia en el meta-nivel es la encargada de enviar el mensaje de fallo al objeto externo. De esta forma se evitan gran cantidad de mensajes, porque no hay necesidad de recorrer el conjunto de todas las conexiones de la red.

El conjunto de reglas `LinkListLoadReq` es otro ejemplo de simplificación de las reglas usando reflexión. Las dos reglas (4) sustituyen a las tres reglas definidas en la Sección 3.2.4.

Mantener el control en el nivel objeto requiere el mensaje adicional `LinkListLoadReq2` que se utiliza únicamente de forma interna para la implementación. La segunda y la tercera reglas (Sección 3.2.4) definen el comportamiento de este mensaje. Se utilizan para recorrer la lista de enlaces que soportan la conexión, enviando el mensaje apropiado a sus nodos para verificar si pueden soportar la capacidad pedida. Al mismo tiempo, las reglas recopilan los mensajes de retorno satisfactorios de los nodos del enlace anterior de la lista. Solo en el caso de que todos los nodos de los enlaces de la lista puedan soportar la capacidad pedida, se envía un mensaje de éxito por medio de la tercera regla al objeto conexión.

Usando reflexión, se evita este mensaje extra, manteniendo la especificación libre de estas cuestiones de implementación. La regla `LinkListLoadReq` se utiliza para recorrer la lista de enlaces. Si un nodo no puede soportar la capacidad pedida, entonces la regla `PortNodeReq` no encajará y la estrategia generará de forma automática un fallo en el meta-nivel, eliminando la necesidad de recopilar los mensajes de retorno de los nodos. Como en el caso anterior, esto evita el uso de una gran cantidad de mensajes.

3.3.3. Mejorando el control al cambiar las estrategias

El lenguaje de estrategias nos permite simplificar no solo las reglas sino también el protocolo llevando a cabo simultáneamente los procesos de verificación y de modificación. También es posible diferenciar las dos situaciones de fallo y enviar mensajes diferentes al objeto externo como en el caso no reflexivo. Considérense las tres reglas siguientes:

```
crl [ChDemand2] : ChDemand(O,N,No1,No2,<< S;D >>)
   < N : Mediator | Config : T >
=> < N : Mediator | Config : T' >
   (To O AckChDemand No1 and No2 in N)
if rew T' in M-NET with idle := rew T'c in M-NET with stratChDemand2 .

crl [ChDemand2] : ChDemand(O,N,No1,No2,<< S;D >>)
   < N : Mediator | Config : T >
=> < N : Mediator | > (To O NoConnectionBetween No1 and No2 in N)
if rew T' in M-NET with NoConnection :=
   rew T'c in M-NET with stratChDemand2 .

crl [ChDemand2] : ChDemand(O,N,No1,No2,<< S;D >>)
   < N : Mediator | Config : T >
=> < N : Mediator | > (To O ServiceCapacityNoSupported)
if rew T' in M-NET with NoPortCapacity :=
   rew T'c in M-NET with stratChDemand2 .
```

donde el meta-término `T'c` denota la configuración `__[T,MCom(O,N,No1,No2,« S;D »)]` (con `MCom(O,N,No1,No2,« S;D »)` sobrerrayado) y `stratChDemand2` denota la estrategia

```
(apply(MCom) ;; Iterate(apply(LinkListLoad);
    (apply(PortNode) ;; idle orelse NoPortCapacity);
    (apply(PortNode) ;; idle orelse NoPortCapacity))
orelse (apply(MComNS) ;; Iterate(apply(LinkListLoad);
    (apply(PortNode) ;; idle orelse NoPortCapacity);
    (apply(PortNode) ;; idle orelse NoPortCapacity))
orelse NoConnection)) .
```

La idea principal es usar la operación `_;;_orelse_`, que nos permite verificar si las reglas apropiadas se pueden aplicar en el nivel objeto. En el caso de que el proceso tenga éxito, la estrategia finaliza con `idle`; en otro caso, se genera una estrategia especial (bien `NoConnection` o bien `NoPortCapacity`), y el proceso de reducción finaliza con la operación `_;;_orelse_` o aplicando una de las siguientes ecuaciones:

```
eq NoConnection andthen S = NoConnection .
eq (rew T in M with NoPortCapacity) andthen S =
    rew T in M with NoPortCapacity .
```

En el nivel objeto las reglas `MCom` y `MComNS` integran las reglas `Com-mod1` y `Com-mod2` definidas en la Sección 3.2.4. Estas reglas verifican la existencia de una conexión entre

los nodos dados y cambian el atributo `DemandList` del objeto conexión. Si el servicio ya está definido, se aplica la regla `MCom` y se incrementa la demanda del servicio; en otro caso, no se puede encajar la regla `MCom` y se aplica `MComNS`, añadiendo el nuevo servicio a la lista de demanda.

```
crl [MCom] : MCom(O,N,No1,No2,<< S;D >>)
   < C : C-Connection | Nodes : << M1;M2 >>,
         DemandList : Dl1 << S;D1 >> Dl2, LinkList : Ll >
   < S : C-Service | Capacity : Cp >
=> < C : C-Connection | DemandList : Dl1 << S;D1 + D >> Dl2 >
   < S : C-Service | > LinkListLoad(S,Ll,D * Cp)
if ((M1 == No1) and (M2 == No2)) or ((M1 == No2) and (M2 == No1)) .

crl [MComNS] : MCom(O,N,No1,No2,<< S;D >>)
   < C : C-Connection | Nodes : << M1;M2 >>, DemandList : Dl1,
                        LinkList : Ll >
   < S : C-Service | Capacity : Cp >
=> < C : C-Connection | DemandList : Dl1 << S;D >> >
   < S : C-Service | > LinkListLoad(S,Ll,D * Cp)
if ((M1 == No1) and (M2 == No2)) or ((M1 == No2) and (M2 == No1)) .
```

La regla `LinkListLoad` modifica el ancho de banda del primer enlace de la lista y envía mensajes para cambiar el número de puertos de sus dos nodos terminales.

```
crl [LinkListLoad] : LinkListLoad(S,L Ll,B)
   < L : C-Link | Nodes : << No1;No2 >>, Load : Ld >
=> < L : C-Link | Load : Ld + B > LinkListLoad(S,Ll,B)
   PortNode(S,No1,B) PortNode(S,No2,B)
if Ll =/= nil .

crl [LinkListLoad] : LinkListLoad(S,L Ll,B)
   < L : C-Link | Nodes : << No1;No2 >>, Load : Ld >
=> < L : C-Link | Load : Ld + B >
   PortNode(S,No1,B) PortNode(S,No2,B)
if Ll == nil .
```

Finalmente, la regla que verifica si un nodo soporta una capacidad dada y, si es posible, cambia el número de puertos del nodo es

```
crl [PortNode] : PortNodes(S,No,B) < S : C-Service | Capacity : Cp >
   < No : C-Node | Eq : < Et : C-Eq-Trans | Capacity : Cap >, Used : U >
=> < No : C-Node | Used : U + B DIV Cap > < S : C-Service | >
if (Cp <= Cap) .
```

Cuando el mediador de la red recibe un mensaje `ChDemand`, la condición de la regla se evalúa reduciendo la expresión de estrategia

```
rew T'c in M-NET with stratChDemand2 .                                  (5)
```

Si existe la conexión entre los nodos `No1` y `No2` en la configuración representada por `T'c`, entonces se puede aplicar bien la regla `MCom` o la regla `MComNS` en el nivel objeto, y la expresión de estrategia se reduce en ambos casos, por medio de las ecuaciones que definen la operación `_;;_orelse_`, a la expresión siguiente:

```
rew T''c in M-NET with Iterate(apply(LinkListLoad);
    (apply(PortNode) ;; idle orelse NoPortCapacity);
    (apply(PortNode) ;; idle orelse NoPortCapacity)) .
```

A continuación, se aplica la estrategia recursiva y si los nodos pueden tratar la capacidad pedida, el proceso de reducción continua. Cuando ya no haya más enlaces en la conexión, la regla `LinkListLoad` no se puede encajar y produce un fallo en la estrategia `Iterate`, que termina la aplicación de la estrategia recursiva. El proceso de reducción termina con la reducción de la expresión de estrategia a la forma `rew T' in M-NET with idle` usando la operación `metaApply`. Se satisface entonces la condición de la primera regla y se aplica la regla cambiando la configuración de la red y enviando el mensaje de éxito al objeto externo.

Si la conexión no existe, entonces la expresión de estrategia (5) se reduce al término `rew T'' in M-NET with NoConnection` que satisface la condición de la segunda regla. El caso en el cual un nodo no puede soportar la capacidad pedida se trata de forma similar generando el mensaje de error `NoPortCapacity`.

Obsérvese que este enfoque reduce todavía más el número de mensajes intercambiados en el sistema, debido a la integración de los procesos de verificación y modificación.

3.4. Conclusiones

La lógica de reescritura y el lenguaje Maude son muy apropiados para la especificación de sistemas orientados a objetos al integrar los tipos de datos algebraicos con las clases de objetos, y permitir la definición de aspectos estáticos y dinámicos de un sistema dentro del mismo lenguaje. Resultan asimismo adecuados para la especificación de las relaciones entre los objetos, especialmente la relación de contenido que puede ser especificada de forma muy elegante usando subobjetos contenidos en otros objetos y las reglas de creación y borrado propuestas. Hemos mostrado en este capítulo cómo el concepto de subconfiguración tal como se introduce en [Mes93a] ayuda a la creación de especificaciones claras y puede ser incorporado en futuras versiones de Maude.

La reflexión separa de forma clara el comportamiento del nivel objeto de los aspectos de control y gestión, incrementando la *modularidad* de la aplicación. De esta forma el nivel objeto se simplifica mucho, debido a que las reglas de reescritura se controlan en el metanivel eliminando la necesidad de reglas extra en el nivel objeto. Como hemos explicado, las estrategias internas nos permiten controlar en el metanivel

- el orden en que se aplican las reglas de reescritura,
- la gestión de situaciones de fallo, bien en general o distinguiendo diferentes tipos de fallos,

- la aplicación sucesiva de una estrategia, por medio de la operación Iterate, que nos proporciona un patrón genérico para recorrer listas, eliminando por tanto el control de las listas específicas,
- la integración de los procesos de verificación y modificación,
- los requisitos de atomicidad de algunas secuencias de transiciones que deben ejecutarse en orden y sin interrupciones por parte de otras ejecuciones.

En particular, hemos mostrado cómo la reflexión de la lógica de reescritura puede utilizarse para monitorizar funcionalidad que es intrínsecamente reflexiva en el sentido orientado a objetos como es el caso del mediador de la red.

Otra ventaja es la *adaptabilidad*. El mismo nivel objeto puede gestionarse de distintas formas cambiando la estrategia que lo controla.

Desde luego, el precio que pagamos por estas ventajas es la necesidad de ir al metanivel. Sin embargo, ya hemos hecho notar que Maude proporciona este acceso en general (debido a que la lógica de reescritura es reflexiva) así como a través del lenguaje interno de estrategias. Solo necesitamos escribir las estrategias específicas requeridas para nuestra aplicación, lo que es considerablemente más simple que complicar las reglas de reescritura del nivel objeto debido a cuestiones de control.

Otro beneficio importante proporcionado por las especificaciones en Maude es que la estructura reflexiva facilita un entorno distribuido. Se pueden definir redes distintas, que intercambien datos en el metanivel, controlando cada una de ellas su propio nivel objeto. Entonces, es fácil definir una metared que controle todas las redes definidas.

Con respecto al diseño del lenguaje Maude, en nuestra opinión existen algunos aspectos que requieren más atención. Por ejemplo, como ya hemos remarcado, algunas veces es necesario definir mensajes internos a una clase que no deben ser vistos desde el exterior de la clase. Esto no es posible en el diseño actual del lenguaje, ya que todos los mensajes reciben el mismo tratamiento. Con mayor generalidad, es necesario un estudio más detallado del tema de la *encapsulación* en el contexto de Maude.

Desde un punto de vista semejante, la distinción en Maude entre la herencia de clases y la herencia de módulos no es completamente satisfactoria, debido a la imposibilidad de redefinir atributos o mensajes en las subclases. Por una parte, esto fuerza la creación de subclases adicionales, como por ejemplo la clase `C-ATM-Node` en nuestra aplicación, y por otra parte crea relaciones de herencia entre módulos que en principio no tienen razón para existir. Más aún, desde el punto de vista del modelado, no permite la existencia en una subclase de especializaciones de métodos que son consistentes con clasificaciones de herencia de la vida real. Como hemos hecho notar para la especificación de los mensajes de coste, de la Sección 3.2.2, sería útil tener una abstracción. Entonces todas las subclases exhibirían un conjunto común de operaciones, aunque cada una de ellas tendría un comportamiento diferente.

Capítulo 4

Propiedades modales y temporales

La lógica de reescritura es principalmente una lógica *de cambio*, donde la deducción se corresponde directamente con la computación, y no una lógica para hablar *sobre el cambio* de una forma más global e indirecta, tal como las distintas lógicas modales y temporales que se pueden encontrar en la literatura.

Empezamos definiendo una lógica de acción modal (VLRL) que captura las reglas de reescritura como acciones. La novedad principal de esta lógica es una modalidad espacial asociada con los constructores de estados que nos permite razonar sobre la estructura de estos, manifestando que el estado actual se puede descomponer en regiones que satisfacen ciertas propiedades. A continuación, definimos una lógica temporal para razonar sobre propiedades de los cómputos generados por teorías de reescritura y las reglas que nos permiten verificar estas propiedades a partir de propiedades especificadas en VLRL.

La definición de la lógica y el ejemplo desarrollado de la máquina expendedora se presentaron en la conferencia *Recent Trends in Algebraic Development Techniques, WADT'99 (Chateau de Bonas, Francia)* y se publicaron en [FMMO+00]. La definición de las transiciones en contexto se ha presentado en *APPIA-GULP-PRODE'01 Joint Conference on Declarative Programming (Évora, Portugal)* publicándose en [PMO01]. Por último, el desarrollo de la modalidad espacial y en particular la definición de las propiedades espaciales básicas se han presentado en la conferencia *PROLE 2002 (El Escorial, Madrid)* publicándose en [Pit02]. Un compendio con todo el trabajo ha sido enviado para su publicación en una revista [FMMO+03].

4.1. Programas, especificaciones y verificación

Una noción fundamental en los desarrollos que vamos a describir es la noción de *programa* (o sistema software) para la lógica de reescritura. Los formalismos modales que vamos a presentar pretenden soportar la especificación y verificación de esos programas. Por lo tanto, debemos dejar claro lo que entendemos por programas, sus especificaciones y su verificación.

Los programas en lógica de reescritura manipulan clases de equivalencia de términos.

Estas clases de equivalencia están fijadas por la signatura de reescritura $\langle \Sigma, E \rangle$. Por consiguiente, una signatura, entre otras cosas, define el universo de objetos sobre los que opera un programa.

Los programas introducen la noción de cambio en este universo. Con el fin de definir el punto de referencia con respecto al cual se produce el cambio, los programas deben precisar qué objetos del universo corresponden a *estados*. Por lo tanto, asumimos que un programa determina un tipo específico *State* en Σ. Es más, un programa determina la forma en que se producen los cambios de estado. Estos vienen dados por las reglas etiquetadas, posiblemente condicionales, $r(\overline{x}) : [t(\overline{x})]_E \rightarrow [t'(\overline{x})]_E$ donde t y t' son Σ-términos de tipo *State* y $\overline{x}$ denota el conjunto de variables que aparecen en t o en t'. Formalmente, consideramos un programa sobre una signatura $\langle \Sigma, E \rangle$ como una terna $\langle State, L, R \rangle$ donde *State* es un tipo determinado de Σ, L es un conjunto de etiquetas con una aridad asociada, y R es un conjunto de reglas con etiquetas en L. Esta es la noción de teoría de reescritura usada normalmente en lógica de reescritura, excepto en lo referente a señalar el tipo de los estados, que es necesario para definir las nociones de *cómputo* y *observación* sobre las que se escribirán las especificaciones.

Las reglas de reescritura que constituyen el programa no reescriben necesariamente el estado directamente, esto es, no denotan por sí mismas transiciones de estado. En cambio, proporcionan los componentes a partir de los cuales se forman las transiciones de estado, las cuales aplican una o más reglas de reescritura en el contexto de un estado dado. Nos interesan las transiciones de estado que pueden considerarse *atómicas*, en el sentido de que, aun en el caso de que más de una reescritura pueda tener lugar durante la transición, esto sucede porque la estructura del estado permite realizar estas reescrituras de forma concurrente.

Consideramos una *especificación* como la expresión de las propiedades que debe cumplir un sistema. Con el fin de expresar estas propiedades debemos determinar de qué modo vamos a *observar* el comportamiento del sistema. Estas *observaciones* estarán determinadas por una familia *At* de *atributos* sobre los estados, cada uno de los cuales tiene un tipo asociado s. Las observaciones en que estamos interesados pueden requerir la extensión del universo fijado por la signatura, para tener disponibles tipos y operaciones auxiliares relacionadas con las observaciones, pero que no forman parte del programa original. Por lo tanto, consideramos que una especificación requiere una extensión $\langle \Sigma^+, E^+ \rangle$ de la signatura $\langle \Sigma, E \rangle$ y la familia *At* de atributos. Nótese que las observaciones no forman parte de la extensión de la signatura, de forma que podemos distinguir las operaciones interesantes desde el punto de vista de las observaciones de otras operaciones auxiliares (pero útiles), dado que en la lógica de especificación que proponemos las observaciones tienen un tratamiento especial debido a la naturaleza implícita de los estados, esto es, el estado no figura como argumento de la observación. Solo se admiten extensiones conservativas de la signatura original, con el fin de proteger el universo sobre el que se desarrollan los programas (ver el ejemplo de la Sección 4.2.2).

Como hemos mencionado anteriormente, la lógica de reescritura es esencialmente una *lógica de cambio*. Desde el punto de vista computacional, cada paso de reescritura es una transición local, posiblemente paralela, en un sistema concurrente, y desde el punto de vista lógico, cada paso de reescritura es una derivación lógica en un sistema formal. De

esta forma, un paso de reescritura define una propiedad de un único paso en un cómputo. Además, queremos definir propiedades del cómputo completo (puede ser infinito), como por ejemplo invariantes. Estas son las propiedades de seguridad (*safety*) y de vivacidad (*liveness*) expresadas en las lógicas modales y temporales. Por lo tanto, las propiedades de seguridad y vivacidad de una especificación se escriben en una lógica diferente de la de los programas; concretamente, consideraremos una lógica temporal. El problema más interesante es precisamente determinar la relación que debe establecerse entre programas y especificaciones, de forma que los programas puedan ser verificados respecto a las especificaciones. Con este propósito, hemos desarrollado una lógica, *Verification Logic for Rewriting Logic* (VLRL), que toma los programas como modelos y permite derivar propiedades de observaciones a través de reglas de inferencia que relacionan el programa y la lógica (o lógicas) de especificación. En este escenario, dados un programa P y una especificación S, verificar que P satisface S consiste en determinar un conjunto V de sentencias en VLRL tales que P es un modelo de V y de V se deriva S.

En los apartados siguientes, presentamos la lógica VLRL empezando por la signatura de verificación y la definición del lenguaje de acciones que permite capturar las reglas de reescritura como acciones de la lógica, los modelos para una signatura de verificación, el lenguaje modal y la relación de satisfacción. A continuación presentamos un procedimiento de decisión que permite comprobar si una fórmula VLRL es satisfactible en un modelo dado. Después proporcionamos la teoría de demostración utilizada en los ejemplos del Capítulo 5, discutimos algunos aspectos a tener en cuenta respecto a la completitud de la lógica y proporcionamos una serie de reglas para derivar propiedades espaciales básicas. A continuación presentamos una nueva modalidad que captura la idea de transición *en contexto*. Por último se define una lógica temporal y se proporciona un sistema de inferencia interfaz de esta lógica temporal con la lógica VLRL en forma de un conjunto de reglas que nos permiten derivar propiedades en lógica temporal a partir de propiedades en VLRL.

4.2. La lógica de verificación VLRL

4.2.1. Signaturas de verificación

Dada una signatura de reescritura $\langle \Sigma, E \rangle$, una *signatura de verificación* consta de

- un tipo determinado *State* de Σ;
- tipos y operaciones adicionales que definen una extensión Σ^+ de Σ, junto con un conjunto E^+ de ecuaciones que axiomatizan la extensión de forma que se protege[1] la signatura original, es decir, tal que $T_{\Sigma^+,E^+}|_\Sigma \simeq T_{\Sigma,E}$;
- una familia *At* de *observaciones*, cada una de las cuales tiene un tipo asociado s de Σ^+;

[1] Para una teoría $\langle \Sigma, E \rangle$ en lógica ecuacional de pertenencia, que es la versión de lógica ecuacional utilizada en Maude, la extensión que protege $\langle \Sigma^+, E^+ \rangle$ debe entenderse que da lugar a una biyección en el nivel de tipos [BJM00]. Sin embargo, los ejemplos que aquí se presentan solo implican teorías ecuacionales con tipos ordenados $\langle \Sigma, E \rangle$ para las cuales tenemos un isomorfismo real de las álgebras iniciales.

- una colección L (de etiquetas) indexada sobre cadenas de tipos en Σ. El índice corresponde a la secuencia de tipos de las variables que aparecen en la regla que define la acción asociada con la etiqueta (ver Sección 4.2.3).

La idea es definir una signatura en el sentido de otros enfoques de especificación y diseño de programas como *CommUnity* [FM97], es decir, tenemos observaciones del estado del sistema y símbolos de acción para dar cuenta de los cambios de estado elementales.

Nos serán útiles dos nociones de signatura relacionadas. Denotamos por Σ_c^+ la extensión de Σ^+ con cada observación de tipo s vista como una constante de tipo s, y por Σ_f^+ la extensión de Σ^+ con cada observación de tipo s vista como un símbolo de función de tipo $State \to s$, donde s es el tipo asociado con el atributo. La primera signatura es útil en la definición de la lógica donde, como es usual en las lógicas temporales y modales, el estado está implícito, mientras que la segunda signatura es útil cuando se presenta la especificación algebraica de las operaciones, donde el estado aparece como un argumento adicional entre otros.

Normalmente consideramos conjuntos de ecuaciones en Σ_f^+ que definen o restringen los valores que toman los atributos en los estados, dando lugar a teorías ecuacionales $\langle \Sigma_f^+, E_f^+ \rangle$ que extienden y protegen $\langle \Sigma^+, E^+ \rangle$.

4.2.2. Ejemplo: una máquina expendedora

Ilustraremos nuestro método usando como ejemplo la máquina expendedora que permite comprar bizcochos y manzanas [MOM99] que se mostró como ejemplo de módulo de sistemas en la Sección 2.2.2. Un bizcocho vale un dólar y una manzana tres cuartos. La máquina solo acepta dólares y devuelve un cuarto cuando el usuario compra una manzana. También ofrece la posibilidad de cambiar cuatro cuartos en un dólar.

La teoría de reescritura $\langle \Sigma, E, L, R \rangle$ que especifica la máquina expendedora viene dada por el siguiente módulo de sistema de Maude:

```
mod VENDING-MACHINE is
  sort State .
  ops $ q a c : -> State .
  op __ : State State -> State [assoc comm] .
  rl [buy-c] : $ => c .
  rl [buy-a] : $ => a q .
  rl [change] : q q q q => $ .
endm
```

Los posibles estados de la máquina expendedora se representan mediante clases de equivalencia de términos de tipo `State` en Σ, los cuales se corresponden con multiconjuntos *no vacíos* de recursos construidos a partir de las cuatro constantes `$`, `q`, `a`, `c` por medio del operador binario asociativo y conmutativo `__`, escrito en notación (vacía) infija. Las tres reglas de reescritura etiquetadas definen las acciones posibles de la máquina expendedora, permitiéndose a varias transiciones concurrentes tener lugar de forma simultánea.

Para obtener Σ^+, añadimos a la signatura de reescritura el tipo `Nat` de los números naturales, junto con las operaciones y ecuaciones que lo axiomatizan. Esto se denota por medio de la instrucción de importación de módulos `protecting`.

En Σ_c^+ las observaciones de interés para nosotros son $\#\$$, $\#q$, $\#c$ y $\#a$, las cuales representan el número de dólares, cuartos, bizcochos y manzanas, respectivamente, en un estado de la máquina, y *wealth*, representando el valor total de un estado, medido en cuartos.

La siguiente teoría de reescritura $\langle \Sigma_f^+, E_f^+, L, R \rangle$ extiende $\langle \Sigma, E, L, R \rangle$ añadiendo los números naturales, declarando las observaciones como funciones sobre los estados y añadiendo ecuaciones que definen estas funciones mediante inducción estructural, con la excepción de la observación *wealth*, que se define en términos de las otras observaciones por medio de una expresión aritmética.

```
mod VENDING-MACHINE-OBSERVED is
  protecting VENDING-MACHINE .
  protecting NAT .
  ops #$ #q #c #a : State -> Nat .
  op wealth : State -> Nat .
  vars P1 P2 : State .
  eq #$($) = 1 .   eq #$(q) = 0 .   eq #$(c) = 0 .   eq #$(a) = 0 .
  eq #q(q) = 1 .   eq #q($) = 0 .   eq #q(c) = 0 .   eq #q(a) = 0 .
  eq #c(c) = 1 .   eq #c($) = 0 .   eq #c(q) = 0 .   eq #c(a) = 0 .
  eq #a(a) = 1 .   eq #a($) = 0 .   eq #a(q) = 0 .   eq #a(c) = 0 .
  eq #$(P1 P2) = #$(P1) + #$(P2) .
  eq #q(P1 P2) = #q(P1) + #q(P2) .
  eq #c(P1 P2) = #c(P1) + #c(P2) .
  eq #a(P1 P2) = #a(P1) + #a(P2) .
  eq wealth(P1) = 4*#$(P1)+#q(P1)+4*#c(P1)+3*#a(P1) .
endm
```

4.2.3. El lenguaje de acciones

Como ya se ha mencionado, las reglas de reescritura no pueden ser siempre consideradas como acciones *per se*. Pueden requerir que se haga explícito el contexto en el que se aplican. Esto es posible gracias al lenguaje de acciones definido a continuación, el cual define un subconjunto de todas las reescrituras concurrentes que se realizan en un paso [Mes92] (véase la Sección 2.1.1).

Empezamos definiendo las *pre-acciones*, α. Estas se corresponden con el cociente del conjunto de términos de pruebas obtenido a partir de las siguientes reglas de deducción:

- *Identidades*[2]: para cada $[t] \in T_{\Sigma,E}(X)$, $\dfrac{}{[t] : [t] \rightarrow [t]}$,

[2] Obsérvese que $[t]$ denota tanto el estado como la correspondiente transición identidad para ese estado; esta notación resulta muy útil para distinguir el contexto en una transición.

- Σ-*estructura*: para cada $f \in \Sigma$,

$$\frac{\alpha_1 : [t_1] \longrightarrow [t'_1] \quad \ldots \quad \alpha_n : [t_n] \longrightarrow [t'_n]}{f(\alpha_1, \ldots, \alpha_n) : [f(t_1, \ldots, t_n)] \longrightarrow [f(t'_1, \ldots, t'_n)]} \ ,$$

- *Reemplazamiento*: para cada regla de reescritura $r : [t(x_1, \ldots, x_n)] \longrightarrow [t'(x_1, \ldots, x_n)]$ en R,

$$\frac{}{r(\overline{w}) : [t(\overline{w}/\overline{x})] \rightarrow [t'(\overline{w}/\overline{x})]} \ ,$$

módulo las siguientes ecuaciones:

- *Identidades*: $f([t_1], \ldots, [t_n]) = [f(t_1, \ldots, t_n)]$,
- *Axiomas en* E: $t(\overline{\alpha}) = t'(\overline{\alpha})$, para cada ecuación $t(\overline{x}) = t'(\overline{x})$ en E.

Las *acciones* son las pre-acciones que reescriben términos de tipo *State*.

Obsérvese también que la definición anterior de acciones supone que las reglas en R son incondicionales. La extensión a reglas condicionales es directa, reflejo de la correspondiente extensión del lenguaje de términos de pruebas tal como está definido en [Mes92].

Con respecto al lenguaje de los términos de pruebas asociado con la lógica de reescritura [Mes92, Sección 3.1] (ver Sección 2.1.2), hemos omitido la regla de transitividad y hemos restringido la regla de reemplazamiento a transiciones identidad. El reemplazamiento total se puede obtener como la composición secuencial de una aplicación de la regla de reemplazamiento restringido anterior y una regla de Σ-estructura para reescribir *dentro*. Nuestra intención es considerar el *modelo inicial* de una teoría de reescritura (ver la construcción de $\mathcal{T}_{\mathcal{R}}(X)$ en [Mes92], resumida en la Seccion 2.1.2) como un sistema de transiciones, pero necesitamos tener una noción de atomicidad para poder hablar del sucesor de un estado o de un estado *siguiente*.

Por lo tanto, intuitivamente, las acciones atómicas se corresponden con la aplicación paralela de reglas, pero no existe composición secuencial (debido a la falta de transitividad) ni anidamiento (porque no permitimos aplicaciones anidadas de la regla de reemplazamiento). Estas restricciones no cambian esencialmente las transiciones posibles en el sistema, ya que un resultado básico en [Mes92] demuestra que para cada prueba $[\alpha] : [t] \longrightarrow [t']$ en $\mathcal{T}_{\mathcal{R}}(X)$ bien $[t] = [t']$ y $[\alpha]$ es la identidad, o bien existe una secuencia de morfismos $[\alpha_i]$, $0 \leq i \leq n$, cuyos términos α_i describen las denominadas *reescrituras secuenciales en un paso*

$$[t] \xrightarrow{\alpha_0} [t_1] \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_{n-1}} [t_n] \xrightarrow{\alpha_n} [t']$$

tales que $[\alpha] = [\alpha_0; \ldots; \alpha_n]$.

Este resultado afirma en particular que cualquier transición en el modelo inicial asociado con un programa en la lógica de reescritura se puede escribir como una secuencia de pre-acciones en nuestro sentido.

4.2.4. Modelos

Los modelos para una *signatura de verificación* $\langle \Sigma^+, E^+, \mathit{State}, \mathit{At}, L\rangle$ sobre una signatura de reescritura $\langle \Sigma, E\rangle$ se obtienen a partir de programas (teorías de reescritura) $\langle \Sigma, E, L, R\rangle$, es decir, a partir de asignaciones de reglas de reescritura R a etiquetas L[3]. Cada una de estas teorías define una estructura de Kripke de la siguiente forma:

- el conjunto de estados (mundos posibles) es el conjunto cociente $T_{\Sigma,E,\mathit{State}}$ de clases de equivalencia de términos básicos de tipo *State* módulo las ecuaciones E;
- la familia de transiciones entre estados viene dada por las *acciones* básicas, esto es, clases de equivalencia de términos básicos de pruebas apropiados que reescriben estados.

Como se ha explicado anteriormente, este sistema de transiciones se obtiene a partir del modelo inicial $\mathcal{T}_{\mathcal{R}}$ de una teoría de reescritura $\mathcal{R}$ olvidando parte de la estructura de este modelo, como por ejemplo la composición secuencial que corresponde a la transitividad.

Para obtener los modelos de una signatura de verificación, todavía debemos proporcionar una interpretación para las observaciones: para cada tipo s, una *interpretación de las observaciones* I lleva observaciones de tipo s en funciones de tipo $T_{\Sigma,E,\mathit{State}} \to T_{\Sigma^+,E^+,s}$. Obsérvese que, como la signatura de reescritura está protegida por la extensión a la signatura de verificación, $T_{\Sigma,E,\mathit{State}}$ es el *mismo* conjunto que $T_{\Sigma^+,E^+,\mathit{State}}$. Por otra parte, no podemos escribir $T_{\Sigma,E,s}$ como el codominio de la función ya que el tipo s puede no ser un tipo del programa. Por ejemplo, `Nat` es el tipo de todas las observaciones que hemos definido en el ejemplo de la máquina expendedora en la Sección 4.2.2, pero `Nat` no es un tipo del programa de la máquina expendedora `VENDING-MACHINE`.

Cada interpretación de las observaciones I se puede ver como una extensión de T_{Σ^+,E^+} a una Σ_f^+-algebra. Por lo tanto, las interpretaciones de observaciones se pueden definir axiomáticamente a través de conjuntos de ecuaciones E_f^+ sobre la signatura extendida Σ_f^+, como ya hemos mencionado en la Sección 4.2.1. De hecho, podemos querer reducir el conjunto de interpretaciones de observaciones que queremos admitir para la verificación a aquellas que satisfagan algunas propiedades.

Por lo tanto, debemos tomar los *modelos* como estructuras de Kripke asociadas a las teorías de reescritura o programas sobre una signatura extendida $\langle \Sigma_f^+, E_f^+\rangle$. Llamamos a estos programas *programas observados*, ya que incluyen junto con el programa original toda la información sobre las observaciones.

En nuestro ejemplo de la máquina expendedora, la teoría de reescritura asociada al módulo de sistema `VENDING-MACHINE-OBSERVED` define el programa observado además de proporcionar una interpretación única para las observaciones. Esto va a ocurrir en todos

[3]Obsérvese que las etiquetas coinciden en ambos sitios; la única diferencia es que hemos asociado una aridad con cada etiqueta en la verificación debido a que las utilizamos como operaciones para construir las acciones. También, en general, en un programa la misma etiqueta se puede utilizar para diferentes reglas pero asumiremos como en [Mes92, Sección 3.1] que las etiquetas se han desambiguado según ha sido necesario.

los ejemplos que se proponen, pero en general es posible dar en el programa observado solamente algunas restricciones para las observaciones que no determinen las posibles interpretaciones de forma única; por tanto, simplemente asumiremos que la interpretación de las observaciones I satisface las ecuaciones en E_f^+.

4.2.5. El lenguaje modal

El lenguaje de términos asociado con una signatura de verificación $\langle \Sigma^+, E^+, \mathit{State}, \mathit{At}, L\rangle$ es el álgebra de términos $T_{\Sigma_c^+}(X)$, es decir, la extensión de Σ^+ con atributos como constantes (ya que en el lenguaje modal el estado está implícito), donde X contiene un conjunto infinito de variables para cada tipo en Σ_c^+.

El lenguaje modal asociado con una signatura de verificación viene dado por la siguiente gramática:

$$\varphi ::= \mathit{true} \mid t_1 = t_2 \mid \neg\varphi \mid \varphi_1 \supset \varphi_2 \mid [\alpha]\varphi \mid f_{\overline{d}}[\varphi_1, \ldots, \varphi_n]$$

donde $t_1, t_2 \in T_{\Sigma_c^+}(X)$ son dos términos del mismo tipo, α es una acción, $f : s_1 \ldots s_m \to \mathit{State} \in \Sigma$, $\overline{d}$ es una secuencia de datos correspondientes a los argumentos de f que no son de tipo *State*, y los φ_i están en correspondencia uno a uno con los argumentos de f que son de tipo *State*.

Aplicamos el mismo convenio de notación a las acciones y a los términos que representan estados. Dados $f : s_1 \ldots s_m \to \mathit{State} \in \Sigma$, una secuencia $\overline{d}$ de datos correspondientes a los argumentos de f que no son de tipo *State*, y las acciones α_i en correspondencia una a una con los argumentos de f que son de tipo *State*, denotamos por $f_{\overline{d}}(\alpha_1, \ldots, \alpha_n)$ la acción que resulta de la aplicación de f a la mezcla de $\overline{d}$ y $\overline{\alpha}$ dada por $s_1, \ldots, s_m$. De forma análoga, $f_{\overline{d}}(t_1, \ldots, t_n)$, donde cada t_i es un término de tipo *State* y ninguno de los d_i es de tipo *State*, denota el término obtenido al aplicar f a la mezcla de $\overline{d}$ y $\overline{t}$ de acuerdo con la secuencia de tipos $s_1, \ldots, s_m$.

Por ejemplo, si f es una función de tipo $\mathit{State}\ s_1\ \mathit{State}\ s_2 \to \mathit{State}$, d_i es de tipo s_i, t_i es de tipo *State*, α_i son acciones y φ_i son fórmulas ($i = 1, 2$), entonces tenemos que $f_{d_1 d_2}(t_1, t_2)$ coincide con el término $f(t_1, d_1, t_2, d_2)$ de tipo *State*, $f_{d_1 d_2}(\alpha_1, \alpha_2)$ es una acción, y $f_{d_1 d_2}[\varphi_1, \varphi_2]$ es una fórmula.

Dada una acción α, $[\alpha]$ es la modalidad usual que captura las transiciones de estado realizadas por la acción. Además, el lenguaje que se propone presenta una nueva modalidad $f_{\overline{d}}[_]$ asociada a la construcción de estados. Como se formaliza a continuación, esta modalidad *espacial* nos permite razonar sobre la *estructura* de los estados, esto es, sobre el hecho de que el estado actual del sistema se puede descomponer en componentes que satisfacen ciertas propiedades (obsérvese que esta descomposición puede no ser única debido a las ecuaciones en la signatura de reescritura).

Usaremos las conectivas proposicionales usuales para la conjunción ($\wedge$), disyunción ($\vee$) y equivalencia lógica ($\equiv$).

En los ejemplos, permitiremos también cuantificación sobre variables *rígidas*, esto es, variables que permanecen constantes en el modelo y sirven para conectar el valor de diferentes variables. Esta clase de cuantificación no incrementa el poder expresivo de la lógica y se utiliza únicamente para facilitar la lectura de las fórmulas.

4.2.6. La relación de satisfacción

La relación de satisfacción se define para un estado, una interpretación de los atributos y una sustitución de Σ^+-términos básicos por variables. Para simplificar la notación omitiremos la referencia al conjunto E^+ de ecuaciones sobre el que se forman los estados como clases de equivalencia de términos. El lenguaje de términos $T_{\Sigma_c^+}(X)$ se interpreta en T_{Σ^+,E^+} de la siguiente forma, donde $[\![t']\!]^{I,\sigma}[t]$ es el valor (un elemento de T_{Σ^+,E^+}) de t' en el estado $[t]$, para la interpretación de los atributos I y la sustitución básica σ:

- $[\![x]\!]^{I,\sigma}[t] = \sigma(x)$,
- $[\![obs]\!]^{I,\sigma}[t] = I(obs)([t])$,
- $[\![f(t_1,\ldots,t_m)]\!]^{I,\sigma}[t] = f([\![t_1]\!]^{I,\sigma}[t],\ldots,[\![t_m]\!]^{I,\sigma}[t])$,
  para cada operación $f : s_1 \ldots s_m \to s$ en Σ^+ .

Las acciones están sujetas también a interpretación, ya que pueden contener variables. Denotamos por $[\![\alpha]\!]^{I,\sigma}$ la transición de estado dada por la acción básica obtenida al aplicar la sustitución σ a la acción α.

La satisfacción de una fórmula en un estado dado $[t]$ se define entonces como sigue:

- $[t], I, \sigma \models true$,
- $[t], I, \sigma \models t_1 = t_2$ si y solo si $[\![t_1]\!]^{I,\sigma}[t] = [\![t_2]\!]^{I,\sigma}[t]$,
- $[t], I, \sigma \models \neg\varphi$ si y solo si no es el caso que $[t], I, \sigma \models \varphi$,
- $[t], I, \sigma \models \varphi_1 \supset \varphi_2$ si y solo si $[t], I, \sigma \models \varphi_1$ implica $[t], I, \sigma \models \varphi_2$,
- $[t], I, \sigma \models [\alpha]\varphi$ si y solo si $[\![\alpha]\!]^{I,\sigma} : [t] \to [t']$ implica $[t'], I, \sigma \models \varphi$,
- $[t], I, \sigma \models f_{\overline{d}}[\varphi_1,\ldots,\varphi_n]$ si y solo si para cada $f_{\overline{w}}(t_1,\ldots,t_n) \in [t]$, donde $\overline{w} = [\![\overline{d}]\!]^{I,\sigma}[t]$, existe algún $i \in \{1,\ldots,n\}$ tal que $[t_i], I, \sigma \models \varphi_i$.

Como es usual, la modalidad de acción captura propiedades que se cumplen después de que se realice la transición. Obsérvese que, para una interpretación de atributos y una sustitución dadas, las acciones son deterministas pero parciales, es decir, no se aplican a todos los estados. De hecho, una vez instanciada una acción solo se puede aplicar a un estado. Esto es debido al hecho explicado en la Sección 4.2.3 de que una acción lleva consigo toda la información del contexto en el que se aplican las reglas de reescritura.

La modalidad de acción tiene una modalidad dual con la siguiente interpretación:

- $[t], I, \sigma \models \langle\alpha\rangle\varphi$ si y solo si $[\![\alpha]\!]^{I,\sigma} : [t] \to [t']$ y $[t'], I, \sigma \models \varphi$.

Obsérvese que esta operación de carácter *existencial* requiere que la acción denote una transición existente desde el estado actual. Por otra parte, la modalidad de carácter *universal* $[\alpha]$ no impone el requisito de que la transición exista realmente. Por lo tanto,

$[\alpha]\varphi$ se cumple de forma trivial si, para una sustitución dada, α no puede reescribir el estado actual.

El operador espacial nos permite hablar sobre la estructura de los estados. Su dual es quizá más intuitivo:

- $[t], I, \sigma \models f_{\overline{d}}\langle\varphi_1, \ldots, \varphi_n\rangle$ si y solo si existe un término $f_{\overline{w}}(t_1, \ldots, t_n) \in [t]$ con $\overline{w} = [\![\overline{d}]\!]^{I,\sigma}[t]$ tal que $[t_i], I, \sigma \models \varphi_i$ para todo $i = 1, \ldots, n$.

Esta fórmula nos permite decir que el estado actual es descomponible de acuerdo con la estructura f en componentes que satisfacen las fórmulas φ_i. La modalidad $f_{\overline{d}}[\varphi_1, \ldots, \varphi_n]$ introducida anteriormente utiliza en cambio la condición *si el estado es descomponible en la forma especificada, entonces para cada posible descomposición uno de los subestados cumple una propiedad dada.* Desde este punto de vista de descomposición, obsérvese de nuevo el carácter existencial de la construcción $f_{\overline{d}}\langle\varphi_1, \ldots, \varphi_n\rangle$, en contra del carácter universal de la construcción $f_{\overline{d}}[\varphi_1, \ldots, \varphi_n]$.

Finalmente, decimos que una fórmula es válida en un modelo si y solo si se satisface en todos los estados para cualquier interpretación de los atributos y para cualquier sustitución básica de las variables. Escribiremos $\Phi \vdash \varphi$ para denotar que si Φ es un conjunto de fórmulas válidas, entonces φ es también una fórmula válida.

Algunos ejemplos de satisfacción de fórmulas para el programa `VENDING-MACHINE` visto en la Sección 4.2.2 son:

- El estado `$qq` satisface la propiedad $__\langle \#\$ = 0, \#\$ = 1\rangle$, que expresa que el estado se puede descomponer en dos subestados, el primero de los cuales no contiene ningún dólar y el segundo contiene un dólar. Se satisfacen también las propiedades: $__\langle \#\$ = 1, \#q = 2\rangle$ que expresa que el estado se puede descomponer en dos subestados el primero de los cuales tiene un dólar y el segundo dos cuartos, y $__\langle \#\$ = 1 \wedge \#q = 1, \#q = 1\rangle$ que expresa que el estado se puede descomponer en dos subestados el primero de los cuales tiene un dólar y un cuarto, y el segundo un cuarto .

- En el estado `qqqq`, se satisfacen las fórmulas $[$`change`$](\#q = 0)$, que expresa que si se puede realizar la acción `change` el estado al que se llega no tendrá ningún cuarto, y $\langle$`change`$\rangle(\#q = 0)$ que expresa que la acción `change` puede tener lugar y el estado al que se llega no tendrá ningún cuarto. Pero no se satisface $\langle$`buy-c`$\rangle(\#q = 0)$ ya que la acción `buy-c` no puede aplicarse a este estado.

- El estado `$q` *no* satisface la fórmula $\langle$`buy-c`$\rangle$*true* que expresa que la acción `buy-c` puede tener lugar, ya que para que esta acción pueda tener lugar el estado debe estar formado únicamente por un dólar.

4.3. Un procedimiento de decisión para la lógica VLRL

En esta sección se muestra la existencia de un algoritmo de decisión para las fórmulas VLRL. El algoritmo parte de la existencia para un modelo VLRL de un *modelo filtrado*

con un número finito de estados. El modelo filtrado se obtiene para una fórmula dada φ. En nuestro caso, la fórmula φ utilizada es la fórmula que queremos comprobar si es satisfactible. El modelo filtrado cumple que cualquier fórmula de la clausura de φ es satisfactible en él si y solo si es satisfactible en el modelo VLRL [HKT00, Gol92].

El modelo filtrado proporciona de forma inmediata un sencillo procedimiento de decisión. Para determinar si una fórmula φ es satisfactible comprobamos si φ se satisface en algún estado del modelo filtrado. Dado que la fórmula φ puede ser compleja, buscaremos un procedimiento de decisión que una vez obtenido el modelo filtrado sea fácilmente automatizable.

El algoritmo que proponemos a continuación se basa en etiquetar los estados del modelo filtrado con el conjunto de fórmulas que se verifican en ese estado. El procedimiento ha sido adaptado del algoritmo utilizado por Fisher en [Fis92].

4.3.1. Definición del modelo filtrado

Para obtener el modelo filtrado para una fórmula dada φ, identificamos los estados del modelo VLRL que no son distinguibles por ninguna fórmula de la clausura de φ.

Formalmente, sea φ una *VLRL*-formula. La *clausura* de φ, $CL(\varphi)$, es el menor conjunto de fórmulas tal que:

- $\varphi \in CL(\varphi)$
- $\mathit{true} \in CL(\varphi)$.
- $\neg\psi \in CL(\varphi)$ si y solo si $\psi \in CL(\varphi)$ (con $\varphi \equiv \neg\neg\varphi$ y $\mathit{true} \equiv \neg \mathit{false}$).
- si $f_{\overline{d}}[\psi_1, \ldots, \psi_n] \in CL(\varphi)$ entonces $\psi_i \in CL(\varphi)$ para todo $i = 1, \ldots, n$.
- si $\psi_1 \supset \psi_2 \in CL(\varphi)$ entonces $\psi_1, \psi_2 \in CL(\varphi)$.
- si $[\alpha]\psi \in CL(\varphi)$ entonces $\psi \in CL(\varphi)$.

El cardinal del conjunto $CL(\varphi)$ es finito ya que $CL(\varphi)$ contiene todas las subfórmulas de φ y sus negaciones.

Definimos el conjunto de acciones de una fórmula φ, $AT(\varphi)$, como el conjunto de acciones que aparecen en alguna modalidad de acción de φ, esto es, si $[\alpha]\psi \in CL(\varphi)$ entonces $\alpha \in AT(\varphi)$.

Dado un *VLRL*-modelo[4], $\mathcal{M}$, una sustitución básica de variables, σ, y una *VLRL*-formula, φ, obtenemos un modelo finito para φ, identificando los estados que no son distinguibles por ninguna fórmula de $CL(\varphi)$.

Sea $\equiv_{CL(\varphi)}$ la relación de equivalencia en $T_{\Sigma,E,State}$ definida por

$$[t]_E \equiv_{CL(\varphi)} [t']_E \text{ si para todo } \psi \in CL(\varphi), [t]_E, I, \sigma \models \psi \text{ si y solo si } [t']_E, I, \sigma \models \psi.$$

Utilizamos $[t]_{CL(\varphi)}$ para denotar la clase de equivalencia $\{[t']_E \mid [t]_E \equiv_{CL(\varphi)} [t']_E\}$.

[4]Recuérdese que el modelo incluye una interpretación de observaciones I.

Definimos el *modelo filtrado* de $\mathcal{M}$ por φ para una sustitución básica dada σ como

$$\mathcal{M}^{CL(\varphi)} = (S^{CL(\varphi)}, \{\alpha \in AT(\varphi)\}, I^{CL(\varphi)}, \sigma)$$

donde

- $S^{CL(\varphi)} = \{[t]_{CL(\varphi)} \mid [t]_E \in T_{\Sigma,E,State}\}$.
- $[\alpha] : [t]_{CL(\varphi)} \to [t']_{CL(\varphi)}$ si y solo si existe un $[t]_E \in [t]_{CL(\varphi)}$ y un $[t']_E \in [t']_{CL(\varphi)}$ tales que $[\![\alpha]\!] : [t]_E \to [t']_E$.
- La interpretación de las observaciones $I^{CL(\varphi)}$ proyecta observaciones de tipo s en funciones de tipo
$$T_{\Sigma,E,State}/\!\equiv_{CL(\varphi)} \to T_{\Sigma^+,E^+,s}.$$
Para cada clase de equivalencia $[t]_{CL(\varphi)}$ de $T_{\Sigma,E,State}/\!\equiv_{CL(\varphi)}$ se selecciona un representante $[t']_E$; entonces, dada una observación a, su interpretación en un estado $[t]_{CL(\varphi)}$ es $a([t]_{CL(\varphi)}) = v$ si y solo si la interpretación de la observación en $\mathcal{M}$ es $a([t']_E) = v$ para $[t']_E \in [t]_{CL(\varphi)}$ el representante seleccionado para la clase $[t]_{CL(\varphi)}$.

  Obsérvese que, aunque la interpretación de las observaciones depende del representante seleccionado (es posible que $a([q]_E) \neq a([q']_E)$ para $[q]_E, [q']_E \in [t]_{CL(\varphi)}$), el valor de verdad de la fórmula $\psi \in CL(\varphi)$ no depende del representante seleccionado $[t']_E$, ya que si $[t']_E, [t'']_E \in [t]_{CL(\varphi)}$ tenemos que para todo $\psi \in CL(\varphi)$ $[t']_E, I, \sigma \models \psi$ si y solo si $[t'']_E, I, \sigma \models \psi$.

La satisfacción de una fórmula $\psi \in CL(\varphi)$ se define como

$$[t]_{CL(\varphi)} \models \psi \text{ si y solo si } [t]_E, I, \sigma \models \psi \text{ para todo } [t]_E \in [t]_{CL(\varphi)} \ .$$

Obsérvese que debido a la definición de la clase de equivalencia $[t]_{CL(\varphi)}$, $[t]_E, I, \sigma \models \psi$ para todo $[t]_E \in [t]_{CL(\varphi)}$ si $[t]_E, I, \sigma \models \psi$ para algún $[t]_E \in [t]_{CL(\varphi)}$.

La satisfacción de las combinaciones proposicionales de fórmulas de $CL(\varphi)$ se define de la forma usual

$$[t]_{CL(\varphi)} \models \psi \supset \psi' \text{ si y solo si } [t]_{CL(\varphi)} \models \psi \text{ implica } [t]_{CL(\varphi)} \models \psi' \ ,$$
$$[t]_{CL(\varphi)} \models \neg\psi \text{ si y solo si no es el caso } [t]_{CL(\varphi)} \models \psi \ .$$

Utilizaremos las conectivas derivadas $\wedge$ y $\vee$.

Lema 1 *El número de estados de $\mathcal{M}^{CL(\varphi)}$ es menor o igual que 2^n, donde n denota el número de fórmulas en $CL(\varphi)$.*

Demostración. $S^{CL(\varphi)}$ no puede tener más conjuntos que el número de asignaciones posibles de valores de verdad a las fórmulas en $CL(\varphi)$.

Lema 2 *$\psi \in CL(\varphi)$ es satisfactible en $\mathcal{M}$ para una sustitución básica dada σ si y solo si es satisfactible en $\mathcal{M}^{CL(\varphi)}$.*

Demostración. Supóngase que ψ es satisfactible en $\mathcal{M}$; entonces existe $[t]_E$ tal que $[t]_E, I, \sigma \models \psi$. Considérese $[t]_{CL(\varphi)}$, entonces por la definición de la relación de satisfacción $[t]_{CL(\varphi)} \models \psi$.

Recíprocamente, supóngase que ψ es satisfactible en $\mathcal{M}^{CL(\varphi)}$; entonces existe $[t]_{CL(\varphi)}$ tal que $[t]_{CL(\varphi)} \models \psi$. Por la definición de la relación de satisfacción $[t']_E, I, \sigma \models \psi$ para todo $[t']_E \in [t]_{CL(\varphi)}$. Por lo tanto ψ es satisfactible en $\mathcal{M}$ para la sustitución σ.

4.3.2. Procedimiento de decisión

El algoritmo que se define en esta sección comprueba si una *VLRL*-formula φ es satisfactible en $\mathcal{M}^{CL(\varphi)}$.

Primero definimos dos relaciones entre los estados de $\mathcal{M}^{CL(\varphi)}$.

- $[t']_{CL(\varphi)}$ es un α-sucesor de $[t]_{CL(\varphi)}$ si existe una acción $[\alpha] : [t]_{CL(\varphi)} \to [t']_{CL(\varphi)}$ en $\mathcal{M}^{CL(\varphi)}$.
- $([t_1]_{CL(\varphi)}, \ldots, [t_n]_{CL(\varphi)})$ es un sucesor-*espacial* de $[t]_{CL(\varphi)}$ si existe $[f_{\overline{d}}(t_1, \ldots, t_n)]_E \in [t]_{CL(\varphi)}$ con $[t_i]_E \in [t_i]_{CL(\varphi)}$ para todo $i = 1, \ldots, n$.

La idea es asociar a cada estado del modelo filtrado un conjunto de etiquetas que denominamos $label([t]_{CL(\varphi)})$, con las fórmulas de la clausura de la fórmula φ, que queremos comprobar si es satisfactible, que cumplen una serie de propiedades. Como se demostrará en el Teorema 1 estas propiedades hacen que cada estado $[t]_{CL(\varphi)}$ satisfaga las fórmulas de $label([t]_{CL(\varphi)})$.

El algoritmo realiza un máximo de n pasadas, siendo n el cardinal del conjunto $CL(\varphi)$. En cada pasada, cada estado se reetiqueta con nuevas fórmulas de $CL(\varphi)$ que cumplen las propiedades requeridas.

Algoritmo

1. Añadimos las fórmulas básicas a los estados que las satisfacen.

$$\bullet \text{ si } t_1 = t_2 \in CL(\varphi) \text{ y } [t]_{CL(\varphi)} \models t_1 = t_2, \text{ entonces añadir } t_1 = t_2 \text{ a } label([t]_{CL(\varphi)}), \tag{4.1}$$

$$\bullet \text{ si } \neg(t_1 = t_2) \in CL(\varphi) \text{ y } [t]_{CL(\varphi)} \not\models t_1 = t_2, \text{ entonces añadir } \neg(t_1 = t_2) \text{ a } label([t]_{CL(\varphi)}). \tag{4.2}$$

2. Considérense las fórmulas $\phi_1 \supset \phi_2 \in CL(\varphi)$ y $\neg(\phi_1 \supset \phi_2) \in CL(\varphi)$.

$$\bullet \text{ si } \phi_1 \supset \phi_2 \in CL(\varphi) \text{ y } \neg\phi_1 \in label([t]_{CL(\varphi)}), \text{ entonces añadir } \phi_1 \supset \phi_2 \text{ a } label([t]_{CL(\varphi)}), \tag{4.3}$$

$$\bullet \text{ si } \phi_1 \supset \phi_2 \in CL(\varphi) \text{ y } \phi_2 \in label([t]_{CL(\varphi)}), \text{ entonces añadir } \phi_1 \supset \phi_2 \text{ a } label([t]_{CL(\varphi)}), \tag{4.4}$$

$$\bullet \text{ si } \neg(\phi_1 \supset \phi_2) \in CL(\varphi) \text{ y } \phi_1 \in label([t]_{CL(\varphi)}) \text{ y } \neg\phi_2 \in label([t]_{CL(\varphi)}), \text{ entonces añadir } \neg(\phi_1 \supset \phi_2) \text{ a } label([t]_{CL(\varphi)}). \tag{4.5}$$

3. Considérense las fórmulas con modalidad de acción $[\alpha]\phi$ y $\langle\alpha\rangle\phi$.

   - si $[\alpha]\phi \in CL(\varphi)$ y $[t]_{CL(\varphi)}$ no tiene α-sucesor, entonces añadir $[\alpha]\phi$ a $label([t]_{CL(\varphi)})$, (4.6)
   - si $[\alpha]\phi \in CL(\varphi)$ y $\phi \in label([t']_{CL(\varphi)})$ para $[t']_{CL(\varphi)}$ un α-sucesor de $[t]_{CL(\varphi)}$, entonces añadir $[\alpha]\phi$ a $label([t]_{CL(\varphi)})$, (4.7)
   - si $\langle\alpha\rangle\phi \in CL(\varphi)$ y $[t]_{CL(\varphi)}$ tiene un α-sucesor $[t']_{CL(\varphi)}$ y $\phi \in label([t']_{CL(\varphi)})$, entonces añadir $\langle\alpha\rangle\phi$ a $label([t]_{CL(\varphi)})$. (4.8)

4. Considérense las fórmulas con modalidad espacial $f_{\overline{d}}[\phi_1, \ldots, \phi_n]$ y $f_{\overline{d}}\langle\phi_1, \ldots, \phi_n\rangle$.

   - si $f_{\overline{d}}[\phi_1, \ldots, \phi_n] \in CL(\varphi)$ y para cada sucesor-*espacial* de $[t]_{CL(\varphi)}$ existe un $i \in \{1, \ldots, n\}$ tal que $\phi_i \in label([t_i]_{CL(\varphi)})$, entonces añadir $f_{\overline{d}}[\phi_1, \ldots, \phi_n]$ a $label([t]_{CL(\varphi)})$, (4.9)
   - si $f_{\overline{d}}\langle\phi_1, \ldots, \phi_n\rangle \in CL(\varphi)$ y existe un sucesor-*espacial* de $[t]_{CL(\varphi)}$ tal que $\phi_i \in label([t_i]_{CL(\varphi)})$ para todo $i \in \{1, \ldots, n\}$, entonces añadir $f_{\overline{d}}\langle\phi_1, \ldots, \phi_n\rangle$ a $label([t]_{CL(\varphi)})$. (4.10)

5. Repetir los pasos 2, 3 y 4 hasta que no se añadan nuevas fórmulas a ningún conjunto $label([t]_{CL(\varphi)})$.

4.3.3. Corrección y complejidad del algoritmo

Lema 3 *El algoritmo termina y en el peor caso la complejidad del algoritmo es del orden de $\mathcal{O}(n^2 * t)$ donde n es el cardinal del conjunto $CL(\varphi)$ y t es el número de estados en $S^{CL(\varphi)}$.*

Demostración. Cada paso del algoritmo termina ya que el número de estados es finito y el número de fórmulas en $CL(\varphi)$ es finito. El proceso de repetición termina, ya que el número de fórmulas es finito y por lo tanto en algún momento ya no se podrán añadir más fórmulas a los conjuntos de etiquetas.

Los pasos 2, 3, 4 se repiten como mucho n veces, ya que en cada pasada si se añade una fórmula a la etiqueta de un estado esta fórmula ya no se volverá a añadir a la etiqueta de este estado. De esta forma en cada pasada disminuye el número de formulas que hay que tratar para cada estado. Además si en una pasada no se añade ninguna formula a la etiqueta de un estado, ya no se añadiran más fórmulas a la etiqueta de este estado.

Cada vez que el algoritmo realiza los pasos 2, 3, 4, cada fórmula no básica de $CL(\varphi)$ se comprueba en todos los conjuntos $label([t]_{CL(\varphi)})$.

Teorema 1 *Sea $\phi \in CL(\varphi)$ y $[t]_{CL(\varphi)} \in S^{CL(\varphi)}$; entonces, $\phi \in label([t]_{CL(\varphi)})$ si y solo si $[t]_{CL(\varphi)} \models \phi$.*

Demostración. La demostración se realiza por inducción sobre la estructura de las fórmulas. Distinguimos los mismos casos que en el algoritmo de la Sección 4.3.2.

Caso $\phi \equiv t_1 = t_2$.

$$
\begin{aligned}
& t_1 = t_2 \in \mathit{label}([t]_{CL(\varphi)}) \\
\Leftrightarrow\ & t_1 = t_2 \text{ satisface la condición (4.1)} \\
\Leftrightarrow\ & [t]_{CL(\varphi)} \models t_1 = t_2.
\end{aligned}
$$

Case $\phi \equiv \neg(t_1 = t_2)$.

$$
\begin{aligned}
& \neg(t_1 = t_2) \in \mathit{label}([t]_{CL(\varphi)}) \\
\Leftrightarrow\ & \neg(t_1 = t_2) \text{ satisface la condición (4.2)} \\
\Leftrightarrow\ & [t]_{CL(\varphi)} \not\models (t_1 = t_2) \\
\Leftrightarrow\ & [t]_{CL(\varphi)} \models \neg(t_1 = t_2).
\end{aligned}
$$

Caso $\phi \equiv \phi_1 \supset \phi_2$.

$$
\begin{aligned}
& \phi_1 \supset \phi_2 \in \mathit{label}([t]_{CL(\varphi)}) \\
\Leftrightarrow\ & \phi_1 \supset \phi_2 \text{ satisface la condición (4.3) o la condición (4.4)} \\
\Leftrightarrow\ & \neg\phi_1 \in \mathit{label}([t]_{CL(\varphi)}) \text{ o } \phi_2 \in \mathit{label}([t]_{CL(\varphi)}) \\
\Leftrightarrow\ & [t]_{CL(\varphi)} \models \neg\phi_1 \text{ o } [t]_{CL(\varphi)} \models \phi_2, \text{ por hipótesis de inducción,} \\
\Leftrightarrow\ & [t]_{CL(\varphi)} \models \phi_1 \supset \phi_2, \text{ por lógica proposicional.}
\end{aligned}
$$

Caso $\phi \equiv \neg(\phi_1 \supset \phi_2)$.

$$
\begin{aligned}
& \neg(\phi_1 \supset \phi_2) \in \mathit{label}([t]_{CL(\varphi)}) \\
\Leftrightarrow\ & \neg(\phi_1 \supset \phi_2) \text{ satisface la condición (4.5)} \\
\Leftrightarrow\ & \phi_1 \in \mathit{label}([t]_{CL(\varphi)}) \text{ y } \neg\phi_2 \in \mathit{label}([t]_{CL(\varphi)}) \\
\Leftrightarrow\ & [t]_{CL(\varphi)} \models \phi_1 \text{ y } [t]_{CL(\varphi)} \models \neg\phi_2, \text{ por hipótesis de inducción,} \\
\Leftrightarrow\ & [t]_{CL(\varphi)} \models \neg(\phi_1 \supset \phi_2), \text{ por lógica proposicional.}
\end{aligned}
$$

Caso $\phi \equiv [\alpha]\phi'$.

$[\alpha]\phi' \in \mathit{label}([t]_{CL(\varphi)})$

$\Leftrightarrow$ $[\alpha]\phi'$ satisface la condición (4.6) o la condición (4.7)

$\Leftrightarrow$ $[t]_{CL(\varphi)}$ no tiene α-sucesor o $\phi' \in \mathit{label}([t']_{CL(\varphi)})$ para $[t']_{CL(\varphi)}$ un α-sucesor de $[t]_{CL(\varphi)}$

$\Leftrightarrow$ no existe $[\alpha] : [t]_{CL(\varphi)} \to [t']_{CL(\varphi)}$ o existe $[\alpha] : [t]_{CL(\varphi)} \to [t']_{CL(\varphi)}$ y $\phi' \in \mathit{label}([t']_{CL(\varphi)})$, por la definición de la relación α-sucesor,

$\Leftrightarrow$ no existe $[\alpha] : [t]_{CL(\varphi)} \to [t']_{CL(\varphi)}$ o existe $[\alpha] : [t]_{CL(\varphi)} \to [t']_{CL(\varphi)}$ y $[t']_{CL(\varphi)} \models \phi'$, por hipótesis de inducción,

$\Leftrightarrow$ $[t]_{CL(\varphi)} \models [\alpha]\phi'$, por la definición de la relación de satisfacción.

Case $\phi \equiv \langle\alpha\rangle\phi'$.

$\langle\alpha\rangle\phi' \in \mathit{label}([t]_{CL(\varphi)})$

$\Leftrightarrow$ $\langle\alpha\rangle\phi'$ satisface la condición (4.8)

$\Leftrightarrow$ $[t]_{CL(\varphi)}$ tiene un α-sucesor $[t']_{CL(\varphi)}$ y $\phi' \in \mathit{label}([t']_{CL(\varphi)})$

$\Leftrightarrow$ existe $[\alpha] : [t]_{CL(\varphi)} \to [t']_{CL(\varphi)}$ y $[t']_{CL(\varphi)} \models \phi'$, por hipótesis de inducción,

$\Leftrightarrow$ $[t]_{CL(\varphi)} \models \langle\alpha\rangle\phi'$, por la definición de la relación de satisfacción.

Caso $\phi \equiv f_{\overline{d}}[\phi_1, \ldots, \phi_n]$.

$f_{\overline{d}}[\phi_1, \ldots, \phi_n] \in \mathit{label}([t]_{CL(\varphi)})$

$\Leftrightarrow$ $f_{\overline{d}}[\phi_1, \ldots, \phi_n]$ satisface la condición (4.9)

$\Leftrightarrow$ para cada sucesor-*espacial* de $[t]_{CL(\varphi)}$ existe algún $i \in \{1, \ldots, n\}$ tal que $\phi_i \in \mathit{label}([t_i]_{CL(\varphi)})$

$\Leftrightarrow$ para cada $[f_{\overline{d}}(t_1, \ldots, t_n)]_E \in [t]_{CL(\varphi)}$ existe algún $i \in \{1, \ldots, n\}$ tal que $\phi_i \in \mathit{label}([t_i]_{CL(\varphi)})$, por la definición de la relación sucesor-*espacial*,

$\Leftrightarrow$ para cada $[f_{\overline{d}}(t_1, \ldots, t_n)]_E \in [t]_{CL(\varphi)}$ existe algún $i \in \{1, \ldots, n\}$ tal que $[t_i]_{CL(\varphi)} \models \phi_i$, por hipótesis de inducción,

$\Leftrightarrow$ $[t]_{CL(\varphi)} \models f_{\overline{d}}[\phi_1, \ldots, \phi_n]$, por la definición de la relación de satisfacción.

Case $\phi \equiv f_{\overline{d}}\langle\phi_1, \ldots, \phi_n\rangle$.

$f_{\overline{d}}\langle\phi_1, \ldots, \phi_n\rangle \in \mathit{label}([t]_{CL(\varphi)})$

$\Leftrightarrow$ $f_{\overline{d}}\langle\phi_1, \ldots, \phi_n\rangle$ satisface la condición (4.10)

$\Leftrightarrow$ existe un sucesor-*espacial* de $[t]_{CL(\varphi)}$ tal que $\phi_i \in \mathit{label}([t_i]_{CL(\varphi)})$ para todo $i \in \{1, \ldots, n\}$

$\Leftrightarrow$ existe $[f_{\overline{d}}(t_1, \ldots, t_n)]_E \in [t]_{CL(\varphi)}$ tal que $\phi_i \in \mathit{label}([t_i]_{CL(\varphi)})$ para todo $i \in \{1, \ldots, n\}$, por la definición de la relación sucesor-*espacial*,

$\Leftrightarrow$ existe $[f_{\overline{d}}(t_1, \ldots, t_n)]_E \in [t]_{CL(\varphi)}$ tal que $[t_i]_{CL(\varphi)} \models \phi_i$ para todo $i \in \{1, \ldots, n\}$, por hipótesis de inducción,

$\Leftrightarrow$ $[t]_{CL(\varphi)} \models f_{\overline{d}}\langle\phi_1, \ldots, \phi_n\rangle$, por la definición de la relación de satisfacción.

4.3.4. Ejemplo de aplicación del algoritmo de decisión

Considérese la siguiente especificación de cadenas de caracteres,

```
sorts Alpha String .
subsort Alpha < String .
ops a b : -> Alpha .
op __ : String String -> String [assoc] .
rl [ch1] : a => b .
rl [ch2] : b => a .
```

con una observación *first* que devuelve el primer carácter de una cadena.

Queremos comprobar si se satisface la fórmula

$$\varphi \equiv __\langle \mathit{first} = \mathtt{a}, \mathit{first} = \mathtt{b}\rangle \supset \langle __(\mathtt{ch1}, \mathtt{ch2})\rangle __\langle \mathit{first} = \mathtt{b}, \mathit{first} = \mathtt{a}\rangle.$$

El modelo filtrado

La clausura de φ es el conjunto:

$$\begin{aligned} CL(\varphi) = \{ & \mathit{true}, \mathit{false}, \mathit{first} = \mathtt{a}, \neg(\mathit{first} = \mathtt{a}), \mathit{first} = \mathtt{b}, \neg(\mathit{first} = \mathtt{b}), __\langle \mathit{first} = \mathtt{a}, \mathit{first} = \mathtt{b}\rangle, \\ & \neg(__\langle \mathit{first} = \mathtt{a}, \mathit{first} = \mathtt{b}\rangle), \langle __(\mathtt{ch1}, \mathtt{ch2})\rangle __\langle \mathit{first} = \mathtt{b}, \mathit{first} = \mathtt{a}\rangle, \\ & \neg(\langle __(\mathtt{ch1}, \mathtt{ch2})\rangle __\langle \mathit{first} = \mathtt{b}, \mathit{first} = \mathtt{a}\rangle), __\langle \mathit{first} = \mathtt{b}, \mathit{first} = \mathtt{a}\rangle, \neg(__\langle \mathit{first} = \mathtt{b}, \mathit{first} = \mathtt{a}\rangle), \\ & __\langle \mathit{first} = \mathtt{a}, \mathit{first} = \mathtt{b}\rangle \supset \langle __(\mathtt{ch1}, \mathtt{ch2})\rangle __\langle \mathit{first} = \mathtt{b}, \mathit{first} = \mathtt{a}\rangle, \\ & \neg(__\langle \mathit{first} = \mathtt{a}, \mathit{first} = \mathtt{b}\rangle \supset \langle __(\mathtt{ch1}, \mathtt{ch2})\rangle __\langle \mathit{first} = \mathtt{b}, \mathit{first} = \mathtt{a}\rangle)\} \end{aligned}$$

El modelo filtrado no depende en este caso de la sustitución σ ya que el programa observado no tiene variables.

Para construir los estados del modelo filtrado, consideramos las distintas combinaciones de las fórmulas de $CL(\varphi)$, teniendo en cuenta que en cada combinación solo consideramos una fórmula o su negación.

El modelo filtrado tiene cinco estados:

- $[\mathtt{ab}]_{CL(\varphi)}$, con la cadena `ab`. El conjunto de fórmulas que se verifican en este estado es:

  $$\begin{aligned} \{ & \mathit{true}, \mathit{first} = \mathtt{a}, \neg(\mathit{first} = \mathtt{b}), __\langle \mathit{first} = \mathtt{a}, \mathit{first} = \mathtt{b}\rangle, \\ & \langle __(\mathtt{ch1}, \mathtt{ch2})\rangle __\langle \mathit{first} = \mathtt{b}, \mathit{first} = \mathtt{a}\rangle, \neg(__\langle \mathit{first} = \mathtt{b}, \mathit{first} = \mathtt{a}\rangle), \\ & __\langle \mathit{first} = \mathtt{a}, \mathit{first} = \mathtt{b}\rangle \supset \langle __(\mathtt{ch1}, \mathtt{ch2})\rangle __\langle \mathit{first} = \mathtt{b}, \mathit{first} = \mathtt{a}\rangle\}. \end{aligned}$$

- $[\mathtt{aba}]_{CL(\varphi)}$ con las cadenas de longitud mayor que dos, cuyo primer carácter es `a` y cuyo segundo carácter es `b`. El conjunto de fórmulas que se verifican en este estado es:

  $$\begin{aligned} \{ & \mathit{true}, \mathit{first} = \mathtt{a}, \neg(\mathit{first} = \mathtt{b}), __\langle \mathit{first} = \mathtt{a}, \mathit{first} = \mathtt{b}\rangle, \\ & \neg(\langle __(\mathtt{ch1}, \mathtt{ch2})\rangle __\langle \mathit{first} = \mathtt{b}, \mathit{first} = \mathtt{a}\rangle), \neg(__\langle \mathit{first} = \mathtt{b}, \mathit{first} = \mathtt{a}\rangle), \\ & \neg(__\langle \mathit{first} = \mathtt{a}, \mathit{first} = \mathtt{b}\rangle \supset \langle __(\mathtt{ch1}, \mathtt{ch2})\rangle __\langle \mathit{first} = \mathtt{b}, \mathit{first} = \mathtt{a}\rangle)\}. \end{aligned}$$

  Obsérvese que la acción $__(\mathtt{ch1}, \mathtt{ch2})$ no puede tener lugar en este estado ya que las cadenas de caracteres tienen longitud mayor que 2.

  Obsérvese también que este estado verifica la propiedad $__\langle \mathit{first} = \mathtt{a}, \mathit{first} = \mathtt{b}\rangle$ ya que por ejemplo, $[__(\mathtt{a}, \mathtt{ba})]_E \in [\mathtt{aba}]_{CL(\varphi)}$ y $[__(\mathtt{a}, \mathtt{ba})]_E, I, \sigma \models __\langle \mathit{first} = \mathtt{a}, \mathit{first} = \mathtt{b}\rangle$ para la interpretación de las observaciones proporcionada y para cualquier sustitución σ ya que los términos utilizados no tienen variables.

- $[\mathtt{ba}]_{CL(\varphi)}$, con las cadenas cuyo primer carácter es b y cuyo segundo carácter es a. El conjunto de fórmulas que se verifican en este estado es:

$$\{true, \neg(first = \mathtt{a}), first = \mathtt{b}, \neg(__\langle first = \mathtt{a}, first = \mathtt{b}\rangle), \neg(\langle __(\mathtt{ch1}, \mathtt{ch2})\rangle __\langle first = \mathtt{b}, first = \mathtt{a}\rangle), \neg(__\langle first = \mathtt{b}, first = \mathtt{a}\rangle), __\langle first = \mathtt{a}, first = \mathtt{b}\rangle \supset \langle __(\mathtt{ch1}, \mathtt{ch2})\rangle __\langle first = \mathtt{b}, first = \mathtt{a}\rangle\}.$$

- $[\mathtt{aa}]_{CL(\varphi)}$ con la constante a y las cadenas cuyo primer carácter es a y cuyo segundo carácter es a. El conjunto de fórmulas que se verifican en este estado es:

$$\{true, first = \mathtt{a}, \neg(first = \mathtt{b}), \neg(__\langle first = \mathtt{a}, first = \mathtt{b}\rangle), \neg(\langle __(\mathtt{ch1}, \mathtt{ch2})\rangle __\langle first = \mathtt{b}, first = \mathtt{a}\rangle), \neg(__\langle first = \mathtt{b}, first = \mathtt{a}\rangle), \neg(__\langle first = \mathtt{a}, first = \mathtt{b}\rangle \supset \langle __(\mathtt{ch1}, \mathtt{ch2})\rangle __\langle first = \mathtt{b}, first = \mathtt{a}\rangle)\}.$$

- $[\mathtt{bb}]_{CL(\varphi)}$ con la constante b y las cadenas cuyo primer carácter es b y cuyo segundo carácter es b. El conjunto de fórmulas que se verifican en este estado es:

$$\{true, \neg(first = \mathtt{a}), first = \mathtt{b}, \neg(__\langle first = \mathtt{a}, first = \mathtt{b}\rangle), \neg(\langle __(\mathtt{ch1}, \mathtt{ch2})\rangle __\langle first = \mathtt{b}, first = \mathtt{a}\rangle), \neg(__\langle first = \mathtt{b}, first = \mathtt{a}\rangle), __\langle first = \mathtt{a}, first = \mathtt{b}\rangle \supset \langle __(\mathtt{ch1}, \mathtt{ch2})\rangle __\langle first = \mathtt{b}, first = \mathtt{a}\rangle\}.$$

La única transición del modelo filtrado es $[__(\mathtt{ch1}, \mathtt{ch2})] : [\mathtt{ab}]_{CL(\varphi)} \rightarrow [\mathtt{ba}]_{CL(\varphi)}$ ya que el conjunto de acciones de la fórmula φ considerada tiene un único elemento $\{__(\mathtt{ch1}, \mathtt{ch2})\}$.

Definición de las relaciones

Encontramos las relaciones α-sucesor y sucesor-*espacial*.

Como solo existe una transición en el modelo filtrado, solo tenemos una relación del tipo α-sucesor: $[\mathtt{ab}]_{CL(\varphi)}$ tiene un $__(\mathtt{ch1}, \mathtt{ch2})$-sucesor $[\mathtt{ba}]_{CL(\varphi)}$.

Los sucesores-*espaciales* se obtienen de las posibles descomposiciones de los elementos de cada clase. De esta forma, la clase $[\mathtt{ab}]_{CL(\varphi)}$ solo contiene la cadena ab y esta cadena solo admite una descomposición cuyo primer elemento es a y cuyo segundo elemento es b. Como $[\mathtt{a}]_E \in [\mathtt{aa}]_{CL(\varphi)}$ y $[\mathtt{b}]_E \in [\mathtt{bb}]_{CL(\varphi)}$ el sucesor espacial es $([\mathtt{aa}]_{CL(\varphi)}, [\mathtt{bb}]_{CL(\varphi)})$.

Si consideramos la clase $[\mathtt{aba}]_{CL(\varphi)}$, que contiene las cadenas de longitud mayor que 2 cuyo primer elemento es a y cuyo segundo elemento es b, vemos que los elementos de esta clase admiten numerosas descomposiciones. Así , las cadenas pueden descomponerse en una primera cadena formada por la constante a y una segunda cadena cuyo único requisito es que empiece por b. Tenemos que $[\mathtt{a}]_E \in [\mathtt{aa}]_{CL(\varphi)}$, pero las cadenas que empiezan por b pueden estar en la clase $[\mathtt{ba}]_{CL(\varphi)}$ o en la clase $[\mathtt{bb}]_{CL(\varphi)}$. De esta forma tenemos que $([\mathtt{aa}]_{CL(\varphi)}, [\mathtt{ba}]_{CL(\varphi)})$ y $([\mathtt{aa}]_{CL(\varphi)}, [\mathtt{bb}]_{CL(\varphi)})$ son sucesores-*espaciales* de $[\mathtt{aba}]_{CL(\varphi)}$. Pero las cadenas pueden descomponerse de otras formas, por ejemplo una primera cadena ab y una segunda cadena a la que no se le impone ningún requisito. Aparecen entonces los sucesores-*espaciales* $([\mathtt{ab}]_{CL(\varphi)}, [\mathtt{ab}]_{CL(\varphi)})$, $([\mathtt{ab}]_{CL(\varphi)}, [\mathtt{aba}]_{CL(\varphi)})$, $([\mathtt{ab}]_{CL(\varphi)}, [\mathtt{ba}]_{CL(\varphi)})$, etc.

La relación completa de los sucesores-*espaciales* de todos los estados se muestra en la tabla siguiente.

Estado	sucesores-*espaciales*
$[\mathtt{ab}]_{CL(\varphi)}$	$([\mathtt{aa}]_{CL(\varphi)}, [\mathtt{bb}]_{CL(\varphi)})$
$[\mathtt{aba}]_{CL(\varphi)}$	$([\mathtt{ab}]_{CL(\varphi)}, [\mathtt{ab}]_{CL(\varphi)})$, $([\mathtt{ab}]_{CL(\varphi)}, [\mathtt{aba}]_{CL(\varphi)})$, $([\mathtt{ab}]_{CL(\varphi)}, [\mathtt{ba}]_{CL(\varphi)})$, $([\mathtt{ab}]_{CL(\varphi)}, [\mathtt{aa}]_{CL(\varphi)})$, $([\mathtt{ab}]_{CL(\varphi)}, [\mathtt{bb}]_{CL(\varphi)})$, $([\mathtt{aba}]_{CL(\varphi)}, [\mathtt{ab}]_{CL(\varphi)})$, $([\mathtt{aba}]_{CL(\varphi)}, [\mathtt{aba}]_{CL(\varphi)})$, $([\mathtt{aba}]_{CL(\varphi)}, [\mathtt{ba}]_{CL(\varphi)})$, $([\mathtt{aba}]_{CL(\varphi)}, [\mathtt{aa}]_{CL(\varphi)})$, $([\mathtt{aba}]_{CL(\varphi)}, [\mathtt{bb}]_{CL(\varphi)})$, $([\mathtt{aa}]_{CL(\varphi)}, [\mathtt{ba}]_{CL(\varphi)})$, $([\mathtt{aa}]_{CL(\varphi)}, [\mathtt{bb}]_{CL(\varphi)})$
$[\mathtt{ba}]_{CL(\varphi)}$	$([\mathtt{ba}]_{CL(\varphi)}, [\mathtt{ab}]_{CL(\varphi)})$, $([\mathtt{ba}]_{CL(\varphi)}, [\mathtt{aba}]_{CL(\varphi)})$, $([\mathtt{ba}]_{CL(\varphi)}, [\mathtt{ba}]_{CL(\varphi)})$, $([\mathtt{ba}]_{CL(\varphi)}, [\mathtt{aa}]_{CL(\varphi)})$, $([\mathtt{ba}]_{CL(\varphi)}, [\mathtt{bb}]_{CL(\varphi)})$, $([\mathtt{bb}]_{CL(\varphi)}, [\mathtt{ab}]_{CL(\varphi)})$, $([\mathtt{bb}]_{CL(\varphi)}, [\mathtt{aba}]_{CL(\varphi)})$, $([\mathtt{bb}]_{CL(\varphi)}, [\mathtt{aa}]_{CL(\varphi)})$
$[\mathtt{aa}]_{CL(\varphi)}$	$([\mathtt{aa}]_{CL(\varphi)}, [\mathtt{ab}]_{CL(\varphi)})$, $([\mathtt{aa}]_{CL(\varphi)}, [\mathtt{aba}]_{CL(\varphi)})$, $([\mathtt{aa}]_{CL(\varphi)}, [\mathtt{ba}]_{CL(\varphi)})$, $([\mathtt{aa}]_{CL(\varphi)}, [\mathtt{aa}]_{CL(\varphi)})$, $([\mathtt{aa}]_{CL(\varphi)}, [\mathtt{bb}]_{CL(\varphi)})$,
$[\mathtt{bb}]_{CL(\varphi)}$	$([\mathtt{bb}]_{CL(\varphi)}, [\mathtt{ab}]_{CL(\varphi)})$, $([\mathtt{bb}]_{CL(\varphi)}, [\mathtt{aba}]_{CL(\varphi)})$, $([\mathtt{bb}]_{CL(\varphi)}, [\mathtt{ba}]_{CL(\varphi)})$, $([\mathtt{bb}]_{CL(\varphi)}, [\mathtt{aa}]_{CL(\varphi)})$, $([\mathtt{bb}]_{CL(\varphi)}, [\mathtt{bb}]_{CL(\varphi)})$,

Ejecución del algoritmo

Paso 1

Consideramos las fórmulas básicas $\mathit{first} = \mathtt{a}$, $\neg(\mathit{first} = \mathtt{a})$, $\mathit{first} = \mathtt{b}$, $\neg \mathit{first} = \mathtt{b}$ y las añadimos al conjunto *label* de aquellos estados que las satisfacen. El conjunto *label* de cada estado se muestra en la tabla siguiente.

Estado	$\mathit{label}([t]_{CL(\varphi)})$
$[\mathtt{ab}]_{CL(\varphi)}$	$\mathit{first} = \mathtt{a}$; $\neg(\mathit{first} = \mathtt{b})$
$[\mathtt{aba}]_{CL(\varphi)}$	$\mathit{first} = \mathtt{a}$; $\neg(\mathit{first} = \mathtt{b})$
$[\mathtt{ba}]_{CL(\varphi)}$	$\neg(\mathit{first} = \mathtt{a})$; $\mathit{first} = \mathtt{b}$
$[\mathtt{aa}]_{CL(\varphi)}$	$\mathit{first} = \mathtt{a}$; $\neg(\mathit{first} = \mathtt{b})$
$[\mathtt{bb}]_{CL(\varphi)}$	$\neg(\mathit{first} = \mathtt{a})$; $\mathit{first} = \mathtt{b}$

Paso 2.1

Consideramos las fórmulas $__\langle \mathit{first} = \mathtt{a}, \mathit{first} = \mathtt{b}\rangle \supset \langle __(\mathtt{ch1}, \mathtt{ch2})\rangle __\langle \mathit{first} = \mathtt{b}, \mathit{first} = \mathtt{a}\rangle$ y $\neg(__\langle \mathit{first} = \mathtt{a}, \mathit{first} = \mathtt{b}\rangle \supset \langle __(\mathtt{ch1}, \mathtt{ch2})\rangle __\langle \mathit{first} = \mathtt{b}, \mathit{first} = \mathtt{a}\rangle)$ y comprobamos las condiciones para añadirlas a los conjuntos *label* de estados.

Dado que $\neg(__\langle \mathit{first} = \mathtt{a}, \mathit{first} = \mathtt{b}\rangle) \notin \mathit{label}([t]_{CL(\varphi)})$ y $__\langle \mathit{first} = \mathtt{b}, \mathit{first} = \mathtt{a}\rangle \notin \mathit{label}([t]_{CL(\varphi)})$ para ningún $[t]_{CL(\varphi)}$, $__\langle \mathit{first} = \mathtt{a}, \mathit{first} = \mathtt{b}\rangle \supset \langle __(\mathtt{ch1}, \mathtt{ch2})\rangle __\langle \mathit{first} = \mathtt{b}, \mathit{first} = \mathtt{a}\rangle$ no se añade a ningún conjunto.

Dado que $__\langle \mathit{first} = \mathtt{a}, \mathit{first} = \mathtt{b}\rangle \notin \mathit{label}([t]_{CL(\varphi)})$ y $\neg(__\langle \mathit{first} = \mathtt{b}, \mathit{first} = \mathtt{a}\rangle) \notin \mathit{label}([t]_{CL(\varphi)})$ para ningún $[t]_{CL(\varphi)}$, $\neg(__\langle \mathit{first} = \mathtt{a}, \mathit{first} = \mathtt{b}\rangle \supset \langle __(\mathtt{ch1}, \mathtt{ch2})\rangle __\langle \mathit{first} = \mathtt{b}, \mathit{first} = \mathtt{a}\rangle)$ no se añade a ningún conjunto.

Paso 3.1

Consideramos las fórmulas $\langle__(\mathtt{ch1},\mathtt{ch2})\rangle__\langle \mathit{first}=\mathtt{b},\mathit{first}=\mathtt{a}\rangle$ y $\neg(\langle__(\mathtt{ch1},\mathtt{ch2})\rangle__\langle \mathit{first}=\mathtt{b},\mathit{first}=\mathtt{a}\rangle)$ y comprobamos las condiciones respectivas para añadirlas al conjunto *label* de los estados. Obsérvese que $\neg(\langle__(\mathtt{ch1},\mathtt{ch2})\rangle__\langle \mathit{first}=\mathtt{b},\mathit{first}=\mathtt{a}\rangle)\equiv[__(\mathtt{ch1},\mathtt{ch2})]\neg__\langle \mathit{first}=\mathtt{b},\mathit{first}=\mathtt{a}\rangle$.

Tenemos que $[\mathtt{ba}]_{CL(\varphi)}$ es un $__(\mathtt{ch1},\mathtt{ch2})$-sucesor de $[\mathtt{ab}]_{CL(\varphi)}$. Entonces, la condición (4.6) es satisfecha por los estados $[\mathtt{aba}]_{CL(\varphi)}$, $[\mathtt{ba}]_{CL(\varphi)}$, $[\mathtt{aa}]_{CL(\varphi)}$, y $[\mathtt{bb}]_{CL(\varphi)}$, y las condiciones (4.7) y (4.8) no se satisfacen ya que ni $__\langle \mathit{first}=\mathtt{b},\mathit{first}=\mathtt{a}\rangle$ ni $\neg__\langle \mathit{first}=\mathtt{b},\mathit{first}=\mathtt{a}\rangle$ pertenecen a $\mathit{label}([\mathtt{ba}]_{CL(\varphi)})$.

Los conjuntos de fórmulas son los siguientes:

Estado	$\mathit{label}([t]_{CL(\varphi)})$
$[\mathtt{ab}]_{CL(\varphi)}$	$\mathit{first}=\mathtt{a}$; $\neg(\mathit{first}=\mathtt{b})$
$[\mathtt{aba}]_{CL(\varphi)}$	$\mathit{first}=\mathtt{a}$; $\neg(\mathit{first}=\mathtt{b})$; $[__(\mathtt{ch1},\mathtt{ch2})]\neg__\langle \mathit{first}=\mathtt{b},\mathit{first}=\mathtt{a}\rangle$
$[\mathtt{ba}]_{CL(\varphi)}$	$\neg(\mathit{first}=\mathtt{a})$; $\mathit{first}=\mathtt{b}$; $[__(\mathtt{ch1},\mathtt{ch2})]\neg__\langle \mathit{first}=\mathtt{b},\mathit{first}=\mathtt{a}\rangle$
$[\mathtt{aa}]_{CL(\varphi)}$	$\mathit{first}=\mathtt{a}$; $\neg(\mathit{first}=\mathtt{b})$; $[__(\mathtt{ch1},\mathtt{ch2})]\neg__\langle \mathit{first}=\mathtt{b},\mathit{first}=\mathtt{a}\rangle$
$[\mathtt{bb}]_{CL(\varphi)}$	$\neg(\mathit{first}=\mathtt{a})$; $\mathit{first}=\mathtt{b}$; $[__(\mathtt{ch1},\mathtt{ch2})]\neg__\langle \mathit{first}=\mathtt{b},\mathit{first}=\mathtt{a}\rangle$

Paso 4.1

Consideramos la fórmula $__\langle \mathit{first}=\mathtt{a},\mathit{first}=\mathtt{b}\rangle$ y el estado $[\mathtt{ab}]_{CL(\varphi)}$. Este estado tiene un solo sucesor-*espacial* $([\mathtt{aa}]_{CL(\varphi)},[\mathtt{bb}]_{CL(\varphi)})$; como $\mathit{first}=\mathtt{a}\in \mathit{label}([\mathtt{aa}]_{CL(\varphi)})$ y $\mathit{first}=\mathtt{b}\in \mathit{label}([\mathtt{bb}]_{CL(\varphi)})$, añadimos $__\langle \mathit{first}=\mathtt{a},\mathit{first}=\mathtt{b}\rangle$ a $\mathit{label}([\mathtt{ab}]_{CL(\varphi)})$.

Comprobamos todos los estados y las otras tres fórmulas con modalidad espacial y obtenemos los siguientes conjuntos. Obsérvese que $\neg(__\langle \mathit{first}=\mathtt{a},\mathit{first}=\mathtt{b}\rangle\equiv__[\neg(\mathit{first}=\mathtt{a}),\neg(\mathit{first}=\mathtt{b})]$.

Estado	$\mathit{label}([t]_{CL(\varphi)})$
$[\mathtt{ab}]_{CL(\varphi)}$	$\mathit{first}=\mathtt{a}$; $\neg(\mathit{first}=\mathtt{b})$; $__\langle \mathit{first}=\mathtt{a},\mathit{first}=\mathtt{b}\rangle$; $__[\neg(\mathit{first}=\mathtt{b}),\neg(\mathit{first}=\mathtt{a})]$
$[\mathtt{aba}]_{CL(\varphi)}$	$\mathit{first}=\mathtt{a}$; $\neg(\mathit{first}=\mathtt{b})$; $[__(\mathtt{ch1},\mathtt{ch2})]\neg__\langle \mathit{first}=\mathtt{b},\mathit{first}=\mathtt{a}\rangle$; $__\langle \mathit{first}=\mathtt{a},\mathit{first}=\mathtt{b}\rangle$; $__[\neg(\mathit{first}=\mathtt{b}),\neg(\mathit{first}=\mathtt{a})]$
$[\mathtt{ba}]_{CL(\varphi)}$	$\neg(\mathit{first}=\mathtt{a})$; $\mathit{first}=\mathtt{b}$; $[__(\mathtt{ch1},\mathtt{ch2})]\neg__\langle \mathit{first}=\mathtt{b},\mathit{first}=\mathtt{a}\rangle$; $__\langle \mathit{first}=\mathtt{b},\mathit{first}=\mathtt{a}\rangle$; $__[\neg(\mathit{first}=\mathtt{a}),\neg(\mathit{first}=\mathtt{b})]$
$[\mathtt{aa}]_{CL(\varphi)}$	$\mathit{first}=\mathtt{a}$; $\neg(\mathit{first}=\mathtt{b})$; $[__(\mathtt{ch1},\mathtt{ch2})]\neg__\langle \mathit{first}=\mathtt{b},\mathit{first}=\mathtt{a}\rangle$; $__[\neg(\mathit{first}=\mathtt{a}),\neg(\mathit{first}=\mathtt{b})]$; $__[\neg(\mathit{first}=\mathtt{b}),\neg(\mathit{first}=\mathtt{a})]$
$[\mathtt{bb}]_{CL(\varphi)}$	$\neg(\mathit{first}=\mathtt{a})$; $\mathit{first}=\mathtt{b}$; $[__(\mathtt{ch1},\mathtt{ch2})]\neg__\langle \mathit{first}=\mathtt{b},\mathit{first}=\mathtt{a}\rangle$; $__[\neg(\mathit{first}=\mathtt{a}),\neg(\mathit{first}=\mathtt{b})]$; $__[\neg(\mathit{first}=\mathtt{b}),\neg(\mathit{first}=\mathtt{a})]$

Paso 2.2

Consideramos las fórmulas $__\langle \mathit{first}=\mathtt{a},\mathit{first}=\mathtt{b}\rangle\supset\langle__(\mathtt{ch1},\mathtt{ch2})\rangle__\langle \mathit{first}=\mathtt{b},\mathit{first}=\mathtt{a}\rangle$ y $\neg(__\langle \mathit{first}=\mathtt{a},\mathit{first}=\mathtt{b}\rangle\supset\langle__(\mathtt{ch1},\mathtt{ch2})\rangle__\langle \mathit{first}=\mathtt{b},\mathit{first}=\mathtt{a}\rangle)$ y comprobamos

las respectivas condiciones para añadirlas a los conjuntos *label* de los estados. La condición (4.3) se satisface para los estados $[\mathtt{ba}]_{CL(\varphi)}$, $[\mathtt{aa}]_{CL(\varphi)}$ y $[\mathtt{bb}]_{CL(\varphi)}$. Las condiciones (4.4) y (4.5) no se satisfacen para ningún estado.

Los conjuntos de fórmulas son ahora los siguientes:

estado	$label([t]_{CL(\varphi)})$
$[\mathtt{ab}]_{CL(\varphi)}$	$\mathit{first} = \mathtt{a}$; $\neg(\mathit{first} = \mathtt{b})$; $__\langle \mathit{first} = \mathtt{a}, \mathit{first} = \mathtt{b}\rangle$; $__[\neg(\mathit{first} = \mathtt{b}), \neg(\mathit{first} = \mathtt{a})]$
$[\mathtt{aba}]_{CL(\varphi)}$	$\mathit{first} = \mathtt{a}$; $\neg(\mathit{first} = \mathtt{b})$; $[__(\mathtt{ch1}, \mathtt{ch2})]\neg__\langle \mathit{first} = \mathtt{b}, \mathit{first} = \mathtt{a}\rangle$; $__\langle \mathit{first} = \mathtt{a}, \mathit{first} = \mathtt{b}\rangle$; $__[\neg(\mathit{first} = \mathtt{b}), \neg(\mathit{first} = \mathtt{a})]$
$[\mathtt{ba}]_{CL(\varphi)}$	$\neg(\mathit{first} = \mathtt{a})$; $\mathit{first} = \mathtt{b}$; $[__(\mathtt{ch1}, \mathtt{ch2})]\neg__\langle \mathit{first} = \mathtt{b}, \mathit{first} = \mathtt{a}\rangle$; $__\langle \mathit{first} = \mathtt{b}, \mathit{first} = \mathtt{a}\rangle$; $__[\neg(\mathit{first} = \mathtt{a}), \neg(\mathit{first} = \mathtt{b})]$; $__\langle \mathit{first} = \mathtt{a}, \mathit{first} = \mathtt{b}\rangle \supset \langle__(\mathtt{ch1}, \mathtt{ch2})\rangle__\langle \mathit{first} = \mathtt{b}, \mathit{first} = \mathtt{a}\rangle$
$[\mathtt{aa}]_{CL(\varphi)}$	$\mathit{first} = \mathtt{a}$; $\neg(\mathit{first} = \mathtt{b})$; $[__(\mathtt{ch1}, \mathtt{ch2})]\neg__\langle \mathit{first} = \mathtt{b}, \mathit{first} = \mathtt{a}\rangle$; $__[\neg(\mathit{first} = \mathtt{a}), \neg(\mathit{first} = \mathtt{b})]$; $__[\neg(\mathit{first} = \mathtt{b}), \neg(\mathit{first} = \mathtt{a})]$; $__\langle \mathit{first} = \mathtt{a}, \mathit{first} = \mathtt{b}\rangle \supset \langle__(\mathtt{ch1}, \mathtt{ch2})\rangle__\langle \mathit{first} = \mathtt{b}, \mathit{first} = \mathtt{a}\rangle$
$[\mathtt{bb}]_{CL(\varphi)}$	$\neg(\mathit{first} = \mathtt{a})$; $\mathit{first} = \mathtt{b}$; $[__(\mathtt{ch1}, \mathtt{ch2})]\neg__\langle \mathit{first} = \mathtt{b}, \mathit{first} = \mathtt{a}\rangle$; $__[\neg(\mathit{first} = \mathtt{a}), \neg(\mathit{first} = \mathtt{b})]$; $__[\neg(\mathit{first} = \mathtt{b}), \neg(\mathit{first} = \mathtt{a})]$; $__\langle \mathit{first} = \mathtt{a}, \mathit{first} = \mathtt{b}\rangle \supset \langle__(\mathtt{ch1}, \mathtt{ch2})\rangle__\langle \mathit{first} = \mathtt{b}, \mathit{first} = \mathtt{a}\rangle$

Paso 3.2

Consideramos las fórmulas $\langle__(\mathtt{ch1}, \mathtt{ch2})\rangle__\langle \mathit{first} = \mathtt{b}, \mathit{first} = \mathtt{a}\rangle$ y $\neg(\langle__(\mathtt{ch1}, \mathtt{ch2})\rangle__\langle \mathit{first} = \mathtt{b}, \mathit{first} = \mathtt{a}\rangle)$ y comprobamos las respectivas condiciones para añadirlas a los conjuntos *label* de los estados. No añadimos las fórmulas que ya han sido añadidas en algún paso previo.

El conjunto $[\mathtt{ab}]_{CL}(\varphi)$ satisface la condición (4.8) por lo que añadimos la fórmula $\langle__(\mathtt{ch1}, \mathtt{ch2})\rangle__\langle \mathit{first} = \mathtt{b}, \mathit{first} = \mathtt{a}\rangle$ a su conjunto de etiquetas.

Los conjuntos de fórmulas son ahora los siguientes:

Estado	$label([t]_{CL(\varphi)})$
$[\mathtt{ab}]_{CL(\varphi)}$	$\mathit{first} = \mathtt{a}$; $\neg(\mathit{first} = \mathtt{b})$; $__\langle \mathit{first} = \mathtt{a}, \mathit{first} = \mathtt{b}\rangle$; $__[\neg(\mathit{first} = \mathtt{b}), \neg(\mathit{first} = \mathtt{a})]$; $\langle__(\mathtt{ch1}, \mathtt{ch2})\rangle__\langle \mathit{first} = \mathtt{b}, \mathit{first} = \mathtt{a}\rangle$
$[\mathtt{aba}]_{CL(\varphi)}$	$\mathit{first} = \mathtt{a}$; $\neg(\mathit{first} = \mathtt{b})$; $[__(\mathtt{ch1}, \mathtt{ch2})]\neg__\langle \mathit{first} = \mathtt{b}, \mathit{first} = \mathtt{a}\rangle$; $__\langle \mathit{first} = \mathtt{a}, \mathit{first} = \mathtt{b}\rangle$; $__[\neg(\mathit{first} = \mathtt{b}), \neg(\mathit{first} = \mathtt{a})]$
$[\mathtt{ba}]_{CL(\varphi)}$	$\neg(\mathit{first} = \mathtt{a})$; $\mathit{first} = \mathtt{b}$; $[__(\mathtt{ch1}, \mathtt{ch2})]\neg__\langle \mathit{first} = \mathtt{b}, \mathit{first} = \mathtt{a}\rangle$; $__\langle \mathit{first} = \mathtt{b}, \mathit{first} = \mathtt{a}\rangle$; $__[\neg(\mathit{first} = \mathtt{a}), \neg(\mathit{first} = \mathtt{b})]$; $__\langle \mathit{first} = \mathtt{a}, \mathit{first} = \mathtt{b}\rangle \supset \langle__(\mathtt{ch1}, \mathtt{ch2})\rangle__\langle \mathit{first} = \mathtt{b}, \mathit{first} = \mathtt{a}\rangle$
$[\mathtt{aa}]_{CL(\varphi)}$	$\mathit{first} = \mathtt{a}$; $\neg(\mathit{first} = \mathtt{b})$; $[__(\mathtt{ch1}, \mathtt{ch2})]\neg__\langle \mathit{first} = \mathtt{b}, \mathit{first} = \mathtt{a}\rangle$; $__[\neg(\mathit{first} = \mathtt{a}), \neg(\mathit{first} = \mathtt{b})]$; $__[\neg(\mathit{first} = \mathtt{b}), \neg(\mathit{first} = \mathtt{a})]$; $__\langle \mathit{first} = \mathtt{a}, \mathit{first} = \mathtt{b}\rangle \supset \langle__(\mathtt{ch1}, \mathtt{ch2})\rangle__\langle \mathit{first} = \mathtt{b}, \mathit{first} = \mathtt{a}\rangle$
$[\mathtt{bb}]_{CL(\varphi)}$	$\neg(\mathit{first} = \mathtt{a})$; $\mathit{first} = \mathtt{b}$; $[__(\mathtt{ch1}, \mathtt{ch2})]\neg__\langle \mathit{first} = \mathtt{b}, \mathit{first} = \mathtt{a}\rangle$; $__[\neg(\mathit{first} = \mathtt{a}), \neg(\mathit{first} = \mathtt{b})]$; $__[\neg(\mathit{first} = \mathtt{b}), \neg(\mathit{first} = \mathtt{a})]$; $__\langle \mathit{first} = \mathtt{a}, \mathit{first} = \mathtt{b}\rangle \supset \langle__(\mathtt{ch1}, \mathtt{ch2})\rangle__\langle \mathit{first} = \mathtt{b}, \mathit{first} = \mathtt{a}\rangle$

Paso 4.2

Las dos fórmulas con modalidad espacial y sus negaciones ya han sido añadidas a todos los conjuntos de todos los estados, por lo tanto no es necesario volver a tratarlas.

Paso 2.3

Consideramos las fórmulas $__\langle \mathit{first} = \mathtt{a}, \mathit{first} = \mathtt{b}\rangle \supset \langle __(\mathtt{ch1}, \mathtt{ch2})\rangle __\langle \mathit{first} = \mathtt{b}, \mathit{first} = \mathtt{a}\rangle$ y $\neg(__\langle \mathit{first} = \mathtt{a}, \mathit{first} = \mathtt{b}\rangle \supset \langle __(\mathtt{ch1}, \mathtt{ch2})\rangle __\langle \mathit{first} = \mathtt{b}, \mathit{first} = \mathtt{a}\rangle)$ y comprobamos las respectivas condiciones para añadirlas a los conjuntos *label* de los estados. La condición (4.4) es satisfecha por $[\mathtt{ab}]_{CL(\varphi)}$ y la condición (4.5) es satisfecha por $[\mathtt{aba}]_{CL(\varphi)}$.

Los conjuntos de fórmulas son ahora los siguientes:

Estado	$\mathit{label}([t]_{CL(\varphi)})$
$[\mathtt{ab}]_{CL(\varphi)}$	$\mathit{first} = \mathtt{a}$; $\neg(\mathit{first} = \mathtt{b})$; $__\langle \mathit{first} = \mathtt{a}, \mathit{first} = \mathtt{b}\rangle$; $__[\neg(\mathit{first} = \mathtt{b}), \neg(\mathit{first} = \mathtt{a})]$; $\langle __(\mathtt{ch1}, \mathtt{ch2})\rangle __\langle \mathit{first} = \mathtt{b}, \mathit{first} = \mathtt{a}\rangle$; $__\langle \mathit{first} = \mathtt{a}, \mathit{first} = \mathtt{b}\rangle \supset \langle __(\mathtt{ch1}, \mathtt{ch2})\rangle __\langle \mathit{first} = \mathtt{b}, \mathit{first} = \mathtt{a}\rangle$
$[\mathtt{aba}]_{CL(\varphi)}$	$\mathit{first} = \mathtt{a}$; $\neg(\mathit{first} = \mathtt{b})$; $[__(\mathtt{ch1}, \mathtt{ch2})]\neg__\langle \mathit{first} = \mathtt{b}, \mathit{first} = \mathtt{a}\rangle$; $__\langle \mathit{first} = \mathtt{a}, \mathit{first} = \mathtt{b}\rangle$; $__[\neg(\mathit{first} = \mathtt{b}), \neg(\mathit{first} = \mathtt{a})]$; $\neg(__\langle \mathit{first} = \mathtt{a}, \mathit{first} = \mathtt{b}\rangle \supset \langle __(\mathtt{ch1}, \mathtt{ch2})\rangle __\langle \mathit{first} = \mathtt{b}, \mathit{first} = \mathtt{a}\rangle)$
$[\mathtt{ba}]_{CL(\varphi)}$	$\neg(\mathit{first} = \mathtt{a})$; $\mathit{first} = \mathtt{b}$; $[__(\mathtt{ch1}, \mathtt{ch2})]\neg__\langle \mathit{first} = \mathtt{b}, \mathit{first} = \mathtt{a}\rangle$; $__\langle \mathit{first} = \mathtt{b}, \mathit{first} = \mathtt{a}\rangle$; $__[\neg(\mathit{first} = \mathtt{a}), \neg(\mathit{first} = \mathtt{b})]$; $__\langle \mathit{first} = \mathtt{a}, \mathit{first} = \mathtt{b}\rangle \supset \langle __(\mathtt{ch1}, \mathtt{ch2})\rangle __\langle \mathit{first} = \mathtt{b}, \mathit{first} = \mathtt{a}\rangle$
$[\mathtt{aa}]_{CL(\varphi)}$	$\mathit{first} = \mathtt{a}$; $\neg(\mathit{first} = \mathtt{b})$; $[__(\mathtt{ch1}, \mathtt{ch2})]\neg__\langle \mathit{first} = \mathtt{b}, \mathit{first} = \mathtt{a}\rangle$; $__[\neg(\mathit{first} = \mathtt{a}), \neg(\mathit{first} = \mathtt{b})]$; $__[\neg(\mathit{first} = \mathtt{b}), \neg(\mathit{first} = \mathtt{a})]$; $__\langle \mathit{first} = \mathtt{a}, \mathit{first} = \mathtt{b}\rangle \supset \langle __(\mathtt{ch1}, \mathtt{ch2})\rangle __\langle \mathit{first} = \mathtt{b}, \mathit{first} = \mathtt{a}\rangle$
$[\mathtt{bb}]_{CL(\varphi)}$	$\neg(\mathit{first} = \mathtt{a})$; $\mathit{first} = \mathtt{b}$; $[__(\mathtt{ch1}, \mathtt{ch2})]\neg__\langle \mathit{first} = \mathtt{b}, \mathit{first} = \mathtt{a}\rangle$; $__[\neg(\mathit{first} = \mathtt{a}), \neg(\mathit{first} = \mathtt{b})]$; $__[\neg(\mathit{first} = \mathtt{b}), \neg(\mathit{first} = \mathtt{a})]$; $__\langle \mathit{first} = \mathtt{a}, \mathit{first} = \mathtt{b}\rangle \supset \langle __(\mathtt{ch1}, \mathtt{ch2})\rangle __\langle \mathit{first} = \mathtt{b}, \mathit{first} = \mathtt{a}\rangle$

Como todas las fórmulas ya han sido tratadas para todos los estados, no se añaden más fórmulas a ningún conjunto en los pasos 3.3, 4.3, 2.4, 3.4, y 4.4. Por lo tanto el algoritmo termina, ya que no se añade ninguna fórmula en los pasos 2.4, 3.4 y 4.4.

Como la fórmula φ pertenece a $\mathit{label}([\mathtt{ab}]_{CL(\varphi)})$, es satisfactible.

4.4. Teoría de demostración para la lógica VLRL

4.4.1. Modalidad de acción

Los modelos de Kripke que hemos adoptado tienen propiedades estándar en lo que se refiere a la modalidad de acción. La corrección de todas ellas se prueba de forma inmediata.

La primera regla se suele denotar como K [Gol92]:

$$\vdash [\alpha](\varphi \supset \psi) \supset ([\alpha]\varphi \supset [\alpha]\psi) \ . \tag{4.11}$$

Debido a que las acciones son deterministas, como se ha explicado anteriormente, la propiedad *es posible hacer α en el estado actual y en el estado resultante se obtendrá la propiedad φ* es equivalente a la afirmación *después de realizar α el estado resultante tiene la propiedad φ y es posible realizar α*:

$$\vdash \langle\alpha\rangle\varphi \equiv [\alpha]\varphi \wedge \langle\alpha\rangle true \ . \tag{4.12}$$

La modalidad también actúa sobre la conjunción[5]:

$$\vdash [\alpha](\varphi \wedge \psi) \supset ([\alpha]\varphi \wedge [\alpha]\psi) \ . \tag{4.13}$$

La inevitabilidad también se cumple:

$$\varphi \vdash [\alpha]\varphi \ . \tag{4.14}$$

Hay formas más fuertes de inevitabilidad que se aplican a propiedades que no incluyen observaciones de estado. Respecto a la modalidad de acción, en la regla siguiente, sean t_1 y t_2 términos en $T_{\Sigma^+}(X)$:

$$\vdash t_1 = t_2 \supset [\alpha](t_1 = t_2) \ . \tag{4.15}$$

La cuantificación sobre variables que no sean de estado (rígidas) conmuta con las modalidades de acción:

$$\vdash \exists X.[\alpha]\varphi \equiv [\alpha]\exists X.\varphi \ . \tag{4.16}$$

La siguiente regla define la dualidad entre los dos operadores:

$$\vdash \langle\alpha\rangle\varphi \equiv \neg[\alpha]\neg\varphi \ . \tag{4.17}$$

A partir de estas reglas se pueden derivar las siguientes:

$$\vdash [\alpha]true \tag{4.18}$$

se deriva directamente de (4.14).

$$\varphi \supset \psi \vdash [\alpha]\varphi \supset [\alpha]\psi \tag{4.19}$$

se deriva directamente de (4.14) y (4.11).

$$\varphi \supset \psi \vdash \langle\alpha\rangle\varphi \supset \langle\alpha\rangle\psi \tag{4.20}$$

La derivación es como sigue:

$\vdash \varphi \supset \psi$	
$\vdash \neg\psi \supset \neg\varphi$	por lógica proposicional
$\vdash [\alpha]\neg\psi \supset [\alpha]\neg\varphi$	por (4.11)
$\vdash \neg\langle\alpha\rangle\psi \supset \neg\langle\alpha\rangle\varphi$	por (4.17)
$\vdash \langle\alpha\rangle\varphi \supset \langle\alpha\rangle\psi$	por lógica proposicional

[5] La conjunción se define a partir de la implicación como en la lógica proposicional.

$$\vdash (\langle\alpha\rangle\varphi \vee \langle\alpha\rangle\psi) \supset \langle\alpha\rangle(\varphi \vee \psi) \tag{4.21}$$

La derivación es como sigue:

$\vdash [\alpha](\neg\varphi \wedge \neg\psi) \supset ([\alpha]\neg\varphi \wedge [\alpha]\neg\psi)$	por (4.13)
$\vdash \neg\langle\alpha\rangle\neg(\neg\varphi \wedge \neg\psi) \supset (\neg\langle\alpha\rangle\varphi \wedge \neg\langle\alpha\rangle\psi)$	por (4.17)
$\vdash \neg\langle\alpha\rangle(\varphi \vee \psi) \supset \neg(\langle\alpha\rangle\varphi \vee \langle\alpha\rangle\psi)$	por lógica proposicional
$\vdash (\langle\alpha\rangle\varphi \vee \langle\alpha\rangle\psi) \supset \langle\alpha\rangle(\varphi \vee \psi)$	por lógica proposicional

$$\vdash (\langle\alpha\rangle\varphi \wedge [\alpha]\psi) \supset \langle\alpha\rangle(\varphi \wedge \psi) \tag{4.22}$$

La derivación es como sigue:

$\vdash [\alpha](\psi \supset \neg\varphi) \supset ([\alpha]\psi \supset [\alpha]\neg\varphi)$	por (4.11)
$\vdash [\alpha](\varphi \supset \neg\psi) \supset ([\alpha]\psi \supset [\alpha]\neg\varphi)$	por lógica proposicional
$\vdash [\alpha](\neg\varphi \vee \neg\psi) \supset (\neg[\alpha]\psi \vee [\alpha]\neg\varphi)$	por lógica proposicional
$\vdash \neg\langle\alpha\rangle\neg(\neg\varphi \vee \neg\psi) \supset (\neg[\alpha]\psi \vee [\alpha]\neg\varphi)$	por (4.17)
$\vdash \neg\langle\alpha\rangle(\varphi \wedge \psi) \supset \neg([\alpha]\psi \wedge \neg[\alpha]\neg\varphi)$	por lógica proposicional
$\vdash ([\alpha]\psi \wedge \neg[\alpha]\neg\varphi) \supset \langle\alpha\rangle(\varphi \wedge \psi)$	por lógica proposicional
$\vdash ([\alpha]\psi \wedge \langle\alpha\rangle\varphi) \supset \langle\alpha\rangle(\varphi \wedge \psi)$	por (4.17)
$\vdash (\langle\alpha\rangle\varphi \wedge [\alpha]\psi) \supset \langle\alpha\rangle(\varphi \wedge \psi)$	por lógica proposicional

$$\vdash (\langle\alpha\rangle\varphi \wedge \langle\alpha\rangle\psi) \supset \langle\alpha\rangle(\varphi \wedge \psi) \tag{4.23}$$

Como consecuencia de (4.22) y (4.12) tenemos:

$$\vdash [\alpha](\varphi \vee \psi) \supset (\langle\alpha\rangle\varphi \vee [\alpha]\psi) \tag{4.24}$$

Se deriva de (4.22) aplicando la regla (4.17) y lógica proposicional.

$$\vdash (\langle\alpha\rangle(\varphi \vee \psi) \supset (\langle\alpha\rangle\varphi \vee \langle\alpha\rangle\psi) \tag{4.25}$$

La derivación es como sigue:

$\vdash [\alpha](\varphi \vee \psi) \supset (\langle\alpha\rangle\varphi \vee [\alpha]\psi)$	por (4.24)
$\vdash [\alpha](\psi \vee \varphi) \supset (\langle\alpha\rangle\psi \vee [\alpha]\varphi)$	por (4.24)
$\vdash [\alpha](\psi \vee \varphi) \supset (\langle\alpha\rangle\psi \vee \langle\alpha\rangle\varphi)$	por lógica proposicional
$\vdash \langle\alpha\rangle(\psi \vee \varphi) \supset (\langle\alpha\rangle\psi \vee \langle\alpha\rangle\varphi)$	por lógica proposicional

$$\vdash ([\alpha]\varphi \wedge [\alpha]\psi) \supset [\alpha](\varphi \wedge \psi) \tag{4.26}$$

La derivación es como sigue:

$\vdash \langle\alpha\rangle(\neg\varphi \vee \neg\psi) \supset (\langle\alpha\rangle\neg\varphi \vee \langle\alpha\rangle\neg\psi)$ por (4.25)
$\vdash \langle\alpha\rangle\neg(\varphi \wedge \psi) \supset (\neg[\alpha]\varphi \vee \neg[\alpha]\psi)$ por lógica proposicional y (4.17)
$\vdash \neg[\alpha](\varphi \wedge \psi) \supset \neg([\alpha]\varphi \wedge [\alpha]\psi)$ por lógica proposicional y (4.17)
$\vdash ([\alpha]\varphi \wedge [\alpha]\psi) \supset [\alpha](\varphi \wedge \psi)$ por lógica proposicional

4.4.2. Modalidad espacial

El operador espacial satisface una versión generalizada del axioma K de las lógicas modales:

$$\{\varphi_i \supset \psi_i \mid i \in \{1,\ldots,n\}\} \vdash f_{\overline{d}}[\varphi_1,\ldots,\varphi_n] \supset f_{\overline{d}}[\psi_1,\ldots,\psi_n] \ . \tag{4.27}$$

El operador dual también la satisface

$$\{\varphi_i \supset \psi_i \mid i \in \{1,\ldots,n\}\} \vdash f_{\overline{d}}\langle\varphi_1,\ldots,\varphi_n\rangle \supset f_{\overline{d}}\langle\psi_1,\ldots,\psi_n\rangle \ . \tag{4.28}$$

A partir de estas reglas podemos derivar:

$$\vdash f_{\overline{d}}[\overline{\mathit{false}}] \supset f_{\overline{d}}[\varphi_1,\ldots,\varphi_n] \tag{4.29}$$

$$\varphi_1,\ldots,\varphi_n \vdash f_{\overline{d}}\langle\overline{\mathit{true}}\rangle \supset f_{\overline{d}}\langle\varphi_1,\ldots,\varphi_n\rangle \ . \tag{4.30}$$

También podemos derivar los resultados siguientes para la conjunción y la disyunción:

$$\vdash f_{\overline{d}}\langle\varphi_1,\ldots,\psi_1 \wedge \psi_2,\ldots\varphi_n\rangle \supset f_{\overline{d}}\langle\varphi_1,\ldots,\psi_1,\ldots,\varphi_n\rangle \wedge f_{\overline{d}}\langle\varphi_1,\ldots,\psi_2,\ldots,\varphi_n\rangle \tag{4.31}$$

$$\vdash f_{\overline{d}}\langle\varphi_1,\ldots,\psi_1,\ldots\varphi_n\rangle \supset f_{\overline{d}}\langle\varphi_1,\ldots,\psi_1 \vee \psi_2,\ldots,\varphi_n\rangle \ . \tag{4.32}$$

Las propiedades más interesantes son las relativas a la interacción entre la modalidad espacial y la modalidad de acción:

$$\{\varphi_i \supset [\alpha_i]\psi_i \mid i \in \{1,\ldots,n\}\} \vdash f_{\overline{d}}\langle\varphi_1,\ldots,\varphi_n\rangle \supset [f_{\overline{d}}(\alpha_1,\ldots,\alpha_n)]f_{\overline{d}}\langle\psi_1,\ldots,\psi_n\rangle \tag{4.33}$$

$$\{\varphi_i \supset \langle\alpha_i\rangle\mathit{true} \mid i \in \{1,\ldots,n\}\} \vdash f_{\overline{d}}\langle\varphi_1,\ldots,\varphi_n\rangle \supset \langle f_{\overline{d}}(\alpha_1,\ldots,\alpha_n)\rangle\mathit{true} \tag{4.34}$$

$$\vdash \langle f_{\overline{d}}(\alpha_1,\ldots,\alpha_n)\rangle\mathit{true} \supset f_{\overline{d}}\langle\langle\alpha_1\rangle\mathit{true},\ldots,\langle\alpha_n\rangle\mathit{true}\rangle \tag{4.35}$$

La primera regla relaciona los efectos de una ejecución concurrente con los efectos de las acciones individuales. La segunda refleja el hecho de que una reescritura concurrente puede tener lugar siempre que las acciones que la componen puedan tener lugar. La tercera proporciona un interesante puente entre los operadores espaciales y de acción, reflejando la doble naturaleza de la Σ-estructura actuando al mismo tiempo en estados y en acciones.

Tenemos una versión de (4.33) para la modalidad espacial dual:

$$\{\varphi_i \supset \langle\alpha_i\rangle\psi_i \mid i \in \{1,\ldots,n\}\} \vdash f_{\overline{d}}\langle\varphi_1,\ldots,\varphi_n\rangle \supset \langle f_{\overline{d}}(\alpha_1,\ldots,\alpha_n)\rangle f_{\overline{d}}\langle\psi_1,\ldots,\psi_n\rangle \ . \tag{4.36}$$

Las reglas siguientes se derivan directamente de las reglas de inferencia (4.33) y (4.34)

$$\vdash f_{\overline{d}}\langle[\alpha_1]\psi_1, \ldots, [\alpha_n]\psi_n\rangle \supset [f_{\overline{d}}(\alpha_1, \ldots, \alpha_n)] f_{\overline{d}}\langle\psi_1, \ldots, \psi_n\rangle \quad (4.37)$$
$$\vdash f_{\overline{d}}\langle\langle\alpha_1\rangle true, \ldots, \langle\alpha_n\rangle true\rangle \supset \langle f_{\overline{d}}(\alpha_1, \ldots, \alpha_n)\rangle true \ . \quad (4.38)$$

Como consecuencia tenemos

$$\vdash f_{\overline{d}}\langle\langle\alpha_1\rangle\psi_1, \ldots, \langle\alpha_n\rangle\psi_n\rangle \supset \langle f_{\overline{d}}(\alpha_1, \ldots, \alpha_n)\rangle f_{\overline{d}}\langle\psi_1, \ldots, \psi_n\rangle \ . \quad (4.39)$$

Finalmente, tenemos una regla que establece que si un estado se puede descomponer entonces uno de los subestados verifica *true*.

$$\vdash f_{\overline{d}}[\varphi_1, \ldots, true, \ldots, \varphi_n] \ , \quad (4.40)$$

y tenemos también una regla que refleja la dualidad entre los dos operadores espaciales:

$$\vdash f_{\overline{d}}[\varphi_1, \ldots, \varphi_n] \equiv \neg f_{\overline{d}}\langle\neg\varphi_1, \ldots, \neg\varphi_n\rangle \ . \quad (4.41)$$

Al igual que ocurre con la modalidad de acción, hay formas más fuertes de inevitabilidad que se aplican a propiedades que no incluyen atributos de estado. En la regla siguiente, sean t_i y t'_i términos en $T_{\Sigma^+}(X)$:

$$\vdash f_{\overline{d}}\langle\overline{true}\rangle \supset (t_1 = t'_1 \wedge \ldots \wedge t_n = t'_n \supset f_{\overline{d}}\langle t_1 = t'_1, \ldots, t_n = t'_n\rangle) \ . \quad (4.42)$$

4.4.3. Algunas consideraciones sobre la completitud de la lógica

Una lógica es completa si todas las fórmulas válidas se pueden probar usando los axiomas y reglas de la lógica. En VLRL una fórmula es *válida* en un modelo si se satisface en todos los estados, para cualquier interpretación de las observaciones y para cualquier sustitución básica de las variables.

La lógica VLRL se utiliza para razonar sobre programas concretos, por lo tanto, la completitud de la lógica será relativa a las propiedades válidas que dependen de la estructura del sistema.

Por ejemplo, en la especificación de cadenas de caracteres formadas por dos constantes `a` y `b`, dada en la Sección 4.3.4, la propiedad $first = \texttt{a} \equiv \neg first = \texttt{b}$ es válida. Pero esta propiedad es válida porque el alfabeto utilizado tiene solo dos constantes. Por lo tanto, en principio, la completitud de la lógica que podemos esperar es una completitud *relativa al razonamiento de primer orden*, esto es, está limitada a los razonamientos modales por encima de la lógica de primer orden.

Sin embargo, existen también propiedades espaciales y de acción que dependen del programa que se esté considerando. Estas propiedades tienen que ver principalmente con las condiciones que permiten realizar las acciones así como las condiciones para la descomposición de los estados en subestados. Como veremos en detalle en los ejemplos del Capítulo 5, este tipo de propiedades, tomados como axiomas en demostraciones de propiedades temporales, se pueden dividir en tres grupos.

- Las *propiedades relativas a las transiciones del sistema*, que expresan los cambios que tienen lugar sobre las observaciones cuando la transición asociada a una regla de reescritura tiene lugar.

  Por ejemplo, en la especificación de las cadenas de caracteres la propiedad $[\texttt{ch1}](\mathit{first} = \texttt{b})$ que expresa que, si la regla `ch1` puede tener lugar entonces la cadena resultante empezará por la constante `b`, es válida.

  Estas propiedades pueden obtenerse analizando las propiedades básicas que cumplen las observaciones en el estado representado por la parte derecha de una regla de reescritura.

- Las *propiedades sobre las condiciones que permiten realizar las acciones.*

  Por ejemplo, en la especificación de las cadenas de caracteres la propiedad $\texttt{a}\langle\rangle \supset \langle\texttt{ch1}\rangle\mathit{true}$ que expresa que, en un estado formado por la constante `a` la acción `ch1` puede tener lugar es válida.

  Estas propiedades pueden obtenerse analizando las propiedades básicas que cumplen las observaciones en el estado representado por la parte izquierda de una regla de reescritura.

- Las *propiedades espaciales* que expresan cuándo un estado puede descomponerse en subestados, y las propiedades que se satisfacen en cada subestado.

  Por ejemplo, en la especificación de las cadenas de caracteres la propiedad $\mathit{first} = \texttt{a} \supset __\langle \mathit{first} = \texttt{a}, \mathit{true}\rangle$ es válida.

Además la interacción entre los operadores espacial y de acción es muy general, permitiendo realizar acciones bien en el sistema global o en las partes del sistema.

Por ejemplo, consideremos la especificación anterior de las cadenas de caracteres junto con la regla de reescritura adicional

```
var S : String .
rl [Addb] : aS => abS .
```

Entonces la propiedad

$$\mathit{first} = \texttt{a} \supset \langle\texttt{Addb}\rangle\langle \mathit{first} = \texttt{a}, \mathit{first} = \texttt{b}\rangle$$

es válida, pero no es derivable usando el conjunto de axiomas y reglas del Apéndice A ya que estas reglas no nos permiten relacionar acciones en el sistema global con propiedades de las partes del sistema.

A pesar de que hemos intentado capturar este tipo de propiedades mediante reglas adicionales, hasta el momento no hemos sido capaces de demostrar un resultado de completitud relativa. Este tema queda por lo tanto abierto para trabajo futuro.

4.4.4. Propiedades espaciales básicas

En esta sección se definen métodos para derivar propiedades espaciales de forma automática a partir de la definición de las observaciones y de la sintaxis del programa.

Los métodos se basan en la signatura de las observaciones y en su axiomatización. No se intenta proporcionar métodos que deriven todas las posibles propiedades, sino simplemente aquellas más comunes en los procesos de verificación de propiedades más complejas.

Observaciones de tipo `Nat`. Supongamos que la observación puede definirse como la suma de las observaciones de las partes del sistema

$$\begin{aligned} & f : t_1 \ldots t_m \to \mathit{State} \\ & \mathit{obs} : \mathit{State} \to \mathit{Nat} \\ & \mathit{obs}(f(V_1, \ldots, V_m)) = \mathit{obs}(C_1) + \ldots + \mathit{obs}(Cn) \end{aligned}$$

donde $T_1, \ldots, T_m$ son tipos, $V_1, \ldots, V_m$ son variables y $C_1, \ldots, C_n$ son las variables de tipo *State* de entre $V_1, \ldots, V_m$.
Entonces, tenemos las siguientes reglas:

$$\vdash f_{\overline{d}}\langle \mathit{obs} = N_1, \ldots, \mathit{obs} = N_n \rangle \supset \mathit{obs} = N_1 + \ldots + N_n \tag{4.43}$$

$$\begin{aligned} \vdash\ & f_{\overline{d}}\langle \overline{\mathit{true}} \rangle \supset (\mathit{obs} = N \supset \\ & \exists N_1, \ldots, N_n . (f_{\overline{d}}\langle \mathit{obs} = N_1, \ldots, \mathit{obs} = N_n \rangle \wedge N = N_1 + \ldots + N_n)) \ . \end{aligned} \tag{4.44}$$

Obsérvese que en la primera regla (4.43) el resultado se deriva para todos los números naturales $N_1, \ldots, N_n$, pero en la segunda (4.44) sólo se requiere una posible descomposición.

Por ejemplo, en el caso de la máquina expendedora, tenemos

$$\begin{aligned} & \vdash __\langle \#\$ = N_1, \#\$ = N_2 \rangle \supset \#\$ = N_1 + N_2 \\ & \vdash __\langle \mathit{true}, \mathit{true} \rangle \supset (\#\$ = N \supset \\ & \quad \exists N_1, N_2 . (__\langle \#\$ = N_1, \#\$ = N_2 \rangle \wedge N = N_1 + N_2)) \ . \end{aligned}$$

Las reglas (4.43) y (4.44) son también correctas para los operadores relacionales $\geq$, $>$,$\leq$, y $<$.

Las reglas se pueden generalizar para el caso en que existan varias observaciones de este tipo

$$\begin{aligned} \vdash\ & f_{\overline{d}}\langle \mathit{obs}_1 = N_{1,1} \wedge \ldots \wedge \mathit{obs}_k = N_{1,k}, \ldots, \mathit{obs}_1 = N_{n,1} \wedge \ldots \wedge \mathit{obs}_k = N_{n,k} \rangle \\ & \supset \mathit{obs}_1 = N_{1,1} + \ldots + N_{n,1} \wedge \ldots \wedge \mathit{obs}_k = N_{1,k} + \ldots + N_{n,k} \end{aligned} \tag{4.45}$$

$$\begin{aligned} \vdash\ & f_{\overline{d}}\langle \overline{\mathit{true}} \rangle \supset (\mathit{obs}_1 = N_1 \wedge \ldots \wedge \mathit{obs}_k = N_k \supset \exists N_{1,1}, \ldots, N_{1,k}, \ldots, N_{n,1}, \ldots, N_{n,k} . \\ & (f_{\overline{d}}\langle \mathit{obs}_1 = N_{1,1} \wedge \ldots \wedge \mathit{obs}_k = N_{1,k}, \ldots, \mathit{obs}_1 = N_{n,1} \wedge \ldots \wedge \mathit{obs}_k = N_{n,k} \rangle \wedge \\ & N_1 = N_{1,1} + \ldots + N_{n,1} \wedge \ldots \wedge N_k = N_{1,k} + \ldots + N_{n,k})) \end{aligned} \tag{4.46}$$

Observaciones de tipo `Set` y `Bool`. Si observamos una propiedad sobre un conjunto de elementos

$$\begin{aligned} &obs : State \longrightarrow Set[X] \\ &obs(f(V_1, \ldots, V_m)) = obs(C_1) \cup \ldots \cup obs(C_n) \end{aligned}$$

entonces las reglas son

$$\vdash f_{\overline{d}}\langle obs = S_1, \ldots, obs = S_n\rangle \supset obs = \bigcup_{i=1}^{n} S_i \tag{4.47}$$

$$\begin{aligned} \vdash\ & f_{\overline{d}}\langle\overline{true}\rangle \supset (obs = S \supset \exists S_1, \ldots, S_n. \\ & (S = \bigcup_{i=1}^{n} S_i \wedge f_{\overline{d}}\langle obs = S_1, \ldots, obs = S_n\rangle)\ . \end{aligned} \tag{4.48}$$

Tenemos un resultado similar para las observaciones de tipo lógico:

$$\begin{aligned} &obs : State \longrightarrow Bool \\ &obs(f(V_1, \ldots, V_m)) = obs(C_1) \vee \ldots \vee obs(C_n) \end{aligned}$$

$$\vdash f_{\overline{d}}\langle obs = V_1, \ldots, obs = V_n\rangle \supset obs = V_1 \vee \ldots \vee V_n \tag{4.49}$$

$$\begin{aligned} \vdash\ & f_{\overline{d}}\langle\overline{true}\rangle \supset (obs = V \supset \exists V_1, \ldots, V_n. \\ & (V = V_1 \vee \ldots \vee V_n \wedge f_{\overline{d}}\langle obs = V_1, \ldots, obs = V_n\rangle))\ . \end{aligned} \tag{4.50}$$

Configuraciones con estados vacíos. Si la estructura proporcionada por la función f admite estados vacíos, lo que se define como un axioma estructural en la signatura de la función, entonces el estado siempre puede descomponerse y tenemos:

$$\vdash \varphi \supset f_{\overline{d}}\langle\varphi, true, \ldots, true\rangle\ . \tag{4.51}$$

La implicación contraria puede no ser cierta, como ocurre en el ejemplo de la Sección 5.5.2.

Se comprueba también que (4.51) no es válida si la función no permite estados vacíos. Tómese, por ejemplo, el estado `q q q` formado por tres cuartos de dólar; entonces se tiene que $\#q \geq 3$ se verifica, pero $__\langle \#q \geq 3, true\rangle$ no.

Configuraciones conmutativas y asociativas. Si la estructura proporcionada por la función f tiene la propiedad conmutativa, lo que se define, como en el caso de las estructuras con estados vacíos, mediante un axioma estructural en la signatura de la función, entonces, si el estado se puede descomponer en partes que cumplen unas ciertas propiedades, tambien se podrá descomponer de forma que los subestados cumplan una permutación de las propiedades iniciales. La regla se expresa como:

$$\vdash f\langle\varphi_1, \varphi_2\rangle \equiv f\langle\varphi_2, \varphi_1\rangle\ . \tag{4.52}$$

De forma similar, si la estructura tiene la propiedad asociativa, lo que se detecta así mismo mediante un axioma estructural en la signatura de la función, entonces tenemos

$$\begin{aligned}\vdash f_{\overline{d}}\langle f_{\overline{d}}\langle\psi_1,\ldots,\psi_n\rangle,\varphi_2,\ldots,\varphi_n\rangle &\equiv f_{\overline{d}}\langle\psi_1,f_{\overline{d}}\langle\psi_2,\ldots,\psi_n,\varphi_2\rangle,\ldots,\varphi_n\rangle\\ &\equiv\ldots\equiv f_{\overline{d}}\langle\psi_1,\ldots,\psi_{n-1},f_{\overline{d}}\langle\psi_n,\varphi_2,\ldots,\varphi_n\rangle\rangle\ .\end{aligned} \tag{4.53}$$

Observaciones sobre un elemento único del sistema. Considérense ahora observaciones que identifiquen de forma única un elemento del sistema, de forma que si una propiedad φ relacionada con dicha observación es cierta en un subestado de un estado no puede ser cierta en ningún otro subestado de dicho estado.

Entonces si la propiedad φ relacionada con la observación es válida en un subestado lo será también en el sistema:

$$\vdash f_{\overline{d}}\langle\varphi,\mathit{true},\ldots,\mathit{true}\rangle\supset\varphi\ . \tag{4.54}$$

Se tiene también la seguridad de que una propiedad φ sobre una observación única no se cumple en los subestados en los que el elemento observado no se encuentra:

$$\vdash\varphi\supset f_{\overline{d}}\langle\varphi,\neg\varphi,\ldots,\neg\varphi\rangle\ . \tag{4.55}$$

Cuando tenemos una propiedad única φ que no se cumple en el sistema, entonces tampoco se cumplirá en ninguna de las partes del sistema:

$$\vdash\neg\varphi\supset f_{\overline{d}}\langle\neg\varphi,\ldots,\neg\varphi\rangle\ . \tag{4.56}$$

Si tenemos propiedades únicas diferentes en cada subestado, $\varphi_1,\ldots,\varphi_n$, entonces podemos derivar su conjunción,

$$\vdash f_{\overline{d}}\langle\varphi_1,\ldots,\varphi_n\rangle\supset\varphi_1\wedge\ldots\wedge\varphi_n\ , \tag{4.57}$$

pero el recíproco no siempre se satisface.

Pero estas reglas se diferencian de las anteriores en que no podemos identificar estas observaciones de forma automática a partir de la signatura del programa.

Acciones. Con respecto a las acciones, tenemos que si una transición puede realizarse en un estado, entonces no puede realizarse en ningún otro estado, ya que una transición sólo puede tener lugar en un estado. Por lo tanto, en el caso de que la configuración del sistema no admita estados vacíos podemos derivar:

$$\vdash f_{\overline{d}}\langle\overline{\mathit{true}}\rangle\supset(\langle\alpha\rangle\mathit{true}\supset f_{\overline{d}}\langle\neg\langle\alpha\rangle\mathit{true},\ldots,\neg\langle\alpha\rangle\mathit{true}\rangle)\ . \tag{4.58}$$

Si la estructura admite estados vacíos, entonces la regla no se cumple, ya que el sistema puede descomponerse en él mismo y estados vacíos.

Por otra parte, si una transición puede realizarse en un subestado, no puede realizarse en el sistema completo (en configuraciones sin estados vacíos):

$$\vdash f_{\overline{d}}\langle\mathit{true},\ldots,\langle\alpha\rangle\mathit{true},\ldots,\mathit{true}\rangle\supset\neg\langle\alpha\rangle\mathit{true} \tag{4.59}$$

`obs: State->Nat`	$\vdash f_{\overline{d}}\langle obs = N_1, \ldots, obs = N_n\rangle \supset obs = N_1 + \ldots + N_n$ * $\vdash f_{\overline{d}}\langle\overline{true}\rangle \supset (obs = N \supset \exists N_1, \ldots, N_n.$ $(f_{\overline{d}}\langle obs = N_1, \ldots, obs = N_n\rangle \wedge N = N_1 + \ldots + N_n))$ *
`obs: State->Set[X]`	$\vdash f_{\overline{d}}\langle obs = S_1, \ldots, obs = S_n\rangle \supset obs = \bigcup_{i=1,\ldots,n} S_i$ $\vdash f_{\overline{d}}\langle\overline{true}\rangle \supset (obs = S \supset \exists S_1, \ldots, S_n.$ $(S = \bigcup_{i=1,\ldots,n} S_i \wedge f_{\overline{d}}\langle obs = S_1, \ldots, obs = S_n\rangle))$
`obs: State->Bool`	$\vdash f_{\overline{d}}\langle obs = V_1, \ldots, obs = V_n\rangle \supset obs = V_1 \vee \ldots \vee V_n$ $\vdash f_{\overline{d}}\langle\overline{true}\rangle \supset (obs = V \supset \exists V_1, \ldots, V_n.$ $(V = V_1 \vee \ldots \vee V_n \wedge f_{\overline{d}}\langle obs = V_1, \ldots, obs = V_n\rangle)$
Configuraciones con estados vacíos	$\vdash \varphi \supset f_{\overline{d}}\langle \varphi, true, \ldots, true\rangle$
Conf. conmutativas	$\vdash f_{\overline{d}}\langle \varphi_1, \ldots, \varphi_n\rangle \equiv f_{\overline{d}}\langle perm(\overline{\varphi})\rangle$
Conf. asociativas	$\vdash f_{\overline{d}}\langle f_{\overline{d}}\langle \psi_1, \ldots, \psi_n\rangle, \varphi_2, \ldots, \varphi_n\rangle \equiv f_{\overline{d}}\langle \psi_1, f_{\overline{d}}\langle \psi_2, \ldots, \psi_n, \varphi_2\rangle, \ldots, \varphi_n\rangle$ $\equiv \ldots, \equiv f_{\overline{d}}\langle \psi_1, \ldots, \psi_{n-1}, f_{\overline{d}}\langle \psi_n, \varphi_2, \ldots, \varphi_n\rangle\rangle$
Observaciones de un elemento único	$\vdash f_{\overline{d}}\langle obs, true, \ldots, true\rangle \supset obs$ $\vdash f_{\overline{d}}\langle obs_1, \ldots, obs_n\rangle \supset obs_1 \wedge \ldots \wedge obs_n$ $\vdash obs \supset f_{\overline{d}}\langle obs, \neg obs, \ldots, \neg obs\rangle$ $\vdash \neg obs \supset f_{\overline{d}}\langle \neg obs, \ldots, \neg obs\rangle$
Acciones	$\vdash f_{\overline{d}}\langle\overline{true}\rangle \supset (\langle\alpha\rangle true \supset f_{\overline{d}}\langle \neg\langle\alpha\rangle true, \ldots, \neg\langle\alpha\rangle true\rangle)$** $\vdash f_{\overline{d}}\langle\langle\alpha\rangle true, true, \ldots, true\rangle \supset \neg\langle\alpha\rangle true$ **

* Estos resultados se cumplen también para los operadores relacionales $\geq, >, \leq, <$.
** Solo en configuraciones sin estados vacíos.

Figura 4.1: Resumen de las reglas espaciales básicas

La Figura 4.1 resume todas las posibilidades discutidas en esta sección.

Ejemplos sobre la aplicación de estos métodos se muestran en la Sección 5.5 al demostrar propiedades temporales del sistema Mobile Maude.

4.5. Transiciones *en contexto*

En el ejemplo de la máquina expendedora de la Sección 4.2.2, podemos querer establecer propiedades como las siguientes:

$$\begin{aligned}
&\langle \texttt{buy-c}\rangle true \equiv \#\$ \geq 1,\\
&\langle \texttt{buy-a}\rangle true \equiv \#\$ \geq 1,\\
&\langle \texttt{change}\rangle true \equiv \#q \geq 4,\\
&(\#\$ = M) \supset [\texttt{buy-c}](\#\$ = M - 1),\\
&(\#q = M) \supset [\texttt{buy-c}](\#q = M),
\end{aligned}$$

tratando de caracterizar por medio de algunas fórmulas las condiciones que permiten realizar las acciones y sus efectos en un estado con respecto a las observaciones.

Pero desafortunadamente o bien no se cumplen, como las tres primeras, o bien son de escasa utilidad, como las dos últimas. El problema es que una acción como `buy-c` solo tiene lugar en el estado global con lo cual solo puede ocurrir cuando el estado es exactamente `$`, y por lo tanto la variable M de las dos últimas fórmulas no tiene utilidad porque necesariamente $\#\$ = 1$ y $\#q = 0$. Así pues, una fórmula correcta pero no tan interesante, sería

$$\langle \texttt{buy-c} \rangle \mathit{true} \equiv \#\$ = 1 \wedge \#q = 0 \wedge \#a = 0 \wedge \#c = 0 \ .$$

Una extensión natural del lenguaje modal es considerar nuevas modalidades que simplifiquen el tratamiento de las transiciones identidad. Para ello, diremos que una acción $\beta(\alpha)$ pone a otra acción α en contexto si $\beta(\alpha)$ está construida solo mediante las reglas de identidad y Σ-estructura encima de α.

Dado un estado $[t]$, una interpretación de las observaciones I y una sustitución básica σ:

- $[t], I, \sigma \models [[\alpha]]\varphi$ si y solo si $[t], I, \sigma \models [\alpha]\varphi$ y $[t], I, \sigma \models [\beta(\alpha)]\varphi$, para todas las acciones $\beta(\alpha)$ que ponen α en contexto.

- $[t], I, \sigma \models \langle\langle\alpha\rangle\rangle\varphi$ si y solo si o bien $[t], I, \sigma \models \langle\alpha\rangle\varphi$ o bien existe una acción $\beta(\alpha)$ que pone α en contexto tal que $[t], I, \sigma \models \langle\beta(\alpha)\rangle\varphi$.

Las nuevas modalidades nos permiten formular propiedades sobre partes del sistema en un contexto más amplio sin complicar la notación por la necesidad de hacer explícitas las *transiciones identidad*. La modalidad de acción $[[\alpha]]$ captura las propiedades que se cumplen después de reescribir *cualquier subtérmino* del término que representa el estado. Por ejemplo, la Figura 4.2 representa un sistema que tiene solo una función, f, con tres argumentos de tipo *State*: t_1, t_2 y t_3, y varios argumentos $\overline{w}$ que no tienen tipo *State*. El primer caso representa la propiedad $[\alpha]\varphi$, esto es, dado un estado $[t]$, si la transición α tiene lugar, obtenemos un estado $[t']$ que cumple φ. El segundo caso representa $[f_{\overline{w}}(\alpha, [t_2], [t_3])]\varphi$, esto es, si el estado se descompone utilizando $f_{\overline{w}}(t_1, t_2, t_3)$ y $[\![\alpha]\!]^{I,\sigma} : [t_1] \to [t_1']$, entonces después de que la transición $f_{\overline{w}}(\alpha, [t_2], [t_3])$ tenga lugar, el estado satisface φ. Los casos tercero y cuarto representan los casos para $[\![\alpha]\!]^{I,\sigma} : [t_2] \to [t_2']$ y $[\![\alpha]\!]^{I,\sigma} : [t_3] \to [t_3']$, respectivamente. Si algún subtérmino, t_1, t_2 o t_3, se puede a su vez descomponer, entonces tendríamos que considerar también la aplicación de la acción α a los subtérminos resultantes.

La modalidad de acción $[[\alpha]]$ no requiere que la transición α tenga lugar, pero, si tiene lugar, el estado resultante debe satisfacer la fórmula dada.

La idea de la modalidad de acción $\langle\langle\alpha\rangle\rangle$ es que podemos realizar una acción en contexto si o bien podemos realizar la acción en el estado global o bien podemos realizar la acción en un subestado realizando transiciones identidad en el resto de los subestados en los que se descompone el estado global. En este caso se requiere que una de las transiciones pueda tener lugar. Por ejemplo, la Figura 4.2 representa $\langle\langle\alpha\rangle\rangle\varphi$ cuando al menos una de las cuatro transiciones posibles tiene lugar.

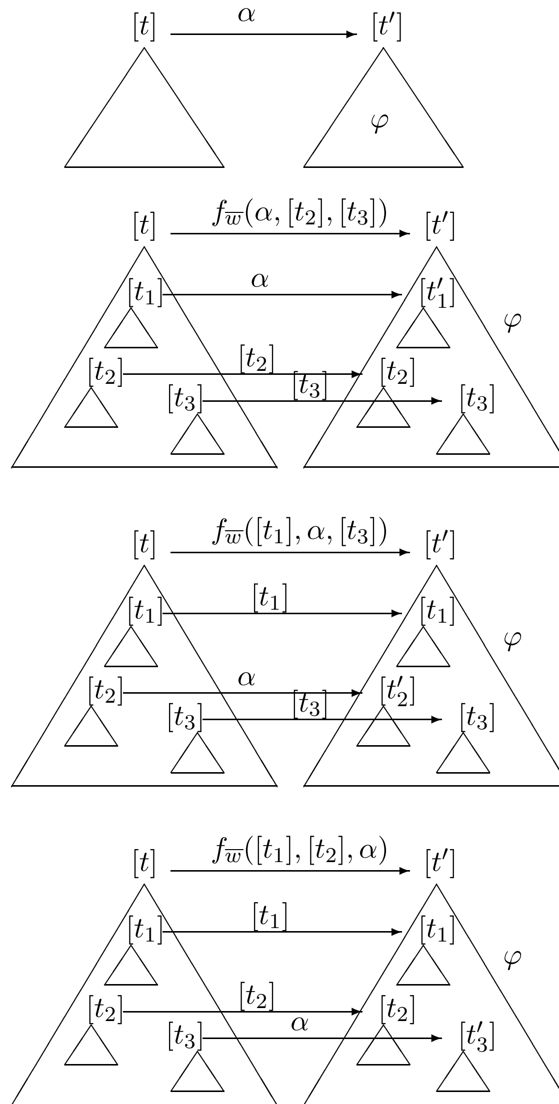

Figura 4.2: Ejemplo de transiciones *en contexto*

Consideramos las siguientes propiedades sobre las nuevas modalidades. Primero, tenemos una versión de (4.12) para las nuevas modalidades:

$$\vdash [[\alpha]]\varphi \wedge \langle\langle\alpha\rangle\rangle \mathit{true} \supset \langle\langle\alpha\rangle\rangle\varphi \ . \tag{4.60}$$

La principal diferencia con (4.12) es que, si reescribimos dentro de un estado, el hecho de que una transición tenga lugar en un subestado dando lugar a una propiedad no garantiza que la transición no pueda tener lugar en un subestado diferente dando lugar a una propiedad diferente. Debido a esto, el recíproco no siempre es cierto. Considérese, por ejemplo, un alfabeto con dos constantes `a` y `b`; dos operaciones para construir cadenas: la constante `nil`, y la operación de concatenación, que se declara asociativa y con identidad `nil`; y reglas para cambiar una constante del alfabeto en la otra.

```
sorts Alpha String .
subsort Alpha < String .
ops a b : -> Alpha .
op nil : String .
op __ : String String -> String [assoc id:nil] .
rl [ch1] : a => b .
rl [ch2] : b => a .
```

Consideramos una observación *first* que devuelve el primer carácter de una cadena. Entonces, por ejemplo, en el estado `abaab`, la propiedad $\langle\langle\texttt{ch1}\rangle\rangle(\mathit{first} = b)$ se satisface, dado que podemos aplicar la regla `ch1` *en contexto* al primer carácter de la cadena. Sin embargo, la propiedad $[[\texttt{ch1}]](\mathit{first} = b)$ no se satisface, ya que podemos aplicar la regla de reescritura a la segunda o tercera `a` de la cadena.

Las siguientes reglas se justifican directamente a partir de la definición:

$$\vdash \langle\alpha\rangle\varphi \supset \langle\langle\alpha\rangle\rangle\varphi \tag{4.61}$$

$$\vdash [[\alpha]]\varphi \supset [\alpha]\varphi \ . \tag{4.62}$$

La siguiente propiedad expresa cuando una acción puede realizarse *en contexto*,

$$\varphi_i \supset \langle\alpha\rangle\psi \vdash f_{\overline{d}}\langle\varphi_1,\ldots,\varphi_n\rangle \supset \langle\langle\alpha\rangle\rangle f_{\overline{d}}\langle\varphi_1,\ldots,\varphi_{i-1},\psi,\varphi_{i+1},\ldots,\varphi_n\rangle$$
$$\text{para cualquier } i \in \{1,\ldots,n\} \ . \tag{4.63}$$

4.6. Especificación de propiedades temporales

Como se ha explicado en el Capítulo 1, consideramos que las especificaciones de los sistemas de reescritura pueden declararse en una lógica diferente a la lógica de reescritura. En este nivel solo tratamos con propiedades abstractas del comportamiento que pueden ser observadas a través de las observaciones, sin necesidad de considerar la existencia de estados específicos o de transiciones específicas (reglas de reescritura).

Se pueden definir diferentes lógicas dependiendo de la naturaleza de las propiedades sobre las que se quiere razonar. Para cada lógica, debe darse una interfaz para soportar

la verificación de los programas sobre las especificaciones. Esta interfaz toma la forma de una colección de reglas de inferencia que permite inferir propiedades dadas en la lógica de especificación a partir de propiedades de los programas dadas en VLRL. La corrección del sistema de inferencia de la interfaz para una lógica de especificación dada se basa en una proyección que abstrae los modelos (comportamientos) de la lógica de especificación a partir de los programas.

Para verificar que un programa P (de hecho, un programa observado, en el sentido de que el programa se toma sobre una extensión de la signatura de reescritura original que define cómo se interpretan los atributos, tal como se explica en la Sección 4.2.4) satisface una especificación S, uno debe encontrar un conjunto de propiedades V en VLRL que se satisfacen en P y del cual se deriva S. Idealmente, podriamos ser capaces de generar V de forma sistemática a partir de P, de forma que la verificación se puede reducir a la prueba de que de V se deriva S.

4.6.1. Lógica temporal sobre VLRL

Para ilustrar estas ideas, utilizaremos una lógica temporal de tiempo ramificado para escribir las especificaciones[6]. Se proporcionan tres operadores lógicos con el siguiente significado:

- $\mathrm{AX}\varphi$: la propiedad φ se cumple en todos los posibles estados sucesores,
- $\mathrm{A}(\varphi \mathrm{W} \psi)$: en cualquier cómputo, φ se cumplirá hasta que se cumpla ψ (*weak until*),
- $\mathrm{EX}\varphi$: la propiedad φ se cumple en algún estado sucesor.

Obsérvese que $\mathrm{AG}\varphi$ (la propiedad φ se cumple en todos los posibles estados futuros) es equivalente a $\mathrm{A}(\varphi \mathrm{W} \mathit{false})$.

Dado que estamos interesados en fórmulas temporales que expresen propiedades que se cumplen en estados que pueden ser alcanzados desde estados que cumplen algunas *buenas* propiedades, en los ejemplos siguientes usamos fórmulas temporales de la forma

$$\Psi ::= \varphi \supset \mathrm{AX}\psi \mid \varphi \supset \mathrm{A}(\varphi \mathrm{W} \psi) \mid \varphi \supset \mathrm{EX}\psi \ ,$$

donde φ y ψ son fórmulas modales en VLRL.

Los modelos de la lógica temporal son los árboles de computación generados por el sistema de transiciones asociado con el modelo inicial del programa. Una *rama* del árbol es una secuencia infinita de estados $[t_0], [t_1], \ldots$ donde para cada par de estados $[t_i]$ y $[t_{i+1}]$ existe alguna acción α_i tal que $\alpha_i : [t_i] \to [t_{i+1}]$.

La satisfacción de una fórmula en un estado dado se define de la forma usual:

- $[t], I, \sigma \models \varphi \supset \mathrm{AX}\psi$ si y solo si $[t], I, \sigma \models \varphi$ implica que para toda acción α, si $[\![\alpha]\!]^{I,\sigma} : [t] \to [t']$ entonces $[t'], I, \sigma \models \psi$,

[6] Remitimos al lector a [Gol92, MP92, Sti92], entre muchos otros textos, para más información sobre lógicas temporales.

- $[t], I, \sigma \models \varphi \supset \mathrm{A}(\varphi \mathrm{W} \psi)$ si y solo si $[t], I, \sigma \models \varphi$ implica que para toda *rama* $[t_0], [t_1], \ldots$ con $[t] = [t_0]$, o bien $[t_i], I, \sigma \models \varphi$ para todo i, o bien existe algún $k \geq 0$ tal que $[t_k], I, \sigma \models \psi$ y $[t_j], I, \sigma \models \varphi$ para todo $0 \leq j < k$,

- $[t], I, \sigma \models \varphi \supset \mathrm{EX}\psi$ si y solo si $[t], I, \sigma \models \varphi$ implica que para alguna acción α, $[\![\alpha]\!]^{I,\sigma} : [t] \to [t']$ y $[t'], I, \sigma \models \psi$.

Para definir la interfaz de VLRL con esta lógica temporal usamos el hecho de que los modelos del programa y de la especificación se construyen sobre las mismas estructuras semánticas. Las reglas de inferencia de la interfaz son:

$$\varphi \supset \psi,\ \{\varphi \supset [\alpha]\psi \mid \alpha \text{ acción}\} \vdash \varphi \supset \mathrm{AX}\psi \tag{4.64}$$

$$\{\varphi \wedge \neg\psi \supset [\alpha](\varphi \vee \psi) \mid \alpha \text{ acción}\} \vdash \varphi \supset \mathrm{A}(\varphi \mathrm{W} \psi) \tag{4.65}$$

$$\varphi \supset \langle\alpha\rangle\psi \vdash \varphi \supset \mathrm{EX}\psi \quad \text{donde } \alpha \text{ es una acción.} \tag{4.66}$$

La primera regla dice que si φ implica ψ y φ implica que en el estado que resulta después de hacer cualquier acción α se satisface ψ, entonces para cualquier estado en el que se satisfaga φ, el estado siguiente en cualquier cómputo satisfará ψ. Las otras reglas se explican de forma similar.

Obsérvese que el lado izquierdo de estas reglas está formado por fórmulas de la lógica de acción modal VLRL, mientras que las del lado derecho son fórmulas temporales. Obsérvese también que, como α puede ser de la forma $\beta(\alpha')$ donde β pone a α' en contexto, por la definición de la modalidad $\langle\langle\alpha'\rangle\rangle$, (4.66) se puede expresar también como:

$$\varphi \supset \langle\langle\alpha\rangle\rangle\psi \vdash \varphi \supset \mathrm{EX}\psi \ . \tag{4.67}$$

La corrección de las reglas de inferencia es inmediata porque, como se ha hecho notar anteriormente, usamos las mismas estructuras de Kripke como modelos para VLRL y para la lógica temporal.

Uso de las reglas de inferencia con conjuntos de acciones infinitos. ¿Cómo podemos demostrar el lado izquierdo de cualquiera de las dos primeras reglas de inferencia de la interfaz anterior cuando el conjunto de acciones es infinito? Sea $P(\alpha)$ la propiedad que hay que probar sobre α; $P(\alpha) = (\varphi \supset [\alpha]\psi)$ en el primer caso, y $P(\alpha) = (\varphi \wedge \neg\psi \supset [\alpha](\varphi \vee \psi))$ en el segundo. Proponemos utilizar inducción estructural sobre las acciones de la forma siguiente:

- para cada $f : s_1 \ldots s_m \to \mathit{State}$ en Σ, con $\mathit{State} \notin \{s_1, \ldots, s_m\}$, probar $f_{\overline{x}}\langle\rangle \supset P([f_{\overline{x}}()])$;

- para cada etiqueta r, probar $P(\alpha(r))$ para cada *mínimo* $\alpha(r)$ que pone a r en contexto, como se explica a continuación;

- para cada $f : s_1 \ldots s_m \to \mathit{State}$ en Σ, con $\mathit{State} \in \{s_1, \ldots, s_m\}$, probar

$$\{P(\alpha_{i_j}) \mid j \in \{1, \ldots, k\}\} \vdash f_{\overline{x}}\langle\overline{true}\rangle \supset P(f_{\overline{x}}(\overline{\alpha})),$$

  donde α_{i_j} son acciones que se corresponden una a una con los argumentos de f de tipo *State*.

Una acción pone una etiqueta r en contexto si contiene $r(\overline{x})$ como un subtérmino. La idea es que, cuando r no reescribe estados pero reescribe dentro de ellos, tenemos que considerar los posibles contextos en los que se puede aplicar la regla de reescritura. Sin embargo, como puede haber un número infinito de contextos, queremos restringir la prueba a los contextos mínimos en el sentido de que la información que se pone en el contexto sea lo menor posible. Por ejemplo, si $f : s_1 s_2 s_3 \to \mathit{State}$ y r es una regla que reescribe términos de tipo s_2, un $\alpha(r)$ mínimo es $f(x, r, y)$ donde x y y son variables nuevas de tipo s_1 y s_3, respectivamente.

En el ejemplo de la máquina expendedora así como en todos los que se consideran en el Capítulo 5 todas las reglas reescriben estados, y en este caso un $\alpha(r)$ mínimo para r es el mismo r. De hecho, como las reglas de reescritura se aplican a estados, no es necesario considerar contextos más amplios.

4.6.2. Completitud de las reglas de inferencia de la interfaz de la lógica temporal con VLRL

Como se indicaba en la sección anterior, los modelos de la lógica temporal son los árboles de computación generados por el sistema de transiciones asociado con el modelo inicial del programa y una *rama* del árbol es una secuencia infinita de estados $[t_0], [t_1], \ldots$ donde para cada par de estados $[t_i]$ y $[t_{i+1}]$ existe alguna acción α_i tal que $\alpha_i : [t_i] \to [t_{i+1}]$.

Para probar la completitud de las reglas nos centramos en el razonamiento temporal siguiendo la aproximación dada en [MP95, página 221]. Asumimos que la parte temporal de las reglas es válida y mostraremos que la respectiva parte modal es también válida.

Completitud de la regla AX

La regla de inferencia AX de la interfaz es:

$$\varphi \supset \psi,\ \{\varphi \supset [\alpha]\psi \mid \alpha \text{ acción}\} \vdash \varphi \supset \mathrm{AX}\psi\ \ .$$

Debemos probar que si $[t], I, \sigma \models \varphi \supset \mathrm{AX}\psi$ entonces $[t], I, \sigma \models \varphi \supset \psi$ y $[t], I, \sigma \models \varphi \supset [\alpha]\psi$ para toda acción α.

Si $[t], I, \sigma \models \varphi \supset \mathrm{AX}\psi$ entonces, por la definición de la relación de satisfacción de los operadores de la lógica temporal, $[t], I, \sigma \models \varphi$ implica que para toda acción α, si $[\![\alpha]\!]^{I,\sigma} : [t] \to [t']$ entonces $[t'], I, \sigma \models \psi$.

Tomamos en primer lugar $\alpha = [t]$, la transición identidad relacionada con el estado $[t]$. Al ser α la transicion identidad, el estado $[t]$ coincide con el estado $[t']$ y por lo tanto $[t], I, \sigma \models \varphi$ implica $[t], I, \sigma \models \psi$. Entonces por la definición de la relación de satisfacción para el operador de implicación $[t], I, \sigma \models \varphi \supset \psi$.

En segundo lugar, por la definición de la relación de satisfacción de la modalidad de acción $[t], I, \sigma \models \varphi$ implica que $[t], I, \sigma \models [\alpha]\psi$. Por lo tanto, por la definición de la relación de satisfacción del operador de implicación, $[t], I, \sigma \models \varphi \supset [\alpha]\psi$.

Completitud de la regla AW

La regla de inferencia AW de la interfaz viene dada por:

$$\{\varphi \wedge \neg\psi \supset [\alpha](\varphi \vee \psi) \mid \alpha \text{ acción}\} \vdash \varphi \supset \mathrm{A}(\varphi \mathrm{W} \psi) \ .$$

Debemos probar que si $\varphi \supset \mathrm{A}(\varphi \mathrm{W} \psi)$ es válida, entonces $\varphi \wedge \neg\psi \supset [\alpha](\varphi \vee \psi)$ es también válida para toda acción α.

Supongamos $\varphi \supset \mathrm{A}(\varphi \mathrm{W} \psi)$ es válida. Por la definición de la relación de satisfacción, $[t], I, \sigma \models \varphi$ implica que para toda rama $[t_0], [t_1], \ldots$ con $[t] = [t_0]$, o bien $[t_i], I, \sigma \models \varphi$ para todo i, o bien existe algún $k \geq 0$ tal que $[t_k], I, \sigma \models \psi$ y $[t_j], I, \sigma \models \varphi$ para todo $0 \leq j < k$.

Debemos probar que $\varphi \wedge \neg\psi \supset [\alpha](\varphi \vee \psi)$ es válida para toda acción α. Supongamos $[t], I, \sigma \models \varphi \wedge \neg\psi$ entonces:

- En el caso de que no exista $\alpha : [t] \to [t']$, por la definición de la relación de satisfacción de la modalidad de acción se obtiene directamente $[t], I, \sigma \models [\alpha](\varphi \vee \psi)$.

- En el caso de que exista $\alpha : [t] \to [t']$, hay una rama $[t], [t'], \ldots$ entonces:
  - Si $[t_i], I, \sigma \models \varphi$ para todo i, en particular $[t'], I, \sigma \models \varphi$. Por la definición de la relación de satisfacción de los operadores de la lógica proposicional $[t'], I, \sigma \models \varphi \vee \psi$. Aplicando la definición de la relación de satisfacción para la modalidad de acción tenemos que $[t], I, \sigma \models [\alpha](\varphi \vee \psi)$.
  - Si existe algún $k \geq 0$ tal que $[t_k], I, \sigma \models \psi$ y $[t_j], I, \sigma \models \varphi$ para todo $0 \leq j < k$. Entonces:
    - $k \neq 0$ ya que por hipótesis $[t], I, \sigma \models \neg\psi$.
    - Si $k = 1$ entonces $[t'], I, \sigma \models \psi$ y por la definición de la relación de satisfacción de la lógica proposicional $[t'], I, \sigma \models \varphi \vee \psi$. Por la definición de la relación de satisfacción de la modalidad de acción $[t], I, \sigma \models [\alpha](\varphi \vee \psi)$.
    - Si $k > 1$ entonces $[t'], I, \sigma \models \varphi$ y por la definición de la relación de satisfacción de la lógica proposicional $[t'], I, \sigma \models \varphi \vee \psi$. Por la definición de la relación de satisfacción de la modalidad de acción $[t], I, \sigma \models [\alpha](\varphi \vee \psi)$.

Completitud de la regla EX

La regla de inferencia EX de la interfaz está dada por:

$$\varphi \supset \langle\alpha\rangle\psi \vdash \varphi \supset \mathrm{EX}\psi \quad \text{donde } \alpha \text{ es una acción.}$$

Debemos probar que si la fórmula $\varphi \supset \mathrm{EX}\psi$ es válida entonces existe una acción α tal que $\varphi \supset \langle\alpha\rangle\psi$.

Por la definición de la relación de satisfacción para el operador temporal EX, tenemos que $[t], I, \sigma \models \varphi$ implica que para alguna acción α, $[\![\alpha]\!]^{I,\sigma} : [t] \to [t']$ y $[t'], I, \sigma \models \psi$. Por la definición de la relación de satisfacción de la modalidad de acción, $[t], I, \sigma \models \varphi$ implica $[t], I, \sigma \models \langle\alpha\rangle\psi$. Por tanto, $[t], I, \sigma \models \varphi \supset \langle\alpha\rangle\psi$.

4.6.3. Ejemplo: Propiedades temporales de la máquina expendedora

Vamos a ilustrar la aplicación de las reglas de inferencia de la lógica temporal con la lógica VLRL mediante la prueba de propiedades temporales del programa `VENDING-MACHINE` para una máquina expendedora (Sección 4.2.2).

Primero, podemos obtener fácilmente las siguientes 12 fórmulas considerando sistemáticamente la forma en que cada una de las 3 reglas de reescritura modifica el valor de cada una de las 4 observaciones básicas. Estas propiedades se derivan de la axiomatización de las interpretaciones de las observaciones que deben proporcionarse junto al programa para formar un programa observado. Estamos investigando la posibilidad de derivar automáticamente estas propiedades a partir del programa. En nuestro ejemplo, la axiomatización `VENDING-MACHINE-OBSERVED` se dió en la Sección 4.2.2.

$$(\#\$ = M) \supset [[\texttt{buy-c}]](\#\$ = M - 1)$$
$$(\#q = M) \supset [[\texttt{buy-c}]](\#q = M)$$
$$(\#c = M) \supset [[\texttt{buy-c}]](\#c = M + 1) \tag{4.68}$$
$$(\#a = M) \supset [[\texttt{buy-c}]](\#a = M) \tag{4.69}$$
$$(\#\$ = M) \supset [[\texttt{buy-a}]](\#\$ = M - 1)$$
$$(\#q = M) \supset [[\texttt{buy-a}]](\#q = M + 1)$$
$$(\#c = M) \supset [[\texttt{buy-a}]](\#c = M) \tag{4.70}$$
$$(\#a = M) \supset [[\texttt{buy-a}]](\#a = M + 1) \tag{4.71}$$
$$(\#\$ = M) \supset [[\texttt{change}]](\#\$ = M + 1)$$
$$(\#q = M) \supset [[\texttt{change}]](\#q = M - 4)$$
$$(\#c = M) \supset [[\texttt{change}]](\#c = M)$$
$$(\#a = M) \supset [[\texttt{change}]](\#a = M) \ .$$

Las siguientes fórmulas caracterizan las condiciones que permiten realizar las acciones en el programa `VENDING-MACHINE`.

$$\langle \texttt{buy-c} \rangle \mathit{true} \equiv \texttt{\$}\langle\rangle \tag{4.72}$$
$$\langle \texttt{buy-a} \rangle \mathit{true} \equiv \texttt{\$}\langle\rangle \tag{4.73}$$
$$\langle \texttt{change} \rangle \mathit{true} \equiv \texttt{__}\langle \texttt{q q q q} \rangle$$
$$\langle\langle \texttt{buy-c} \rangle\rangle \mathit{true} \equiv \#\$ \geq 1 \tag{4.74}$$
$$\langle\langle \texttt{buy-a} \rangle\rangle \mathit{true} \equiv \#\$ \geq 1 \tag{4.75}$$
$$\langle\langle \texttt{change} \rangle\rangle \mathit{true} \equiv \#q \geq 4$$

donde

$$\begin{aligned} \texttt{\$}\langle\rangle &\equiv \#\$ = 1 \wedge \#a = 0 \wedge \#c = 0 \wedge \#q = 0 \\ \texttt{__}\langle \texttt{q q q q} \rangle &\equiv \#\$ = 0 \wedge \#a = 0 \wedge \#c = 0 \wedge \#q = 4 \ . \end{aligned}$$

Ahora probamos algunas propiedades temporales sobre la máquina expendedora.

Comprando un bizcocho

Si el número de dólares es mayor que 0, entonces es posible comprar un bizcocho:

$$(\#\$ > 0 \wedge \#c = N) \supset \mathrm{EX}(\#c = N + 1) \ .$$

Usando la regla de inferencia de la interfaz (4.67), necesitamos probar que para alguna acción α

$$(\#\$ > 0 \wedge \#c = N) \supset \langle\langle \alpha \rangle\rangle (\#c = N + 1) \ .$$

Uniendo las fórmulas (4.68) y (4.74), obtenemos

$$(\#c = N \wedge \#\$ > 0) \supset [[\texttt{buy-c}]](\#c = N + 1) \wedge \langle\langle \texttt{buy-c} \rangle\rangle \mathit{true} \ .$$

Aplicando (4.60) al consecuente anterior obtenemos el resultado:

$$(\#c = N \wedge \#\$ > 0) \supset \langle\langle \texttt{buy-c} \rangle\rangle (\#c = N + 1) \ .$$

El valor total es un invariante

El valor total de la máquina es un *invariante*:

$$\mathit{wealth} = N \supset \mathrm{AG}(\mathit{wealth} = N) \ .$$

Por la definición de la conectiva temporal AG, $\mathrm{AG}\varphi \equiv \mathrm{A}(\varphi \mathrm{W} \mathit{false})$, y aplicando la regla de inferencia de la interfaz de la lógica temporal y VLRL (4.65), es suficiente con probar

$$\{\mathit{wealth} = N \supset [\alpha](\mathit{wealth} = N) \mid \alpha \text{ acción}\} \ .$$

Usando inducción estructural (ver Sección 4.6.1), tenemos que probar:

- constantes:
  1. $\texttt{\$}\langle\rangle \supset (\mathit{wealth} = N \supset [[\texttt{\$}]](\mathit{wealth} = N))$
  2. $\texttt{q}\langle\rangle \supset (\mathit{wealth} = N \supset [[\texttt{q}]](\mathit{wealth} = N))$
  3. $\texttt{c}\langle\rangle \supset (\mathit{wealth} = N \supset [[\texttt{c}]](\mathit{wealth} = N))$
  4. $\texttt{a}\langle\rangle \supset (\mathit{wealth} = N \supset [[\texttt{a}]](\mathit{wealth} = N))$
- reglas:
  5. $\mathit{wealth} = N \supset [\texttt{buy-c}](\mathit{wealth} = N)$
  6. $\mathit{wealth} = N \supset [\texttt{buy-a}](\mathit{wealth} = N)$
  7. $\mathit{wealth} = N \supset [\texttt{change}](\mathit{wealth} = N)$
- operaciones sobre estados (en este caso, solo $__$):

8. $(\mathit{wealth} = N_1 \supset [\alpha_1](\mathit{wealth} = N_1)),$
   $(\mathit{wealth} = N_2 \supset [\alpha_2](\mathit{wealth} = N_2))$
   $\vdash __\langle \mathit{true}, \mathit{true}\rangle \supset (\mathit{wealth} = N \supset [__(\alpha_1, \alpha_2)](\mathit{wealth} = N)) \quad .$

La demostración de las siete primeras propiedades es inmediata. Probamos la última propiedad de la siguiente forma. Asúmanse las hipótesis; usando la regla de inferencia de VLRL (4.33), obtenemos

$$__\langle \mathit{wealth} = N_1, \mathit{wealth} = N_2\rangle \supset [__(\alpha_1, \alpha_2)]__\langle \mathit{wealth} = N_1, \mathit{wealth} = N_2\rangle \ .$$

A partir de aquí y con la siguiente propiedad de la observación *wealth* derivada de aplicar la regla (4.43) para obtener propiedades espaciales básicas de observaciones de tipo `Nat` (Sección 4.4.4)

$$\begin{aligned}&__\langle \mathit{true}, \mathit{true}\rangle \supset (\mathit{wealth} = N \supset \\ &\quad \exists N_1, N_2 . __\langle \mathit{wealth} = N_1, \mathit{wealth} = N_2\rangle \wedge N = N_1 + N_2) \ ,\end{aligned}$$

obtenemos

$$\begin{aligned}&__\langle \mathit{true}, \mathit{true}\rangle \supset (\mathit{wealth} = N \supset \\ &\quad \exists N_1, N_2 . [__(\alpha_1, \alpha_2)]__\langle \mathit{wealth} = N_1, \mathit{wealth} = N_2\rangle \wedge N = N_1 + N_2) \ .\end{aligned}$$

Dado que N, N_1, N_2 no son variables de tipo estado, podemos aplicar (4.15):

$$\begin{aligned}&__\langle \mathit{true}, \mathit{true}\rangle \supset (\mathit{wealth} = N \supset \\ &\quad \exists N_1, N_2 . [__(\alpha_1, \alpha_2)]__\langle \mathit{wealth} = N_1, \mathit{wealth} = N_2\rangle \\ &\qquad\qquad \wedge [__(\alpha_1, \alpha_2)](N = N_1 + N_2)) \ .\end{aligned}$$

Aplicando ahora (4.26) y (4.16), obtenemos

$$\begin{aligned}&__\langle \mathit{true}, \mathit{true}\rangle \supset (\mathit{wealth} = N \supset \\ &[__(\alpha_1, \alpha_2)]\exists N_1, N_2 . __\langle \mathit{wealth} = N_1, \mathit{wealth} = N_2\rangle \wedge (N = N_1 + N_2)) \ .\end{aligned}$$

Finalmente, usando la propiedad de los atributos

$$__\langle \mathit{wealth} = N_1, \mathit{wealth} = N_2\rangle \wedge N = N_1 + N_2 \supset \mathit{wealth} = N \ ,$$

derivada de aplicar la regla (4.44) para obtener propiedades espaciales básicas de observaciones de tipo `Nat`, llegamos a la conclusión deseada

$$__\langle \mathit{true}, \mathit{true}\rangle \supset (\mathit{wealth} = N \supset [__(\alpha_1, \alpha_2)](\mathit{wealth} = N)) \ .$$

Comprando un bizcocho y una manzana

En un estado con al menos dos dólares, es posible comprar concurrentemente un bizcocho y una manzana:

$$(\#\$ \geq 2 \wedge \#c = M \wedge \#a = N) \supset \mathrm{EX}(\#c = M + 1 \wedge \#a = N + 1) \ .$$

Usamos la regla de inferencia de la interfaz (4.67), y probamos

$$(\#\$ \geq 2 \wedge \#c = M \wedge \#a = N) \supset \langle\langle\alpha\rangle\rangle(\#c = M + 1 \wedge \#a = N + 1)$$

para alguna acción α.

Por aritmética, obtenemos

$$\begin{array}{l}(\#\$ \geq 2 \wedge \#c = M \wedge \#a = N \wedge \#q = S) \supset \\ \quad (\exists K.\#\$ = 2 + K \wedge K \geq 0 \wedge \#c = M \wedge \#a = N \wedge \#q = S)\end{array} \tag{4.76}$$

Ahora distinguimos dos casos.

Caso (i) El estado está compuesto solo por dos dólares,

$$K = 0 \wedge M = 0 \wedge N = 0 \wedge S = 0 \ .$$

Aplicando a la fórmula (4.76) la propiedad estructural derivada mediante la regla (4.46) para obtener propiedades espaciales básicas, que afirma que un estado con solo dos dólares se puede descomponer en dos subestados no vacíos con un dólar en cada uno de ellos, obtenemos

$$\begin{array}{l}(\#\$ = 2 \wedge \#c = 0 \wedge \#a = 0 \wedge \#q = 0) \supset \\ \quad __\langle \#\$ = 1 \wedge \#c = 0 \wedge \#a = 0 \wedge \#q = 0, \\ \qquad \#\$ = 1 \wedge \#c = 0 \wedge \#a = 0 \wedge \#q = 0\rangle \ .\end{array} \tag{4.77}$$

Las reglas de inferencia de VLRL (4.33) y (4.62) aplicadas a las propiedades de las observaciones (4.68), (4.69), (4.70) y (4.71) nos dan

$$\begin{array}{l}__\langle \#c = 0 \wedge \#a = 0, \#c = 0 \wedge \#a = 0\rangle \supset \\ \quad [__(\mathtt{buy\text{-}c}, \mathtt{buy\text{-}a})]__\langle \#c = 1 \wedge \#a = 0, \#c = 0 \wedge \#a = 1\rangle \ .\end{array} \tag{4.78}$$

Por otra parte, la regla de inferencia de VLRL (4.34) aplicada a las propiedades de las observaciones (4.72) y (4.73) nos da

$$\begin{array}{l}__\langle \#\$ = 1 \wedge \#c = 0 \wedge \#a = 0 \wedge \#q = 0, \\ \quad \#\$ = 1 \wedge \#c = 0 \wedge \#a = 0 \wedge \#q = 0\rangle \supset \\ \qquad \langle __(\mathtt{buy\text{-}c}, \mathtt{buy\text{-}a})\rangle \mathit{true} \ .\end{array} \tag{4.79}$$

Uniendo las fórmulas (4.78) y (4.79), y usando la equivalencia (4.12), obtenemos

$$\begin{array}{l}__\langle \#\$ = 1 \wedge \#c = 0 \wedge \#a = 0 \wedge \#q = 0, \\ \quad \#\$ = 1 \wedge \#c = 0 \wedge \#a = 0 \wedge \#q = 0\rangle \supset \\ \qquad \langle __(\mathtt{buy\text{-}c}, \mathtt{buy\text{-}a})\rangle __\langle \#c = 1 \wedge \#a = 0, \#c = 0 \wedge \#a = 1\rangle \ .\end{array} \tag{4.80}$$

Finalmente, encadenando las fórmulas (4.77) y (4.80), y la siguiente propiedad estructural de las observaciones derivada mediante la regla (4.45) para obtener propiedades espaciales básicas

$$__\langle \#c = M \wedge \#a = 0, \#c = 0 \wedge \#a = N\rangle \supset \#c = M \wedge \#a = N \ ,$$

obtenemos

$$\begin{aligned}(\#\$ = 2 \wedge \#c = 0 \wedge \#a = 0 \wedge \#q = 0) \supset \\ \langle __(\texttt{buy-c}, \texttt{buy-a})\rangle(\#c = 1 \wedge \#a = 1) \ ,\end{aligned} \tag{4.81}$$

que nos proporciona la conclusión deseada para el caso (i), después de aplicar (4.61), con la acción $\alpha = __(\texttt{buy-c}, \texttt{buy-a})$.

Caso (ii) El estado no está compuesto únicamente por dos dólares,

$$K > 0 \vee M > 0 \vee N > 0 \vee S > 0 \ .$$

Aplicando a la fórmula (4.76) la propiedad estructural

$$\begin{aligned}&(\#\$ = N_1 + N_2 \wedge \#c = M \wedge \#a = N \wedge \#q = S \\ &\quad \wedge N_1 > 0 \wedge (N_2 > 0 \vee M > 0 \vee N > 0 \vee S > 0)) \supset \\ &\qquad __\langle \#\$ = N_1 \wedge \#c = 0 \wedge \#a = 0 \wedge \#q = 0, \\ &\qquad\quad \#\$ = N_2 \wedge \#c = M \wedge \#a = N \wedge \#q = S\rangle\end{aligned}$$

que afirma que un estado con suficientes dólares se puede descomponer en dos subestados no vacíos con algunos dólares en cada uno de ellos, y todas las manzanas, bizcochos y los cuartos en uno de los subestados, obtenemos

$$\begin{aligned}&\exists K.(\#\$ = 2 + K \wedge K \geq 0 \wedge \#c = M \wedge \#a = N \wedge \#q = S) \supset \\ &\quad \exists K.__\langle \#\$ = 2 \wedge \#c = 0 \wedge \#a = 0 \wedge \#q = 0, \\ &\qquad \#\$ = K \wedge \#c = M \wedge \#a = N \wedge \#q = S\rangle \ .\end{aligned} \tag{4.82}$$

Ahora aplicamos (4.63) a (4.81), y lo encadenamos con (4.82) para obtener

$$\begin{aligned}&\exists K.(\#\$ = 2 + K \wedge K \geq 0 \wedge \#c = M \wedge \#a = N \wedge \#q = S) \supset \\ &\quad \exists K.\langle\langle __(\texttt{buy-c}, \texttt{buy-a})\rangle\rangle __\langle \#c = 1 \wedge \#a = 1, \\ &\qquad\qquad \#\$ = K \wedge \#c = M \wedge \#a = N \wedge \#q = S\rangle \ .\end{aligned} \tag{4.83}$$

Aplicamos la siguiente propiedad estructural

$$\begin{aligned}&__\langle \#c = 1 \wedge \#a = 1, \#\$ = K \wedge \#c = M \wedge \#a = N \wedge \#q = S\rangle \supset \\ &\quad \#c = M + 1 \wedge \#a = N + 1\end{aligned}$$

al consecuente de (4.83), obteniendo

$$\begin{aligned}&\exists K.(\#\$ = 2 + K \wedge K \geq 0 \wedge \#c = M \wedge \#a = N \wedge \#q = S) \supset \\ &\quad \exists K.\langle\langle __(\texttt{buy-c}, \texttt{buy-a})\rangle\rangle(\#c = M + 1 \wedge \#a = N + 1) \ .\end{aligned}$$

Encadenando (4.76) con la fórmula anterior y después con la implicación

$$\begin{aligned}\exists K.\langle\langle __(\texttt{buy-c}, \texttt{buy-a})\rangle\rangle(\#c = M+1 \wedge \#a = N+1) \supset \\ \langle\langle __(\texttt{buy-c}, \texttt{buy-a})\rangle\rangle(\#c = M+1 \wedge \#a = N+1)\ .\end{aligned}$$

derivamos el resultado esperado para el caso (ii).

4.7. Conclusiones

La lógica de verificación para la lógica de reescritura nos permite verificar programas, esto es sistemas de software, contra sus especificaciones en un nivel más abstracto. Los principales beneficios del uso de VLRL son:

1. Las propiedades que constituyen la especificación se definen en una lógica más abstracta sobre la lógica ejecutable de los programas.

2. Mediante el uso de reglas de interfaz con otras lógicas modales y temporales permite la demostración de propiedades expresadas en estas lógicas a partir de la lógica VLRL. De esta forma facilita la especificación del sistema al permitir formular las propiedades en la lógica más apropiada.

3. Se integra bien con la lógica de reescritura, que es la lógica usada para la ejecución concurrente.

4. Se aplica a las teorías de reescritura en toda su generalidad, acomodándose por lo tanto a muchos tipos diferentes de sistemas concurrentes.

5. Se adapta bien a aplicaciones de programación prácticas en términos de poder expresivo y facilidad de demostración, y soporta razonar sobre estas aplicaciones a un alto nivel de abstracción.

El operador espacial es una de las principales innovaciones de la lógica VLRL y resulta muy útil en la especificación de sistemas concurrentes sobre todo cuando se combina con el operador de acción.

Capítulo 5

Especificación de sistemas orientados a objetos en VLRL

Las configuraciones orientadas a objetos son unas de las más utilizadas en el actual modelado de los sistemas. En este capítulo se utilizan la lógica VLRL y la lógica temporal introducidas en el capítulo anterior para definir y verificar propiedades de diversos sistemas orientados a objetos.

El lenguaje Maude proporciona muchas facilidades para la especificación de sistemas orientados a objetos, como la posibilidad de definir clases, mensajes, y la relación de herencia entre las clases (ver Sección 2.2.3). En el ejemplo sobre la especificación de un modelo de red de telecomunicaciones desarrollado en el Capítulo 3 se muestra la definición de varias clases y mensajes, así como la implementación de los métodos asociados a los mensajes mediante reglas de reescritura. El lenguaje permite definir un objeto como valor de un atributo de otro objeto, lo que se utiliza en la especificación de la red de telecomunicaciones para definir la relación de contenido entre objetos. Facilita además el proceso de creación de objetos mediante el uso de objetos de clase `ProtoObject` que proporcionan identificadores únicos a los objetos del sistema.

Por otra parte, la lógica VLRL, aunque permite la especificación de propiedades sobre sistemas con cualquier estructura, se beneficia de las facilidades proporcionadas por Maude para la especificación de sistemas orientados a objetos, tanto en la definición de las observaciones del sistema como en la posterior verificación de las propiedades.

En Maude, el estado de un sistema concurrente orientado a objetos se denomina *configuración* y tiene la estructura de un multiconjunto compuesto por objetos y mensajes que evoluciona mediante reescrituras concurrentes usando las reglas que describen los efectos de los eventos de comunicación entre algunos objetos y mensajes de la configuración. El tipo *State* definido en la lógica VLRL coincidirá en estos sistemas con la configuración, o bien con una restricción de ella. Obsérvese que como la construcción de multiconjuntos es asociativa, conmutativa y con elemento neutro, los estados siempre se pueden descomponer. Esta propiedad se mantiene cuando restringimos la noción de configuración para definir el tipo *State* siempre que se mantenga la estructura de multiconjunto con elemento neutro.

Los tres ejemplos desarrollados en este capítulo han sido presentados en distintas conferencias. El protocolo sencillo de exclusión mutua (Sección 5.1) se presentó en la conferencia *Recent Trends in Algebraic Development Techniques, WADT'99 (Chateau de Bonas, Francia)* junto con la definición de la lógica VLRL y se publicó en [FMMO+00]. El mundo de bloques (Sección 5.2) se presentó en la *APPIA-GULP-PRODE'01 Joint Conference on Declarative Programming (Évora, Portugal)* junto con la definición de las transiciones *en contexto*, publicándose en [PMO01]. Por último, el sistema Mobile Maude (Sección 5.5) se ha presentado en la conferencia *PROLE 2002 (El Escorial, Madrid)* junto con las reglas de derivación de las propiedades espaciales básicas, publicándose en [Pit02].

5.1. Un ejemplo sencillo de exclusión mutua

En primer lugar, desarrollamos un ejemplo basado en un sencillo protocolo de exclusión mutua. El módulo Maude que especifica el sistema es:

```
(omod MUTUAL-EXCLUSION-PROTOCOL is
  sort Status .
  op bcs : -> Status .   *** objeto puede entrar en sección crítica
  op ics : -> Status .   *** objeto está en sección crítica
  op acs : -> Status .   *** objeto no está listo para entrar en secc. crit.
  class obj | status : Status .
  msg token : -> Msg .
  var X : Oid .
  rl [enter] : < X : obj | status : bcs > token => < X : obj | status : ics > .
  rl [exit]  : < X : obj | status : ics > => < X : obj | status : acs > token .
  rl [ready] : < X : obj | status : acs > => < X : obj | status : bcs > .
   endom)
```

Las observaciones de interés son $\#cs$, $\#tk$: `Nat` que devuelven el número de objetos en la sección crítica (esto es, aquellos cuyo atributo `status` tiene el valor `ics`) y el número de `token`s en el sistema, respectivamente.

El programa observado viene dado por la siguiente teoría de reescritura, donde hemos añadido el tipo `Nat` y sus operaciones a la signatura de reescritura del sistema y definimos ecuacionalmente los atributos de observación[1].

```
(omod MEP-OBSERVED is
  protecting MUTUAL-EXCLUSION-PROTOCOL .
  protecting NAT .
  ops #cs #tk : Configuration -> Nat .
  var C : Configuration .
  var S : Status .
  eq #cs(< X : obj | status : ics > C) = 1 + #cs(C) .
  eq #cs(< X : obj | status : bcs > C) = #cs(C) .
  eq #cs(< X : obj | status : acs > C) = #cs(C) .
```

[1] Obsérvese que algunas ecuaciones son condicionales para evitar problemas de ejecución como la no terminación debido a la identidad `none`, aunque son matemáticamente correctas sin la condición.

```
    eq #cs(token C) = #cs(C) .
    eq #cs(none) = 0 .
    eq #tk(< X : obj | status : S > C) = #tk(C) .
    eq #tk(token C) = 1 + #tk(C) .
    eq #tk(none) = 0 .
  endom)
```

Algunas fórmulas que modelan las acciones del sistema son

$$(\#cs = N \wedge \#tk = M) \supset [[\texttt{enter(a)}]](\#cs = N + 1 \wedge \#tk = M - 1)$$
$$(\#cs = N \wedge \#tk = M) \supset [[\texttt{exit(a)}]](\#cs = N - 1 \wedge \#tk = M + 1)$$
$$(\#cs = N \wedge \#tk = M) \supset [[\texttt{ready(a)}]](\#cs = N \wedge \#tk = M)$$
$$\langle\langle \texttt{enter(a)} \rangle\rangle true \supset \#tk > 0$$
$$\langle \texttt{enter(a)} \rangle true \supset \#tk = 1 \tag{5.1}$$

donde las acciones, como `enter(a)`, se construyen usando la regla de *reemplazamiento*, con etiqueta `enter` y una sustitución de la variable `X` de la regla por el término `a`.

La siguiente propiedad estructural se obtiene a partir de la regla (4.44) para derivar propiedades espaciales básicas de observaciones de tipo `Nat`, en la que se considera que los estados siempre se pueden descomponer:

$$\#tk = N \equiv \exists N_1, N_2 . __\langle \#tk = N_1, \#tk = N_2 \rangle \wedge N = N_1 + N_2 \ \ . \tag{5.2}$$

5.1.1. Probando la exclusión mutua

Probamos la siguiente propiedad, que afirma que en un estado con un único “token”, no es posible que dos objetos `a` y `b` entren simultáneamente en la sección crítica.

$$\#tk = 1 \supset [__(\texttt{enter(a)}, \texttt{enter(b)})] false \ \ .$$

Asúmase el antecedente $\#tk = 1$. Por la propiedad estructural (5.2),

$$\exists N_1, N_2 . __\langle \#tk = N_1, \#tk = N_2 \rangle \wedge 1 = N_1 + N_2 \ \ ,$$

y por aritmética

$$__\langle \#tk = 1, \#tk = 0 \rangle \vee __\langle \#tk = 0, \#tk = 1 \rangle \ \ . \tag{5.3}$$

En el primer caso, a partir de la propiedad (5.1) para modelar las acciones del sistema obtenemos por contraposición de la lógica proposicional y aplicando la regla (4.17) de dualidad de la modalidad de acción

$$\#tk = 0 \supset [\texttt{enter(b)}] false \ \ . \tag{5.4}$$

Por medio de la regla de inferencia de VLRL (4.33) aplicada a (5.4) y a la tautología $\#tk = 1 \supset [\texttt{enter(a)}] true$ obtenemos

$$__\langle \#tk = 1, \#tk = 0 \rangle \supset [__(\texttt{enter(a)}, \texttt{enter(b)})] __\langle true, false \rangle \ \ ,$$

y a partir de aquí aplicando contraposición y la regla (4.41) de dualidad topológica a la regla de inferencia (4.40) se sigue

$$__\langle \#tk = 1, \#tk = 0\rangle \supset [__(\texttt{enter(a)}, \texttt{enter(b)})] \mathit{false} \ .$$

De forma análoga, en el segundo caso, podemos probar

$$__\langle \#tk = 0, \#tk = 1\rangle \supset [__(\texttt{enter(a)}, \texttt{enter(b)})] \mathit{false} \ .$$

5.1.2. Probando un invariante

Consideremos ahora la demostración de que la suma $\#cs + \#tk$ es *invariante*:

$$\#cs + \#tk = N \supset \mathrm{AG}(\#cs + \#tk = N) \ .$$

Utilizando la técnica de inducción estructural descrita en la Sección 4.6.1, es suficiente con probar:

- constantes[2]:

  1. $\texttt{none}\langle\rangle \supset (\#cs + \#tk = N \supset [[\texttt{none}]](\#cs + \#tk = N))$
  2. $\texttt{token}\langle\rangle \supset (\#cs + \#tk = N \supset [[\texttt{token}]](\#cs + \#tk = N))$
  3. $\texttt{<_:_|>}_{\texttt{x1,x2}}\langle\rangle \supset (\#cs + \#tk = N \supset$ $[[\texttt{<_:_|>}_{\texttt{x1,x2}}]](\#cs + \#tk = N))$
  4. $\texttt{<_:_|_>}_{\texttt{x1,x2,x3}}\langle\rangle \supset (\#cs + \#tk = N \supset$ $[[\texttt{<_:_|_>}_{\texttt{x1,x2,x3}}]](\#cs + \#tk = N))$

- reglas:

  5. $\#cs + \#tk = N \supset [\texttt{enter(a)}](\#cs + \#tk = N)$
  6. $\#cs + \#tk = N \supset [\texttt{exit(a)}](\#cs + \#tk = N)$
  7. $\#cs + \#tk = N \supset [\texttt{ready(a)}](\#cs + \#tk = N)$

- operaciones sobre estados (en este caso, solo $__$):

  8. $(\#cs + \#tk = N_1 \supset [\alpha_1](\#cs + \#tk = N_1)),$ $(\#cs + \#tk = N_2 \supset [\alpha_2](\#cs + \#tk = N_2))$ $\vdash __\langle \mathit{true}, \mathit{true}\rangle \supset (\#cs + \#tk = N \supset [__(\alpha_1, \alpha_2)](\#cs + \#tk = N)) \ .$

[2]Algunas constantes de tipo *State* se obtienen del módulo `CONFIGURATION`, importado en el proceso de transformación de un módulo orientado a objetos en un módulo de sistema (ver Sección 2.2.3). En el proceso de transformación añadimos una constante por cada identificador de mensaje. No ponemos las operaciones constantes derivadas de los identificadores de las clases y de los atributos ya que no son de tipo *State*.

La demostración de las siete primeras propiedades es inmediata. Consideremos la prueba del paso de inducción. Asúmanse las hipótesis; usando la regla de inferencia de VLRL (4.33) derivamos

$$\begin{aligned} &__\langle \#cs + \#tk = N_1, \#cs + \#tk = N_2 \rangle \supset \\ &\quad [__(\alpha_1, \alpha_2)]__\langle \#cs + \#tk = N_1, \#cs + \#tk = N_2 \rangle \ . \end{aligned} \tag{5.5}$$

Aplicando la siguiente propiedad estructural de las configuraciones derivada de las reglas (4.43) y (4.44)

$$\#cs + \#tk = N \equiv \exists N_1, N_2 . __\langle \#cs + \#tk = N_1, \#cs + \#tk = N_2 \rangle \wedge N = N_1 + N_2 \ ,$$

al antecedente ($\#cs + \#tk = N$), y teniendo en cuenta la fórmula (5.5), obtenemos

$$[__(\alpha_1, \alpha_2)]__\langle \#cs + \#tk = N_1, \#cs + \#tk = N_2 \rangle \ ,$$

y es suficiente con aplicar la misma propiedad estructural en el otro sentido para concluir que $[__(\alpha_1, \alpha_2)](\#cs + \#tk = N)$.

5.2. Un mundo de bloques

En este ejemplo, ilustramos el uso de la modalidad espacial y de la modalidad de acción, e introducimos el uso de variables en la definición de propiedades. El ejemplo está basado en una versión orientada a objetos del mundo de los bloques [CDE+00a, Sección 9.5] que se mostró en la Sección 2.2.3 como ejemplo de sistema orientado a objetos en Maude. Repetimos a continuación el código mostrado en dicha sección. Un bloque se representa por medio de un objeto con dos atributos, `under`, que indica si el bloque está debajo de otro bloque o si está libre, y `on`, que indica si el bloque está sobre otro bloque o si está sobre la mesa. Un robot se representa por medio de otro objeto con un atributo `hold`, que indica si el robot está vacío o si sostiene un bloque. Las acciones se representan por medio de mensajes.

Primero presentamos el sistema en Maude, y a continuación probamos algunas propiedades sobre el número de bloques en el sistema; por último, introducimos observaciones para probar propiedades sobre la posición ocupada por los bloques en el sistema.

```
(omod OO-BLOCKSWORLD is
  protecting QID .
  sorts BlockId RobotId Up Down Hold .
  subsorts Qid < BlockId RobotId < Oid .
  subsorts BlockId < Up Down Hold .
  op clear : -> Up .      *** el bloque no tiene nada encima
  op catchup : -> Up .    *** el bloque está cogido por el robot
  op table : -> Down .    *** el bloque está en la mesa
  op catchd : -> Down .   *** el bloque está cogido por el robot
  op empty : -> Hold .    *** el robot está vacío
  class Block | under : Up, on : Down .
```

```
  class Robot | hold : Hold .
  msgs pickup putdown : RobotId BlockId -> Msg .
  msgs unstack stack : RobotId BlockId BlockId -> Msg .
  vars X Y : BlockId .
  var R : RobotId .

  rl [pickup] : pickup(R,X) < R : Robot | hold : empty >
                < X : Block | under : clear, on : table >
             => < R : Robot | hold : X >
                < X : Block | under : catchup, on : catchd > .
  rl [putdown] : putdown(R,X) < R : Robot | hold : X >
                 < X : Block | under : catchup, on : catchd >
              => < R : Robot | hold : empty >
                 < X : Block | under : clear, on : table > .
  rl [unstack] : unstack(R,X,Y) < R : Robot | hold : empty >
                 < X : Block | under : clear, on : Y >
                 < Y : Block | under : X >
              => < R : Robot | hold : X >
                 < X : Block | under : catchup, on : catchd >
                 < Y : Block | under : clear > .
  rl [stack] : stack(R,X,Y) < R : Robot | hold : X >
               < X : Block | under : catchup, on : catchd >
               < Y : Block | under : clear >
            => < R : Robot | hold : empty >
               < X : Block | under : clear, on : Y >
               < Y : Block | under : X > .
endom)
```

5.3. Observando el número de bloques

5.3.1. Definición de las observaciones

Definimos las siguientes observaciones para formular propiedades sobre el número de bloques en el sistema:

- *#blTable*, número de bloques sobre la mesa,
- *#blOnBl*, número de bloques que están sobre otros bloques,
- *#blHold*, número de bloques que están sostenidos por robots,
- *#blocks*, número total de bloques en el sistema,
- *#empty*, número de robots vacíos,
- *#robots*, número total de robots.

El programa observado viene dado por la siguiente teoría de reescritura:

```
(omod OO-BLOCKSWORLD-NUMBLOCKS-OBSERVED is
  protecting OO-BLOCKSWORLD .
  protecting NAT .

  op #blTable : Configuration -> Nat .
  op #blOnBl : Configuration -> Nat .
  op #blHold : Configuration -> Nat .
  op #blocks : Configuration -> Nat .
  op #empty : Configuration -> Nat .
  op #robots : Configuration -> Nat .

  var C : Configuration .
  vars X Y : BlockId .
  var R : RobotId .
  var M : Msg .

  eq #blTable(< X : Block | on : table > C) = 1 + #blTable(C) .
  eq #blTable(< X : Block | on : catchd > C) = #blTable(C) .
  eq #blTable(< X : Block | on : Y > C)  = #blTable(C) .
  eq #blTable(< R : Robot | > C) = #blTable(C) .
  eq #blTable(none) = 0 .
  eq #blTable(M C) = #blTable(C) .

  eq #blOnBl(< X : Block | on : table > C) = #blOnBl(C) .
  eq #blOnBl(< X : Block | on : catchd > C) = #blOnBl(C) .
  eq #blOnBl(< X : Block | on : Y > C)  = 1 + #blOnBl(C) .
  eq #blOnBl(< R : Robot | > C) = #blOnBl(C) .
  eq #blOnBl(none) = 0 .
  eq #blOnBl(M C) = #blOnBl(C) .

  eq #blHold(< X : Block | > C) = #blHold(C) .
  eq #blHold(< R : Robot | hold : empty > C) = #blHold(C) .
  eq #blHold(< R : Robot | hold : X > C) = 1 + #blHold(C) .
  eq #blHold(none) = 0 .
  eq #blHold(M C) = #blHold(C) .

  eq #blocks(C) = #blTable(C) + #blOnBl(C) + #blHold(C) .

  eq #empty(< X : Block | > C) = #empty(C) .
  eq #empty(< R : Robot | hold : X > C) = #empty(C) .
  eq #empty(< R : Robot | hold : empty > C) = 1 + #empty(C) .
  eq #empty(none) = 0 .
  eq #empty(M C) = #empty(C) .

  eq #robots(C) = #blHold(C) + #empty(C) .
endom)
```

Podemos tener, por ejemplo, un estado con tres bloques y dos robots, en el que el bloque `a` está sobre la mesa y bajo el bloque `c`, el bloque `b` está libre y sobre la mesa, y el bloque `c` está sobre el bloque `a` y libre. Los dos robots están vacíos. El estado queda

descrito por la siguiente configuración.

```
< 'a : Block | under : 'c, on : table >
< 'c : Block | under : clear, on : 'a >
< 'b : Block | under : clear, on : table >
< 'r : Robot | hold : empty >
< 's : Robot | hold : empty >
```

En lo que sigue consideraremos únicamente configuraciones consistentes, en el sentido de que describan situaciones reales posibles del mundo de los bloques; por ejemplo, los bloques no pueden estar duplicados, si un bloque está sobre otro, entonces el primero está debajo del segundo, no puede haber dos bloques encima de uno dado, etcétera.

5.3.2. Ejemplos de modalidad espacial y modalidad de acción

En esta sección presentamos varios ejemplos sobre el uso de la modalidad espacial y la modalidad de acción en la especificación de propiedades. En la configuración anterior de la Sección 5.3.1 se satisfacen las siguientes fórmulas:

- El estado se puede descomponer en un subestado con dos bloques, uno sobre el otro, y un robot, y otro subestado con un bloque sobre la mesa y el otro robot:

$$__\langle \#blocks = 2 \wedge \#blOnBl = 1 \wedge \#robots = 1,\\ \#blocks = 1 \wedge \#blTable = 1 \wedge \#robots = 1\rangle \ .$$

- El estado también puede descomponerse en un subestado con todos los bloques y otro con todos los robots:

$$__\langle \#blocks = 3 \wedge \#robots = 0, \#blocks = 0 \wedge \#robots = 2\rangle \ .$$

Con respecto a la modalidad de acción, en el estado

```
< 'a : Block | under : catchup, on : catchd >
< 'c : Block | under : 'b, on : table >
< 'b : Block | under : clear, on : 'c >
< 'r : Robot | hold : empty > < 's : Robot | hold : 'a >
putdown('s,'a)  unstack('r,'b,'c)
```

se satisfacen las siguientes fórmulas:

- La regla `putdown` se puede aplicar al robot `s` y al bloque `a`:

$$\langle\langle \mathtt{putdown(s,a)} \rangle\rangle \mathit{true} \ .$$

  Obsérvese que utilizamos la modalidad $\langle\langle _ \rangle\rangle$ en lugar de la modalidad *simple* $\langle _ \rangle$ dado que queremos aplicar la transición *en contexto* al subtérmino

```
< 'a : Block | under : catchup, on : catchd >
< 's : Robot | hold : 'a > putdown('s,'a)
```

- Las reglas `unstack` y `putdown` pueden aplicarse concurrentemente:

$$\langle __(\mathtt{unstack}(\mathtt{r},\mathtt{b},\mathtt{c}),\ \ \mathtt{putdown}(\mathtt{s},\mathtt{a}))\rangle \mathit{true}\ \ .$$

  Aquí podemos usar la modalidad *simple* $\langle _ \rangle$ dado que aplicamos la acción a la configuración completa.

- Las reglas `unstack` y `putdown` pueden aplicarse concurrentemente, y después de realizarse la transición, un subestado tendrá un bloque sostenido por un robot y el otro subestado tendrá un bloque sobre la mesa:

$$\langle __(\mathtt{unstack}(\mathtt{r},\mathtt{b},\mathtt{c}),\ \ \mathtt{putdown}(\mathtt{s},\mathtt{a}))\rangle __\langle \#\mathit{blHold} = 1, \#\mathit{blTable} = 1\rangle\ \ .$$

- Si las reglas `unstack` y `putdown` se aplican concurrentemente, después de realizarse la transición un subestado tendrá dos bloques, uno de ellos sostenido por un robot y un robot, y el otro subestado tendrá un robot vacío y un bloque sobre la mesa:

$$\begin{aligned}&[__(\mathtt{unstack}(\mathtt{r},\mathtt{b},\mathtt{c}),\ \ \mathtt{putdown}(\mathtt{s},\mathtt{a}))]\\ &\quad __\langle \#\mathit{blHold} = 1 \wedge \#\mathit{blocks} = 2 \wedge \#\mathit{robots} = 1,\\ &\qquad \#\mathit{empty} = 1 \wedge \#\mathit{blTable} = 1\rangle\ \ .\end{aligned}$$

- Si las reglas `pickup` y `putdown` se aplican concurrentemente *en contexto*, entonces, después de realizarse la transición, un subestado tendrá uno o más bloques sobre la mesa y el otro subestado tendrá uno o más bloques sostenidos por los robots:

$$[[__(\mathtt{pickup}(\mathtt{r},\mathtt{b}),\ \ \mathtt{putdown}(\mathtt{s},\mathtt{a}))]]__\langle \#\mathit{blHold} \geq 1, \#\mathit{blTable} \geq 1\rangle\ \ .$$

- En todos los subestados posibles, si aplicamos la regla `putdown` al robot `s` y al bloque `a`, entonces el número de bloques sobre la mesa en ese subestado será mayor o igual que uno:

$$__[[\mathtt{putdown}(\mathtt{s},\mathtt{a})]\#\mathit{blTable} \geq 1, \mathit{false}]\ \ .$$

  Obsérvese que al ser *false* la propiedad que debe cumplir el segundo subestado se fuerza el que en cualquier descomposición del estado, el primer subestado cumpla la propiedad $[\mathtt{putdown}(\mathtt{s},\mathtt{a})]\#\mathit{blTable} \geq 1$. De esta forma como la operación $__$ es conmutativa obtenemos que la propiedad se cumple en todos los subestados posibles.

5.3.3. Propiedades básicas sobre el número de bloques

En esta sección, presentamos algunas propiedades básicas que se cumplen en el sistema `OO-BLOCKSWORLD` y que pueden probarse directamente a partir de la relación de satisfacción.

Propiedades relativas a las transiciones del sistema

Para cada regla de reescritura expresamos los cambios que tienen lugar sobre cada observación cuando la transición asociada a la regla se realiza. Por ejemplo, si consideramos la regla de reescritura `pickup` y la correspondiente acción $\texttt{pickup}(\texttt{u},\texttt{a})$, donde `u` y `a` representan una sustitución de las variables de la regla, podemos formular las siguientes propiedades sobre el número de bloques:

$$\begin{aligned}
&\#\mathit{blHold} = N \supset [[\texttt{pickup}(\texttt{u},\texttt{a})]](\#\mathit{blHold} = N + 1)\\
&\#\mathit{blTable} = N \supset [[\texttt{pickup}(\texttt{u},\texttt{a})]](\#\mathit{blTable} = N - 1)\\
&\#\mathit{blOnBl} = N \supset [[\texttt{pickup}(\texttt{u},\texttt{a})]](\#\mathit{blOnBl} = N)\\
&\#\mathit{empty} = N \supset [[\texttt{pickup}(\texttt{u},\texttt{a})]](\#\mathit{empty} = N - 1)\\
&\#\mathit{blocks} = N \supset [[\texttt{pickup}(\texttt{u},\texttt{a})]](\#\mathit{blocks} = N)\\
&\#\mathit{robots} = N \supset [[\texttt{pickup}(\texttt{u},\texttt{a})]](\#\mathit{robots} = N)\ .
\end{aligned}$$

Propiedades sobre las condiciones que permiten realizar las acciones

Si observamos el lado izquierdo de cada regla de reescritura, podemos definir las siguientes propiedades:

$$\begin{aligned}
&\langle\langle\texttt{pickup}(\texttt{u},\texttt{a})\rangle\rangle \mathit{true} \supset \#\mathit{blTable} \geq 1 \wedge \#\mathit{empty} \geq 1\\
&\langle\langle\texttt{putdown}(\texttt{u},\texttt{a})\rangle\rangle \mathit{true} \supset \#\mathit{blHold} \geq 1\\
&\langle\langle\texttt{unstack}(\texttt{u},\texttt{a},\texttt{b})\rangle\rangle \mathit{true} \supset \#\mathit{blOnBl} \geq 1 \wedge \#\mathit{empty} \geq 1\\
&\langle\langle\texttt{stack}(\texttt{u},\texttt{a},\texttt{b})\rangle\rangle \mathit{true} \supset \#\mathit{blHold} \geq 1\ .
\end{aligned}$$

Obsérvese que las correspondientes equivalencias no se cumplen dado que no conocemos ni la posición del bloque `a`, ni la situación del robot.

Propiedades espaciales sobre el número de bloques

Estas propiedades expresan cuándo un estado puede descomponerse en subestados, y las propiedades que se satisfacen en cada subestado. En el caso del número de bloques se derivan de las reglas (4.43) y (4.44) de obtención de propiedades espaciales básicas para observaciones de tipo `Nat`, teniendo en cuenta que como la configuración es orientada a objetos los estados siempre se pueden descomponer; por ejemplo, tenemos

$$\#\mathit{blocks} = N \equiv \exists N_1, N_2 . __\langle \#\mathit{blocks} = N_1, \#\mathit{blocks} = N_2\rangle \wedge (N = N_1 + N_2)\ . \tag{5.6}$$

5.3.4. Una propiedad temporal sobre el número de bloques

El número de bloques en el sistema es invariante:

$$\#\mathit{blocks} = N \supset \mathrm{AG}(\#\mathit{blocks} = N)\ .$$

Dada la definición de la conectiva temporal AG, $\mathrm{AG}\varphi \equiv \mathrm{A}(\varphi \mathrm{W} \mathit{false})$ y aplicando la segunda regla de inferencia de la interfaz entre la lógica temporal y VLRL (4.65), es suficiente con probar

$$\{\#\mathit{blocks} = N \supset [\alpha](\#\mathit{blocks} = N) \mid \alpha \text{ acción}\} \ .$$

Usando la inducción estructural sobre las acciones (ver Sección 4.6.1), debemos probar:

- constantes:
  1. $\texttt{<_:_|>}_{\texttt{x1,x2}}\langle\rangle \supset (\#\mathit{blocks} = N) \supset [[\texttt{<_:_|>}_{\texttt{x1,x2}}]](\#\mathit{blocks} = N)$
  2. $\texttt{<_:_|_>}_{\texttt{x1,x2,x3}}\langle\rangle \supset (\#\mathit{blocks} = N) \supset [[\texttt{<_:_|_>}_{\texttt{x1,x2,x3}}]](\#\mathit{blocks} = N)$
  3. $\texttt{none}\langle\rangle \supset (\#\mathit{blocks} = N) \supset [[\texttt{none}]](\#\mathit{blocks} = N)$
  4. $\texttt{pickup}_{\texttt{u,a}}\langle\rangle \supset (\#\mathit{blocks} = N) \supset [[\texttt{pickup}_{\texttt{u,a}}]](\#\mathit{blocks} = N)$
  5. $\texttt{putdown}_{\texttt{u,a}}\langle\rangle \supset (\#\mathit{blocks} = N) \supset [[\texttt{putdown}_{\texttt{u,a}}]](\#\mathit{blocks} = N)$
  6. $\texttt{unstack}_{\texttt{u,a,b}}\langle\rangle \supset (\#\mathit{blocks} = N) \supset [[\texttt{unstack}_{\texttt{u,a,b}}]](\#\mathit{blocks} = N)$
  7. $\texttt{stack}_{\texttt{u,a,b}}\langle\rangle \supset (\#\mathit{blocks} = N) \supset [[\texttt{stack}_{\texttt{u,a,b}}]](\#\mathit{blocks} = N)$

- reglas:
  8. $(\#\mathit{blocks} = N) \supset [\texttt{pickup}(\mathtt{u}, \mathtt{x})](\#\mathit{blocks} = N)$
  9. $(\#\mathit{blocks} = N) \supset [\texttt{putdown}(\mathtt{u}, \mathtt{x})](\#\mathit{blocks} = N)$
  10. $(\#\mathit{blocks} = N) \supset [\texttt{unstack}(\mathtt{u}, \mathtt{x}, \mathtt{y})](\#\mathit{blocks} = N)$
  11. $(\#\mathit{blocks} = N) \supset [\texttt{stack}(\mathtt{u}, \mathtt{x}, \mathtt{y})](\#\mathit{blocks} = N)$

- operaciones sobre estados (en este caso, solo $__$):
  12. $((\#\mathit{blocks} = N_1) \supset [\alpha_1](\#\mathit{blocks} = N_1)),$
  $((\#\mathit{blocks} = N_2) \supset [\alpha_2](\#\mathit{blocks} = N_2))$
  $\vdash __\langle \mathit{true}, \mathit{true}\rangle \supset (\#\mathit{blocks} = N \supset [__(\alpha_1, \alpha_2)](\#\mathit{blocks} = N)) \ .$

La demostración de las siete primeras propiedades es inmediata. La demostración de las propiedades 8–11 se basa en las propiedades sobre las transiciones del sistema, y en la propiedad (4.62).

Probamos la fórmula del último caso de la siguiente forma. Asumiendo las hipótesis, derivamos mediante la regla de inferencia (4.33) de VLRL

$$__\langle \#\mathit{blocks} = N_1, \#\mathit{blocks} = N_2\rangle \supset [__(\alpha_1, \alpha_2)]__\langle \#\mathit{blocks} = N_1, \#\mathit{blocks} = N_2\rangle \ .$$

Aplicando este resultado a la propiedad espacial (5.6), obtenemos

$$\#\mathit{blocks} = N \supset \exists N_1, N_2.([__(\alpha_1, \alpha_2)]__\langle \#\mathit{blocks} = N_1, \#\mathit{blocks} = N_2\rangle) \\ \wedge (N = N_1 + N_2) \ .$$

Dado que N, N_1, N_2 no son variables de estado, podemos aplicar (4.15):

$$\#\mathit{blocks} = N \supset \exists N_1, N_2.[__(\alpha_1, \alpha_2)]__\langle \#\mathit{blocks} = N_1, \#\mathit{blocks} = N_2\rangle \\ \wedge [__(\alpha_1, \alpha_2)](N = N_1 + N_2) \ .$$

Aplicando ahora (4.26) y (4.16), obtenemos

$$\#blocks = N \supset [__(\alpha_1, \alpha_2)] \exists N_1, N_2 . __ \langle \#blocks = N_1, \#blocks = N_2 \rangle \\ \wedge (N = N_1 + N_2) \ .$$

Finalmente, usando de nuevo la propiedad espacial (5.6) en el otro sentido obtenemos el resultado:

$$\#blocks = N \supset [__(\alpha_1, \alpha_2)](\#blocks = N) \ .$$

5.4. Observando la posición de los bloques

5.4.1. Definición de las observaciones

Para formular propiedades sobre la posición de los bloques en el sistema necesitamos tener en cuenta cada bloque de forma individual. Generalizamos la definición de las observaciones dada en la Sección 4.2 y consideramos familias de observaciones, una por cada bloque o robot en el sistema. En particular, consideramos las siguientes observaciones:

- $blTable_x$, es cierto si y solo si el bloque x está sobre la mesa,
- $blClear_x$, es cierto si y solo si no hay ningún bloque sobre el bloque x,
- $blOnBl_{x,y}$, es cierto si y solo si el bloque x está sobre el bloque y,
- $blHold_{r,x}$, es cierto si y solo si el bloque x está sostenido por el robot r,
- $blHold_x$, es cierto si y solo si el bloque x está sostenido por algún robot,
- $empty_r$, es cierto si y solo si el robot r está vacío.

El programa observado en Maude es el siguiente.

```
(omod OO-BLOCKSWORLD-POSITION-OBSERVED is
  protecting OO-BLOCKSWORLD .

  op blTable : Configuration Oid -> Bool .
  op blClear : Configuration Oid -> Bool .
  op blOnBl : Configuration Oid Oid -> Bool .
  op blHold : Configuration Oid Oid -> Bool .
  op blHold : Configuration Oid -> Bool .
  op empty : Configuration Oid -> Bool .

  var C : Configuration .
  vars X Y Z : BlockId .
  vars R S : RobotId .
  var M : Msg .

  eq blTable(< X : Block | on : table > C,X) = true .
```

```
  eq blTable(< X : Block | on : Y > C,X) = blTable(C,X) .
  ceq blTable(< Y : Block | > C,X) = blTable(C,X) if X =/= Y .
  eq blTable(< R : Robot | > C,X) = blTable(C,X) .
  eq blTable(none,X) = false .
  eq blTable(M C,X) = blTable(C,X) .

  eq blClear(< X : Block | under : clear > C,X) = true .
  eq blClear(< X : Block | under : Y > C,X) = blClear(C,X) .
  ceq blClear(< Y : Block | > C,X) = blClear(C,X) if X =/= Y .
  eq blClear(< R : Robot | > C,X) = blClear(C,X) .
  eq blClear(none,X) = false .
  eq blClear(M C,X) = blClear(C,X) .

  eq blOnBl(< X : Block | on : table > C,X,Y) = blOnBl(C,X,Y) .
  eq blOnBl(< X : Block | on : Z > C,X,Y) = Y == Z or blOnBl(C,X,Y) .
  ceq blOnBl(< Z : Block | > C,X,Y) = blOnBl(C,X,Y) if X =/= Z .
  eq blOnBl(< R : Robot | > C,X,Y) = blOnBl(C,X,Y) .
  eq blOnBl(none,X,Y) = false .
  eq blOnBl(M C,X,Y) = blOnBl(C,X,Y) .

  eq blHold(< X : Block | > C,R,Y) = blHold(C,R,Y) .
  eq blHold(< R : Robot | hold : empty > C,S,Y) = blHold(C,S,Y) .
  eq blHold(< R : Robot | hold : X > C,S,Y) =
     (X == Y and R == S) or blHold(C,S,Y) .
  eq blHold(none,R,Y) = false .
  eq blHold(M C,R,Y) = blHold(C,R,Y) .

  eq blHold(< X : Block | > C,Y) = blHold(C,Y) .
  eq blHold(< R : Robot | hold : empty > C,Y) = blHold(C,Y) .
  eq blHold(< R : Robot | hold : X > C,Y) = (X == Y) or blHold(C,Y) .
  eq blHold(none,Y) = false .
  eq blHold(M C,Y) = blHold(C,Y) .

  eq empty(< X : Block | > C,R) = empty(C,R) .
  eq empty(< R : Robot | hold : empty > C,S) = (R == S) or empty(C,S) .
  eq empty(< R : Robot | hold : X > C,S) = empty(C,S) .
  eq empty(none,R) = false .
  eq empty(M C,R) = empty(C,R) .
endom)
```

5.4.2. Propiedades básicas sobre la posición de los bloques

En lo que sigue, `a`, `b`, `c`, y `d` denotan cuatro bloques diferentes, y `u` y `v` son dos robots diferentes[3].

[3] Algunas propiedades se cumplen también cuando las constantes denotan el mismo objeto.

Propiedades relativas a las transiciones del sistema

$$
\begin{aligned}
& \mathit{blTable}_{\mathtt{a}} \wedge \mathit{blClear}_{\mathtt{a}} \wedge \mathit{empty}_{\mathtt{u}} \supset [[\mathtt{pickup}(\mathtt{u},\mathtt{a})]]\,\mathit{blHold}_{\mathtt{u},\mathtt{a}} \\
& \mathit{blOnBl}_{\mathtt{a},\mathtt{b}} \supset [[\mathtt{pickup}(\mathtt{u},\mathtt{c})]]\,\mathit{blOnBl}_{\mathtt{a},\mathtt{b}} && (5.7) \\
& \mathit{blHold}_{\mathtt{u},\mathtt{a}} \supset [[\mathtt{putdown}(\mathtt{u},\mathtt{a})]](\mathit{empty}_{\mathtt{u}} \wedge \mathit{blClear}_{\mathtt{a}} \wedge \mathit{blTable}_{\mathtt{a}}) \\
& \mathit{blOnBl}_{\mathtt{a},\mathtt{b}} \supset [[\mathtt{putdown}(\mathtt{u},\mathtt{c})]]\,\mathit{blOnBl}_{\mathtt{a},\mathtt{b}} && (5.8) \\
& \mathit{empty}_{\mathtt{u}} \wedge \mathit{blClear}_{\mathtt{a}} \wedge \mathit{blOnBl}_{\mathtt{a},\mathtt{b}} \supset \\
& \qquad [[\mathtt{unstack}(\mathtt{u},\mathtt{a},\mathtt{b})]](\mathit{blHold}_{\mathtt{u},\mathtt{a}} \wedge \mathit{blClear}_{\mathtt{b}}) && (5.9) \\
& \mathit{blHold}_{\mathtt{u},\mathtt{a}} \wedge \mathit{blClear}_{\mathtt{b}} \supset \\
& \qquad [[\mathtt{stack}(\mathtt{u},\mathtt{a},\mathtt{b})]](\mathit{empty}_{\mathtt{u}} \wedge \mathit{blClear}_{\mathtt{a}} \wedge \mathit{blOnBl}_{\mathtt{a},\mathtt{b}}) \\
& \mathit{blOnBl}_{\mathtt{a},\mathtt{b}} \supset [[\mathtt{stack}(\mathtt{u},\mathtt{c},\mathtt{d})]]\,\mathit{blOnBl}_{\mathtt{a},\mathtt{b}} && (5.10)
\end{aligned}
$$

Propiedades sobre las condiciones que permiten realizar las acciones

Definimos una propiedad por cada regla de reescritura en el sistema.

$$
\begin{aligned}
& \langle\langle \mathtt{pickup}(\mathtt{u},\mathtt{a})\rangle\rangle \mathit{true} \supset \mathit{blTable}_{\mathtt{a}} \wedge \mathit{blClear}_{\mathtt{a}} \wedge \mathit{empty}_{\mathtt{u}} \\
& \langle\langle \mathtt{putdown}(\mathtt{u},\mathtt{a})\rangle\rangle \mathit{true} \supset \mathit{blHold}_{\mathtt{u},\mathtt{a}} \\
& \langle\langle \mathtt{unstack}(\mathtt{u},\mathtt{a},\mathtt{b})\rangle\rangle \mathit{true} \supset \mathit{blOnBl}_{\mathtt{a},\mathtt{b}} \wedge \mathit{blClear}_{\mathtt{a}} \wedge \mathit{empty}_{\mathtt{u}} && (5.11) \\
& \langle\langle \mathtt{stack}(\mathtt{u},\mathtt{a},\mathtt{b})\rangle\rangle \mathit{true} \supset \mathit{blHold}_{\mathtt{u},\mathtt{a}} \wedge \mathit{blClear}_{\mathtt{b}}
\end{aligned}
$$

Propiedades de observaciones derivadas

En el conjunto de observaciones que hemos definido, hay algunas relaciones entre atributos que pueden expresarse por medio de las siguientes fórmulas:

$$
\begin{aligned}
& \mathit{blClear}_{\mathtt{a}} \supset \neg\mathit{blOnBl}_{\mathtt{b},\mathtt{a}} \wedge \neg\mathit{blHold}_{\mathtt{u},\mathtt{a}} && (5.12) \\
& \mathit{blOnBl}_{\mathtt{a},\mathtt{b}} \supset \neg\mathit{blTable}_{\mathtt{a}} \wedge \neg\mathit{blClear}_{\mathtt{b}} \wedge \neg\mathit{blHold}_{\mathtt{a}} \wedge \neg\mathit{blHold}_{\mathtt{b}} && (5.13) \\
& \mathit{blHold}_{\mathtt{u},\mathtt{a}} \supset \mathit{blHold}_{\mathtt{a}} && (5.14)
\end{aligned}
$$

Propiedades espaciales sobre la posición de los bloques

Todas las observaciones declaradas son observaciones de un único elemento del sistema en el sentido de la Sección 4.4.4, y podemos por lo tanto aplicar las reglas allí definidas para obtener propiedades espaciales básicas.

$$
\begin{aligned}
& \mathit{blOnBl}_{\mathtt{a},\mathtt{b}} \wedge \mathit{empty}_{\mathtt{u}} \equiv \\
& \quad __\langle(\mathit{blOnBl}_{\mathtt{a},\mathtt{b}} \wedge \mathit{empty}_{\mathtt{u}}), (\neg\mathit{blHold}_{\mathtt{a}} \wedge \neg\mathit{blOnBl}_{\mathtt{a},\mathtt{b}} \wedge \neg\mathit{blTable}_{\mathtt{a}})\rangle && (5.15) \\
& \neg\mathit{blHold}_{\mathtt{u},\mathtt{a}} \equiv __\langle\neg\mathit{blHold}_{\mathtt{u},\mathtt{a}}, \neg\mathit{blHold}_{\mathtt{u},\mathtt{a}}\rangle && (5.16) \\
& \mathit{blOnBl}_{\mathtt{a},\mathtt{b}} \equiv __\langle\mathit{blOnBl}_{\mathtt{a},\mathtt{b}}, \mathit{true}\rangle && (5.17) \\
& \mathit{blOnBl}_{\mathtt{a},\mathtt{b}} \vee \mathit{blHold}_{\mathtt{a}} \equiv __\langle\mathit{blOnBl}_{\mathtt{a},\mathtt{b}} \vee \mathit{blHold}_{\mathtt{a}}, \mathit{true}\rangle && (5.18)
\end{aligned}
$$

5.4.3. Una propiedad temporal sobre la posición de los bloques

Si un bloque está bajo otro bloque entonces no puede ser cogido por ningún robot vacío:

$$blOnBl_{\mathtt{x},\mathtt{y}} \wedge empty_{\mathtt{r}} \supset \mathrm{AX}(\neg blHold_{\mathtt{r},\mathtt{y}}) \ .$$

Aplicando la primera regla de inferencia de la interfaz de la lógica temporal y VLRL (4.64) es suficiente con probar

$$blOnBl_{\mathtt{x},\mathtt{y}} \wedge empty_{\mathtt{r}} \supset \neg blHold_{\mathtt{r},\mathtt{y}} \ ,$$
$$\{blOnBl_{\mathtt{x},\mathtt{y}} \wedge empty_{\mathtt{r}} \supset [\alpha](\neg blHold_{\mathtt{r},\mathtt{y}}) \mid \alpha \text{ acción}\} \ .$$

Para la primera fórmula, obtenemos de (5.13) y (5.14)

$$blOnBl_{\mathtt{x},\mathtt{y}} \wedge empty_{\mathtt{r}} \supset blOnBl_{\mathtt{x},\mathtt{y}} \supset \neg blHold_{\mathtt{r},\mathtt{y}} \ .$$

Para la segunda fórmula, usando la inducción estructural sobre las acciones, debemos probar:

- constantes (las demostraciones son inmediatas):
  1. $\mathtt{<_:_|>}_{\mathtt{x1,x2}}\langle\rangle \supset (blOnBl_{\mathtt{x},\mathtt{y}} \wedge empty_{\mathtt{r}}) \supset [[\mathtt{<_:_|>}_{\mathtt{x1,x2}}]](\neg blHold_{\mathtt{r},\mathtt{y}})$
  2. $\mathtt{<_:_|_>}_{\mathtt{x1,x2,x3}}\langle\rangle \supset (blOnBl_{\mathtt{x},\mathtt{y}} \wedge empty_{\mathtt{r}}) \supset$ $[[\mathtt{<_:_|_>}_{\mathtt{x1,x2,x3}}]](\neg blHold_{\mathtt{r},\mathtt{y}})$
  3. $\mathtt{none}\langle\rangle \supset (blOnBl_{\mathtt{x},\mathtt{y}} \wedge empty_{\mathtt{r}}) \supset [[\mathtt{none}]](\neg blHold_{\mathtt{r},\mathtt{y}})$
  4. $\mathtt{pickup}_{\mathtt{u},\mathtt{a}}\langle\rangle \supset (blOnBl_{\mathtt{x},\mathtt{y}} \wedge empty_{\mathtt{r}}) \supset [[\mathtt{pickup}_{\mathtt{u},\mathtt{a}}]](\neg blHold_{\mathtt{r},\mathtt{y}})$
  5. $\mathtt{putdown}_{\mathtt{u},\mathtt{a}}\langle\rangle \supset (blOnBl_{\mathtt{x},\mathtt{y}} \wedge empty_{\mathtt{r}}) \supset [[\mathtt{putdown}_{\mathtt{u},\mathtt{a}}]](\neg blHold_{\mathtt{r},\mathtt{y}})$
  6. $\mathtt{unstack}_{\mathtt{u},\mathtt{a},\mathtt{b}}\langle\rangle \supset (blOnBl_{\mathtt{x},\mathtt{y}} \wedge empty_{\mathtt{r}}) \supset [[\mathtt{unstack}_{\mathtt{u},\mathtt{a},\mathtt{b}}]](\neg blHold_{\mathtt{r},\mathtt{y}})$
  7. $\mathtt{stack}_{\mathtt{u},\mathtt{a},\mathtt{b}}\langle\rangle \supset (blOnBl_{\mathtt{x},\mathtt{y}} \wedge empty_{\mathtt{r}}) \supset [[\mathtt{stack}_{\mathtt{u},\mathtt{a},\mathtt{b}}]](\neg blHold_{\mathtt{r},\mathtt{y}})$.

- reglas:
  8. $(blOnBl_{\mathtt{x},\mathtt{y}} \wedge empty_{\mathtt{r}}) \supset [\mathtt{pickup}(\mathtt{u},\mathtt{z})](\neg blHold_{\mathtt{r},\mathtt{y}})$.
     Aplicar las propiedades (5.7), (5.13), (5.14) y (4.62) para obtener el resultado.
  9. $(blOnBl_{\mathtt{x},\mathtt{y}} \wedge empty_{\mathtt{r}}) \supset [\mathtt{putdown}(\mathtt{u},\mathtt{z})](\neg blHold_{\mathtt{r},\mathtt{y}})$.
     Aplicar las propiedades (5.8), (5.13), (5.14) y (4.62) para obtener el resultado.
  10. $(blOnBl_{\mathtt{x},\mathtt{y}} \wedge empty_{\mathtt{r}}) \supset [\mathtt{unstack}(\mathtt{u},\mathtt{z1},\mathtt{z2})](\neg blHold_{\mathtt{r},\mathtt{y}})$.
      Distinguimos dos casos:
      (i) Si $blClear_{\mathtt{x}}$, entonces aplicar (5.9), y después (5.12) y (4.62) para obtener el resultado.
      (ii) Si $\neg blClear_{\mathtt{x}}$, entonces aplicar el contrarrecíproco de (5.11) y (4.62).
  11. $(blOnBl_{\mathtt{x},\mathtt{y}} \wedge empty_{\mathtt{r}}) \supset [\mathtt{stack}(\mathtt{u},\mathtt{z1},\mathtt{z2})](\neg blHold_{\mathtt{r},\mathtt{y}})$.
      Aplicar las propiedades (5.10), (5.13), (5.14) y (4.62) para obtener el resultado.

- operaciones sobre estados (en este caso, solo $__$):

12. $((\mathit{blOnBl}_{\mathtt{x,y}} \wedge \mathit{empty}_{\mathtt{r}}) \supset [\alpha_1](\neg \mathit{blHold}_{\mathtt{r,y}})),$
$((\mathit{blOnBl}_{\mathtt{x,y}} \wedge \mathit{empty}_{\mathtt{r}}) \supset [\alpha_2](\neg \mathit{blHold}_{\mathtt{r,y}}))$
$\vdash __\langle \mathit{true}, \mathit{true}\rangle \supset (\mathit{blOnBl}_{\mathtt{x,y}} \wedge \mathit{empty}_{\mathtt{r}}) \supset [__(\alpha_1, \alpha_2)](\neg \mathit{blHold}_{\mathtt{r,y}})\ .$

Usamos la siguiente propiedad modal de las observaciones, que dice que si un bloque no existe, después de la ejecución de cualquier transición tampoco existirá.

$$\neg \mathit{blHold}_{\mathtt{a}} \wedge \neg \mathit{blOnBl}_{\mathtt{a,b}} \wedge \neg \mathit{blTable}_{\mathtt{a}} \supset [\alpha](\neg \mathit{blHold}_{\mathtt{a}} \wedge \neg \mathit{blOnBl}_{\mathtt{a,b}} \wedge \neg \mathit{blTable}_{\mathtt{a}})\ .$$

Aplicando la propiedad derivada de las observaciones (5.14) y la lógica proposicional obtenemos:

$$\neg \mathit{blHold}_{\mathtt{a}} \wedge \neg \mathit{blOnBl}_{\mathtt{a,b}} \wedge \neg \mathit{blTable}_{\mathtt{a}} \supset [\alpha](\neg \mathit{blHold}_{\mathtt{u,a}})\ .$$

Combinamos este resultado con la primera hipótesis:

$$(\mathit{blOnBl}_{\mathtt{x,y}} \wedge \mathit{empty}_{\mathtt{r}}) \vee (\neg \mathit{blHold}_{\mathtt{y}} \wedge \neg \mathit{blOnBl}_{\mathtt{y,z}} \wedge \neg \mathit{blTable}_{\mathtt{y}}) \supset [\alpha_1](\neg \mathit{blHold}_{\mathtt{r,y}})\ .$$

De la segunda hipótesis, usando la regla de inferencia (4.33) de VLRL, derivamos

$$__\langle(\mathit{blOnBl}_{\mathtt{x,y}} \wedge \mathit{empty}_{\mathtt{r}}) \vee (\neg \mathit{blHold}_{\mathtt{y}} \wedge \neg \mathit{blOnBl}_{\mathtt{y,z}} \wedge \neg \mathit{blTable}_{\mathtt{y}}), (\mathit{blOnBl}_{\mathtt{x,y}} \wedge \mathit{empty}_{\mathtt{r}})\rangle \supset [__(\alpha_1, \alpha_2)](__\langle \neg \mathit{blHold}_{\mathtt{r,y}}, \neg \mathit{blHold}_{\mathtt{r,y}}\rangle)\ .$$

Usando la propiedad estructural (5.15):

$$(\mathit{blOnBl}_{\mathtt{x,y}} \wedge \mathit{empty}_{\mathtt{r}}) \supset [__(\alpha_1, \alpha_2)](__\langle \neg \mathit{blHold}_{\mathtt{r,y}}, \neg \mathit{blHold}_{\mathtt{r,y}}\rangle)\ .$$

Finalmente, usando la propiedad estructural (5.16) obtenemos el resultado.

5.4.4. Otra propiedad temporal sobre la posición de los bloques

Si un bloque está debajo de otro, permanecerá bajo él hasta que el bloque que está encima sea cogido por un robot:

$$\mathit{blOnBl}_{\mathtt{x,y}} \supset \mathrm{A}(\mathit{blOnBl}_{\mathtt{x,y}}\ \mathrm{W}\ \mathit{blHold}_{\mathtt{x}})\ .$$

Aplicando la segunda regla de inferencia de la interfaz de la lógica temporal con VLRL (4.65), es suficiente con probar

$$\{\mathit{blOnBl}_{\mathtt{x,y}} \wedge \neg \mathit{blHold}_{\mathtt{x}} \supset [\alpha](\mathit{blOnBl}_{\mathtt{x,y}} \vee \mathit{blHold}_{\mathtt{x}}) \mid \alpha \text{ acción}\}\ .$$

Usando inducción estructural, debemos probar:

- constantes (las demostraciones son inmediatas):

1. $\mathtt{<_:_|>}_{\mathtt{x1,x2}}\langle\rangle \supset (\mathit{blOnBl}_{\mathtt{x,y}} \wedge \neg \mathit{blHold}_{\mathtt{x}}) \supset$
$[[\mathtt{<_:_|>}_{\mathtt{x1,x2}}]](\mathit{blOnBl}_{\mathtt{x,y}} \vee \mathit{blHold}_{\mathtt{x}})$

2. $\mathtt{<_:_|_>}_{\mathtt{x1,x2,x3}}\langle\rangle \supset (\mathit{blOnBl}_{\mathtt{x,y}} \wedge \neg \mathit{blHold}_{\mathtt{x}}) \supset$
$[[\mathtt{<_:_|_>}_{\mathtt{x1,x2,x3}}]](\mathit{blOnBl}_{\mathtt{x,y}} \vee \mathit{blHold}_{\mathtt{x}})$
3. $\mathtt{none}\langle\rangle \supset (\mathit{blOnBl}_{\mathtt{x,y}} \wedge \neg \mathit{blHold}_{\mathtt{x}}) \supset [[\mathtt{none}]](\mathit{blOnBl}_{\mathtt{x,y}} \vee \mathit{blHold}_{\mathtt{x}})$
4. $\mathtt{pickup}_{\mathtt{u,a}}\langle\rangle \supset (\mathit{blOnBl}_{\mathtt{x,y}} \wedge \neg \mathit{blHold}_{\mathtt{x}}) \supset$
$[[\mathtt{pickup}_{\mathtt{u,a}}]](\mathit{blOnBl}_{\mathtt{x,y}} \vee \mathit{blHold}_{\mathtt{x}})$
5. $\mathtt{putdown}_{\mathtt{u,a}}\langle\rangle \supset (\mathit{blOnBl}_{\mathtt{x,y}} \wedge \neg \mathit{blHold}_{\mathtt{x}}) \supset$
$[[\mathtt{putdown}_{\mathtt{u,a}}]](\mathit{blOnBl}_{\mathtt{x,y}} \vee \mathit{blHold}_{\mathtt{x}})$
6. $\mathtt{unstack}_{\mathtt{u,a,b}}\langle\rangle \supset (\mathit{blOnBl}_{\mathtt{x,y}} \wedge \neg \mathit{blHold}_{\mathtt{x}}) \supset$
$[[\mathtt{unstack}_{\mathtt{u,a,b}}]](\mathit{blOnBl}_{\mathtt{x,y}} \vee \mathit{blHold}_{\mathtt{x}})$
7. $\mathtt{stack}_{\mathtt{u,a,b}}\langle\rangle \supset (\mathit{blOnBl}_{\mathtt{x,y}} \wedge \neg \mathit{blHold}_{\mathtt{x}}) \supset$
$[[\mathtt{stack}_{\mathtt{u,a,b}}]](\mathit{blOnBl}_{\mathtt{x,y}} \vee \mathit{blHold}_{\mathtt{x}})$.

- reglas:

8. $(\mathit{blOnBl}_{\mathtt{x,y}} \wedge \neg \mathit{blHold}_{\mathtt{x}}) \supset [\mathtt{pickup}(\mathtt{u},\mathtt{z})](\mathit{blOnBl}_{\mathtt{x,y}} \vee \mathit{blHold}_{\mathtt{x}})$.
La propiedad se deriva directamente de (5.7) y (4.62).
9. $(\mathit{blOnBl}_{\mathtt{x,y}} \wedge \neg \mathit{blHold}_{\mathtt{x}}) \supset [\mathtt{putdown}(\mathtt{u},\mathtt{z})](\mathit{blOnBl}_{\mathtt{x,y}} \vee \mathit{blHold}_{\mathtt{x}})$.
La propiedad se deriva directamente de (5.8) y (4.62).
10. $(\mathit{blOnBl}_{\mathtt{x,y}} \wedge \neg \mathit{blHold}_{\mathtt{x}}) \supset [\mathtt{unstack}(\mathtt{u},\mathtt{z1},\mathtt{z2})](\mathit{blOnBl}_{\mathtt{x,y}} \vee \mathit{blHold}_{\mathtt{x}})$.
Distinguimos dos casos:
(i) Si $\mathit{empty}_{\mathtt{u}} \wedge \mathit{blClear}_{\mathtt{x}}$, entonces basta aplicar (5.9), (5.14) y (4.62) para obtener el resultado.
(ii) Si $\neg \mathit{empty}_{\mathtt{u}} \vee \neg \mathit{blClear}_{\mathtt{x}}$, entonces basta aplicar el contrarrecíproco de (5.11) y (4.62).
11. $(\mathit{blOnBl}_{\mathtt{x,y}} \wedge \neg \mathit{blHold}_{\mathtt{x}}) \supset [\mathtt{stack}(\mathtt{u},\mathtt{z1},\mathtt{z2})](\mathit{blOnBl}_{\mathtt{x,y}} \vee \mathit{blHold}_{\mathtt{x}})$.
La propiedad se deriva directamente de (5.10) y (4.62).

- operaciones sobre estados:

12. $((\mathit{blOnBl}_{\mathtt{x,y}} \wedge \neg \mathit{blHold}_{\mathtt{x}}) \supset [\alpha_1](\mathit{blOnBl}_{\mathtt{x,y}} \vee \mathit{blHold}_{\mathtt{x}}),$
$(\mathit{blOnBl}_{\mathtt{x,y}} \wedge \neg \mathit{blHold}_{\mathtt{x}}) \supset [\alpha_2](\mathit{blOnBl}_{\mathtt{x,y}} \vee \mathit{blHold}_{\mathtt{x}})$
$\vdash __\langle \mathit{true}, \mathit{true}\rangle \supset (\mathit{blOnBl}_{\mathtt{x,y}} \wedge \neg \mathit{blHold}_{\mathtt{x}}) \supset$
$[__(\alpha_1, \alpha_2)](\mathit{blOnBl}_{\mathtt{x,y}} \vee \mathit{blHold}_{\mathtt{x}})$

De la primera hipótesis y la tautología $\mathit{true} \supset [\alpha]\mathit{true}$, usando la regla de inferencia (4.33) de VLRL, derivamos

$$__\langle(\mathit{blOnBl}_{\mathtt{x,y}} \wedge \neg \mathit{blHold}_{\mathtt{x}}), \mathit{true}\rangle \supset [__(\alpha_1, \alpha_2)](__\langle(\mathit{blOnBl}_{\mathtt{x,y}} \vee \mathit{blHold}_{\mathtt{x}}), \mathit{true}\rangle) \ .$$

Ahora nótese que, por la propiedad derivada de las observaciones (5.13) tenemos $\mathit{blOnBl}_{\mathtt{a,b}} \supset \neg \mathit{blHold}_{\mathtt{a}}$ por lo que $\mathit{blOnBl}_{\mathtt{a,b}} \equiv \mathit{blOnBl}_{\mathtt{a,b}} \wedge \neg \mathit{blHold}_{\mathtt{a}}$.

Entonces, usando la propiedad estructural (5.17),

$$\mathit{blOnBl}_{\mathtt{x,y}} \supset [__(\alpha_1, \alpha_2)](__\langle(\mathit{blOnBl}_{\mathtt{x,y}} \vee \mathit{blHold}_{\mathtt{x}}), \mathit{true}\rangle) \ .$$

Finalmente, obtenemos el resultado deseado usando la propiedad estructural (5.18).

5.5. El sistema Mobile Maude

En este ejemplo profundizamos en el uso de la modalidad espacial, especialmente en la definición de propiedades espaciales básicas utilizando las reglas definidas en la Sección 4.4.4.

El sistema Mobile Maude proporciona una especificación de un modelo de movilidad de objetos entre procesos (Sección 2.2.6). En Mobile Maude hay dos entidades principales: los *procesos* y los *objetos móviles* que pueden moverse de un proceso a otro e intercambian mensajes con otros objetos, en el mismo proceso o en otro diferente.

Con el fin de probar propiedades en VLRL definimos el *estado* como un multiconjunto de procesos y mensajes. Debemos restringir la noción general de configuración en sistemas orientados a objetos como multiconjunto de objetos y mensajes, ya que los objetos móviles se encuentran en los procesos y cuando se mueven de un proceso a otro se encapsulan en un mensaje, por lo que nunca aparecen al nivel de los procesos

5.5.1. Definición de las observaciones

Primero observamos el número de objetos móviles en el sistema. Para ello se define la observación *#mo* como el número total de objetos móviles en un *estado*

```
op #mo : State -> Nat .

var C : State .
var PI : Pid .
var S : Set[Mid] .
var M : Message .
var PO : Mid .

eq #mo(< PI : P | guests : S > C) = size(S) + #mo(C) .
eq #mo(none) = 0 .
eq #mo(go(PI,< PO : MO | >) C) = 1 + #mo(C) .
ceq #mo(M C) = #mo(C) if M =/= go(PI,< PO : MO | >) .
```

donde `none` representa el multiconjunto vacío y `go` es el mensaje que cambia el proceso en el que se está ejecutando un objeto. El mensaje es generado por el objeto móvil mediante la regla `message-out-move`, a continuación, la regla `go-proc` envía el mensaje al estado, y por último la regla `arrive-proc` permite que el proceso al que se dirige el objeto móvil lo capture (ver Sección 2.2.6).

Para observar si existe algún objeto activo en un estado definimos una observación *act*, y para observar si un objeto móvil dado está en modo `active` definimos la observación act_{O}. Esta noción de observación parametrizada, que ya se utilizó al observar la posición de los bloques en la Sección 5.4, se utiliza para considerar familias de observaciones, una por cada objeto móvil en el sistema:

```
op act : State -> Bool .
op act : State Mid -> Bool .
```

```
vars S : State .
var PI : Pid .
var C : Configuration .
var M : Message .
vars MI MI' : Mid .
var MD : Mode .

eq act(none) = false .
eq act(M S) = act(S) .
eq act(< PI : P | cf : C < MI : MO | mode : active > > S) = true .
eq act(< PI : P | cf : C < MI : MO | mode : idle > > S) =
   act(< PI : P | cf : C > S) .
eq act(< PI : P | cf : none > S) = act(S) .

eq act(none,MI) = false  .
eq act(M S,MI) = act(S, MI) .
eq act(< PI : P | cf : C < MI : MO | mode : MD > > S,MI) =
   (MD == active) or act(S,MI) .
ceq act(< PI : P | cf : C <MI' : MO | > S,MI) =
    act(< PI : P | cf : C > S,MI) if MI =/= MI' .
eq act(< PI : P | cf : none > S,MI) = act(S,MI) .
```

5.5.2. Propiedades espaciales básicas de Mobile Maude

Observaciones de tipo Nat

La observación $\#mo$ es de tipo Nat, por lo que se pueden utilizar las reglas (4.43) y (4.44) para derivar las siguientes propiedades espaciales básicas:

$$\begin{aligned}&\vdash __\langle \#mo = N_1, \#mo = N_2\rangle \supset \#mo = N_1 + N_2\\&\vdash __\langle true, true\rangle \supset (\#mo = N \supset\\&\quad \exists N_1, N_2.(__\langle \#mo = N_1, \#mo = N_2\rangle \wedge N = N_1 + N_2))\end{aligned} \tag{5.19}$$

donde la operación de concatenación __ es la única operación definida que construye estados, y se declara como asociativa, conmutativa y con identidad none.

En este caso la propiedad (5.19) no se verifica para todos los valores de N_1 y N_2 tales que $N = N_1 + N_2$, ya que los objetos móviles se encuentran en los procesos y el estado se descompone en procesos y mensajes. En la configuración[4]

```
< P1 : P | cf : < O1 : MO | > < O2 : MO | > < O3 : MO | > >
< P2 : P | cf : < O4 : MO | > < O5 : MO | > >
< P3 : P | cf : < O6 : MO | > < O7 : MO | > >
```

no es posible dividir el estado en dos subestados tales que: $__\langle \#mo = 3, \#mo = 4\rangle$ ya que un proceso no puede dividirse entre dos subestados.

[4]Omitimos los atributos de los objetos que no son relevantes para el ejemplo.

Configuraciones con estados vacíos

Dado que la operación de construcción de estados admite el estado vacío podemos derivar a partir de la regla (4.51)

$$\#mo < N \supset __\langle \#mo < N, true \rangle \ .$$

El recíproco no siempre se cumple. Podemos considerar, por ejemplo, la propiedad:

$$__\langle \#mo < N, true \rangle \ .$$

Claramente podemos tener más de N objetos móviles en el sistema completo, pues es suficiente con tenerlos en el segundo subestado.

Se comprueba también que (4.51) no se cumple si la función no permite estados vacíos. Tómese, por ejemplo, el estado formado por tres procesos y ningún mensaje definido previamente y supóngase que $_ | _$ es una operación asociativa y conmutativa y sin elemento neutro que concatena multiconjuntos, se tiene que $\#mo > 6$ se verifica, pero $_ | _ \langle \#mo > 6, true \rangle$ no.

Observaciones sobre un elemento único del sistema

Considérense ahora observaciones relativas a un único objeto del sistema, como por ejemplo $act_{\mathtt{o}}$ que observa el modo de un objeto móvil, identificado por su nombre `o`. Utilizando la regla (4.54) derivamos que si un objeto está activo en un subestado, entonces se encuentra activo en el estado global $__\langle act_{\mathtt{o}}, true \rangle \supset act_{\mathtt{o}}$.

La regla (4.55) nos dice que si un objeto se encuentra activo en un subestado, entonces no se encuentra en ningún otro subestado: $act_{\mathtt{o}} \supset __\langle act_{\mathtt{o}}, \neg act_{\mathtt{o}} \rangle$. Si tenemos propiedades diferentes en cada subestado, entonces la regla (4.57) nos permite derivar su conjunción: $__\langle act_{\mathtt{o}}, act_{\mathtt{o'}} \rangle \supset act_{\mathtt{o}} \wedge act_{\mathtt{o'}}$, pero el recíproco no se cumple siempre.

La regla (4.56) nos permite afirmar que si tenemos una propiedad no se cumple en un sistema entonces no se cumple en ningún subestado, suponiendo que la propiedad no es válida en el estado vacío; por ejemplo, $\neg act_{\mathtt{o}} \supset __\langle \neg act_{\mathtt{o}}, \neg act_{\mathtt{o}} \rangle$. De nuevo esta propiedad no se cumple en general si la observación no es única: $\neg \#mo < N \not\supset __\langle \neg \#mo < N, \neg \#mo < N \rangle$, aunque puede ser cierta en algún caso: $\neg act \supset __\langle \neg act, \neg act \rangle$.

5.5.3. Combinando acciones con propiedades espaciales

La combinación de la modalidad espacial y la modalidad de acción nos permite definir de forma natural propiedades sobre la ejecución concurrente de acciones.

Por ejemplo, supongamos que dos objetos móviles `o1` y `o2` comienzan a moverse de un proceso a otro.

La regla `message-out-move` encapsula los objetos en mensajes que son enviados por el sistema al proceso correspondiente. En lo que sigue se utiliza la abreviatura `mom` para esta regla de reescritura para mejorar la legibilidad de las propiedades.

```
rl [message-out-move] :
   < M : MO | s : '_&_[T,'go[T']], mode : active >
=> go(downPid(T'),
   < M : MO | s : ''_&_[T,'none'MsgSet], mode : idle >) .
```

El objeto es capturado por el proceso receptor al aplicar la regla `arrive-proc` (ver Sección 2.2.6).

Usando la regla de inferencia (4.38) podemos derivar que si las acciones pueden realizarse, entonces pueden realizarse de forma concurrente, y obtenemos la equivalencia usando la regla de inferencia (4.35):

$$__\langle \langle \texttt{mom(o1,T,P2)} \rangle \mathit{true}, \langle \texttt{mom(o2,T',P1)} \rangle \mathit{true} \rangle \equiv \\ \langle __(\texttt{mom(o1,T,P2)},\texttt{mom(o2,T',P1)}) \rangle \mathit{true}$$

donde `o`, `T` y `T'` son una sustitución dada de las variables de la regla de reescritura `mom`.

Usando la regla de inferencia (4.39) obtenemos que el resultado de ejecutar las acciones en cada subestado es el mismo que el resultado de ejecutarlas concurrentemente.

$$__\langle \langle \texttt{mom(o1,T,P2)} \rangle (\neg act_{\texttt{o1}}), \langle \texttt{mom(o2,T',P1)} \rangle (\neg act_{\texttt{o2}}) \rangle \supset \\ \langle __(\texttt{mom(o1,T,P2)}, \texttt{mom(o2,T,P1)}) \rangle __\langle \neg act_{\texttt{o1}}, \neg act_{\texttt{o2}} \rangle \ .$$

También podemos combinar *propiedades sobre las transiciones del sistema* como:

$$[\texttt{mom(o,T,P)}] \neg act_{\texttt{o}}$$

y derivar por medio de la regla de inferencia (4.33):

$$__\langle \mathit{true}, \mathit{true} \rangle \supset [__(\texttt{mom(o1,T,P2)}, \texttt{mom(o2,T,P1)})] __\langle \neg act_{\texttt{o1}}, \neg act_{\texttt{o2}} \rangle \ .$$

5.5.4. Uso de propiedades espaciales para probar propiedades temporales

Ahora podemos probar propiedades tales como *el número de objetos móviles en el sistema es siempre mayor o igual que cero*

$$\#mo \geq 0 \supset \mathrm{AG}(\#mo \geq 0) \ .$$

Por la definición de la conectiva temporal, $\mathrm{AG}\varphi \equiv \mathrm{A}(\varphi \mathrm{W} \mathit{false})$, y aplicando la regla de inferencia (4.65), es suficiente con probar

$$\{\#mo \geq 0 \supset [\alpha](\#mo \geq 0) \mid \alpha \text{ action}\} \ .$$

Como el número de acciones puede ser infinito, aplicamos inducción estructural. Entonces tenemos que probar que la propiedad se cumple para las acciones constantes, para las acciones derivadas de las etiquetas de las reglas, y para las acciones formadas a partir de operaciones sobre los estados, tal como se ha hecho, por ejemplo, en la Sección 5.1.2.

Ilustramos cómo probar la propiedad para las acciones formadas a partir de operaciones sobre estados, ya que es aquí donde se utilizan las propiedades espaciales.

La única operación de estado que tenemos es $__$, para la cual debemos probar que:

$$\begin{array}{l}(\#mo \geq 0 \supset [\alpha_1](\#mo \geq 0)), (\#mo \geq 0 \supset [\alpha_2](\#mo \geq 0)) \\ \quad \vdash __\langle true, true\rangle \supset (\#mo \geq 0 \supset [__(\alpha_1, \alpha_2)](\#mo \geq 0)) \ .\end{array}$$

Suponiendo las hipótesis y usando la regla de inferencia (4.33) de VLRL, tenemos

$$__\langle \#mo \geq 0, \#mo \geq 0\rangle \supset [__(\alpha_1, \alpha_2)]__\langle \#mo \geq 0, \#mo \geq 0\rangle \ .$$

Ahora se aplica la propiedad espacial básica (4.44) para derivar

$$\begin{array}{l}__\langle true, true\rangle \supset (\#mo \geq 0 \supset \\ \exists N_1, N_2 \geq 0.([__(\alpha_1, \alpha_2)]__\langle \#mo \geq 0, \#mo \geq 0\rangle \wedge N = N_1 + N_2) \ .\end{array}$$

Dado que N_1 y N_2 no son variables de estado podemos aplicar la regla de inferencia (4.15) para introducir la modalidad de acción en el término $N = N_1 + N_2$:

$$\begin{array}{l}__\langle true, true\rangle \supset (\#mo \geq 0 \supset \\ \exists N_1, N_2 \geq 0.([__(\alpha_1, \alpha_2)]__\langle \#mo \geq 0, \#mo \geq 0\rangle \wedge [__(\alpha_1, \alpha_2)](N = N_1 + N_2)) \ .\end{array}$$

Entonces, aplicando (4.26) y (4.16) obtenemos:

$$\begin{array}{l}__\langle true, true\rangle \supset (\#mo \geq 0 \supset \\ [__(\alpha_1, \alpha_2)]\exists N_1, N_2.__\langle \#mo \geq 0, \#mo \geq 0\rangle \wedge N = N_1 + N_2)) \ .\end{array}$$

Finalmente, aplicando la propiedad espacial básica (4.43) llegamos a la conclusión deseada:

$$__\langle true, true\rangle \supset (\#mo \geq 0 \supset [__(\alpha_1, \alpha_2)](\#mo \geq 0)) \ .$$

5.6. Conclusiones

La lógica VLRL permite la demostración formal de propiedades modales y temporales de sistemas, siendo especialmente adecuada en sistemas con configuraciones orientadas a objetos, ya que en estos casos la definición del estado del sistema viene dada por la propia configuración de este. Por otra parte, en las configuraciones orientadas a objetos, el operador espacial de la lógica resulta de gran utilidad al permitir definir propiedades solo sobre una serie de objetos y mensajes existentes en la configuración total.

La definición de propiedades está completamente definida por el usuario a través de las observaciones, como se muestra en todos los ejemplos. Esta definición además no es restrictiva, ya que permite no solo observaciones constantes cómo las utilizadas en los dos primeros ejemplos, sino también el observar objetos o mensajes concretos del sistema mediante el uso de variables en las observaciones, como ocurre en el segundo ejemplo sobre el mundo de los bloques.

Se comprueba además que la demostración de las propiedades invariantes de los sistemas es muy sistemática, lo que facilitará su automatización en el futuro.

Capítulo 6

Verificación semi-formal del protocolo IEEE 1394

En este capítulo mostramos cómo puede realizarse una prueba semi-formal de la corrección total, que incluye propiedades de seguridad y propiedades de vivacidad, de un algoritmo especificado en Maude.

Consideramos tres especificaciones, a diferentes niveles de abstracción, del protocolo de elección de líder del bus en serie multimedia IEEE 1394. La especificación del protocolo se da en el lenguaje Maude y se prueba su corrección razonando por inducción sobre las reglas de reescritura que el protocolo elige un único lider.

La descripción del protocolo en Maude se debe a Alberto Verdejo. En las secciones siguientes se muestra solamente el código necesario para realizar la prueba semi-formal del algoritmo; la descripción completa y una detallada explicación del mismo puede encontrarse en [VPMO02, Ver03].

El trabajo contenido en este capítulo se presentó por primera vez en el *Third International Workshop on Rewriting Logic and its Applications, WRLA 2000 (Kanazawa, Japón)*, y fue publicado en [VPMO00]. También se presentó en el *International Workshop on Application of Formal Methods to IEEE 1394 Standard (Berlín, Alemania)* [VPMO01b, VPMO01a] en 2001, evento que ha dado lugar a un volumen especial de la revista *Formal Aspects of Computing*, donde se ha incluido una versión extendida de nuestro trabajo [VPMO02].

6.1. Descripción informal del protocolo

El bus en serie multimedia IEEE 1394 [IEE95] permite transportar todas las formas de vídeo digitalizado y audio entre sistemas y dispositivos. El IEEE 1394 completo es complejo, abarcando diferentes subprotocolos, cada uno de los cuales tiene que ver con tareas diferentes (transferencia de datos entre nodos de la red, elección del líder, etc.). El estándar está descrito en capas y cada capa se divide en diferentes fases [IEE95]. En esta sección solo se trata la fase de identificación del árbol (elección del líder) de la capa física.

Informalmente, la fase de identificación del árbol del IEEE 1394 es un protocolo de elección de líder que tiene lugar después de un reajuste del bus en la red (es decir, cuando se añade o se elimina un nodo). Inmediatamente después del reajuste del bus, todos los nodos tiene la misma condición y solo conocen a que nodos están conectados; debe entonces seleccionarse un líder que sirva como gestor del bus para las otras fases del protocolo. El protocolo solo termina con éxito si la red inicial es conexa y sin ciclos.

Cada nodo lleva a cabo una serie de negociaciones con sus vecinos para establecer el sentido de la relación padre-hijo entre ellos. Concretamente, si un nodo tiene n conexiones entonces recibe peticiones "sé mi padre" de todas o de todas menos una de ellas.

Suponiendo que se han realizado n o $n-1$ peticiones, el nodo se mueve a una fase de reconocimiento, donde manda mensajes de acuse de recibo "tú eres mi hijo" a todos los nodos que mandaron una petición "sé mi padre" en la fase anterior. Cuando todos los acuses de recibo se han mandado, bien el nodo tiene n hijos y es el nodo raíz, o bien el nodo envía una petición "sé mi padre" en la conexión que todavía no se ha utilizado y espera un acuse de recibo del padre.

La comunicación entre los nodos es asíncrona, por lo que es posible que dos nodos pidan simultáneamente el uno al otro que sea su padre, dando lugar a un conflicto. Para resolverlo, cada nodo selecciona un valor booleano al azar. El valor booleano especifica una espera larga o corta antes de reenviar la petición "sé mi padre". Esto puede dar lugar a un nuevo conflicto, pero los principios de justicia garantizan que eventualmente un nodo llegará a ser la raíz.

6.2. Primera especificación del protocolo (con comunicación síncrona)

Comenzamos con una especificación simple del protocolo, sin consideraciones de tiempo, y donde la comunicación entre los nodos se supone síncrona, es decir, un mensaje se envía y recibe simultáneamente, y por lo tanto no hay necesidad de acuse de recibo y el conflicto no puede darse.

En esta primera especificación los nodos se representan mediante objetos de clase `Node` con los siguientes atributos:

- `neig : SetIden`, conjunto de identificadores de los nodos vecinos con los que el nodo todavía no se ha comunicado, y
- `done : Bool`, una bandera que se coloca cuando esta fase del protocolo ha finalizado para este nodo.

El módulo orientado a objetos que describe el protocolo es el siguiente.

```
(omod FIREWIRE-SYNC is
    protecting IDENTIFIERS .
    subsort Iden < Oid .
    class Node | neig : SetIden, done : Bool .
```

```
    msg leader_ : Iden -> Msg .

    vars I J : Iden .  var NEs : SetIden .
    rl [rec] :
       < I : Node | neig : J NEs, done : false >
       < J : Node | neig : I, done : false >
    => < I : Node | neig : NEs >  < J : Node | done : true > .
    rl [leader] :
       < I : Node | neig : empty, done : false >
    => < I : Node | done : true >  (leader I) .
endom)
```

6.3. Especificación con comunicación asíncrona y con tiempo

La especificación anterior es muy simple, pero no es una descripción exacta del protocolo real donde la comunicación es asíncrona debido al retardo introducido por los enlaces. Este retardo hace que sea necesario considerar una noción de tiempo en la especificación y aplicar por lo tanto los conceptos introducidos en la Sección 2.1.4.

En esta segunda especificación del protocolo [Ver03, VPMO02] cada nodo pasa por una serie de fases diferentes, como se explica en la Sección 6.1. Las fases están declaradas en el siguiente módulo:

```
(fmod PHASES is
    sort Phase .
    ops rec ack waitParent contention self : -> Phase .
endfm)
```

Cuando un nodo está en la fase `rec`, está recibiendo peticiones "sé mi padre" de sus vecinos. En la fase `ack`, el nodo envía acuses de recibo "tú eres mi hijo" a todos los nodos que enviaron una petición "sé mi padre" en la fase anterior. En la fase `waitParent`, el nodo espera el acuse de recibo de su padre. En la fase `contention`, el nodo espera un tiempo largo o corto antes de reenviar la petición "sé mi padre". El nodo está en la fase `self` o bien cuando ha sido seleccionado como líder, o bien cuando ha recibido un acuse de recibo de su padre.

Los atributos de la clase `Node`, definidos en el módulo `FIREWIRE-ASYNC` que extiende el módulo `TIMED-OO-SYSTEM`, módulo que especifica como el tiempo afecta a los objetos y mensajes (véase fin de la sección y explicaciones en [Ver03, VPMO02]), son los siguientes:

```
class Node | neig : SetIden, children : SetIden,
             phase : Phase, rootConDelay : DefTime .
```

El atributo `children` representa el conjunto de hijos que deben ser respondidos; `phase` representa la fase en que se encuentra el nodo, y `rootConDelay` es una alarma que se utiliza en la fase `contention`. El tipo `DefTime` extiende `Time`, que representa los valores

del tiempo [ÖM02], con una nueva constante `noTimeValue` utilizada cuando el reloj no está operativo.

```
sort DefTime .  subsort Time < DefTime .
op noTimeValue : -> DefTime .
```

Se introducen dos nuevos mensajes:

```
msg from_to_be‘my‘parent‘with‘delay_ : Iden Iden Time -> Msg .
msg from_to_acknowledgement‘with‘delay_ : Iden Iden Time -> Msg .
```

Por ejemplo, el mensaje `from I to J be my parent with delay T` indica que el nodo `I` ha enviado una petición “sé mi padre” al nodo `J`, y esta petición llegará al nodo `J` en `T` unidades de tiempo. Un mensaje con retardo 0 es *urgente*, en el sentido de que debe ser atendido antes de que pase ninguna unidad de tiempo. La operación `mte`, definida en el módulo `TIMED-OO-SYSTEM` asegura que se cumple este requisito.

La primera regla[1] afirma que un nodo `I` en la fase `rec`, con más de un vecino, puede recibir una petición “sé mi padre” con retardo 0 de su vecino `J`:

```
vars I J K : Iden .  vars NEs CHs : SetIden .
crl [rec] :
    (from J to I be my parent with delay 0)
    < I : Node | neig : J NEs, children : CHs, phase : rec >
=>  < I : Node | neig : NEs, children : J CHs >
if NEs =/= empty .
```

Cuando un nodo está en la fase `rec` y solo tiene una conexión sin utilizar, bien puede moverse a la fase `ack`, o bien puede recibir la última petición antes de ir a esa fase:

```
rl [recN-1] :
   < I : Node | neig : J, children : CHs, phase : rec >
=> < I : Node | phase : ack >    .

rl [recLeader] :
   (from J to I be my parent with delay 0)
   < I : Node | neig : J, children : CHs, phase : rec >
=> < I : Node | neig : empty, children : J CHs, phase : ack > .
```

En la fase de reconocimiento el nodo envía acuses de recibo “tú eres mi hijo” a todos los nodos que le enviaron peticiones “sé mi padre”:

```
rl [ack] :
   < I : Node | children : J CHs, phase : ack >
=> < I : Node | children : CHs >
   (from I to J acknowledgement with delay timeLink(I,J)) .
```

[1] Aunque por simplicidad presentamos reglas locales que reescriben términos de tipo `Configuration`, de hecho en la especificación completa se utilizan reglas globales que reescriben términos de tipo `ClockedSystem`.

La operación `timeLink : Iden Iden ->Time` representa una tabla con los valores de los retardos entre los nodos.

Cuando se han enviado todos los acuses de recibo, bien el conjunto `neig` está vacío y este es el nodo raíz, o bien envía una petición "sé mi padre" a la conexión que todavía no se ha utilizado. Obsérvese que los nodos hojas se mueven directamente a este punto.

```
rl [ackLeader] :
   < I : Node | neig : empty, children : empty, phase : ack >
=> < I : Node | phase : self >  (leader I) .

rl [ackParent] :
   < I : Node | neig : J, children : empty, phase : ack >
=> < I : Node | phase : waitParent >
   (from I to J be my parent with delay timeLink(I,J)) .

rl [wait1] :
   (from J to I acknowledgement with delay 0)
   < I : Node | neig : J, phase : waitParent >
=> < I : Node | phase : self >  .
```

Si se ha enviado una petición de padre a un nodo, el nodo espera su respuesta. Si en lugar de esta, llega una petición de padre del nodo, entonces ambos nodos entran en un conflicto.

```
rl [wait2] :
   (from J to I be my parent with delay 0)
   < I : Node | neig : J, phase : waitParent >
   < RAN : RandomNGen | seed : N >
=> < I : Node | phase : contention,
     rootConDelay : if (N % 2 == 0) then ROOT-CONT-FAST
                    else ROOT-CONT-SLOW fi >
   < RAN : RandomNGen | seed : random(N) > .

rl [contenReceive] :
   (from J to I be my parent with delay 0)
   < I : Node | neig : J, phase : contention >
=> < I : Node | neig : empty, children : J, phase : ack,
                rootConDelay : noTimeValue > .

rl [contenSend] :
   < I : Node | neig : J, phase : contention, rootConDelay : 0 >
=> < I : Node | phase : waitParent, rootConDelay : noTimeValue >
   (from I to J be my parent with delay timeLink(I,J)) .
```

Los objetos de clase `RandomNGen` son generadores de números pseudoaleatorios:

```
class RandomNGen | seed : MachineInt .
op random : MachineInt -> MachineInt .   *** siguiente numero aleatorio
var N : MachineInt .
eq random(N) = ((104 * N) + 7921) % 10609 .
```

Falta por definir cómo el tiempo afecta a los objetos y mensajes. Primero se define un módulo general:

```
(omod TIMED-OO-SYSTEM is protecting TIMEDOMAIN .
  sorts State ClockedSystem .
  subsort Configuration < State .
  op `_|_` : State Time -> ClockedSystem .
  op delta : Configuration Time -> Configuration .
  vars CF CF' : Configuration .  var T : Time .
  eq delta(none, T) = none .
  ceq delta(CF CF', T) = delta(CF, T) delta(CF', T)
                     if CF =/= none and CF' =/= none .
  op mte : Configuration -> TimeInf .
  eq mte(none) = INF .
  ceq mte(CF CF') = min(mte(CF), mte(CF'))
                    if CF =/= none and CF' =/= none .
endom)
```

luego se añaden las ecuaciones específicas de nuestro sistema:

```
vars T T' : Time .  var DT : DefTime .
eq delta(< I : Node | rootConDelay : DT >, T) =
  if DT == noTimeValue then < I : Node | >
  else < I : Node | rootConDelay : DT monus T > fi .
eq delta(< RAN : RandomNGen | >, T) = < RAN : RandomNGen | > .
eq delta(leader I, T) = leader I .
eq delta(from I to J be my parent with delay T, T') =
    from I to J be my parent with delay (T monus T') .
eq delta(from I to J acknowledgement with delay T, T') =
    from I to J acknowledgement with delay (T monus T') .

eq mte(< I : Node | neig : J K NEs, phase : rec >) = INF .
eq mte(< I : Node | neig : J, phase : rec >) = 0 .
eq mte(< I : Node | phase : ack >) = 0 .
eq mte(< I : Node | phase : waitParent >) = INF .
eq mte(< I : Node | phase : contention, rootConDelay : T >) = T .
eq mte(< I : Node | phase : self >) = INF .
eq mte(< RAN : RandomNGen | >) = INF .
eq mte( from I to J be my parent with delay T ) = T .
eq mte( from I to J acknowledgement with delay T ) = T .
eq mte( leader I ) = INF .
```

La regla `tick` permite pasar el tiempo si no existe ninguna regla que pueda aplicarse de forma inmediata:

```
var C : Configuration .
crl [tick] :  C | T  =>  delta(C, mte(C))  |  T plus mte(C)
             if mte(C) =/= INF and mte(C) =/= 0 .
```

Debido a la definición de la operación `mte`, esta regla sólo puede aplicarse cuando no se puede aplicar ninguna otra.

6.4. Tercera descripción del protocolo

Hay dos consideraciones de tiempo que no se han tenido en cuenta en la segunda descripción. La primera de ellas es si el valor `CONFIG-TIMEOUT` se ha sobrepasado. Esto indica que la red se ha configurado incorrectamente y contiene un ciclo y en este caso se debe generar un mensaje de error. La segunda cuestión trata el parámetro *raíz forzada*, `fr`. Normalmente un nodo puede pasar a la fase `ack` cuando se han realizado $n-1$ comunicaciones, sin embargo el parámetro `fr` fuerza al nodo a esperar un poco más, con la esperanza de que se realicen las n comunicaciones y el nodo se convierta en líder. Estas dos consideraciones afectan solo a la primera fase del protocolo.

Se añaden tres nuevos atributos a la clase `Node`:

```
class Node | neig : SetIden, children : SetIden,
             phase : Phase, rootConDelay : DefTime,
             CONFIG-TIMEOUTalarm : DefTime,
             fr : Bool, FORCE-ROOTalarm : DefTime .
```

El atributo `CONFIG-TIMEOUTalarm` es una *alarma* que se inicializa al valor constante `CONFIG-TIMEOUT`, que decrece cada vez que pasa el tiempo. Si alcanza el valor 0, la red tiene un ciclo y se genera el siguiente mensaje de error:

```
msg error : -> Msg .
```

y el atributo `phase` del nodo se pone al nuevo valor `error`.

El atributo `fr` se pone a `true` cuando el nodo pretende ser el líder. En este caso, el atributo `FORCE-ROOTalarm` se inicializa a la constante de tiempo `FRTIME`, que determina cuánto esperará el nodo antes de pasar a la siguiente fase.

Se añaden dos reglas, que controlan cuándo las alarmas notifican que se ha alcanzado el valor 0:

```
rl [error] :
   < I : Node | phase : rec, CONFIG-TIMEOUTalarm : 0 >
=> < I : Node | phase : error > error .

rl [stopAlarm] :
   < I : Node | phase : rec, fr : true, FORCE-ROOTalarm : 0 >
=> < I : Node | fr : false, FORCE-ROOTalarm : noTimeValue >  .
```

La regla `recN-1` se modifica ya que ahora un nodo en la fase `rec` pasa a la siguiente fase solo si su atributo `fr` tiene el valor `false`:

```
rl [recN-1] :
   < I : Node | neig : J, children : CHs, fr : false, phase : rec >
=> < I : Node | phase : ack, CONFIG-TIMEOUTalarm : noTimeValue,
                FORCE-ROOTalarm : noTimeValue >  .
```

Las alarmas se desactivan cuando se recibe la última petición "sé mi padre" con el nodo en la fase `rec`:

```
rl [recLeader] :
   (from J to I be my parent with delay 0)
   < I : Node | neig : J, children : CHs, phase : rec >
=> < I : Node | neig : empty, children : J CHs, phase : ack, fr : false,
                FORCE-ROOTalarm : noTimeValue,
                CONFIG-TIMEOUTalarm : noTimeValue >  .
```

El resto de la reglas no se modifica. Sin embargo, es necesario redefinir las operaciones `delta` y `mte`:

```
eq delta(< A : Node | phase : rec, CONFIG-TIMEOUTalarm : DT,
           FORCE-ROOTalarm : DT' >, T) =
   if DT == noTimeValue then
     (if DT' == noTimeValue then  < A : Node | >
      else  < A : Node | FORCE-ROOTalarm : DT' monus T >
      fi)
   else
     (if DT' == noTimeValue then
            < A : Node | CONFIG-TIMEOUTalarm : DT monus T >
      else  < A : Node | CONFIG-TIMEOUTalarm : DT monus T,
                         FORCE-ROOTalarm : DT' monus T >
      fi)
   fi .

eq delta(< A : Node | phase : contention, rootConDelay : DT >, T) =
  if DT == noTimeValue then < A : Node | >
  else < A : Node | rootConDelay : DT monus T > fi .

ceq delta(< A : Node | phase : PH >, T) = < A : Node | >
    if (PH == ack or PH == waitParent or PH == self or PH == error) .

eq delta( leader I, T ) = leader I .
eq delta( error, T ) = error .

eq delta( from I to J be my parent with delay T, T') =
    from I to J be my parent with delay (T monus T') .

eq delta( from I to J acknowledgement with delay T, T') =
    from I to J acknowledgement with delay (T monus T') .

eq mte( from I to J be my parent with delay T ) = T .
eq mte( from I to J acknowledgement with delay T ) = T .
eq mte( leader I ) = INF .
eq mte( error ) = INF .

eq mte(< A : Node | neig : I J NEs, phase : rec, CONFIG-TIMEOUTalarm : DT,
           FORCE-ROOTalarm : DT'  >) =
```

```
    if DT == noTimeValue then
      (if DT' == noTimeValue then INF  else DT'  fi)
    else
      (if DT' == noTimeValue then DT  else  min(DT, DT')  fi)
    fi .
  eq mte(< A : Node | neig : J, phase : rec, fr : true,
           FORCE-ROOTalarm : T >) = T .
  eq mte(< A : Node | neig : J, phase : rec, fr : false >) = 0 .

  eq mte(< A : Node | phase : ack >) = 0 .
  eq mte(< A : Node | phase : waitParent >) = INF .
  eq mte(< A : Node | phase : contention, rootConDelay : T >) = T .
  eq mte(< A : Node | phase : self >) = INF .
  eq mte(< A : Node | phase : error >) = INF .
```

6.5. Verificación de propiedades

Las propiedades deseables para este protocolo son que se elija un único líder, y que este líder se elija en algún momento. Para probarlas, definimos las *observaciones* que nos permiten formular estas propiedades a partir de una configuración del sistema y observamos los cambios producidos por las reglas de reescritura en las configuraciones hasta que se elige el líder.

6.5.1. Verificación de la especificación síncrona

Para el caso síncrono, definimos la siguiente observación:

- *nodes* es un conjunto de pares $\langle A; S\rangle$ donde A es el identificador de un nodo de la red y S es el conjunto de nodos tales que $B \in S$ si y solo si `<A : Node | neig : B NEs, done : false >` y `<B : Node | done : false >` aparecen en la configuración.

Si consideramos la segunda componente de cada par $\langle A; S\rangle$ como la lista de adyacencia de los nodos representados en la primera componente, entonces *nodes* representa una red (grafo dirigido) con los nodos de la configuración inicial para los cuales el protocolo todavía no ha finalizado.

Suponemos en la demostración que la red es inicialmente *correcta*, en el sentido de que el conjunto *nodes* representa una red simétrica (esto es, los enlaces son bidireccionales), conexa y sin ciclos. Se comprueba que si estas condiciones se satisfacen inicialmente, entonces siempre se satisfacen.

Las propiedades deseables del protocolo se derivan a partir de las siguientes:

1. Si hay al menos dos pares en el conjunto *nodes*, entonces se puede aplicar la regla `rec`. Sabemos que si $|nodes| \geq 2$, entonces existen `A` y `B` tales que `<A ; B >` está en *nodes*, ya que este es conexo y sin ciclos. Debido a que la red es simétrica, sabemos que existe `NEs` tal que `<B ; A NEs >` está en *nodes*. Por lo tanto, se puede aplicar la regla `rec`.

2. El cardinal de *nodes* siempre decrece en una unidad cuando se aplica una regla. La demostración es inmediata a partir de las reglas que modelan el sistema.

3. Como *nodes* es simétrico, si solo hay un par <A ; S > en *nodes*, su conjunto de *vecinos* S es empty.

4. Como *nodes* es conexo, puede haber como mucho un elemento en *nodes* tal que su conjunto de *vecinos* es empty.

Propiedad de seguridad: Se elige un único líder. Para que un nodo sea elegido líder el valor de su atributo neig debe ser empty. Por la propiedad (4) solo puede haber un elemento en *nodes* tal que su conjunto de vecinos sea empty y por la propiedad (1) dicho elemento puede no ser eliminado del conjunto *nodes* mientras existan otros elementos en el conjunto.

Propiedad de vivacidad: El líder se elige en algún momento. La propiedad se deriva por inducción sobre el número de elementos del conjunto *nodes*.

Si *nodes* tiene un único elemento, entonces por la propiedad (3) su conjunto de *vecinos* es empty. Se aplica entonces la regla de reescritura leader al nodo del conjunto *nodes* y se obtiene el líder.

Si *nodes* tiene más de un elemento entonces por la propiedad (1) se puede aplicar la regla rec y por la propiedad (2) el cardinal del conjunto *nodes* decrece.

6.5.2. Verificación de la segunda especificación

Extendemos el método anterior para probar la corrección de la especificación con tiempo. La idea principal es tener observaciones diferentes para el conjunto de nodos en cada fase y buscar conjuntos de nodos que representen redes simétricas, conexas y sin ciclos. Probaremos que si los conjuntos no están vacíos entonces puede producirse alguna acción, y el número de elementos en los conjuntos decrece hasta que todos los conjuntos están vacíos.

Dada una configuración de objetos y mensajes, consideramos las siguientes observaciones definidas mediante conjuntos de pares:

Rec_n : < A ; B NEs CHs > en $Rec_n, n > 0$ si

< A : Node | neig : B NEs, children : CHs, phase : rec >,

donde n es el número de identificadores de nodos en el atributo neig.

Ack_c : < A ; B CHs> en $Ack_c, c > 0$ si

< A : Node | neig : B, children : CHs, phase : ack > o

< A : Node | neig : empty, children : B CHs, phase : ack >

donde c es el número de identificadores de nodos en el atributo children.

Ack_0 : < A ; B > en Ack_0 si

< A : Node | neig : B, children : empty, phase : ack >,

< A ; empty > en Ack_0 si

< A : Node | neig : empty, children : empty, phase : ack >

Wait : < A ; B > en *Wait* si < A : Node | neig : B, phase : waitParent >

y no hay ningún mensaje from B to A acknowledgement with delay T en el sistema.

$Contention_t$: < A ; B > en $Contention_t$ si

< A : Node | neig : B, phase : contention, rootConDelay : T >

donde t es el valor del atributo rootConDelay.

Todos los conjuntos son disjuntos, ya que un nodo no puede estar en dos fases al mismo tiempo.

Propiedades de la red

El conjunto *Nodes* se define como:

$$Nodes = \bigcup_n Rec_n \cup \bigcup_c Ack_c \cup \bigcup_t Contention_t \cup Wait \ .$$

No hay dos pares en *Nodes* con la primera componente igual; entonces, si consideramos la segunda componente de cada par como la lista de adyacencia del nodo representado en la primera componente, *Nodes* representa una red (grafo dirigido), e inicialmente $Nodes = \bigcup_n Rec_n$, porque todos los otros subconjuntos están vacíos. Obsérvese que el conjunto que contiene solo el par <A ; empty > representa una red con un único nodo.

Si $\bigcup_n Rec_n$ representa, inicialmente, una red simétrica, conexa y sin ciclos, entonces *Nodes* representará siempre una red simétrica, conexa y sin ciclos.

***Nodes* es simétrico.** Si el conjunto *Nodes* representa inicialmente una red simétrica, en el sentido de que si hay un enlace entre los nodos A y B también hay un enlace entre los nodos B y A, entonces siempre representará una red simétrica. Se comprueba que cuando aplicamos una regla de reescritura, bien se elimina un par de un subconjunto pero es añadido en otro, o bien se eliminan tanto <A ; B NEs CHs> como <B ; A NEs' CHs'> del conjunto *Nodes*, o bien se le añaden ambos.

***Nodes* es conexo.** Probamos que, si *Nodes* representa inicialmente una red conexa, siempre representará una red conexa, comprobando que cuando se elimina un par del conjunto *Nodes*, es de la forma <A ; B > o <A ; empty > y esto significa que representa una hoja de la red, esto es, está conectado como mucho a otro nodo. Entonces, eliminando solo hojas, la red permanece conexa. Los estados en los que se han eliminado pares del conjunto *Nodes* se alcanzan aplicando alguna de las reglas de reescritura siguientes:

- ackLeader. Mirando al lado izquierdo de la regla, se requiere que el nodo esté en fase ack y los atributos neig y children sean ambos empty; por lo tanto, el par que representa la observación es de la forma <A ; empty >.

- `ack`. Cuando se aplica esta regla el mensaje `from A to B acknowledgement with delay T` se añade al sistema; entonces el par `<B ; A >` se elimina del conjunto *Nodes* ya que debe estar en el subconjunto *Wait*. Si se aplica esta regla, es porque `B` está en el atributo `children` de `A`, y esto significa que se había enviado previamente un mensaje "sé mi padre" desde `B` a `A`. Por lo tanto, el nodo `B` ha estado en la fase `waitParent`, y debe estar todavía en esta fase, ya que no puede haber más mensajes de acuse de recibo en el sistema. Así pues, cuando se aplica `ack`, paramos de observar el nodo `B`, porque estamos seguros de que la regla `wait1` será aplicada y el nodo `B` alcanzará la fase `self`.

***Nodes* no tiene ciclos.** Si *Nodes* representa inicialmente una red sin ciclos, entonces representará siempre una red sin ciclos. Dado que ninguna de las reglas de reescritura introduce nuevos pares en el conjunto *Nodes* y como al comienzo este representa una red sin ciclos, estos no se pueden crear.

Propiedad de seguridad: Se elige un único líder

Se prueba que se elige un único líder comprobando que si se aplica una regla de reescritura en el sistema, por lo menos se elimina un nodo de la red representada por el conjunto *Nodes*. Entonces si el algoritmo acaba, esto es, si el conjunto *Nodes* llega a estar vacío, al final la red representada por *Nodes* tendrá un único nodo que estará representado por un par de la forma `<A ; empty >`. Por lo tanto se podrá aplicar la regla `ackLeader` y se declarará un líder. No puede haber más de un líder, ya que la red es conexa.

Si el conjunto *Nodes* se vacía, tiene que haber un líder. Hay dos reglas que eliminan pares de *Nodes*:

- `ack`. Si observamos el estado que se alcanza cuando aplicamos esta regla, se ha eliminado un identificador de nodo `B` de la segunda componente del par `<A ; B CHs >`, y un par de la forma `<B ; A >`. En la red representada por *Nodes* esto significa que hemos eliminado el nodo `B` de la red.
- `ackLeader`. Si observamos el estado alcanzado cuando aplicamos esta regla, se ha eliminado un par de la forma `<A ; empty >` del conjunto *Nodes*, y esto significa que hemos eliminado el nodo `A` de la red.

En ambos casos se elimina un solo nodo de la red representada por *Nodes* cada vez que aplicamos una regla de reescritura. Si *Nodes* llega a estar vacío, al final la red debe tener solo un nodo que es de la forma `<A ; empty >`. Entonces podemos aplicar la regla `ackLeader` y se elige un líder.

Existe solo un líder. Dado que la red representada por *Nodes* es siempre conexa, solo puede haber un par de la forma `<A ; empty >` en *Nodes* si la red tiene solo un nodo. Como no añadimos nodos a la red, solo podemos tener un líder.

Propiedades de vivacidad

Probamos que si hay pares en *Nodes* entonces podemos aplicar alguna regla de reescritura en el sistema, y si aplicamos una regla, un cierto número positivo que depende de los pares en *Nodes* decrece, y se hace cero cuando ya no hay más pares en *Nodes*. Entonces *Nodes* debe llegar a ser vacío, lo que significa que el algoritmo ha terminado. La fase `contention` presenta algunos problemas, ya que la función en algunos casos no decrece cuando se aplican las reglas que tratan el conflicto. En esta parte probamos que el algoritmo termina usando la suposición de que estamos en un sistema justo y el conflicto no puede ocurrir siempre.

Propiedad 1: Si hay pares en *Nodes*, entonces se puede aplicar por lo menos una regla en el sistema.

Dado que la red representada por los pares en *Nodes* es acíclica, o bien la red tiene solo un nodo, o bien la red tiene al menos una hoja, esto es, hay un par de la forma `<A ; B CHs >` en *Nodes* con `B` el único valor en el atributo `neig` del nodo `A`. En el primer caso podemos aplicar la regla `ackLeader`. En el segundo caso, y como la red es simétrica, hay un par `<B ; A NEs CHs' >` en *Nodes*. La Tabla 6.1 muestra las reglas de reescritura que pueden aplicarse para cada par de nodos. Cuando no se presenta el segundo par, significa que no importa el subconjunto en el que se encuentra. En los casos de la regla de reescritura `tick`, queremos expresar que esta regla se puede aplicar si no hay ninguna otra regla que pueda aplicarse.

Propiedad 2: Un nodo puede entrar en la fase `contention` solo un número finito de veces.

Ahora probamos que en un sistema *justo*, y suponiendo que

$$\texttt{ROOT-CONT-FAST} \gg max_{\{I,J\}}(\texttt{timeLink(I,J)}) \tag{6.1}$$

$$\texttt{ROOT-CONT-SLOW} \gg max_{\{I,J\}}(\texttt{timeLink(I,J)}) \tag{6.2}$$

$$\texttt{ROOT-CONT-FAST} - \texttt{ROOT-CONT-SLOW} \gg max_{\{I,J\}}(\texttt{timeLink(I,J)}) \tag{6.3}$$

un nodo no puede estar siempre cambiando entre las fases `contention` y `waitParent` aplicando las reglas `wait2` y `contenSend`; o lo que es lo mismo, en algún momento se aplicará la regla de reescritura `contenReceive`.

Entendemos por *justicia* que todas las reglas de reescritura que pueden ser aplicadas serán aplicadas en algún momento, y que el generador de números aleatorios produce tanto números pares como impares y por lo tanto el atributo `rootConDelay` de un nodo en la fase `contention` puede ser `ROOT-CONT-FAST` o `ROOT-CONT-SLOW`. Las ecuaciones (6.1), (6.2), y (6.3) expresan que ambas constantes son mucho mayores que el retardo máximo de los enlaces entre los nodos, y que su diferencia es también mucho mayor [IEE95].

Las configuraciones que podemos tener cuando un conflicto tiene lugar son las siguientes:

1. Ambos nodos están en fase `contention`. Entonces,

Primer par	Segundo par	Regla de reescritura
`< A ; B CHs >` $\in Rec_1$		`recN-1`
`< A ; B CHs >` $\in Ack_c$		`ack`
`< A ; B >` $\in Ack_0$		`ackParent`
`< A ; B >` $\in Wait$	`< B ; A NEs CHs >` $\in Rec_n$ `A to B be my parent delay 0`	`rec`
	`< B ; A NEs CHs >` $\in Rec_n$ `A to B be my parent delay T`	`tick`
	`< B ; A CHs >` $\in Rec_1$	`recN-1`
	`< B ; A CHs >` $\in Ack_c$	`ack`
	`< B ; A CHs >` $\in Ack_0$	`ackParent`
	`< B ; A CHs >` $\in Wait$ `A to B be my parent delay 0`	`wait2`
	`< B ; A CHs >` $\in Wait$ `A to B be my parent delay T`	`tick`
	`< B ; A CHs >` $\in Wait$ `B to A be my parent delay 0`	`wait2`
	`< B ; A CHs >` $\in Wait$ `B to A be my parent delay T`	`tick`
	`< B ; A CHs >` $\in Contention_t$	`tick`
	`< B ; A CHs >` $\in Contention_0$	`contenSend`
`< A ; B >` $\in Contention_t$		`tick`
`< A ; B >` $\in Contention_0$		`contenSend`

Tabla 6.1: Reglas que pueden aplicarse en el sistema

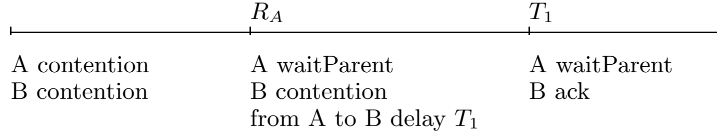

Figura 6.1: Conflicto 1 con $R_A < R_B$

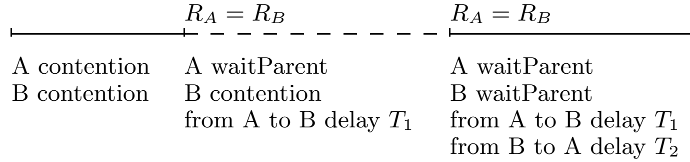

Figura 6.2: Conflicto 1 con $R_A = R_B$

- Si $R_A < R_B$, donde R_A es la constante `rootConDelay` seleccionada por el nodo `A`, el sistema alcanza el momento en el tiempo R_A y, por medio de la regla `contenSend`, `A` pasa a la fase `waitParent` y se envía un mensaje `from A to B be my parent with delay` T_1. Entonces por la suposición (6.3), ocurre el momento en el tiempo T_1 antes que R_B y este mensaje llega al nodo `B` cuando está todavía en la fase `contention`. Entonces el nodo `B` entrará en la fase `ack` por medio de la regla `contenReceive`. Esta situación corresponde a la Figura 6.1.

- Si $R_B < R_A$, la situación es simétrica a la anterior, y el nodo `A` llegará a la fase `ack`.

- Si $R_A = R_B$, entonces R_A y R_B ocurren de forma simultánea, y el sistema aplicará la regla `contenSend` a ambos nodos antes de que se consuma el tiempo del mensaje "sé mi padre" del primer nodo que aplica la regla `contenSend`. Esto significa que ambos nodos llegarán a la fase `waitParent` y los dos mensajes `from A to B be my parent with delay` T_1 y `from B to A be my parent with delay` T_2 estarán en el sistema. Ahora estamos en la configuración inicial 4 (ver más abajo) y ambos nodos irán de nuevo a la fase `contention`. Esta situación se muestra en la Figura 6.2, donde suponemos que el sistema aplica la regla `contenSend` antes al nodo `A`.

  En este caso, hacemos uso del hecho de que estamos en un sistema justo y las constantes seleccionadas por `A` y `B` serán en algún momento diferentes y por lo tanto no se producirá este caso siempre.

2. El nodo `A` está en la fase `contention`, el nodo `B` está en la fase `waitParent`, y hay un mensaje `from A to B be my parent with delay` T_1 en el sistema. Entonces, por las suposiciones (6.1) y (6.2), T_1 ocurre antes que R_A, y por medio de la regla `wait2`, `B` llega a la fase `contention`, y estamos en el caso 1. Ver Figura 6.3.

3. El nodo `A` está en fase `waitParent`, el nodo `B` está en fase `contention`, y hay un mensaje `from B to A be my parent with delay` T_2 en el sistema. Esta situación es simétrica al caso 2.

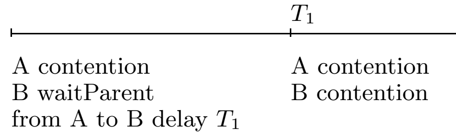

Figura 6.3: Conflicto 2

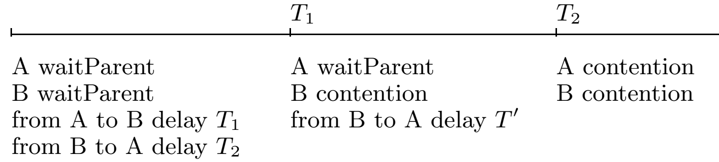

Figura 6.4: Conflicto 4

4. Ambos nodos están en la fase `waitParent`, y hay dos mensaje `from A to B be my parent with delay` $\mathtt{T}_1$ y `from B to A be my parent with delay` $\mathtt{T}_2$ en el sistema. Entonces, por las suposiciones (6.1) y (6.2), tanto T_1 como T_2 ocurren antes que ninguno de los R_A y R_B puedan tener lugar. Esto significa que tanto `A` como `B` llegan a la fase `contention`, y estamos de nuevo en el primer caso. Ver Figura 6.4.

Si un nodo sale de la fase `contention` por medio de la regla `contenReceive`, no volverá a la fase `contention` ya que estará en la fase `ack` sin vecinos. Entonces, las únicas reglas que se le pueden aplicar son primero `ack`, y después `ackLeader`.

Propiedad 3: Al aplicar las reglas decrece $f(\mathit{Configuration})$.

Sea N el número total de nodos en la red y T el retardo máximo de la tabla `timeLink`. Definimos:

$$\begin{aligned}
rec(I) &= \begin{cases} 5 * N * T * n & \text{si } \texttt{< I ; NEs >} \in Rec_n \\ 0 & \text{en otro caso} \end{cases} \\
ack(I) &= \begin{cases} 4 * T * (n+1) & \text{si } \texttt{< I ; J CHs >} \in Ack_n \\ 0 & \text{en otro caso} \end{cases} \\
wait(I) &= \begin{cases} 1 & \text{si } \texttt{< I ; J >} \in Wait \\ 0 & \text{en otro caso} \end{cases} \\
nm(C) &= \text{número de mensajes con tiempo en la configuración } C \\
times(C) &= \text{suma de tiempos de los mensajes en la configuración } C
\end{aligned}$$

Considérese la función

$$f(C) = \Big(\sum_{I \in \texttt{Node}} rec(I) + ack(I) + wait(I) \Big) + nm(C) + times(C) \ .$$

En la Tabla 6.2 mostramos el valor de la función f en una configuración y el valor de la misma función después de que hayamos aplicado una regla de reescritura en el sistema.

Antes de la regla	Regla	Después de la regla
$5 * N * T * n$	`rec`	$5 * N * T * (n-1) - 1$
$5 * N * T$	`recN-1`	$4 * T * (n+1) \leq 4 * T * N$
$5 * N * T$	`recLeader`	$4 * T * (n+1) - 1 < 4 * T * N$
$4 * T * (n+1)$	`ack`	$\leq 4 * T * n + T + 1$
$4 * T$	`ackLeader`	0
$4 * T$	`ackParent`	$\leq 2 + T$
1	`wait1`	0
2	`wait2`	0
$f(C)$	`tick`	$f(C) - \mathtt{mte}(C) * nm(C)$

Tabla 6.2: Valores de la función f en la segunda especificación

Todos los valores son relativos, en el sentido de que solo representan el valor del subestado que cambia cuando se aplica la regla de reescritura. Se observa que en todos los casos el valor de la función decrece.

Las reglas `contenReceive` y `contenSend` no se representan en la tabla porque no decrementan el valor de la función, sino que, por el contrario, lo incrementan. Esto no importa ya que hemos probado anteriormente que estas reglas no pueden ser aplicadas siempre para un par de nodos, y que si dos nodos resuelven su conflicto no tendrán otro.

Dado que $f(C) \geq 0$ y decrece cuando aplicamos las reglas de reescritura, entonces, aunque puede ser incrementado en una cantidad finita, concluimos que no podemos aplicar reglas de reescritura siempre en el sistema.

Corrección total

Dado que no podemos aplicar reglas siempre en el sistema (Propiedad 3), el conjunto *Nodes*, llegará a estar vacío (Propiedad 1), y si este conjunto se vacía debe haber un y solo un lider (Sección 6.5.2).

6.5.3. Verificación de la tercera especificación

Primero probamos que usando esta descripción del protocolo se detectan los ciclos. Después mostraremos cómo las nuevas reglas afectan a la prueba de corrección dada para la descripción asíncrona con tiempo sin detección de ciclos en la Sección 6.5.2.

Si hay un ciclo en la red, se genera un mensaje de error.

Propiedad 1: Si un nodo está en un ciclo, entonces no cambia de la fase `rec` hasta que se genera un mensaje de error.

Si un nodo está en un ciclo tiene por lo menos dos vecinos. Las reglas que cambian un nodo de la fase `rec` son `recN-1` y `recLeader`, que no pueden ser aplicadas ya que el nodo tiene más de un vecino, y la regla `error`, que genera el mensaje de error.

Propiedad 2: Si la red tiene un ciclo, entonces el atributo `CONFIG-TIMEOUTalarm` de algún nodo que está en el ciclo se pone a cero.

Dividimos el conjunto de nodos en dos subconjuntos: uno con los nodos que están en un ciclo, y el otro con los nodos que no están en un ciclo. Si un nodo no está en un ciclo, hay un número finito de reescrituras que pueden ocurrir antes de que alcance la fase `self`. Si un nodo está en un ciclo, se puede aplicar la regla `rec` como mucho tantas veces como el número de nodos en la red que están conectados a este nodo pero no están en el ciclo. Una vez que no se puede aplicar ninguna regla de reescritura diferente de `tick`, solo puede pasar el tiempo cambiando el atributo `CONFIG-TIMEOUTalarm`. Este atributo decrecerá en los nodos del ciclo hasta que alguno de ellos sea cero.

Propiedad 3: Si el atributo `CONFIG-TIMEOUTalarm` de un nodo se pone a cero y el nodo no cambia de la fase `rec`, el valor del atributo `CONFIG-TIMEOUTalarm` no se modifica.

Las reglas que cambian el valor del atributo `CONFIG-TIMEOUTalarm` son:

- `recN-1` y `recLeader`, que cambian la fase del nodo a la fase `ack`.
- `tick`, que decrece el valor del atributo. Pero esta regla no puede aplicarse si el atributo `CONFIG-TIMEOUTalarm` es cero, ya que en este caso la operación `mte` se evalúa a cero.
- `error`, que cambia la fase del nodo a la fase `error`.

Propiedad 4: Si hay un ciclo en la red entonces se genera un error.

Por las propiedades 2 y 3, hay un nodo cuyo atributo `CONFIG-TIMEOUTalarm` se pone a cero, y este valor no puede cambiar, y por la propiedad 1, el nodo debe estar en la fase `rec`. Mirando al lado izquierdo de la regla de reescritura `error` comprobamos que puede ser aplicada, y como suponemos que estamos en un sistema *justo* el mensaje de error se generará.

Corrección total

Probamos que, si no hay un error, se elige un único líder y que este líder en algún momento se elige. Podemos extender el método usado en la demostración anterior de la Sección 6.5.2 con el fin de que se consideren las reglas nuevas y las modificadas.

Consideramos las mismas observaciones que en el caso sin detección de ciclos, ya que aunque hemos introducido una nueva fase, `error`, esta es una fase *final* en el sentido de que una vez que un nodo la alcanza no existe ninguna regla de reescritura en el sistema que se le pueda aplicar.

Dado que suponemos que no hay ningún error, también tenemos que la red representada por el conjunto *Nodes* es simétrica, conexa, y sin ciclos.

La demostración de las propiedades de seguridad no cambia, ya que suponemos que no hay ningún error, y por lo tanto no introducimos ninguna regla de reescritura que elimine pares del conjunto *Nodes*.

Antes de la regla	Regla	Después de la regla
$5*N*T*n+1$	`error`	0
$5*N*T*n+1$	`stopAlarm`	$5*N*T*n$
$5*N*T$	`recN-1`	$4*T*(n+1)$
$5*N*T+1$	`recLeader`	$4*T*(n+1)$

Tabla 6.3: Valores de la función f en la tercera especificación

La demostración de las propiedades de vivacidad se modifica ligeramente. Primero, la propiedad 1 de la Sección 6.5.2 necesita considerar la nueva definición de la regla de reescritura `recN-1`. Si el primer par es `< A ; B Chs >` $\in Rec_1$ entonces tenemos los siguientes casos adicionales en la Tabla 6.1:

- si `fr` es `false`, entonces aplicar la regla `recN-1`;
- si `FORCE-ROOTalarm` es 0 y `fr` es `true`, entonces aplicar la regla `stopAlarm`;
- si `FORCE-ROOTalarm` es mayor que 0 y `fr` es `true`, entonces aplicar la regla `tick`.

La demostración de la propiedad 2 no cambia, ya que las nuevas reglas no afectan a la fase `contention`.

Para la propiedad 3, se muestra en la Tabla 6.3 cómo las nuevas reglas decrementan el valor de la función, con la siguiente definición de la función f para los casos *nm(C)* y *times(C)* de forma que se tengan en cuenta también objetos con valores de tiempo:

$$\begin{aligned} nm(C) &= \text{número de objetos y mensajes con el tiempo diferente a } \texttt{noTimeValue} \\ &\quad\ \text{en la configuración } C \\ times(C) &= \text{suma de tiempos en objetos y mensajes en la configuración } C \end{aligned}$$

Dado que se verifican las tres propiedades, la propiedad de vivacidad se satisface y por lo tanto tenemos la corrección total, como en el caso anterior.

6.6. Conclusiones

En este capítulo se ha mostrado un método semi-formal de verificar propiedades temporales de sistemas especificados en lógica de reescritura. Este método tiene la ventaja de poderse aplicar tanto a propiedades de seguridad como a propiedades de vivacidad y de permitir la demostración de propiedades más complejas que las que por el momento pueden verificarse con la lógica VLRL.

Sin embargo, la definición de las observaciones como conjuntos de objetos que cumplen una serie de propiedades ha sido realizada de forma específica para esta aplicación con el fin de demostrar la corrección total del sistema, basándonos en que el número de objetos en los conjuntos decrece y por lo tanto el algoritmo termina en algún momento, y no

se trata por lo tanto de un método aplicable a cualquier sistema o a cualquier tipo de propiedad. Más general es, en cambio, el tipo de demostración aplicado al probar que un nodo puede entrar en la fase `contention` solo un número finito de veces (Sección 6.5.2) ya que esta demostración se basa en comprobar los casos en que pueden aplicarse las reglas de reescritura.

Capítulo 7

Temas de trabajo futuro

La lógica modal VLRL propuesta en esta tesis completa el marco matemático de especificación de sistemas computacionales basado en lógica de reescritura. Este marco cubre las especificaciones basadas en la definición de un modelo del sistema a través del lenguaje Maude, mientras que las propiedades abstractas del segundo nivel de especificación se definen bien en la propia lógica modal VLRL, o bien en otra lógica modal o temporal utilizándose en este último caso la lógica VLRL en la demostración de las propiedades mediante interfaces que conectan ambas lógicas.

Este método de especificación es especialmente adecuado para configuraciones orientadas a objetos como se ha demostrado en los ejemplos desarrollados en el Capítulo 5, ya que Maude captura de forma clara la semántica de estos sistemas, mientras que las lógicas temporales y modales permiten expresar las propiedades dinámicas inherentes en ellos.

Existe todavía mucho trabajo por hacer tanto en el desarrollo del lenguaje Maude y especialmente Mobile Maude, como en el desarrollo de la lógica VLRL.

En el área del lenguaje Maude y en relación con los sistemas orientados a objetos, tal como se hizo notar en el Capítulo 3, sería conveniente refinar los mecanismos de creación de objetos, permitiendo asignar valores iniciales a los atributos que tengan como tipo una configuración, y los mecanismos de herencia de clases y herencia de módulos posibilitando el refinar atributos y métodos en las subclases, así como permitir la ocultación de información mejorando el mecanismo de encapsulación.

En cuanto al uso de reflexión en las especificaciones, actualmente cada usuario tiene que escribirse sus propias estrategias, bien partiendo de cero o bien adaptando y extendiendo las estrategias propuestas por otros investigadores, como ocurre en la especificación de la red de telecomunicaciones (Capítulo 3). Para solventar esta situación sería conveniente diseñar un lenguaje básico de estrategias para el usuario, integrado en el sistema Maude, que sea suficientemente expresivo para cubrir gran parte de los casos de uso esencial de estrategias que se precisan en la práctica. El que Maude sea reflexivo nos permitirá realizar un prototipo de implementación del lenguaje de estrategias en el propio metanivel de Maude, mediante técnicas de reflexión y metaprogramación.

Relacionado con el desarrollo de aplicaciones reflexivas, sería interesante complementar la aplicación desarrollada en el Capítulo 3 con la especificación de otras redes y la defini-

ción de una meta-red que las gestionara permitiendo el intercambio de datos entre ellas. Utilizando una definición reflexiva de las redes, esta aplicación requeriría el uso de dos niveles de reflexión. En esta línea sería interesante explotar más el uso de subconfiguraciones en la especificación de la red sin utilizar reflexión, de forma que la propia red pudiese controlar los distintos procesos. La comparación de estas opciones sería de interés en el desarrollo de especificaciones reflexivas. Todos estos temas tienen relación con el marco de la reflexión para objetos distribuidos propuesto recientemente por Meseguer y Talcott en [MT02].

En el área de la lógica VLRL es necesaria la implementación de herramientas automáticas de ayuda a la verificación de propiedades. Estas herramientas deben proporcionar:

- La derivación de propiedades VLRL básicas a partir de la signatura del programa. Para ello deben estudiarse las condiciones que nos permiten derivar las propiedades relativas a las transiciones del sistema y las propiedades sobre las condiciones que permiten realizar las acciones, a partir del lado derecho e izquierdo de las reglas de reescritura, respectivamente. Las propiedades espaciales básicas pueden ser derivadas a partir de las reglas dadas en la Sección 4.4.4. Sin embargo estas reglas no son completas y se requiere el estudio de nuevas reglas para observaciones más complejas.
- La demostración de propiedades VLRL utilizando las reglas de inferencia proporcionadas en esta tesis.
- La demostración de propiedades modales y temporales expresadas en otras lógicas. Para ello debe implementarse el interfaz que se proporciona en esta tesis así como estudiar nuevos interfaces.

Las herramientas que se desarrollen deben estar incluidas dentro del marco lógico que proporciona la lógica de reescritura. En este sentido, su implementación debe seguir la metodología general propuesta en [Cla00, CDE+99b]. Al igual que el asistente ITP (*Inductive Theorem Prover*), desarrollado por Clavel [Cla01] para demostrar propiedades inductivas de programas funcionales en Maude, las herramientas que automaticen la lógica VLRL, utilizarán tres niveles de reflexión. En el nivel objeto se sitúa el programa. En el meta-nivel se encuentran las reglas de inferencia de la lógica, que serán implementadas como reglas de reescritura en una teoría de Maude. Esta teoría, sin embargo, se define al meta-nivel, puesto que sus reglas implementan reglas de inferencia que actúan sobre teorías objeto. En el meta-meta-nivel se implementan las estrategias que controlan la aplicación de las reglas de inferencia.

Utilizar el mismo esquema de desarrollo que el ITP facilitará la integración de las dos herramientas y la reutilización, por ejemplo, del módulo de estrategias o del propio entorno de demostración.

En un aspecto más teórico sería conveniente avanzar en los estudios sobre la completitud de la lógica, formalizando los axiomas necesarios sobre el operador de acción y el operador espacial que hacen que la lógica sea completa respecto a los axiomas proporcionados en el Apéndice A.

Una vez se posea la capacidad de realizar demostraciones automáticas sería muy interesante proceder al desarrollo de aplicaciones prácticas que validen el potencial de la lógica

en la verificación de sistemas reales, especialmente el potencial del operador espacial en la declaración de propiedades de aplicaciones distribuidas. Entre estos sistemas destacan por su actualidad los sistemas de agentes y los de objetos móviles. En esta última línea tenemos los trabajos desarrollados por Durán y Verdejo entre otros [DV02, BLMO+02] para formalizar varios sistemas en el lenguaje Mobile Maude. La lógica VLRL puede ser utilizada para definir y probar no solo propiedades del propio sistema Mobile Maude, como se muestra en la Sección 5.5, sino también propiedades abstractas de los sistemas especificados en Mobile Maude.

El énfasis actual en sistemas distribuidos y móviles ha favorecido la introducción de nuevas lógicas modales y temporales (DTL, Mob-adtl, KLAIM, etc.), adaptadas a las nuevas características de estos sistemas, sean estas distribución, movilidad o ambas. Tiene interés, por lo tanto, estudiar una posible extensión de la lógica VLRL con operaciones extraidas de las lógicas modales y temporales para la especificación de sistemas móviles que hemos mencionado. En esta misma línea de investigación es interesante completar las reglas de interfaz de la Sección 4.6.1 de la lógica temporal de tiempo ramificado para cubrir los operadores AF y A_U_ lo que nos permitiría expresar propiedades de vivacidad.

En cuanto a las lógicas espaciales y de ambientes merece la pena realizar un estudio de la relación entre estas lógicas y la lógica VLRL, más detallado que la somera comparación realizada en la Sección 1.1.5. El estudio debe comenzar por definir una semántica operacional ejecutable del cálculo de ambientes en Maude, a través de la cual obtendremos los mecanismos adecuados para introducir dentro de Mobile Maude elementos del cálculo de ambientes. La inmersión opuesta resulta más problemática al ser Maude un marco en exceso flexible. Definiendo entonces las apropiadas reglas de interfaz de los operadores de la lógica de ambientes con VLRL podríamos realizar demostraciones de propiedades especificadas en lógica de ambientes en el propio marco de Maude.

Con el nombre de *model checking* en general se denominan diversas técnicas que persiguen la implementación eficiente de algoritmos que comprueban si un sistema especificado en un cierto lenguaje (no necesariamente ejecutable) de modelado satisface una propiedad definida mediante otro lenguaje de especificación (normalmente una versión de lógica temporal). Tales técnicas son más útiles en la practica si en el caso de respuesta negativa son capaces de devolver información en forma de contraejemplo que permita depurar el modelo original. Sería interesante estudiar la aplicación de técnicas de *model checking* para la lógica VLRL. En particular debe estudiarse la posible aplicación del procedimiento de decisión propuesto en la Sección 4.3, como base del procedimiento de *model checking*.

Recientemente se ha integrado en Maude 2.0 un *model checker* para comprobar propiedades expresadas en lógica temporal lineal sobre sistemas de reescritura en Maude. Algunas de las propiedades que hemos mostrado en esta tesis se pueden expresar en esta lógica y sería interesante comparar el uso del *model checker* para probarlas automáticamente.

En el caso de análisis de propiedades inductivas, como las demostradas en el Capítulo 6, que dan lugar a demostraciones por inducción estructural sobre las reglas que determinan la evolución dinámica del mismo, sería necesario contar con un demostrador de teoremas adecuado al tipo de inducción que se quiere realizar. El ITP mencionado anteriormente es precisamente un asistente especializado en la demostración de propiedades por inducción.

Sin embargo, si queremos tratar nuevas formas de inducción, como la inducción sobre reglas, resulta necesario extender la versión actual del asistente. Es necesario así mismo el estudio de más sistemas que faciliten las direcciones de extensión oportunas.

Apéndice A

Recopilación de las reglas de inferencia de la lógica VLRL

Axiomas de la lógica

$$\vdash [\alpha](\varphi \supset \psi) \supset ([\alpha]\varphi \supset [\alpha]\psi) \quad (4.11)$$

$$\vdash \langle\alpha\rangle\varphi \equiv [\alpha]\varphi \wedge \langle\alpha\rangle \mathit{true} \quad (4.12)$$

$$\vdash [\alpha](\varphi \wedge \psi) \supset ([\alpha]\varphi \wedge [\alpha]\psi) \quad (4.13)$$

$$\varphi \vdash [\alpha]\varphi \quad (4.14)$$

$$\vdash [\alpha]\varphi \equiv \neg\langle\alpha\rangle\varphi \quad (4.17)$$

$$\{(\varphi_i \supset \psi_i) \mid i \in \{1,\ldots,n\}\} \vdash f_{\overline{d}}[\varphi_1,\ldots,\varphi_n] \supset f_{\overline{d}}[\psi_1,\ldots,\psi_n] \quad (4.27)$$

$$\vdash f_{\overline{d}}[\varphi_1,\ldots,\mathit{true},\ldots,\varphi_n] \quad (4.40)$$

$$\vdash f_{\overline{d}}[\varphi_1,\ldots,\varphi_n] \equiv \neg f_{\overline{d}}\langle\neg\varphi_1,\ldots,\neg\varphi_n\rangle \quad (4.41)$$

$$\vdash \exists X.[\alpha]\varphi \equiv [\alpha]\exists X.\varphi \quad (4.16)$$

$$\varphi \vdash \varphi(\overline{w}/\overline{x})$$

Reglas de inferencia derivadas

$$\vdash t_1 = t_2 \supset [\alpha](t_1 = t_2) \quad (4.15)$$

$$\vdash f_{\overline{d}}\langle\overline{true}\rangle \supset (t_1 = t'_1 \wedge \ldots \wedge t_n = t'_n \supset f_{\overline{d}}\langle t_1 = t'_1, \ldots, t_n = t'_n\rangle) \quad (4.42)$$

$$\vdash [\alpha]true \quad (4.18)$$

$$(\varphi \supset \psi) \vdash ([\alpha]\varphi \supset [\alpha]\psi) \quad (4.19)$$

$$(\varphi \supset \psi) \vdash (\langle\alpha\rangle\varphi \supset \langle\alpha\rangle\psi) \quad (4.20)$$

$$\vdash (\langle\alpha\rangle\varphi \vee \langle\alpha\rangle\psi) \supset \langle\alpha\rangle(\varphi \vee \psi) \quad (4.21)$$

$$\vdash (\langle\alpha\rangle\varphi \wedge [\alpha]\psi) \supset \langle\alpha\rangle(\varphi \wedge \psi) \quad (4.22)$$

$$\vdash (\langle\alpha\rangle\varphi \wedge \langle\alpha\rangle\psi) \supset \langle\alpha\rangle(\varphi \wedge \psi) \quad (4.23)$$

$$\vdash [\alpha](\varphi \vee \psi) \supset (\langle\alpha\rangle\varphi \vee [\alpha]\psi) \quad (4.24)$$

$$\vdash (\langle\alpha\rangle(\varphi \vee \psi) \supset (\langle\alpha\rangle\varphi \vee \langle\alpha\rangle\psi) \quad (4.25)$$

$$\vdash ([\alpha]\varphi \wedge [\alpha]\psi) \supset [\alpha](\varphi \wedge \psi) \quad (4.26)$$

$$\{(\varphi_i \supset \psi_i) \mid i \in \{1, \ldots, n\}\} \vdash f_{\overline{d}}\langle\varphi_1, \ldots, \varphi_n\rangle \supset f_{\overline{d}}\langle\psi_1, \ldots, \psi_n\rangle \quad (4.28)$$

$$\vdash f_{\overline{d}}[\overline{false}] \supset f_{\overline{d}}[\varphi_1, \ldots, \varphi_n] \quad (4.29)$$

$$\varphi_1, \ldots, \varphi_n \vdash f_{\overline{d}}\langle\overline{true}\rangle \supset f_{\overline{d}}\langle\varphi_1, \ldots, \varphi_n\rangle \quad (4.30)$$

$$\vdash f_{\overline{d}}\langle\varphi_1, \ldots, \psi_1 \wedge \psi_2, \ldots \varphi_n\rangle \supset f_{\overline{d}}\langle\varphi_1, \ldots, \psi_1, \ldots, \varphi_n\rangle \wedge f_{\overline{d}}\langle\varphi_1, \ldots, \psi_2, \ldots, \varphi_n\rangle \quad (4.31)$$

$$\vdash f_{\overline{d}}\langle\varphi_1, \ldots, \psi_1, \ldots \varphi_n\rangle \supset f_{\overline{d}}\langle\varphi_1, \ldots, \psi_1 \vee \psi_2, \ldots, \varphi_n\rangle \quad (4.32)$$

$$\{\varphi_i \supset [\alpha_i]\psi_i \mid i \in \{1, \ldots, n\}\} \vdash f_{\overline{d}}\langle\varphi_1, \ldots, \varphi_n\rangle \supset [f_{\overline{d}}(\alpha_1, \ldots, \alpha_n)]f_{\overline{d}}\langle\psi_1, \ldots, \psi_n\rangle \quad (4.33)$$

$$\{\varphi_i \supset \langle\alpha_i\rangle true \mid i \in \{1, \ldots, n\}\} \vdash f_{\overline{d}}\langle\varphi_1, \ldots, \varphi_n\rangle \supset \langle f_{\overline{d}}(\alpha_1, \ldots, \alpha_n)\rangle true \quad (4.34)$$

$$\vdash \langle f_{\overline{d}}(\alpha_1, \ldots, \alpha_n)\rangle true \supset f_{\overline{d}}\langle\langle\alpha_1\rangle true, \ldots, \langle\alpha_n\rangle true\rangle \quad (4.35)$$

$$\{\varphi_i \supset \langle\alpha_i\rangle\psi_i \mid i \in \{1, \ldots, n\}\} \vdash f_{\overline{d}}\langle\varphi_1, \ldots, \varphi_n\rangle \supset \langle f_{\overline{d}}(\alpha_1, \ldots, \alpha_n)\rangle f_{\overline{d}}\langle\psi_1, \ldots, \psi_n\rangle \quad (4.36)$$

$$\vdash f_{\overline{d}}\langle[\alpha_1]\psi_1, \ldots, [\alpha_n]\psi_n\rangle \supset [f_{\overline{d}}(\alpha_1, \ldots, \alpha_n)]f_{\overline{d}}\langle\psi_1, \ldots, \psi_n\rangle \quad (4.37)$$

$$\vdash f_{\overline{d}}\langle\langle\alpha_1\rangle true, \ldots, \langle\alpha_n\rangle true\rangle \supset \langle f_{\overline{d}}(\alpha_1, \ldots, \alpha_n)\rangle true \quad (4.38)$$

$$\vdash f_{\overline{d}}\langle\langle\alpha_1\rangle\psi_1, \ldots, \langle\alpha_n\rangle\psi_n\rangle \supset \langle f_{\overline{d}}(\alpha_1, \ldots, \alpha_n)\rangle f_{\overline{d}}\langle\psi_1, \ldots, \psi_n\rangle \quad (4.39)$$

Reglas de derivación de propiedades espaciales básicas

$$\vdash f_{\overline{d}}\langle \texttt{obs} = N_1, \ldots, \texttt{obs} = N_n\rangle \supset \texttt{obs} = N_1 + \ldots + N_n \; * \tag{4.43}$$

$$\vdash f_{\overline{d}}\langle \overline{\mathit{true}}\rangle \supset (\texttt{obs} = N \supset \exists N_1, \ldots, N_n. \\ (f_{\overline{d}}\langle \texttt{obs} = N_1, \ldots, \texttt{obs} = N_n\rangle \wedge N = N_1 + \ldots + N_n)) \; * \tag{4.44}$$

$$\vdash f_{\overline{d}}\langle \texttt{obs} = S_1, \ldots, \texttt{obs} = S_n\rangle \supset \texttt{obs} = \bigcup_{i=1,\ldots,n} S_i \tag{4.47}$$

$$\vdash f_{\overline{d}}\langle \overline{\mathit{true}}\rangle \supset (\texttt{obs} = S \supset \exists S_1, \ldots, S_n. \\ (S = \bigcup_{i=1,\ldots,n} S_i \wedge f_{\overline{d}}\langle \texttt{obs} = S_1, \ldots, \texttt{obs} = S_n\rangle)) \tag{4.48}$$

$$\vdash f_{\overline{d}}\langle \texttt{obs} = V_1, \ldots, \texttt{obs} = V_n\rangle \supset \texttt{obs} = V_1 \vee \ldots \vee V_n \tag{4.49}$$

$$\vdash f_{\overline{d}}\langle \overline{\mathit{true}}\rangle \supset (\texttt{obs} = V \supset \exists V_1, \ldots, V_n. \\ (V = V_1 \vee \ldots \vee V_n \wedge f_{\overline{d}}\langle \texttt{obs} = V_1, \ldots, \texttt{obs} = V_n\rangle)) \tag{4.50}$$

$$\vdash \varphi \supset f_{\overline{d}}\langle \varphi, \mathit{true}, \ldots, \mathit{true}\rangle \tag{4.51}$$

$$\vdash f_{\overline{d}}\langle \varphi_1, \ldots, \varphi_n\rangle \equiv f_{\overline{d}}\langle \mathit{perm}(\overline{\varphi})\rangle \tag{4.52}$$

$$\vdash f_{\overline{d}}\langle f_{\overline{d}}\langle \psi_1, \ldots, \psi_n\rangle, \varphi_2, \ldots, \varphi_n\rangle \equiv f_{\overline{d}}\langle \psi_1, f_{\overline{d}}\langle \psi_2, \ldots, \psi_n, \varphi_2\rangle, \ldots, \varphi_n\rangle \\ \equiv \ldots, \equiv f_{\overline{d}}\langle \psi_1, \ldots, \psi_{n-1}, f_{\overline{d}}\langle \psi_n, \varphi_2, \ldots, \varphi_n\rangle\rangle \tag{4.53}$$

$$\vdash f_{\overline{d}}\langle \texttt{obs}, \mathit{true}, \ldots, \mathit{true}\rangle \supset \texttt{obs} \tag{4.54}$$

$$\vdash f_{\overline{d}}\langle \texttt{obs}_1, \ldots, \texttt{obs}_n\rangle \supset \texttt{obs}_1 \wedge \ldots \wedge \texttt{obs}_n \tag{4.57}$$

$$\vdash \texttt{obs} \supset f_{\overline{d}}\langle \texttt{obs}, \neg\texttt{obs}, \ldots, \neg\texttt{obs}\rangle \tag{4.55}$$

$$\vdash \neg\texttt{obs} \supset f_{\overline{d}}\langle \neg\texttt{obs}, \ldots, \neg\texttt{obs}\rangle \tag{4.56}$$

$$\vdash f_{\overline{d}}\langle \overline{\mathit{true}}\rangle \supset (\langle\alpha\rangle \mathit{true} \supset f_{\overline{d}}\langle \neg\langle\alpha\rangle \mathit{true}, \ldots, \neg\langle\alpha\rangle \mathit{true}\rangle) \; ** \tag{4.58}$$

$$\vdash f_{\overline{d}}\langle \langle\alpha\rangle \mathit{true}, \mathit{true}, \ldots, \mathit{true}\rangle \supset \neg\langle\alpha\rangle \mathit{true} \; ** \tag{4.59}$$

* Estos resultados se cumplen también para los operadores relacionales $\geq$, $>$, $\leq$, $<$.

** Sólo en configuraciones sin estados vacíos.

Bibliografía

[BAPM83] M. Ben-Ari, A. Pnueli y Z. Manna. The temporal logic of branching time. *Acta Informática.*, 20:207–226, 1983.

[BJM00] A. Bouhoula, J.-P. Jouannaud y J. Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236:35–132, 2000.

[BKKM02] P. Borovanský, C. Kirchner, H. Kirchner y P.-E. Moreau. ELAN from a rewriting logic point of view. *Theoretical Computer Science*, 285(2):155–185, 2002.

[BLMO+02] M. Bradley, L. Llana, N. Martí-Oliet, T. Robles, J. Salvachua y A. Verdejo. Transforming information in RDF to rewriting logic. En R. P. na, A. Herranz y J. J. Moreno, editores, *Segundas Jornadas sobre Programación y Lenguajes. (PROLE 2002)*, páginas 167–182. 2002.

[BS84] R. Bull y K. Segerberg. Basic modal logic. En D. Gabbay y F. Guenthner, editores, *Handbook of Philosophical Logic*, volumen II, páginas 1–88. D. Reidel Publishing Company, 1984.

[CC01] L. Caires y L. Cardelli. A spatial logic for concurrency (Part I). En N. Kobayashi y B. C. Pierce, editores, *Proceedings of the 10th Symposium on Theoretical Aspects of Computer Science (TACS 2001)*, volumen 2215 de *Lecture Notes in Computer Science*, páginas 1–30. Springer-Verlag, 2001.

[CC02] L. Caires y L. Cardelli. A spatial logic for concurrency (Part II). Informe técnico 3/2002/DI/PLM/FCTUNL, DI/PLM FCT Universidade Nova de Lisboa, 2002.

[CDE+98] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet y J. Meseguer. Metalevel computation in Maude. En C. Kirchner y H. Kirchner, editores, *Proceedings Second International Workshop on Rewriting Logic and its Applications, WRLA'98, Pont-à-Mousson, France, September 1–4, 1998*, volumen 15 de *Electronic Notes in Theoretical Computer Science*, páginas 3–24. Elsevier, 1998. http://www.elsevier.nl/locate/entcs/volume15.html.

[CDE+99a] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer y J. F. Quesada. Maude: Specification and programming in

rewriting logic, enero 1999. Manual distribuido como documentación del sistema Maude, Computer Science Laboratory, SRI International. http://maude.csl.sri.com/manual.

[CDE+99b] M. Clavel, F. Durán, S. Eker, J. Meseguer y M.-O. Stehr. Maude as a formal meta-tool. En J. M. Wing, J. Woodcock y J. Davies, editores, *FM'99 — Formal Methods, World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 20–24, 1999 Proceedings, Volume II*, volumen 1709 de *Lecture Notes in Computer Science*, páginas 1684–1703. Springer-Verlag, 1999.

[CDE+00a] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer y J. F. Quesada. A Maude tutorial, marzo 2000. Tutorial distribuido como documentación del sistema Maude, Computer Science Laboratory, SRI International. Presentado en *European Joint Conference on Theory and Practice of Software, ETAPS 2000*, Berlín, Alemania, 25 Marzo , 2000. http://maude.csl.sri.com/tutorial.

[CDE+00b] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer y J. F. Quesada. Towards Maude 2.0. En K. Futatsugi, editor, *Proceedings Third International Workshop on Rewriting Logic and its Applications, WRLA 2000, Kanazawa, Japan, September 18–20, 2000*, volumen 36 de *Electronic Notes in Theoretical Computer Science*, páginas 297–318. Elsevier, 2000. http://www.elsevier.nl/locate/entcs/volume36.html.

[CDE+02] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer y J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002.

[CE81] E. M. Clarke y E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. En *Proc. Workshop on Logics of Programs*, volumen 131 de *Lecture Notes in Computer Science*, páginas 52–71. Springer-Verlag, 1981.

[CELM96] M. Clavel, S. Eker, P. Lincoln y J. Meseguer. Principles of Maude. En J. Meseguer, editor, *Proceedings First International Workshop on Rewriting Logic and its Applications, WRLA'96, Asilomar, California, September 3–6, 1996*, volumen 4 de *Electronic Notes in Theoretical Computer Science*, páginas 65–89. Elsevier, 1996. http://www.elsevier.nl/locate/entcs/volume4.html.

[CG98] L. Cardelli y A. D. Gordon. Mobile ambients. En M. Nivat, editor, *Foundations of Software Science and Computational Structures, First International Conference, FoSSaCS'98, Held as Part of the ETAPS'98, Lisbon, Portugal, March 28 – April 4, 1998, Proceedings*, volumen 1378 de *Lecture Notes in Computer Science*, páginas 140–155. Springer-Verlag, 1998. http://www.luca.demon.co.uk/Bibliography.html.

[CG00] L. Cardelli y A. D. Gordon. Anytime, anywhere: Modal logics for mobile ambients. En *27th ACM Symposium on Principles of Programming Languages*, páginas 365–377. ACM, 2000.

[Cla00] M. Clavel. *Reflection in Rewriting Logic: Metalogical Foundations and Metaprogramming Applications*. CSLI Publications, 2000.

[Cla01] M. Clavel. The ITP tool. En A. Neponucemo, J. F. Quesada y F. J. Salguero, editores, *Logic, Language and Information. Proceedings of the First Workshop on Logic and Languages*, páginas 55–62. Kronos, 2001.

[CM88] K. M. Chandy y J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[CM96a] M. Clavel y J. Meseguer. Axiomatizing reflective logics and languages. En G. Kiczales, editor, *Proceedings of Reflection'96, San Francisco, California, April 1996*, páginas 263–288. 1996.

[CM96b] M. Clavel y J. Meseguer. Reflection and strategies in rewriting logic. En J. Meseguer, editor, *Proceedings First International Workshop on Rewriting Logic and its Applications, WRLA'96, Asilomar, California, September 3–6, 1996*, volumen 4 de *Electronic Notes in Theoretical Computer Science*, páginas 125–147. Elsevier, 1996. http://www.elsevier.nl/locate/entcs/volume4.html.

[CMP02] M. Clavel, J. Meseguer y M. Palomino. Reflection in membership equational logic, Horn logic with equality, and rewriting logic. En U. Montanari, editor, *Proceedings Fourth International Workshop on Rewriting Logic and its Applications, WRLA 2002, Pisa, Italy, September 19–21, 2002*, volumen 71 de *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002. http://www.elsevier.nl/locate/entcs/volume71.html.

[DELM00] F. Durán, S. Eker, P. Lincoln y J. Meseguer. Principles of Mobile Maude. En D. Kotz y F. Mattern, editores, *Agent Systems, Mobile Agents, and Applications, Second International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents, ASA/MA 2000, Zurich, Switzerland, September 13–15, 2000, Proceedings*, volumen 1882 de *Lecture Notes in Computer Science*, páginas 73–85. Springer-Verlag, 2000.

[Den98] G. Denker. From rewrite theories to temporal logic theories. En C. Kirchner y H. Kirchner, editores, *Proceedings Second International Workshop on Rewriting Logic and its Applications, WRLA'98, Pont-à-Mousson, France, September 1–4, 1998*, volumen 15 de *Electronic Notes in Theoretical Computer Science*, páginas 273–294. Elsevier, 1998. http://www.elsevier.nl/locate/entcs/volume15.html.

[DF02] R. Diaconescu y K. Futatsugi. Logical foundations of CafeOBJ. *Theoretical Computer Science*, 285(2):289–318, 2002.

[Dur99] F. Durán. *A Reflective Module Algebra with Applications to the Maude Language*. Tesis Doctoral, Universidad de Málaga, España, junio 1999. http://maude.csl.sri.com/papers.

[DV02] F. Durán y A. Verdejo. A conference reviewing system in Mobile Maude. En F. Gadducci y U. Montanari, editores, *Proceedings Fourth International Workshop on Rewriting Logic and its Applications, WRLA 2002, Pisa, Italy, September 19–21, 2002*, volumen 71 de *Electronic Notes in Theoretical Computer Science*, páginas 79–95. Elsevier, 2002. http://www.elsevier.nl/locate/entcs/volume71.html.

[ECSD98] H.-D. Ehrich, C. Caleiro, A. Sernadas y G. Denker. Logics for specifying concurrent information systems. En J. Chomicki y G. Saake, editores, *Logics for Databases and Information Systems*, páginas 167–198. Kluwer Academic Publishers, 1998.

[EH85] E. A. Emerson y J. Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. *Journal of Computer and System Sciences*, 30(1), febrero 1985.

[EH86] E. A. Emerson y J. Y. Halpern. "Sometimes" and "not never" re-revisited: on branching versus linear time temporal logic. *Journal of the ACM*, 33(1):151–178, 1986.

[Ehr99] H.-D. Ehrich. Object specification. En E. Astesiano, H.-J. Kreowski y B. Krieg-Brückner, editores, *Algebraic Foundations of Systems Specification*, IFIP State-of-the-Art Reports, páginas 435–466. Springer-Verlag, 1999.

[Eme90] E. A. Emerson. Temporal and modal logic. En J. V. Leeuwen, editor, *Handbook of theoretical computer science*, capítulo 16, páginas 997–1072. Elsevier Science Publishers B.V., 1990.

[FB86] M. Fisher y H. Barringer. Program logics - A short survey. Informe técnico UMCS-86-11-1, Department of Computer Science, University of Manchester, 1986.

[Fis92] M. Fisher. A model checker for linear time temporal logic. *Formal Aspects of Computing*, 4:299–319, 1992.

[FM91a] J. L. Fiadeiro y T. Maibaum. Describing, structuring, and implementing objects. En J. W. de Bakker, W. P. de Roever y G. Rozenberg, editores, *Foundations of Object-Oriented Languajes*, volumen 489 de *Lecture Notes in Computer Science*, páginas 274–310. Springer-Verlag, 1991.

[FM91b] J. L. Fiadeiro y T. Maibaum. Temporal reasoning over deontic specifications. *Journal of Logic and Computation*, 1(3):357–395, 1991.

[FM91c] J. L. Fiadeiro y T. Maibaum. Towards object calculi. En G. Saake y A. Sernadas, editores, *Proc. Workshop ISCORE'91, Information Systems: Correctness and Reusability*, páginas 129–178. Informatik-Bericht 91-03, Technische Universität Braunschweig, 1991.

[FM92] J. L. Fiadeiro y T. Maibaum. Temporal theories as modularisation units for concurrent system specification. *Formal Aspects of Computing*, 4(3):239–272, 1992.

[FM97] J. L. Fiadeiro y T. Maibaum. Categorical semantics of parallel program design. *Science of Computer Programming*, 28(2–3):111–138, 1997.

[FMMO+00] J. L. Fiadeiro, T. Maibaum, N. Martí-Oliet, J. Meseguer y I. Pita. Towards a verification logic for rewriting logic. En D. Bert, C. Choppy y P. Mosses, editores, *Recent Trends in Algebraic Development Techniques, 14th International Workshop WADT'99, Chateau de Bonas, France, September 1999, Selected Papers*, volumen 1827 de *Lecture Notes in Computer Science*, páginas 438–458. Springer-Verlag, 2000.

[FMMO+03] J. L. Fiadeiro, T. Maibaum, N. Martí-Oliet, J. Meseguer y I. Pita. Towards a verification logic for rewriting logic, 2003. Enviado para su publicación en revista.

[Fut00] K. Futatsugi, editor. *Proceedings Third International Workshop on Rewriting Logic and its Applications, WRLA 2000, Kanazawa, Japan, September 18–20, 2000*, volumen 36 de *Electronic Notes in Theoretical Computer Science*. Elsevier, 2000. http://www.elsevier.nl/locate/entcs/volume36.html.

[GKK+88] J. A. Goguen, C. Kirchner, H. Kirchner, A. Mégrelis, J. Meseguer y T. Winkler. An introduction to OBJ3. En S. Kaplan y J.-P. Jouannaud, editores, *Conditional Term Rewriting Systems, First International Workshop Orsay, France, July 8–10, 1987, Proceedings*, volumen 308 de *Lecture Notes in Computer Science*, páginas 258–263. Springer-Verlag, 1988.

[GM87] J. A. Goguen y J. Meseguer. Unifying functional, object-oriented and relational programming with logical semantics. En B. Shriver y P. Wegner, editores, *Research Directions in Object-Oriented Programming*, páginas 417–477. The MIT Press, 1987.

[Gol92] R. Goldblatt. *Logics of Time and Computation*. CSLI Publications, segunda edición, 1992.

[GWM+00] J. A. Goguen, T. Winkler, J. Meseguer, K. Futatsugi y J. P. Jouannaud. Introducing OBJ. En J. A. Goguen y G. Malcolm, editores, *Software Engineering with OBJ: Algebraic Specification in Action*, páginas 3–167. Kluwer Academic Publishers, 2000.

[HKT00] D. Harel, D. Kozen y J. Tiuryn. *Dynamic Logic*. The MIT Press, 2000.

[HM80] M. Hennessy y R. Milner. On observing nondeterminism and concurrency. En J. W. de Bakker y J. van Leeuwen, editores, *Proceedings* 7th *ICALP, Noordwijkerhout*, volumen 85 de *Lecture Notes in Computer Science*, páginas 299–309. Springer-Verlag, julio 1980.

[HM85] M. Hennessy y R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the Association for Computing Machinery*, 32(1):137–172, 1985.

[IEE95] Institute of Electrical and Electronics Engineers. *IEEE Standard for a High Performance Serial Bus. Std 1394-1995*, agosto 1995.

[ISO90] ISO/IEC. DIS 10165-1/ITU-TS X.720, Management Information Model, 1990.

[KK98] C. Kirchner y H. Kirchner, editores. *Proceedings Second International Workshop on Rewriting Logic and its Applications, WRLA'98, Pont-à-Mousson, France, September 1–4, 1998*, volumen 15 de *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998. http://www.elsevier.nl/locate/entcs/volume15.html.

[Koz83] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[KP95] Y. Kesten y A. Pnueli. A complete proof system for QPTL. En D. Kozen, editor, *Tenth Annual IEEE Symposium on Logic in Computer Science*. IEEE computer society press, San Diego, California, junio 1995.

[KW97] P. Kosiuczenko y M. Wirsing. Timed rewriting logic with an application to object-based specification. *Science of Computer Programming*, 28(2–3):225–246, abril 1997.

[Lec97] U. Lechner. *Object-Oriented Specification of Distributed Systems*. Tesis Doctoral, Fakultät für Mathematik und Informatik, Universität Passau, junio 1997.

[Lec98] U. Lechner. Object-oriented specification of distributed systems. En C. Kirchner y H. Kirchner, editores, *Proceedings Second International Workshop on Rewriting Logic and its Applications, WRLA'98, Pont-à-Mousson, France, September 1–4, 1998*, volumen 15 de *Electronic Notes in Theoretical Computer Science*, páginas 405–414. Elsevier, 1998. http://www.elsevier.nl/locate/entcs/volume15.html.

[LLNW96] U. Lechner, C. Lengauer, F. Nickl y M. Wirsing. (Objects + concurrency) & reusability — A proposal to circumvent the inheritance anomaly. En P. Cointe, editor, *ECOOP'96 — Object-Oriented Programming, 10th European Conference, Linz, Austria, July 8–12, 1996, Proceedings*, volumen 1098 de *Lecture Notes in Computer Science*, páginas 232–247. Springer-Verlag, 1996.

[LMOM94] P. Lincoln, N. Martí-Oliet y J. Meseguer. Specification, transformation, and programming of concurrent systems in rewriting logic. En G. E. Blelloch, K. M. Chandy y S. Jagannathan, editores, *Specification of Parallel Algorithms, DIMACS Workshop, May 9–11, 1994*, volumen 18 de *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, páginas 309–339. American Mathematical Society, 1994.

[LS77] E. J. Lemmon y D. Scott. An introduction to modal logic. *American Philosophical Quarterly, Monograph Series*, 11, 1977.

[Mes90a] J. Meseguer. Rewriting as a unified model of concurrency. En J. C. M. Baeten y J. W. Klop, editores, *CONCUR'90, Theories of Concurrency: Unification and Extension, Amsterdam, The Netherlands, August 1990, Proceedings*, volumen 458 de *Lecture Notes in Computer Science*, páginas 384–400. Springer-Verlag, 1990.

[Mes90b] J. Meseguer. Rewriting as a unified model of concurrency. Informe técnico SRI-CSL-90-02, SRI International, Computer Science Laboratory, febrero 1990. Revisado junio 1990.

[Mes92] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[Mes93a] J. Meseguer. A logical theory of concurrent objects and its realization in the Maude language. En G. Agha, P. Wegner y A. Yonezawa, editores, *Research Directions in Concurrent Object-Oriented Programming*, páginas 314–390. The MIT Press, 1993.

[Mes93b] J. Meseguer. Solving the inheritance anomaly in concurrent object-oriented programming. En O. M. Nierstrasz, editor, *ECOOP'93 — Object-Oriented Programming, 7th European Conference, Kaiserslautern, Germany, July 26–30, 1993, Proceedings*, volumen 707 de *Lecture Notes in Computer Science*, páginas 220–246. Springer-Verlag, 1993.

[Mes96] J. Meseguer, editor. *Proceedings First International Workshop on Rewriting Logic and its Applications, WRLA'96, Asilomar, California, September 3–6, 1996*, volumen 4 de *Electronic Notes in Theoretical Computer Science*. Elsevier, septiembre 1996. http://www.elsevier.nl/locate/entcs/volume4.html.

[Mes98a] J. Meseguer. Membership algebra as a logical framework for equational specification. En F. Parisi-Presicce, editor, *Recent Trends in Algebraic Development Techniques, 12th International Workshop, WADT'97, Tarquinia, Italy, June 3–7, 1997, Selected Papers*, volumen 1376 de *Lecture Notes in Computer Science*, páginas 18–61. Springer-Verlag, 1998.

[Mes98b] J. Meseguer. Research directions in rewriting logic. En U. Berger y H. Schwichtenberg, editores, *Computational Logic, Proceedings of the NATO Advanced Study Institute on Computational Logic held in Marktoberdorf, Germany, July 29 – August 6, 1997*, volumen 165 de *NATO ASI Series F: Computer and Systems Sciences*, páginas 347–398. Springer-Verlag, 1998.

[MOM99] N. Martí-Oliet y J. Meseguer. Action and change in rewriting logic. En R. Pareschi y B. Fronhöfer, editores, *Dynamic Worlds: From the Frame Problem to Knowledge Management*, volumen 12 de *Applied Logic Series*, páginas 1–53. Kluwer Academic Publishers, 1999.

[MOM02] N. Martí-Oliet y J. Meseguer. Rewriting logic: Roadmap and bibliography. *Theoretical Computer Science*, 285(2):121–154, 2002.

[MP92] Z. Manna y A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems. Specifications*. Springer-Verlag, 1992.

[MP95] Z. Manna y A. Pnueli. *Temporal Verification of Reactive Systems. Safety*. Springer-Verlag, 1995.

[MT02] J. Meseguer y C. Talcott. Semantic models for distributed object reflection. En B. Magnusson, editor, *ECOOP 2002 – Object Oriented Programming, 16th European Conference, Málaga, Spain, June 2002, Proceedings*, volumen 2374 de *Lecture Notes in Computer Science*, páginas 1–36. Springer-Verlag, 2002.

[Ölv00] P. C. Ölveczky. *Specification and Analysis of Real-Time and Hybrid Systems in Rewriting Logic*. Tesis Doctoral, University of Bergen, Norway, 2000. http://maude.csl.sri.com/papers.

[ÖM02] P. C. Ölveczky y J. Meseguer. Specification of real-time and hybrid systems in rewriting logic. *Theoretical Computer Science*, 285(2):359–405, 2002.

[Pit02] I. Pita. Defining in VLRL spatial properties of systems specified in rewriting logic. En R. Peña, A. Herranz y J. J. Moreno, editores, *PROLE 2002*, páginas 183–199. El Escorial, Madrid, noviembre 2002.

[PMO96] I. Pita y N. Martí-Oliet. A Maude specification of an object-oriented database model for telecommunication networks. En J. Meseguer, editor, *First Int. Workshop on Rewriting Logic and its Applications, Asilomar, California*, volumen 4 de *Electronic Notes in Theoretical Computer Science*, páginas 404–422. Elsevier Science, 1996.

[PMO97] I. Pita y N. Martí-Oliet. Using reflection to specify transaction sequences in rewriting logic. En *APPIA-GULP-PRODE'97, Joint Conference on Declarative Programming*. Grado, Italia, 1997.

[PMO99] I. Pita y N. Martí-Oliet. Using reflection to specify transaction sequences in rewriting logic. En J. L. Fiadeiro, editor, *Recent Trends in Algebraic Development Techniques, WADT'98, Lisboa, Portugal*, volumen 1589 de *Lecture Notes in Computer Science*, páginas 261–276. Springer-Verlag, 1999.

[PMO01] I. Pita y N. Martí-Oliet. Proving modal and temporal properties of rewriting logic programs. En L. M. Pereira y P. Quaresma, editores, *APPIA-GULP-PRODE'01, Joint Conference on Declarative Programming*, páginas 277–295. Évora, Portugal, septiembre 2001.

[PMO02] I. Pita y N. Martí-Oliet. A Maude specification of an object-oriented model for telecommunication networks. *Theoretical Computer Science*, 285(2):407–439, 2002.

[Sti92] C. Stirling. Modal and temporal logics. En S. Abramsky, D. Gabbay y T. Maibaum, editores, *Handbook of Logic in Computer Science. Volume 2. Background: Computational Structures*, páginas 477–563. Oxford University Press, 1992.

[Sti96] C. Stirling. Modal and temporal logics for processes. En F. Moller y G. Birtwistle, editores, *Logics for Concurrency: Structure vs Automata*, volumen 1043 de *Lecture Notes in Computer Science*, páginas 149–237. Springer-Verlag, 1996.

[Sti01] C. Stirling. *Modal and temporal properties of processes*. Springer, 2001.

[Var01] M. Vardi. Branching vs linear time: Final showdown. En T. Margaria y W. Yi, editores, *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, volumen 2031 de *Lecture Notes in Computer Science*, páginas 1–22. Springer-Verlag, 2001.

[Ver03] A. Verdejo. *Maude como marco semántico ejecutable*. Tesis Doctoral, Universidad Complutense de Madrid, 2003.

[VPMO00] A. Verdejo, I. Pita y N. Martí-Oliet. The leader election protocol of IEEE 1394 in Maude. En K. Futatsugi, editor, *Proceedings Third International Workshop on Rewriting Logic and its Applications, WRLA 2000, Kanazawa, Japan, September 18–20, 2000*, volumen 36 de *Electronic Notes in Theoretical Computer Science*, páginas 385–406. Elsevier, 2000. http://www.elsevier.nl/locate/entcs/volume36.html.

[VPMO01a] A. Verdejo, I. Pita y N. Martí-Oliet. The leader election protocol of IEEE 1394 in Maude. Informe técnico 118-01, Dpto. Sistemas Informáticos y Programación, Universidad Complutense de Madrid, 2001.

[VPMO01b] A. Verdejo, I. Pita y N. Martí-Oliet. Specification and verification of the leader election protocol of IEEE 1394 in rewriting logic. En S. Maharaj,

J. Romijn y C. Shankland, editores, *International Workshop on Application of Formal Methods to IEEE 1394 Standard*, páginas 39–43. Berlín, Alemania, marzo 2001.

[VPMO02] A. Verdejo, I. Pita y N. Martí-Oliet. Specification and verification of the tree identify protocol of IEEE 1394 in rewriting logic. *Formal Aspects of Computing*, 14(2), 2002.

[ZP92] L. L. Zueros y I. Pita. Diseño orientado a objetos aplicado a la gestión integrada de redes. En *Proc. Telecom I+D Conference*, páginas 359–368. Madrid, 1992.