UNIVERSIDAD DE GRANADA

Departamento de Lenguajes y Sistemas Informáticos

DESDE ESPECIFICACIONES LÓGICAS DE INTERVALOS A AUTÓMATAS DE PROPIEDAD: UNA CONSTRUCCIÓN *TABLEAU* PARA SU APLICACIÓN EN COMPROBACIÓN DE MODELOS *ON-THE-FLY*

TESIS DOCTORAL

Miguel J. Hornos Barranco

2002

Editor: Editorial de la Universidad de Granada
Autor: Miguel J. Hornos Barranco
D.L.: GR 757-2012
ISBN: 978-84-694-9335-9

UNIVERSIDAD DE GRANADA

DESDE ESPECIFICACIONES LÓGICAS DE INTERVALOS A AUTÓMATAS DE PROPIEDAD: UNA CONSTRUCCIÓN *TABLEAU* PARA SU APLICACIÓN EN COMPROBACIÓN DE MODELOS *ON-THE-FLY*

Memoria presentada por

Miguel J. Hornos Barranco

Para optar al grado de

DOCTOR EN INFORMÁTICA

DIRECTOR:

Manuel I. Capel Tuñón

2002

Departamento de Lenguajes y Sistemas Informáticos

D. MANUEL I. CAPEL TUÑÓN, Profesor Titular de Universidad del Departamento de Lenguajes y Sistemas Informáticos de la Universidad de Granada

CERTIFICA

Que esta memoria de tesis, titulada *Desde especificaciones lógicas de intervalos a autómatas de propiedad: una construcción tableau para su aplicación en comprobación de modelos on-the-fly*, ha sido realizada por D. MIGUEL J. HORNOS BARRANCO bajo mi dirección en el departamento y universidad arriba mencionados.

Granada, a 30 de Septiembre del 2002

Fdo.: Manuel I. Capel Tuñón	Fdo.: Miguel J. Hornos Barranco
Director de la Tesis	Doctorando

A quienes siempre
han creído en mí y
me han apoyado,
en especial a Pilar

AGRADECIMIENTOS

En primer lugar quisiera dar las gracias a mi director de tesis, Manuel I. Capel Tuñón, por haber despertado mi interés en el campo de las lógicas temporales y en su aplicación a la especificación y verificación de sistemas, por haberme introducido en estos temas, por la confianza depositada en mí desde el primer momento y por el tiempo y el esfuerzo que ha invertido en la dirección de esta tesis.

También deseo expresar mi gratitud a todos los miembros del Departamento de Lenguajes y Sistemas Informáticos de la Universidad de Granada, por su compañerismo y el clima distendido que han sabido crear a la largo de todos estos años. En particular, me gustaría acentuar mi agradecimiento a su director, Juan Carlos Torres Cantero, por la magnífica labor que a mi juicio está realizando, a los componentes del grupo de investigación GEDES, con José Parets Llorca a la cabeza, por todo el apoyo recibido, y a los compañeros que imparten clase en la Facultad de Ciencias Económicas y Empresariales de Granada, especialmente a Mavi Hurtado Torres, Mª del Mar Abad Grau y Rosana Montes Soldado, con las que mantengo un mayor contacto, por el agradable ambiente de trabajo creado gracias a sus valores humanos, su inestimable colaboración y ayuda para resolver cualquier problema y su apoyo en los momentos difíciles.

Los resultados y aportaciones de esta tesis se deben en gran parte a la inspiración recibida del artículo de Rob Gerth, Doron A. Peled, Moshe Y. Vardi y Pierre Wolper [Gerth95], así como a la utilización de la lógica de intervalos creada por Laura K. Dillon, George Kutty, P. Michael Melliar-Smith, Louise E. Moser y Y. Srinivas Ramakrishna. Mi agradecimiento a todos ellos y a los autores de los trabajos referenciados a lo largo de esta memoria de tesis, por haberme enseñado tanto a través de sus publicaciones. Igualmente debo agradecer la ayuda recibida de varios investigadores del *Laboratory for Theoretical Computer Science* de la *Helsinki University of Technology* (Finlandia), en especial de Marko Mäkelä, por haber puesto a mi disposición el código fuente de su traductor LBT (*LTL to Büchi automaton Translator*) y por proporcionarme importante información sobre él a través del correo electrónico, lo que me facilitó la tarea de la implementación del

algoritmo que hemos desarrollado y, por consiguiente, de la construcción de nuestra herramienta.

Mi reconocimiento y gratitud a todos aquellos profesores que he tenido en las distintas etapas de mi formación y que supieron transmitirme sus conocimientos y responder a innumerables interrogantes que se me plantearon, siendo sin duda la fuente de gran parte de lo que hoy sé. Dentro de todos ellos, debo destacar a Dª Flora, profesora que me marcó profundamente, ya que en varios cursos de la extinta E.G.B. sentó en mí unas sólidas bases matemáticas y me enseñó cómo se debían resolver correctamente y expresar de forma elegante la solución de aquellos primeros problemas en que se ponía a prueba nuestra capacidad de razonamiento, siendo además para mí un modelo a seguir como persona, por sus valores y su saber estar y actuar.

Otra persona que me ha ayudado bastante es Francisco Molina Carretero, al que debo agradecer que siempre esté ahí cuando lo necesito y cuyas palabras de sosiego y aliento me han tranquilizado y animado enormemente en muchos momentos de mi vida, algunos de ellos durante el desarrollo del trabajo de investigación que ahora culmina con esta memoria de tesis.

Gracias también a David Martínez Pérez y a Mario Abad Grau por su colaboración en la creación del gráfico que encabeza las distintas unidades (capítulos y apéndices) que componen este documento, al primero por la idea básica de su diseño y la confección de una versión preliminar y al segundo por la ayuda recibida para la realización de las últimas modificaciones efectuadas sobre el mismo.

Finalmente, mi más intenso agradecimiento a mi familia y seres queridos, especialmente a Pilar, mi mujer, por creer siempre en mí y permanecer continuamente a mi lado, compartiendo mis alegrías y penas, mis sueños y desengaños, mis ilusiones y decepciones, mis éxitos y fracasos, por su comprensión y apoyo constante y por sus palabras de ánimo y estímulo durante la realización de esta tesis.

RESUMEN

Este trabajo constituye el punto de partida para la verificación automática de sistemas mediante comprobación de modelos *on-the-fly*, especificando los requisitos o propiedades temporales que el sistema debe cumplir con fórmulas de una lógica de intervalos, que nos permite razonar a nivel de intervalos de tiempo en lugar de instantes.

Las lógicas temporales de intervalos permiten especificar formalmente los requisitos o propiedades temporales de sistemas complejos, como son los sistemas reactivos, en general, y los concurrentes, en particular, en cuyo estudio y análisis se centra nuestro interés. La ventaja de este tipo de lógicas, frente a las lógicas temporales tradicionales, es que permiten establecer de forma sucinta el contexto temporal en el que ciertos requisitos deben aplicarse, haciendo posible una especificación más concisa y comprensible de las propiedades a verificar, especialmente de aquéllas más complejas.

Uno de los objetivos actuales de la comunidad científica consiste en intentar que las técnicas de especificación formal sean más fáciles de entender para un grupo cada vez más amplio de personas que tienen responsabilidad en el proceso de desarrollo del software. Una posible manera de alcanzar dicho objetivo sería adoptar un lenguaje de descripción basado en una lógica temporal que soporte una representación gráfica semánticamente equivalente a la de sus fórmulas textuales. Hemos adoptado la lógica temporal de intervalos denominada *Future Interval Logic* (FIL), ya que esta lógica cuenta con una representación gráfica muy natural e intuitiva, que hace que las especificaciones sean más fáciles de desarrollar y de comprender, incluso para aquellas personas involucradas en el desarrollo de software que no tengan una buena formación matemático-lógica. Por tanto, constituye la base sobre la que se pueden construir herramientas software amigables para el razonamiento formal acerca de las propiedades temporales de sistemas complejos (concurrentes y reactivos), que permitan una más fácil especificación y verificación de los mismos y donde el formalismo de especificación se integre perfectamente dentro de su interfaz gráfica.

La principal contribución de esta tesis es que, para dicha lógica, proporciona el diseño y la implementación de un algoritmo basado en la extensión del método *tableau* y capaz de operar *on-the-fly*, esto es, capaz de generar el autómata que expresa la propiedad a comprobar al mismo tiempo que, y guiado por, el autómata que describe el sistema a verificar. Esto es así debido a que desde el primer momento se ha concebido para ser integrado dentro de una herramienta de comprobación de modelos *on-the-fly*. Éste es el primer algoritmo conocido de este tipo para una lógica de intervalos, estando convencidos de su relevancia, puesto que durante mucho tiempo se ha considerado que estos métodos no eran aplicables a las lógicas de intervalos. La importancia que, dentro de las técnicas de verificación automática, tiene la comprobación de modelos *on-the-fly* se debe a que en muchos casos ayuda a paliar el problema de la *explosión de estados*, que ocurre cuando el espacio de estados del sistema es demasiado grande como para poder llevar a cabo la verificación. Para solucionarlo, este método construye el espacio de estados conforme se va necesitando (construcción *on-the-fly*), a medida que se realiza la verificación, por lo que sólo debe construir la parte que va explorando. Así, a menudo es posible obtener una respuesta del algoritmo sin necesidad de construir completamente el espacio de estados del correspondiente autómata.

En la tesis se demuestra que el autómata construido por el algoritmo para una especificación expresada en FIL acepta exactamente las mismas secuencias de palabras infinitas que son modelos de dicha especificación. Por último, se compara nuestra herramienta con otra de características similares, pero cuyo formalismo de especificación es una lógica temporal tradicional, la LTL (*Linear Temporal Logic*), mostrándose que los autómatas producidos por ambas para fórmulas equivalentes son de una complejidad similar, aunque los derivados con nuestro algoritmo salen ligeramente favorecidos en la mayoría de los casos analizados.

ÍNDICE DE CONTENIDOS

LISTA DE DEFINICIONES

LISTA DE EJEMPLOS

LISTA DE FIGURAS

LISTA DE TABLAS

Capítulo 1

INTRODUCCIÓN

CONTENIDO

RESUMEN Y ORGANIZACIÓN

Este primer capítulo sirve de introducción al resto de la memoria de esta tesis, pretendiendo dar una visión general de lo que se expondrá de manera detallada a lo largo de la misma. Así, en la primera sección se presenta la motivación que tuvimos para emprender esta línea de investigación, así como una justificación del porqué del trabajo desarrollado. En la segunda sección se recogen de forma clara los objetivos que se pretenden alcanzar con esta tesis, desde los más generales hasta los más específicos. Finalmente, la tercera sección muestra cómo se ha organizado esta memoria, describiendo brevemente el contenido de cada uno de los restantes capítulos y de los apéndices que la conforman.

1.1. Motivación y justificación del trabajo desarrollado

Gran parte de los sistemas que se diseñan y desarrollan en la actualidad, tanto a nivel de hardware como de software, son concurrentes y reactivos. Un sistema *concurrente* se caracteriza porque puede haber muchas acciones ocurriendo en él simultáneamente. Un sistema *reactivo* puede definirse como aquél que se ejecuta infinitamente y que reacciona ante cualquier estímulo (evento, suceso, acción o como quiera denominarse) que se produzca en su entorno, manteniendo una interacción continua con él. De este modo, se diferencian claramente de los sistemas funcionales, cuyo comportamiento se describe sólo en base a la transformación que aplican a los datos de entrada para producir los resultados o datos de salida, en un sentido puramente estático, o sea, sin requerir que se relacionen las entradas y salidas con el tiempo en que éstas se producen. Por el contrario, los sistemas reactivos tienen como resultado secuencias complejas de eventos, en las que por regla general existen restricciones temporales explícitas. Bajo la denominación de sistemas concurrentes y reactivos se engloban muchas aplicaciones complejas del mundo real, que están entre las más difíciles de diseñar y desarrollar, algunas de las cuales se encuentran, por ejemplo, dentro de los siguientes tipos de sistemas: sistemas operativos, protocolos de comunicaciones, sistemas de control de procesos, sistemas empotrados y ciertos circuitos electrónicos, entre otros.

Dentro de los sistemas reactivos, nuestro interés se centra fundamentalmente en los sistemas software concurrentes. El razonamiento con este tipo de sistemas es considerablemente más difícil que el razonamiento con los sistemas secuenciales. Así, un razonamiento informal sobre un sistema concurrente, que a primera vista parece convincente, a menudo resulta ser inválido en un análisis más detenido. Además, el diseñador de este tipo de sistemas, debido a la naturaleza asíncrona de los mismos, puede fácilmente no considerar todas los posibles entremezclamientos de eventos. Consecuentemente, el uso de métodos formales es esencial para el análisis y el establecimiento de la corrección de los sistemas concurrentes y reactivos, especialmente en aquellas aplicaciones en las que la seguridad es crítica, como por ejemplo: el software de control de una central nuclear, de un cohete espacial o de un avión, donde un pequeño error en su especificación y diseño puede poner en peligro bastantes vidas humanas.

Varios estudios han mostrado que los errores en las especificaciones son los más frecuentes en el desarrollo del software y los más caros de corregir [Boehm81] [Fairle85]. Los métodos formales ayudan a reducir significativamente el número de errores en las especificaciones, ya que eliminan la imprecisión y la ambigüedad y reducen la incompletitud y la inconsistencia. Además, y a diferencia de lo que ocurre con las especificaciones informales, tales como las descripciones en lenguaje natural, si los requisitos de un sistema se expresan en un lenguaje formal, se les puede aplicar un *análisis formal*, que puede ser de distinto tipo, según interese:

- *Comprobación de completitud y consistencia*, que permite detectar posibles errores en las especificaciones ejecutando un análisis estático, es decir, usando para ello sólo información sintáctica.

- *Validación*, que es soportada mediante *simulación* conducida por las especificaciones. Así, al ejecutar una serie de simulaciones (cada una de las cuales representa una posible ejecución del sistema), el usuario puede determinar si el comportamiento del sistema representado por las especificaciones es consistente con su intención. En general, se puede decir que la simulación es a la especificación lo que la prueba es al software. Debido a que sólo se realiza un camino en cada ejecución del simulador, éste sólo puede mostrar la presencia de errores y no la ausencia de éstos. A pesar de esta limitación, los simuladores han probado su valor, puesto que proporcionan un camino de ejecución, en el que el comportamiento inesperado es rápidamente identificado.

- *Verificación formal*, que comprueba o determina la validez de las especificaciones (sobre todas las ejecuciones posibles) para propiedades críticas de la aplicación.

La sociedad actual demanda continuamente un incremento en la complejidad de los sistemas automatizados (muchos de ellos, utilizados por todos nosotros diariamente). Consecuentemente, los métodos industriales más tradicionales, como los de simulación y prueba, son ahora poco prácticos para eliminar completamente los errores de esta clase de sistemas. Téngase en cuenta que para muchos de estos sistemas (concurrentes y reactivos) el número de pruebas posibles a realizar sería infinito. Como es sabido, uno de los principales objetivos de la ingeniería del software consiste en encontrar los errores en el diseño de una aplicación en fases tempranas de su desarrollo (ciclo de vida). Los métodos mencionados, al confiar en la prueba

"exhaustiva", claramente fracasan en este empeño. Estos métodos podrían no encontrar un error hasta que el sistema estuviera completamente instalado y funcionando. Un error que se detecta tarde puede conllevar el rediseño y/o implementación de muchas partes del código para corregirlo, implicando un incremento considerable tanto en el tiempo como el coste necesario para su desarrollo.

Por lo tanto, nuestro trabajo irá encaminado hacia la especificación y verificación formal de los sistemas concurrentes y reactivos. Para el desarrollo de este tipo de sistemas se ha demostrado que el *ciclo de vida en espiral*, que permite un refinamiento sucesivo hasta obtener un modelo correcto del sistema en la fase de análisis o de diseño (o sea, antes de implementar nada), es el mejor, ya que permite conseguir un ahorro en tiempo y dinero que no sería posible si se siguiera el ciclo de vida clásico. Por otra parte, el uso de métodos formales no condiciona en absoluto a la hora de escoger, por ejemplo, el lenguaje de programación concreto con el que se implementará el sistema.

A pesar del relativamente continuo y permanente interés en la especificación y verificación de sistemas concurrentes y reactivos dentro de la comunidad teórica, los métodos formales han sido poco aceptados a nivel práctico por parte de la comunidad industrial. Esta impopularidad se debe, en gran parte, a la creencia generalizada [Hall90] [Bowen95] entre los profesionales del sector (diseñadores, analistas, programadores, etc.) de que los métodos formales son innecesariamente enrevesados, difíciles y cargados de formalismos matemático-lógicos como para que el esfuerzo y el tiempo invertidos en su utilización merezcan la pena. Para ser empleados a nivel práctico por estos profesionales, los métodos formales deben ser bastante cercanos a la forma en que ellos normalmente razonan para justificar la corrección de sus diseños. Por otra parte, esos métodos deben ser suficientemente rigurosos como para encontrar, o ayudar a encontrar, cualquier fallo en esos diseños. Huelga decir que esto requiere el soporte automatizado de herramientas eficientes.

Como ya se ha explicado, la especificación y verificación de sistemas reactivos es una tarea difícil, especialmente si dichos sistemas son también concurrentes. No obstante, la lógica temporal [Emerso90] es un formalismo apropiado para razonar acerca del ordenamiento relativo de eventos en este tipo de sistemas [Pnueli86a] [Pnueli86b] [Manna92]. Sin embargo, los diseñadores de estos sistemas han

encontrado difícil razonar en lógica temporal y relacionar dicha lógica con sus diseños hardware y software. Además, muchas especificaciones de sistemas requieren que un cierto comportamiento se satisfaga bajo unas circunstancias concretas y no bajo otras distintas. Por consiguiente, tales especificaciones deben describir no sólo los requisitos, sino también el contexto temporal en el que esos requisitos se aplican. Pues bien, la mayoría de las lógicas temporales tradicionales no proporcionan representaciones sucintas para el establecimiento de contextos temporales. Esto supone que, para conseguir con ellas la especificación deseada, en muchas ocasiones se deba complicar bastante su expresión, resultando especificaciones poco intuitivas y comprensibles. Sin duda, la representación textual de la lógica temporal ha contribuido en parte a estas dificultades. Todo ello ha obstaculizado el uso de dichas lógicas en el desarrollo de aplicaciones industriales.

Las representaciones gráficas facilitan la comprensión y el razonamiento humano. Sin embargo, las representaciones gráficas usadas por los diseñadores de sistemas son a menudo informales y carecen de significado bien definido. Por tanto, parece lógico que, para permitir a los ingenieros de hardware y software describir sistemas reactivos y razonar con ellos con mayor facilidad y rigor, se deba utilizar una lógica temporal con una sintaxis y una semántica rigurosamente definidas, que cuente además con una representación gráfica que encaje con la manera de pensar que el diseñador del sistema tiene acerca del dominio del problema. De este modo, se puede desarrollar un entorno gráfico que permita el uso de dicha lógica de una forma más intuitiva.

Todas estas razones nos han llevado a emplear como base para el trabajo desarrollado en esta tesis una lógica que puede ayudar a superar los problemas planteados, denominada unas veces GIL (*Graphical Interval Logic*) [Dillon94a], cuando se pretende resaltar su cualidad como instrumento de representación visual e intuitiva, y otras veces FIL (*Future Interval Logic*) [Ramakr92], indicándose con ello que sólo trata con situaciones futuras, es decir, que no cuenta con ningún operador del pasado. Se trata de una lógica de intervalos, que permite que el razonamiento se lleve a cabo a nivel de intervalos de tiempo, en lugar de instantes; lo que ayuda a simplificar la especificación de sistemas reactivos y concurrentes, ya que los intervalos están explícitamente diseñados para facilitar la definición de contextos temporales y de propiedades que se deben cumplir en dichos contextos. Además, la ventaja fundamental de esta lógica, con respecto a otras lógicas de intervalos, es que es elementalmente decidible [Ramakr96a]. Para la mayoría del

resto de este tipo de lógicas, el problema de decisión es, a lo sumo, no elemental y, a menudo, indecidible.

Nuestro interés se centra en la aplicación de esta lógica a la especificación y verificación automática de sistemas concurrentes [Hornos01b], utilizando concretamente la técnica denominada *comprobación de modelos on-the-fly* [Hornos02]. Por tanto, nuestro campo de aplicación difiere del de los creadores de dicha lógica, que la han utilizado para desarrollar un prototipo de demostrador de teoremas [Mellia94] con una interfaz gráfica amigable. Por otra parte, los procedimientos de decisión que éstos han implementado [Ramakr93a] se presentan siguiendo el conocido *enfoque basado en teoría de autómatas*[1] [Vardi86], que, como es sabido, produce el peor caso de complejidad. Este caso puede ser evitado a menudo si se sigue el también conocido método *tableau* [Wolper85] [D'Agos99]. Para la implementación del algoritmo que se presenta como principal contribución de esta tesis hacemos uso de dicho método; además, al igual que el algoritmo en el que tiene sus raíces [Gerth95], está pensado para operar *on-the-fly*, cualidad de la que también carecen los algoritmos previos implementados para la lógica que utilizamos.

1.2. Objetivos

Los objetivos principales que se pretender conseguir con esta tesis pueden resumirse, ordenándolos desde los más generales a los más concretos, en los siguientes puntos:

- Contribuir al desarrollo de los métodos formales aplicados a la especificación y verificación de los sistemas reactivos en general y concurrentes en particular.
- Aplicar una lógica de intervalos a la especificación y verificación *automática* de este tipo de sistemas, lo que es algo novedoso dentro de este terreno.
- Especificar las propiedades temporales de un sistema en un formalismo que se ajusta bastante bien a la manera en que los seres humanos razonan.

[1] Aunque en el Apéndice B de [Ramakr93] se da una descripción (de alto nivel, en palabras de los autores) de cómo se puede realizar una implementación basada en *tableau* para RTFIL (la extensión de FIL para tiempo real), ésta se presenta sin ninguna demostración ni detalle alguno de implementación.

- Propiciar el desarrollo de métodos automatizados (asistidos por herramientas) amigables, basados en representaciones gráficas intuitivas de especificaciones formales, que acerquen los métodos formales a los ingenieros de software y a los diseñadores de sistemas.
- Extender los algoritmos de decisión basados en el método *tableau*, tan habituales entre las lógicas tradicionales, para que puedan operar con fórmulas de una lógica de intervalos. Hasta hace muy poco tiempo esto parecía una tarea nada fácil, e incluso imposible.
- Diseñar y desarrollar un algoritmo, basado en el método *tableau* y pensado para operar *on-the-fly*, que permita la traducción eficiente del requisito especificado por una fórmula de la lógica empleada a una descripción más operacional y semánticamente equivalente (autómata de Büchi).
- Implementar una herramienta, basada en el algoritmo anterior, que nos permita aplicar los conceptos teóricos desarrollados al terreno práctico y experimentar con ellos.
- Emplear esa herramienta para el estudio de diversas especificaciones, especialmente aquéllas que se utilizan con más frecuencia para describir los requisitos de los sistemas en los que estamos interesados, con el fin de comprobar si éstas son consistentes (satisfacibles) y coherentes con el comportamiento que se pretendía especificar.
- Estudiar y comparar los resultados obtenidos con nuestra herramienta con los producidos para especificaciones equivalentes por otra herramienta de características similares, pero con un formalismo lógico subyacente distinto.

1.3. Estructura de esta memoria de tesis

Además de este capítulo, la presente memoria consta de otros seis capítulos más, de tres apéndices y, por supuesto, de la relación de referencias bibliográficas empleadas a lo largo de la misma, que se adjunta al final.

En el Capítulo 2: El Marco Lógi co se presenta el formalismo lógico utilizado como base para el desarrollo del trabajo realizado. Se trata de una lógica de intervalos que cuenta con dos representaciones distintas: una gráfica y otra textual. Sus fórmulas nos van a servir para especificar las propiedades o requisitos temporales

que un sistema debe cumplir. Este capítulo constituye, por tanto, el punto de partida obligado para que aquellas personas no familiarizadas con dicha lógica puedan comprender el resto de los capítulos que componen la memoria.

En el Capítulo 3: Técnicas Automatizadas de Verificación se establece el marco en el que se encuadra nuestro trabajo con respecto a las técnicas de verificación existentes en la actualidad. En él se describirán los aspectos más importantes de cada una de las tres técnicas más utilizadas para comprobar que un sistema es correcto: la *demostración de teoremas*, la *comprobación de modelos* y la *realización de pruebas*, resaltando las principales fortalezas y debilidades de cada técnica. Asimismo se comentará cómo la combinación de varias de estas técnicas pueden mejorar los resultados obtenidos.

En el Capítulo 4: Verificación Basada en Autómatas se expone el método de verificación concreto en el que se aplica directamente lo desarrollado en esta tesis, explicando los pasos que componen el procedimiento a seguir para verificar automáticamente sistemas concurrentes y cómo implementar eficientemente dicho procedimiento. Además se dan los fundamentos teóricos sobre los que se asientan las bases para modelar los distintos procesos (concurrentes) que componen el sistema que se pretende verificar, así como la estrategia general de conversión de especificaciones realizadas en la lógica empleada a autómatas de Büchi (sobre los que se llevará finalmente a cabo la verificación). Finalmente, también se muestra cómo se puede utilizar el trabajo realizado para comprobar la satisfacibilidad o la validez de una especificación efectuada en dicha lógica.

En el Capítulo 5: Generación Automática del Autómata de Propiedad se exponen las principales aportaciones de nuestro trabajo, por lo que se puede decir que es el capítulo más importante de esta tesis. En él se describe detalladamente el algoritmo que hemos desarrollado para traducir una fórmula de nuestra lógica a un autómata de Büchi semánticamente equivalente, centrándonos especialmente en las reglas que determinan cómo se expanden o descomponen las fórmulas de intervalo; estas reglas constituyen el núcleo y principal novedad de nuestro algoritmo. También se explica cómo determinar adecuadamente los estados de aceptación del autómata generado y cómo convertir un autómata con varios conjuntos de estados de aceptación (de los que genera el algoritmo presentado) en uno equivalente, pero con un único conjunto de estados de aceptación. Por último, se demuestra formalmente la corrección del algoritmo presentado.

En el Capítulo 6: Diseño e Implementación de la Herramienta se completa y complementa lo expuesto en el capítulo anterior, ya que en él se presentan las principales decisiones que se han tomado a la hora de diseñar e implementar una herramienta basada en el algoritmo explicado en el mismo. Así, se ofrece una descripción del diseño realizado en base a diagramas que muestran las relaciones existentes entre las distintas clases implementadas, comentándose además la estructura general de la implementación que hemos llevado a cabo. También se presenta la interfaz de entrada y de salida de dicha herramienta, así como una serie de resultados experimentales obtenidos con ella. Por último, se compara nuestra herramienta con otra de características similares (basada en el método *tableau* y capaz de operar *on-the-fly*), pero que tiene como formalismo de especificación una lógica temporal tradicional en lugar de una lógica de intervalos, no sólo en sus respectivas implementaciones y diseños, sino también en los resultados obtenidos por ambas para especificaciones equivalentes.

En el Capítulo 7 : Conclusiones se presentan los trabajos relacionados con el nuestro y se exponen las conclusiones y principales aportaciones del trabajo de investigación que se ha llevado a cabo. Asimismo, se señalan posibles líneas futuras de investigación como continuación al trabajo desarrollado.

En el Apéndice A: Glosario de Términos y Acrónimos se incluyen todos los acrónimos (junto con las expresiones que dan lugar a ellos) y los términos más importantes y significativos que se emplean a lo largo de esta memoria de tesis. Para cada uno de ellos se presenta una breve definición o descripción.

En el Apéndice B: Extendiendo la Lógica con Nuevos Operadores: Búsquedas e Intervalos Fuertes se indica cómo se puede extender el formalismo lógico admitido por nuestra herramienta con la inclusión de dos nuevos operadores y cómo modificar su implementación para conseguirlo. Además se comentan las principales consecuencias de esta extensión.

Finalmente, en el Apéndice C: Documentación del Código Fuente de FBT se presenta una documentación que ha sido generada directa y automáticamente a partir del código fuente de nuestra herramienta, denominada FBT. En él se detallan los distintos elementos (clases, funciones, atributos, tipos, etc.) que componen cada uno de los seis ficheros en el que se distribuye su implementación, dándose una pequeña descripción del cometido o significado de cada elemento.

Dado que a lo largo de esta memoria de tesis se utilizan los autómatas como formalismo de representación, la Figura 1.1 muestra mediante un grafo similar al sistema de transiciones de éstos las dependencias existentes entre las distintas unidades (capítulos y apéndices) que la componen. Así, para la lectura de esta memoria puede seguirse el sentido de las flechas de línea continua, mientras que las flechas de línea discontinua dirigidas al Apéndice A representan consultas puntuales a dicho apéndice.

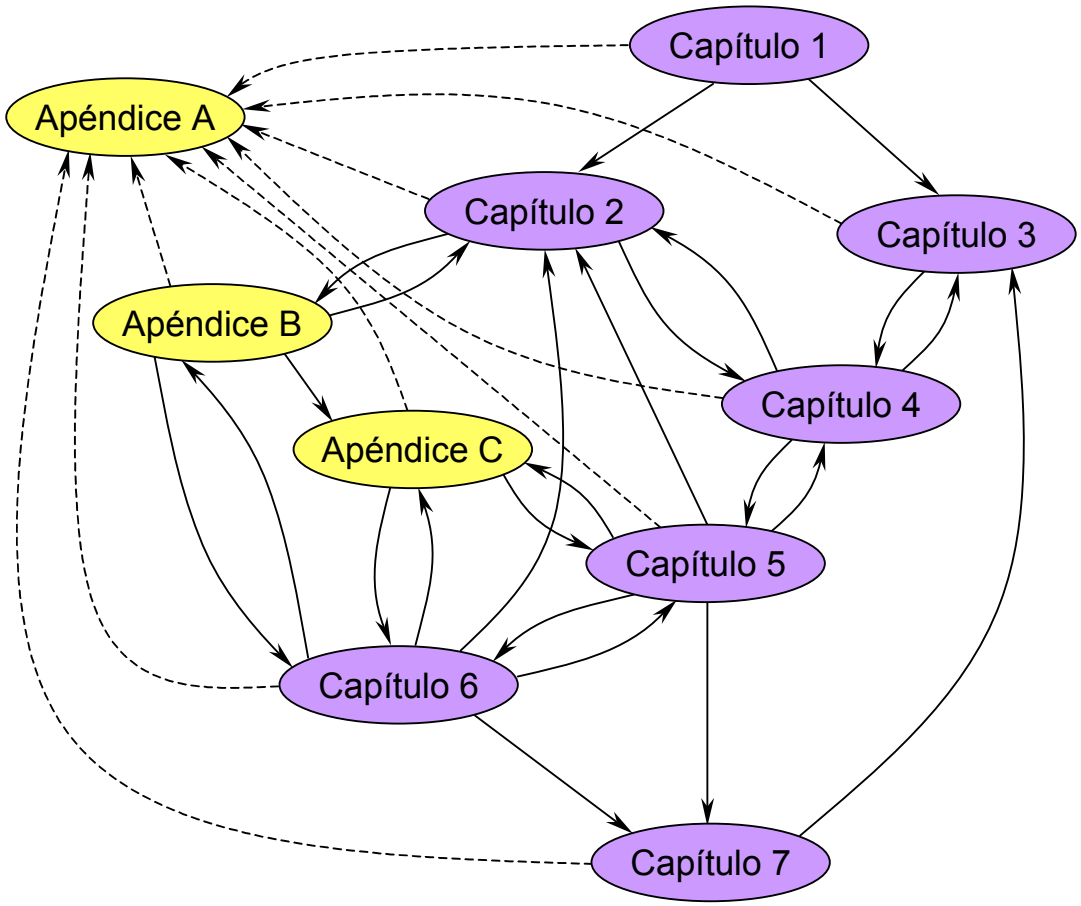
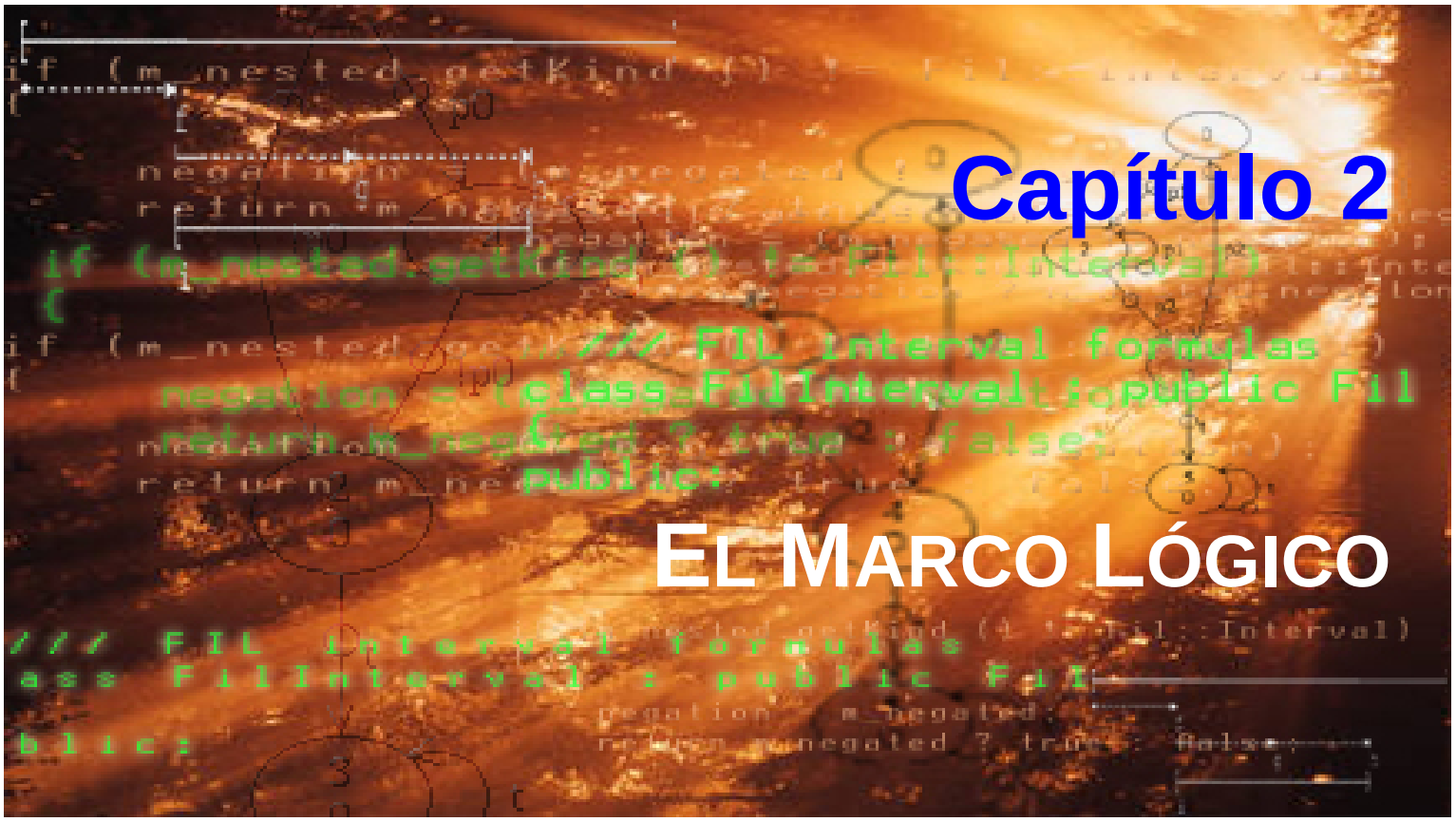


Figura 1.1. Dependencias entre las distintas unidades que componen la memoria de tesis

Para finalizar, indicar que todas las alusiones a referencias bibliográficas se pondrán entre corchetes, utilizando los seis primeros caracteres del apellido del primer autor de ese trabajo, seguido por los dos últimos dígitos del año de publicación. Si, según esto, dos referencias distintas tuvieran la misma "codificación", entonces se añade una letra minúscula diferente (empezando por la *a* y en orden alfabético ascendente) al final de cada una de ellas, con el fin de distinguirlas entre sí.

ÍNDICE DE REFERENCIAS EMPLEADAS

[Boehm81], *4*
[Bowen95], *5*
[D'Agos99], *7*
[Dillon94a], *6*
[Emerso90], *5*
[Fairle85], *4*
[Gerth95], *7*
[Hall90], *5*
[Hornos01b], *7*
[Hornos02], *7*
[Manna92], *5*
[Mellia94], *7*
[Pnueli86a], *5*
[Pnueli86b], *5*
[Ramakr92], *6*
[Ramakr93], *7*
[Ramakr96], *6*
[Vardi86], *7*
[Wolper85], *7*



Capítulo 2

El Marco Lógico

Contenido

Resumen y Organización

El objetivo de este capítulo es presentar la lógica que vamos a emplear para especificar las propiedades de un sistema. Se trata de una lógica que cuenta con dos representaciones distintas: una gráfica y otra textual, y cuyo elemento clave o modalidad temporal básica es el intervalo. En la primera sección se describen brevemente las principales características de dicha lógica. En la segunda, se introduce de una manera informal la notación empleada para representar gráficamente sus fórmulas, mientras que en la tercera sección se ilustra cómo se puede usar esta lógica para la especificación de propiedades temporales de un sistema. A continuación, en la sección cuarta, se define formalmente tanto la sintaxis como la semántica de sus fórmulas, para lo que se utiliza su representación textual, estableciendo además todos los conceptos lógicos necesarios para ello. Finalmente, la última sección describe la terminología lógica más general e importante empleada en la redacción de esta memoria de tesis.

2.1. Descripción general de la lógica utilizada

A lo largo de esta memoria se utilizará una lógica de intervalos para el razonamiento con secuencias de estados, a la que se denominará GIL (*Graphical Interval Logic*) [Dillon94a] cuando sus fórmulas se representen de forma gráfica y FIL (*Future Interval Logic*) [Ramakr92] cuando la representación de dichas fórmulas sea textual. Un estado es una determinada interpretación, en el sentido de la Lógica Proposicional clásica, que asigna valor de verdad o falsedad a cada una de las proposiciones atómicas del lenguaje. El tiempo, en nuestro sistema lógico, es isomorfo con el conjunto de los enteros no negativos, o sea, es discreto, lineal y cuenta con un punto inicial, pero no así con un punto final.

Las lógicas de intervalos permiten que el razonamiento se lleve a cabo a nivel de intervalos de tiempo, en lugar de instantes. Sin embargo, y a diferencia de otras lógicas de intervalos, los elementos primitivos en el modelo semántico de nuestra lógica no son intervalos, sino instantes. Un intervalo se forma identificando sus extremos, que son instantes que satisfacen determinadas propiedades.

Los puntos extremos de un intervalo se buscan en el *contexto global*, que representa la secuencia infinita de estados correspondiente a una ejecución del sistema. Una vez localizados los extremos de un determinado intervalo, la semántica de la fórmula que lo acompaña (fórmula anidada) se restringe a la subtraza finita delimitada por dichos puntos. Por tanto, cada *intervalo* representa un *contexto temporal específico*, que es una subtraza del contexto global. Dicha subtraza se puede modelar como una secuencia infinita de estados mediante repetición de su último estado, a lo que se denomina *tartamudeo* (en inglés, *stuttering*).

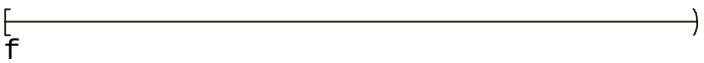
2.2. Introducción al lenguaje gráfico de especificación GIL

GIL (*Graphical Interval Logic*) [Dillon94a] es una lógica temporal que cuenta con una notación visual intuitiva y natural, de tal modo que sus fórmulas se parecen a los diagramas temporales informales que los diseñadores de sistemas suelen dibujar. Sin embargo, a diferencia de estos últimos, la especificación de un sistema expresada en GIL no sacrifica los beneficios que proporciona el empleo de una lógica formal.

En esta sección se introduce informalmente la notación o lenguaje gráfico empleado en GIL, mientras que en la sección 2.4 se definirá más formalmente su semántica, apoyándonos para ello en la representación textual asociada a cada una de sus fórmulas.

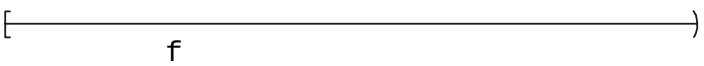
Toda fórmula GIL empieza (en su parte superior) con un *intervalo*, que representa una traza infinita de estados correspondiente a una ejecución del sistema, donde el tiempo progresa desde la izquierda a la derecha. Todos los intervalos se representan mediante una línea continua delimitada con un corchete por su izquierda y con un paréntesis por su derecha, indicando que son *medio-abiertos*, esto es, incluyen su extremo izquierdo, pero no su extremo derecho. El intervalo es el elemento clave en GIL, dado que limita o restringe el ámbito en el que una determinada propiedad se debe cumplir. Así, pueden establecerse tres tipos de propiedades para su cumplimiento dentro de un intervalo:

- ***Propiedad inicial***. La fórmula que expresa la propiedad (`f`) se dibuja alineada a la izquierda, justo debajo del corchete que señala el extremo izquierdo del intervalo. Con ello se afirma que `f` se cumple en el *primer estado* del intervalo.



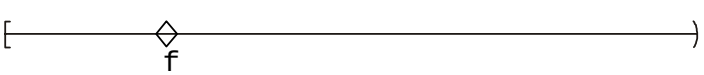
(2.1)

- ***Propiedad invariante***. La fórmula `f` se coloca debajo del intervalo y sangrada a la derecha de su extremo izquierdo, con lo que se expresa que `f` se cumple en *cada estado* del intervalo, o sea, que es una propiedad invariante sobre todo ese intervalo.



(2.2)

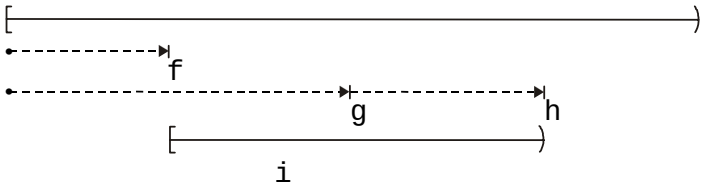
- ***Propiedad de eventualidad***. Se dibuja un rombo sobre el intervalo, con la fórmula objetivo de la eventualidad alineada a la izquierda justo debajo del rombo. Así, se afirma que `f` se satisface en *algún estado* del intervalo.



(2.3)

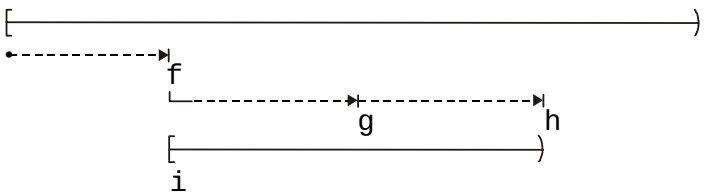
Estas propiedades pueden establecerse sobre el intervalo que representa una ejecución completa del sistema (lo que en la sección 2.1 se ha denominado *contexto global*), tal y como se muestra en las fórmulas anteriores, o sobre un intervalo extraído de otro más grande. Sintácticamente, un intervalo se define mediante dos patrones de búsqueda, uno por cada uno de sus extremos. Cada *patrón de búsqueda*

es una secuencia de una o más búsquedas. Una *búsqueda* localiza, empezando en un estado concreto, el primer estado de su futuro reflexivo[1] en el que se cumple una determinada propiedad, denominada *fórmula objetivo* de la búsqueda. Cada *búsqueda* se representa gráficamente mediante un segmento de línea discontinua con una punta de flecha en su extremo derecho y con su fórmula objetivo alineada a la izquierda debajo de dicho extremo. Cuando un patrón de búsqueda contiene varias búsquedas, cada búsqueda sucesiva empieza en el estado localizado por la búsqueda previa. La última búsqueda en un patrón de búsqueda localiza el estado que corresponde al extremo del intervalo definido por ese patrón de búsqueda. Así, en la fórmula (2.4) se extrae un intervalo del contexto global, utilizando para ello dos patrones de búsqueda: el primero sólo contiene una búsqueda, que localiza la propiedad `f`, mientras que el segundo está compuesto por dos búsquedas, cuyas fórmulas objetivos son `g` y `h` respectivamente. Obsérvese que el cumplimiento de la propiedad `i` se restringe sólo al intervalo interno, siendo una propiedad invariante sobre dicho intervalo.



(2.4)

El símbolo • representa el punto donde la búsqueda comienza. En la fórmula (2.4) anterior, ambos patrones de búsqueda empiezan en el mismo estado, esto es, al principio del contexto global. Sin embargo, en muchas ocasiones se requiere que el patrón que busca el extremo derecho del intervalo lo haga a partir del estado localizado para su extremo izquierdo. La fórmula (2.5) indica cómo se representa esto en GIL. Así, en esta fórmula la búsqueda de `g` comienza en el estado donde primero se cumpla `f`, y no a partir del estado inicial de la ejecución, como en la fórmula (2.4).



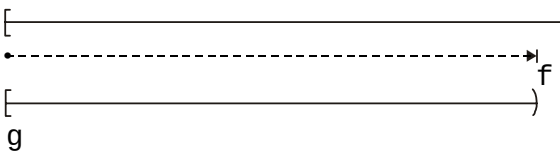
(2.5)

[1] El *futuro reflexivo* o *futuro no estricto* es el que incluye el estado en el que empieza la búsqueda, o sea, el instante actual.

Obsérvese que en la fórmula (2.5) la propiedad `i` se debe cumplir sólo en su estado inicial, mientras que en la fórmula (2.4) dicha propiedad debe cumplirse en cada estado del intervalo construido.

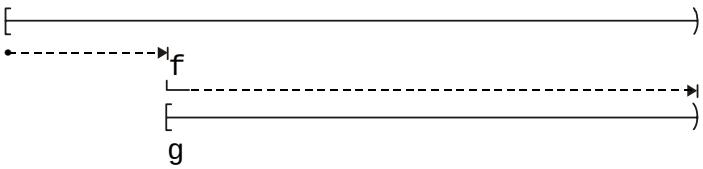
Existen dos tipos especiales de fórmulas de intervalo, que podríamos denominar, atendiendo al tipo de intervalo que se obtiene, como:

- ***Prefijo***. Se tiene una fórmula de este tipo cuando el extremo izquierdo del intervalo que se pretende construir coincide con el del intervalo a partir del cual se extrae. En este caso, sólo es necesario dibujar el patrón de búsqueda que localiza el extremo derecho del intervalo, tal y como se muestra en la siguiente fórmula:



(2.6)

- ***Sufijo***. Este tipo de fórmulas expresa que se requiere que el extremo derecho del intervalo a construir coincida con el del intervalo inmediatamente superior (el que proporciona su contexto de creación). Tal y como se aprecia en la siguiente fórmula, no se pone fórmula objetivo en la búsqueda que nos lleva al final del intervalo. Obsérvese que `f` y `g` se satisfacen en el mismo estado.

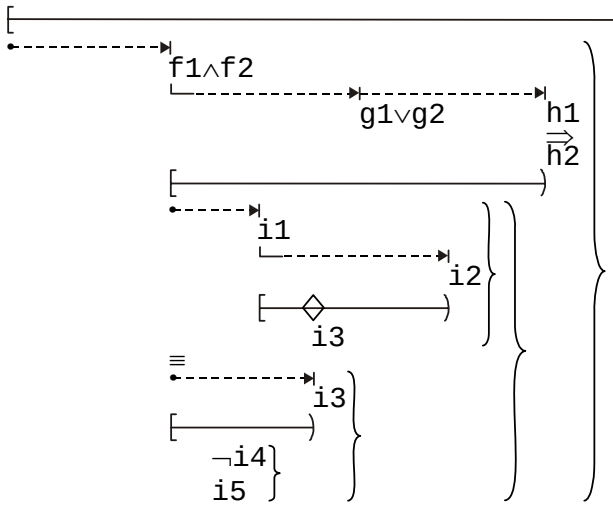


(2.7)

Dado que con frecuencia se requiere establecer fórmulas de intervalo *sufijo* en muchas aplicaciones prácticas, existe una notación especial más abreviada para las mismas. Así, la siguiente fórmula tiene exactamente el mismo significado que la fórmula anterior.

(2.8)

En todos los ejemplos anteriores, cada fórmula `f`, `g`, `h` o `i` puede ser cualquier fórmula GIL. Así, dichas fórmulas pueden reemplazarse por otras más complejas. Por ejemplo, a partir de la fórmula (2.5) se puede obtener la siguiente fórmula:



(2.9)

en la que se usan los operadores lógicos infijos de la Lógica Proposicional, con su semántica habitual: ∧ (conjunción), ∨ (disyunción), ⇒ (implicación), ≡ (equivalencia) y ¬ (negación), para combinar o componer fórmulas. Estos operadores se pueden colocar horizontal o verticalmente, siendo la disposición vertical especialmente útil cuando los operandos de dichas conectivas son fórmulas complejas. Para expresar la conjunción de dos fórmulas, éstas se pueden yuxtaponer verticalmente sin la intervención de ningún operador, dado que ésta es la operación por defecto en el agrupamiento vertical de fórmulas. Por tanto, en la fórmula (2.9) se expresa que la conjunción de ¬`i4` e `i5` es una propiedad invariante sobre su intervalo inferior.

En GIL todas las fórmulas se leen de arriba a abajo y de izquierda a derecha, empezando en el intervalo superior, que representa el contexto global. Además, para facilitar su lectura y eliminar cualquier posible ambigüedad en las fórmulas, se pueden utilizar llaves para delimitarlas (por su parte derecha), haciendo más explícita su estructura. Estas llaves también se emplean para indicar el orden en que deben aplicarse los distintos operadores que intervengan en una fórmula. De este modo, en la fórmula (2.9) se ve claramente que, a partir del estado inicial de la ejecución, se busca en el contexto global un intervalo (delimitado por los estados donde respectivamente se cumplen las fórmulas `f1`∧`f2` y `h1`⇒`h2`), dentro del cual (anidado al mismo) debe satisfacerse, como propiedad inicial, la equivalencia entre las dos fórmulas de intervalo que se muestran en su parte inferior.

Detalles adicionales de la sintaxis visual de GIL pueden encontrarse en [Dillon94a] [Kutty94a].

2.3. Especificación de propiedades en GIL

A la hora de especificar un sistema del mundo real hay que considerar un modelo abstracto de dicho sistema. Este modelo se debe quedar sólo con los aspectos del sistema que sean relevantes de cara a especificar correctamente su funcionamiento (comportamiento temporal). Adoptamos un *modelo basado en estados*, de modo que una secuencia de estados representa una posible ejecución del sistema. Las propiedades o señales directamente observables en el mismo, y que dependen del tiempo, se capturan o representan mediante las *proposiciones atómicas*, que pueden tomar distintos valores de verdad en diferentes instantes de tiempo o estados de una ejecución. GIL es una lógica que está especialmente pensada para la especificación y verificación del comportamiento temporal de los sistemas reactivos en general y de los sistemas concurrentes en particular [Dillon94a] [Kutty93a] [Kutty94b]. Estos sistemas se caracterizan por tener ejecuciones infinitas.

Una especificación (o propiedad que debe cumplir un sistema) se establece imponiendo una serie de restricciones de ordenación temporal sobre los eventos (proposiciones atómicas) que pueden ocurrir en el mismo. Así, sólo aquellas secuencias infinitas de estados que cumplan esa especificación en su primer estado serán consideradas ejecuciones legales o correctas del sistema. Una buena especificación no debe restringir el comportamiento del sistema en aspectos que no sean esenciales o relevantes con respecto a la propiedad que se quiere especificar; en caso contrario, se estaría sobreespecificando, lo que restaría libertad a quien posteriormente tuviera que implementar el sistema.

2.3.1. Refinamiento, abstracción y composición de especificaciones

La transformación de la fórmula (2.5) en la (2.9), efectuada en la sección 2.2, nos muestra que GIL permite el *refinamiento* de especificaciones, o sea, el paso de especificaciones abstractas a otras más concretas, lo que generalmente conlleva el empleo en estas últimas de proposiciones diferentes a aquellas que aparecen en las especificaciones abstractas. Asimismo se permite la *abstracción* de especificaciones, que consiste en aplicar la transformación inversa, esto es, a partir de especificaciones concretas obtener otras más abstractas, relacionando proposiciones de estas últimas con ciertas fórmulas más complejas de las primeras. GIL también permite la *composición* de especificaciones, llevándose a cabo mediante la

conjunción de las mismas, al igual que en otras lógicas [Abadi95], lo que es especialmente útil para expresar la especificación de un sistema concurrente como la conjunción de las especificaciones de los procesos que lo componen, simplificándose así la especificación del sistema global, al poderse descomponer en especificaciones modulares fácilmente relacionables.

Como ya se ha apuntado en la sección 2.2, GIL cuenta con una interpretación reflexiva del futuro, esto es, el futuro de un estado incluye a dicho estado, lo que hace que esta lógica sea *insensible al tartamudeo finito*. Por tanto, la búsqueda de n veces la propiedad `f` a partir de un estado i, dará como resultado un estado j, que será el mismo que se localizaría si esa misma propiedad `f` se hubiera buscado una sola vez a partir del estado i. Esta característica facilita el uso de la abstracción jerárquica y el refinamiento de especificaciones cuando se razona sobre la concurrencia de un sistema [Lampor83].

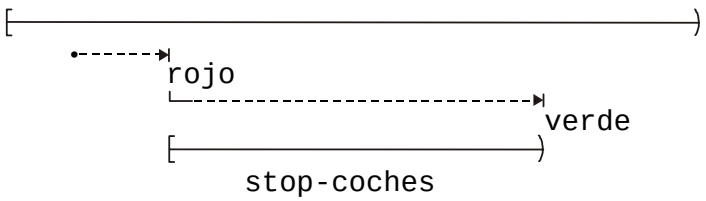
2.3.2. Ejemplos de especificaciones en GIL

Los requisitos funcionales típicos que se necesitan especificar para un determinado sistema, tales como propiedades de seguridad y de vivacidad, se pueden expresar en GIL de una manera intuitiva y, a diferencia de lo que ocurre cuando se utilizan otras lógicas, las especificaciones resultantes son fáciles de comprender. De manera informal, se puede decir que una propiedad de *seguridad* afirma que "algo malo" nunca ocurre, mientras que una propiedad de *vivacidad* afirma que "algo bueno" eventualmente ocurrirá. Una clasificación más completa y rigurosa de los diferentes tipos de propiedades temporales existentes puede encontrarse en [Manna92]. En el Ejemplo 2.1 se especifican algunas propiedades de esos tipos, utilizando GIL y aplicándolas a la especificación de un semáforo. En ningún caso, la lista de propiedades mostradas en dicho ejemplo pretende ser exhaustiva, sino que debe interpretarse como una simple muestra de la gran variedad de propiedades que se pueden especificar usando GIL.

Ejemplo 2.1: Especificaciones de algunas propiedades temporales en GIL

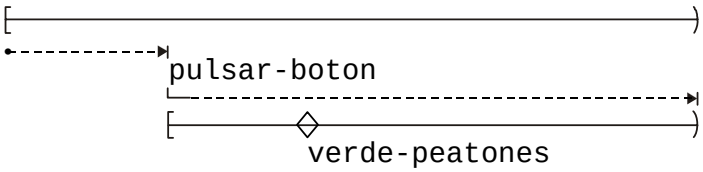
- ***Seguridad***. El significado de la fórmula (2.10) es que siempre que el semáforo esté en `rojo` (intervalo desde que se pone en `rojo` hasta que se enciende de nuevo el `verde`), los coches deben parar. Toda esa fórmula es una propiedad de

seguridad que debe satisfacerse en cada estado de la ejecución. A su vez, `stop-coches` es una propiedad de seguridad que debe cumplirse siempre que `rojo` esté encendido, o sea, está limitada a dicho intervalo.



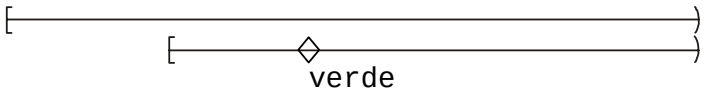
(2.10)

- ***Respuesta***. La fórmula (2.11) afirma que a la petición realizada por un peatón al pulsar el botón, el semáforo eventualmente responderá activando el verde para los peatones.



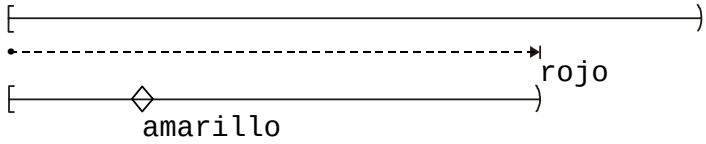
(2.11)

- ***Recurrencia***. La fórmula (2.12) establece que para cada estado de la ejecución, siempre existirá un estado en el futuro en el que el semáforo esté en `verde`, por lo que se dice que es una propiedad recurrente. Esta fórmula también se puede leer como *infinitamente a menudo* se enciende el `verde`.



(2.12)

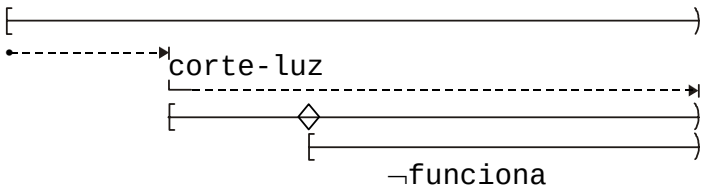
- ***Precedencia***. La fórmula (2.13) afirma que antes de que se encienda el `rojo`, debe encenderse el `amarillo` en algún instante previo.



(2.13)

- ***Persistencia***. Supongamos que el semáforo deja de funcionar indefinidamente si se produce un corte en el suministro eléctrico. El instante en el que deja de funcionar será el mismo en el que se produce el corte (si no está conectado a un acumulador o batería) o instantes después (dependiendo de la duración de dicha batería). La fórmula (2.14) especifica esta propiedad. Se dice que es una propiedad persistente, debido a que describe la *eventual estabilización* de una propiedad (en este caso, $\neg$`funciona`), de modo que se permite un retraso

arbitrario hasta que dicha propiedad se dé, pero una vez que lo haga, ésta debe cumplirse invariantemente durante el resto de la ejecución.



(2.14)

2.3.3. Comparación con la LTL

GIL es exactamente igual de expresiva que el fragmento de la LTL (*Linear Temporal Logic*) [Emerso90] con el operador *hasta (until)*, pero sin el operador *siguiente (next)* [Kutty95].

Las lógicas de intervalos, y en especial la que nosotros utilizamos, permiten representaciones gráficas naturales, que son, por regla general, más intuitivas y fáciles de entender que las representaciones textuales de otras lógicas, tales como la LTL. Algunas fórmulas de esta última son a menudo de difícil comprensión, debido a la presencia en las mismas de operadores *hasta* profundamente anidados. El Ejemplo 2.2 tiene como objetivo ilustrar lo que se acaba de decir, especificando en ambas lógicas (GIL y LTL) un mismo sistema.

Ejemplo 2.2: Especificación de un sistema usando las lógicas GIL y LTL

Supongamos que el comportamiento temporal de un sistema en el que a, b, c y d son las señales observables viene dado por los siguientes requisitos o restricciones:

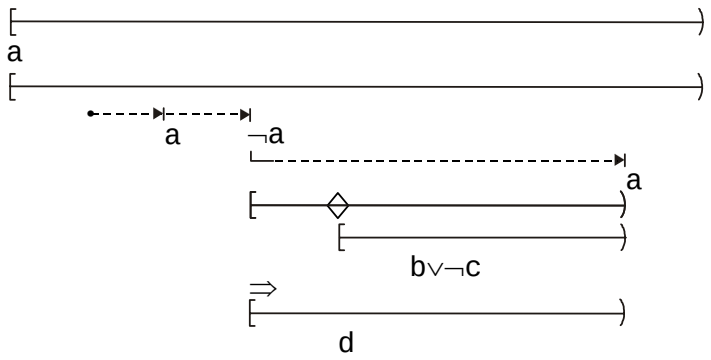
- La propiedad a se cumple inicialmente.
- Para cada intervalo maximal finito en el que a no se satisface, si hay un punto dentro del mismo a partir del cual se cumple b o ¬c en cada estado del resto del intervalo, entonces d se satisface invariantemente en todo el intervalo.

Haciendo uso de la LTL, la fórmula (2.15) muestra una especificación de este comportamiento. En ella, U, $\mathcal{W}$ y P respectivamente representan los operadores

hasta débil, *hasta fuerte* y *precede a*, definiéndose éste último como f_1 P f_2 = $\neg(\neg f_1$ U $f_2)$.

$$a \wedge \Box(a \,U\, (\neg a \,\mathcal{W}\, (((b \vee \neg c) \,\mathcal{W}\, a) \,P\, a) \Rightarrow d \,U\, a)) \tag{2.15}$$

Este mismo comportamiento se especifica mediante la siguiente fórmula GIL:



(2.16)

No es fácil ver que la fórmula (2.15) representa el comportamiento deseado. Esto es debido a que la LTL no tiene ni noción semántica de intervalo ni ningún elemento sintáctico que exprese concisamente tal noción, por lo que hay que recurrir al anidamiento de operadores *hasta*, que dificulta bastante la comprensión de las fórmulas.

Sin embargo, ese mismo comportamiento se expresa de forma más natural y comprensible en la representación gráfica de GIL, tal y como muestra la fórmula (2.16). Según la notación explicada en la sección 2.2, dicha expresión representa la conjunción de dos fórmulas: la propiedad inicial a y una propiedad invariante (sobre la ejecución completa) que localiza todos y cada uno de los intervalos maximales donde a es ininterrumpidamente falsa, y para cada uno de ellos indica que si se cumple b∨¬c continuamente sobre un sufijo del intervalo, entonces la propiedad d se satisface en cada estado del intervalo maximal.

Como conclusión, puede decirse que los intervalos en GIL permiten construir sentencias más sucintas y comprensibles que las que admite la LTL con su operador *hasta*. La representación natural y visualmente intuitiva de las especificaciones GIL hace que éstas sean más fáciles de desarrollar y de comprender que la representación textual de lógicas temporales más tradicionales,

incluso para aquellas personas involucradas en el proceso de desarrollo de sistemas (tanto hardware como software) que no tengan una buena formación matemático-lógica. Esto es debido a que GIL es un formalismo lógico que se ajusta bastante bien a la manera en que los seres humanos razonan. Por tanto, la utilización de GIL pretende reducir las reticencias que los diseñadores de sistemas tienen a la hora de usar métodos formales para la especificación de los mismos, acercando el lenguaje de especificación al modo en que los diseñadores razonan, para que éstos consideren dicho lenguaje como una herramienta útil para desarrollar su trabajo y no como algo difícil de utilizar para lograr sus propósitos.

2.4. FIL como base formal de GIL

FIL (*Future Interval Logic*) [Ramakr92] [Ramakr93a] [Ramakr96a], denominada así porque las búsquedas utilizadas para la construcción de los intervalos siempre se realizan en el futuro, nunca en el pasado. FIL es la base formal de GIL, esto es, ambas lógicas comparten la misma semántica, de modo que existe una correspondencia clara (tal y como se verá en el apartado 2.4.5) entre las fórmulas textuales de FIL y sus correspondientes fórmulas gráficas en GIL.

2.4.1. Sintaxis

Definición 2.1: Sintaxis de FIL

La *sintaxis de FIL*, o sea, las reglas que permiten determinar el conjunto $\mathscr{L}_{\mathrm{FIL}}(\mathscr{P})$ de todas las fórmulas bien formadas de la lógica para un determinado conjunto finito $\mathscr{P}$ de proposiciones atómicas, donde $p \in \mathscr{P}$, se define, utilizando la notación BNF (*Backus-Naur Form*) [Naur60], del siguiente modo:

$$
\begin{array}{lcl}
f & ::== & p \mid \neg f \mid f_1 \wedge f_2 \mid I f \\
I & ::== & [\theta_1 \mid \theta_2) \mid [- \mid \theta_2) \mid [\theta_1 \mid \rightarrow) \\
\theta & ::== & \rightarrow f \mid \rightarrow f, \theta
\end{array}
$$

■

Como puede apreciarse, en esta definición se distinguen claramente tres *categorías sintácticas*, que son:

- La de las *fórmulas bien formadas* de la lógica, definida por la primera regla de producción. El conjunto de todas las fórmulas bien formadas de FIL,

para el conjunto de proposiciones $\mathscr{P}$, se denotará a partir de ahora como **fbf**($\mathscr{P}$).

- La de las *modalidades de intervalo*, especificada mediante la segunda regla, y que se denotará como **mdintv**($\mathscr{P}$).
- La de los *patrones de búsqueda*, determinada por la última de las reglas, y que identificaremos mediante el conjunto **ptrbq**($\mathscr{P}$). Este conjunto incluirá también los dos patrones de búsqueda triviales: – y $\rightarrow$, cuyo significado se explicará informalmente en el apartado 2.4.2 y se formalizará en el apartado 2.4.4 de esta sección.

Definición 2.2: Fórmula puramente proposicional y fórmula de intervalo

Se dice que una fórmula FIL es *puramente proposicional* cuando no contiene ninguna modalidad de intervalo, o sea, cuando esa fórmula pertenece a la Lógica Proposicional clásica. Por el contrario, se dice que es una *fórmula de intervalo* si su estructura viene dada por $I\,f$, donde I representa una modalidad de intervalo y f representa a cualquier otra fórmula FIL anidada a dicho intervalo. ■

Ejemplo 2.3: Fórmula puramente proposicional y fórmula de intervalo

Sean a, b y c proposiciones atómicas y $f = \neg(a \wedge \neg b) \wedge c$ y $\eta = [\rightarrow a \mid \rightarrow \neg b, \rightarrow c)\neg c$ dos fórmulas FIL, siendo la primera de ellas, f, puramente proposicional y la otra, η, una fórmula de intervalo, al empezar esta última con una modalidad de intervalo, cuyo primer patrón de búsqueda está formado por una sola búsqueda ($\theta_1 = \rightarrow a$), mientras que su segundo patrón contiene dos búsquedas ($\theta_2 = \rightarrow \neg b, \rightarrow c$).

Las reglas de producción de la Definición 2.1 establecen una *sintaxis restringida* para FIL. Esta sintaxis se puede extender, como es habitual, definiendo una serie de operadores derivados, que se utilizarán como abreviaturas para determinadas fórmulas, permitiendo que la expresión de dichas fórmulas sea más concisa y clara. Algunas de estas extensiones se definen como sigue:

$$\neg\neg f \quad = \quad f$$
$$f \vee g \quad = \quad \neg(\neg f \wedge \neg g)$$

$$f \Rightarrow g \quad = \quad \neg f \vee g$$
$$f \equiv g \quad = \quad (f \Rightarrow g) \wedge (g \Rightarrow f)$$

De igual modo, las constantes lógicas serán abreviaciones para las siguientes expresiones:

$$\mathbf{T} \quad = \quad p \vee \neg p$$
$$\mathbf{F} \quad = \quad \neg \mathbf{T}$$

También pueden definirse algunos de los operadores temporales de la LTL. Así:

$$\Box f \quad = \quad [\rightarrow \neg f \mid \rightarrow) \mathbf{F}$$
$$\Diamond f \quad = \quad \neg [\rightarrow f \mid \rightarrow) \mathbf{F}$$
$$f_1 \mathsf{U} f_2 \quad = \quad [\rightarrow (\neg f_1 \vee f_2) \mid \rightarrow) f_2$$
$$f_1 \mathcal{U} f_2 \quad = \quad \neg [\rightarrow (\neg f_1 \vee f_2) \mid \rightarrow) \neg f_2$$

donde $\Box$ representa el operador denominado *siempre* o *de ahora en adelante* (*henceforth*), $\Diamond$ es el operador *eventualmente* (*eventually*), U es el operador *hasta débil* (*weak until*) y $\mathcal{U}$ es el operador *hasta fuerte* (*strong until*). El significado de las fórmulas FIL que aparecen a la derecha de las definiciones anteriores se explicará en el siguiente apartado, dentro del Ejemplo 2.4.

Nuevos operadores (en concreto, los de *búsqueda fuerte* e *intervalo fuerte*), que simplifican la especificación de determinadas propiedades, se definen en el Apéndice B como abreviaciones de expresiones que contienen los operadores presentados.

2.4.2. Semántica informal

Intuitivamente, una *modalidad de intervalo* $[\theta_1 \mid \theta_2)$ identifica una subsecuencia de estados contiguos (es decir, una subtraza[2]) de la traza considerada, que representa una de las posibles ejecuciones del sistema. De esta forma, dada la fórmula $[\theta_1 \mid \theta_2) f$, dicha subtraza especifica la estructura sobre la que la subfórmula anidada, f, debe cumplirse. Dicho de otro modo, un intervalo proporciona un *contexto* en el que se debe evaluar la fórmula anidada.

[2] En la definición formal de la semántica (ver apartado 2.4.4) se identificará esa subtraza (que es una traza finita) con la traza infinita obtenida al repetir indefinidamente su último estado. A esta operación de repetición se le denomina *tartamudeo* (*stuttering*).

Tal y como se ha dicho en la sección 2.2, la semántica o significado de una *búsqueda*, por ejemplo $\rightarrow f$, que comienza en un estado determinado de la traza, es que localiza el primer estado del futuro reflexivo en el que su fórmula objetivo, f, se cumple. Cuando varias búsquedas se componen secuencialmente dentro de un mismo *patrón de búsqueda no trivial*, como por ejemplo en $\rightarrow f, \rightarrow \neg f$, cada búsqueda subsiguiente comienza en el estado que la búsqueda previa localizó.

En el caso de que la fórmula objetivo de una búsqueda no se cumpla en ningún punto del futuro (no estricto) a partir del estado donde la búsqueda comenzó, dentro del contexto actual o intervalo previamente establecido, se dice que *la búsqueda falla* o *fracasa*. Como consecuencia, el intervalo no puede construirse, dado que al menos uno de sus extremos no se puede encontrar. Tampoco podría construirse el intervalo si su extremo derecho es localizado antes o en el mismo estado que su extremo izquierdo; en este caso, se dice que el *intervalo* es *vacío* o que el intervalo *se colapsa*. En ambos casos (en los que un intervalo no se puede construir) se produce como resultado el *contexto nulo*.

Por consiguiente, y con el ánimo de precisar aún más, se puede decir que la modalidad de intervalo $[\theta_1 \mid \theta_2)$ define un contexto, que es o el contexto nulo o la subsecuencia que comienza en el estado localizado por las búsquedas especificadas en θ_1 y que termina en el estado que precede al localizado por las búsquedas que componen θ_2. Cuando para dicha modalidad de intervalo se produce el *contexto nulo* (se determina que ese intervalo no se puede construir, porque una de sus búsquedas falla o porque el estado localizado por θ_1 no precede al localizado por θ_2), entonces se asume que la fórmula $[\theta_1 \mid \theta_2)f$ se satisface *vacuamente*. Así, la interpretación semántica de $[\theta_1 \mid \theta_2)f$ sería: "si el subcontexto $[\theta_1 \mid \theta_2)$ puede ser identificado dentro del contexto actual, entonces f debe cumplirse dentro de dicho subcontexto". Esta *semántica*, que se podría calificar como *por defecto a verdadero*, produce el siguiente significado para $\neg[\theta_1 \mid \theta_2)f$ en un estado: "un intervalo de la forma $[\theta_1 \mid \theta_2)$ existe en el futuro reflexivo y f no se cumple en el primer estado de ese intervalo".

Los *patrones de búsqueda triviales* – y $\rightarrow$ tienen el siguiente significado: – nos deja en el punto donde estamos y $\rightarrow$ nos lleva al final del contexto actual. Por tanto, se puede decir que toda modalidad de intervalo del tipo $[- \mid \theta_2)$, denominada *modalidad de intervalo actual*[3], intenta construir un *prefijo del contexto actual*,

[3] Se denomina así debido a que empieza en el estado actual.

empezando en el instante actual o punto de evaluación de la fórmula y extendiéndose hasta el estado localizado por θ_2, pero sin incluirlo. Por el contrario, las modalidades de intervalo del tipo $[\theta_1 | \rightarrow)$ intentan construir un *sufijo del contexto actual*, empezando en el estado localizado por θ_1 y extendiéndose hasta, e incluyendo, el último estado del contexto actual.

Aplicando lo explicado, en el Ejemplo 2. 4 se exponen las interpretaciones semánticas de algunas fórmulas FIL especialmente interesantes, entre ellas las que corresponden a las definiciones de los principales operadores temporales de la LTL.

Ejemplo 2.4: Interpretaciones semánticas de algunas fórmulas FIL

$[\rightarrow\neg f \mid \rightarrow)\,\mathbf{F}$	Dado que **F** no puede cumplirse en ningún contexto, excepto en el contexto nulo, esta fórmula nunca se puede satisfacer en una traza que comience en un punto tal que $\neg f$ se cumpla en algún instante de su futuro reflexivo. En otras palabras, esta fórmula sólo se satisface cuando f se cumple invariantemente desde el estado actual en adelante, por lo que es equivalente a la fórmula $\Box f$ de la LTL.
$\neg[\rightarrow f \mid \rightarrow)\,\mathbf{F}$	Es la fórmula dual a la anterior, afirmando que hay algún estado en el futuro (no estricto) donde f se cumple, por lo que es equivalente a la fórmula $\Diamond f$ de la LTL.
$[\rightarrow(\neg f_1 \vee f_2) \mid \rightarrow)\, f_2$	Esta fórmula es cierta si: (a) no se puede construir el intervalo (fracasa la búsqueda de su extremo izquierdo), lo que significa que desde su punto de evaluación en adelante se cumple invariantemente $f_1 \wedge \neg f_2$; o (b) si se puede construir el intervalo (tiene éxito la búsqueda de su extremo izquierdo) y f_2 se cumple en el primer instante del mismo, con lo que se daría f_1 hasta el estado en el que se satisface f_2. Por tanto, esta fórmula expresa lo mismo que la fórmula $f_1 \,\mathsf{U}\, f_2$ de la LTL, donde U es el operador *hasta débil*.
$\neg[\rightarrow(\neg f_1 \vee f_2) \mid \rightarrow)\neg f_2$	Al estar negado el intervalo, esta fórmula sólo es cierta si dicho intervalo puede construirse y f_2 ($\equiv\neg\neg f_2$) se satisface en

	su primer estado. Obsérvese que f_1 se cumple desde el punto de evaluación hasta que se da f_2 y que esta última debe satisfacerse obligatoriamente en algún estado del futuro reflexivo. Así, esta fórmula es exactamente equivalente a la fórmula $f_1 \mathcal{U} f_2$ de la LTL, donde $\mathcal{U}$ es el operador *hasta fuerte*.
$[-\mid\rightarrow f_2)f_1$	Afirma que, si f_2 se cumple en algún instante del futuro (estricto), f_1 debe cumplirse en el primer estado del intervalo (prefijo del contexto actual) que queda después de quitar el primer sufijo que satisface f_2 en su primer estado[4].

2.4.3. Formalización del concepto de modelo

Puesto que empleamos una lógica de tiempo discreto y lineal, siguiendo la tradición de la lógica temporal lineal, una fórmula f de nuestra lógica (GIL o FIL) se interpreta sobre un *modelo* o secuencia de estados, cuya formalización se establece en la siguiente definición:

Definición 2.3: Modelo

Un *modelo* $\sigma\mathcal{M}$ es una ω-traza lineal del tipo $\sigma\mathcal{M} = \langle \sigma\mathcal{M}(i) \rangle_{i \in \omega}$, donde cada elemento de la secuencia, $\sigma\mathcal{M}(i)$, representa un *estado* que asigna un valor de verdad o falsedad a cada una de las proposiciones primitivas que integran el conjunto $\mathcal{P}$. Se trata, pues, de un modelo basado en estados, donde cada estado $\sigma\mathcal{M}(i)$ puede ser considerado como el conjunto de proposiciones atómicas que son verdaderas en ese instante. Así, un modelo se puede representar como una correspondencia del tipo $\sigma\mathcal{M}: \omega \rightarrow 2^{\mathcal{P}}$. ■

Como es usual, asumimos (sin pérdida de generalidad) que cada modelo representa una secuencia infinita de estados, dado que una traza finita puede ser modelada mediante la traza infinita obtenida al repetir mediante *tartamudeo* (*stuttering*) su último estado, tal y como se ha dicho en la sección 2.1.

[4] Obsérvese que esto no es equivalente a la fórmula de la *Lógica de Procesos* $f_1 \mathcal{C} f_2$, donde $\mathcal{C}$ es el operador *corte* (*chop*), que requiere que haya algún punto en la traza tal que f_1 se satisfaga sobre el prefijo hasta ese punto y que f_2 se cumpla sobre su sufijo.

Por consiguiente, a partir de ahora utilizaremos los términos *modelo*, (*ω*-)*traza* y *secuencia de estados* como sinónimos y, a menos que se especifique lo contrario, harán referencia a una traza infinita. De igual modo, también se emplearán como sinónimos los vocablos *estado*, *instante* y *punto* de un modelo.

Una fórmula *f* puede evaluarse en cualquier estado de un modelo σ. Por tanto, la búsqueda de una fórmula, así como la construcción de un determinado subcontexto (intervalo) puede empezar en cualquier estado del modelo. En las definiciones del siguiente apartado se formalizan estas operaciones; en ellas, $\langle\sigma, i\rangle$ denota que la operación correspondiente se evalúa en el instante *i* del modelo σ. Además, en ellas se requiere la noción de un modelo especial, denominado *modelo nulo* y representado como $\sigma_\perp$, que es aquél que trivialmente satisface cualquier fórmula bien formada.

2.4.4. Semántica formal

La semántica formal de FIL (establecida en la Definición 2.6) se apoya en dos funciones cuyas definiciones se dan a continuación: la función *búsqueda* $\mathcal{B}$, que localiza el punto resultante de una búsqueda, y la función *subcontexto* $\mathcal{S}$, que construye un subcontexto (intervalo) dentro del contexto actual.

Intuitivamente, la función $\mathcal{B}$ es una función parametrizada, tal que dados un patrón de búsqueda θ, un modelo σ y un punto *i*, da como resultado el punto *j* del modelo σ, localizado a partir del punto *i* del mismo por las búsquedas que componen el patrón de búsqueda θ. Por otra parte, dados un modelo σ y dos puntos *i* y *j* del mismo, la función subcontexto $\mathcal{S}$ produce el modelo σ', que consiste en el intervalo de σ que contiene los puntos que van desde el *i* hasta el inmediatamente anterior al *j*, repitiendo su último estado para obtener una secuencia infinita.

Notación: En las siguientes definiciones formales se utilizará $\perp$ para representar un *punto indefinido*, cuyo significado es que no se ha podido localizar la fórmula objetivo de una búsqueda. Además, se empleará ω para denotar el primer ordinal límite del conjunto de puntos que integran el contexto (modelo) actual, o sea, el conjunto $\{0, 1, 2, ..., \omega-1\}$ formado por todos los números ordinales finitos que le preceden. Cuando el rango de puntos posibles englobe al propio límite, se utilizará $\omega+1$, denotando el conjunto $\{0, 1, 2, ..., \omega\}$. Por otra parte,

mediante el símbolo $\cup$ se representará la unión disjunta de dos conjuntos, mientras que min hará referencia al elemento minimal de un conjunto ordenado de puntos.

Definición 2.4: Función búsqueda

La función *búsqueda*

$$\mathscr{B}\text{: } \mathbf{ptrbq}(\mathscr{P}) \times (2^{\mathscr{P}})^{\omega} \cup \{\circ\mathscr{M}_{\perp}\} \times \omega \cup \{\perp\} \to (\omega+1) \cup \{\perp\}$$

se define como sigue:

- Si $\circ\mathscr{M} = \circ\mathscr{M}_{\perp}$ o $i = \perp$, entonces $\mathscr{B}(\theta, \langle \circ\mathscr{M}, i \rangle) = \perp$
- Si $\circ\mathscr{M} \neq \circ\mathscr{M}_{\perp}$ e $i \neq \perp$, entonces:

$$\mathscr{B}(-, \langle \circ\mathscr{M}, i \rangle) = i$$

$$\mathscr{B}(\to, \langle \circ\mathscr{M}, i \rangle) = \omega$$

$$\mathscr{B}(\to a, \langle \circ\mathscr{M}, i \rangle) = \begin{cases} \perp & \text{si } \forall j \geq i, \langle \mathscr{M}, j \rangle \not\models a \\ \min\{ j \mid j \geq i, \langle \mathscr{M}, j \rangle \models a \} & \text{en cualquier otro caso} \end{cases}$$

$$\mathscr{B}(\to a, \theta, \langle \circ\mathscr{M}, i \rangle) = \mathscr{B}(\theta, \langle \circ\mathscr{M}, \mathscr{B}(\to a, \langle \circ\mathscr{M}, i \rangle) \rangle) \qquad \blacksquare$$

Definición 2.5: Función subcontexto

La función *subcontexto*

$$\mathscr{S}\text{: } \mathbf{mdintv}(\mathscr{P}) \times (2^{\mathscr{P}})^{\omega} \cup \{\circ\mathscr{M}_{\perp}\} \times \omega \to ((2^{\mathscr{P}})^{\omega}) \cup \{\perp\}$$

se define como

$$\mathscr{S}([\theta_1 \mid \theta_2), \langle \circ\mathscr{M}, i \rangle) = \circ\mathscr{M}[\mathscr{B}(\theta_1, \langle \circ\mathscr{M}, i \rangle), \mathscr{B}(\theta_2, \langle \circ\mathscr{M}, i \rangle))$$

donde $\circ\mathscr{M}[i, j)$, con $i, j \in (\omega + 1) \cup \{\perp\}$, denota el *modelo* que representa el *subcontexto* que se pretende construir, y que se define como sigue:

$$\circ\mathscr{M}[i, j) = \circ\mathscr{M}_{\perp} \quad \text{si } i = \perp \text{ o } j = \perp \text{ o } j \leq i$$

$$\circ\mathscr{M}[i, j)\,(k) = \left\{ \begin{array}{ll} \mathscr{M}(k + i), & \forall k;\ 0 \leq k < j - i \\ \mathscr{M}(j - 1), & \forall k;\ j - i \leq k < \omega \end{array} \right\} \quad \text{en otro caso}$$

donde k representa un estado o instante *local* (es decir, dentro del subcontexto construido) y donde el modelo resultante es una traza infinita, puesto que se repite el valor de su último estado para todos aquellos puntos que queden a su derecha. ■

Haciendo uso de las definiciones que se acaban de dar, la siguiente definición formaliza la semántica de FIL para la sintaxis restringida que se estableció en la Definición 2.1.

Definición 2.6: Semántica de FIL

La *semántica de FIL* interpreta si una fórmula perteneciente a $\mathscr{L}_{\text{FIL}}(\mathscr{P})$ se cumple o no en un punto $i \in \omega$ de un modelo $\sigma\mathscr{M} \in (2^{\mathscr{P}})^{\omega} \cup \{\sigma\mathscr{M}_{\perp}\}$, usando para ello la *relación de satisfacibilidad*

$$\models \subseteq (2^{\mathscr{P}})^{\omega} \cup \{\sigma\mathscr{M}_{\perp}\} \times \omega \times \mathbf{fbf}(\mathscr{P})$$

que se define del siguiente modo:

- Si $\sigma\mathscr{M} = \sigma\mathscr{M}_{\perp}$, entonces $\langle \sigma\mathscr{M}, i\rangle \models f, \forall f \in \mathbf{fbf}(\mathscr{P})$ y $\forall i \in \omega$
- Si $\sigma\mathscr{M} \neq \sigma\mathscr{M}_{\perp}$, entonces:

$$\begin{aligned}
&\langle \sigma\mathscr{M}, i\rangle \models \mathbf{T}, && \forall i \in \omega \\
&\langle \sigma\mathscr{M}, i\rangle \models p && \text{sii } p \in \sigma\mathscr{M}(i), \text{ donde } p \in \mathscr{P} \\
&\langle \sigma\mathscr{M}, i\rangle \models \neg f && \text{sii } \langle \sigma\mathscr{M}, i\rangle \not\models f \\
&\langle \sigma\mathscr{M}, i\rangle \models f \wedge g && \text{sii } \langle \sigma\mathscr{M}, i\rangle \models f \text{ y } \langle \sigma\mathscr{M}, i\rangle \models g \\
&\langle \sigma\mathscr{M}, i\rangle \models I f && \text{sii } \langle \sigma\mathscr{M}', 0\rangle \models f, \text{ donde } \sigma\mathscr{M}' = \mathscr{S}(I, \langle \sigma\mathscr{M}, i\rangle)
\end{aligned}$$ ■

Por último, se definirán las nociones de satisfacibilidad y validez de una fórmula, así como la de satisfacibilidad de un conjunto de fórmulas; todas ellas en base a las relación establecida en la definición anterior.

Definición 2.7: Satisfacibilidad de una fórmula

Una fórmula f es *satisfacible* si y sólo si existe algún modelo $\sigma\mathscr{M} \in (2^{\mathscr{P}})^{\omega}$ tal que $\langle \sigma\mathscr{M}, 0\rangle \models f$, en cuyo caso se dice que el modelo $\sigma\mathscr{M}$ *satisface* la fórmula f. ■

Obsérvese que esta definición de satisfacibilidad se corresponde con la interpretación *anclada* de las fórmulas temporales [Manna89], en la que las fórmulas se evalúan en el primer estado de un modelo.

Definición 2.8: Validez de una fórmula

Una fórmula f es *válida*, y se denota como $\models f$, si y sólo si cada modelo del conjunto $(2^{\mathcal{P}})^{\omega}$ satisface f. ■

Es fácil ver que la relación existente entre validez y satisfacibilidad de una fórmula es la siguiente: una fórmula f es válida exactamente cuando $\neg f$ es no satisfacible.

Definición 2.9: Satisfacibilidad de un conjunto de fórmulas

Un conjunto finito de fórmulas $S=\{f_1, f_2, ..., f_n\}$ es satisfacible si y sólo si la conjunción de todos sus elementos, $f_1 \wedge f_2 \wedge ... \wedge f_n$, es satisfacible. El conjunto S vacío siempre es satisfacible (o sea, cualquier modelo y en cualquier instante lo satisface), dado que la conjunción vacía se identifica con la constante lógica **T**. ■

2.4.5. Equivalencia entre las representaciones de GIL y FIL

Al principio de la sección 2.1 se ha indicado que la lógica empleada tiene dos representaciones posibles: una de tipo gráfico (usando fórmulas GIL) y otra de tipo textual (utilizando fórmulas FIL). De todo lo dicho en este capítulo se deduce que existe una correspondencia clara entre las fórmulas de GIL y las de FIL. Así, los patrones de búsqueda y los intervalos en GIL se corresponden directamente con los de FIL, mientras que otros constructos de GIL, como los que expresan una propiedad invariante o una eventualidad, se corresponden con ciertos operadores de la sintaxis extendida de FIL (□ y ◊ en los ejemplos citados). Con el fin de poner de manifiesto dicha correspondencia de forma explícita, en la Tabla 2.1 se presentan las fórmulas textuales que representan en FIL exactamente lo mismo que las correspondientes fórmulas gráficas representadas en GIL a lo largo de todo este capítulo.

Tabla 2.1. Representación en FIL de las fórmulas gráficas mostradas (en GIL) en este capítulo

FIL	GIL
f	(2.1)
□f	(2.2)
◊f	(2.3)
[→f | →g,→h)□i	(2.4)
[→f | →f,→g,→h)i	(2.5)
[– | →f)g	(2.6)
[→f | →)g	(2.7)(2.8)
[→f1∧f2|→f1∧f2,→g1∨g2,→h1⇒h2)([→i1|→i1,→i2)◊i3≡[–|→i3)□(¬i4∧i5))	(2.9)
□[→rojo | →rojo,→verde) □stop-coches	(2.10)
[→pulsar-boton | →) ◊ verde-peatones	(2.11)
□◊verde	(2.12)
[– | →rojo) ◊amarillo	(2.13)
[→corte-luz | →) ◊□¬funciona	(2.14)
a ∧ □[→a,→¬a | →a,→¬a,→a) (◊□(b∨¬c)⇒□d)	(2.16)

La correspondencia existente entre estas fórmulas textuales y sus representaciones gráficas homólogas es fácil de ver. No obstante, en [Dillon94a] se da una demostración formal de la *equivalencia* que existe entre las fórmulas textuales de FIL y su representación gráfica en GIL, así como de la no ambigüedad del lenguaje gráfico.

La semántica del lenguaje $\mathscr{L}_{\mathrm{GIL}}(\mathscr{P})$ de las fórmulas bien formadas de GIL relativas al conjunto de proposiciones atómicas $\mathscr{P}$ se obtiene traduciendo sus fórmulas al lenguaje $\mathscr{L}_{\mathrm{FIL}}(\mathscr{P})$ y aplicando la semántica definida formalmente en el apartado anterior para este último lenguaje. Además, las fórmulas GIL se interpretan sobre el mismo conjunto de modelos que las de FIL y las nociones de satisfacibilidad y validez de fórmulas se aplican de igual modo en ambas lógicas. Por tanto, FIL y GIL sólo se diferencian en la sintaxis o representación utilizada en sus fórmulas (textual y gráfica respectivamente), compartiendo ambas la misma

semántica, o sea, tienen el mismo poder expresivo, por lo que se puede decir que son, en esencia, la misma lógica.

Debido a que un mismo contenido semántico se puede expresar en GIL de varias formas (por ejemplo, colocando algunas subfórmulas vertical u horizontalmente), dos fórmulas con distintas representaciones en GIL pueden tener la misma traducción a FIL, al ser estructuralmente equivalentes. Un ejemplo de esto se muestra en la Tabla 2.1 , donde las fórmulas (2.7) y (2.8) en GIL se corresponden con la misma fórmula FIL.

La idea es desarrollar un editor gráfico, similar al implementado en [Kutty93b] [Kutty94a], de modo que el diseñador del sistema pueda realizar la especificación de propiedades directamente en GIL, de tal manera que esta herramienta traduzca automáticamente las fórmulas GIL que constituyen dicha especificación en sus equivalentes fórmulas FIL. Estas últimas serán las utilizadas por nuestro algoritmo (presentado en el Capítulo 5) para verificar que el sistema cumple esa especificación.

Por todo ello y por motivos de compactación y facilidad tipográfica, la representación textual será la que se utilizará mayormente a lo largo del resto de esta memoria, por lo que a partir de ahora FIL será el acrónimo que se utilizará con más frecuencia para hacer referencia a la lógica empleada.

2.5. Descripción de los principales términos lógicos empleados

Con el ánimo de que el material presentado en esta memoria sea autocontenido, a continuación se recogen y describen algunos de los términos lógicos más generales que se utilizan en la misma. El lector familiarizado con el paradigma de la lógica temporal lineal puede perfectamente saltarse esta sección, sin que eso afecte ni a la lectura ni a la comprensión del resto de esta memoria.

Tal y como ya se ha comentado, el *lenguaje* de una lógica L para el conjunto de proposiciones atómicas $\mathscr{P}$ se denota como $\mathscr{L}_L(\mathscr{P})$ y representa al conjunto de fórmulas bien formadas definido por su *sintaxis*, o sea, por las reglas de formación de sentencias correctas que constituyen dicha lógica. La *semántica* de la lógica L asocia una interpretación a cada una de las sentencias del lenguaje $\mathscr{L}_L(\mathscr{P})$ sobre un

conjunto o universo de *modelos* (concepto formalizado en el apartado 2.4.3), indicando si una fórmula $f \in \mathscr{L}_L(\mathscr{P})$ es cierta (se cumple) o no sobre un determinado modelo.

Sobre un determinado universo de modelos, una *teoría* T es el subconjunto de sentencias del lenguaje de la lógica (esto es, $T \subseteq \mathscr{L}_L(\mathscr{P})$) que son *válidas* en el conjunto de modelos que componen ese universo, o sea, sentencias que son ciertas cuando se interpretan sobre cualquier modelo de dicho universo. Por consiguiente, cualquier sentencia de una teoría es una *tautología*. Una teoría T es *completa* si para cualquier fórmula bien formada de la lógica $f \in \mathscr{L}_L(\mathscr{P})$, o $f \in T$ o $\neg f \in T$. Se dice que T es *inconsistente* si hay alguna sentencia $f \in \mathscr{L}_L(\mathscr{P})$, tal que $f \in T$ y $\neg f \in T$, siendo *consistente* en caso contrario. Una teoría T es *decidible* si existe un algoritmo, a menudo denominado *procedimiento de decisión*, para determinar la pertenencia de cualquier sentencia del lenguaje $\mathscr{L}_L(\mathscr{P})$ a la teoría T, es decir, dada una fórmula $f \in \mathscr{L}_L(\mathscr{P})$, el procedimiento de decisión debe terminar afirmando o negando la pertenencia de f a T. Por el contrario, T es *indecidible* si no se puede construir dicho algoritmo. Por extensión, se dice que una lógica L es decidible o indecidible según que exista o no el mencionado algoritmo. La indecibilidad de una lógica sólo significa eso (imposibilidad de construir un procedimiento de decisión para la misma) y no que su semántica sea ambigua.

El problema que resuelve el procedimiento de decisión de una lógica es el de la *validez* de una determinada sentencia o fórmula bien formada f de su lenguaje, o sea, si f es cierta para *cada* modelo del universo considerado. Su dual es el problema de *satisfacibilidad*, que determina si existe *algún* modelo (en ese universo) en el que f sea cierta.

Una *axiomatización* $\mathscr{A}(T)$ para una teoría T es un conjunto de *axiomas* (sentencias básicas de esa teoría), junto con un conjunto de *reglas de inferencia* (pares de sentencias de T que sirven para derivar unas sentencias a partir de otras). La *teoría deductiva* $\mathscr{D}(\mathscr{A}(T))$ correspondiente a una axiomatización $\mathscr{A}(T)$ es el conjunto de sentencias derivables desde sus axiomas al aplicar finitamente sus reglas de inferencia. Cuando $\mathscr{D}(\mathscr{A}(T)) \subseteq T$ se dice que la axiomatización es *sólida*. Cuando $\mathscr{D}(\mathscr{A}(T))=T$ entonces la axiomatización es también *completa*. En determinados contextos, donde un sistema deductivo es visto sólo como un medio para generar una teoría T, un procedimiento de decisión para T puede ser considerado como una axiomatización para dicha teoría.

Una determinada *propiedad* se puede identificar con un conjunto de modelos (formado por todos aquellos que la cumplen). Una lógica puede *expresar una propiedad* si y sólo si hay alguna sentencia en su lenguaje tal que el conjunto de modelos que la satisfacen es exactamente el definido por dicha propiedad. Sobre un determinado universo o conjunto de modelos, dos lógicas L_1 y L_2 tienen el *mismo poder expresivo* (también se dice que son *expresivamente equivalentes*) si cualquier propiedad expresable en L_1 es también expresable en L_2 y viceversa. Por el contrario, L_1 es (estrictamente) *más expresiva* que L_2 si L_1 puede expresar cada propiedad expresable en L_2, pero no viceversa.

ÍNDICE DE REFERENCIAS EMPLEADAS

[Abadi95], *21*
[Dillon94], *15*, *19*, *20*, *35*
[Emerso90], *23*
[Kutty93a], *20*
[Kutty93b], *36*
[Kutty94a], *19*, *36*
[Kutty94b], *20*
[Kutty95], *23*
[Lampor83], *21*
[Manna89], *34*
[Manna92], *21*
[Naur60], *25*
[Ramakr92], *15*, *25*
[Ramakr93], *25*
[Ramakr96], *25*

Capítulo 3

TÉCNICAS AUTOMATIZADAS DE VERIFICACIÓN

CONTENIDO

RESUMEN Y ORGANIZACIÓN

Si en el capítulo anterior se estableció el marco que definía el formalismo lógico empleado en esta tesis, el presente capítulo hace lo propio con respecto a la técnica de verificación en la que se enmarca nuestro trabajo, estudiándose además las principales técnicas automatizadas que se utilizan hoy día para comprobar que un sistema es correcto. Así, en la primera sección se da una introducción a la verificación automatizada, dedicando las tres siguientes secciones a describir los aspectos fundamentales de cada una de las tres técnicas más utilizadas para tal fin: la *demostración de teoremas*, la *comprobación de modelos* y la *realización de pruebas*, resaltando las fortalezas y debilidades de cada técnica. Dado que nuestro interés se centra en la comprobación de modelos, la sección dedicada a la misma se tratará con un mayor detenimiento y extensión. Finalmente, en la última sección se comparan brevemente las principales ventajas e inconvenientes de las técnicas mencionadas, llegándose a la conclusión de que la combinación de varias de ellas podría ser beneficiosa y presentándose una serie de enfoques que ponen en práctica esta idea.

3.1. Introducción a la verificación automatizada

Aunque la verificación se aplica tanto a sistemas hardware como software, nuestro interés se centra en éstos últimos, por lo que nos ceñiremos a ellos. Se puede decir que *verificar un programa* es demostrar, de manera matemático-formal, que el programa satisface una especificación escrita en un lenguaje lógico. Aunque desde un principio, tomando como punto de partida los trabajos de Floyd [Floyd67] y Hoare [Hoare69], la verificación parecía ser un enfoque prometedor para asegurar la corrección del software, después de más de 30 años de investigación, aún no se ha conseguido encontrar una tecnología ampliamente utilizable. Hay al menos dos razones para ello:

1) La verificación consiste en comparar dos descripciones de lo que el programa debe hacer: el propio programa y una especificación lógica, que en principio es más corta y abstracta. Para un sistema complejo, *la especificación lógica puede ser muy larga y difícil de obtener, no siendo obvia su corrección*, lo que convierte en hipotética la certeza que la verificación parece ofrecer.

2) El proceso de verificación requiere *demostraciones muy largas* (se puede necesitar mucho más tiempo que el empleado para escribir el propio programa) *y muy propensas al error*, ya que son demasiado específicas y aburridas para ser comprobadas de forma fidedigna por el hombre, y la tecnología de verificación aún no está lo suficientemente madura como para ser ampliamente utilizada.

No cabe duda de que el objetivo final, que está en la mente de muchos investigadores en este campo, es la *verificación completamente automática* (o sea, algorítmica) *del software*. Así, una herramienta o algoritmo de verificación de este tipo aceptaría como entradas tanto el sistema software a verificar como la especificación que éste debe cumplir y, sin intervención alguna del usuario, respondería si ese software satisface o no dicha especificación. Aunque ciertamente éste es un objetivo muy atractivo y deseable, gracias a la teoría de la computación [Sipser96] [Lewis97] [Hopcro79] [Papadi94] se sabe que esa herramienta no se puede construir para un amplia clase de programas. Sin embargo, esta restricción teórica no ha impedido la búsqueda, con los consiguientes hallazgos, de soluciones prácticas para comprobar la corrección del software [Peled01a]. Algunas de las ideas empleadas para conseguirlo son:

- Utilizar la abstracción para ocultar o eliminar muchos de los detalles (irrelevantes para nuestros propósitos) del software a verificar. De este modo, se obtiene un *modelo abstracto* (más simple) de ese programa, de modo que lo que se verifica realmente no es el propio programa (código fuente), sino su modelo abstracto (sistema de transición de estados, por ejemplo). Como consecuencia, en estos casos, el proceso de verificación logra un aumento, aunque no absoluto, en la confianza de que el sistema verificado es correcto. La abstracción mencionada suele realizarse manualmente, puesto que la existencia de un algoritmo de abstracción que pudiera *siempre* convertir en decidible el problema de la verificación de cualquier programa, entraría en contradicción con el resultado teórico de indecibilidad que se acaba de dar en el párrafo anterior.

- Restringir la verificación a una clase de programas más reducida, para la que dicho problema sea decidible, es decir, que exista un algoritmo de verificación automática que se pueda aplicar a cualquier programa de esa clase. Así, por ejemplo, para la comprobación de modelos (método explicado en la sección 3.3) esa clase será, por regla general, la de los sistemas de estados finitos.

- Centrarse en las partes cruciales del sistema, en lugar de intentar verificar todo el sistema. Así, por ejemplo, se puede verificar: el algoritmo básico subyacente de un producto software, o el protocolo de comunicaciones existente en un sistema concurrente, o una versión restringida de un programa, en la que se ponen límites (normalmente pequeños) a los valores que las variables pueden tomar, a los tamaños de las colas de mensajes, etc. De este modo, se simplifica bastante la parte que se debe verificar, al mismo tiempo que nos aseguramos de que lo esencial de ese software es correcto.

- Combinar varios métodos de verificación, pudiendo ser unos automáticos y otros manuales o semiautomáticos. Así, por ejemplo, para un determinado sistema software se puede aplicar manualmente el razonamiento deductivo para demostrar que la abstracción empleada preserva (en el modelo simplificado) las propiedades del sistema original que son de nuestro interés. A continuación, se puede comprobar automáticamente la corrección del modelo abstracto. Por tanto, al combinar ambos métodos, se consigue reducir el problema de la verificación a uno más simple y decidible.

3.2. Demostración de teoremas

La lógica suele considerarse como un marco formal para el razonamiento deductivo, en el que a partir de fórmulas que describen el dominio de interés (una axiomatización de dicho dominio) uno trata de obtener la fórmula que desea demostrar, con la ayuda de reglas de deducción perfectamente definidas. Para la verificación de programas, esto significa que se necesita traducir el programa a fórmulas, axiomatizar los objetos (enteros, cadenas de caracteres, ...) que el programa manipula y finalmente desarrollar la demostración.

La demostración de teoremas, también denominada *verificación deductiva*, es la primera técnica que surgió para verificar formalmente programas y algoritmos, ya que los trabajos de Floyd [Floyd67] y Hoare [Hoare69] preconizaron esta idea a finales de los años sesenta.

El proceso de verificación llevado a cabo con esta técnica se realiza fundamentalmente a mano, por lo que a menudo es lento, largo, tedioso y bastante propenso a errores. Por ello, y a pesar de que no se puede realizar de forma completamente automática, se han desarrollado una serie de potentes herramientas, denominadas *demostradores de teoremas*, con el fin de que sirvan de ayuda para realizar demostraciones de corrección y para imponer el rigor necesario a la hora de aplicar la verificación deductiva. Algunas de las herramientas más importantes de este tipo son: ACL2[1] [Kaufma00a] [Kaufma00b], COQ[2], HOL[3] [Gordon93], ISABELLE[4] [Nipkow02], PVS[5] [Owre92], STEP[6] [Manna94] [Bjorne00] y TLV[7].

Aunque estas herramientas no son capaces de completar automáticamente una demostración por sí solas, sí pueden (basándose en ciertas heurísticas) sugerir al usuario cómo continuar una demostración a partir de un determinado punto. Además, proporcionan gran ayuda durante el proceso de verificación, ya que, por ejemplo, pueden: mantener varias demostraciones alternativas incompletas, alternar entre subobjetivos que aún no se han demostrado y guardar las

[1] http://www.cs.utexas.edu/users/moore/acl2/
[2] http://coq.inria.fr/
[3] http://www.cl.cam.ac.uk/Research/HVG/HOL/
[4] http://www.cl.cam.ac.uk/Research/HVG/Isabelle/
[5] http://pvs.csl.sri.com
[6] http://www-step.stanford.edu/
[7] http://www.wisdom.weizmann.ac.il/~verify/tlv/index.shtml

demostraciones terminadas (para que puedan ser utilizadas en otras demostraciones o ser actualizadas más adelante). Una vez que una demostración se ha completado, muchos demostradores de teoremas proporcionan la opción de producir una copia (o impresión) de la misma en un formato que es "fácil" de leer y que, por ejemplo, permite su inclusión en un libro o artículo.

3.2.1. Fortalezas de la demostración de teoremas

Como puntos más fuertes e importantes de la verificación deductiva podemos citar los siguientes:

1) *Utilización de teorías matemáticas y lógicas con mayor rigor*, lo que a veces conlleva algunas consecuencias gratificantes, como por ejemplo: encontrar errores en un programa, permitir una mejor comprensión del código que se ha verificado y generalizar el algoritmo verificado para capturar nuevos casos que no fueron previstos.

2) *No está limitada a sistemas de estados finitos*, a diferencia de otras técnicas (entre ellas, la mayoría de las que se aplican en comprobación de modelos). Permiten la verificación de programas con varios dominios (enteros, reales, etc.) y estructuras de datos (pilas, colas, árboles, etc.) e incluso de programas parametrizados (por ejemplo, programas con un número indeterminado de procesos idénticos).

3) *Proporciona una mayor confianza en el programa verificado*, aunque en muchas ocasiones no se verifique el propio código, sino una versión abstracta más simple del algoritmo implementado. En algunos sistemas esenciales y críticos, donde un mal funcionamiento puede causar gran daño, estaría justificada la verificación completa (del código) de ese sistema, incluso cuando esto implique considerables esfuerzos y recursos.

4) *Incremento de la probabilidad de encontrar errores*, debido a que con frecuencia este tipo de verificación es realizado por personas distintas a las que diseñaron y desarrollaron el código. Así, hay más personas que intentan entenderlo y corregirlo.

5) *Introducción de la noción de invariante*, que es un predicado o afirmación que relaciona valores de distintas variables (incluyendo al contador de programa) y que se tiene que cumplir a lo largo de toda la ejecución del código. Este

concepto, surgido gracias a la verificación deductiva, puede incrementar la fiabilidad del software. Así, insertando estos invariantes (como predicados adicionales a comprobar) dentro del código fuente, éstos pueden utilizarse para realizar una *verificación en tiempo de ejecución*, de modo que si se viola alguno de esos invariantes, el programa es abortado para informar inmediatamente del problema detectado.

6) *Los sistemas de demostración pueden utilizarse para definir la semántica formal de las construcciones de los lenguajes de programación*. Así, por ejemplo, el sistema de demostración de Hoare [Hoare69] puede considerarse como la definición semántica formal de lenguajes de programación parecidos a *Pascal*, de modo que tanto sus reglas como axiomas ayudan a entender mejor los distintos tipos de sentencias. Este sistema de demostración ha sido extendido para poder verificar programas concurrentes [Apt97] [France92] [Schnei97], de manera que los sistemas de demostración resultantes permiten además el tratamiento de variables compartidas, comunicaciones síncronas y asíncronas y llamadas a procedimientos. Otra de sus características es que se suelen hacer a medida de un lenguaje de programación, tal como *Pascal concurrente* o CSP. Un sistema de demostración genérico y que no está ligado a ninguna sintaxis concreta puede encontrarse en [Manna83]. Utilizar uno de estos sistemas de demostración con el fin aquí indicado puede ayudar a desarrollar lenguajes más claros y comprensibles, ya que si una construcción (tipo de sentencia) del mismo es difícil de combinar dentro del sistema de demostración, esto puede indicar que no está bien definida o que no se ha entendido bien su significado.

3.2.2. Debilidades de la demostración de teoremas

Como principales desventajas o inconvenientes de la verificación deductiva se pueden mencionar las siguientes:

1) *Consume mucho tiempo y esfuerzo*, siendo con diferencia la técnica de verificación más lenta. De hecho, se suele invertir considerablemente más tiempo en la verificación deductiva que en la programación del correspondiente código, de modo que, en un proyecto grande, este tipo de verificación se puede convertir en el cuello de botella del mismo.

2) *Depende en gran medida del ingenio y habilidad de la persona que realiza la verificación*, requiriéndose que ésta tenga bastante pericia y un extenso bagaje

matemático-lógico; por esta razón, suele ser un matemático, un lógico o un ingeniero (informático) bien formado en este campo.

3) *Proceso manual en su mayor parte*, ya que la persona encargada de efectuar la verificación deductiva, posiblemente ayudada por el diseñador y/o el programador, debe establecer los correspondientes *invariantes* (que se deben cumplir a lo largo de toda la ejecución) y *afirmaciones intermedias* (que se deben cumplir en determinados puntos de la misma). Aunque se han desarrollado algunas heurísticas para este fin, por regla general no es factible obtener esas expresiones automáticamente. Además, aunque se utilice un demostrador de teoremas, hay partes de una demostración que no se pueden automatizar.

4) *Las demostraciones pueden contener errores*. Después de invertir una cantidad considerable de tiempo en una demostración, una persona tenderá a utilizar "atajos", por ejemplo, dejando sin demostrar aquellos fragmentos de código que parezcan demasiado obvios y triviales. Esta práctica, que puede ser fuente de errores, no está permitida cuando se utiliza un demostrador de teoremas. En este sentido, estas herramientas imponen rigor al proceso de verificación, asegurando que cada paso intermedio se obtiene conforme a las reglas y axiomas del sistema de demostración utilizado. Aún así, debe tenerse en cuenta que pueden existir errores potenciales en la propia herramienta de verificación (al tratarse ésta de un programa más de software).

5) *Riesgo de introducir demasiadas suposiciones, que pueden restringir la generalidad de una demostración a determinados casos particulares*. En efecto, se ha observado que muchas personas tienden a añadir como axiomas lo que no son más que suposiciones adicionales (que se pueden dar o no), creyendo que son obvias y que acortan la demostración. Así, se obtienen demostraciones más débiles de lo que se piensa, ya que en lugar de demostrar que cierta propiedad se cumple sobre un determinado dominio, en realidad lo que se ha demostrado es que dicha propiedad se cumple bajo las suposiciones añadidas durante la demostración. Por lo tanto, se puede creer erróneamente que esa propiedad es correcta para todo el dominio considerado, cuando lo cierto es que no lo es. En el peor de los casos, si existe alguna contradicción interna en las suposiciones introducidas, la demostración no tiene ningún valor, pero la persona que la ha realizado no tiene constancia de ello. Con el fin de evitar este tipo de errores, algunos demostradores de teoremas restringen el uso de estas suposiciones,

bien obligando al usuario a demostrar cada nueva suposición (utilizando los teoremas ya demostrados con la herramienta) o bien marcando como *inseguras* las demostraciones que utilicen suposiciones adicionales (no demostradas).

6) *Aunque la demostración sea correcta, el código puede contener errores*. Debido a la naturaleza detallada de este tipo de verificación y a la dificultad de realizarla sobre el código completo, con frecuencia se verifica no el propio código, sino un modelo simplificado del mismo. Por tanto, pueden existir diferencias entre la versión abstracta verificada y el código realmente implementado, que hacen posible que se cumpla lo enunciado en la primera frase de este párrafo. Para evitar esto se tendría también que verificar formalmente la fidelidad de la abstracción realizada.

3.3. Comprobación de modelos

Los orígenes de esta técnica se encuentran en los trabajos de Clarke y Emerson [Clarke81] [Emerso80] y de Quielle y Sifakis [Quiell82], realizados ambos de forma independiente. La *comprobación de modelos* (*model checking*), tratada ampliamente en [Clarke99] y [Bérard01] y de la que se da una excelente introducción en [Wolper95], es considerada desde sus comienzos como una técnica de verificación *automática* de programas [Clarke86]. Este método se basa en un problema (de lógica formal) bastante simple: comprobar que una determinada propiedad (expresada mediante una fórmula de una lógica temporal) se satisface en un dominio finito específico (el sistema de transición que representa al sistema a verificar). En otras palabras, consiste en comprobar que cualquier ejecución de ese sistema de transición es un *modelo*[8] de la fórmula, de ahí el nombre asignado a esta técnica[9]. Por consiguiente, para comprobar que un programa satisface una determinada propiedad, sólo hay que evaluar la correspondiente fórmula sobre el sistema de transición cuyos estados representan los posibles estados del programa

[8] Aquí la palabra *modelo* se emplea en el sentido de "secuencia de estados que satisface una determinada especificación". En la bibliografía sobre comprobación de modelos, esta palabra se utiliza a veces para designar al sistema de transición que representa el comportamiento del sistema que se desea verificar, y se denomina así debido a que se trata de una versión simplificada del sistema original, obtenida a partir del mismo mediante un proceso de abstracción.

[9] Obsérvese que, independientemente del significado asociado a la palabra *modelo* (ver nota anterior), el nombre de la técnica, *comprobación de modelos*, es suficientemente descriptivo y siempre hace referencia a lo mismo.

y cuyas transiciones de un estado a otro se corresponden con la ejecución de una instrucción o paso del programa.

La comprobación de modelos no pretende ser un método completamente general, sino que, en principio, *sólo es aplicable a sistemas cuyos estados tienen descripciones cortas y fácilmente manipulables*. Sistemas típicos de esta clase son aquellos cuya complejidad reside más en el control que en los datos, tales como por ejemplo: el hardware, protocolos concurrentes, sistemas de control de procesos y más generalmente los denominados como *sistemas reactivos* [Pnueli86a], cuyo papel se describe más bien en función de sus posibles secuencias de interacción con el entorno que por la transformación que aplican a datos complejos.

Los sistemas de transición utilizados en esta técnica de verificación son modelos globales del sistema que se desea analizar, por lo que carecen de entradas y son *no deterministas*. El no determinismo en el sistema de transición proviene o del modelado de la concurrencia por entremezclamiento o de la ausencia de información acerca del comportamiento de algún componente del sistema o de su entorno. Así, por ejemplo, un valor arbitrario producido por el entorno se puede modelar mediante una elección no determinista de dicho valor.

3.3.1. Enfoque lineal frente a enfoque ramificado

Puesto que los sistemas de transición que modelan los programas son no deterministas, una cuestión que se debe abordar es cómo manejar (dentro de la lógica para especificar las propiedades) la multiplicidad de sucesores que cada estado puede tener. Esto se puede hacer explícitamente (enfoque ramificado) o implícitamente (enfoque lineal):

- En el *enfoque ramificado* [Clarke81] [Clarke86] los operadores temporales introducidos en la lógica especifican si se consideran *todos* los sucesores posibles o sólo *algunos* de ellos. Por ejemplo, se puede decir que “en algún estado futuro” una propiedad será cierta, o que “en todos los estados alcanzables” una condición se cumplirá.

- En el *enfoque lineal* [Lichte85a] [Vardi86] la lógica es diseñada para describir secuencias lineales de estados (cada estado tiene sólo un sucesor). El problema de la comprobación de modelos consiste entonces en comprobar que todas las secuencias lineales que puede generar el sistema

de transición (todas las posibles ejecuciones del programa) satisfacen la fórmula de la lógica temporal lineal utilizada.

El enfoque lineal es adecuado cuando las propiedades que se desea comprobar se expresan en términos de ejecuciones del programa, mientras que el enfoque ramificado es más apropiado cuando las propiedades se expresan en términos de la estructura del programa. Para más detalles, consultar [Wolper87] y [Emerso90].

3.3.2. El problema de la explosión de estados

El espacio de estados de un sistema concurrente compuesto por varios procesos puede llegar a tener tantos estados como el número resultante de multiplicar los estados de cada uno de los procesos que lo integran. Así, por ejemplo, si un sistema está compuesto por n procesos idénticos, cada uno de ellos con m estados, donde los procesos no se comunican entre sí, entonces tendremos m^n estados (n elecciones independientes de m posibilidades). Obviamente, en casos más realistas, la interacción entre los distintos procesos a través de, por ejemplo, variables compartidas, puede limitar considerablemente el número de posibles estados. A pesar de ello, el número de estados sigue siendo por regla general bastante elevado en sistemas medianamente complejos, siendo éste el principal factor restrictivo a la hora de aplicar la comprobación de modelos, ya que el sistema de transición que representa al programa a menudo tiene demasiados estados para ser construido o incluso puede ser infinito. A esto es a lo que se conoce con el nombre de *problema de la explosión de estados*.

Para intentar paliar este problema han surgido varias estrategias, aunque ninguna de ellas garantiza una solución definitiva para combatirlo. Ha de ser la experiencia y la realización de experimentos lo que determine qué estrategia o combinación de ellas utilizar en cada caso. Dentro de estas estrategias podemos distinguir, a grandes rasgos, entre dos tipos de técnicas:

I) Un primer grupo es el formado por aquellas *técnicas que intentan construir sólo parte del espacio de estados*, manteniendo la capacidad de comprobar las propiedades de interés. Entre ellas se encuentran las siguientes:

   a) *Técnicas de reducción de orden parcial* [Godefr90] [Valmar90] [Peled93], que intentan evitar la representación antieconómica (en cuanto al número de estados) de la concurrencia por entremezclamiento. Para ello,

explotan la conmutatividad existente entre transiciones concurrentes cuando, desde el punto de vista de la propiedad a comprobar, diferentes reordenaciones de eventos concurrentes generan la "misma" ejecución, o sea, esa especificación no distingue entre dichas ejecuciones.

b) *Técnicas de abstracción* [Clarke92] [Graf93], que sustituyen el sistema que se desea comprobar por uno más simple, en el que se han suprimido los detalles irrelevantes para la propiedad que se va a comprobar.

c) *Enfoques "on-the-fly"* [Jard91] [Courco92] [Couvre99], que construyen sólo la parte del espacio de estados del sistema que es explorada por la fórmula particular que está siendo comprobada. En otras palabras, el espacio de estados se confecciona a medida de la propiedad a comprobar, y guiada por ésta.

d) *Simetría* [Emerso93], que aprovecha el hecho de que ciertas permutaciones de los componentes de un estado generan la "misma" ejecución para una determinada propiedad. Recuérdese que cada estado del sistema (global) está formado por una serie de componentes (uno por cada proceso). Así, esta técnica se basa en la observación de que algunas especificaciones (especialmente para sistemas hardware) no pueden distinguir entre ciertas ejecuciones obtenidas al permutar componentes que pertenecen a diferentes procesos. En estos casos, se utiliza un representante para cada conjunto de estados que son equivalentes bajo permutación, evitándose así tener que explorar todos los posibles estados del sistema.

II) Otra dirección distinta la constituye el *enfoque simbólico* para comprobación de modelos (*symbolic model checking*) [Burch92] [McMill93], donde la idea es representar implícitamente en lugar de explícitamente los estados y transiciones que modelan el programa. La representación implícita habitual es una codificación eficiente de las fórmulas booleanas, conocida con el nombre de *Binary Decision Diagrams* (BDDs) [Bryant92]. Lo verdaderamente importante de esta representación es que las fórmulas temporales se pueden comprobar directamente sobre ella, sin tener que construir una representación explícita del espacio de estados. Esto cambia completamente el factor restrictivo: la cuestión ya no es el tamaño del espacio de estados, sino el tamaño de su representación BDD. Como consecuencia, a veces se pueden comprobar sistemas con un número astronómico de estados. Este enfoque se basa en la

utilización de la lógica temporal ramificada CTL (*Computation Tree Logic*) [Ben-Ar83], siendo bastante adecuado para la verificación de circuitos hardware [Meinel98].

3.3.3. Fortalezas de la comprobación de modelos

Entre las principales fortalezas de la comprobación de modelos se encuentran las siguientes:

1) *Proceso de verificación en gran parte automático*, aunque no completamente, ya que normalmente se requiere que el sistema a verificar sea modelado, esto es, traducido a una representación manipulable por la herramienta utilizada. Este proceso debe realizarlo el usuario, quien, por regla general, crea un modelo abstracto más simple que el sistema original, con el fin de reducir la complejidad del problema de la verificación. A pesar de esta intervención del usuario en el proceso de verificación, comparándola con la que exigen otras técnicas o métodos de verificación, su participación es más bien pequeña.

2) *Relativamente fácil de implementar*, debido a que la comprobación de modelos se basa en una serie de ideas simples para las que no es muy difícil construir un buen algoritmo. Prueba de ello es la proliferación de herramientas de comprobación de modelos existente en la actualidad, que permiten comprobar un amplio rango de propiedades. Entre estas herramientas se encuentran: SPIN[10] [Holzma91] [Holzma97], SMV[11] [Clarke96] [McMill93], VIS[12] [VIS96], COSPAN/FORMALCHECK [Har'El90], MURPHI[13] [Dill92] y PEP[14] [Grahlm97] [Grahlm99], por citar algunas de las más representativas.

3) *Genera un contraejemplo* cuando el sistema modelado no satisface la propiedad especificada. Este contraejemplo es una ejecución que incumple dicha propiedad. La información suministrada por el contraejemplo puede ser utilizada tanto para corregir el modelo abstracto del sistema como para depurar el código del programa original. Puesto que el principal objetivo de una herramienta de comprobación de modelos es la rápida detección de posibles errores, con frecuencia se considera mucho más valioso que ésta devuelva un

[10] http://netlib.bell-labs.com/netlib/spin/whatispin.html
[11] http://www-2.cs.cmu.edu/~modelcheck/smv.html
[12] http://www-cad.eecs.berkeley.edu/Respep/Research/vis/
[13] http://sprout.stanford.edu/dill/murphi.html
[14] http://theoretica.informatik.uni-oldenburg.de/~pep/

contraejemplo a que concluya indicando que el modelo del programa satisface la especificación.

4) *Se puede utilizar fácil y repetidamente* para comprobar una amplia variedad de propiedades, proporcionando información útil, incluso sin tener una especificación exhaustiva del sistema que se está desarrollando. Por consiguiente, la comprobación de modelos puede ser vista como una *herramienta de depuración minuciosa* que, en lugar de observar unas pocas de las ejecuciones posibles, comprueba de forma fidedigna que todas las posibles ejecuciones del programa satisfacen una determinada propiedad.

5) *Permite llevar a la práctica la verificación automática de software*, al menos para alguna versión abstracta del producto software. Recordemos que la complejidad de la verificación automática de software es un problema computacionalmente duro que, hasta hace no mucho tiempo, fue considerado como muy difícil de llevar a cabo. Esto se ha superado hoy en día, en parte gracias al vertiginoso avance del hardware, con memorias cada vez de mayor capacidad y procesadores cada vez más rápidos, y en parte gracias a la gran cantidad de heurísticas especialmente desarrolladas para la comprobación de modelos.

6) *Herramientas optimizadas para un tipo particular de software*. Claramente, el tipo de optimización dependerá de las heurísticas especiales que implemente la herramienta de verificación. Así, por ejemplo, una herramienta de comprobación de modelos puede estar optimizada para verificar programas (concurrentes) cuyos procesos se comunican de forma asíncrona mediante paso de mensajes, mientras que la existencia de variables compartidas entre dichos procesos podría neutralizar la mayoría de sus optimizaciones. Por consiguiente, habría que escoger la herramienta más adecuada al tipo de sistema a verificar.

3.3.4. Debilidades de la comprobación de modelos

Además del problema de la explosión de estados, tratado en el apartado 3.3.2, y que constituye su principal factor restrictivo, como debilidades de la comprobación de modelos podemos citar las siguientes:

1) *Requiere modelar* (*manualmente*, por regla general) *el sistema a verificar* en el lenguaje de modelado que incluye la herramienta utilizada. Aunque en ocasiones se dispone de un mecanismo de traducción automática entre el código

fuente original y la sintaxis empleada por el lenguaje de modelado interno de la herramienta, al utilizarlo se suelen perder ciertas optimizaciones y heurísticas que las herramientas de comprobación de modelos incluyen para su sintaxis interna. Lo mismo puede decirse de las traducciones automáticas realizadas entre formalismos utilizados por diferentes herramientas: su resultado es mucho menos eficiente que el obtenido al usar directamente la sintaxis específica de la herramienta con la que se va a realizar la verificación. Además, modelar el sistema de modo manual permite aplicar cierta abstracción al modelo, con el fin de mejorar la ejecución del proceso de verificación propiamente dicho.

2) *Se verifica un modelo abstracto del programa, no el propio programa.* Como consecuencia, se ha de evaluar cuidadosamente el resultado del proceso de verificación, debido a la posibilidad de modelar incorrectamente el sistema (por ejemplo, aplicando una abstracción inadecuada). Así, puede que el modelo sea una *infraespecificación* del programa (esto es, que no considere algunas de sus ejecuciones); en ese caso, puede ocurrir que la herramienta de comprobación de modelos informe de que el modelo satisface la especificación, cuando el programa real no lo hace. Por el contrario, el modelo puede ser una *sobreespecificación* del programa (es decir, que contenga ejecuciones que no se correspondan con las del programa); por esta razón, cuando la herramienta de comprobación de modelos devuelve un contraejemplo, el usuario debe comprobar, de nuevo manualmente, que se trata de un error real en el programa y no sólo en el modelo abstracto.

3) *El usuario debe*, a menudo, *afinar la verificación*, suministrando ciertos parámetros, tales como: el tamaño de la pila de búsqueda, la cantidad de memoria disponible y la ordenación de las variables del modelo. Esto conlleva que el usuario de una herramienta de comprobación de modelos sea, en muchos casos, un experto en su manejo, y en ocasiones, su propio desarrollador.

4) *Las herramientas existentes a menudo no terminan el proceso de verificación*, debido a restricciones de memoria o de tiempo, siendo incluso difícil predecir si una herramienta en particular tendrá éxito o fracasará en determinar si cierta propiedad se cumple o no en un sistema concreto. Una de las razones que explican esto es que el tamaño del espacio de estados requerido para verificar un programa de una determinada longitud puede variar considerablemente.

3.4. Realización de pruebas

La *realización de pruebas* (*testing*) [Myers79] [Beizer90] [Kaner93] [Patton00] consiste en ejecutar el programa a comprobar para un muestreo de sus posibles ejecuciones (seleccionadas de acuerdo a algún criterio), en un intento de encontrar errores. Cada ejecución se compara con el resultado esperado y cualquier desviación se anota como error.

Dado que se basa en la ejecución de sólo algunas de todas las posibles ejecuciones del programa a comprobar (la ejecución de todas ellas es normalmente inviable y poco práctico), esta técnica no pretende demostrar la ausencia de errores en el programa ni que éste cumple su objetivo correctamente, ni puede garantizar que se encuentren todos los errores (ni siquiera alguno). Por todo ello, algunos investigadores no consideran que esta técnica sea un método formal. Sin embargo, es con diferencia la técnica más utilizada para mejorar la calidad del software.

Puede ser necesario añadir código al programa que se pretende comprobar para prepararlo convenientemente antes de realizar el proceso de pruebas. El objetivo de este código adicional puede ser, por ejemplo: automatizar dicho proceso, seguir la pista de los resultados de las pruebas y comprobar su corrección y/o elaborar estadísticas acerca de todo el proceso de pruebas. En otros casos, puede ser necesario obligar al programa a empezar en un estado que no sea inicial (posiblemente asignando determinados valores a sus variables) o comparar los valores de algunas variables internas cuando se alcanza cierto punto de la ejecución. Para estas tareas también habrá que añadir código al programa original. En algunas ocasiones, este código puede ser añadido automáticamente en tiempo de compilación.

Existen distintos tipos de pruebas, que se aplican en diferentes niveles y fases. Mencionaremos a continuación algunos de los más importantes, entre ellos, por ejemplo: las *pruebas de unidad* (o *de módulo*), que son las de más bajo nivel, dado que se aplican a pequeños fragmentos de código separadamente; las *pruebas de integración*, que comprueban si los diferentes módulos (que pueden haber sido programados por distintos equipos) se acoplan bien; las *pruebas de sistema*, que comprueban la funcionalidad del software como un todo, y las *pruebas de aceptación*, que normalmente son realizadas por el cliente con el fin de determinar si el programa se ajusta a sus necesidades. Las pruebas de nivel inferior (de unidad o de integración) señalan exactamente dónde se encuentra el correspondiente error,

mientras que las de nivel superior (de sistema o de aceptación) sólo dan una idea aproximada de cuál puede ser la causa del mismo. Así, las primeras permiten encontrar rápidamente un error (siendo más fácil y económico corregirlo), además hacen posible que diferentes partes del software se puedan desarrollar y probar simultáneamente.

Las *pruebas de caja blanca* (*white box testing*) [Cole00] se basan en inspeccionar los detalles internos del código. Para enfatizar más el hecho de que el código es visible, algunos prefieren designarlas con otros nombres, como: *pruebas de caja transparente*, *clara*, *de cristal* o *abierta*. Se denomina *camino de ejecución* a una secuencia de puntos de control e instrucciones que aparecen en el código a comprobar, o sea, a un camino en el diagrama de flujo de dicho código. El número de caminos de ejecución en un programa puede ser muy grande e incluso no estar acotado, por lo que normalmente es inviable recorrerlos todos. Para hacer frente a esta limitación existen diferentes *criterios de cobertura*, que establecen cómo se deben escoger los distintos casos de prueba para que, comprobando un número relativamente pequeño de caminos de ejecución, se logre alcanzar un alto grado de probabilidad de encontrar errores potenciales. Algunos de esos criterios son:

- *Cobertura de sentencias*, donde cada sentencia ejecutable del programa debe aparecer en al menos un caso de prueba.
- *Cobertura de arcos*, donde cada arco ejecutable (del diagrama de flujo del programa) debe aparecer en algún caso de prueba.
- *Cobertura de condiciones*, donde cada condición ejecutable del programa debe aparecer como mínimo en dos casos de prueba, en uno de ellos evaluada como cierta y en el otro como falsa.

Con el *análisis de la cobertura del código* se puede medir la calidad del conjunto de casos de prueba considerado (denominado *test suite* en inglés). Para ello, se compara dicho conjunto contra el código, según el criterio de cobertura elegido. Así, por ejemplo, es posible obtener el porcentaje de sentencias o de arcos cubiertos, dando así una indicación de qué partes del código (sentencias, condiciones, etc.) necesitan ser cubiertas por nuevos casos de prueba. Este análisis, que puede servir para determinar la probabilidad de que haya errores en el código después del proceso de pruebas, algunas veces identifica casos de prueba redundantes para obtener la cobertura deseada.

Las *pruebas de caja negra* (*black box testing*) [Beizer95] no consideran la estructura interna del sistema a comprobar, bien porque no se conozca o no esté accesible o para no estar influenciado por la implementación del mismo. Por lo tanto, están más orientadas a comprobar la funcionalidad del sistema en etapas más avanzadas del desarrollo (pruebas de sistema y de aceptación). Por el contrario, las pruebas de caja blanca, al inspeccionar directamente el código, son más apropiadas para las primeras fases de pruebas (de unidad y de integración). Para las pruebas de caja negra a menudo se modela el sistema a comprobar mediante un grafo o un autómata y se utilizan algoritmos de grafos para generar el conjunto de casos de prueba que se ejecutarán sobre el código, mientras que en las pruebas de caja blanca se usa el propio código para obtener dicho conjunto. Parece lógico, pues, que los criterios de cobertura empleados en las pruebas de caja negra se apliquen con respecto al espacio de estados del grafo modelado, intentándose cubrir por regla general todos los arcos o todos los estados del mismo, aunque existen otros criterios.

Existen diferentes clases de herramientas que ayudan a realizar las pruebas. Entre ellas se encuentran las que sirven para: *generación de casos de prueba*, que modelan el sistema a comprobar y utilizan el modelo para generar el conjunto de pruebas según el criterio de cobertura elegido; *evaluación de la cobertura*, que indican el grado de cobertura logrado al utilizar un determinado conjunto de pruebas (*suite test*), y *ejecución del test*, que ejecutan los casos de prueba sobre el programa a probar e informan de los resultados del test, para lo que pueden requerir la compilación de dicho programa, con el fin de incluirle código adicional necesario para ejecutar las pruebas (como se ha descrito al principio de esta sección). Hay gran cantidad de herramientas de pruebas (*testing tool*), pero debido a que la mayoría de ellas son comerciales (no estando disponibles sin cargo alguno para propósitos académicos o de investigación), sólo indicamos aquí algunas de ellas: GCT[15] (*Generic Coverage Tool*), PET (*Path Exploration Tool*) [Gunter99] [Gunter00], TESTMASTER, TESTPARTNER, TESTWORKS/COVERAGE, CODETEST y TESTCENTER.

3.4.1. Fortalezas de la realización de pruebas

Como puntos más fuertes o principales ventajas de la realización de pruebas podemos mencionar las siguientes:

[15] ftp://cs.uiuc.edu/pub/testing/GCT/GCT.README

1) *Se aplica directamente sobre el propio sistema (código) a comprobar*, sin necesidad de generar un modelo más simple del mismo, como ocurre con frecuencia en las otras dos técnicas presentadas.

2) *Es una técnica simple*, que no requiere de formalismos matemático-lógicos, lo que facilita su utilización por un mayor número de personas. Por esta razón, es la técnica más popular para aumentar la calidad del software.

3) *Proporciona una solución práctica (a un coste razonable) para sistemas grandes*. La realización de pruebas consume mucho menos tiempo y requiere bastantes menos recursos y esfuerzos humanos que la verificación deductiva (demostración de teoremas). Además, es aplicable incluso en aquellos casos en los que es demasiado difícil y complicado realizar ésta última, y donde la verificación automática (comprobación de modelos) no es viable, como por ejemplo: en sistemas con un espacio de estados enorme o infinito, o en sistemas con estructura de datos complejas.

4) *Se adapta a la estrategia empleada para desarrollar el sistema que se pretende comprobar*. Así, en programas de gran tamaño, compuesto por muchos procedimientos, tareas y/o módulos, desarrollados normalmente por diferentes equipos, esta técnica permite que cada equipo pruebe por separado la parte del software que ha implementado, antes de integrarla al resto y de realizar las pruebas (globales) a nivel de sistema. Con ello se consigue acelerar el proceso de desarrollo (al poder trabajar los distintos equipos simultáneamente y en paralelo) y que la comprobación de errores pueda empezar incluso antes de que todos los equipos hayan completado su trabajo.

3.4.2. Debilidades de la realización de pruebas

Los inconvenientes o desventajas más importantes de esta técnica son los que se exponen a continuación:

1) *No es una técnica exhaustiva*, al basarse en un muestreo de las ejecuciones del sistema, en lugar de en la comprobación sistemática de todas ellas. Puesto que se aplica directamente a programas reales con un número muy grande (o incluso infinito) de estados, es imposible o poco práctico comprobar todas sus ejecuciones.

2) *Proporciona menos garantías de corrección que las otras dos técnicas*. En efecto, no puede garantizar que el sistema comprobado se comportará correctamente bajo cualquier circunstancia posible. Tampoco garantiza la detección de todos los errores existentes en el código, es más, ni siquiera de algunos de ellos.

3) *Se debe añadir código al programa original*, con el fin de ejecutar convenientemente los distintos casos de prueba y de observar y comprobar adecuadamente los resultados obtenidos (tal y como se ha comentado al principio de la sección 3.4).

4) *Los casos de prueba tienen un cierto sesgo*, ya que están influenciados por el criterio de cobertura elegido y por las estructuras a partir de las que se generan (código implementado, en las pruebas de caja blanca, y grafo modelado, en las pruebas de caja negra).

5) *Realizada a menudo por el propio programador*. En cierto sentido, el objetivo de un programador es el opuesto al de la persona que realiza las pruebas, ya que ésta tratará de mostrar que el código contiene errores (al contrario que aquél). Por lo tanto, no se aconseja que un programador compruebe su propio código, ya que puede estar demasiado confiado en la bondad del mismo e influenciado por su estructura interna (que él mismo ha creado y que puede no ser la más adecuada). Sin embargo, en la práctica, esto es lo más habitual.

3.5. Combinación de métodos formales

Cada uno de los métodos presentados en este capítulo tienen sus ventajas e inconvenientes, siendo los que se enuncian a continuación, a modo de resumen, los más importantes:

- La verificación automática (comprobación de modelos), es muy atractiva, puesto que es una técnica exhaustiva (comprueba que todas las posibles ejecuciones del sistema son correctas) y sólo requiere una mínima intervención por parte del usuario. Sin embargo, su efectividad depende en gran medida del tamaño del sistema verificado, decreciendo conforme éste aumenta.

- La verificación deductiva (demostración de teoremas) puede, por el contrario, ser aplicada a sistemas con un número infinito de estados, pero es bastante

lenta y requiere una amplia participación y gran destreza por parte del usuario.

- La realización de pruebas (*testing*), no siendo un método formal propiamente dicho, tiene la ventaja de que se aplica directamente sobre el propio sistema software, pero no es una técnica exhaustiva, por lo que puede omitir la detección de algunos errores.

Por consiguiente, parece lógico que la cooperación o integración de varios de estos métodos para verificar un sistema pudiera conllevar una suma o combinación de sus fortalezas, al mismo tiempo que se podrían mitigar algunas de sus debilidades. Así, por ejemplo, al combinar la verificación (automática o deductiva) del *diseño* (modelo) de un sistema con la realización de pruebas al *sistema real* (código) se incrementa considerablemente la fiabilidad del software, decreciendo enormemente la posibilidad de que el sistema se comporte de forma errónea o inesperada. A continuación se exponen algunos enfoques existentes que integran (o al menos recogen ideas de) varias de las técnicas presentadas en este capítulo.

Mientras que en la comprobación de modelos se verifica automáticamente que un modelo del sistema cumple ciertas propiedades, en las pruebas de caja negra se usa dicho modelo para generar los casos de prueba que se deben ejecutar sobre el código. Esto hace que, en cierto sentido, ambas técnicas sean complementarias, pudiéndose combinar en un enfoque [Sharyg01] que construye un modelo del sistema y cuyo esquema se muestra en la Figura 3.1 . Ese modelo se usa para ejecutar los casos de prueba sobre el programa, comparando su comportamiento con el de las ejecuciones realizadas. Además, se verifica automáticamente (mediante comprobación de modelos) que éste cumple la propiedad especifica.

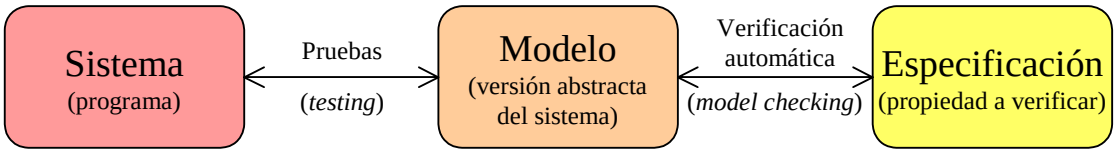


Figura 3.1. Enfoque que combina la realización de pruebas con la comprobación de modelos

Una pega que se le puede poner al enfoque anterior es que se pretende comprobar que el modelo se ajusta al sistema mediante la realización de pruebas,

lo que no constituye una gran garantía. Para incrementar nuestra confianza en este sentido, se puede emplear un enfoque más riguroso (y caro), consistente en sustituir la realización de pruebas por la verificación (deductiva) de la correspondencia existente entre el sistema y su modelo (ver Figura 3.2). Puesto que el modelo es una versión abstracta (simplificada) del sistema real, verificar la corrección de éste último implica en realidad dos tareas de verificación:

(1) Demostrar (deductivamente) que las propiedades esenciales (que incluyen a todas las que se quieren comprobar) se preservan entre el sistema original y su versión simplificada.

(2) Verificar (automáticamente) la corrección del modelo abstracto.

Figura 3.2. Enfoque que combina la verificación deductiva con la comprobación de modelos

La verificación deductiva consiste en demostrar las relaciones de *simulación* existentes (en ambos sentidos) entre la versión original y abstracta del sistema. La simulación hacia delante implica que cualquier ejecución del sistema está recogida dentro del lenguaje aceptado por el modelo ($\mathscr{L}(S) \subseteq \mathscr{L}(M)$), por lo que si éste satisface la especificación ($\mathscr{L}(M) \subseteq \mathscr{L}(\varphi)$), significará que el sistema original también la satisface ($\mathscr{L}(S) \subseteq \mathscr{L}(\varphi)$). La simulación hacia atrás asegura que cada contraejemplo que muestre que el modelo M no cumple la especificación φ, también será un contraejemplo para el sistema original S. Aunque para asegurar la corrección del sistema se necesita demostrar ambas relaciones de simulación, este enfoque también se puede aplicar a sistemas en los que sólo una de ellas puede ser establecida. En este caso, debe tenerse cuidado con las conclusiones que se saquen a partir de los resultados de la comprobación de modelos. Así, si sólo se ha definido la simulación hacia delante, entonces se necesita comprobar que cada contraejemplo encontrado también lo es del sistema original, mientras que si sólo se ha establecido la simulación hacia atrás, entonces que el modelo M satisfaga la especificación φ no implica que el sistema S también lo haga.

En muchas ocasiones, la persona que efectúa la verificación realiza el proceso de abstracción (necesario para obtener el modelo) de una manera informal, basándose en su propia intuición y experiencia. En estos casos, no es seguro sacar conclusiones a partir de los resultados de la verificación automática del modelo, a no ser que se hayan demostrado las dos relaciones de simulación mencionadas anteriormente. Con el fin de evitar esto último, se han sugerido algunas heurísticas para obtener automáticamente abstracciones correctas [Bensal98] [Namjos00] [Graf97].

Otro enfoque que también combina la comprobación de modelos con la verificación deductiva es el que se presenta en [Peled01b], donde automáticamente se genera una demostración cuando la comprobación de modelos no encuentra ningún contraejemplo.

La *comprobación de caja negra* (*black box checking*) [Peled99] es una técnica que combina la exhaustividad de la comprobación de modelos con la capacidad de comprobar directamente el sistema real, propia de las pruebas de caja negra. Por ello, puede considerarse como una mezcla de ambas técnicas, como su nombre indica, aunque también se le denomina *comprobación directa*, con el fin de establecer una clara diferencia con los distintos métodos de pruebas de caja negra [Lee96] con los que a menudo se le relaciona. Esta técnica se suele usar en las pruebas de aceptación para comprobar automáticamente que el sistema (cuya estructura se desconoce) satisface las propiedades especificadas. La idea consiste en empezar a verificar el sistema sin proporcionar inicialmente ningún modelo para el mismo e ir construyendo éste a medida que la verificación avanza. La Figura 3.3 muestra la estrategia empleada, donde en cada momento se verifica (mediante comprobación de modelos) el modelo actual. Si se encuentra un contraejemplo, se comprueba que también lo sea en el sistema real, en cuyo caso se concluye que el sistema no satisface la propiedad (mostrándose dicho contraejemplo como resultado), mientras que si el contraejemplo es una *falsa negativa* (o sea, lo es del modelo, pero no del sistema), entonces se refina automáticamente el modelo (utilizando un algoritmo de aprendizaje) para corregir la diferencia (entre sistema y modelo) encontrada por ese contraejemplo. Por el contrario, si la comprobación de modelos no encuentra ningún contraejemplo, entonces se aplican pruebas de caja negra para buscar alguna discrepancia entre el modelo actual y el sistema; si se encuentra alguna secuencia para la que el comportamiento de ambos difiera, de nuevo se actualiza el modelo con respecto a dicha secuencia, mientras que si se

determina que el modelo se ajusta al sistema, entonces se informa de que éste cumple la propiedad especificada. Se trata, pues, de una variante del problema de la comprobación de modelos que, desafortunadamente, sólo puede ser aplicada bajo ciertas suposiciones (bastante fuertes) y cuyos algoritmos son de una complejidad elevada, razones por las que tanto la comprobación de modelos como la realización de pruebas son más viables que esta técnica en la mayoría de los casos.

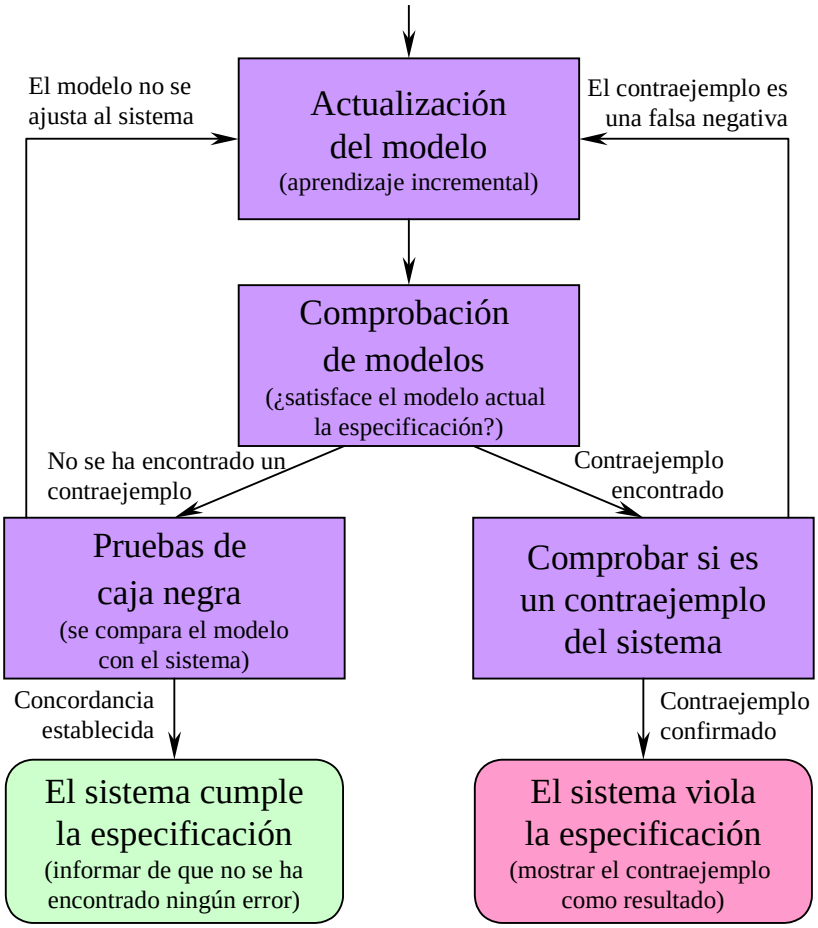


Figura 3.3. Estrategia de la comprobación de caja negra

Una variante del enfoque anterior es la *comprobación de modelos adaptativa* (*adaptive model checking*) [Groce02], donde inicialmente se proporciona un *modelo* del sistema, que puede ser *inexacto*, debido a errores de modelado o a nuevas actualizaciones realizadas en el sistema. Así, se intenta *adaptar* dicho modelo, aprendiendo de las diferencias existentes entre el mismo y el sistema, en lugar de construir el modelo desde cero, como se hace en la comprobación de caja negra; con ello se trata de reducir la alta complejidad de este último enfoque, disminuyendo particularmente el número de llamadas al algoritmo que ejecuta las pruebas de

caja negra (el de mayor complejidad del enfoque). Sin embargo, este método tampoco es la panacea, ya que la comprobación de modelos adaptativa sólo es útil cuando la modificación realizada en el sistema es simple o cuando dicha modificación influye muy poco a la hora de determinar si el sistema cumple o no la propiedad que se desea verificar.

Otro posible enfoque combinado consiste en utilizar la negación de la propiedad a comprobar (idea procedente de la comprobación de modelos) a la hora de generar los casos de prueba, con el fin de dirigir las pruebas de caja negra hacia la localización de errores (ejecuciones que no cumplan la propiedad especificada). Se trata de construir el producto o intersección del autómata modelo del sistema a comprobar (típico en las pruebas de caja negra) con el autómata que representa las ejecuciones no permitidas (negación de la especificación) y utilizar el autómata resultante para generar los casos de prueba a ejecutar sobre el sistema original.

El *método de sala limpia* (*cleanroom method*) [Mills75] [Nascim94] [Stavel99] incluye ideas de la verificación deductiva y de la realización de pruebas. Este método, que ha sido aplicado con éxito en varios proyectos, propone que la verificación de la corrección del software se efectúe mientras éste se desarrolla. Sin embargo, a diferencia de la verificación deductiva, la demostración se realiza más informalmente (sin aplicar en realidad axiomas ni reglas de demostración), de modo que para cada fragmento de programa se tiene que especificar una fórmula que exprese la relación existente entre las variables al principio y al final de la ejecución de dicho fragmento. El siguiente paso sería la *inspección de la demostración*, donde el programador debe convencer a un equipo de expertos de la corrección de su *demostración*, lo que implicaría la corrección del código. Finalmente, se realizan pruebas de integración y de alto nivel, no siendo necesarias las de más bajo nivel, ya que son sustituidas por el proceso más fiable de la inspección de la demostración.

Índice de Referencias Empleadas

[Apt97], *45*
[Beizer90], *54*
[Beizer95], *56*
[Ben-Ar83], *51*
[Bensal98], *61*
[Bérard01], *47*
[Bjorne00], *43*
[Bryant92], *50*
[Burch92], *50*
[Clarke81], *47*, *48*
[Clarke86], *47*, *48*
[Clarke92], *50*
[Clarke96], *51*
[Clarke99], *47*
[Cole00], *55*
[Courco92], *50*
[Couvre99], *50*
[Dill92], *51*
[Emerso80], *47*
[Emerso90], *49*
[Emerso93], *50*
[Floyd67], *41*, *43*
[France92], *45*
[Godefr90], *49*
[Gordon93], *43*
[Graf93], *50*
[Graf97], *61*
[Grahlm97], *51*
[Grahlm99], *51*
[Groce02], *62*
[Gunter00], *56*
[Gunter99], *56*
[Har'El90], *51*
[Hoare69], *41*, *43*, *45*
[Holzma91], *51*
[Holzma97], *51*
[Hopcro79], *41*
[Jard91], *50*
[Kaner93], *54*
[Kaufma00a], *43*
[Kaufma00b], *43*
[Lee96], *61*
[Lewis97], *41*, *64*
[Lichte85a], *48*
[Manna83], *45*
[Manna94], *43*
[McMill93], *50*, *51*
[Meinel98], *51*
[Mills75], *63*
[Myers79], *54*
[Namjos00], *61*
[Nascim94], *63*
[Nipkow02], *43*
[Owre92], *43*
[Papadi94], *41*
[Patton00], *54*
[Peled01a], *41*
[Peled01b], *61*
[Peled93], *49*
[Peled99], *61*
[Pnueli86a], *48*
[Quiell81], *47*
[Schnei97], *45*
[Sharyg01], *59*
[Sipser96], *41*
[Stavel99], *63*
[Valmar90], *49*
[Vardi86], *48*
[VIS96], *51*
[Wolper87], *49*
[Wolper95], *47*

Capítulo 4

Verificación Basada en Autómatas

Contenido

RESUMEN Y ORGANIZACIÓN

En este capítulo se presenta el marco en el que abordaremos el problema de la especificación y verificación de las propiedades temporales de un sistema, teniendo en cuenta que nuestro interés se centra en los sistemas reactivos en general, y dentro de éstos, especialmente en los sistemas concurrentes. Así, la primera sección describe cómo se van a modelar los distintos procesos que componen el sistema (concurrente) que se pretende verificar. Al igual que hacen la mayoría de las técnicas de verificación existentes en la actualidad, modelaremos esos procesos (o sea, el sistema) mediante autómatas finitos. Del mismo modo, siguiendo la tradición, la especificación temporal que dicho sistema debe cumplir, inicialmente expresada mediante fórmulas de una lógica temporal (en nuestro caso: FIL), también se suele representar finalmente mediante un autómata, con el fin de llevar a cabo la verificación. Así, en la segunda sección se presenta, de un modo general e intuitivo, la estrategia de conversión de especificaciones FIL a autómatas de Büchi, así como los fundamentos teóricos en los que se basa ese proceso de traducción. La tercera sección expone el método de verificación en el que se aplica directamente lo desarrollado en esta tesis, explicando los pasos que componen el procedimiento a seguir para verificar automáticamente sistemas concurrentes, y cómo implementar eficientemente dicho procedimiento. De igual modo, la última sección muestra cómo se puede utilizar también el trabajo realizado para comprobar la satisfacibilidad o la validez de una especificación FIL.

4.1. Modelado de procesos

Tal y como se ha explicado en el apartado 2.4.3. Formalización del concepto de modelo, al ser FIL una lógica de tiempo discreto y lineal, cualquiera de sus fórmulas se interpreta sobre un modelo, consistente en una *ω-traza* lineal $\sigma=\langle\sigma(i)\rangle_{i\in\omega}$ de estados $\sigma(i)$, que representa una posible ejecución de un proceso P, de modo que $\sigma\in P$. De esta forma, se identifica un *proceso* con el conjunto de sus comportamientos observables, o sea, un proceso se representará mediante un conjunto numerable de trazas.

Dado que estamos interesados en la verificación de los sistemas reactivos en general y de los sistemas concurrentes en particular, y como estos sistemas suelen tener ejecuciones infinitas, consideraremos que cada traza de un proceso es infinita. Con ello no perdemos generalidad en el modelo adoptado, ya que, como se ha comentado en el apartado 2.4.3, una traza finita puede ser modelada por la traza infinita obtenida al repetir mediante *tartamudeo* su último estado.

4.1.1. Representación de los procesos mediante autómatas de Büchi

Puesto que se identifica un proceso con el conjunto de sus comportamientos, una representación concreta y muy operacional para un proceso podría ser un sistema de transición de estados finitos, capaz de generar un conjunto de *ω*-trazas (aquellas permitidas por su grafo de transición). Así, de aquí en adelante, se modelará un proceso mediante un tipo de *ω-autómata* [Thomas90], denominado *autómata de Büchi* [Büchi62], que es un autómata finito sobre palabras o cadenas infinitas, por lo que tiene un criterio de aceptación especial.

Definición 4.1: Autómata de Büchi

Un *autómata de Büchi*, A, se define mediante una quíntupla $A=(\Sigma,S,\rho,I,F)$, donde:

- Σ es el *alfabeto*, formado por un conjunto finito de símbolos de entrada.
- S es un conjunto finito de *estados*.

- $\rho : S \times \Sigma \to 2^S$ es la *función de transición*, que para cada par compuesto por un estado y un símbolo de entrada devuelve el conjunto de posibles estados siguientes. Obsérvese que se trata de una función *no determinista*, o sea, dado un estado y un determinado símbolo de entrada, puede haber más de un estado siguiente.
- $I \subseteq S$ es el conjunto de *estados iniciales*.
- $F \subseteq S$ es el conjunto de *estados de aceptación*. ■

Se puede decir, pues, que un autómata de Büchi es un formalismo abstracto para definir o representar conjuntos de palabras o cadenas (trazas o secuencias) infinitas, que son aquellas que son *aceptadas* o *reconocidas* por el autómata. El conjunto de palabras que son aceptadas por el autómata se define usando la noción de *ejecución*, definida de la siguiente forma:

Definición 4.2: Ejecución de un autómata de Büchi

Una *ejecución de un autómata de Büchi A* sobre una palabra infinita $\xi \in \Sigma^\omega$ es una ω-traza $\sigma \in S^\omega$, tal que $\sigma(0) \in I$ y $\forall i \in \omega,\ \sigma(i+1) \in \rho(\sigma(i), \xi(i))$. ■

Dicho de otro modo, la definición anterior expresa que una ejecución de un autómata sobre una palabra infinita es una secuencia infinita de estados, tal que el primero es un estado inicial y todos los demás se obtienen con arreglo a la función de transición. Los autómatas de Büchi son *no deterministas*, lo que implica que puede haber varias ejecuciones distintas del autómata para una determinada palabra infinita.

Definición 4.3: Ejecución de aceptación en un autómata de Büchi

En un autómata de Büchi, una *ejecución* es *de aceptación* si contiene cualquier estado de aceptación un número infinito de veces. Expresado de una manera más formal, se podría decir que si $inf(\sigma) = \{s \in S \mid s$ aparece infinitamente a menudo en la ejecución $\sigma\}$, entonces una ejecución σ es de aceptación si y sólo si $inf(\sigma) \cap F \neq \emptyset$. ■

Obsérvese que esta condición de aceptación especial es obligada, puesto que, al no ejecutarse estos autómatas sobre palabras finitas, no sirve el habitual

establecimiento de una serie de estados finales como condición de aceptación, dado que no existe un último estado.

Una *palabra* es *aceptada* por un autómata de Büchi si existe una ejecución de aceptación del autómata sobre esa palabra. El *lenguaje aceptado por un autómata A* es el conjunto de palabras infinitas aceptadas por A, y se denota como $\mathscr{L}(A)$. Los lenguajes aceptados por los autómatas de Büchi se denominan *lenguajes ω-regulares* y se caracterizan por estar formados por cadenas finitas (de símbolos de su alfabeto) que se repiten infinitamente. Entre las propiedades más interesantes de la clase de los lenguajes ω-regulares está el hecho de que esta clase es *cerrada bajo las operaciones de unión, intersección (o producto) y complementación.* Esto significa que dados dos autómatas de Büchi, A_1 y A_2, que aceptan respectivamente los lenguajes $\mathscr{L}(A_1)$ y $\mathscr{L}(A_2)$, hay autómatas de Büchi que aceptan los lenguajes $\mathscr{L}(A_1)\cup\mathscr{L}(A_2)$, $\mathscr{L}(A_1)\cap\mathscr{L}(A_2)$ y $\Sigma^{\omega}-\mathscr{L}(A_1)$ [Gribom89] [Thomas97].

Dado que se va a modelar un proceso mediante el correspondiente autómata, se puede dotar a cada estado de su grafo de transición con transiciones reflexivas, con el fin de que dicho proceso sea insensible al tartamudeo finito. Esas transiciones reflexivas, junto con las condiciones infinitas de aceptación expresadas en la Definición 4. 3, hacen que el conjunto de trazas definido por un *proceso* sea (maximalmente) *cerrado bajo tartamudeo finito.*

4.1.2. Composición consistente de procesos

En un sistema concurrente, las ejecuciones de cada uno de sus procesos tendrán que componerse para dar lugar a la ejecución del sistema global. Por tanto, y dado que una ejecución de un proceso no es más que una ω-traza, lo primero que se va a definir es la composición de trazas:

Definición 4.4: Composición paralela de trazas

Sean $P_1, P_2, \ldots, P_n$ un conjunto finito de procesos (tareas o subsistemas dentro del sistema global) y $\sigma_1, \sigma_2, \ldots, \sigma_n$ trazas que representan respectivamente una posible ejecución de cada uno de dichos procesos. Se define el operador *composición paralela* || sobre esas trazas de la siguiente manera:

$$\sigma_1 \,\|\, \sigma_2 \,\|\, \ldots \,\|\, \sigma_n \;=\; \langle(\sigma_1(i),\, \sigma_2(i),\, \ldots,\, \sigma_n(i))\rangle_{i\in\omega} \qquad \blacksquare$$

Así, cada estado de la traza compuesta, $s_i = (\sigma_1(i), \sigma_2(i), ..., \sigma_n(i))$, es una fotografía o instantánea global del sistema, esto es, un *estado global*.

Definición 4.5: Composición de procesos

Haciendo uso de la definición anterior, se define la *composición de* (un conjunto finito de) *procesos* como sigue:

$$\|_{j\in\{1,...,n\}} P_j = \{\|_{j\in\{1,...,n\}} \sigma_j \mid \sigma_j \in P_j\}$$ ■

Puesto que, como se ha dicho al final del apartado 4.1.1, los procesos son cerrados bajo tartamudeo finito, con la definición anterior, su composición también lo es. Por otra parte, y de acuerdo con nuestra visión y manejo de los procesos como autómatas de Büchi, la composición de procesos así definida se reemplazaría por el producto (síncrono) de los autómatas que representan a cada uno de los procesos P_j, siendo ambas operaciones (la composición de procesos y el producto de sus autómatas) exactamente equivalentes. El resultado práctico de esta propiedad de composicionalidad de procesos es que se puede describir por separado el comportamiento de los distintos subsistemas que componen un sistema, simplificándose así la tarea de la descripción global de sistemas complejos.

Aunque no se necesitará usar la noción formal de *evento* (recuérdese que nuestro modelo está basado en estados, no en eventos), ésta puede ser introducida del siguiente modo: Supongamos que cualquier traza o ejecución de un proceso se produce gracias a la ocurrencia de eventos. Si se piensa en un proceso modelado mediante un autómata finito, estos eventos serían los que etiquetarían los arcos que unen o relacionan dos nodos, indicando que cuando ocurren dichos eventos se producen transiciones instantáneas entre los estados que representan esos nodos. Desde este punto de vista, cuando en un sistema concurrente se componen las ejecuciones de sus procesos, los eventos que aparecerán sobre cada transición de la composición (transición global) serán todos aquellos que aparecen sobre cada una de las transiciones que componen dicha transición global. Así, la concurrencia se representaría mediante el conjunto de eventos simultáneos que se producen en los distintos procesos.

La sincronización entre procesos se produce cuando varios procesos comparten ciertos elementos, denominados *variables de estado* o *eventos*, según el punto de vista adoptado para observar el sistema. Para nuestros propósitos, es indiferente el

punto de vista adoptado, ya que modelaremos dichos elementos (variables o eventos, según el caso) como *proposiciones atómicas*. En este contexto, nos debemos plantear la noción de consistencia de una composición de procesos. Para ello, presentamos a continuación una serie de definiciones previas:

Definición 4.6: Consistencia de un estado de un proceso

Sea $\mathscr{P}$ el conjunto de proposiciones atómicas del sistema considerado, un *estado* $\sigma_j(i)$ de un proceso P_j es *inconsistente* si y sólo si p y $\neg p$ se dan en él, donde $p \in \mathscr{P}$. En caso contrario, se dice que dicho estado es *consistente*. ■

Extendiendo la definición anterior para un conjunto de estados, obtenemos la siguiente definición:

Definición 4.7: Consistencia de un estado global

Un *estado global* $s_i = (\sigma_1(i), \sigma_2(i), ..., \sigma_n(i))$ es *consistente* si y sólo si el conjunto formado por todos sus componentes lo es. ■

Obviamente, la consistencia de un conjunto de estados tiene la siguiente propiedad: Si un conjunto S de estados es inconsistente, entonces también lo es cualquier superconjunto de S.

Finalmente, para definir la composición consistente de procesos, sólo hemos de refinar la Definición 4.4 del siguiente modo:

Definición 4.8: Composición consistente de trazas

Una *traza o ejecución global* $\sigma = \sigma_1 \parallel \sigma_2 \parallel ... \parallel \sigma_n$ es *consistente* si y sólo si todos sus estados (globales) lo son. ■

4.2. Construcción de autómatas a partir de especificaciones FIL

Puesto que FIL es una lógica de tiempo discreto y lineal, una *interpretación temporal* se puede expresar como la relación $\subseteq \mathbb{N} \times 2^{\mathscr{P}}$, donde $\mathbb{N}$ es el conjunto de los

números naturales y $\mathcal{P}$ es el conjunto de las proposiciones atómicas. Así, una interpretación temporal es una *palabra infinita* sobre el alfabeto $2^{\mathcal{P}}$ (que representa el conjunto de subconjuntos de $\mathcal{P}$, o sea, las distintas asignaciones posibles de valores de verdad a cada una de las proposiciones atómicas del sistema especificado).

Desde este punto de vista, una *fórmula* FIL define un *lenguaje de palabras infinitas*, es decir, el conjunto de interpretaciones temporales que la satisfacen. Puesto que, como se ha visto en el apartado 4.1.1, un autómata de Büchi también define un lenguaje de palabras infinitas, parece lógico pensar que para una determinada fórmula FIL (a partir de ella) se pueda construir un autómata de Büchi que acepte exactamente el mismo conjunto de palabras infinitas que satisfacen esa fórmula. Las conexiones existentes entre los autómatas de Büchi y varios tipos de lógicas, entre ellas las temporales, pueden encontrarse en [Thomas90].

Consecuentemente, el propósito de esta sección es el de dar una introducción general e intuitiva de cómo se debe realizar dicha construcción (ver apartado 4.2.1), además de formalizar los conceptos teóricos sobre los que ésta se basa (ver apartado 4.2.2). Los detalles del algoritmo que lleva a cabo este proceso de construcción pueden encontrarse en el Capítulo 5.

4.2.1. Descripción general de la estrategia de construcción

Nuestro objetivo es construir un autómata de Büchi que pueda generar todas las secuencias infinitas que satisfagan una determinada propiedad expresada mediante una fórmula FIL, de tal modo que la correspondencia entre las estructuras aceptadas por el autómata y los modelos que satisfacen dicha fórmula sea una biyección.

Puesto que el autómata generado debe ser semánticamente equivalente a la fórmula FIL de la que se obtiene, dicho autómata debe "codificar" exactamente las mismas "condiciones" que esa fórmula expresa. En el Ejemplo 4.1 se muestran claramente cuáles son esas condiciones.

Ejemplo 4.1: Condiciones a comprobar por el autómata correspondiente a una fórmula FIL

Considérese la fórmula $f = [\rightarrow a \mid \rightarrow b)\Diamond c$, en la que la satisfacibilidad de $\Diamond c$ debe restringirse al contexto identificado por $[\rightarrow a \mid \rightarrow b)$. Para comprobar si f se cumple sobre un determinado modelo, el autómata que se genere debe verificar una de las dos condiciones siguientes:

1. Que el contexto $[\rightarrow a \mid \rightarrow b)$ no se pueda construir, cosa que ocurre si y sólo si una de las siguientes condiciones (codificadas en el autómata de Büchi (A) de la Figura 4.1) se cumple:

   (a) a nunca se cumple sobre el modelo (por tanto, la búsqueda izquierda fracasa).

   (b) b nunca se cumple sobre el modelo (por tanto, la búsqueda derecha fracasa).

   (c) el primer estado en el que b se cumple precede a, o coincide con, el primer estado en el que a se cumple (por tanto, el contexto resultante es vacío).

2. Que el contexto $[\rightarrow a \mid \rightarrow b)$ sea construible (esto implica que a debe cumplirse estrictamente antes que b) y que dentro de este contexto $\Diamond c$ se cumpla. Para ello el autómata debe comprobar que, una vez que se da a (suceso que marca el inicio del contexto), las fórmulas $\Diamond b$ y $[- \mid \rightarrow b)\Diamond c$ se cumplen sobre el sufijo restante de la ejecución de dicho modelo. Así, si a partir de dicho punto, el autómata encuentra una ocurrencia de b antes o en el mismo punto que donde se da una c, entonces el autómata rechaza ese modelo, puesto que esto significa que se alcanza el fin del contexto sin que se haya cumplido c dentro del mismo. Por el contrario, si el autómata encuentra que c se da antes que b, entonces sólo tiene que comprobar que finalmente se da b para aceptar ese modelo. El autómata de Büchi (B) de la Figura 4.1 codifica la condición expresada por este párrafo.

De esta forma, tal y como se estableció en el apartado 2.4.2. Semántica informal, la fórmula f se satisface *propiamente* si se cumple la segunda de las condiciones anteriores, pero si el contexto temporal definido por su intervalo no puede construirse (se da la primera condición), entonces se dice que f se satisface *vacuamente*. Por consiguiente, el autómata de propiedad que se debe construir para dicha fórmula será el autómata (C) de la Figura 4.1, cuyo lenguaje es la unión de los

lenguajes de los autómatas de Büchi (A) y (B) de la misma figura, siendo este autómata (C) una representación exactamente equivalente a *f*, aunque más operacional.

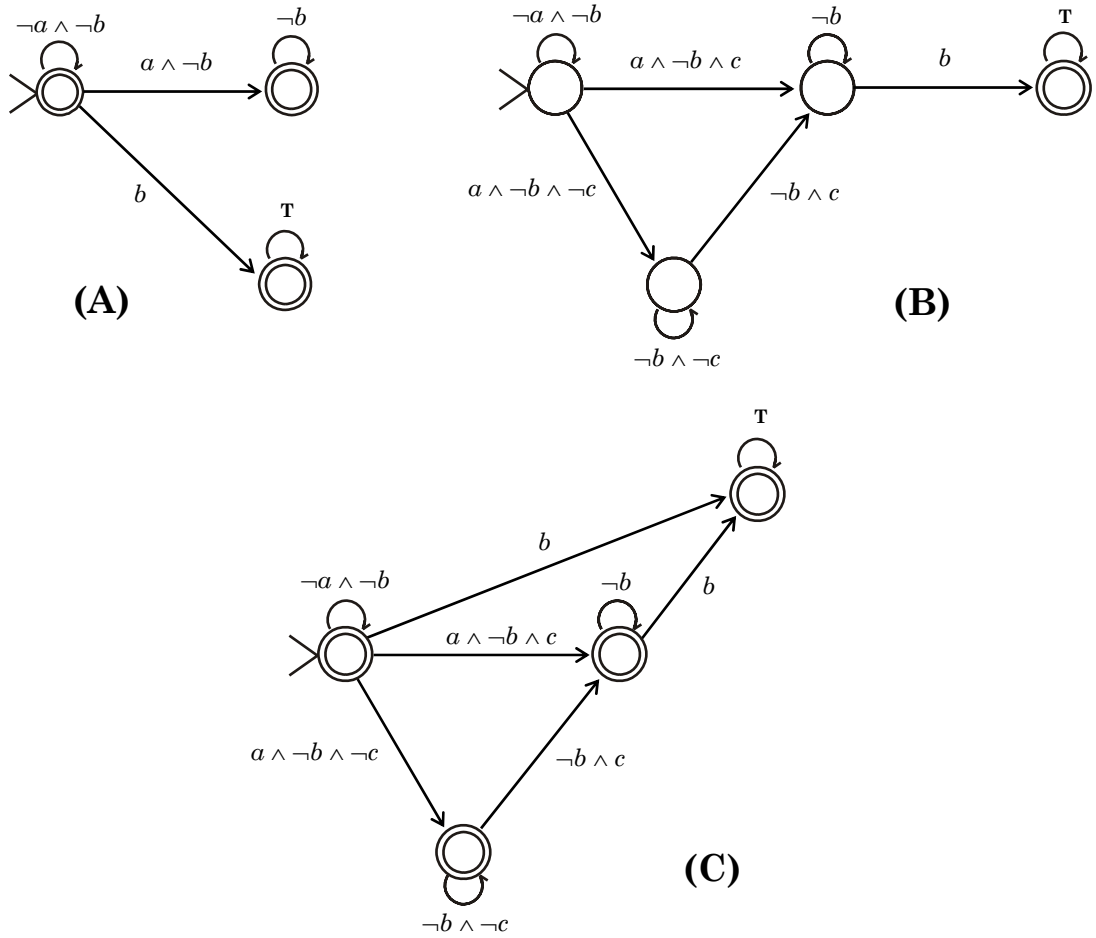


Figura 4.1. Autómatas de Büchi relacionados con la fórmula $[\rightarrow a \mid \rightarrow b)\Diamond c$: **(A)** Comprueba la imposibilidad de construir el intervalo $[\rightarrow a \mid \rightarrow b)$. **(B)** Verifica que el contexto $[\rightarrow a \mid \rightarrow b)$ se puede construir y que *c* se satisface dentro del mismo. **(C)** Su lenguaje es la unión del de los dos anteriores

Todas las "condiciones de verificación" intermedias que se generan durante el proceso de construcción del autómata son "codificables" en FIL, esto es, el autómata recuerda estas condiciones escribiendo las fórmulas FIL correspondientes, que serán las que etiqueten cada nodo del autómata resultante, siendo dichas fórmulas sintácticamente "más simples" que la fórmula original que las generó.

A continuación, se procederá a formalizar estas nociones que, de manera intuitiva, se acaban de dar. Para ello, y con el fin de determinar sucintamente los

estados y transiciones del autómata, en el siguiente apartado se va a definir una relación de orden parcial, parametrizada e irreflexiva, entre *fórmulas de intervalo* (exclusivamente afecta a este tipo de fórmulas), a la que se denomina *relación de reducción*.

4.2.2. Reducción de las fórmulas de intervalo

A medida que se van produciendo determinados eventos o acciones (se satisfacen determinadas proposiciones atómicas) en un punto o instante (estado) de una ejecución (modelo), la fórmula inicial que constituye la propiedad a partir de la que se quiere obtener el autómata se irá reduciendo (haciendo más simple). Se expondrá un ejemplo para que se entienda mejor lo que se acaba de comentar.

Ejemplo 4.2: Introducción a la reducción de fórmulas

Sea $f = [\rightarrow a \mid \rightarrow b)\Diamond c$ la fórmula del Ejemplo 4.1, que en la sintaxis restringida de FIL (ver Definición 2.1) se expresa como $f = [\rightarrow a \mid \rightarrow b)\neg[\rightarrow c \mid \rightarrow)\mathbf{F}$. Supóngase que en un determinado punto i de un modelo $\mathcal{M}$ tanto f como a se cumplen. Claramente, la búsqueda de a en f localiza el estado $\mathcal{M}(i)$, de manera que la fórmula $[- \mid \rightarrow b)\neg[\rightarrow c \mid \rightarrow)\mathbf{F} = [- \mid \rightarrow b)\Diamond c$ también debe cumplirse en el punto i de $\mathcal{M}$. Además, la fórmula $[- \mid \rightarrow b)[\rightarrow c \mid \rightarrow)\mathbf{F} = [- \mid \rightarrow b)\Box\neg c$ no debe cumplirse en i, a menos que ocurra una de estas dos cosas: que b se cumpla en i (es decir, que el intervalo se colapse) o que $[\rightarrow b \mid \rightarrow)\mathbf{F} = \Box\neg b$ se cumpla en i (o sea, que la búsqueda para b fracase).

A continuación se formalizará el concepto de reducción de una fórmula de intervalo, para lo que previamente se definirá el denominado *conjunto reductor* de dicha fórmula. Ambos conceptos han sido definidos en [Ramakr93a] [Ramakr96a] y se recogen aquí con el fin de que el material presentado en esta memoria sea completo, ya que dichos conceptos tienen una importancia capital para la comprensión del núcleo de esta tesis: nuestro algoritmo de traducción (automática) de fórmulas FIL a autómatas de Büchi, tal y como se pondrá de manifiesto en la sección 5.1 del Capítulo 5.

Notación: Sea I una modalidad de intervalo y F un conjunto de fórmulas. Se denotará como $I.F$ al conjunto de fórmulas $\{I\,f \mid f \in F\}$.

Definición 4.9: Conjunto reductor de una fórmula de intervalo

El *conjunto reductor* $\mathbf{red}(f)$ de una fórmula de intervalo f es el menor conjunto de fórmulas bien formadas que no contiene a f, tal que se cumplen las siguientes reglas:

- si $f = [\rightarrow\tau, \theta_1 \mid \theta_2)\,f_1$, $[\theta_1 \mid \rightarrow\tau, \theta_2)\,f_1$, $[\rightarrow\tau \mid \theta_2)\,f_1$ o $[\theta_1 \mid \rightarrow\tau)\,f_1$, entonces $\tau \in \mathbf{red}(f)$
- si $f = \neg f_1$, entonces $\mathbf{red}(f_1) \subseteq \mathbf{red}(f)$
- si $f = [\theta_1 \mid \theta_2)\,f_1$, entonces
  - si $\theta_1 \neq -$, entonces $[\theta_1 \mid \rightarrow)\mathbf{F} \in \mathbf{red}(f)$
  - si $\theta_2 \neq \rightarrow$, entonces $[\theta_2 \mid \rightarrow)\mathbf{F} \in \mathbf{red}(f)$
- si $f = [- \mid \theta_2)\,f_1$, entonces $[- \mid \theta_2).\mathbf{red}(f_1) \subseteq \mathbf{red}(f)$
- si f es de cualquier otra forma, entonces $\mathbf{red}(f) = \varnothing$ ■

Si τ y f son fórmulas FIL, entonces se dice que f es *τ-reducible* si y sólo si $\tau \in \mathbf{red}(f)$. Por el contrario, se habla de que f es *τ-irreducible* cuando no es τ-reducible. El conjunto reductor para una fórmula contiene exactamente aquellas subfórmulas que pueden servir como parámetros para una "operación de reducción" sobre esa fórmula.

Definición 4.10: Relación de reducción

Entre dos fórmulas FIL, f y f', siendo f una fórmula de intervalo, existe una *relación de reducción* con respecto a otra fórmula τ, tal que $\tau \in \mathbf{red}(f)$, denotada como $f \succ_\tau f'$ (se lee: "f es *τ-reducible* a f'" o "f' es un *τ-reducto* de f"), si y sólo si se cumple uno de los siguientes puntos:

1. $f = [\rightarrow\tau, \theta_1 \mid \theta_2)\,f_1$ o $[\theta_1 \mid \rightarrow\tau, \theta_2)\,f_1$ o $[\rightarrow\tau, \theta_1 \mid \rightarrow\tau, \theta_2)\,f_1$ y $f' = [\theta_1 \mid \theta_2)\,f_1$
2. $f = [\rightarrow\tau \mid \theta_2)\,f_1$ o $[\rightarrow\tau \mid \rightarrow\tau, \theta_2)\,f_1$ y $f' = [- \mid \theta_2)\,f_1$
3. $f = [\theta_1 \mid \rightarrow\tau)\,f_1$ y $f' = \mathbf{T}$
4. $f = [\theta_1 \mid \theta_2)\,f_1$, $\tau = [\theta_1 \mid \rightarrow)\mathbf{F}$ o $[\theta_2 \mid \rightarrow)\mathbf{F}$ y $f' = \mathbf{T}$

5. $f = [\rightarrow \tau | \rightarrow) f_1$ y $f' = f_1$
6. $f = \neg f_1$, $f' = \neg f'_1$ y $f_1 \ {}_{\tau}\succ f'_1$
7. $f = [- | \theta_2) f_1$, $\tau = [- | \theta_2) \pi$, $f' = [- | \theta_2) f'_1$ y $f_1 \ {}_{\pi}\succ f'_1$ ■

Intuitivamente, el proceso de reducción se aplica a una determinada fórmula de intervalo f (cuya estructura viene dada por $I\ f_1$) con el fin de simplificarla conforme se van cumpliendo las condiciones que dicha fórmula codifica. De este modo, se verifica si el intervalo I de f no se puede construir (en cuyo caso f sería vacuamente cierta) o si, por el contrario, dicho intervalo se puede construir y, dentro del contexto que establece, se satisface la fórmula f_1 anidada al mismo (con lo que la fórmula f se satisfaría apropiadamente). A la luz de esto se comentarán algunos de los puntos más difíciles de comprender de la definición anterior: En el punto 3, el τ-reducto de f es **T** porque se determina que el intervalo más externo de f no se puede construir, ya que si $\theta_1 = -$ o $\theta_1 = \rightarrow a$, la ocurrencia de a en ese instante conlleva el colapso de dicho intervalo (los dos extremos del intervalo se satisfacen en el mismo instante), mientras que si θ_1 tiene cualquier otra forma, que a se dé en ese estado significa que se encuentra el extremo derecho del intervalo antes que su extremo izquierdo. El punto 4 expresa la imposibilidad de construir el intervalo, al fallar la búsqueda de uno de sus dos extremos. El punto 7 indica que, cuando el intervalo más externo es una modalidad de intervalo actual, antes de que se encuentre su extremo derecho, se debe reducir (cumplir) la fórmula anidada a dicho intervalo.

Ejemplo 4.3: Reducción de una fórmula de intervalo

Para la fórmula f del Ejemplo 4 .1 y del Ejemplo 4.2 , la Figura 4.2 ilustra las definiciones que se acaban de dar. En la figura, se representa $f_1 \ {}_{\tau}\succ f_2$ mediante un arco desde f_1 (arriba) hasta f_2 (abajo) etiquetado con el reductor τ. Múltiples etiquetas sobre un arco indican un surtido de reductores aplicables, llevando todos ellos a la misma f_2. Para una fórmula, el conjunto de etiquetas de todos sus arcos salientes es exactamente el conjunto reductor para esa fórmula.

Obsérvese que si una fórmula f es τ-reducible a f', esto es, $f \ {}_{\tau}\succ f'$, entonces tanto f' como τ son "más simples" que f. Por tanto, las reducciones son esencialmente un mecanismo para "simplificar" las condiciones de verificación que

surgen durante el procedimiento de construcción del autómata que representa la propiedad inicialmente especificada en FIL.

Por consiguiente, se puede concluir este apartado diciendo que una fórmula f' es una reducción de una fórmula de intervalo f con respecto a otra fórmula τ, y se denota como $f \succ_{\tau} f'$, si f' es una subfórmula (apropiada) de f y cuando τ se satisface en un estado de un modelo, entonces f se cumple en ese estado si y sólo si f' lo hace. Esto se puede formalizar del siguiente modo: Si τ, f' y f son fórmulas FIL y $\mathcal{M}$ es un modelo tal que $\langle \mathcal{M}, i \rangle \models \tau$ y $f \succ_{\tau} f'$, entonces $\langle \mathcal{M}, i \rangle \models f$ sii $\langle \mathcal{M}, i \rangle \models f'$.

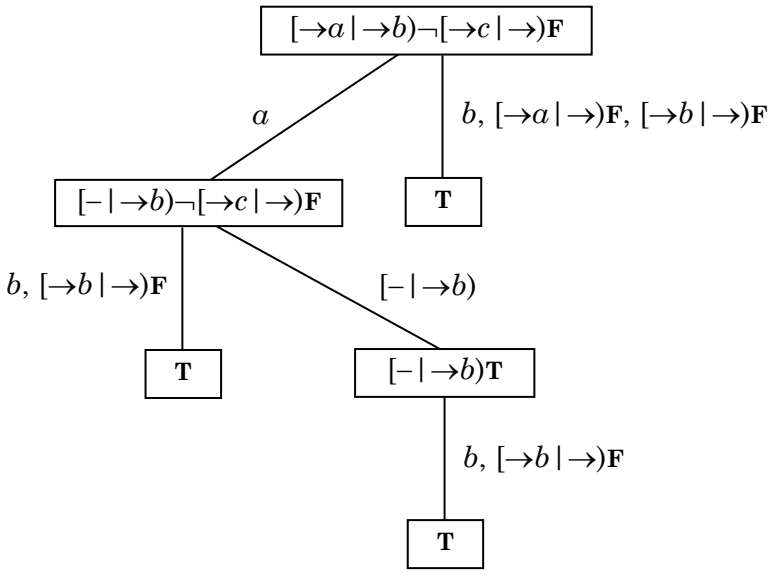


Figura 4.2. Reducciones para la fórmula $f = [\rightarrow a \mid \rightarrow b) \Diamond c$

4.3. Verificación mediante comprobación de modelos

El problema concreto que nosotros abordamos [Hornos01b] es el de la *comprobación de modelos* (*model checking*) [Wolper95] [Clarke99] [Bérard01], planteado en los siguientes términos: Dado un programa P, descrito como la ejecución concurrente de varios procesos P_i (cada uno de ellos representado mediante un sistema de transición de estados finitos) y una especificación del sistema φ (expresada mediante una fórmula FIL), comprobar que todas las ejecuciones infinitas de P satisfacen la especificación φ.

El enfoque para comprobación de modelos que se presenta aquí se basa en la teoría de autómatas. Así, tal y como se ha puesto de manifiesto en las dos primeras secciones de este capítulo, tanto el sistema que se desea verificar como la especificación que éste debe cumplir se pueden representar como autómatas de Büchi sobre el mismo alfabeto. El marco basado en teoría de autómatas para comprobación de modelos fue sugerido inicialmente y de forma independiente por Kurshan y otros [Aggarw83] [Kursha94] y por Vardi y Wolper [Vardi86].

4.3.1. Esquema con los pasos a realizar

Para resolver el problema que se acaba de plantear, se deben realizar estos pasos:

1. Construir el denominado *autómata de propiedad* para $\neg\varphi$. Denotaremos como $A_{\neg\varphi}$ al autómata de Büchi resultante. Más adelante (en el próximo apartado) se explica que se utiliza la negación de φ debido a que produce un algoritmo más eficiente.

2. Calcular el *autómata del sistema* (programa o protocolo) P, realizando el producto ΠP_i de los sistemas de transición de los distintos procesos P_i que integran el programa P, con lo que el autómata resultante determinará el comportamiento global de P.

3. Hallar el *autómata producto*, $P \times A_{\neg\varphi}$, que aceptará todas las ejecuciones infinitas que sean posibles comportamientos de P (aceptadas por el autómata del sistema P) y que violen (no satisfagan) la especificación φ (aceptadas por el autómata de propiedad $A_{\neg\varphi}$).

4. Comprobar el *no vacío* del autómata producto, es decir, comprobar que el lenguaje $\mathscr{L}(P \times A_{\neg\varphi})$ es no vacío, circunstancia que indica la existencia de ejecuciones del programa P que violan la especificación φ, por lo que se puede generar un *contraejemplo* a partir de dicho autómata. Si el programa P es correcto (satisface la especificación φ), el lenguaje $\mathscr{L}(P \times A_{\neg\varphi})$ será vacío.

Obsérvese que, de este modo, la comprobación de modelos se reduce a comprobar el no vacío del autómata producto, siendo necesario un buen algoritmo para realizar dicha comprobación. En el siguiente apartado se describirá más detalladamente el significado del último punto del procedimiento presentado, mientras que en el apartado 4.3.3 se indicará cómo implementar dicho procedimiento de una manera eficiente.

4.3.2. Comprobación del no vacío de un autómata

La Figura 4.3 ilustra gráficamente el significado del cuarto punto expuesto anteriormente; en ella denotamos como Σ^{ω} al conjunto de todas las palabras infinitas que pueden formarse a partir del alfabeto Σ (donde $\Sigma=2^{\mathscr{P}}$, siendo $\mathscr{P}$ el conjunto de proposiciones atómicas), y como A_{φ} al autómata de propiedad que representa el conjunto de comportamientos que satisfacen la especificación φ.

Por consiguiente, el programa P satisface la especificación φ (Figura 4.3(a)) si y sólo si se cumple:

$$\mathscr{L}(P) \subseteq \mathscr{L}(A_{\varphi}) \tag{4.1}$$

Esta expresión afirma que cada comportamiento del programa P (conjunto con fondo verde) está entre los comportamientos permitidos por la especificación φ (conjunto con fondo amarillo).

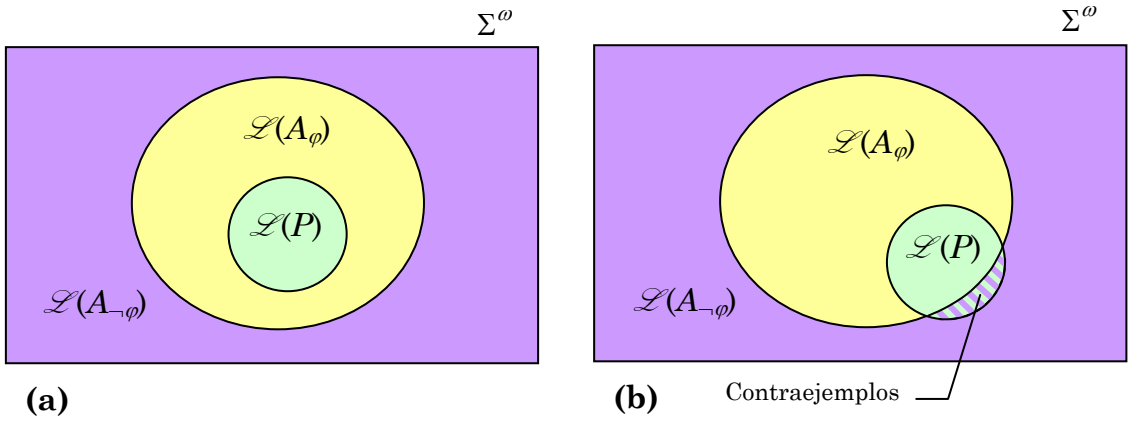


Figura 4.3. (a) El programa P cumple la especificación φ; **(b)** El programa P viola la especificación φ

Teniendo en cuenta que $\mathscr{L}(A_{\neg\varphi}) = \overline{\mathscr{L}(A_{\varphi})} = \Sigma^{\omega} - \mathscr{L}(A_{\varphi})$ y que no se va a construir el autómata A_{φ}, sino el $A_{\neg\varphi}$, la expresión (4.1) puede ser reformulada del siguiente modo:

$$\mathscr{L}(P \times A_{\neg\varphi}) = \mathscr{L}(P) \cap \mathscr{L}(A_{\neg\varphi}) = \varnothing \tag{4.2}$$

Esta última expresión indica que no hay ningún comportamiento del programa P (conjunto con fondo verde) que pertenezca al conjunto de comportamientos que incumplen o violan la especificación φ (conjunto con fondo morado).

Obsérvese que las expresiones (4.1) y (4.2) son equivalentes. Sin embargo, es más fácil implementar un algoritmo para comprobar el vacío del lenguaje de la intersección de dos autómatas que un algoritmo para comprobar la inclusión del lenguaje de un autómata en el de otro. Por otra parte, complementar un autómata de Büchi es difícil [Sistla87], pudiendo ser el autómata resultante exponencialmente más grande que aquél del que se obtiene. Por este motivo, muchas implementaciones evitan realizar explícitamente la complementación. Así, en algunas de ellas, como en [Holzma91], el usuario debe especificar directamente los comportamientos no permitidos en lugar de los permitidos. Otra estrategia, seguida por ejemplo en COSPAN [Kursha94], consiste en utilizar un tipo de ω-autómata más complicado, pero que permite realizar la complementación de forma simple. Sin embargo, la estrategia más utilizada para evitar la complementación en herramientas que traducen una especificación φ de una lógica temporal a un autómata de Büchi consiste en colocar un símbolo de negación delante de dicha especificación y traducir directamente $\neg\varphi$ a un autómata, en lugar de primero convertir φ a un autómata y después complementarlo, lo que puede provocar un incremento de tamaño doblemente exponencial.

Si la expresión (4.2) no se cumple, o sea, si $\mathscr{L}(P \times A_{\neg\varphi}) \neq \varnothing$, significa que el programa P tiene comportamientos que no están permitidos por la especificación φ (zona rayada en la Figura 4.3(b)), por lo que cualquier comportamiento de la intersección de los lenguajes $\mathscr{L}(P) \cap \mathscr{L}(A_{\neg\varphi})$ constituye un *contraejemplo*.

Se podrá determinar que el lenguaje aceptado por un autómata es no vacío si se encuentra una ejecución de aceptación sobre el mismo. Para que esto ocurra (puesto que, por definición, una ejecución de aceptación es aquella que contiene cualquier estado de aceptación un número infinito de veces), debe haber un estado de aceptación que sea alcanzable desde el estado inicial y que pertenezca a un ciclo (o sea, alcanzable también desde sí mismo). Esta propiedad se puede expresar más correctamente en términos de la teoría de grafos [Gibbon85] [Dieste00] [Hopcro79] diciendo que debe existir un *componente fuertemente conectado* que contenga un estado de aceptación y que sea alcanzable desde el estado inicial. Por consiguiente, el problema de comprobar el no vacío de un autómata quedará resuelto cuando se encuentre un estado de aceptación que pertenezca a un ciclo (o componente fuertemente conectado) alcanzable desde el estado inicial. Para ello, puede aplicarse el algoritmo de *búsqueda primero en profundidad* (BPP) de Tarjan

[Tarjan72]; sin embargo, en el siguiente apartado se describirá un algoritmo alternativo, que es más eficiente para realizar comprobación de modelos.

4.3.3. Comprobación de modelos *on-the-fly*

La *comprobación de modelos explícita* realiza los cuatro pasos indicados en el apartado 4.3.1 de un modo secuencial. Por tanto, construye un grafo (autómata de Büchi o estructura similar) para el sistema modelado (programa o protocolo) P, obtenido antes de que se empiece a calcular el autómata producto y la comprobación del no vacío de su lenguaje. En el caso de un programa concurrente, extraer dicho grafo de forma previa no es muy buena idea, ya que se puede producir una explosión combinatoria en el número de estados (ver apartado 3.3.2. El problema de la explosión de estados para más detalles).

El procedimiento descrito en el apartado 4.3.1. Esquema con los pasos a realizar se puede implementar de una manera más eficiente, utilizando la técnica denominada *comprobación de modelos "on-the-fly"* [Holzma96a] [Gerth95] [Jard91]. La idea consiste en ir realizando los tres últimos pasos (o incluso todos ellos) al mismo tiempo (así, sólo se construyen los estados alcanzables del autómata $P \times A_{\neg\varphi}$) y parar justo en el momento en que se determine el no vacío del autómata producto. De este modo, se puede detectar que un sistema viola sus especificaciones construyendo y visitando sólo una pequeña parte de su espacio de estados, evitándose así, en muchos casos, el tener que construir completamente dicho espacio de estados.

Esta técnica emplea un algoritmo [Courco92] [Holzma96b] que utiliza una doble BPP para intentar encontrar, dentro del autómata producto, un ciclo con un estado de aceptación que sea alcanzable desde el estado inicial. Para ello, ambas BPPs se entremezclan, o sea, se ejecutan de forma anidada. El objetivo de la primera BPP es localizar, a partir del estado inicial, un estado de aceptación, mientras que el de la segunda BPP consiste en encontrar un ciclo que lo contenga. De este modo, el problema de la comprobación de modelos se reduce a un *problema de alcanzabilidad*, o sea, a intentar encontrar un estado de aceptación alcanzable desde el estado inicial y desde sí mismo. Cuando esto ocurre, es posible generar un *contraejemplo*, que será la palabra infinita producida por la ejecución de aceptación encontrada, y cuya estructura se puede representar de forma finita mediante una expresión regular del tipo $u \cdot v^{\omega}$, donde tanto u como v son palabras finitas. Así, el

contraejemplo generado tendrá un prefijo finito u, seguido por una parte periódica: la cadena finita v que se repite infinitamente.

Como estructura de datos se utilizan dos pilas (una para cada BPP) y dos matrices de bits (tablas *hash*), que respectivamente sirven para marcar los nodos (poner su bit a 1) que cada BPP va visitando. Cada nodo s del autómata producto estará formado por una pareja de nodos $\langle p, a \rangle$: uno del autómata P y otro del $A_{\neg\varphi}$. Un nodo inicial será aquel en el que ambos componentes sean iniciales. Los sucesores de un nodo salen de la ejecución conjunta de los autómatas P y $A_{\neg\varphi}$, o sea, los componentes de cada sucesor se obtienen siguiendo las transiciones de cada autómata. Puesto que todos los estados del autómata P son de aceptación, un estado s del autómata producto es de aceptación si y sólo si su segundo componente, a (estado del autómata $A_{\neg\varphi}$), lo es.

La primera BPP va formando parejas de nodos (estados del autómata producto) para irlas almacenando en la primera pila. Dado que el autómata de propiedad $A_{\neg\varphi}$ se ha construido en un paso previo, el componente correspondiente al autómata P se ha de calcular conforme se vaya necesitando (construcción *on-the-fly*), de modo que sea compatible con el componente de $A_{\neg\varphi}$. Cuando se hayan visitado todos los sucesores de un nodo s_1, éste se sacará de la primera pila, activándose la segunda BPP sólo si s_1 es un estado de aceptación. El funcionamiento de la segunda BPP es similar al de la primera, sólo que en cada iteración se comprueba si se cierra el ciclo alrededor del estado de aceptación s_1 que la inició, cosa que ocurre si s_1 es sucesor del estado s_2 que actualmente está explorando la segunda BPP, o sea, si s_2 está almacenado en la primera pila. Cuando esto sucede, el procedimiento termina su ejecución, indicando que la especificación es violada y produciendo un contraejemplo a partir del contenido de ambas pilas. Si la segunda BPP no encuentra tal ciclo, la primera BPP se reanuda justo en el punto en que se interrumpió. Si finalmente la primera pila se queda vacía, sin que se haya podido encontrar ningún ciclo de aceptación, entonces se informa de que el sistema modelado cumple las especificaciones.

El contraejemplo se genera del siguiente modo: la primera pila contiene un camino desde un nodo inicial hasta el estado de aceptación s_1, constituyendo por tanto el prefijo finito del contraejemplo (denotado como u anteriormente); mientras que su parte periódica (palabra v) se obtiene como sigue: la segunda pila contiene

un camino desde s_1 hasta s_2, y dado que s_2 aparece en la primera pila, los estados insertados en ella después de s_2 completan el ciclo hasta el estado de aceptación s_1.

En resumen, hay dos formas de realizar la comprobación de modelos *on-the-fly*:

(A) En la primera [Courco92], no se tendrá que construir previamente el autómata completo del sistema P, sino sólo el autómata de propiedad $A_{\neg\varphi}$. Este autómata es, por regla general, más pequeño que el autómata P. Una vez construido $A_{\neg\varphi}$, se utilizará para guiar la construcción del autómata del sistema P, a medida que se vaya calculando la intersección o producto de ambos autómatas. Dicho de otro modo, los estados del autómata P sólo se generarán cuando sean necesarios, mientras se está comprobando el no vacío de la intersección de los lenguajes de ambos autómatas. Así, es posible detectar que un sistema no cumple una especificación antes de construir completamente el autómata P y, por consiguiente, el autómata producto.

(B) Otra forma, apuntada en [Gerth95], consiste en ir generando el autómata de propiedad $A_{\neg\varphi}$ simultáneamente con, y guiado por, la construcción del autómata del sistema P. De este modo, el algoritmo puede en muchos casos producir una respuesta construyendo sólo una parte, tanto del autómata P como del $A_{\neg\varphi}$ y, por consiguiente, del autómata producto.

Por consiguiente, realizada de una u otra forma, la ventaja fundamental de la comprobación de modelos *on-the-fly* es que hace posible una reducción del espacio de estados a construir, como consecuencia de estas dos circunstancias:

1. Que haya estados de P que no se correspondan (no sean compatibles) con ningún estado de $A_{\neg\varphi}$, por lo que nunca se generarán esos estados, ya que nunca serán necesarios para formar parte de un estado del autómata producto. De este modo, en algunos casos es posible que $P \times A_{\neg\varphi}$ tenga menos estados alcanzables que el autómata P.

2. Que un contraejemplo se encuentre antes de completar la construcción de la intersección de ambos autómatas, por lo que no será necesario continuar dicha construcción, dado que el objetivo (comprobar el no vacío del autómata producto) ya se habría conseguido.

Las ventajas de reducir la comprobación de modelos a un problema de alcanzabilidad se han estudiado también en [Jard89], pero sólo para propiedades

de seguridad. Por tanto, en él se describe un procedimiento más simple, pero menos general. Así, es suficiente comprobar con una sola BPP que algún estado "malo" (o sea, que no cumple la propiedad de seguridad) es alcanzable, devolviéndose como contraejemplo el prefijo finito que va desde el estado inicial hasta dicho estado.

4.4. Comprobación de especificaciones FIL

Aunque nuestro algoritmo de traducción de fórmulas FIL a autómatas de Büchi, cuyos detalles se muestran en la sección 5.1, está especialmente pensado para ser utilizado en la verificación de sistemas concurrentes, siguiendo los pasos expuestos en el apartado 4.3.1. Esquema con los pasos a realizar, también puede emplearse, conjuntamente con el algoritmo presentado al final del apartado 4.3.2. Comprobación del no vacío de un autómata, para determinar la satisfacibilidad o la validez de una determinada especificación FIL. Por consiguiente, para estos fines sólo es necesario introducir la correspondiente especificación FIL, sin que haya de por medio ningún sistema que deba cumplir dicha especificación. Los siguientes apartados de esta sección presentan con más detalles el procedimiento a seguir en cada caso.

4.4.1. Satisfacibilidad de una especificación

Tal y como indica la Definición 2.7: Satisfacibilidad de una fórmula, una especificación φ es satisfacible si al menos existe una secuencia (dentro del universo de modelos que pueden construirse con el conjunto de proposiciones atómicas que la forman) que la satisfaga. Dicho de otro modo, φ es satisfacible si no contiene ninguna contradicción. Obviamente, una especificación insatisfacible constituye una propiedad inútil, que no debe ser utilizada, ya que ninguna ejecución la satisface.

Aplicando los dos algoritmos que hemos mencionado, los pasos a dar para comprobar la satisfacibilidad de la especificación FIL φ serían los siguientes:

1. Traducir la especificación φ al autómata de Büchi A_φ que es semánticamente equivalente, siguiendo el algoritmo descrito en la sección 5.1.

2. Comprobar el *no vacío* de dicho autómata, esto es, comprobar que el lenguaje
   $\mathscr{L}(A_\varphi)$ es no vacío, lo que significará que la especificación φ es satisfacible. Esto se lleva a cabo tal y como se ha explicado al final del apartado 4.3.2.

Por consiguiente, estamos reduciendo el problema de la satisfacibilidad de una especificación al de la comprobación del no vacío de su correspondiente autómata de propiedad.

4.4.2. Validez de una especificación

Recordemos que, de acuerdo con la Definición 2.8: Validez de una fórmula, una especificación FIL es válida si y sólo si se cumple para cada modelo del universo de modelos considerado; en este caso, todos aquellos que pueden construirse con el conjunto de proposiciones atómicas que integran esa especificación.

Este tipo de análisis (comprobar la validez de una especificación) puede resultar de gran utilidad, por ejemplo, si estamos interesados en comprobar si una determinada especificación φ es más fuerte (restringe más el número de ejecuciones permitidas) que otra especificación ψ. Esto se puede averiguar fácilmente, comprobando si la implicación $\varphi \Rightarrow \psi$ es válida, en cuyo caso significaría que si un sistema satisface φ, entonces también satisface ψ.

Como ya se comentó al final del apartado 2.4.4. Semántica formal, una especificación φ es válida si y sólo si $\neg\varphi$ es no satisfacible. Por consiguiente, se puede reducir el problema de comprobar la validez de una especificación φ al problema de comprobar la no satisfacibilidad de su negación, esto es, de $\neg\varphi$. Así, adaptamos el procedimiento presentado en el apartado anterior del siguiente modo, con el fin de comprobar la validez de la especificación FIL φ:

1. Traducir la negación de la especificación, $\neg\varphi$, al autómata de Büchi que es semánticamente equivalente, denotado como $A_{\neg\varphi}$.
2. Comprobar el *vacío* de dicho autómata. El lenguaje $\mathscr{L}(A_{\neg\varphi})$ contiene exactamente el conjunto de ejecuciones que satisfacen la propiedad $\neg\varphi$. Si este lenguaje es vacío, entonces $\neg\varphi$ es no satisfacible, por lo que φ es válida.

ÍNDICE DE REFERENCIAS EMPLEADAS

[Aggarw83], *79*
[Bérard01], *78*
[Büchi62], *67*
[Clarke99], *78*
[Courco92], *82*, *84*
[Dieste00], *81*
[Gerth95], *82*, *84*
[Gibbon85], *81*
[Gribom89], *69*
[Holzma91], *81*
[Holzma96a], *82*
[Holzma96b], *82*
[Hopcro79], *81*
[Hornos01b], *78*
[Jard89], *84*
[Jard91], *82*
[Kursha94], *79*, *81*
[Ramakr93], *75*
[Ramakr96], *75*
[Sistla87], *81*
[Tarjan72], *82*
[Thomas90], *67*, *72*
[Thomas97], *69*
[Vardi86], *79*
[Wolper95], *78*

Capítulo 5

Generación Automática del Autómata de Propiedad

Contenido

RESUMEN Y ORGANIZACIÓN

Éste es el capítulo central y más importante de esta tesis, dado que en él se exponen las principales aportaciones de nuestro trabajo. Así, a lo largo de la primera sección se describe detalladamente el algoritmo que hemos desarrollado para traducir una fórmula FIL a un autómata de Büchi semánticamente equivalente, centrándonos especialmente en las reglas de expansión de las fórmulas de intervalo, que constituyen el núcleo y principal novedad de nuestro algoritmo. Según el procesamiento a aplicar, el algoritmo distingue entre distintos tipos de fórmulas, a las que hemos dado nombre, clasificado y definido. Otra parte importante del algoritmo, y en la que se aportan interesantes ideas, es la de cómo determinar adecuadamente los estados de aceptación del autómata generado. También se indica cómo se puede convertir un autómata de Büchi con varios conjuntos de estados de aceptación (de los que genera el algoritmo presentado) en uno equivalente, pero con un único conjunto de estados de aceptación. Por último, en la segunda sección se demuestra formalmente que el algoritmo presentado en la sección anterior es correcto.

5.1. Traducción de fórmulas FIL a autómatas de Büchi

Para generar el autómata de propiedad correspondiente a una especificación φ, o sea, aquél que es semánticamente equivalente a la fórmula FIL que constituye dicha especificación, hemos desarrollado un algoritmo [Hornos01a] [Hornos02] en el que se pueden distinguir dos fases:

- En la primera de ellas, basada en el método *tableau* [Wolper85], se crea un grafo que define los estados y las transiciones del autómata, cuyos nodos se etiquetan con conjuntos de fórmulas obtenidos al expandir sucesivamente (de forma recursiva) la especificación inicial φ y las "subfórmulas" derivadas a partir de ella, de acuerdo con su estructura booleana y con la semántica de FIL. Como ocurre en otros autómatas, generados con el método *tableau* para representar propiedades de vivacidad, las fórmulas que etiquetan sus estados separan lo que debe ser cierto en el instante actual de lo que debe satisfacerse desde el próximo estado en adelante. El resultado de esta primera fase en la construcción del autómata de propiedad es el conjunto formado por todos los nodos que constituyen su sistema de transición, donde cada nodo se almacena en un registro compuesto por varios campos.

- En la segunda fase se debe transformar el grafo construido en la fase anterior en el autómata de Büchi que acepte exactamente las mismas secuencias infinitas que satisfacen la especificación φ. Para ello, se deben imponer una serie de condiciones de aceptación, de cara a obtener los estados que las cumplen (estados de aceptación), en base a los cuales poder discernir si una determinada ejecución del autómata generado es de aceptación o no, esto es, si satisface o no la especificación φ a partir de la que se ha obtenido dicho autómata.

FIL es la primera lógica de intervalos en la que el método *tableau* se ha aplicado con éxito. En parte, la dificultad de aplicar este método a las lógicas de intervalos es consecuencia de la necesidad de codificar sucintamente la noción de ámbito temporal limitado en las ejecuciones del autómata. En efecto, la principal novedad de FIL, la noción de un contexto temporal limitado (establecido por un determinado intervalo), también resulta ser una fuente importante de dificultad al extender los métodos usuales para la construcción del autómata de propiedad. Esto es debido a que en el caso de fórmulas temporales anidadas dentro de algún

intervalo, el "ámbito o alcance" de dichas fórmulas no es el contexto global, sino que debe restringirse al contexto temporal establecido por ese intervalo. En nuestro caso, las fórmulas FIL que etiquetan los nodos (estados) del autómata que se construye codifican correctamente esa noción de alcance temporal acotado, tal y como se explicó en el apartado 4.2.1. Descripción general de la estrategia de construcción. Así, se puede decir, de una manera intuitiva, que las condiciones que el autómata debe comprobar son las expresadas por los conjuntos de fórmulas FIL que etiquetan sus nodos.

Nuestro algoritmo tiene sus raíces en el de [Gerth95], que traduce especificaciones formuladas en LTL a autómatas de Büchi generalizados (ver Definición 5.29), por lo que, al igual que éste, está pensado para ser integrado dentro de una herramienta de comprobación de modelos *on-the-fly*. Tal y como se explicó en el apartado 4.3.3. Comprobación de modelos *on-the-fly*, esto significa que es posible ir generando el autómata de propiedad simultáneamente con, y guiado por, la construcción del autómata del sistema. De este modo, se puede detectar que una propiedad no se cumple construyendo sólo una parte del espacio de estados de ambos autómatas, con lo que se contribuye a paliar el problema de la explosión de estados (descrito en el apartado 3.3.2) que afecta a este método de verificación.

5.1.1. Tipos de fórmulas que distingue el algoritmo

Dentro de FIL, vamos a distinguir los siguientes tipos de fórmulas, todas ellas reconocidas automáticamente por nuestro algoritmo; así, éste puede diferenciar su análisis y procesar adecuadamente las fórmulas de cada tipo. A continuación, y a modo de presentación, se muestra un esquema donde se clasifican los tipos de fórmulas considerados; en él se puede apreciar que éstos se agrupan en dos grandes tipos, distinguiendo a su vez distintos subtipos dentro de ellos. Obsérvese que se establecen acrónimos (entre paréntesis) para aquellas expresiones que, siendo un poco largas, se van a utilizar con frecuencia.

- **Fórmulas proposicionales**
  - Fórmulas Puramente Proposicionales (FPPs)
    - Constantes lógicas
    - Literales

- Fórmulas Proposicionales Mixtas (FPMs)

- **Fórmulas de intervalo**
  - Fórmulas de Intervalo No Actuales (FINAs)
    - Fórmulas de eventualidad (FEs)
  - Fórmulas de Intervalo Actuales (FIAs)
    - FPPs anidadas a una Secuencia de Modalidades de Intervalo Actuales (SMIA)
    - FPMs anidadas a una SMIA
    - FINAs anidadas a una SMIA
    - FEs anidadas a una SMIA
    - Fórmulas Fallo de Búsqueda (FFBs) anidadas a una SMIA

Seguidamente se presentan las definiciones de cada uno de los tipos de fórmulas mencionados, con el fin de concretar exactamente su significado y su estructura. Aunque algunos de estos tipos ya se definieron en el Capítulo 2, sus definiciones se recogen aquí también para que este apartado sea completo, es decir, muestre todos los tipos de fórmulas considerados. Las FEs se utilizarán (en el apartado 5.1.6) para determinar los estados de aceptación del autómata que se pretende generar, mientras que el resto de los tipos serán usados por el algoritmo de construcción del grafo (ver apartado 5.1.3), y más concretamente, por sus reglas de expansión (ver apartado 5.1.4).

Definición 5.1: Fórmula proposicional

Una *fórmula proposicional* es una fórmula que o bien pertenece a la Lógica Proposicional (o sea, es una FPP) o bien su operador principal pertenece a dicha lógica (es decir, es una FPM). ■

Definición 5.2: Fórmula puramente proposicional (FPP)

Una *fórmula puramente proposicional* (FPP) es una constante lógica, un literal o una combinación de dichas fórmulas mediante los siguientes operadores de la

Lógica Proposicional (que son los reconocidos por nuestro algoritmo): *conjunción*, *disyunción*, *implicación*, *equivalencia* o *disyunción exclusiva* (*xor*). También se puede definir como aquella fórmula FIL que no contiene ninguna modalidad de intervalo. ■

Definición 5.3: Constante lógica

Una *constante lógica* es una fórmula cuya interpretación no depende del modelo sobre el que se evalúe, es decir, que siempre tiene la misma interpretación. Como es sabido, sólo existen dos valores lógicos o constantes lógicas, que son el valor de verdad (*true* en inglés) y el de falsedad (*false* en inglés), por lo que (tal y como se dijo en el apartado 2.4.1. Sintaxis) representaremos las fórmulas que hacen referencia a dichos valores mediante **T** y **F** respectivamente. ■

Definición 5.4: Literal

Se denomina *literal* a una proposición atómica, negada o no. ■

Definición 5.5: Fórmula proposicional mixta (FPM)

Una *fórmula proposicional mixta* (FPM) es una fórmula cuyo operador principal pertenece a la Lógica Proposicional clásica, pudiendo ser (en nuestra implementación) uno de los siguientes: *conjunción*, *disyunción*, *implicación*, *equivalencia* o *disyunción exclusiva* (*xor*), y en la que al menos uno de sus operandos contiene alguna fórmula de intervalo. ■

Definición 5.6: Fórmula de intervalo

Una *fórmula de intervalo* es aquella que define un intervalo I que establece el contexto en el que se debe satisfacer la fórmula f anidada al mismo, por lo que su estructura se puede representar como $I\,f$. También puede definirse como aquella fórmula cuyo operador principal es el operador de intervalo. ■

Antes de continuar con las definiciones de los distintos subtipos de fórmulas de intervalo que se han presentado en el esquema del principio de este apartado,

debemos definir algunos conceptos previos que necesitaremos para ello y que son los siguientes:

Definición 5.7: Modalidad de intervalo actual (MIA)

Se denomina *modalidad de intervalo actual* (MIA) a un intervalo del tipo $[-|\theta_2)$, es decir, su primer patrón de búsqueda es el trivial, mientras que el segundo es un patrón de búsqueda no trivial. Por tanto, define un intervalo que intenta construir un prefijo del contexto actual, empezando en el instante actual (de ahí su nombre) y extendiéndose hasta el estado localizado por θ_2, pero sin incluirlo. ■

Definición 5.8: Intervalo más externo de una fórmula de intervalo

El *intervalo más externo de una fórmula de intervalo* es aquél que está situado más a la izquierda en su representación en FIL y más arriba en su representación en GIL. ■

Ahora ya podemos definir los dos subtipos principales de fórmulas de intervalo:

Definición 5.9: Fórmula de intervalo no actual (FINA)

Una *fórmula de intervalo no actual* (FINA) es aquella cuyo intervalo más externo no es una MIA, esto es, su estructura viene dada por $[\theta_1|\theta_2)\psi$, donde θ_1 es un patrón de búsqueda no trivial, θ_2 es cualquier patrón de búsqueda (incluido el trivial) y ψ representa a cualquier fórmula FIL. ■

Definición 5.10: Fórmula de intervalo actual (FIA)

Una *fórmula de intervalo actual* (FIA) es aquella cuyo intervalo más externo es una MIA, por lo que su estructura viene dada por $[-|\theta_2)\psi$, donde θ_2 es un patrón de búsqueda no trivial y ψ representa a cualquier fórmula FIL. ■

Un tipo especial de fórmulas dentro de las FINAs lo constituyen las fórmulas de eventualidad, cuya importancia y utilidad se va a poner de manifiesto en la segunda fase de la construcción del autómata, a diferencia del resto de tipos

mencionados, que se utilizarán en la primera fase de dicha construcción. Este tipo de fórmulas se define como sigue:

Definición 5.11: Fórmula de eventualidad (FE)

Una *fórmula de eventualidad* (FE) es una fórmula de intervalo del tipo $\neg[\theta_1 \mid \rightarrow)\mathbf{F}$, donde θ_1 es un patrón de búsqueda no trivial, denominada así porque expresa que dicho patrón de búsqueda se debe cumplir obligatoriamente (en el futuro reflexivo o no estricto). ■

De nuevo necesitamos definir algunos conceptos previos antes de poder proseguir con las definiciones de los subtipos de FIAs:

Definición 5.12: Secuencia de modalidades de intervalo actuales (SMIA)

Denominaremos *secuencia de modalidades de intervalo actuales* (SMIA) y la denotaremos como $\mathcal{I}$, a una secuencia de uno o más intervalos formada exclusivamente por MIAs. Por tanto, su estructura vendrá dada por $\mathcal{I} = [- \mid \theta_1)[- \mid \theta_2)\ldots[- \mid \theta_n)$, donde cada θ_i es un patrón de búsqueda no trivial. ■

Definición 5.13: Prefijo de una FIA

Se denomina *prefijo de una FIA* a la SMIA por la que empieza esa fórmula. Así, si la estructura de una FIA, η, se representa como $\mathcal{I}\psi$, donde ψ es cualquier fórmula FIL, a excepción de una FIA, su prefijo será la SMIA $\mathcal{I}$, cuyos intervalos son los más externos de η. ■

Una vez definidos estos dos últimos términos, continuamos con las definiciones de los subtipos de FIAs:

Definición 5.14: FPP anidada a una SMIA

Una *FPP anidada a una SMIA* es una FIA que tiene anidada una FPP a la SMIA que constituye su prefijo, es decir, su estructura es del tipo $\mathcal{I}\psi$, donde ψ es una FPP. Por tanto, ψ ha de cumplirse en el instante actual. ■

Definición 5.15: FPM anidada a una SMIA

Una *FPM anidada a una SMIA* es una FIA que tiene una FPM anidada a la SMIA que constituye su prefijo, es decir, su estructura es del tipo $\mathcal{I}\psi$, donde ψ es una FPM. ■

Definición 5.16: FINA anidada a una SMIA

Una *FINA anidada a una SMIA* es una FIA que tiene una FINA anidada a la SMIA que constituye su prefijo, esto es, su estructura viene dada por $\mathcal{I}[\theta_1 | \theta_2)\psi$, donde θ_1 es un patrón de búsqueda no trivial, θ_2 es cualquier patrón de búsqueda (incluido el trivial) y ψ representa a cualquier fórmula FIL. ■

Otros dos nuevos conceptos, que se definen a continuación, son necesarios antes de abordar las últimas definiciones de los tipos que restan:

Definición 5.17: Estructura absoluta de una fórmula de intervalo

Sea η una fórmula de intervalo, entonces se denomina *estructura absoluta de* η, y se denota como $|\eta|$, a la fórmula obtenida a partir de η cuando se eliminan todos los operadores de negación que afectan directamente a (colocados inmediatamente delante de) los distintos operadores de intervalos que componen dicha fórmula. ■

Definición 5.18: Número de intervalos negados de una fórmula de intervalo

Sea η una fórmula de intervalo, entonces se denomina *número de intervalos negados de* η, y se denota como $\lceil\eta\rceil$, al número de operadores de negación que afectan directamente a (colocados inmediatamente delante de) los distintos operadores de intervalos que componen η. ■

Finalmente definiremos los dos últimos subtipos de FIAs que nos quedan:

Definición 5.19: Fórmula Fallo de Búsqueda (FFB) anidada a una SMIA

Una *fórmula fallo de búsqueda* (FFB) *anidada a una SMIA* es una FIA, η, cuya estructura absoluta viene dada por $\mathcal{I}[\theta_1 | \rightarrow)\mathbf{F}$, donde θ_1 es un patrón de búsqueda

no trivial, y donde $\lceil \eta \rceil$ (número de intervalos negados de η) es par. Obsérvese que si θ_1 sólo contiene una búsqueda, entonces se puede decir que η es una *fórmula invariante anidada a una SMIA*. ■

Definición 5.20: FE anidada a una SMIA

Una *FE anidada a una SMIA* es una FIA, η, cuya estructura absoluta viene dada por $\mathcal{J}[\theta_1 | \rightarrow)\mathbf{F}$, donde θ_1 es un patrón de búsqueda no trivial y donde el número de intervalos negados de η, $\lceil \eta \rceil$, es impar. ■

5.1.2. Estructura de un nodo

A continuación, y antes de describir en el siguiente apartado el algoritmo de construcción del grafo correspondiente a una fórmula FIL, se presenta la estructura de datos empleada en el mismo para representar cada uno de los nodos del grafo. Dado que, como ya se ha dicho, nuestro algoritmo tiene sus orígenes en el de [Gerth95], vamos a respetar tanto la estructura donde se almacena cada nodo (registro compuesto por varios campos) como los nombres con los que allí se designan a cada uno de los campos utilizados, debido este último a que son cortos y suficientemente significativos.

La estructura de datos devuelta por el algoritmo de construcción del grafo consiste en una lista de nodos, denominada *GraphNodes*, que codifica tanto los nodos como las transiciones (o sea, el grafo o sistema de transición) del autómata de Büchi que es semánticamente equivalente a la especificación (fórmula suministrada como entrada al algoritmo) que se pretende verificar. Cada uno de los nodos del grafo se representa internamente mediante un registro que contiene los cinco campos siguientes:

- *Name* Número entero (exclusivo) que constituye el identificador de ese nodo. Cada vez que se crea un nuevo nodo se genera automática y consecutivamente (a partir del 1) un número entero que se asigna a este campo y que será su identificador. El 0 se reserva para identificar al nodo inicial, por lo que no puede ser asignado a ningún nodo creado por el algoritmo.

- *Incoming* Conjunto de identificadores de nodos que tienen un arco apuntando a este nodo, por lo que este campo representa el conjunto de los arcos entrantes al nodo. Dicho de otro modo, este campo almacena la lista de nodos que preceden (inmediatamente) al nodo en cuestión. Dado que el identificador 0 no se corresponde con ningún nodo propiamente dicho (o sea, generado por el algoritmo), cuando ese identificador forma parte de este campo en algún nodo, se puede considerar que más que representar un arco real indica que ese nodo es un nodo inicial.

- *New* Conjunto de propiedades temporales (fórmulas FIL) que se deben cumplir en el nodo y que aún no han sido procesadas. Como es obvio, cuando un nodo se ha procesado completamente, este conjunto se queda vacío.

- *Old* Conjunto de propiedades que se deben cumplir en el nodo y que ya han sido procesadas. Todas las fórmulas FIL que finalmente constituyen el conjunto contenido en este campo para un determinado nodo, han formado parte del campo *New* del mismo nodo en algún momento previo durante la ejecución del algoritmo.

- *Next* Propiedades temporales que se deben cumplir en todos los nodos que suceden inmediatamente a éste. Más concretamente, en cada momento de la ejecución del algoritmo, este campo contiene el conjunto de fórmulas FIL que deben cumplirse en todos los estados que son sucesores inmediatos de los estados que satisfacen las propiedades incluidas en el campo *Old* del mismo nodo.

5.1.3. Algoritmo de construcción del grafo

La figuras que se exponen a continuación muestran un esquema en pseudocódigo del algoritmo de construcción del grafo del autómata de Büchi que es semánticamente equivalente a una fórmula FIL (la que se introduce como entrada al mismo). Lo primero que haremos será explicar la notación empleada.

Notación: Dado que las "palabras reservadas" de todas las estructuras de control (por ejemplo: if … then … else …, for each … do …, etc.) que aparecen en este tipo

de pseudocódigo se suelen poner en inglés (es un estándar de hecho) y que todas las variables y funciones que aparecen en las siguientes figuras también se han denominado en inglés, éste será el idioma en el que se escribirá todo su contenido, a excepción de los comentarios, que se presentarán en español, en un tipo de letra de menor tamaño, alineados a la derecha y precedidos del símbolo #. Con ello se consigue homogeneizar la presentación de dichas figuras, sin mezclar varios idiomas dentro del propio pseudocódigo, haciendo más fácil su lectura. Los nombres de las funciones se presentan en **negrita** y con su inicial en minúscula, mientras que el resto de nombres de variables se muestran en *Cursiva* y con su inicial en mayúscula. Cuando un nombre, ya sea de función o de cualquier otra variable, está compuesto por varias palabras, todas se yuxtaponen sin ningún símbolo entre ellas, poniendo en mayúscula la inicial de cada palabra sucesiva para facilitar su lectura; ejemplos de esto último son los nombres **newName** y *CurrentNode*. Para hacer referencia a un determinado campo de un nodo concreto se pondrá el nombre del campo y entre paréntesis el del nodo, como por ejemplo en *New*(*CurrentNode*). Cuando el contenido de un nodo (registro) se muestra explícitamente, éste se encierra entre corchetes. El símbolo ← representa una asignación, ya sea a nivel de campo de un registro o a cualquier otro tipo de variable.

```
1  function createGraph(φ)
2      GraphNodes ← ∅;
3      FirstNode ← [Name←1, Incoming←{0}, New←{φ}, Old←∅, Next←∅];
4      expand(FirstNode, GraphNodes);
```

Figura 5.1. Función que construye el grafo del autómata de Büchi equivalente a una fórmula FIL

Como puede apreciarse en la Figura 5.1, la fórmula φ a partir de la que se desea obtener el correspondiente autómata de Büchi se pasa como parámetro a la función **createGraph**, que es la que inicializa el grafo a vacío (línea 2) y crea el primer nodo (tal y como se indica en la línea 3), llamando para el mismo (en la línea 4) a la función **expand** (ver Figura 5.2). Esta última función es la que lleva a cabo casi todo el trabajo de generación del grafo, dado que expande recursivamente los nodos que va produciendo a partir del inicialmente creado (denominado *FirstNode* en la Figura 5.1), analizando para ello una a una las fórmulas que componen cada nodo. Más concretamente, su misión consiste en expandir el nodo actual (denominado

CurrentNode en la Figura 5.2) con arreglo a las fórmulas que contiene, para lo que analiza individualmente cada una de dichas fórmulas, realizando, como consecuencia de cada uno de esos análisis, una de las siguientes acciones:

- Se actualizan los campos de fórmulas del nodo actual.
- El nodo es dividido en varios, representando cada uno de los nuevos nodos generados las distintas posibles alternativas de que las fórmulas del nodo actual se satisfagan.
- El nodo se descarta, debido a que contiene una contradicción.

El orden de expansión viene dado por la estrategia de *búsqueda primero en profundidad* (BPP). Por lo tanto, nuestro algoritmo expande el primer nodo generado en cada paso, hasta que llega a un nodo completamente procesado, esto es, su campo *New* está vacío (lo que se comprueba en la línea 2 de la Figura 5.2). Entonces hay dos posibilidades (dependiendo del resultado de la comprobación efectuada en las líneas 3-4):

(1) En el conjunto *GraphNodes* ya hay un nodo *N* que encaja con *CurrentNode*, lo que significa que ambos nodos contienen el mismo conjunto de literales en su campo *Old* (determinado por la función **literals** en la Figura 5.2) y el mismo conjunto de fórmulas en el campo *Next*. Por tanto, el campo *Incoming* de *N* debe actualizarse (como se indica en la línea 5), lo que equivale a añadir un arco entrante a *N* y, en definitiva, a *agregar un nuevo arco al grafo*. Además, cualquier fórmula procesada en *CurrentNode* que no esté previamente almacenada en *Old*(*N*) debe ser añadida a dicho campo (línea 6), con el fin de que éste contenga el conjunto maximal de fórmulas que *N* satisface. Una vez hecho esto, el *backtracking* selecciona (siguiendo el orden de la BPP) un nodo de entre los aún no explorados, para procesarlo a continuación.

(2) No existe ningún nodo en el conjunto *GraphNodes* que encaje con el nodo actual, por lo que *CurrentNode* debe incluirse en dicho conjunto (línea 7), lo que equivale a *añadir un nuevo nodo al grafo*. Además, se crea un nuevo nodo, denominado *SuccessorNode*, que es inmediatamente expandido (líneas 8-10). Obsérvese que el nodo creado es sucesor del actual (se introduce el identificador de *CurrentNode* en su campo *Incoming*), debiendo procesar el conjunto de fórmulas que hay en *Next*(*CurrentNode*) (es lo que se asigna a su

```
1  function expand(CurrentNode, GraphNodes)
2    if  New(CurrentNode) = ∅
3    then if ∃ N∈GraphNodes with literals(N)=literals(CurrentNode) and
4                    Next(N)=Next(CurrentNode)
5         then   Add Incoming(CurrentNode) to Incoming(N);    # Se añade un nuevo arco al grafo
6                Add Old(CurrentNode)–Old(N) to Old(N);
7         else   Add CurrentNode to GraphNodes;               # Se añade un nuevo nodo al grafo
8                SuccessorNode ← [Name←newName(), Incoming←{Name(CurrentNode)},
9                    New←Next(CurrentNode), Old← ∅, Next← ∅];
10               expand(SuccessorNode, GraphNodes);
11   else  Select a formula η from New(CurrentNode) and remove it from this field;
12        if  η=F  or  ¬η∈Old(CurrentNode)                     # F es la constante lógica "false"
13        then Discard CurrentNode;                            # El nodo es insatisfacible
14        else  case η of
15             η=T  or  η is a literal:                        # T es la constante lógica "true"
16                if η≠T  then  Add η to Old(CurrentNode);
17                expand(CurrentNode, GraphNodes);
18             η is a formula of either Tabla 5.1 or Tabla 5.2:
19                if new_n (η) is not empty then
20                   Create NewNode from CurrentNode, adding new_n(η) to New(NewNode)
21                        and  η to Old(NewNode);
22                   expand(NewNode, GraphNodes);
23                Add new_c(η) to New(CurrentNode)  and  η  to  Old(CurrentNode);
24                expand(CurrentNode, GraphNodes);
25             default:     # η es cualquier fórmula de intervalo, excepto una FPM anidada a una SMIA
26                ReductionPairs ← reduction(η);
27                for each pair (Reductor, Reduct)∈ReductionPairs do
28                   if Reduct is neither of type 𝒥F nor unsatisfiable then      # 𝒥 es una SMIA
29                      Create NewNode from CurrentNode, adding {Reductor, Reduct} to
30                           New(NewNode)  and  η to Old(NewNode);
31                      expand(NewNode, GraphNodes);
32                for each pair (Reductor, Reduct)∈ReductionPairs do
33                   Add ¬Reductor  to New(CurrentNode);
34                case |η| of
35                   |η| = [θ1 | θ2)ψ, where θ1 ≠ –:                          # η es una FINA
36                      Add η to Next(CurrentNode);
37                   |η| = 𝒥ψ, and ψ is purely propositional:   # η es una FPP anidada a una SMIA
38                      Add ψ (conveniently signed) to New(CurrentNode);
39                   |η| = 𝒥[θ1 | →)F:                  # η es una FFB o una FE anidada a una SMIA
40                      Add ⟨η⟩ to Next(CurrentNode);
41                   |η| = 𝒥[θ1 | θ2)ψ, with θ1 ≠ –:          # η es una FINA anidada a una SMIA
42                      Add η to Next(CurrentNode);
43                      if isNegated([θ1 | θ2)ψ) then Add ¬|𝒥F| to Next(CurrentNode);
44                expand(CurrentNode, GraphNodes);
```

Figura 5.2. Proceso de expansión de los nodos del grafo

campo *New*). Obviamente, la función **newName** es la que genera el identificador para ese nodo.

Si *CurrentNode* no ha sido completamente procesado, entonces se elige (en la línea 11) una fórmula, que denotaremos como η, de su campo *New* (de donde se elimina en ese mismo instante) para analizarla a continuación. Así, en primer lugar se comprueba (línea 12) si η es la constante lógica **F** o si su negación se encuentra almacenada en el campo *Old* del nodo analizado, en cuyo caso el nodo debe descartarse (línea 13), debido a que contiene una contradicción, por lo que es insatisfacible. En caso contrario, η se analiza para determinar, con arreglo a su estructura sintáctica, cuál de las distintas reglas de expansión (líneas 14-44) se debe aplicar, con el fin de actualizar el nodo analizado, *CurrentNode*, o dividirlo en varios, que a su vez se tendrán que expandir, por supuesto, siguiendo el orden establecido por la estrategia de BPP empleada. Estas reglas de expansión se explican detalladamente en el siguiente apartado. Cuando todo el proceso de expansión acaba, *GraphNodes* contiene el conjunto de nodos y transiciones que constituyen el grafo que el algoritmo ha generado a partir de la fórmula FIL φ.

5.1.4. Reglas de expansión

Se puede decir que las reglas de expansión, también denominadas reglas *tableau*, son aquellas que definen cómo se expande o descompone cada tipo de fórmula en las distintas alternativas posibles para que dicha fórmula se satisfaga, y en el caso de fórmulas temporales, separando además lo que se tiene que cumplir en el instante actual de lo que tiene que satisfacerse desde el siguiente instante en adelante.

El contenido de este apartado se va a presentar dividido en dos subapartados, atendiendo al tipo de fórmulas sobre los que se aplican las correspondientes reglas. Así:

- Las reglas de expansión para las *fórmulas proposicionales* se presentan brevemente en el primero de ellos, pues son bien conocidas, ya que son las que se emplean en la LTL (por ejemplo, en [Gerth95] o [Daniel99]) para este tipo de fórmulas.
- Las reglas de expansión para las *fórmulas de intervalo* se explican más extensamente en el segundo subapartado, pues constituyen la principal

novedad y aportación de nuestro trabajo a la hora de construir automáticamente un autómata equivalente a una fórmula FIL.

5.1.4.1. PARA FÓRMULAS PROPOSICIONALES

Los casos más básicos y simples se dan cuando la fórmula analizada, η, es la constante lógica **T** o es un literal (líneas 15-17 de la Figura 5.2). Así, sólo hay que actualizar *CurrentNode*, añadiendo η a su campo *Old* (en el primer caso, ni siquiera esto es necesario) y expandir a continuación el nodo resultante.

El resto de fórmulas de este tipo son aquellas cuyo operador principal (binario) pertenece a la Lógica Proposicional clásica y se expanden conforme a lo indicado en las líneas 18-24 de la Figura 5.2. Los operadores de esta clase que reconoce nuestro algoritmo son los que se recogen en la Tabla 5.1, cuyas filas contienen las reglas de expansión para la *conjunción*, *disyunción*, *implicación*, *equivalencia* y *disyunción exclusiva* (*xor*), en ese orden. En ella, tanto μ como ψ representan a cualquier fórmula FIL. La tabla debe interpretarse del siguiente modo: η (columna de la izquierda) es la fórmula del nodo actual elegida en un momento determinado por el algoritmo para analizarla y expandirla, de modo que la segunda columna, $new_c(\eta)$, representa el conjunto de fórmulas que se añade al campo *New* de *CurrentNode*, mientras que la última columna, $new_n(\eta)$, representa el conjunto de fórmulas que se añade al campo *New* de un nuevo nodo (denominado *NewNode* en la Figura 5.2) que se crea a partir del actual.

Tabla 5.1. Reglas *tableau* para fórmulas cuyo operador principal pertenece a la Lógica Proposicional

η	$new_c(\eta)$	$new_n(\eta)$
$\mu \wedge \psi$	$\{\mu, \psi\}$	—
$\mu \vee \psi$	$\{\mu\}$	$\{\psi\}$
$\mu \Rightarrow \psi$	$\{\neg\mu\}$	$\{\psi\}$
$\mu \equiv \psi$	$\{\mu, \psi\}$	$\{\neg\mu, \neg\psi\}$
$\mu \oplus \psi$	$\{\mu, \neg\psi\}$	$\{\neg\mu, \psi\}$

Con el fin de evitar volver a introducir fórmulas ya analizadas en un nodo, tanto para este tipo de fórmulas como para todas las que se van a estudiar a continuación, antes de insertar cualquier fórmula en el campo *New* de un nodo, siempre se comprueba que la fórmula a insertar no esté ya almacenada en el campo

Old de ese nodo. Si lo está, significa que esa fórmula ya se ha analizado en ese nodo, por lo que no se tendría que introducir de nuevo en su campo *New*. Esta heurística, que evita tener que procesar varias veces la misma fórmula en el mismo nodo, no se refleja explícitamente en la Figura 5.2 , con el fin de simplificarla, de modo que en ella sólo se muestran las partes esenciales y especialmente relevantes del algoritmo de construcción del grafo.

5.1.4.2. PARA FÓRMULAS DE INTERVALO

Dentro de las fórmulas de intervalo, vamos a tratar separadamente un tipo concreto de ellas: las FPMs anidadas a una SMIA, cuyas reglas de expansión son muy parecidas a las que se han visto en el apartado anterior para las fórmulas proposicionales. Para el resto de fórmulas de intervalo necesitaremos obtener los reductores de la fórmula analizada en el proceso de expansión de las mismas.

A) FPM (Fórmula Proposicional Mixta) anidada a una SMIA (Secuencia de Modalidades de Intervalo Actuales)

De acuerdo con la Definición 5.15, estas fórmulas están formadas por una SMIA $\mathcal{I}$, que constituye el prefijo al que se anida una FPM. Por tanto, en la Tabla 5.2 al menos una de las subfórmulas, μ o ψ, debe contener alguna modalidad de intervalo. En caso contrario, la fórmula anidada a $\mathcal{I}$ no sería una FPM, sino una FPP, y η sería, en consecuencia, una FPP anidada a una SMIA, por lo que no se aplicarían las reglas de dicha tabla, sino las que se explicarán en el siguiente subapartado para ese tipo de fórmulas. El resto de elementos de la tabla tiene el mismo significado que en la Tabla 5.1 (obsérvese el paralelismo existente entre ambas tablas).

Tabla 5.2. Reglas de expansión para las FPMs anidadas a una SMIA

η	$new_c(\eta)$	$new_n(\eta)$
$\mathcal{I}(\mu \wedge \psi)$	$\{\mathcal{I}\mu,\ \mathcal{I}\psi\}$	—
$\mathcal{I}(\mu \vee \psi)$	$\{\mathcal{I}\mu\}$	$\{\mathcal{I}\psi\}$
$\mathcal{I}(\mu \Rightarrow \psi)$	$\{\mathcal{I}\neg\mu\}$	$\{\mathcal{I}\psi\}$
$\mathcal{I}(\mu \equiv \psi)$	$\{\mathcal{I}\mu,\ \mathcal{I}\psi\}$	$\{\mathcal{I}\neg\mu,\ \mathcal{I}\neg\psi\}$
$\mathcal{I}(\mu \oplus \psi)$	$\{\mathcal{I}\mu,\ \mathcal{I}\neg\psi\}$	$\{\mathcal{I}\neg\mu,\ \mathcal{I}\psi\}$

Para concluir este apartado puede decirse que, al analizar una FPM anidada a una SMIA, el algoritmo divide el nodo actual en dos (excepto en el caso de la conjunción). Tal y como se muestra en la tabla anterior, la semántica del operador principal de la FPM anidada al prefijo $\mathcal{I}$ es la que establece cómo debe realizarse esa división y la distribución de las subfórmulas resultantes en los distintos nodos. A la hora de determinar adecuadamente el tipo de dicho operador y, por tanto, de aplicar la regla apropiada de la tabla anterior (líneas 18-24 de la Figura 5.2 , ya explicadas en el subapartado previo), obviamente hay que tener en cuenta los operadores de negación que afectan a (colocados inmediatamente delante de) los intervalos que componen la SMIA $\mathcal{I}$, dado que:

- la negación de un intervalo afecta a su fórmula anidada, negándola,
- los operadores de *conjunción* ($\wedge$) y *disyunción* ($\vee$) son duales,
- los operadores de *equivalencia* ($\equiv$) y *disyunción exclusiva* ($\oplus$) son opuestos,
- y la implicación $\mu \Rightarrow \psi$ es una abreviación de la fórmula $\neg\mu \vee \psi$.

B) Resto de fórmulas de intervalo

Las reglas de expansión que hemos desarrollado para este tipo de fórmulas se han derivado a partir de la relación de reducción (definida en el apartado 4.2.2. Reducción de las fórmulas de intervalo) y de la semántica de FIL (cuya formalización se muestra en el apartado 2.4.4. Semántica formal). En estas reglas podemos distinguir dos pasos a realizar, que son los explicados a continuación:

1) Creación de nuevos nodos

Este primer paso (líneas 26-31 de la Figura 5.2) se lleva a cabo para cualquier fórmula de intervalo, excepto para las ya analizadas en el subapartado A) anterior, siendo independiente del subtipo concreto de fórmula de intervalo de que se trate.

Para aplicarlo, primero se llama a la función **reduction** (cuyo pseudocódigo se muestra en la Figura 5.3), a la que se le pasa η como argumento, pues es la fórmula que se quiere reducir. Esta función, que implementa los conceptos formalizados en la Definición 4.9: Conjunto reductor de una fórmula de intervalo y en la Definición 4.10: Relación de reducción, devuelve un conjunto (denominado *ReductionPairs* en la Figura 5.2 y en la Figura 5.3) formado por todos los reductores posibles de η y sus

correspondientes reductos, o sea, por pares de fórmulas FIL de la forma (*Reductor*, *Reduct*). En la Figura 5.3 puede observarse que se ejecuta un fragmento de código distinto dependiendo exclusivamente de la estructura sintáctica del intervalo más externo de η; más concretamente, de que sea trivial su primer patrón de búsqueda (líneas 4-17) o su segundo patrón de búsqueda (líneas 18-27) o de que ninguno de ellos lo sea (líneas 28-46). Obsérvese además que sólo en el primero de estos casos, esta función analiza la fórmula anidada ψ (y sólo cuando ψ es a su vez una fórmula de intervalo), llamándose recursivamente a sí misma con ψ como parámetro (línea 14). En los otros dos casos, ψ no se analiza, es decir, para hallar todos los reductores (y sus reductos) de η sólo hay que analizar su intervalo más externo. El análisis para calcular los reductores de η no depende de que su intervalo más externo esté negado o no, motivo por el que (en la sentencia *case*) no se hace la comprobación sobre la propia η, sino sobre su estructura absoluta, $|\eta|$ (ver Definición 5.17: Estructura absoluta de una fórmula de intervalo). Para terminar de explicar el contenido de la Figura 5.3, describimos brevemente el significado de las funciones que en ella aparecen. Así, las funciones **head**, **tail** y **size**, que llevan como parámetro un patrón de búsqueda, θ, devuelven respectivamente la fórmula objetivo de su primera búsqueda, el patrón de búsqueda resultante de eliminar la primera búsqueda de θ y el número de búsquedas que tiene θ. La función booleana **isNegated** es cierta sólo si la fórmula sobre la que se aplica está negada, o sea, si su intervalo más externo lo está. Esta última función se emplea como condición en la expresión condicional del tipo *condición* ? *valor1* : *valor2* (tomada del lenguaje de programación C++ [Strous93]), que aparece en la parte derecha de ciertas asignaciones. De hecho, sólo se utiliza en las asignaciones a la variable *Reduct* y siempre empleando como *condición* la función **isNegated**(η), de modo que si η está negada, se le asignaría el *valor1* a dicha variable, y el *valor2* si no lo está. La razón por la que **isNegated**(η) sólo se emplea cada vez que se va a asignar un valor a la variable *Reduct* es porque sólo se debe tener en cuenta si el intervalo más externo de η está negado o no en el momento de calcular el reducto correspondiente a cada reductor, no influyendo para nada a la hora de determinar sus reductores.

Una vez calculado el conjunto *ReductionPairs*, se crea *un nuevo nodo por cada par* que éste contenga (líneas 27-30 de la Figura 5.2), siempre que el nodo resultante no haya sido ya generado y que el correspondiente reducto (segundo componente del par) sea satisfacible, para lo que basta comprobar que éste no sea:

```
1  function reduction (η)
2    ReductionPairs ← ∅;
3    case |η| of
4      |η| = [– | θ2) ψ:
5        Reductor ← head(θ2);
6        if size(θ2) > 1
7          then   Reduct ← isNegated(η) ? ¬[– | tail(θ2)) ψ : [– | tail(θ2)) ψ;
8          else   Reduct ← isNegated(η) ? F : T;
9        Add pair (Reductor, Reduct) to ReductionPairs;
10       Reductor ← [θ2 | →)F;
11       Reduct ← isNegated(η) ? F : T;
12       Add pair (Reductor, Reduct) to ReductionPairs;
13       if ψ is an interval formula then
14         for each pair (τ, γ) ∈ reduction (ψ)        # γ es el τ-reducto de ψ, o sea, ψ_τ≻γ
15           Reductor ← [– | θ2) τ;
16           Reduct ← isNegated(η) ? ¬[– | θ2) γ : [– | θ2) γ;
17           Add pair (Reductor, Reduct) to ReductionPairs;
18     |η| = [θ1 | →) ψ:
19       Reductor ← head(θ1);
20       if size(θ1) > 1
21         then   Reduct ← isNegated(η) ? ¬[tail(θ2) | →) ψ : [tail(θ2) | →) ψ;
22         else   Reduct ← isNegated(η) ? ¬ψ : ψ;
23       Add pair (Reductor, Reduct) to ReductionPairs;
24       if ψ ≠ F then
25         Reductor ← [θ1 | →)F;
26         Reduct ← isNegated(η) ? F : T;
27         Add pair (Reductor, Reduct) to ReductionPairs;
28     |η| = [θ1 | θ2) ψ:
29       μ ← head(θ1);  ν ← head(θ2);
30       if μ = ν
31         then if size(θ2) = 1
32                then Reduct ← isNegated(η) ? F : T;
33                else Reduct ← isNegated(η) ? ¬[tail(θ1) | tail(θ2)) ψ : [tail(θ1) | tail(θ2)) ψ;
34              Add pair (ν, Reduct) to ReductionPairs;
35         else if size(θ2) = 1
36                then Reduct ← isNegated(η) ? F : T;
37                else Reduct ← isNegated(η) ? ¬[θ1 | tail(θ2)) ψ : [θ1 | tail(θ2)) ψ;
38              Add pair (ν, Reduct) to ReductionPairs;
39              Reduct ← isNegated(η) ? ¬[tail(θ1) | θ2) ψ : [tail(θ1) | θ2) ψ;
40              Add pair (μ, Reduct) to ReductionPairs;
41       Reductor ← [θ1 | →)F;
42       Reduct ← isNegated(η) ? F : T;
43       Add pair (Reductor, Reduct) to ReductionPairs;
44       Reductor ← [θ2 | →)F;
45       Add pair (Reductor, Reduct) to ReductionPairs;
46   return (ReductionPairs);
```

Figura 5.3. Proceso de reducción de una fórmula de intervalo η

(1) Ni la constante lógica **F** (ver Ejemplo 5.1).

(2) Ni una SMIA $\mathcal{I}$ que tenga anidada la constante lógica **F** (ver Ejemplo 5.2).

(3) Ni una SMIA $\mathcal{I}$ que tenga anidada la constante lógica **T** y cuyo número de negaciones colocadas delante de sus intervalos sea impar (ver Ejemplo 5.3).

Ejemplo 5.1: Actuación del algoritmo cuando un reducto es la constante lógica **F**

Fórmula (η)	*ReductionPairs*
$[\rightarrow\mu\,\vert\rightarrow)\mathbf{F}$	$\{(\mu, \mathbf{F})\}$

Para esta fórmula η, la función **reduction** devuelve el conjunto *ReductionPairs* que se muestra, lo que significa que dicha fórmula sólo tiene un reductor (μ), donde μ representa a cualquier fórmula FIL. Al ser el reducto la fórmula **F**, no se crea un nodo para ese par (motivo por el que se presenta escrito en rojo), dado que cualquier nodo que contenga dicha fórmula, se debe descartar. Además, como para este ejemplo concreto no existen más pares en el conjunto *ReductionPairs*, no se crea ningún nodo en este paso.

Ejemplo 5.2: Actuación del algoritmo cuando un reducto es del tipo $\mathcal{I}$ **F**

Fórmula (η)	*ReductionPairs*
$[-\,\vert\rightarrow\nu)[\rightarrow\mu\,\vert\rightarrow)\mathbf{F}$	$\{(\nu, \mathbf{T}); ([\rightarrow\nu\vert\rightarrow)\mathbf{F}, \mathbf{T}); ([-\,\vert\rightarrow\nu)\mu, [-\,\vert\rightarrow\nu)\mathbf{F})\}$

En este ejemplo, como en todos los siguientes, tanto μ como ν representan a cualquier fórmula FIL. El último par del conjunto *ReductionPairs* tiene como reducto la fórmula $[-\,\vert\rightarrow\nu)\mathbf{F}$, que sólo se satisface cuando el intervalo $[-\,\vert\rightarrow\nu)$ no se puede construir, condición ya codificada por los dos primeros pares del conjunto, motivo por el que no se crea ningún nuevo nodo para ese último par. Obsérvese que cuando el reducto de algún par es **T**, éste se representa en rojo, debido a que, por motivos de eficiencia, no se introduce en el campo *New* del correspondiente nodo.

Ejemplo 5.3: Actuación del algoritmo cuando un reducto es del tipo $\mathcal{J}_{\mathbf{T}}$, donde $\lceil \mathcal{J}_{\mathbf{T}} \rceil$ es impar

Fórmula (η)	*ReductionPairs*
$\neg[-\mid\rightarrow\mu)[-\mid\rightarrow\nu)\mathbf{F}$	$\{(\mu, \mathbf{F})$; $([\rightarrow\mu\mid\rightarrow)\mathbf{F}, \mathbf{F})$; $([-\mid\rightarrow\mu)\nu, \neg[-\mid\rightarrow\mu)\mathbf{T})$; $([-\mid\rightarrow\mu)[\rightarrow\nu\mid\rightarrow)\mathbf{F}, \neg[-\mid\rightarrow\mu)\mathbf{T})\}$

Los dos últimos pares del conjunto *ReductionPairs* tienen como reducto la fórmula $\neg[-\mid\rightarrow\mu)\mathbf{T}$, que es del tipo mencionado en la condición (3) expuesta anteriormente. Esta fórmula expresa una contradicción (que el intervalo debe poder construirse y que **F** se tiene que cumplir dentro de él), por lo que no se crea ningún nodo para esos pares, ni para los dos primeros, al ser éstos del tipo explicado en el Ejemplo 5.1.

Cada nuevo nodo se crea a partir del actual (con su mismo contenido), añadiendo el conjunto {*Reductor*, *Reduct*} a su campo *New* y la fórmula η a su campo *Old* (líneas 29-30 de la Figura 5.2). El nodo resultante se expande inmediatamente (línea 31), como consecuencia de que se sigue la estrategia de BPP.

2) Actualización del nodo actual

Cada nodo creado en el paso anterior constituye una de las posibles alternativas para reducir la fórmula de intervalo analizada, ya que representa el cumplimiento de uno de sus reductores. Sin embargo, aún falta por considerar la última de esas alternativas: *que ninguno de sus reductores se satisfaga* en ese instante. Por lo tanto, el nodo actual debe ser actualizado para representar dicha alternativa. Así, lo primero que se hace es introducir en el campo *New* del nodo actual la negación de todos los reductores de la fórmula analizada (líneas 32-33 de la Figura 5.2). Esta primera acción de este segundo paso, al igual que el paso anterior, se realiza siempre, sea cual sea la fórmula de intervalo analizada (a excepción de las FPMs anidadas a una SMIA, para las que no se necesita llamar a la función **reduction**).

Ejemplo 5.4: Actualización del nodo actual con la negación de todos los reductores de η

Fórmula (η)	Se añade a *New*(*CurrentNode*)
$[\rightarrow\mu\mid\rightarrow\nu)\psi$	$\{\neg\mu, \neg\nu, \neg[\rightarrow\mu\mid\rightarrow)\mathbf{F}, \neg[\rightarrow\nu\mid\rightarrow)\mathbf{F}\}$

Puesto que el conjunto de reductores para esta fórmula es $\{\mu,\ \nu,\ [\rightarrow\mu|\rightarrow)\mathbf{F},\ [\rightarrow\nu|\rightarrow)\mathbf{F}\}$, en este paso, la primera acción a realizar es añadir el conjunto de fórmulas que se indica arriba al campo *New* del nodo que se está analizando. Obsérvese que se añade la negación de todos los reductores, con independencia de si para ese reductor se creó o no un nodo en el paso anterior.

La segunda y última acción a realizar dentro de este paso se explica a continuación, y depende del tipo específico de la fórmula analizada, que puede ser uno de los siguientes:

a) FINA (Fórmula de Intervalo No Actual)

En este caso, o sea, cuando la estructura de la fórmula analizada η es, según la Definición 5.9, del tipo $[\theta_1|\theta_2)\psi$, donde θ_1 es un patrón de búsqueda no trivial, la propia η se debe insertar en el campo *Next* del nodo actual (líneas 35-36 de la Figura 5.2).

Ejemplo 5.5: Última acción del algoritmo cuando η es una FINA

Fórmula (η)	Se añade a *Next*(*CurrentNode*)
$\neg[\rightarrow\mu\|\rightarrow\nu)\psi$	$\{\neg[\rightarrow\mu\|\rightarrow\nu)\psi\}$

Dado que este nodo representa los estados en los que la fórmula η no se reduce, dicha fórmula se deberá volver a analizar en todos los sucesores inmediatos del nodo actual, motivo por el que η se añade a su campo *Next*.

b) FPP (Fórmula Puramente Proposicional) anidada a una SMIA (Secuencia de Modalidades de Intervalo Actuales)

Como su propio nombre indica, y de acuerdo con la Definición 5.14, la estructura de una de estas fórmulas es del tipo $\mathcal{I}\psi$, donde ψ es una FPP. Por lo tanto, y dado que ψ se ha de cumplir en el instante actual, aquí sólo hay que introducir en el campo *New* del nodo actual la FPP que está anidada a la SMIA, pero negada o no, según corresponda, teniendo en cuenta no sólo el signo de la propia subfórmula, sino también el de todos los intervalos que componen la SMIA a la que se anida (líneas

37-38 de la Figura 5.2). Obsérvese que, a diferencia de lo que ocurre al analizar otros tipos de fórmulas de intervalo, aquí no se añade nada al campo *Next* del nodo actual.

Ejemplo 5.6: Última acción del algoritmo cuando η es una FPP anidada a una SMIA

Fórmula (η)	Se añade a *New*(*CurrentNode*)
$\neg[-\mid\rightarrow\mu)[-\mid\rightarrow\nu)(\mathtt{p0}\wedge\mathtt{p1})$	$\{\neg\mathtt{p0}\vee\neg\mathtt{p1}\}$

Tanto en este ejemplo como en los que se expongan a continuación, cuando en una fórmula aparezca una 'p' seguida por un número se estará haciendo referencia a una proposición atómica, siguiendo la notación empleada en la implementación que hemos realizado de nuestro algoritmo (ver apartado 6.2.1). Así, p0 y p1 representan a dos proposiciones atómicas en este ejemplo.

Tal como indica la semántica de FIL, el operador de negación del intervalo más externo de η afecta al resto de sus fórmulas anidadas; así, la fórmula que se debe añadir al campo *New* del nodo actual es la negación de la que realmente está anidada a la SMIA, o sea, $\neg(\mathtt{p0}\wedge\mathtt{p1})$, que es la fórmula que se debe cumplir en el instante actual. Por motivos de eficiencia, nuestra implementación siempre empuja las negaciones tan dentro como es posible en las fórmulas proposicionales, razón por la que la fórmula que realmente se añade es $\neg\mathtt{p0}\vee\neg\mathtt{p1}$, que es semánticamente equivalente a la anterior, dado que la conjunción y la disyunción son operadores duales.

c) FFB (Fórmula Fallo de Búsqueda) o FE (Fórmula de Eventualidad) anidada a una SMIA (Secuencia de Modalidades de Intervalo Actuales)

Si η se corresponde con uno de estos dos tipos de fórmulas, cuyas definiciones se recogen respectivamente en la Definición 5.20 y Definición 5.19, su estructura absoluta, $|\eta|$ (Definición 5.17), es la misma para ambos tipos de fórmulas, y viene dada por:

$$\mathcal{I}[\theta_1\mid\rightarrow)\mathbf{F}$$

donde $\mathcal{I}$ representa a la SMIA que constituye el prefijo de η al que se anida la última de sus subfórmulas, cuya estructura se muestra explícitamente, siendo θ_1 un patrón de búsqueda no trivial.

Tal y como se indicó en las definiciones mencionadas, si el número de operadores de negación (colocados inmediatamente delante) de los distintos intervalos que componen la fórmula η es par, entonces se dice que η es una FFB anidada a una SMIA; en caso contrario, se trata de una FE anidada a una SMIA.

A continuación se definen dos conceptos o tipos de fórmulas que surgen durante el proceso de expansión de una de estas propiedades, a los que hemos denominado *forma normal* de uno u otro tipo de fórmula:

Definición 5.21: Forma normal de una FFB anidada a una SMIA

Si η es una *FFB anidada a una SMIA*, su *forma normal*, denotada como $\langle\eta\rangle$, es una fórmula FIL que tiene la misma estructura que η, pero con todos sus intervalos sin negar. Dicho de otro modo, $\langle\eta\rangle$ coincide con la estructura absoluta de η, o sea, $\langle\eta\rangle = |\eta|$. ■

Definición 5.22: Forma normal de una FE anidada a una SMIA

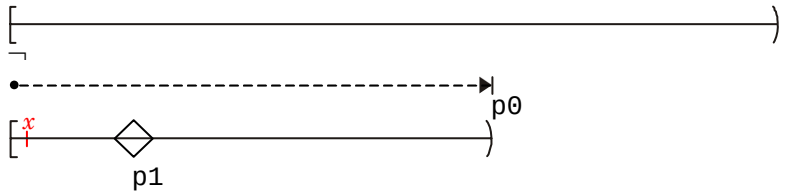
Si η es una *FE anidada a una SMIA*, su *forma normal*, $\langle\eta\rangle$, es una fórmula FIL que tiene la misma estructura que η, con su intervalo más externo negado y con el resto de sus intervalos (todos los internos) sin negar. Obsérvese que, en este caso, $\langle\eta\rangle$ se obtiene al negar $|\eta|$, es decir, $\langle\eta\rangle = \neg|\eta|$. ■

Ambos tipos de fórmulas tienen el mismo tratamiento (líneas 39-40 de la Figura 5.2), consistente en añadir su forma normal, $\langle\eta\rangle$, al campo *Next* del nodo actual. Los detalles de por qué se debe actuar así se ofrecen en los dos ejemplos siguientes:

Ejemplo 5.7: Última acción del algoritmo cuando η es una FFB anidada a una SMIA

Fórmula (η)	Se añade a *Next*(*CurrentNode*)
¬[–|→p0)¬[→p1|→)F	{[–|→p0)[→p1|→)F}

A continuación se incluye también la representación en GIL de esta fórmula, con el fin de que sea más fácil comprender lo que se va a explicar. En ella se añade una marca en rojo indicando cuál es el siguiente instante (x) al instante actual, que es el extremo izquierdo del intervalo.



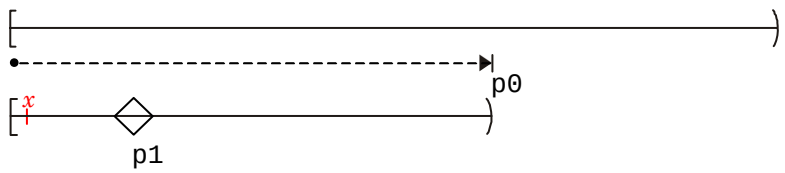
La interpretación semántica de la fórmula η impide que el literal `p0` se dé en el instante actual (ya que al estar negado su intervalo externo, éste no se puede colapsar), pero nada impide que se pueda dar en el resto de instantes futuros, desde el instante siguiente x (inclusive) en adelante. Si se introdujera la propia fórmula η en el campo *Next* del nodo actual, como se hace en el caso de las FINAs, entonces nunca en el futuro se podría cumplir `p0`, lo que es contrario a la semántica de η. El problema se resuelve introduciendo en su lugar su forma normal, $\langle\eta\rangle$, que para este ejemplo es la fórmula [–|→`p0`)[→`p1`|→)**F**, tal y como se indica arriba. Así, desde el siguiente estado en adelante ya se puede dar `p0`, cumpliéndose hasta entonces siempre ¬`p1`.

Conclusión: La *forma normal de una FFB anidada a una SMIA* permite que se pueda encontrar el extremo derecho de los intervalos que forman la SMIA $\mathcal{J}$ que encierran o constituyen el contexto de la FFB (anidada a dicha SMIA).

Ejemplo 5.8: Última acción del algoritmo cuando η es una FE anidada a una SMIA

Fórmula (η)	Se añade a *Next*(*CurrentNode*)
[–|→`p0`)¬[→`p1`|→)**F**	{¬[–|→`p0`)[→`p1`|→)**F**}

Al igual que en el ejemplo anterior, aquí también se incluye, persiguiendo el mismo objetivo, la representación en GIL de esta fórmula, donde la marca en rojo (x) que aparece también conserva el mismo significado.



En este caso, al no estar negado el intervalo más externo de η, `p0` se puede dar en el instante actual, con lo que la fórmula se satisfaría vacuamente. Si se incluyera η en el campo *Next*, eso mismo podría ocurrir en el instante x o en

cualquier de los que le siguen, con lo que no sería necesario que se encontrara (cumpliera) `p1` antes de `p0` para dar como válido lo que, según la semántica asociada a η, no lo es. De nuevo, esta situación se resuelve introduciendo en el campo *Next* su forma normal, $\langle\eta\rangle$, que para este ejemplo concreto es la fórmula $\neg[-|\rightarrow$`p0`$)[\rightarrow$`p1`$|\rightarrow)\mathbf{F}$. Con ello se impide que el extremo derecho del intervalo (`p0`) se pueda dar hasta que la fórmula de eventualidad anidada al mismo se haya satisfecho (o sea, se haya reducido).

Conclusión: La *forma normal de una FE anidada a una SMIA* impide que se reduzcan las MIAs que forman su prefijo $\mathcal{I}$ hasta que no se satisfaga (reduzca) la eventualidad anidada al mismo.

d) FINA (Fórmula de Intervalo No Actual) anidada a una SMIA (Secuencia de Modalidades de Intervalo Actuales)

La estructura de una fórmula η de este tipo, de acuerdo con la Definición 5.16, es la siguiente:

$$\mathcal{I}[\theta_1 \mid \theta_2)\psi$$

donde $\mathcal{I}$ representa a la SMIA que constituye el prefijo de η al que se anida la FINA cuya estructura se muestra explícitamente, siendo θ_1 un patrón de búsqueda no trivial, θ_2 cualquier patrón de búsqueda (incluido el trivial) y ψ cualquier fórmula FIL (incluso otra fórmula de intervalo con una secuencia de subfórmulas de intervalo anidada a ella).

Aquí hay dos posibles actuaciones distintas (líneas 41-43 de la Figura 5.2), dependiendo exclusivamente de si la FINA anidada a $\mathcal{I}$ está negada o no, no teniéndose en consideración para ello los operadores de negación que afecten a los intervalos que forman la SMIA $\mathcal{I}$. Así:

- Si $\eta = \mathcal{I}[\theta_1 \mid \theta_2)\psi$, entonces sólo hay que introducir la propia η en el campo *Next* del nodo actual (ver Ejemplo 5.9).
- Si $\eta = \mathcal{I}\neg[\theta_1 \mid \theta_2)\psi$, además de introducir la propia η en el campo *Next* del nodo actual, hay que meter también en dicho campo la fórmula que expresa que el prefijo $\mathcal{I}$ no puede colapsarse, es decir, $\neg|\mathcal{I}\mathbf{F}|$, donde todos

los intervalos internos de $\mathcal{I}$ no están negados, independientemente de que en η lo estuvieran o no (ver Ejemplo 5.10).

Ejemplo 5.9: Última acción del algoritmo cuando η tiene una FINA no negada anidada a una SMIA

Fórmula (η)	Se añade a *Next*(*CurrentNode*)
$[-\mid\rightarrow\mu)\neg[-\mid\rightarrow\nu)[\theta_1\mid\theta_2)\psi$	$\{[-\mid\rightarrow\mu)\neg[-\mid\rightarrow\nu)[\theta_1\mid\theta_2)\psi\}$

Al no estar negado el intervalo $[\theta_1\mid\theta_2)$, si el contexto $\mathcal{I}[\theta_1\mid\theta_2)$ (donde se tiene que cumplir ψ) se puede construir, será en el futuro reflexivo del instante siguiente (puesto que en el instante actual η no se reduce), motivo por el que η debe volver a analizarse en todos los sucesores inmediatos del nodo actual.

Ejemplo 5.10: Última acción del algoritmo cuando η tiene una FINA negada anidada a una SMIA

Fórmula (η)	Se añade a *Next*(*CurrentNode*)
$[-\mid\rightarrow\mu)\neg[-\mid\rightarrow\nu)\neg[\theta_1\mid\theta_2)\psi$	$\{[-\mid\rightarrow\mu)\neg[-\mid\rightarrow\nu)\neg[\theta_1\mid\theta_2)\psi,\ \neg[-\mid\rightarrow\mu)[-\mid\rightarrow\nu)\mathbf{F}\}$

En este ejemplo, además de ser aplicable lo dicho en el ejemplo anterior, ha de tenerse en cuenta que el operador de negación que hay justo delante del intervalo $[\theta_1\mid\theta_2)$ expresa que dicho intervalo debe *obligatoriamente* poder construirse en el contexto proporcionado por $\mathcal{I}$ (esto es, $[-\mid\rightarrow\mu)\neg[-\mid\rightarrow\nu)$ en este ejemplo), por lo que $\mathcal{I}$ no debe colapsarse en el instante siguiente al actual, motivo por el que también se introduce $\neg\mid\mathcal{I}\mathbf{F}\mid$ en el campo *Next*.

Conclusión: La introducción en el campo *Next* de la fórmula que expresa la *imposibilidad de colapso del prefijo* $\mathcal{I}$ ($\neg\mid\mathcal{I}\mathbf{F}\mid$) impide que dicho prefijo se pueda reducir antes de que se reduzca la FINA $\neg[\theta_1\mid\theta_2)\psi$ anidada al mismo.

Obsérvese que, aunque las fórmulas vistas en el subapartado c) anterior (FFB o FE anidada a una SMIA) se ajustan también a la definición o estructura dada en este subapartado para las FINAs anidadas a una SMIA, el último paso del proceso de expansión de dichas fórmulas siempre se realizará tal y como se ha explicado en el subapartado c), ya que nuestro algoritmo, antes de determinar que η es una

FINA anidada a una SMIA, comprueba primero que no es ni una FFB anidada a una SMIA ni una FE anidada a una SMIA.

El Lema 3.12 publicado en [Ramakr96a] expresa algo muy parecido a lo que se ha dicho en este subapartado d), con la diferencia fundamental de que en dicho lema no se distingue entre FINA anidada a una SMIA, FFB anidada a una SMIA y FE anidada a una SMIA, como nosotros hacemos. Por otra parte, cuando en ese lema se emplea la fórmula que expresa que la SMIA $\mathcal{I}$ no puede colapsarse en el instante siguiente al actual, al no indicar expresamente nada al respecto, se sobreentiende que $\mathcal{I}$ conserva todos los operadores de negación que tuvieran (inmediatamente delante) sus MIAs en η, mientras que nosotros eliminamos esos operadores en dicha fórmula. Así, debemos concluir que si se hubiera aplicado en nuestro algoritmo el mencionado Lema 3.12 tal y como aparece en [Ramakr96a], algunas fórmulas, como la del Ejemplo 5.8 , también se resolverían satisfactoriamente, aunque introduciendo dos fórmulas en el campo *Next* ($[-|\rightarrow$p0$)\neg[\rightarrow$p1$|\rightarrow)$**F** y $\neg[-|\rightarrow$p0$)$**F**) en lugar de una sola (su forma normal: $\neg[-|\rightarrow$p0$)[\rightarrow$p1$|\rightarrow)$**F**), como hace nuestro algoritmo. Sin embargo, otras fórmulas, como la del Ejemplo 5.7 , no se podrían resolver, ya que (como se explicó en dicho ejemplo) al introducir la propia fórmula $\neg[-|\rightarrow$p0$)\neg[\rightarrow$p1$|\rightarrow)$**F** en el campo *Next* se impide que p0 se pueda dar en el futuro, con lo que nunca se encontraría el extremo derecho del intervalo.

5.1.4.3. TABLAS CON EJEMPLOS DE CÓMO SE EXPANDEN CIERTAS FÓRMULAS DE INTERVALO

A modo de conclusión, y para resumir, completar y clarificar todo lo comentado en el subapartado B) acerca de las reglas de expansión de las fórmulas de intervalo que no son FPMs anidadas a una SMIA, se expondrán a continuación una serie de tablas. En todas ellas se hace distinción explícita de cuándo un patrón de búsqueda está formado por una sola búsqueda, representándose, por ejemplo, como $\rightarrow\mu$ o $\rightarrow\nu$, o por más de una, mostrándose en este caso como $\rightarrow\mu,\theta_1$ o $\rightarrow\nu,\theta_2$, donde μ y ν siempre representarán a cualquier fórmula FIL, mientras que θ_i representará al resto de las búsquedas (una o más) del correspondiente patrón de búsqueda. Además, una línea azul doble divide verticalmente todas las tablas en tres zonas:

- La de la izquierda representa la fórmula η elegida por el algoritmo en un momento determinado para su análisis y expansión.

- La central, que representa la creación de nuevos nodos, contiene una columna por cada nuevo nodo que se crea a partir del nodo actual, esto es, conservando el mismo contenido que éste en todos sus campos (excepto η, que se mueve del campo *New* al campo *Old*) y añadiendo además el conjunto de fórmulas {*Reductor*, *Reduct*} que se muestra en la columna $new_i(\eta)$ al campo *New* del nuevo nodo. Cuando en esta parte aparece una celda que contiene el símbolo — o que presenta todo su contenido en rojo (debido a las razones explicadas en el subapartado 1) anterior), significa que *ese nodo no se crea*. Así, por ejemplo, en la Tabla 5.3, para la fórmula de la fila 10 se crean dos nuevos nodos, para la de la fila 11, ninguno, y para la de la fila 14, sólo uno.

- La de la derecha representa la actualización del nodo actual, por lo que las columnas $new_c(\eta)$ y $next_c(\eta)$ muestran el conjunto de fórmulas que hay que añadir en cada caso a los campos del mismo nombre de ese nodo. Obsérvese que, incluso cuando no se crea un nuevo nodo para un determinado par (*Reductor*, *Reduct*) (mostrado en rojo), los correspondientes reductores sí que se utilizan para añadir su negación al campo *New* del nodo actual.

La Tabla 5.3 muestra las reglas de expansión aplicables para todos y cada uno de los posibles tipos de FINAs existentes. En ella ψ hace referencia a cualquier fórmula FIL, excepto en los casos en los que se especifica lo contrario (filas 6, 12, 18 y 24), donde ψ no puede ser la constante lógica **F**. Aquí, la fórmula μ siempre representa la búsqueda inicial del primer patrón de búsqueda del intervalo más externo de η, mientras que ν simboliza lo mismo, pero para su segundo patrón, cuando éste no es trivial. Para indicar que la fórmula objetivo de la primera búsqueda de ambos patrones coincide, μ encabezará ambos patrones, tal y como por ejemplo ocurre en la fórmula de la fila 2.

Obsérvese que otra línea azul doble divide horizontalmente la Tabla 5.3 en dos, indicándose con ello que el intervalo más externo de η no está negado en las doce filas superiores, mientras que sí lo está en las doce inferiores. Obsérvese también que en todos los casos se debe añadir la propia fórmula analizada (η) al campo *Next* del nodo actual y que, como máximo, se crean cuatro nuevos nodos (ver columnas centrales) para este tipo de fórmulas, las FINAs, ya que al expandirlas no se tiene en cuenta la fórmula anidada ψ hasta que se reduce el intervalo más externo de η a una MIA. Esto último parece entrar en contradicción con lo expresado para las fórmulas cuyo intervalo más externo tiene como segundo patrón de búsqueda al

Tabla 5.3. Reglas de expansión para los distintos tipos de FINAs

	η	Notas	$new_1(\eta)$	$new_2(\eta)$	$new_3(\eta)$	$new_4(\eta)$	$new_c(\eta)$	$next_c(\eta)$
1	$[\to\mu,\theta_1 \mid \to\nu,\theta_2)\,\psi$		$\{\mu,\ [\theta_1 \mid \to\nu,\theta_2)\,\psi\}$	$\{\nu,\ [\to\mu,\theta_1 \mid \theta_2)\,\psi\}$	$\{[\to\mu,\theta_1 \mid \to)\mathbf{F},\ \mathbf{T}\}$	$\{[\to\nu,\theta_2 \mid \to)\mathbf{F},\ \mathbf{T}\}$	$\{\neg\mu,\ \neg\nu,\ \neg[\to\mu,\theta_1 \mid \to)\mathbf{F},\ \neg[\to\nu,\theta_2 \mid \to)\mathbf{F}\}$	$\{\eta\}$
2	$[\to\mu,\theta_1 \mid \to\mu,\theta_2)\,\psi$		$\{\mu,\ [\theta_1 \mid \theta_2)\,\psi\}$	$\{[\to\mu,\theta_1 \mid \to)\mathbf{F},\ \mathbf{T}\}$	$\{[\to\mu,\theta_2 \mid \to)\mathbf{F},\ \mathbf{T}\}$	—	$\{\neg\mu,\ \neg[\to\mu,\theta_1 \mid \to)\mathbf{F},\ \neg[\to\mu,\theta_2 \mid \to)\mathbf{F}\}$	$\{\eta\}$
3	$[\to\mu,\theta_1 \mid \to\nu)\,\psi$		$\{\mu,\ [\theta_1 \mid \to\nu)\,\psi\}$	$\{\nu,\ \mathbf{T}\}$	$\{[\to\mu,\theta_1 \mid \to)\mathbf{F},\ \mathbf{T}\}$	$\{[\to\nu \mid \to)\mathbf{F},\ \mathbf{T}\}$	$\{\neg\mu,\ \neg\nu,\ \neg[\to\mu,\theta_1 \mid \to)\mathbf{F},\ \neg[\to\nu \mid \to)\mathbf{F}\}$	$\{\eta\}$
4	$[\to\mu,\theta_1 \mid \to\mu)\,\psi$		$\{\mu,\ \mathbf{T}\}$	$\{[\to\mu,\theta_1 \mid \to)\mathbf{F},\ \mathbf{T}\}$	$\{[\to\mu \mid \to)\mathbf{F},\ \mathbf{T}\}$	—	$\{\neg\mu,\ \neg[\to\mu,\theta_1 \mid \to)\mathbf{F},\ \neg[\to\mu \mid \to)\mathbf{F}\}$	$\{\eta\}$
5	$[\to\mu,\theta_1 \mid \to)\mathbf{F}$		$\{\mu,\ [\theta_1 \mid \to)\mathbf{F}\}$	—	—	—	$\{\neg\mu\}$	$\{\eta\}$
6	$[\to\mu,\theta_1 \mid \to)\,\psi$	$\psi \neq \mathbf{F}$	$\{\mu,\ [\theta_1 \mid \to)\,\psi\}$	$\{[\to\mu,\theta_1 \mid \to)\mathbf{F},\ \mathbf{T}\}$	—	—	$\{\neg\mu,\ \neg[\to\mu,\theta_1 \mid \to)\mathbf{F}\}$	$\{\eta\}$
7	$[\to\mu \mid \to\nu,\theta_2)\,\psi$		$\{\mu,\ [- \mid \to\nu,\theta_2)\,\psi\}$	$\{\nu,\ [\to\mu \mid \theta_2)\,\psi\}$	$\{[\to\mu \mid \to)\mathbf{F},\ \mathbf{T}\}$	$\{[\to\nu,\theta_2 \mid \to)\mathbf{F},\ \mathbf{T}\}$	$\{\neg\mu,\ \neg\nu,\ \neg[\to\mu \mid \to)\mathbf{F},\ \neg[\to\nu,\theta_2 \mid \to)\mathbf{F}\}$	$\{\eta\}$
8	$[\to\mu \mid \to\mu,\theta_2)\,\psi$		$\{\mu,\ [- \mid \theta_2)\,\psi\}$	$\{[\to\mu \mid \to)\mathbf{F},\ \mathbf{T}\}$	$\{[\to\mu,\theta_2 \mid \to)\mathbf{F},\ \mathbf{T}\}$	—	$\{\neg\mu,\ \neg[\to\mu \mid \to)\mathbf{F},\ \neg[\to\mu,\theta_2 \mid \to)\mathbf{F}\}$	$\{\eta\}$
9	$[\to\mu \mid \to\nu)\,\psi$		$\{\mu,\ [- \mid \to\nu)\,\psi\}$	$\{\nu,\ \mathbf{T}\}$	$\{[\to\mu \mid \to)\mathbf{F},\ \mathbf{T}\}$	$\{[\to\nu \mid \to)\mathbf{F},\ \mathbf{T}\}$	$\{\neg\mu,\ \neg\nu,\ \neg[\to\mu \mid \to)\mathbf{F},\ \neg[\to\nu \mid \to)\mathbf{F}\}$	$\{\eta\}$
10	$[\to\mu \mid \to\mu)\,\psi$		$\{\mu,\ \mathbf{T}\}$	$\{[\to\mu \mid \to)\mathbf{F},\ \mathbf{T}\}$	—	—	$\{\neg\mu,\ \neg[\to\mu \mid \to)\mathbf{F}\}$	$\{\eta\}$
11	$[\to\mu \mid \to)\mathbf{F}$		$\{\mu,\ \mathbf{F}\}$	—	—	—	$\{\neg\mu\}$	$\{\eta\}$
12	$[\to\mu \mid \to)\,\psi$	$\psi \neq \mathbf{F}$	$\{\mu,\ \psi\}$	$\{[\to\mu \mid \to)\mathbf{F},\ \mathbf{T}\}$	—	—	$\{\neg\mu,\ \neg[\to\mu \mid \to)\mathbf{F}\}$	$\{\eta\}$
13	$\neg[\to\mu,\theta_1 \mid \to\nu,\theta_2)\,\psi$		$\{\mu,\ \neg[\theta_1 \mid \to\nu,\theta_2)\,\psi\}$	$\{\nu,\ \neg[\to\mu,\theta_1 \mid \theta_2)\,\psi\}$	$\{[\to\mu,\theta_1 \mid \to)\mathbf{F},\ \mathbf{F}\}$	$\{[\to\nu,\theta_2 \mid \to)\mathbf{F},\ \mathbf{F}\}$	$\{\neg\mu,\ \neg\nu,\ \neg[\to\mu,\theta_1 \mid \to)\mathbf{F},\ \neg[\to\nu,\theta_2 \mid \to)\mathbf{F}\}$	$\{\eta\}$
14	$\neg[\to\mu,\theta_1 \mid \to\mu,\theta_2)\,\psi$		$\{\mu,\ \neg[\theta_1 \mid \theta_2)\,\psi\}$	$\{[\to\mu,\theta_1 \mid \to)\mathbf{F},\ \mathbf{F}\}$	$\{[\to\mu,\theta_2 \mid \to)\mathbf{F},\ \mathbf{F}\}$	—	$\{\neg\mu,\ \neg[\to\mu,\theta_1 \mid \to)\mathbf{F},\ \neg[\to\mu,\theta_2 \mid \to)\mathbf{F}\}$	$\{\eta\}$
15	$\neg[\to\mu,\theta_1 \mid \to\nu)\,\psi$		$\{\mu,\ \neg[\theta_1 \mid \to\nu)\,\psi\}$	$\{\nu,\ \mathbf{F}\}$	$\{[\to\mu,\theta_1 \mid \to)\mathbf{F},\ \mathbf{F}\}$	$\{[\to\nu \mid \to)\mathbf{F},\ \mathbf{F}\}$	$\{\neg\mu,\ \neg\nu,\ \neg[\to\mu,\theta_1 \mid \to)\mathbf{F},\ \neg[\to\nu \mid \to)\mathbf{F}\}$	$\{\eta\}$
16	$\neg[\to\mu,\theta_1 \mid \to\mu)\,\psi$		$\{\mu,\ \mathbf{F}\}$	$\{[\to\mu,\theta_1 \mid \to)\mathbf{F},\ \mathbf{F}\}$	$\{[\to\mu \mid \to)\mathbf{F},\ \mathbf{F}\}$	—	$\{\neg\mu,\ \neg[\to\mu,\theta_1 \mid \to)\mathbf{F},\ \neg[\to\mu \mid \to)\mathbf{F}\}$	$\{\eta\}$
17	$\neg[\to\mu,\theta_1 \mid \to)\mathbf{F}$		$\{\mu,\ \neg[\theta_1 \mid \to)\mathbf{F}\}$	—	—	—	$\{\neg\mu\}$	$\{\eta\}$
18	$\neg[\to\mu,\theta_1 \mid \to)\,\psi$	$\psi \neq \mathbf{F}$	$\{\mu,\ \neg[\theta_1 \mid \to)\,\psi\}$	$\{[\to\mu,\theta_1 \mid \to)\mathbf{F},\ \mathbf{F}\}$	—	—	$\{\neg\mu,\ \neg[\to\mu,\theta_1 \mid \to)\mathbf{F}\}$	$\{\eta\}$
19	$\neg[\to\mu \mid \to\nu,\theta_2)\,\psi$		$\{\mu,\ \neg[- \mid \to\nu,\theta_2)\,\psi\}$	$\{\nu,\ \neg[\to\mu \mid \theta_2)\,\psi\}$	$\{[\to\mu \mid \to)\mathbf{F},\ \mathbf{F}\}$	$\{[\to\nu,\theta_2 \mid \to)\mathbf{F},\ \mathbf{F}\}$	$\{\neg\mu,\ \neg\nu,\ \neg[\to\mu \mid \to)\mathbf{F},\ \neg[\to\nu,\theta_2 \mid \to)\mathbf{F}\}$	$\{\eta\}$
20	$\neg[\to\mu \mid \to\mu,\theta_2)\,\psi$		$\{\mu,\ \neg[- \mid \theta_2)\,\psi\}$	$\{[\to\mu \mid \to)\mathbf{F},\ \mathbf{F}\}$	$\{[\to\mu,\theta_2 \mid \to)\mathbf{F},\ \mathbf{F}\}$	—	$\{\neg\mu,\ \neg[\to\mu \mid \to)\mathbf{F},\ \neg[\to\mu,\theta_2 \mid \to)\mathbf{F}\}$	$\{\eta\}$
21	$\neg[\to\mu \mid \to\nu)\,\psi$		$\{\mu,\ \neg[- \mid \to\nu)\,\psi\}$	$\{\nu,\ \mathbf{F}\}$	$\{[\to\mu \mid \to)\mathbf{F},\ \mathbf{F}\}$	$\{[\to\nu \mid \to)\mathbf{F},\ \mathbf{F}\}$	$\{\neg\mu,\ \neg\nu,\ \neg[\to\mu \mid \to)\mathbf{F},\ \neg[\to\nu \mid \to)\mathbf{F}\}$	$\{\eta\}$
22	$\neg[\to\mu \mid \to\mu)\,\psi$		$\{\mu,\ \mathbf{F}\}$	$\{[\to\mu \mid \to)\mathbf{F},\ \mathbf{F}\}$	—	—	$\{\neg\mu,\ \neg[\to\mu \mid \to)\mathbf{F}\}$	$\{\eta\}$
23	$\neg[\to\mu \mid \to)\mathbf{F}$		$\{\mu,\ \mathbf{T}\}$	—	—	—	$\{\neg\mu\}$	$\{\eta\}$
24	$\neg[\to\mu \mid \to)\,\psi$	$\psi \neq \mathbf{F}$	$\{\mu,\ \neg\psi\}$	$\{[\to\mu \mid \to)\mathbf{F},\ \mathbf{F}\}$	—	—	$\{\neg\mu,\ \neg[\to\mu \mid \to)\mathbf{F}\}$	$\{\eta\}$

Tabla 5.4. Reglas de expansión para algunos tipos de FPPs anidadas a una SMIA (ψ es una FPP)

	η	$new_1(\eta)$	$new_2(\eta)$	$new_3(\eta)$	$new_4(\eta)$	$new_c(\eta)$	$next_c(\eta)$
1	$[-\mid\rightarrow\nu,\theta_2)\,\psi$	$\{\nu,\ [-\mid\theta_2)\,\psi\}$	$\{[\rightarrow\nu,\theta_2\mid\rightarrow)\mathbf{F},\ \mathbf{T}\}$	—	—	$\{\neg\nu,\ \neg[\rightarrow\nu,\theta_2\mid\rightarrow)\mathbf{F},\ \psi\}$	$\varnothing$
2	$\neg[-\mid\rightarrow\nu,\theta_2)\,\psi$	$\{\nu,\ \neg[-\mid\theta_2)\,\psi\}$	$\{[\rightarrow\nu,\theta_2\mid\rightarrow)\mathbf{F},\ \mathbf{F}\}$	—	—	$\{\neg\nu,\ \neg[\rightarrow\nu,\theta_2\mid\rightarrow)\mathbf{F},\ \neg\psi\}$	$\varnothing$
3	$[-\mid\rightarrow\nu)\,\psi$	$\{\nu,\ \mathbf{T}\}$	$\{[\rightarrow\nu\mid\rightarrow)\mathbf{F},\ \mathbf{T}\}$	—	—	$\{\neg\nu,\ \neg[\rightarrow\nu\mid\rightarrow)\mathbf{F},\ \psi\}$	$\varnothing$
4	$\neg[-\mid\rightarrow\nu)\,\psi$	$\{\nu,\ \mathbf{F}\}$	$\{[\rightarrow\nu\mid\rightarrow)\mathbf{F},\ \mathbf{F}\}$	—	—	$\{\neg\nu,\ \neg[\rightarrow\nu\mid\rightarrow)\mathbf{F},\ \neg\psi\}$	$\varnothing$
5	$[-\mid\rightarrow\mu,\theta_1)[-\mid\rightarrow\nu,\theta_2)\,\psi$	$\{\mu,\ [-\mid\theta_1)[-\mid\rightarrow\nu,\theta_2)\,\psi\}$	$\{[\rightarrow\mu,\theta_1\mid\rightarrow)\mathbf{F},\ \mathbf{T}\}$	$\{[-\mid\rightarrow\mu,\theta_1)\,\nu,$ $[-\mid\rightarrow\mu,\theta_1)[-\mid\theta_2)\,\psi\}$	$\{[-\mid\rightarrow\mu,\theta_1)[\rightarrow\nu,\theta_2\mid\rightarrow)\mathbf{F},$ $[-\mid\rightarrow\mu,\theta_1)\mathbf{T}\}$	$\{\neg\mu,\ \neg[\rightarrow\mu,\theta_1\mid\rightarrow)\mathbf{F},\ \neg[-\mid\rightarrow\mu,\theta_1)\,\nu,$ $\neg[-\mid\rightarrow\mu,\theta_1)[\rightarrow\nu,\theta_2\mid\rightarrow)\mathbf{F},\ \psi\}$	$\varnothing$
6	$[-\mid\rightarrow\mu,\theta_1)\neg[-\mid\rightarrow\nu,\theta_2)\,\psi$	$\{\mu,\ [-\mid\theta_1)\neg[-\mid\rightarrow\nu,\theta_2)\,\psi\}$	$\{[\rightarrow\mu,\theta_1\mid\rightarrow)\mathbf{F},\ \mathbf{T}\}$	$\{[-\mid\rightarrow\mu,\theta_1)\,\nu,$ $[-\mid\rightarrow\mu,\theta_1)\neg[-\mid\theta_2)\,\psi\}$	$\{[-\mid\rightarrow\mu,\theta_1)[\rightarrow\nu,\theta_2\mid\rightarrow)\mathbf{F},$ $[-\mid\rightarrow\mu,\theta_1)\mathbf{F}\}$	$\{\neg\mu,\ \neg[\rightarrow\mu,\theta_1\mid\rightarrow)\mathbf{F},\ \neg[-\mid\rightarrow\mu,\theta_1)\,\nu,$ $\neg[-\mid\rightarrow\mu,\theta_1)[\rightarrow\nu,\theta_2\mid\rightarrow)\mathbf{F},\ \neg\psi\}$	$\varnothing$
7	$\neg[-\mid\rightarrow\mu,\theta_1)[-\mid\rightarrow\nu,\theta_2)\,\psi$	$\{\mu,\ \neg[-\mid\theta_1)[-\mid\rightarrow\nu,\theta_2)\,\psi\}$	$\{[\rightarrow\mu,\theta_1\mid\rightarrow)\mathbf{F},\ \mathbf{F}\}$	$\{[-\mid\rightarrow\mu,\theta_1)\,\nu,$ $\neg[-\mid\rightarrow\mu,\theta_1)[-\mid\theta_2)\,\psi\}$	$\{[-\mid\rightarrow\mu,\theta_1)[\rightarrow\nu,\theta_2\mid\rightarrow)\mathbf{F},$ $\neg[-\mid\rightarrow\mu,\theta_1)\mathbf{T}\}$	$\{\neg\mu,\ \neg[\rightarrow\mu,\theta_1\mid\rightarrow)\mathbf{F},\ \neg[-\mid\rightarrow\mu,\theta_1)\,\nu,$ $\neg[-\mid\rightarrow\mu,\theta_1)[\rightarrow\nu,\theta_2\mid\rightarrow)\mathbf{F},\ \neg\psi\}$	$\varnothing$
8	$\neg[-\mid\rightarrow\mu,\theta_1)\neg[-\mid\rightarrow\nu,\theta_2)\,\psi$	$\{\mu,$ $\neg[-\mid\theta_1)\neg[-\mid\rightarrow\nu,\theta_2)\,\psi\}$	$\{[\rightarrow\mu,\theta_1\mid\rightarrow)\mathbf{F},\ \mathbf{F}\}$	$\{[-\mid\rightarrow\mu,\theta_1)\,\nu,$ $\neg[-\mid\rightarrow\mu,\theta_1)\neg[-\mid\theta_2)\,\psi\}$	$\{[-\mid\rightarrow\mu,\theta_1)[\rightarrow\nu,\theta_2\mid\rightarrow)\mathbf{F},$ $\neg[-\mid\rightarrow\mu,\theta_1)\mathbf{F}\}$	$\{\neg\mu,\ \neg[\rightarrow\mu,\theta_1\mid\rightarrow)\mathbf{F},\ \neg[-\mid\rightarrow\mu,\theta_1)\,\nu,$ $\neg[-\mid\rightarrow\mu,\theta_1)[\rightarrow\nu,\theta_2\mid\rightarrow)\mathbf{F},\ \psi\}$	$\varnothing$
9	$[-\mid\rightarrow\mu)[-\mid\rightarrow\nu)\,\psi$	$\{\mu,\ \mathbf{T}\}$	$\{[\rightarrow\mu\mid\rightarrow)\mathbf{F},\ \mathbf{T}\}$	$\{[-\mid\rightarrow\mu)\,\nu,\ [-\mid\rightarrow\mu)\mathbf{T}\}$	$\{[-\mid\rightarrow\mu)[\rightarrow\nu\mid\rightarrow)\mathbf{F},$ $[-\mid\rightarrow\mu)\mathbf{T}\}$	$\{\neg\mu,\ \neg[\rightarrow\mu\mid\rightarrow)\mathbf{F},\ \neg[-\mid\rightarrow\mu)\,\nu,$ $\neg[-\mid\rightarrow\mu)[\rightarrow\nu\mid\rightarrow)\mathbf{F},\ \psi\}$	$\varnothing$
10	$[-\mid\rightarrow\mu)\neg[-\mid\rightarrow\nu)\,\psi$	$\{\mu,\ \mathbf{T}\}$	$\{[\rightarrow\mu\mid\rightarrow)\mathbf{F},\ \mathbf{T}\}$	$\{[-\mid\rightarrow\mu)\,\nu,\ [-\mid\rightarrow\mu)\mathbf{F}\}$	$\{[-\mid\rightarrow\mu)[\rightarrow\nu\mid\rightarrow)\mathbf{F},$ $[-\mid\rightarrow\mu)\mathbf{F}\}$	$\{\neg\mu,\ \neg[\rightarrow\mu\mid\rightarrow)\mathbf{F},\ \neg[-\mid\rightarrow\mu)\,\nu,$ $\neg[-\mid\rightarrow\mu)[\rightarrow\nu\mid\rightarrow)\mathbf{F},\ \neg\psi\}$	$\varnothing$
11	$\neg[-\mid\rightarrow\mu)[-\mid\rightarrow\nu)\,\psi$	$\{\mu,\ \mathbf{F}\}$	$\{[\rightarrow\mu\mid\rightarrow)\mathbf{F},\ \mathbf{F}\}$	$\{[-\mid\rightarrow\mu)\,\nu,\ \neg[-\mid\rightarrow\mu)\mathbf{T}\}$	$\{[-\mid\rightarrow\mu)[\rightarrow\nu\mid\rightarrow)\mathbf{F},$ $\neg[-\mid\rightarrow\mu)\mathbf{T}\}$	$\{\neg\mu,\ \neg[\rightarrow\mu\mid\rightarrow)\mathbf{F},\ \neg[-\mid\rightarrow\mu)\,\nu,$ $\neg[-\mid\rightarrow\mu)[\rightarrow\nu\mid\rightarrow)\mathbf{F},\ \neg\psi\}$	$\varnothing$
12	$\neg[-\mid\rightarrow\mu)\neg[-\mid\rightarrow\nu)\,\psi$	$\{\mu,\ \mathbf{F}\}$	$\{[\rightarrow\mu\mid\rightarrow)\mathbf{F},\ \mathbf{F}\}$	$\{[-\mid\rightarrow\mu)\,\nu,\ \neg[-\mid\rightarrow\mu)\mathbf{F}\}$	$\{[-\mid\rightarrow\mu)[\rightarrow\nu\mid\rightarrow)\mathbf{F},$ $\neg[-\mid\rightarrow\mu)\mathbf{F}\}$	$\{\neg\mu,\ \neg[\rightarrow\mu\mid\rightarrow)\mathbf{F},\ \neg[-\mid\rightarrow\mu)\,\nu,$ $\neg[-\mid\rightarrow\mu)[\rightarrow\nu\mid\rightarrow)\mathbf{F},\ \psi\}$	$\varnothing$

trivial (filas 5, 6, 11, 12, 17, 18, 23 y 24 de la tabla), donde se hace referencia explícita al tipo concreto de ψ. Pero eso no es así, ya que esta distinción se hace únicamente atendiendo a que cuando η es del tipo $[\theta|\rightarrow)\mathbf{F}$ (o su negación), donde θ representa a cualquier patrón de búsqueda no trivial, la fórmula que expresa el fallo de su patrón de búsqueda θ coincide con la propia η, por lo que, al no poder ser una fórmula reductor de sí misma, el proceso de reducción no la genera como tal, mientras que cuando η es del tipo $[\theta|\rightarrow)\psi$ (o su negación), con $\psi \neq \mathbf{F}$, entonces $[\theta|\rightarrow)\mathbf{F}$ sí que es un reductor válido para η, siendo ésta la única diferencia en el tratamiento de ambos tipos de fórmulas. Por tanto, se puede concluir que la diferencia no está en la regla a aplicar en el proceso de expansión (que es siempre la misma para todas las FINAs), sino en los pares (*Reductor*, *Reduct*) generados durante la reducción de η.

En la Tabla 5.4 se presentan algunos ejemplos de cómo se expanden las FPPs anidadas a una SMIA. Consecuentemente, ψ debe ser una FPP; en otro caso, η sería una fórmula de otro tipo, por lo que se tendrían que aplicar unas reglas distintas. La línea azul doble que divide horizontalmente la tabla en dos indica que el prefijo $\mathcal{J}$ de las fórmulas superiores (filas 1 a 4) está formado por una sola MIA, mientras que el de las inferiores (filas 5 a 12) está compuesto por dos MIAs. A su vez, cada una de estas dos zonas se divide horizontalmente mediante una línea azul más gruesa, que indica que todos los patrones de búsqueda de las fórmulas que quedan por encima de ella (filas 1, 2 y de 5 a 8) contienen más de una búsqueda, mientras que los de las que quedan por debajo (filas 3, 4 y de 9 a 12) sólo contienen una. Obsérvese que las distintas filas de las cuatro zonas horizontales así resultantes almacenan las distintas "variantes" del mismo tipo de fórmula, surgidas como consecuencia de negar o no cada una de las MIAs que forman su prefijo $\mathcal{J}$. Adviértase también que esos operadores de negación son tenidos en cuenta a la hora de introducir ψ o $\neg\psi$ en el campo *New* del nodo actual. El símbolo $\varnothing$ que alberga la última columna significa que para este tipo de fórmulas no se añade nada al campo *Next* de dicho nodo.

Repárese en que, siguiendo la notación empleada, no es posible plasmar en la Tabla 5.4 las reglas de expansión para todos los casos posibles de FPPs anidadas a una SMIA, puesto que el prefijo $\mathcal{J}$ de estas fórmulas puede estar formado por cualquier número de MIAs, por lo que no está acotado el número de sus posibles tipos, al contrario de lo que ocurre en la Tabla 5.3, que sí lo está. Además, se debe tener en cuenta que la zona de creación de nuevos nodos (columnas centrales) debe

tener dos columnas por cada MIA que tenga el prefijo $\mathcal{I}$ de la fórmula analizada η, lo que haría imposible la representación (en el espacio disponible dentro de un folio) de las reglas correspondientes a fórmulas con prefijos un poco largos.

Lo dicho en el párrafo anterior es también aplicable a las dos tablas siguientes. En la primera de ellas, Tabla 5.5, se muestran las reglas de expansión para algunas FFBs y FEs anidadas a una SMIA. Una línea azul doble horizontal de nuevo divide esta tabla en dos, separando las distintas variantes de los dos tipos de fórmulas que se presentan: las de arriba tienen una sola MIA en su prefijo $\mathcal{I}$ y todos sus patrones de búsqueda contienen más de una búsqueda, mientras que las de abajo albergan fórmulas cuyo prefijo $\mathcal{I}$ está formado por dos MIAs y todos los patrones de búsqueda de η tienen una sola búsqueda. En las distintas filas de ambas zonas, η sólo difiere en los operadores de negación colocados delante de sus intervalos. A su vez, cada zona es dividida horizontalmente en dos por una línea azul más gruesa. Encima de ellas se colocan las FFBs anidadas a una SMIA (el número de intervalos negados, $\lceil \eta \rceil$, es par), por lo que su forma normal, $\langle \eta \rangle$, añadida al campo *Next* del nodo actual, representa a $[-|\rightarrow\mu,\theta_1)[\rightarrow\nu,\theta_2|\rightarrow)\mathbf{F}$ en las filas 1 y 2, y a $[-|\rightarrow\mu)[-|\rightarrow\nu)[\rightarrow\lambda|\rightarrow)\mathbf{F}$ en las filas 5 a 8. Por el contrario, debajo de esa línea azul más gruesa se colocan las FEs anidadas a una SMIA (es decir, $\lceil \eta \rceil$ es impar), por lo que su forma normal, $\langle \eta \rangle$, representa a $\neg[-|\rightarrow\mu,\theta_1)[\rightarrow\nu,\theta_2|\rightarrow)\mathbf{F}$ en las filas 3 y 4, y a $\neg[-|\rightarrow\mu)[-|\rightarrow\nu)[\rightarrow\lambda|\rightarrow)\mathbf{F}$ en las filas 9 a 12.

En la Tabla 5.6 , que muestra las reglas de expansión para algunas FINAs anidadas a una SMIA, se actúa de forma similar, de modo que la línea doble horizontal azul también separa las distintas variantes de los dos tipos de fórmulas que contiene la primera columna (η). En esta tabla, tanto λ como ψ representan a cualquier fórmula FIL, excepto en las cuatro filas inferiores, donde ψ no puede ser la constante lógica **F**. Si lo fuera, esas fórmulas serían FFBs o FEs anidadas a una SMIA (dependiendo de que $\lceil \eta \rceil$ sea par o impar) y no FINAs anidadas a una SMIA. De nuevo, una línea horizontal azul más gruesa separa en dos cada una de las dos zonas mencionadas anteriormente: por encima de la misma se colocan las fórmulas (η) cuya FINA (más externa, en el caso de tener más de una) no está negada (filas 1, 2, 5 y 6), motivo por el que en el campo *Next* del nodo actual sólo se inserta η, mientras que por debajo de esa línea se ponen las que tienen dicha FINA negada (filas 3, 4, 7 y 8), razón por la que en el campo *Next*, como ya se ha explicado, no sólo hay que insertar la propia η, sino también la fórmula que impide que el prefijo $\mathcal{I}$ se colapse, siendo $\neg[-|\rightarrow\mu)\mathbf{F}$ para las fórmulas representadas en la tabla.

Tabla 5.5. Reglas de expansión para algunos tipos de FFBs y FEs anidadas a una SMIA

	η	$new_1(\eta)$	$new_2(\eta)$	$new_3(\eta)$	$new_4(\eta)$	$new_5(\eta)$	$new_c(\eta)$	$next_c(\eta)$
1	$[-\mid\!\rightarrow\mu,\theta_1)[\rightarrow\nu,\theta_2\mid\!\rightarrow)\mathbf{F}$	$\{\mu,$ $[-\mid\theta_1)[\rightarrow\nu,\theta_2\mid\!\rightarrow)\mathbf{F}\}$	$\{[\rightarrow\mu,\theta_1\mid\!\rightarrow)\mathbf{F},$ $\mathbf{T}\}$	$\{[-\mid\!\rightarrow\mu,\theta_1)\,\nu,$ $[-\mid\!\rightarrow\mu,\theta_1)[\theta_2\mid\!\rightarrow)\mathbf{F}\}$	—	—	$\{\neg\mu,\ \neg[\rightarrow\mu,\theta_1\mid\!\rightarrow)\mathbf{F},$ $\neg[-\mid\!\rightarrow\mu,\theta_1)\,\nu\}$	$\{\langle\eta\rangle\}$
2	$\neg[-\mid\!\rightarrow\mu,\theta_1)\neg[\rightarrow\nu,\theta_2\mid\!\rightarrow)\mathbf{F}$	$\{\mu,$ $\neg[-\mid\theta_1)\neg[\rightarrow\nu,\theta_2\mid\!\rightarrow)\mathbf{F}\}$	$\{[\rightarrow\mu,\theta_1\mid\!\rightarrow)\mathbf{F},$ $\mathbf{F}\}$	$\{[-\mid\!\rightarrow\mu,\theta_1)\,\nu,$ $\neg[-\mid\!\rightarrow\mu,\theta_1)\neg[\theta_2\mid\!\rightarrow)\mathbf{F}\}$	—	—	$\{\neg\mu,\ \neg[\rightarrow\mu,\theta_1\mid\!\rightarrow)\mathbf{F},$ $\neg[-\mid\!\rightarrow\mu,\theta_1)\,\nu\}$	$\{\langle\eta\rangle\}$
3	$\neg[-\mid\!\rightarrow\mu,\theta_1)[\rightarrow\nu,\theta_2\mid\!\rightarrow)\mathbf{F}$	$\{\mu,$ $\neg[-\mid\theta_1)[\rightarrow\nu,\theta_2\mid\!\rightarrow)\mathbf{F}\}$	$\{[\rightarrow\mu,\theta_1\mid\!\rightarrow)\mathbf{F},$ $\mathbf{F}\}$	$\{[-\mid\!\rightarrow\mu,\theta_1)\,\nu,$ $\neg[-\mid\!\rightarrow\mu,\theta_1)[\theta_2\mid\!\rightarrow)\mathbf{F}\}$	—	—	$\{\neg\mu,\ \neg[\rightarrow\mu,\theta_1\mid\!\rightarrow)\mathbf{F},$ $\neg[-\mid\!\rightarrow\mu,\theta_1)\,\nu\}$	$\{\langle\eta\rangle\}$
4	$[-\mid\!\rightarrow\mu,\theta_1)\neg[\rightarrow\nu,\theta_2\mid\!\rightarrow)\mathbf{F}$	$\{\mu,$ $[-\mid\theta_1)\neg[\rightarrow\nu,\theta_2\mid\!\rightarrow)\mathbf{F}\}$	$\{[\rightarrow\mu,\theta_1\mid\!\rightarrow)\mathbf{F},$ $\mathbf{T}\}$	$\{[-\mid\!\rightarrow\mu,\theta_1)\,\nu,$ $[-\mid\!\rightarrow\mu,\theta_1)\neg[\theta_2\mid\!\rightarrow)\mathbf{F}\}$	—	—	$\{\neg\mu,\ \neg[\rightarrow\mu,\theta_1\mid\!\rightarrow)\mathbf{F},$ $\neg[-\mid\!\rightarrow\mu,\theta_1)\,\nu\}$	$\{\langle\eta\rangle\}$
5	$[-\mid\!\rightarrow\mu)[-\mid\!\rightarrow\nu)[\rightarrow\lambda\mid\!\rightarrow)\mathbf{F}$	$\{\mu,\ \mathbf{T}\}$	$\{[\rightarrow\mu\mid\!\rightarrow)\mathbf{F},\ \mathbf{T}\}$	$\{[-\mid\!\rightarrow\mu)\,\nu,\ [-\mid\!\rightarrow\mu)\mathbf{T}\}$	$\{[-\mid\!\rightarrow\mu)[\rightarrow\nu\mid\!\rightarrow)\mathbf{F},$ $[-\mid\!\rightarrow\mu)\mathbf{T}\}$	$\{[-\mid\!\rightarrow\mu)[-\mid\!\rightarrow\nu)\lambda,$ $[-\mid\!\rightarrow\mu)[-\mid\!\rightarrow\nu)\mathbf{F}\}$	$\{\neg\mu,\ \neg[\rightarrow\mu\mid\!\rightarrow)\mathbf{F},\ \neg[-\mid\!\rightarrow\mu)\,\nu,$ $\neg[-\mid\!\rightarrow\mu)[-\mid\!\rightarrow\nu)\lambda,$ $\neg[-\mid\!\rightarrow\mu)[\rightarrow\nu\mid\!\rightarrow)\mathbf{F}\}$	$\{\langle\eta\rangle\}$
6	$[-\mid\!\rightarrow\mu)\neg[-\mid\!\rightarrow\nu)\neg[\rightarrow\lambda\mid\!\rightarrow)\mathbf{F}$	$\{\mu,\ \mathbf{T}\}$	$\{[\rightarrow\mu\mid\!\rightarrow)\mathbf{F},\ \mathbf{T}\}$	$\{[-\mid\!\rightarrow\mu)\,\nu,\ [-\mid\!\rightarrow\mu)\mathbf{F}\}$	$\{[-\mid\!\rightarrow\mu)[\rightarrow\nu\mid\!\rightarrow)\mathbf{F},$ $[-\mid\!\rightarrow\mu)\mathbf{F}\}$	$\{[-\mid\!\rightarrow\mu)[-\mid\!\rightarrow\nu)\lambda,$ $[-\mid\!\rightarrow\mu)\neg[-\mid\!\rightarrow\nu)\mathbf{T}\}$	$\{\neg\mu,\ \neg[\rightarrow\mu\mid\!\rightarrow)\mathbf{F},\ \neg[-\mid\!\rightarrow\mu)\,\nu,$ $\neg[-\mid\!\rightarrow\mu)[-\mid\!\rightarrow\nu)\lambda,$ $\neg[-\mid\!\rightarrow\mu)[\rightarrow\nu\mid\!\rightarrow)\mathbf{F}\}$	$\{\langle\eta\rangle\}$
7	$\neg[-\mid\!\rightarrow\mu)[-\mid\!\rightarrow\nu)\neg[\rightarrow\lambda\mid\!\rightarrow)\mathbf{F}$	$\{\mu,\ \mathbf{F}\}$	$\{[\rightarrow\mu\mid\!\rightarrow)\mathbf{F},\ \mathbf{F}\}$	$\{[-\mid\!\rightarrow\mu)\,\nu,\ \neg[-\mid\!\rightarrow\mu)\mathbf{T}\}$	$\{[-\mid\!\rightarrow\mu)[\rightarrow\nu\mid\!\rightarrow)\mathbf{F},$ $\neg[-\mid\!\rightarrow\mu)\mathbf{T}\}$	$\{[-\mid\!\rightarrow\mu)[-\mid\!\rightarrow\nu)\lambda,$ $\neg[-\mid\!\rightarrow\mu)[-\mid\!\rightarrow\nu)\mathbf{T}\}$	$\{\neg\mu,\ \neg[\rightarrow\mu\mid\!\rightarrow)\mathbf{F},\ \neg[-\mid\!\rightarrow\mu)\,\nu,$ $\neg[-\mid\!\rightarrow\mu)[-\mid\!\rightarrow\nu)\lambda,$ $\neg[-\mid\!\rightarrow\mu)[\rightarrow\nu\mid\!\rightarrow)\mathbf{F}\}$	$\{\langle\eta\rangle\}$
8	$\neg[-\mid\!\rightarrow\mu)\neg[-\mid\!\rightarrow\nu)[\rightarrow\lambda\mid\!\rightarrow)\mathbf{F}$	$\{\mu,\ \mathbf{F}\}$	$\{[\rightarrow\mu\mid\!\rightarrow)\mathbf{F},\ \mathbf{F}\}$	$\{[-\mid\!\rightarrow\mu)\,\nu,\ \neg[-\mid\!\rightarrow\mu)\mathbf{F}\}$	$\{[-\mid\!\rightarrow\mu)[\rightarrow\nu\mid\!\rightarrow)\mathbf{F},$ $\neg[-\mid\!\rightarrow\mu)\mathbf{F}\}$	$\{[-\mid\!\rightarrow\mu)[-\mid\!\rightarrow\nu)\lambda,$ $\neg[-\mid\!\rightarrow\mu)\neg[-\mid\!\rightarrow\nu)\mathbf{F}\}$	$\{\neg\mu,\ \neg[\rightarrow\mu\mid\!\rightarrow)\mathbf{F},\ \neg[-\mid\!\rightarrow\mu)\,\nu,$ $\neg[-\mid\!\rightarrow\mu)[-\mid\!\rightarrow\nu)\lambda,$ $\neg[-\mid\!\rightarrow\mu)[\rightarrow\nu\mid\!\rightarrow)\mathbf{F}\}$	$\{\langle\eta\rangle\}$
9	$[-\mid\!\rightarrow\mu)[-\mid\!\rightarrow\nu)\neg[\rightarrow\lambda\mid\!\rightarrow)\mathbf{F}$	$\{\mu,\ \mathbf{T}\}$	$\{[\rightarrow\mu\mid\!\rightarrow)\mathbf{F},\ \mathbf{T}\}$	$\{[-\mid\!\rightarrow\mu)\,\nu,\ [-\mid\!\rightarrow\mu)\mathbf{T}\}$	$\{[-\mid\!\rightarrow\mu)[\rightarrow\nu\mid\!\rightarrow)\mathbf{F},$ $[-\mid\!\rightarrow\mu)\mathbf{T}\}$	$\{[-\mid\!\rightarrow\mu)[-\mid\!\rightarrow\nu)\lambda,$ $[-\mid\!\rightarrow\mu)[-\mid\!\rightarrow\nu)\mathbf{T}\}$	$\{\neg\mu,\ \neg[\rightarrow\mu\mid\!\rightarrow)\mathbf{F},\ \neg[-\mid\!\rightarrow\mu)\,\nu,$ $\neg[-\mid\!\rightarrow\mu)[-\mid\!\rightarrow\nu)\lambda,$ $\neg[-\mid\!\rightarrow\mu)[\rightarrow\nu\mid\!\rightarrow)\mathbf{F}\}$	$\{\langle\eta\rangle\}$
10	$[-\mid\!\rightarrow\mu)\neg[-\mid\!\rightarrow\nu)[\rightarrow\lambda\mid\!\rightarrow)\mathbf{F}$	$\{\mu,\ \mathbf{T}\}$	$\{[\rightarrow\mu\mid\!\rightarrow)\mathbf{F},\ \mathbf{T}\}$	$\{[-\mid\!\rightarrow\mu)\,\nu,\ [-\mid\!\rightarrow\mu)\mathbf{F}\}$	$\{[-\mid\!\rightarrow\mu)[\rightarrow\nu\mid\!\rightarrow)\mathbf{F},$ $[-\mid\!\rightarrow\mu)\mathbf{F}\}$	$\{[-\mid\!\rightarrow\mu)[-\mid\!\rightarrow\nu)\lambda,$ $[-\mid\!\rightarrow\mu)\neg[-\mid\!\rightarrow\nu)\mathbf{F}\}$	$\{\neg\mu,\ \neg[\rightarrow\mu\mid\!\rightarrow)\mathbf{F},\ \neg[-\mid\!\rightarrow\mu)\,\nu,$ $\neg[-\mid\!\rightarrow\mu)[-\mid\!\rightarrow\nu)\lambda,$ $\neg[-\mid\!\rightarrow\mu)[\rightarrow\nu\mid\!\rightarrow)\mathbf{F}\}$	$\{\langle\eta\rangle\}$
11	$\neg[-\mid\!\rightarrow\mu)[-\mid\!\rightarrow\nu)[\rightarrow\lambda\mid\!\rightarrow)\mathbf{F}$	$\{\mu,\ \mathbf{F}\}$	$\{[\rightarrow\mu\mid\!\rightarrow)\mathbf{F},\ \mathbf{F}\}$	$\{[-\mid\!\rightarrow\mu)\,\nu,\ \neg[-\mid\!\rightarrow\mu)\mathbf{T}\}$	$\{[-\mid\!\rightarrow\mu)[\rightarrow\nu\mid\!\rightarrow)\mathbf{F},$ $\neg[-\mid\!\rightarrow\mu)\mathbf{T}\}$	$\{[-\mid\!\rightarrow\mu)[-\mid\!\rightarrow\nu)\lambda,$ $\neg[-\mid\!\rightarrow\mu)[-\mid\!\rightarrow\nu)\mathbf{F}\}$	$\{\neg\mu,\ \neg[\rightarrow\mu\mid\!\rightarrow)\mathbf{F},\ \neg[-\mid\!\rightarrow\mu)\,\nu,$ $\neg[-\mid\!\rightarrow\mu)[-\mid\!\rightarrow\nu)\lambda,$ $\neg[-\mid\!\rightarrow\mu)[\rightarrow\nu\mid\!\rightarrow)\mathbf{F}\}$	$\{\langle\eta\rangle\}$
12	$\neg[-\mid\!\rightarrow\mu)\neg[-\mid\!\rightarrow\nu)\neg[\rightarrow\lambda\mid\!\rightarrow)\mathbf{F}$	$\{\mu,\ \mathbf{F}\}$	$\{[\rightarrow\mu\mid\!\rightarrow)\mathbf{F},\ \mathbf{F}\}$	$\{[-\mid\!\rightarrow\mu)\,\nu,\ \neg[-\mid\!\rightarrow\mu)\mathbf{F}\}$	$\{[-\mid\!\rightarrow\mu)[\rightarrow\nu\mid\!\rightarrow)\mathbf{F},$ $\neg[-\mid\!\rightarrow\mu)\mathbf{F}\}$	$\{[-\mid\!\rightarrow\mu)[-\mid\!\rightarrow\nu)\lambda,$ $\neg[-\mid\!\rightarrow\mu)\neg[-\mid\!\rightarrow\nu)\mathbf{T}\}$	$\{\neg\mu,\ \neg[\rightarrow\mu\mid\!\rightarrow)\mathbf{F},\ \neg[-\mid\!\rightarrow\mu)\,\nu,$ $\neg[-\mid\!\rightarrow\mu)[-\mid\!\rightarrow\nu)\lambda,$ $\neg[-\mid\!\rightarrow\mu)[\rightarrow\nu\mid\!\rightarrow)\mathbf{F}\}$	$\{\langle\eta\rangle\}$

Tabla 5.6. Reglas de expansión para algunos tipos de FINAs anidadas a una SMIA

	η	$new_1(\eta)$	$new_2(\eta)$	$new_3(\eta)$	$new_4(\eta)$	$new_5(\eta)$	$new_6(\eta)$	$new_c(\eta)$	$next_c(\eta)$
1	$[-\mid\rightarrow\mu)[\rightarrow\nu\mid\rightarrow\lambda)\,\psi$	$\{\mu,\ \mathbf{T}\}$	$\{[\rightarrow\mu\mid\rightarrow)\mathbf{F},$ $\mathbf{T}\}$	$\{[-\mid\rightarrow\mu)\,\nu,$ $[-\mid\rightarrow\mu)[-\mid\rightarrow\lambda)\,\psi\}$	$\{[-\mid\rightarrow\mu)\lambda,\ [-\mid\rightarrow\mu)\mathbf{T}\}$	$\{[-\mid\rightarrow\mu)[\rightarrow\nu\mid\rightarrow)\mathbf{F},$ $[-\mid\rightarrow\mu)\mathbf{T}\}$	$\{[-\mid\rightarrow\mu)[\rightarrow\lambda\mid\rightarrow)\mathbf{F},$ $[-\mid\rightarrow\mu)\mathbf{T}\}$	$\{\neg\mu,\ \neg[\rightarrow\mu\mid\rightarrow)\mathbf{F},$ $\neg[-\mid\rightarrow\mu)\,\nu,\ \neg[-\mid\rightarrow\mu)\lambda,$ $\neg[-\mid\rightarrow\mu)[\rightarrow\nu\mid\rightarrow)\mathbf{F},$ $\neg[-\mid\rightarrow\mu)[\rightarrow\lambda\mid\rightarrow)\mathbf{F}\}$	$\{\eta\}$
2	$\neg[-\mid\rightarrow\mu)[\rightarrow\nu\mid\rightarrow\lambda)\,\psi$	$\{\mu,\ \mathbf{F}\}$	$\{[\rightarrow\mu\mid\rightarrow)\mathbf{F},$ $\mathbf{F}\}$	$\{[-\mid\rightarrow\mu)\,\nu,$ $\neg[-\mid\rightarrow\mu)[-\mid\rightarrow\lambda)\,\psi\}$	$\{[-\mid\rightarrow\mu)\lambda,$ $\neg[-\mid\rightarrow\mu)\mathbf{T}\}$	$\{[-\mid\rightarrow\mu)[\rightarrow\nu\mid\rightarrow)\mathbf{F},$ $\neg[-\mid\rightarrow\mu)\mathbf{T}\}$	$\{[-\mid\rightarrow\mu)[\rightarrow\lambda\mid\rightarrow)\mathbf{F},$ $\neg[-\mid\rightarrow\mu)\mathbf{T}\}$	$\{\neg\mu,\ \neg[\rightarrow\mu\mid\rightarrow)\mathbf{F},$ $\neg[-\mid\rightarrow\mu)\,\nu,\ \neg[-\mid\rightarrow\mu)\lambda,$ $\neg[-\mid\rightarrow\mu)[\rightarrow\nu\mid\rightarrow)\mathbf{F},$ $\neg[-\mid\rightarrow\mu)[\rightarrow\lambda\mid\rightarrow)\mathbf{F}\}$	$\{\eta\}$
3	$[-\mid\rightarrow\mu)\neg[\rightarrow\nu\mid\rightarrow\lambda)\,\psi$	$\{\mu,\ \mathbf{T}\}$	$\{[\rightarrow\mu\mid\rightarrow)\mathbf{F},$ $\mathbf{T}\}$	$\{[-\mid\rightarrow\mu)\,\nu,$ $[-\mid\rightarrow\mu)\neg[-\mid\rightarrow\lambda)\,\psi\}$	$\{[-\mid\rightarrow\mu)\lambda,\ [-\mid\rightarrow\mu)\mathbf{F}\}$	$\{[-\mid\rightarrow\mu)[\rightarrow\nu\mid\rightarrow)\mathbf{F},$ $[-\mid\rightarrow\mu)\mathbf{F}\}$	$\{[-\mid\rightarrow\mu)[\rightarrow\lambda\mid\rightarrow)\mathbf{F},$ $[-\mid\rightarrow\mu)\mathbf{F}\}$	$\{\neg\mu,\ \neg[\rightarrow\mu\mid\rightarrow)\mathbf{F},$ $\neg[-\mid\rightarrow\mu)\,\nu,\ \neg[-\mid\rightarrow\mu)\lambda,$ $\neg[-\mid\rightarrow\mu)[\rightarrow\nu\mid\rightarrow)\mathbf{F},$ $\neg[-\mid\rightarrow\mu)[\rightarrow\lambda\mid\rightarrow)\mathbf{F}\}$	$\{\eta,$ $\neg[-\mid\rightarrow\mu)\mathbf{F}\}$
4	$\neg[-\mid\rightarrow\mu)\neg[\rightarrow\nu\mid\rightarrow\lambda)\,\psi$	$\{\mu,\ \mathbf{F}\}$	$\{[\rightarrow\mu\mid\rightarrow)\mathbf{F},$ $\mathbf{F}\}$	$\{[-\mid\rightarrow\mu)\,\nu,$ $\neg[-\mid\rightarrow\mu)\neg[-\mid\rightarrow\lambda)\,\psi\}$	$\{[-\mid\rightarrow\mu)\lambda,$ $\neg[-\mid\rightarrow\mu)\mathbf{F}\}$	$\{[-\mid\rightarrow\mu)[\rightarrow\nu\mid\rightarrow)\mathbf{F},$ $\neg[-\mid\rightarrow\mu)\mathbf{F}\}$	$\{[-\mid\rightarrow\mu)[\rightarrow\lambda\mid\rightarrow)\mathbf{F},$ $\neg[-\mid\rightarrow\mu)\mathbf{F}\}$	$\{\neg\mu,\ \neg[\rightarrow\mu\mid\rightarrow)\mathbf{F},$ $\neg[-\mid\rightarrow\mu)\,\nu,\ \neg[-\mid\rightarrow\mu)\lambda,$ $\neg[-\mid\rightarrow\mu)[\rightarrow\nu\mid\rightarrow)\mathbf{F},$ $\neg[-\mid\rightarrow\mu)[\rightarrow\lambda\mid\rightarrow)\mathbf{F}\}$	$\{\eta,$ $\neg[-\mid\rightarrow\mu)\mathbf{F}\}$
5	$[-\mid\rightarrow\mu)[\rightarrow\nu,\theta_2\mid\rightarrow)\,\psi$ $(\psi\neq\mathbf{F})$	$\{\mu,\ \mathbf{T}\}$	$\{[\rightarrow\mu\mid\rightarrow)\mathbf{F},$ $\mathbf{T}\}$	$\{[-\mid\rightarrow\mu)\,\nu,$ $[-\mid\rightarrow\mu)[\theta_2\mid\rightarrow)\,\psi\}$	$\{[-\mid\rightarrow\mu)[\rightarrow\nu,\theta_2\mid\rightarrow)\mathbf{F},$ $[-\mid\rightarrow\mu)\mathbf{T}\}$	—	—	$\{\neg\mu,\ \neg[\rightarrow\mu\mid\rightarrow)\mathbf{F},$ $\neg[-\mid\rightarrow\mu)\,\nu,$ $\neg[-\mid\rightarrow\mu)[\rightarrow\nu,\theta_2\mid\rightarrow)\mathbf{F}\}$	$\{\eta\}$
6	$\neg[-\mid\rightarrow\mu)[\rightarrow\nu,\theta_2\mid\rightarrow)\,\psi$ $(\psi\neq\mathbf{F})$	$\{\mu,\ \mathbf{F}\}$	$\{[\rightarrow\mu\mid\rightarrow)\mathbf{F},$ $\mathbf{F}\}$	$\{[-\mid\rightarrow\mu)\,\nu,$ $\neg[-\mid\rightarrow\mu)[\theta_2\mid\rightarrow)\,\psi\}$	$\{[-\mid\rightarrow\mu)[\rightarrow\nu,\theta_2\mid\rightarrow)\mathbf{F},$ $\neg[-\mid\rightarrow\mu)\mathbf{T}\}$	—	—	$\{\neg\mu,\ \neg[\rightarrow\mu\mid\rightarrow)\mathbf{F},$ $\neg[-\mid\rightarrow\mu)\,\nu,$ $\neg[-\mid\rightarrow\mu)[\rightarrow\nu,\theta_2\mid\rightarrow)\mathbf{F}\}$	$\{\eta\}$
7	$[-\mid\rightarrow\mu)\neg[\rightarrow\nu,\theta_2\mid\rightarrow)\,\psi$ $(\psi\neq\mathbf{F})$	$\{\mu,\ \mathbf{T}\}$	$\{[\rightarrow\mu\mid\rightarrow)\mathbf{F},$ $\mathbf{T}\}$	$\{[-\mid\rightarrow\mu)\,\nu,$ $[-\mid\rightarrow\mu)\neg[\theta_2\mid\rightarrow)\,\psi\}$	$\{[-\mid\rightarrow\mu)[\rightarrow\nu,\theta_2\mid\rightarrow)\mathbf{F},$ $[-\mid\rightarrow\mu)\mathbf{F}\}$	—	—	$\{\neg\mu,\ \neg[\rightarrow\mu\mid\rightarrow)\mathbf{F},$ $\neg[-\mid\rightarrow\mu)\,\nu,$ $\neg[-\mid\rightarrow\mu)[\rightarrow\nu,\theta_2\mid\rightarrow)\mathbf{F}\}$	$\{\eta,$ $\neg[-\mid\rightarrow\mu)\mathbf{F}\}$
8	$\neg[-\mid\rightarrow\mu)\neg[\rightarrow\nu,\theta_2\mid\rightarrow)\,\psi$ $(\psi\neq\mathbf{F})$	$\{\mu,\ \mathbf{F}\}$	$\{[\rightarrow\mu\mid\rightarrow)\mathbf{F},$ $\mathbf{F}\}$	$\{[-\mid\rightarrow\mu)\,\nu,$ $\neg[-\mid\rightarrow\mu)\neg[\theta_2\mid\rightarrow)\,\psi\}$	$\{[-\mid\rightarrow\mu)[\rightarrow\nu,\theta_2\mid\rightarrow)\mathbf{F},$ $\neg[-\mid\rightarrow\mu)\mathbf{F}\}$	—	—	$\{\neg\mu,\ \neg[\rightarrow\mu\mid\rightarrow)\mathbf{F},$ $\neg[-\mid\rightarrow\mu)\,\nu,$ $\neg[-\mid\rightarrow\mu)[\rightarrow\nu,\theta_2\mid\rightarrow)\mathbf{F}\}$	$\{\eta,$ $\neg[-\mid\rightarrow\mu)\mathbf{F}\}$

Finalmente, obsérvese que en todas y cada una de las reglas que contienen las tablas mostradas en este subapartado (5.1.4.3), siempre se cumple la siguiente relación de equivalencia:

$$\eta \equiv \bigvee_i (\bigwedge new_i(\eta)) \vee (\bigwedge new_c(\eta) \wedge X \bigwedge next_c(\eta))$$

donde $\bigvee_i$ expresa la disyunción de cada uno de los casos presentados en las celdas centrales de la tabla correspondiente, $\bigwedge Y$ denota la conjunción de las fórmulas FIL del conjunto Y (la conjunción del conjunto vacío es considerada como la constante lógica **T**) y X tiene el mismo significado que el operador *siguiente* (*next*) de la LTL, por lo que el término que encabeza representa el conjunto de fórmulas que deben satisfacerse en el próximo estado. Así, esa equivalencia expresa que la fórmula η se satisface si y sólo si se cumple una de las distintas alternativas en las que se ha expandido.

La expresión anterior se puede adaptar para las fórmulas de la Tabla 5.1 y de la Tabla 5.2 mediante la siguiente equivalencia:

$$\eta \equiv \bigwedge new_c(\eta) \vee \bigwedge new_n(\eta)$$

Por lo tanto, de acuerdo con estas dos expresiones de equivalencia, debemos concluir diciendo que en cada paso del proceso de expansión de un nodo se conserva la semántica de la fórmula que se expande.

5.1.5. Ejemplo de ejecución del algoritmo de construcción del grafo

Este apartado tiene como objetivo clarificar, mediante un ejemplo sencillo, cómo actúa la parte del algoritmo presentada hasta ahora, o sea, la primera fase del mismo. La fórmula del ejemplo que se presenta a continuación se ha escogido para que el número de nodos que se crean durante su proceso de expansión sea lo suficientemente pequeño como para poder ser representado gráficamente (ver Figura 5.4). Pero al mismo tiempo, muestra claramente el funcionamiento y los aspectos principales del algoritmo, no sólo de la parte ya explicada, sino también de su segunda fase, que se tratará en el siguiente apartado, por lo que en dicho apartado también se hará referencia a este ejemplo.

Ejemplo 5.11: Ejecución del algoritmo de construcción del grafo para $\varphi = \neg[\rightarrow c \mid \rightarrow)\mathbf{F} = \Diamond c$

Sea la fórmula $\varphi = \neg[\rightarrow c \mid \rightarrow)\mathbf{F} = \Diamond c$, donde c es una proposición atómica que indica que un proceso se encuentra en su región crítica, por lo que φ expresa la eventualidad de que dicho proceso entre en la misma. La Figura 5.4 muestra el árbol de nodos que se procesan durante la ejecución del algoritmo para esta fórmula. En su raíz se encuentra el nodo que contiene en el campo *New* la fórmula φ para la que se quiere construir el autómata de propiedad (obsérvese que este nodo es el que se crea en la Figura 5.1). El significado de las líneas que unen dos nodos es el siguiente: las discontinuas indican que un nodo (el superior) se ha dividido en varios (los inferiores), mientras que las continuas con puntas de flecha representan que el nodo inferior es sucesor del superior (obsérvese que en estos casos el de abajo tiene en su campo *Incoming* el identificador del nodo superior y en su campo *New* el mismo conjunto de fórmulas que el de arriba en su campo *Next*). Cuando un nodo se divide en varios, el situado más a la derecha de los nodos resultantes representa la actualización del nodo superior (repárese en que el identificador de ambos nodos es el mismo), por lo que, para resaltar que no se trata en realidad de un nuevo nodo, se dibuja con línea discontinua y en otro color. Las dos ocasiones en que se divide un nodo en la figura coincide con el análisis de la fórmula φ, por lo que la regla de expansión que se aplica es la que se muestra en la fila 23 de la Tabla 5.3. Así, al campo *New* del nuevo nodo que se crea (situado abajo a la izquierda) se le añade el único reductor de φ, o sea, c (recuérdese que, al ser $\mathbf{T}$ el reducto, éste no se añade por motivos de eficiencia), mientras que al nodo que representa la actualización del superior (situado abajo a la derecha) se le añade la negación de todos sus reductores (en este caso, sólo $\neg c$) al campo *New* y la propia φ a su campo *Next*.

Los números que aparecen en la esquina superior derecha de cada nodo de la Figura 5.4 indican el orden de expansión de ese nodo. Dos números en un mismo nodo significa que dicho nodo se expande dos veces, debido a que la primera vez que se ejecuta la función **expand** (presentada en la Figura 5.2) para el mismo, se produce una actualización de ese nodo, subrayándose los cambios realizados, de modo que en la siguiente llamada a dicha función, se expande el nodo resultante de la citada actualización (de ahí que el segundo de sus números también esté subrayado). Mirando estos números, se puede comprobar que se utiliza la estrategia de BPP. Así, el nodo 1 se divide en dos (el nodo 2 y su actualización), procesándose primero el nodo 2 y todos sus sucesores antes de procesar la actualización del nodo 1 y todos los suyos.

Cuando uno de los números que indican el orden de expansión de un nodo se encierra dentro de un círculo en la Figura 5 .4, significa que en ese punto de la ejecución *se añade un nuevo nodo* (el que se está procesando) al conjunto *GraphNodes* (línea 7 de la Figura 5.2); mientras que si uno de estos números aparece precedido del símbolo ^ significa que en ese momento de la ejecución el nodo en cuestión ya forma parte de *GraphNodes* (se ha almacenado en un paso previo), por lo que se actualiza el campo *Incoming* del nodo almacenado (línea 5 de la Figura 5.2), lo que equivale a *añadir un arco* entrante al mismo. Se puede comprobar que, cuando la función **expand** acaba de ejecutarse, el conjunto *GraphNodes* tiene para φ el contenido que se expone en la Tabla 5. 7 y cuya representación gráfica se muestra en la Figura 5.5.

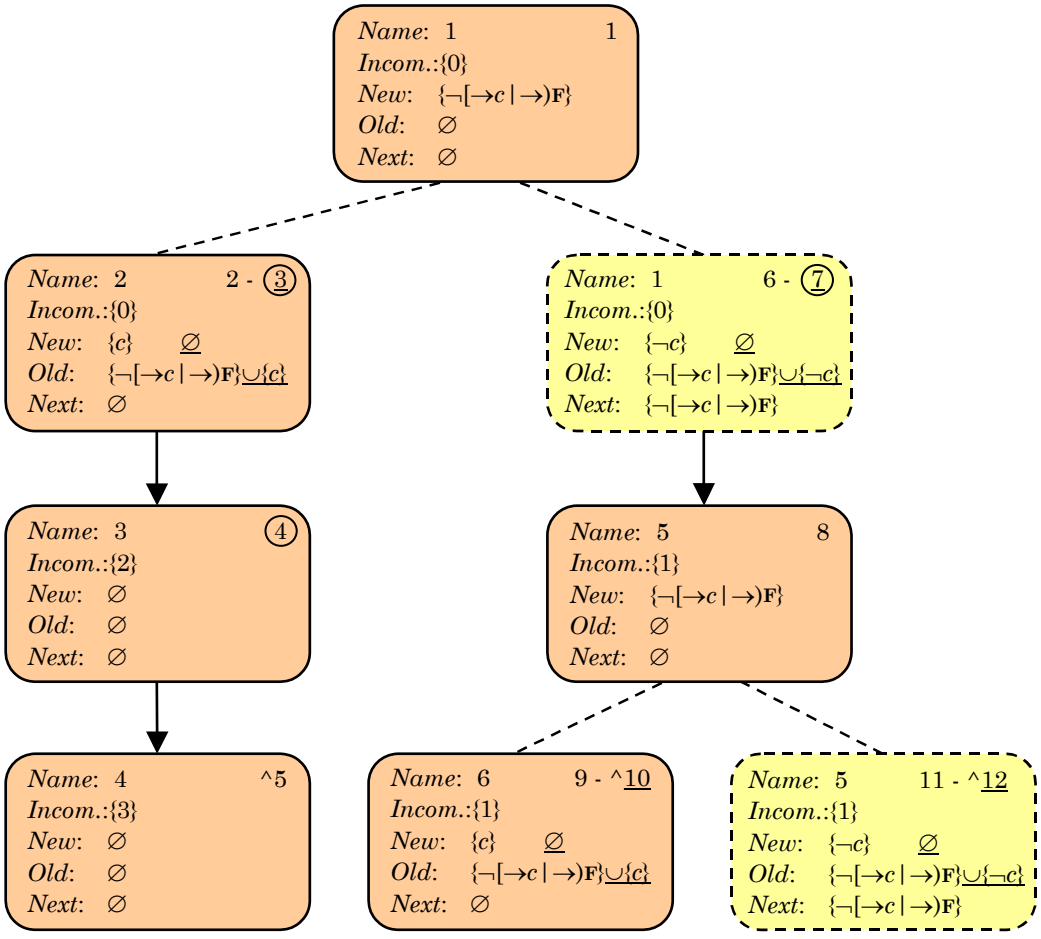


Figura 5.4. Nodos que procesa el algoritmo para $\varphi = \neg[\rightarrow c \mid \rightarrow)\mathbf{F} = \Diamond c$, siguiendo la estrategia de BPP

Tabla 5.7. Contenido de *GraphNodes* para $\varphi = \neg[\rightarrow c \mid \rightarrow)\mathbf{F} = \Diamond c$

Name	*Incoming*	*New*	*Old*	*Next*
2	{0, 1}	∅	{$\neg[\rightarrow c \mid \rightarrow)\mathbf{F}$, c}	∅
3	{2, 3}	∅	∅	∅
1	{0, 1}	∅	{$\neg[\rightarrow c \mid \rightarrow)\mathbf{F}$, $\neg c$}	{$\neg[\rightarrow c \mid \rightarrow)\mathbf{F}$}

Obsérvese que en la Tabla 5.7 todos los nodos del conjunto *GraphNodes* tienen vacío el campo *New*, lo que significa que el algoritmo sólo devuelve nodos totalmente expandidos. El grafo mostrado en la Figura 5.5 se obtiene a partir de la Tabla 5.7 del siguiente modo: Las transiciones vienen dadas por el campo *Incoming* de los distintos nodos, de tal modo que para cada identificador $r \in Incoming(s)$ se dibuja una transición del tipo $r \rightarrow s$. Si el identificador $0 \in Incoming(s)$, entonces s es un nodo inicial (caso de los nodos 1 y 2, que se marcan con el símbolo > en la Figura 5.5). La etiqueta asociada a cada nodo del grafo es la conjunción de todos los literales almacenados en su campo *Old*. Cuando un nodo no contiene ningún literal en este campo, como es el caso del nodo 3, entonces se etiqueta con el valor **T**, que representa que cualquier combinación de literales es admisible (se satisface) en él.

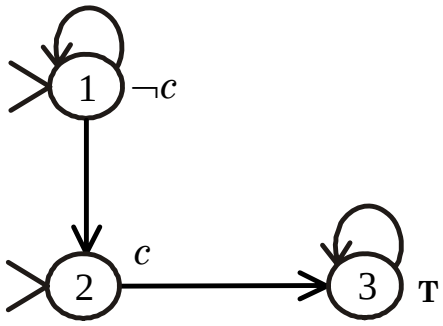


Figura 5.5. Grafo generado por el algoritmo para $\varphi = \neg[\rightarrow c \mid \rightarrow)\mathbf{F} = \Diamond c$

5.1.6. Transformación del grafo en un autómata de Büchi

El grafo construido hasta el momento (en la primera fase del algoritmo) permite que algunas ejecuciones produzcan secuencias que no satisfacen la especificación φ (por ejemplo, en el grafo de la Figura 5.5, una ejecución que dé vueltas infinitamente alrededor del nodo 1). Esto es debido a que muchas fórmulas de intervalo pueden aplazar para siempre su cumplimiento. Dicho de otro modo, hasta aquí sólo se ha

construido el grafo o sistema de transición que corresponde a una fórmula FIL, no el autómata de Büchi equivalente a la misma, puesto que aún no se han determinado cuáles son sus estados de aceptación. Por tanto, ese grafo es incapaz de discernir si una determinada ejecución es de aceptación o no, siendo necesario transformarlo en un autómata de Büchi.

La Definición 4.1 describe los autómatas de Büchi en su representación más habitual, o sea, aquella en la que se etiquetan los arcos o transiciones (ver Figura 4.1). Sin embargo, obsérvese que en la Figura 5.5 son los nodos o estados, y no los arcos, los que aparecen etiquetados. Por esta razón, se incluye a continuación la siguiente definición, donde las etiquetas se asignan a los estados en lugar de a las transiciones.

Definición 5.23: Autómata de Büchi etiquetado

Un *autómata de Büchi etiquetado*, A, se define mediante una sextupla $A=(\Sigma,S,R,L,I,F)$, donde Σ, S, I y F tienen el mismo significado que en la Definición 4.1 y donde los nuevos elementos se definen como sigue:

- $R\subseteq S\times S$ es la *relación de transición*, que empareja o relaciona dos estados si y sólo si existe una transición entre ambos. Por tanto, si $(r,s)\in R$ significa que hay un arco o transición que parte del estado r y llega hasta el estado s.
- $L:S\to\Sigma$ es la *función de etiquetado*, que asigna a cada estado un elemento del alfabeto. ■

La definición anterior y la Definición 4.1 son exactamente equivalentes, pudiéndose utilizar ambas alternativas para describir el mismo comportamiento (expresado como un autómata de Büchi). Así, si se comparan ambas definiciones, puede apreciarse que la función de transición ρ de la Definición 4.1 se ha sustituido en la Definición 5.23 por la relación de transición R y por la función de etiquetado L. De este modo, existe una biyección entre ambas definiciones, tal que si $x\in\Sigma$ y $r, s \in S$, entonces $s\in\rho(r, x)$, según la Definición 4.1, si y sólo si $(r, s)\in R$ y $L(s)=x$, según la Definición 5. 23. Por tanto, habrá que adaptar la definición de *ejecución* para el autómata de Büchi que se acaba de definir, realizando el correspondiente cambio de notación:

Definición 5.24: Ejecución de un autómata de Büchi etiquetado

Una *ejecución de un autómata de Büchi etiquetado* sobre una palabra infinita $\xi \in \Sigma^{\omega}$ es una ω-traza $\sigma \in S^{\omega}$, tal que $\sigma(0) \in I$ y $\forall i \in \omega$, $(\sigma(i), \sigma(i+1)) \in R$ y $L(\sigma(i+1)) = \xi(i)$. ■

Sin embargo, la Definición 4.3: Eje cución de aceptación en un autómata de Büchi se puede aplicar tal cual está a la nueva variante de autómata de Büchi definida.

Los autómatas de Büchi del nuevo tipo, al igual que los definidos en el capítulo anterior, son *no deterministas* si dado un estado y un determinado símbolo de entrada, tienen más de un estado siguiente. Con la nueva notación, esto se expresa formalmente como sigue:

Definición 5.25: No determinismo de un autómata de Büchi etiquetado

Un *autómata de Büchi etiquetado* es *no determinista* si tiene dos estados $s_1, s_2 \in S$, tal que $L(s_1) = L(s_2)$, de modo que se cumple una de estas dos condiciones:

(1) $s_1, s_2 \in I$, o

(2) Hay al menos dos transiciones $(r, s_1), (r, s_2) \in R$ desde el mismo estado $r \in S$. ■

Los autómatas de Büchi hasta aquí definidos (de una forma u otra) se adaptan bien para modelar el sistema que se pretende verificar. Así, en el autómata del sistema es suficiente con etiquetar cada estado (o transición) con un único elemento del alfabeto Σ, que viene dado por $2^{\mathscr{P}}$, donde $\mathscr{P}$ es el conjunto de proposiciones atómicas del sistema. Por tanto, cada estado (o transición) se corresponde con una única asignación de valores de verdad a las proposiciones atómicas del conjunto $\mathscr{P}$. El autómata de propiedad que tratamos de construir también se define sobre el mismo alfabeto. Sin embargo, en la práctica, a menudo resulta bastante ineficiente utilizar un autómata de propiedad que asocie a cada estado una única asignación a las variables proposicionales del conjunto $\mathscr{P}$. Por este motivo, nuestro algoritmo, con el fin de obtener una representación más simple y más pequeña del autómata de propiedad, combina varios estados en uno: aquéllos que, conteniendo el mismo conjunto de literales, tienen los mismos sucesores (líneas 3-4 de la Figura 5.2). De esta forma, se logra una representación más compacta del autómata generado. Por consiguiente, debemos redefinir la noción de autómata de Büchi, con el fin de

adaptarla a la que realmente utiliza nuestro algoritmo. El único cambio que se debe realizar, con respecto a la Definición 5.23, es que ahora cada estado se debe corresponder con varias asignaciones sobre las variables proposicionales del conjunto $\mathscr{P}$.

Definición 5.26: Autómata de Büchi etiquetado compacto

Un *autómata de Büchi etiquetado compacto*, A, también se define mediante una sextupla $A=(\Sigma,S,R,L,I,F)$, donde todos sus elementos tienen el mismo significado que en la Definición 5.23, a excepción de la *función de etiquetado*, que se define como sigue:

- $L:S\rightarrow 2^{\Sigma}$, esto es , se asigna a cada estado un conjunto de elementos del alfabeto. ■

Obsérvese que con esta redefinición no se logra extender el lenguaje que un autómata puede reconocer, sino simplemente una representación más compacta, razón por la que se le ha denominado de esta forma.

También necesitamos redefinir la noción de ejecución, con el fin de adaptarla al nuevo tipo de autómata de Büchi definido. Sin embargo, la noción de ejecución de aceptación se puede utilizar tal y como se especificó en la Definición 4.3.

Definición 5.27: Ejecución de un autómata de Büchi etiquetado compacto

Una *ejecución de un autómata de Büchi etiquetado compacto* sobre una palabra infinita $\xi\in\Sigma^{\omega}$ es una ω-traza $\sigma\in S^{\omega}$, tal que $\sigma(0)\in I$ y $\forall i\in\omega$, $(\sigma(i),\sigma(i+1))\in R$ y $\xi(i)\in L(\sigma(i+1))$. ■

Para esta clase de autómatas de Büchi el *no determinismo* se define formalmente como sigue:

Definición 5.28: No determinismo de un autómata de Büchi etiquetado compacto

Un *autómata de Büchi etiquetado compacto* es *no determinista* si tiene al menos dos estados s_1, $s_2\in S$ para los que $\exists\ x\in\Sigma$, tal que $x\in L(s_1)$ y $x\in L(s_2)$, es decir, el

mismo símbolo del alfabeto está incluido en las etiquetas de esos estados, de modo que se cumple una de las dos condiciones enunciadas en la Definición 5.25, y que son:

(1) $s_1, s_2 \in I$, o

(2) Hay al menos dos transiciones $(r, s_1),(r, s_2) \in R$ desde el mismo estado $r \in S$. ■

De ahora en adelante, y salvo que se indique lo contrario, denominaremos autómata de Büchi (sin más) al definido en último lugar, sin que haya lugar a confusión ni ambigüedad posible, dado que el autómata generado por nuestro algoritmo se ajusta mejor a dicha definición, como ya se ha comentado.

Obsérvese que, en la Figura 5.5, los nodos no se han etiquetado tal y como indica la Definición 5.26 . Sea *etiq*(*s*) la etiqueta que muestra el nodo *s* en dicha figura. Recuérdese que a cada nodo se le asignó como etiqueta la conjunción de los literales almacenados en su campo *Old*. En la Figura 5.5 se ha utilizado este etiquetado alternativo y equivalente para una mejor comprensión del grafo dibujado y por motivos de simplicidad, basándonos en el hecho de que los nodos devueltos por el algoritmo (almacenados en el conjunto *GraphNodes*) no asignan explícitamente valores de verdad a todas las proposiciones atómicas del sistema. Sin embargo, y dado que el grafo pretende representar un conjunto de interpretaciones temporales, que en cada estado asignan valores de verdad a cada proposición atómica del conjunto $\mathscr{P}$, en realidad y de acuerdo con la función de etiquetado L de la Definición 5.26, la etiqueta de cada nodo, $L(s)$, estaría formada por el conjunto de todos los elementos del alfabeto $\Sigma = 2^{\mathscr{P}}$ (subconjuntos de $\mathscr{P}$) que satisfagan la conjunción expresada por la etiqueta *etiq*(*s*). Esto se puede expresar de manera formal del siguiente modo: $L(s) = \{X \mid X \subseteq \mathscr{P} \wedge Pos(s) \subseteq X \wedge Neg(s) \cap X = \varnothing\}$, donde $Pos(s)$ y $Neg(s)$ representan respectivamente las ocurrencias positivas y negativas de las proposiciones atómicas en ese nodo. Ambos conjuntos se definen formalmente como: $Pos(s) = Old(s) \cap \mathscr{P}$ y $Neg(s) = \{p \mid \neg p \in Old(s) \wedge p \in \mathscr{P}\}$. Así, cada elemento $X \in L(s)$ es un conjunto de proposiciones atómicas que debe interpretarse como la asignación del valor **T** a las proposiciones atómicas $p \in X$ y como la asignación del valor **F** a las proposiciones atómicas $p \notin X$.

Dado que cada estado se etiqueta con un conjunto de subconjuntos de $\mathscr{P}$ (elementos del alfabeto), cada arco dibujado $(r, s) \in R$ puede representar varias transiciones del estado *r* al estado *s*, una por cada uno de los elementos del alfabeto

que pertenecen a $L(s)$. El siguiente ejemplo ilustra claramente lo que se acaba de expresar.

Ejemplo 5.12: Interpretación de las etiquetas de los nodos del grafo generado

Sea $\mathscr{P}=\{a, b, c\}$ y $(r, s)\in R$ uno de los arcos del grafo generado por nuestro algoritmo. Supongamos que $etiq(s) = a\wedge\neg c$, entonces sólo hay dos asignaciones de valores de verdad a los elementos de $\mathscr{P}$ que satisfacen dicha conjunción, que son $\{a\}$ y $\{a, b\}$ (conjuntos de proposiciones que incluyen a y no incluyen c, pero que pueden o no contener a b), por lo que $L(s) = \{\{a\}, \{a, b\}\}$. Por consiguiente, el arco (r, s) representa dos posibles transiciones, una por cada una de estas asignaciones, o dicho de otro modo, una por cada uno de los elementos del conjunto $L(s)$.

Por tanto, las etiquetas de los nodos del grafo de la Figura 5.5, según la Definición 5.26 y considerando el conjunto $\mathscr{P}$ especificado al principio de este ejemplo, serían: $L(1) = \{\varnothing, \{a\}, \{b\}, \{a, b\}\}$, $L(2) = \{\{c\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$ y $L(3) = \{\varnothing, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$.

En este contexto, el *no determinismo* de un autómata de Büchi, se puede expresar equivalentemente utilizando las etiquetas $etiq(s)$ en lugar de $L(s)$, que es lo que se emplea en la Definición 5.28. Así, se sustituiría la primera parte de esa definición por la siguiente expresión: en el autómata existen al menos dos estados $s_1, s_2 \in S$, tal que $etiq(s_1) \wedge etiq(s_2) \neq \mathbf{F}$ (esto es, la conjunción de sus etiquetas no equivale a la constante lógica **F**, o dicho de otro modo, no conlleva una contradicción), de manera que se cumple una de las dos condiciones enunciadas en la citada definición.

En resumen, el objetivo de este apartado es la obtención de un autómata de propiedad que se ajuste a la Definición 5.2 6. Hasta el momento, a partir de la estructura devuelta por el algoritmo presentado en el apartado 5.1.3. Algoritmo de construcción del grafo, o sea, a partir de *GraphNodes*, sólo se han concretado los cinco primeros elementos de la sextupla que define al autómata de Büchi que se desea construir. Estos elementos se obtienen del siguiente modo:

- $\Sigma = 2^{\mathscr{P}}$, donde $\mathscr{P}$ es el conjunto de proposiciones atómicas considerado
- $S = \{s \mid s \in GraphNodes\}$

- $(r, s) \in R$ sii $r \in Incoming(s)$
- $x \in L(s)$ sii $x \models etiq(s)$, donde $x \in \Sigma$
- $I = \{s \mid 0 \in Incoming(s)\}$

Por tanto, sólo nos falta determinar el último de sus componentes, es decir, el conjunto F, que contiene sus estados de aceptación. De esto nos encargamos en los distintos subapartados que se presentan a continuación. Tal y como se verá en ellos, al hallar este sexto componente para muchos de los grafos generados por nuestro algoritmo, no se obtiene un único conjunto de estados de aceptación, como se indica en la Definición 5.26, sino varios. En estos casos se dice que el autómata de Büchi obtenido es un *autómata de Büchi generalizado*, y se define como sigue:

Definición 5.29: Autómata de Büchi generalizado

Un *autómata de Büchi generalizado*, G, se define mediante una sextupla $G = (\Sigma, S, R, L, I, \mathcal{F})$, donde los cinco primeros componentes se corresponden exactamente con los de la Definición 5.26, siendo el último de ellos como se define a continuación:

- $\mathcal{F} \subseteq 2^S$ es un *conjunto de conjuntos de estados de aceptación*, o sea, $\mathcal{F} = \{F_1, F_2, \ldots, F_k\}$, donde cada $F_i \subseteq S$ es un conjunto de estados de aceptación. $\mathcal{F}$ puede ser vacío, en cuyo caso todas las palabras infinitas (sobre el alfabeto Σ) para las que se produzca una ejecución del autómata G son aceptadas. ■

La noción de *ejecución* para un autómata de Büchi generalizado se corresponde con la expresada en la Definición 5.27. Sin embargo, debe redefinirse el concepto de *ejecución de aceptación* para estos nuevos autómatas:

Definición 5.30: Ejecución de aceptación sobre un autómata de Büchi generalizado

Una ejecución $\sigma \in S^\omega$ sobre un autómata de Büchi generalizado, G, es de aceptación si para cada conjunto de estados de aceptación $F_i \in \mathcal{F}$ de G existe al menos un estado $s \in F_i$ que aparece infinitamente a menudo sobre σ. Esta condición se puede expresar más formalmente del siguiente modo: una ejecución σ sobre G es de aceptación si y sólo si $\forall F_i \in \mathcal{F}$ se cumple que $inf(\sigma) \cap F_i \neq \emptyset$ (recuérdese que $inf(\sigma)$

designa al conjunto de estados que aparecen infinitamente a menudo en la ejecución σ). ■

Esta definición de condición de aceptación para los autómatas de Büchi generalizados no aumenta la expresividad de los autómatas de Büchi clásicos o simples (los que se ajustan a la Definición 5.26). De hecho, en el subapartado 5.1.6.4 se verá cómo se puede convertir un autómata de Büchi generalizado en un autómata de Büchi clásico que acepta el mismo lenguaje.

5.1.6.1. FÓRMULAS DE EVENTUALIDAD COMO CLAVE PARA HALLAR LOS ESTADOS DE ACEPTACIÓN

Una vez construido el grafo que representa todos los estados y transiciones del autómata que se pretende generar, y dado que, en muchas ocasiones, no todas las ejecuciones del mismo pueden ser aceptadas, es necesario determinar los estados que deben ser de aceptación. Para ello, lo primero que se debe hacer es buscar en el conjunto *GraphNodes* (que representa a ese grafo) todas las *fórmulas de eventualidad* (FEs) que se han generado durante el proceso de expansión de la fórmula φ (introducida como entrada al algoritmo), que constituye la especificación para la que se desea obtener el autómata de Büchi equivalente.

Se deben considerar las FEs porque dichas fórmulas pueden posponer indefinidamente su cumplimiento (motivo por el que los estados que representen ese comportamiento no deben ser de aceptación), y *sólo* ese tipo de fórmulas debido a las siguientes razones:

(1) El único operador temporal que tiene FIL es el de intervalo. Aunque el traductor que hemos implementado, al que hemos denominado FBT (*FIL to Büchi automaton Translator*) y del que se dan más detalles en el Capítulo 6 y en el Apéndice C, también reconoce las fórmulas con operadores temporales de la LTL, éstas son consideradas únicamente como abreviaciones de las correspondientes fórmulas de intervalo FIL, que son las que realmente se almacenan y procesan en FBT.

(2) La componente de eventualidad de una fórmula de intervalo, durante el proceso de expansión de la misma, se *reduce* a (se expresa como) la(s) correspondiente(s) FE(s).

(3) Si una fórmula de intervalo ϕ es una subfórmula (operando) de una FPM o está anidada a una secuencia de modalidades de intervalo cuyo segundo patrón de búsqueda es el trivial o forma parte de la fórmula objetivo de algún patrón de búsqueda, el proceso de expansión garantiza que finalmente se analizará dicha fórmula ϕ de manera individualizada.

Así, una FE ε que se encuentre en el conjunto *GraphNodes* estará ahí debido a que se incluye en él durante el proceso de expansión de φ, como consecuencia de que se dé uno de los siguientes casos:

a) Es la propia fórmula introducida, o sea, $\varphi = \varepsilon$.

b) Es una subfórmula de φ, entendiéndose esto cuando ocurre uno de los siguientes subcasos:

- Es un operando de una FPM ψ (que puede ser φ o cualquiera de sus subfórmulas). Ejemplo: $\psi = f \wedge \varepsilon$.
- Es la fórmula objetivo (o su negación) de alguna de las búsquedas que forman parte de los patrones de búsqueda de alguna fórmula de intervalo ϕ (que puede estar o no anidada a otra fórmula de intervalo). Ejemplo: $\phi = [\theta_1 \mid \theta_2)f$, donde $\varepsilon \in \theta_1$ o $\varepsilon \in \theta_2$ o $\neg\varepsilon \in \theta_1$ o $\neg\varepsilon \in \theta_2$.
- Es el reducto que se obtiene de una fórmula de intervalo ϕ, formada por una secuencia de una o más modalidades de intervalo cuyo segundo patrón de búsqueda es el trivial, cuando se satisface la última de las búsquedas de su intervalo más interno. Ejemplo: $\phi = \neg[\theta_1 \mid \rightarrow)[\theta_2 \mid \rightarrow)\neg\varepsilon$.
- Resulta de la aplicación sucesiva y/o combinación de los subcasos anteriores tantas veces como sea necesario. Ejemplo: $\varphi = [\theta_1 \mid \theta_2)[\rightarrow\psi, \theta_3 \mid \theta_4)f_1$, donde $\psi = f_2 \Rightarrow [\theta_5 \mid \rightarrow)[\theta_6 \mid \rightarrow)\varepsilon \vee f_3$.

c) Es la negación del reductor que expresa el fallo de uno de los patrones de búsqueda de una fórmula de intervalo ϕ, pudiendo ser ésta tanto la propia φ como alguna de sus subfórmulas. Ejemplo: Si $\phi = [\theta_1 \mid \theta_2)f$, entonces $\varepsilon_1 = \neg[\theta_1 \mid \rightarrow)\mathbf{F}$ y $\varepsilon_2 = \neg[\theta_2 \mid \rightarrow)\mathbf{F}$.

d) Es el reducto obtenido a partir de otra fórmula de eventualidad ε' previamente existente. Ejemplo: $\varepsilon' = \neg[\rightarrow f, \theta_1 \mid \rightarrow)\mathbf{F}$ y $\varepsilon = \neg[\theta_1 \mid \rightarrow)\mathbf{F}$.

5.1.6.2. ESTABLECIMIENTO DE LOS CONJUNTOS O CONDICIONES DE ACEPTACIÓN

La Figura 5.6 muestra el pseudocódigo de la función que calcula los conjuntos de aceptación mencionados en la Definición 5.29 . Como se ha comentado en el subapartado anterior, lo primero que se ha de hacer es buscar todas las FEs que haya en el conjunto *GraphNodes*, pero esas fórmulas no se buscan en todos los campos del conjunto *GraphNodes*, sino sólo en su campo *Next* (líneas 3-6 de la Figura 5.6). La razón para ello es que si una fórmula de eventualidad no se reduce (es decir, no se satisface total o parcialmente) en un estado, entonces se pasa al campo *Next* de ese nodo para ser analizada en sus sucesores inmediatos. Y dado que el proceso de expansión toma en consideración no sólo los casos en los que se cumple algún reductor de la fórmula analizada, sino también el caso en que ninguno de ellos se da, debemos concluir que cualquier fórmula de eventualidad procesada durante la expansión de φ (esto es, contenida en el conjunto *GraphNodes*) estará en el conjunto formado por los campos *Next* de todos los nodos de *GraphNodes*.

```
1  function calculateAcceptanceSets (GraphNodes)
2      Eventualities ← ∅;
3      for each Node ∈ GraphNodes do
4          for each η ∈ Next(Node) do
5              if η is an eventuality formula then          # η es una fórmula del tipo ¬[θ | →)F
6                  Add η to Eventualities;
7      AcceptanceSets ← ∅;   Identifier ← 0;
8      for each ε ∈ Eventualities do          # Recuérdese que ε es una fórmula del tipo ¬[θ | →)F
9          if ∃ AcceptSet ∈ AcceptanceSets with Key(AcceptSet) = lastSearch(θ)
10             then Add ε to EventualityFormulas(AcceptSet);
11             else Add (lastSearch(θ), Identifier, {ξ}) to AcceptanceSets;
12                  Identifier ← Identifier + 1;
13     return (AcceptanceSets);
```

Figura 5.6. Función que calcula los conjuntos o condiciones de aceptación

El conjunto de FEs (denominado *Eventualities* en la Figura 5.6) así determinado sirve de base para establecer las *condiciones de aceptación de Büchi generalizadas*, o sea, los conjuntos de estados de aceptación que se deben considerar para convertir

el grafo obtenido a partir de la fórmula φ en el autómata de Büchi equivalente a la misma. Así, se define un conjunto de estados de aceptación para cada conjunto de FEs que tengan en común su última búsqueda (líneas 7-12 de la Figura 5.6, donde la función **lastSearch** devuelve la fórmula objetivo de la última búsqueda del patrón de búsqueda que lleva como parámetro). La razón para hacer esto es que una FE no se cumplirá totalmente hasta que lo haga su última búsqueda. Además, una FE que tenga n búsquedas (con n>1) en su patrón izquierdo (por ejemplo: $\neg[\rightarrow$`p0`,$\rightarrow$`p1`,$\rightarrow$`p2`$\mid\rightarrow)$**F**) genera n–1 subfórmulas de eventualidad (en ese mismo ejemplo: $\neg[\rightarrow$`p1`,$\rightarrow$`p2`$\mid\rightarrow)$**F** y $\neg[\rightarrow$`p2`$\mid\rightarrow)$**F**), por reducción de dicha fórmula y de sus sucesivos reductos, tal y como indica el caso d) expuesto en el subapartado 5.1.6.1. Todas ellas deben pertenecer al mismo conjunto o condición de aceptación.

La estructura de datos elegida para contener los conjuntos de aceptación (denominada *AcceptanceSets* en la Figura 5.6) es un diccionario o mapa (*map*) con tres campos (ver Figura 5.7), siendo el primero de ellos su campo clave (denominado *Key* en la Figura 5.6 y que, por definición, no admite duplicados), donde se almacena la fórmula objetivo de la última búsqueda común a todas las eventualidades que forman parte de ese conjunto de aceptación. El segundo campo alberga el número o identificador del conjunto de aceptación (*Identifier* en la Figura 5.6), mientras que en el último (referenciado como *EventualitiesFormulas* en la Figura 5.6) se meten todas las fórmulas del conjunto *Eventualities* cuya última búsqueda sea la que indica su campo clave. Tanto el Ejemplo 5 .13 como el Ejemplo 5 .14, que se muestran a continuación, tratan de aportar una mayor intuición acerca de lo que se acaba de explicar.

Clave: última búsqueda	Identificador	{Fórmulas de Eventualidad}

Figura 5.7. Estructura donde se almacena un conjunto o condición de aceptación

Ejemplo 5.13: Obtención de las condiciones de aceptación para $\varphi = \neg[\rightarrow c \mid \rightarrow)$**F**

Mirando la Tabla 5.7 , que muestra el contenido de *GraphNodes* para la fórmula $\varphi = \neg[\rightarrow c \mid \rightarrow)$**F**, claramente se observa que la única FE que hay en el campo *Next* es la propia φ (en el nodo 1), por lo que sólo se generará un conjunto de estados de aceptación, el que se muestra en la Figura 5.8.

c	0	$\{\neg[\rightarrow c \mid \rightarrow)\mathbf{F}\}$

Figura 5.8. Único conjunto de aceptación para $\varphi = \neg[\rightarrow c \mid \rightarrow)\mathbf{F}$

Ejemplo 5.14: Obtención de los conjuntos de aceptación para $\varphi = \neg[\rightarrow \mathtt{p0} \mid \rightarrow)\mathbf{F} \vee \neg[\rightarrow \mathtt{p1}, \rightarrow \mathtt{p2} \mid \rightarrow)\mathbf{F}$

Para la fórmula $\varphi = \neg[\rightarrow \mathtt{p0} \mid \rightarrow)\mathbf{F} \vee \neg[\rightarrow \mathtt{p1}, \rightarrow \mathtt{p2} \mid \rightarrow)\mathbf{F}$, se almacena en *Eventualities* el siguiente conjunto de FEs: $\{\neg[\rightarrow \mathtt{p0} \mid \rightarrow)\mathbf{F}$, $\neg[\rightarrow \mathtt{p1}, \rightarrow \mathtt{p2} \mid \rightarrow)\mathbf{F}$, $\neg[\rightarrow \mathtt{p2} \mid \rightarrow)\mathbf{F}\}$. A partir de él se genera el mapa que se muestra en la Figura 5.9.

p0	0	$\{\neg[\rightarrow \mathtt{p0} \mid \rightarrow)\mathbf{F}\}$
p2	1	$\{\neg[\rightarrow \mathtt{p1}, \rightarrow \mathtt{p2} \mid \rightarrow)\mathbf{F}, \neg[\rightarrow \mathtt{p2} \mid \rightarrow)\mathbf{F}\}$

Figura 5.9. Conjuntos de aceptación para $\varphi = \neg[\rightarrow \mathtt{p0} \mid \rightarrow)\mathbf{F} \vee \neg[\rightarrow \mathtt{p1}, \rightarrow \mathtt{p2} \mid \rightarrow)\mathbf{F}$

5.1.6.3. DETERMINACIÓN DE LOS ESTADOS DE ACEPTACIÓN

Finalmente, sólo queda determinar qué estados pertenecen a cada uno de los conjuntos de aceptación que se han calculado, teniendo en cuenta que *un estado cumple las condiciones de aceptación* impuestas por un determinado conjunto de aceptación *sólo*:

- Si en él no se tienen que satisfacer ninguna de las FEs que pertenecen a dicho conjunto.
- O, en caso contrario (es decir, ese estado ha de satisfacer alguna(s) de ellas), si inmediatamente se cumplen (esto es, se reducen en dicho estado).

Ambas condiciones se comprueban rápida y fácilmente buscando de nuevo en el campo *Next* de cada nodo, para ver si en él aparece alguna de las FEs de cada conjunto de aceptación. Si es así, entonces ese estado no pertenece al conjunto de aceptación correspondiente, perteneciendo al mismo en caso contrario. La razón es que si no encuentra ninguna de las FEs pertenecientes a un conjunto de aceptación en el campo *Next* de un nodo, se puede concluir que en ese estado se cumple una de las dos condiciones de aceptación mostradas anteriormente, mientras que si se

encuentra alguna de ellas, rápidamente se infiere que la primera condición no es aplicable y que la segunda no se da.

Los dos ejemplos siguientes muestran respectivamente cómo se determinan los estados de aceptación para cada uno de los dos ejemplos anteriores:

Ejemplo 5.15: Determinación de los estados de aceptación para $\varphi = \neg[\rightarrow c \mid \rightarrow)\mathbf{F}$

En el Ejemplo 5.13 se determinó que para la fórmula $\varphi = \neg[\rightarrow c \mid \rightarrow)\mathbf{F}$ sólo había un conjunto de estados de aceptación (mostrado en la Figura 5.8), que contiene sólo a la propia φ en su tercer campo. Por tanto, lo único que hay que hacer ahora es buscar en el campo *Next* de la Tabla 5.7 la fórmula φ. Así, se determina que en el grafo de la Figura 5.5 hay dos estados de aceptación (nodos 2 y 3), dado que en dicho campo no aparece la FE buscada. Por el contrario, el nodo 3 no es un estado de aceptación, al encontrarse dicha fórmula en su campo *Next*.

Ejemplo 5.16: Determinación de los estados de aceptación para $\varphi = \neg[\rightarrow \mathtt{p0} \mid \rightarrow)\mathbf{F} \vee \neg[\rightarrow \mathtt{p1}, \rightarrow \mathtt{p2} \mid \rightarrow)\mathbf{F}$

La Figura 5.10 representa el autómata de Büchi generalizado que nuestro traductor, FBT, genera para la fórmula del Ejemplo 5.14. El estado inicial es el que aparece sombreado y siempre se numera con el 0. En realidad, se trata de un estado ficticio, que indica que sus inmediatos sucesores son estados iniciales. Obsérvese además que el etiquetado es distinto, pero equivalente, al de la Figura 5.5. Allí se etiquetan los nodos, mientras que aquí (para ofrecer una mayor intuición) se etiquetan las transiciones, de modo que todos los arcos entrantes a un nodo tienen la misma etiqueta, formada por la conjunción de los literales almacenados en el campo *Old* de dicho nodo (se utiliza la notación prefija, representando '&' a la conjunción, '!' a la negación y 't' a la constante lógica **T**). El número superior que etiqueta cada nodo es el identificador de ese estado, mientras que el/los números inferiores identifican a los correspondientes conjuntos de aceptación. Así, puede observarse que los estados 3, 4, 9, 10, 12, 13 y 15 (o sea, todos excepto el estado número 5) pertenecen al primer conjunto de aceptación, identificado con el número 0, mientras que los estados 3, 4, 5, 9 y 12 pertenecen al segundo y último de los conjuntos de aceptación.

Con el fin de que el lector pueda profundizar más en la comprensión de este ejemplo y comprobar lo que se acaba de comentar, así como lo indicado en el Ejemplo 5.14, la Figura 5.11 muestra las fórmulas que forman parte de los distintos campos que integran cada uno de los nodos del conjunto *GraphNodes*, así como el contenido del mapa que almacena los conjuntos de aceptación. Debido a que lo mostrado ha sido generado directamente como salida por FBT (ejecutándolo en modo de depuración para la fórmula de este ejemplo), la sintaxis en que aparecen escritas las fórmulas FIL es la que reconoce nuestra herramienta. Esta sintaxis se explica detallamente en el apartado 6.2.1.

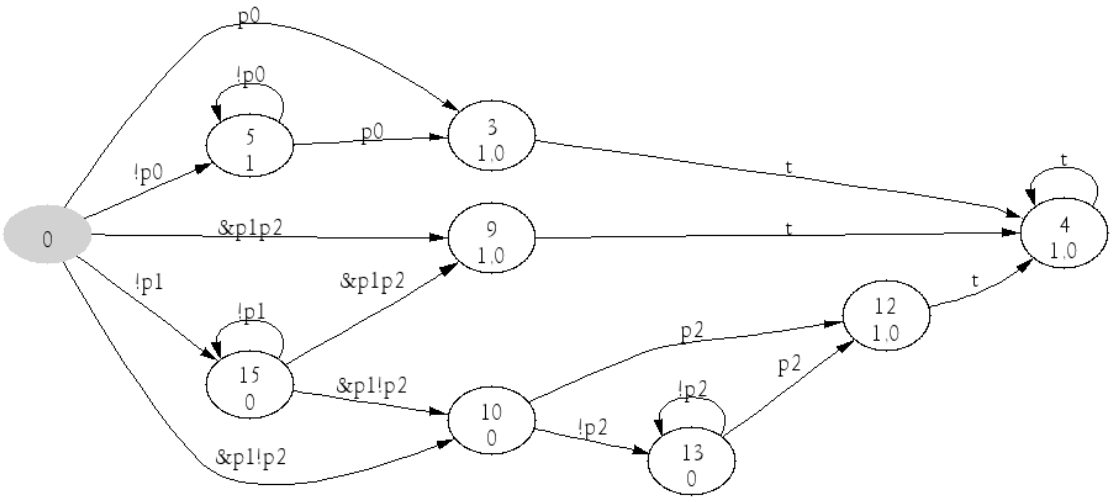


Figura 5.10. Autómata de Büchi generalizado para $\varphi = \neg[\rightarrow \mathtt{p0} \mid \rightarrow)\mathbf{F} \vee \neg[\rightarrow \mathtt{p1}, \rightarrow \mathtt{p2} \mid \rightarrow)\mathbf{F}$

Tal y como se indica en la Definición 5.29, si un autómata no tiene ningún conjunto de aceptación, entonces todas sus ejecuciones son aceptadas. El siguiente ejemplo muestra una fórmula para la que ocurre esto.

Ejemplo 5.17: Autómata con ningún estado de aceptación

La Figura 5.12 muestra el autómata de Büchi generalizado que FBT produce para la fórmula $\varphi = [\rightarrow \mathtt{p0}, \rightarrow \mathtt{p1} \mid \rightarrow)\mathbf{F}$, mientras que la Figura 5.13 presenta el contenido interno de los nodos que lo integran, indicando además que ese autómata no tiene ningún conjunto de estados de aceptación, razón por la que todas sus ejecuciones son aceptadas.

```
Formula:    | ! [ p0 > f ! [ , p1 p2 > f

--------------- NODE 3 ---------------
Incoming:  0 5
Old:       ! [ p0 > f ;   | ! [ p0 > f ! [ , p1 p2 > f ;   p0 ;
Literals:  p0 ;
Next:
--------------- NODE 4 ---------------
Incoming:  3 4 9 12
Old:
Literals:
Next:
--------------- NODE 5 ---------------
Incoming:  0 5
Old:       ! p0 ;   ! [ p0 > f ;   | ! [ p0 > f ! [ , p1 p2 > f ;
Literals:  ! p0 ;
Next:      ! [ p0 > f ;
--------------- NODE 9 ---------------
Incoming:  0 15
Old:       p1 ;   p2 ;   ! [ , p1 p2 > f ;   | ! [ p0 > f ! [ , p1 p2 > f ;
            ! [ p2 > f ;
Literals:  p1 ;   p2 ;
Next:
--------------- NODE 10 ---------------
Incoming:  0 15
Old:       p1 ;   ! [ , p1 p2 > f ;   | ! [ p0 > f ! [ , p1 p2 > f ;
            ! [ p2 > f ;   ! p2 ;
Literals:  p1 ;   ! p2 ;
Next:      ! [ p2 > f ;
--------------- NODE 12 ---------------
Incoming:  10 13
Old:       p2 ;   ! [ p2 > f ;
Literals:  p2 ;
Next:
--------------- NODE 13 ---------------
Incoming:  10 13
Old:       ! [ p2 > f ;   ! p2 ;
Literals:  ! p2 ;
Next:      ! [ p2 > f ;
--------------- NODE 15 ---------------
Incoming:  0 15
Old:       ! [ , p1 p2 > f ;   | ! [ p0 > f ! [ , p1 p2 > f ;   ! p1 ;
Literals:  ! p1 ;
Next:      ! [ , p1 p2 > f ;

                Number of nodes = 8 + initial node
                Number of transitions = 18
                Number of acceptance sets = 2

=============== ACCEPTANCE SETS ===============
Map key:   p2
   Acceptance set number:  1
   Eventuality formulas:   ! [ , p1 p2 > f ;   ! [ p2 > f ;
Map key:   p0
   Acceptance set number:  0
   Eventuality formulas:   ! [ p0 > f ;
```

Figura 5.11. Contenido de los nodos y mapa de aceptación para $\varphi = \neg[\rightarrow \mathtt{p0} \mid \rightarrow)\mathbf{F} \vee \neg[\rightarrow \mathtt{p1}, \rightarrow \mathtt{p2} \mid \rightarrow)\mathbf{F}$

Obsérvese que las etiquetas de todos los nodos del autómata representado en la Figura 5.12 sólo contienen un identificador de estado; dicho de otro modo, ninguna de ellas contiene ningún identificador de conjunto de aceptación. Sin embargo, esta condición no es suficiente para determinar que todas sus ejecuciones son aceptadas,

ya que para eso es necesario saber si el mapa que contiene sus conjuntos de aceptación está vacío (tal y como indica la Figura 5.13 para la fórmula del Ejemplo 5.17) o no. El Ejemplo 5.18 ilustra esto último, con el fin de facilitar la comprensión de lo que se acaba de decir.

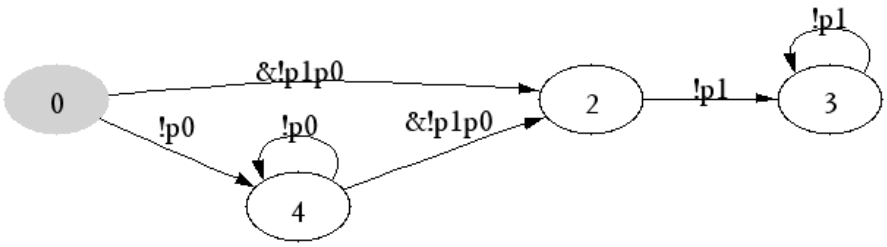


Figura 5.12. Autómata de Büchi generalizado que FBT produce para $\varphi = [\rightarrow \mathtt{p0}, \rightarrow \mathtt{p1} \mid \rightarrow)\mathbf{F}$

```
Formula:    [ , p0 p1 > f

--------------- NODE 2 ---------------
Incoming:  0 4
Old:       [ , p0 p1 > f ;   [ p1 > f ;   ! p1 ;   p0 ;
Literals:  ! p1 ;   p0 ;
Next:      [ p1 > f ;
--------------- NODE 3 ---------------
Incoming:  2 3
Old:       [ p1 > f ;   ! p1 ;
Literals:  ! p1 ;
Next:      [ p1 > f ;
--------------- NODE 4 ---------------
Incoming:  0 4
Old:       [ , p0 p1 > f ;   ! p0 ;
Literals:  ! p0 ;
Next:      [ , p0 p1 > f ;

               Number of nodes = 3 + initial node
               Number of transitions = 6
               Number of acceptance sets = 0
```

Figura 5.13. Contenido de los nodos del autómata de Büchi generado para $\varphi = [\rightarrow \mathtt{p0}, \rightarrow \mathtt{p1} \mid \rightarrow)\mathbf{F}$

Ejemplo 5.18: Autómata con conjuntos de aceptación, pero sin estados de aceptación

La Figura 5.14 muestra el grafo del autómata de Büchi que FBT genera para la fórmula $\varphi = \neg[\rightarrow \mathtt{p1}, \rightarrow \mathtt{p2} \mid \rightarrow \mathtt{p1})\mathtt{p0}$. Al igual que en la Figura 5.12, ningún estado del autómata contiene ningún identificador de conjunto de aceptación, pero esto no nos indica que sus ejecuciones (es decir, cualquier secuencia infinita de $\neg\mathtt{p1}$) sean aceptadas. De hecho, no acepta ninguna ejecución, ya que dicho autómata tiene dos conjuntos de aceptación (ver Figura 5.15) y, como se aprecia en la Figura 5.14, ninguno de sus estados pertenecen a ellos. Esto es coherente con la semántica asociada a la

fórmula φ de este ejemplo, dado que dicha fórmula expresa una contradicción (el operador de negación colocado delante del intervalo indica que éste obligatoriamente debe poder construirse, lo que es imposible, puesto que en el momento en que se dé p1, el intervalo se colapsaría, es decir, su extremo izquierdo no se encontraría estrictamente antes que su extremo derecho), motivo por el que es insatisfacible, esto es, no existe ninguna secuencia que la cumpla.

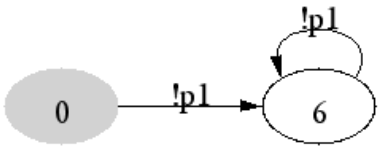


Figura 5.14. Grafo generado por FBT para $\varphi = \neg[\rightarrow p1, \rightarrow p2 \mid \rightarrow p1)p0$

```
Formula:    ! [ , p1 p2 p1 p0

--------------- NODE 6 ---------------
Incoming:  0 6
Old:       ! [ , p1 p2 p1 p0 ;   ! [ , p1 p2 > f ;   ! [ p1 > f ;   ! p1 ;
Literals:  ! p1 ;
Next:      ! [ , p1 p2 p1 p0 ;   ! [ , p1 p2 > f ;   ! [ p1 > f ;

             Number of nodes = 1 + initial node
             Number of transitions = 2
             Number of acceptance sets = 2

=============== ACCEPTANCE SETS ===============
Map key:   p1
   Acceptance set number:  1
   Eventuality formulas:   ! [ p1 > f ;
Map key:   p2
   Acceptance set number:  0
   Eventuality formulas:   ! [ , p1 p2 > f ;
```

Figura 5.15. Contenido de los nodos y estados de aceptación para $\varphi = \neg[\rightarrow p1, \rightarrow p2 \mid \rightarrow p1)p0$

Conclusión: No debe olvidarse que el autómata de Büchi generado por FBT no sólo está formado por su grafo o sistema de transiciones, que es lo que representan la Figura 5. 10, la Figura 5.12 y la Figura 5.14 mostradas en este subapartado, sino también por sus conjuntos o condiciones de aceptación, que no se representan en esas figuras, a no ser que ciertos estados pertenezcan a los mismos, tal y como sucede en la Figura 5.10.

Algo similar a lo descrito en el Ejemplo 5.18 ocurrirá para ciertas fórmulas FIL que no sean satisfacibles; sin embargo, para otras fórmulas FIL insatisfacibles,

como por ejemplo $\neg[\rightarrow$p0$|\rightarrow)\mathbf{F} \wedge [\rightarrow$p0$|\rightarrow)\mathbf{F}$, simplemente no se generará ningún nodo en el proceso de creación del grafo, con lo que directamente se concluirá que no existe ningún modelo o secuencia que satisfaga dicha fórmula.

5.1.6.4. TRANSFORMACIÓN DEL AUTÓMATA DE BÜCHI GENERALIZADO EN UNO CLÁSICO

Muchas herramientas de verificación, como por ejemplo MARIA[1] [Mäkelä02], que es en la que pretendemos integrar nuestro traductor (debido a que dispone de un comprobador de modelos *on-the-fly*), trabajan con los autómatas de Büchi generalizados que FBT genera, siendo incluso más eficientes que si se les suministra un autómata de Büchi (simple, clásico o normal, o sea, con un único conjunto de estados de aceptación). No obstante, en este subapartado se va a explicar cómo realizar, en caso de ser necesaria, la conversión de un autómata de Büchi generalizado a uno clásico que acepte el mismo lenguaje.

Para transformar un autómata de Büchi generalizado, $G=(\Sigma,S,R,L,I,\mathscr{F})$, donde $\mathscr{F}=\{F_1, F_2, \ldots, F_k\}$, tal y como expresa la Definición 5.29, en un autómata de Büchi (simple), A, que se ajuste a la Definición 5.26, se puede utilizar el método descrito en [Courco92], que básicamente consiste en crear k copias de G, renombrando sus estados y modificando algunas de sus transiciones, tal y como se indica a continuación más formalmente. Así, si $\mathscr{F}=\varnothing$, entonces $A=(\Sigma,S,R,L,I,S)$, es decir, todos los estados del autómata A son de aceptación. Si k=1, entonces el autómata G sería un autómata de Büchi (clásico), esto es, $A=(\Sigma,S,R,L,I,F_1)$. Mientras que si k>1, el autómata resultante sería $A=(\Sigma,S',R',L',I',F)$, definiéndose cada uno de sus elementos del siguiente modo:

- $S'=\bigcup_{i=1..k}\{s_i \mid s \in S\}$, esto es, el número de estados $|S'|$ de A es $k|S|$.
- $R' \subseteq S' \times S'$ es la relación de transición que se obtiene a partir de la relación R, de tal modo que para i=1..k:
  - $\forall r_i \in F_i$ en la copia G_i, si $(r, s) \in R$ entonces $(r_i, s_{(i \bmod k)+1}) \in R'$
  - $\forall r_i \notin F_i$ en la copia G_i, si $(r, s) \in R$ entonces $(r_i, s_i) \in R'$

[1] http://www.tcs.hut.fi/maria/

Obsérvese que en una ejecución del autómata A, si estamos visitando estados de la copia G_i, sólo si llegamos a un estado que pertenezca al conjunto de aceptación F_i nos movemos al sucesor correspondiente de la siguiente copia, $G_{(i \bmod k)+1}$. En otro caso, también nos movemos al sucesor apropiado, pero de la misma copia. Al definir de este modo la relación de transición R' se consigue que el cambio de una copia a la siguiente coincida con la ocurrencia de algún estado de aceptación de la copia origen de dicha transición, con lo que se garantiza que en una *ejecución de aceptación* al menos un estado de aceptación de cada copia sea visitado infinitamente a menudo.

- $L'(s_i) = L(s)$, $\forall i$=1..k, es decir, la etiqueta de cada estado del autómata A es la etiqueta del estado correspondiente del autómata G.
- El conjunto de estados iniciales I' del autómata A es el de la primera copia de G (G_1), esto es, $I' = I_1$, aunque se podría haber elegido como tal el conjunto de estados iniciales de cualquier otra copia.
- De modo similar, se puede escoger como conjunto de estados de aceptación de A el conjunto F_i de cualquier copia G_i, como por ejemplo el conjunto F_k de la última copia, es decir, $F = F_k$ de la copia G_k.

Aunque en el caso de los autómatas de Büchi generalizados que produce nuestro traductor (FBT) se puede aplicar el procedimiento que se acaba de describir, no es preciso hacerlo, puesto que existe un modo más simple de realizar dicha transformación. Para ello, basta con obtener un único conjunto de estados de aceptación, F, que estará formado por los estados que están en la intersección de los distintos F_i, o sea, $F = \cap F_i$. Así, dado un autómata de Büchi generalizado, $G = (\Sigma, S, R, L, I, \{F_1, F_2, \ldots, F_k\})$ con k>1, generado por FBT, para saber visualmente si un estado será finalmente de aceptación o no, basta comprobar si en su etiqueta aparecen los números o identificadores de todos sus conjuntos de aceptación. Por tanto, $A = (\Sigma, S, R, L, I, \cap F_i)$.

Ejemplo 5.19: Conversión de un autómata de Büchi generalizado en un autómata de Büchi (simple)

La Figura 5.10 muestra un autómata de Büchi generalizado que ha sido construido por FBT. Este autómata tiene dos conjuntos de estados de aceptación, como puede apreciarse en la parte inferior de Figura 5.11. Por lo tanto, sólo aquellos estados que

tengan los identificadores de ambos conjuntos de aceptación serán los estados de aceptación del autómata de Büchi (clásico) que se desea obtener, siendo los estados 3, 4, 9 y 12 los que cumplen esa condición, tal y como puede observarse en la Figura 5.10.

5.2. Demostración de corrección del algoritmo de traducción

Puesto que el algoritmo que se ha presentado a lo largo de toda la sección 5.1 es una nueva versión para FIL del algoritmo publicado en [Gerth95] para la LTL, utilizaremos el mismo esquema para su demostración, reescribiendo sus lemas para adaptarlos a la semántica de FIL, a las peculiaridades de nuestro algoritmo y a nuestra notación. El teorema que se muestra a continuación es el que se debe demostrar:

Teorema 5.1. El autómata A_φ, construido por nuestro algoritmo para la especificación (fórmula FIL) φ, acepta exactamente las mismas palabras infinitas sobre $(2^P)^\omega$ que satisfacen (son modelos de) φ.

Demostración: El Lema 5.8 y el Lema 5.9 demuestran el teorema en ambas direcciones.

Notación y definiciones preliminares: Denotaremos mediante $\Phi(s)$ y $\Theta(s)$ respectivamente a los valores de los campos *Old*(*s*) y *Next*(*s*) en el punto donde la construcción del nodo *s* se ha completado, esto es, cuando se añade a *GraphNodes* (línea 7 de la Figura 5.2). Por otra parte, tal y como se indicó en el subapartado 5.1.4.3, $\bigwedge Y$ denotará la conjunción de todas las formulas FIL que integran el conjunto *Y*, considerándose igual a **T** la conjunción del conjunto vacío. Además, utilizaremos X con el mismo significado que tiene el operador *siguiente* (*next*) en la LTL, por lo que cualquier término encabezado por dicho operador representa el conjunto de fórmulas que deben satisfacerse en el próximo estado.

Sea $\xi = x_0x_1x_2\ldots$ una secuencia proposicional o *palabra infinita* del lenguaje $\mathcal{L}((2^\mathcal{P})^\omega)$, entonces ξ_i denotará el sufijo de ξ a partir del punto *i*, o sea, $\xi_i = x_ix_{i+1}x_{i+2}\ldots$.

A un nodo r se le denominará *raíz* (*rooted* en inglés) si cumple una de estas dos condiciones:

(1) r es el primer nodo generado por el algoritmo (ver Figura 5.1), por lo que $New(r)=\{\varphi\}$.

(2) r se crea directamente como sucesor de algún nodo q que se acaba de añadir al grafo (líneas 8-9 de la Figura 5.2). Por consiguiente, $New(r)=Next(q)$.

Cuando un nodo se divide en varios, a todos los nodos resultantes de esa división se les denomina nodos *descendientes*. Y a los nodos completamente procesados $s_1, s_2, \ldots, s_n$ que se obtienen a partir del nodo raíz r mediante el proceso de expansión llevado a cabo por el algoritmo (dividiendo y/o actualizando sucesivamente el nodo r y sus descendientes), se les denomina *descendientes terminales* de r.

Finalmente, recuérdese que en el apartado 4.2.2 se denotó como $\mathbf{red}(\eta)$ al *conjunto reductor de* η, o sea, al conjunto que contiene todos los reductores de la fórmula de intervalo η. Y que la expresión $\eta \succ_{\tau} \eta'$ indica la *relación de reducción* existente entre η y su *reducto* η' con respecto a uno de sus reductores $\tau \in \mathbf{red}(\eta)$.

Lema 5.1. Sea $\sigma = s_0 s_1 s_2 \ldots$ una ejecución de A_φ y η una fórmula de intervalo tal que $\eta \in \Phi(s_0)$. Entonces se cumple uno de los siguientes casos:

(1) $\forall i \geq 0$ y $\forall \tau \in \mathbf{red}(\eta)$, $\tau \notin \Phi(s_i)$ y $\eta \in \Theta(s_i)$. Esto significa que ningún reductor se cumple en ninguno de los estados de σ, motivo por el que η se propaga indefinidamente sin reducir al siguiente estado de σ.

(2) $\exists j \geq 0$ y $\exists \tau \in \mathbf{red}(\eta)$: $\eta \succ_{\tau} \eta'$, tal que τ, $\eta' \in \Phi(s_j)$ y $\forall i$: $0 \leq i < j$, $\tau \notin \Phi(s_i)$ y $\eta \in \Theta(s_i)$. Dicho de otro modo, en algún punto minimal j de σ se satisface algún reductor τ de η y su τ-reducto (denotado como η'), de modo que en todos los estados previos no se cumple ninguno de sus reductores, por lo que η se propaga hasta el punto j.

Demostración: Se sigue directamente del algoritmo construcción del grafo (Figura 5.2), que muestra cómo aplicar las reglas de expansión (líneas 12-44) que hemos derivado a partir de la relación de reducción y de la semántica de FIL.

Lema 5.2. Cuando un nodo y se divide en sus *inmediatos* descendientes $z_1, z_2, \ldots, z_n$ se cumple la siguiente equivalencia: $(\bigwedge New(y) \wedge \bigwedge Old(y) \wedge \mathsf{X}\bigwedge Next(y)) \leftrightarrow \bigvee_{1\le i\le n}(\bigwedge New(z_i) \wedge \bigwedge Old(z_i) \wedge \mathsf{X}\bigwedge Next(z_i))$. De forma similar, cuando y se actualiza a y' se cumple: $(\bigwedge New(y) \wedge \bigwedge Old(y) \wedge \mathsf{X}\bigwedge Next(y)) \leftrightarrow (\bigwedge New(y') \wedge \bigwedge Old(y') \wedge \mathsf{X}\bigwedge Next(y'))$.

Demostración: De nuevo es trivial a partir del algoritmo y de las reglas de expansión para las fórmulas FIL (ver Figura 5.2 y Tablas 5.1 a 5.6).

Lema 5.3. Sea r un nodo *raíz*, $s_1, s_2, \ldots, s_n$ todos sus *descendientes terminales* y $\Omega(r)$ el conjunto de fórmulas en el campo $New(r)$ cuando se crea dicho nodo. Entonces, se cumple la equivalencia $\bigwedge\Omega(r) \leftrightarrow \bigvee_{1\le i\le n}(\bigwedge\Phi(s_i) \wedge \mathsf{X}\bigwedge\Theta(s_i))$. Además, si $\xi \models \bigvee_{1\le i\le n}(\bigwedge\Phi(s_i) \wedge \mathsf{X}\bigwedge\Theta(s_i))$, entonces existe algún i tal que $\xi \models \bigwedge\Phi(s_i) \wedge \mathsf{X}\bigwedge\Theta(s_i)$ y para cada $\eta\in\Phi(s_i)$ tal que $\xi \models \tau$, tenemos que $\tau, \eta'\in\Phi(s_i)$, donde $\eta \succ_\tau \eta'$.

Demostración: Si r es un nodo raíz, entonces $Old(r) = Next(r) = \varnothing$. Así, para demostrar la equivalencia anterior, sólo hay que aplicar sucesivamente el Lema 5.2 en cada punto donde r o sus descendientes se dividen o actualizan, hasta que se obtiene el conjunto $\{s_i\}$ de nodos *descendientes terminales* de r, con $New(s_i) = \varnothing$. Durante el proceso de expansión de un nodo que contiene la fórmula η, si a η se le aplica uno de sus reductores, τ, entonces el algoritmo añade las fórmulas τ y η' al campo *New* de uno de sus inmediatos descendientes y, por consiguiente, τ y η' estarán incluidos en $\Phi(s_i)$ antes de que s_i se añada al grafo (o sea, a *GraphNodes*).

Lema 5.4. Sea ξ una palabra infinita, tal que $\xi_i \models \bigwedge\Phi(q) \wedge \mathsf{X}\bigwedge\Theta(q)$. Entonces, existe una transición $(q, s)\in R$ en A_φ tal que $\xi_{i+1} \models \bigwedge\Phi(s) \wedge \mathsf{X}\bigwedge\Theta(s)$ y $\Theta(q)\subseteq\Phi(s)$. Además, sea $\Gamma = \{\tau \mid \eta \succ_\tau \eta'$ y $\eta\in\Phi(q)$ y $\tau\notin\Phi(q)$ y $\xi_{i+1} \models \tau$ y $\xi_{i+1} \models \eta'\}$, entonces existe un sucesor s de q tal que $\Gamma\subseteq\Phi(s)$.

Demostración: Cuando se añade un nodo q a *GraphNodes*, inmediatamente se genera como su sucesor un nodo raíz r, con $\Omega(r) = \Theta(q)$. Por lo tanto, ξ_{i+1} debe satisfacer $\Omega(r)$ y, como del Lema 5.3 se sigue que existe un descendiente terminal s_i de r tal que $\xi_{i+1} \models \bigwedge\Phi(s_i) \wedge \mathsf{X}\bigwedge\Theta(s_i)$ y para cada $\eta\in\Phi(s_i)$ con $\xi_{i+1} \models \tau$, siendo $\eta \succ_\tau \eta'$, se tiene que $\tau, \eta'\in\Phi(s_i)$, entonces se puede concluir que s_i es el sucesor requerido s.

Lema 5.5. Para cada estado inicial $s\in I$ del autómata A_φ, se tiene que $\varphi\subseteq\Phi(s)$.

Demostración: Puesto φ se asigna inicialmente al campo *New* del primer nodo (denominado *FirstNode* en la Figura 5.1), cada nodo inicial s del grafo incluirá a φ en $\Phi(s)$ debido a que es un descendiente terminal de *FirstNode*.

Lema 5.6. Sea A_φ el autómata construido por nuestro algoritmo para la especificación FIL φ. Entonces se cumple la siguiente equivalencia: $\varphi \leftrightarrow \bigvee_{s\in I}(\bigwedge\Phi(s) \wedge \mathsf{X}\bigwedge\Theta(s))$.

Demostración: La equivalencia anterior se deriva fácilmente a partir del Lema 5.3, identificando $\Omega(r)$ con φ.

Lema 5.7. Sea $\sigma = s_0 s_1 s_2 \ldots$ una ejecución de A_φ que acepta la palabra infinita ξ, entonces $\xi \models \bigwedge\Phi(s_0)$.

Demostración: Si la ejecución σ satisface las condiciones de aceptación de A_φ, entonces en algún estado s_i de σ se aplica un reductor τ de cualquier fórmula de intervalo $\eta\in\Phi(s_0)$, por lo que del Lema 5.4 se deduce que un sufijo ξ_i de la palabra infinita ξ satisface τ y η' (el τ-reducto de η). Por consiguiente, aplicando sucesivamente el Lema 5.4 a partir del estado inicial s_0, alcanzaremos las condiciones de aceptación de A_φ y al mismo tiempo habremos demostrado que cualquier sufijo ξ_i satisface $\bigwedge\Phi(s_i)$ sobre la secuencia completa de estados de σ, y consecuentemente para el estado inicial s_0.

Lema 5.8. Si σ es una ejecución de A_φ que acepta la palabra infinita ξ, entonces $\xi \models \varphi$.

Demostración: Del Lema 5.7 se sigue que $\xi \models \bigwedge\Phi(s_0)$, donde $s_0\in I$. Entonces, por el Lema 5.5, $\varphi\subseteq\Phi(s_0)$. Consecuentemente, $\xi \models \varphi$.

Lema 5.9. Si $\xi \models \varphi$, entonces existe una ejecución σ de A_φ que acepta ξ.

Demostración: El Lema 5.6 afirma que existe un nodo $s_0\in I$ tal que $\xi \models \bigwedge\Phi(s_0) \wedge \mathsf{X}\bigwedge\Theta(s_0)$. A partir de él, se puede generar σ aplicando repetidamente el Lema 5.4. Para obtener una ejecución de aceptación, sólo el caso (2) del Lema 5.1 es aplicable. Así, para cada fórmula de intervalo $\eta\in\Phi(s_0)$ o ninguno de sus reductores se cumple en s_0 y por tanto η se propaga a los estados siguientes, o un reductor τ se cumple en s_0, y por el Lema 5.3 τ, $\eta'\in\Phi(s_0)$. Entonces se aplica repetidamente el Lema 5.4 hasta que se cumplan las condiciones de aceptación de A_φ. Por lo tanto, σ acepta ξ.

ÍNDICE DE REFERENCIAS EMPLEADAS

[Courco92], *145*
[Daniel99], *103*
[Gerth95], *92*, *98*, *103*, *147*
[Hornos01a], *91*
[Hornos02], *91*
[Mäkelä02], *145*
[Ramakr96a], *117*
[Strous93], *107*
[Wolper85], *91*

Capítulo 6

Diseño e Implementación de la Herramienta

Contenido

RESUMEN Y ORGANIZACIÓN

Este capítulo completa y complementa al capítulo anterior, ya que en él se exponen las principales decisiones que se han tomado a la hora de diseñar e implementar el algoritmo presentado en el Capítulo 5. Así, en la primera sección se ofrece una descripción general de la estructura (estática) de la implementación que hemos realizado. Hemos denominado FBT (*FIL to Büchi automaton Translator*) al traductor o herramienta resultante. La segunda sección presenta la interfaz de FBT, tanto de entrada como de salida, esto es, la sintaxis que hay que utilizar para suministrar una especificación FIL a nuestra herramienta, así como la sintaxis que ésta emplea para devolvernos el resultado de su ejecución. Como este resultado es textual, en el último apartado de esa sección se indica cómo se consigue una representación gráfica a partir del mismo. En la tercera sección se muestran y comentan una serie de resultados experimentales obtenidos con FBT. Finalmente, en la cuarta sección se compara nuestro traductor con otro para la LTL (en cuya implementación nos hemos basado para realizar la nuestra), resaltando las mejoras y optimizaciones que hemos llevado a cabo con respecto a aquél y comparando los resultados obtenidos con ambos traductores para especificaciones equivalentes en sus respectivos formalismos lógicos, o sea, en FIL y LTL.

6.1. Estructura general de la implementación

Como ya se ha comentado, a la implementación que hemos realizado del algoritmo presentado en el capítulo anterior le hemos denominado FBT (*FIL to Büchi automaton Translator*). Para ello, nos hemos basado en la implementación que del algoritmo publicado en [Gerth95] han realizado en una universidad de Finlandia (*Helsinki University of Technology*) y a la que han denominado LBT (*LTL to Büchi automaton Translator*)[1]. Así, hemos reutilizado parte de este código, realizando modificaciones para adaptarlo a las fórmulas FIL, para mejorarlo y optimizarlo en ciertas partes (ver sección 6.4) y para añadirle las clases y funciones requeridas por las fórmulas de la lógica que empleamos que no están presentes en la LTL, esto es, todas aquéllas que contienen alguna modalidad de intervalo.

La implementación, que se ha realizado en C++ [Strous93], se distribuye en seis ficheros distintos, que suman en total más de 2200 líneas de código (incluyendo comentarios). En el Apéndice C: Documentación del Código Fuente de FBT se indica cuáles son esos ficheros, así como el contenido de cada uno de ellos, para lo que se ha documentado cada uno de sus componentes, ya sean variables, funciones o clases. Para éstas últimas se detallan sus atributos, tipos declarados en ellas y métodos que implementan, clasificándolos en públicos, protegidos y privados, e indicando también si son estáticos o no y cuáles son sus clases amigas (*friends*). Además, para una mejor descripción de cada clase, se incluyen diagramas de herencia y/o de colaboración en los que se muestran la relación existente entre esa clase y el resto de clases implementadas. Al mismo tiempo, se documentan también todos los parámetros de las funciones y métodos que allí aparecen, así como lo que devuelven tras su ejecución (si es el caso). Para todos y cada uno de los elementos relacionados en el mencionado apéndice, desde los ficheros a los parámetros, se da una breve descripción de su cometido o de su significado.

Siguiendo las recomendaciones encontradas en [Deitel99] sobre buenas prácticas de programación, en la implementación de cada clase se ha puesto en primer lugar la *parte pública*, con el fin de hacer hincapié en la interfaz de la clase, mostrando a continuación su *parte privada*, o lo que es lo mismo, los detalles de cómo se implementa dicha clase. Así, la persona que eche un vistazo al código fuente o a la documentación generada a partir de él (Apéndice C) centra primero su atención en aquellos elementos de la clase con lo que se puede interactuar.

---

[1] http://www.tcs.hut.fi/Software/maria/tools/lbt/

6.1.1. Diagramas de clases en UML

Aunque tanto la metodología OMT (*Object Modeling Technique*) [Rumbau96] como la surgida posteriormente y en la que ésta se integra, UML (*Unified Modeling Language*) [Rumbau99], están pensadas para diseñar un sistema antes de construirlo, en este apartado hacemos uso de dicha metodología en el sentido contrario. Es lo que se conoce con el nombre de *ingeniería inversa*, esto es, a partir de una implementación ya realizada, generar los modelos correspondientes en UML. En nuestro caso sólo obtendremos el modelo de objetos, y más concretamente, los diagramas de clases. El objetivo es mostrar, en este lenguaje universalmente conocido, la estructura estática de la implementación que hemos realizado, o sea, los aspectos estáticos y estructurales de los "datos" de nuestro sistema, descritos en términos de clases de objetos y de relaciones (*asociaciones*, en la terminología de UML) entre ellas.

Debido a que se utiliza ingeniería inversa, los nombres (en inglés) que aparecen en los diagramas de clases que se exponen en este apartado se corresponden exactamente con los que se han utilizado en la implementación (como se puede comprobar en el Apéndice C), a excepción de *predecesor* y *sucesor* (en la Figura 6.2), que se muestran en castellano justamente para indicar su no correspondencia con ningún componente del código implementado. Así, todos los *nombres de rol* (salvo la excepción mencionada) empiezan por "m_" en dichos diagramas; esto es debido a que hemos respetado la notación original de la implementación en la que nos hemos basado (LBT), que utiliza el prefijo "m_" para designar a todos y cada uno de los atributos (privados) de las clases, y que quiere decir algo así como "almacenado en la *memoria* interna del objeto". Con esta notación se pretende diferenciar claramente los atributos (internos) de las clases del resto de parámetros y variables utilizadas por el programa. En estos diagramas no se muestran ni los atributos ni los métodos que componen cada clase, debido a que esos elementos ya se incluyen en el Apéndice C. La única excepción a esto la constituyen los mencionados nombres de rol, que representan el nombre de un atributo de la clase fuente de la asociación correspondiente. Así, por ejemplo, *m_nested* (ver Figura 6.1) es un atributo de la clase *FilInterval*.

En la Figura 6.1 se muestra el diagrama de clases en UML que describe la estructura de las distintas clases de fórmulas FIL que se han considerado en la implementación de FBT. Por esta razón, todos los nombres de clases que en él aparecen, a excepción de *SearchPattern* (clase que representa a los patrones de

búsqueda), empiezan por "Fil". La clase *Fil* es una clase abstracta, que define los elementos comunes a sus subclases, de ahí que todas ellas se relacionen con la primera mediante una *generalización* (representada por un triángulo en UML), ya que esas subclases refinan o especializan a la mencionada superclase. *FilAtom* es la clase que representa a los literales, mientras que la clase *FilConstant* representa a las constantes lógicas (**T** o **F**). La clase *FilJunct* implementa las conjunciones y disyunciones de fórmulas FIL, razón por la que se representa mediante una *agregación* (cuyo símbolo en UML es el rombo) de dos fórmulas FIL cualesquiera. Obsérvese que la multiplicidad o cardinalidad de esa asociación se indica en este caso mediante un 2, que especifica que es ése exactamente el número de fórmulas FIL que relaciona un objeto o instancia de la clase *FilJunct*. Por la misma razón, esa misma representación es la que se ha utilizado para la clase *FilIff*, que implementa las equivalencias y disyunciones exclusivas de dos fórmulas FIL. Finalmente, la clase *FilInterval* es la encargada de construir las fórmulas de intervalo, cuya estructura está formada por un intervalo, representado mediante dos patrones de búsqueda (ya que su cometido es localizar sendos extremos del intervalo), y por *una* fórmula FIL cualquiera (*m_nested*) que se anida a dicho intervalo. Un patrón de búsqueda, o sea, una instancia de la clase *SearchPattern* no es más que una secuencia (*m_sp*) de cero (caso de ser un patrón trivial) o más fórmulas FIL. Así, en nuestra implementación cada búsqueda se representa únicamente mediante su fórmula objetivo.

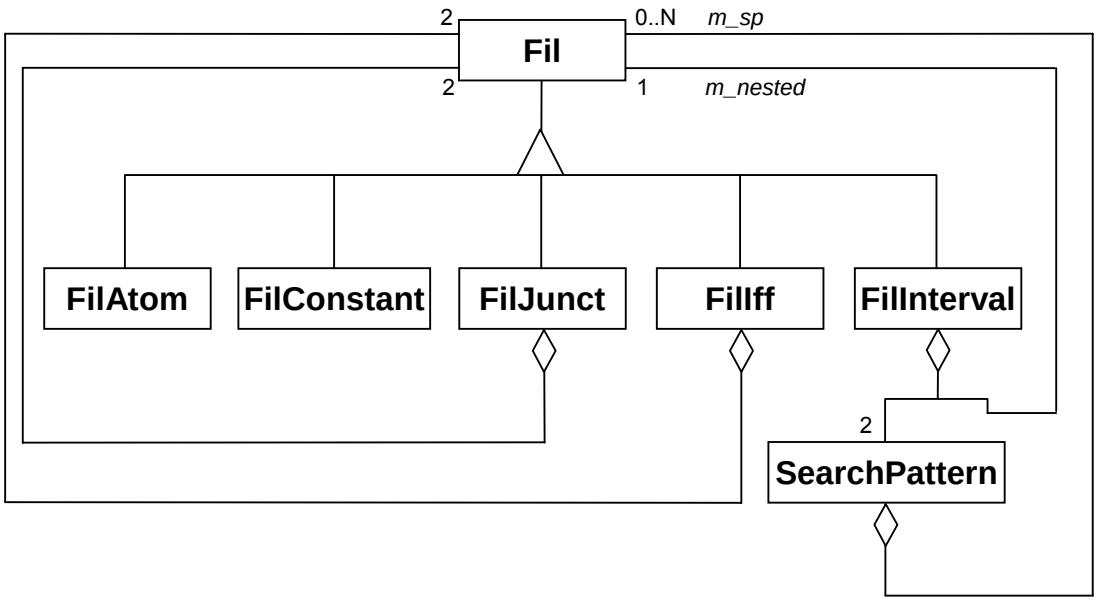


Figura 6.1. Diagrama de clases UML: Estructura de las distintas clases de fórmulas FIL

Por otra parte, el diagrama de clases en UML que se representa en la Figura 6.2 describe la estructura del grafo que FBT genera a partir de una fórmula FIL. Así, *FilGraph* es la clase que representa al grafo, por lo que sus instancias están formadas por la agregación de (cero o más) nodos, o sea, objetos de la clase *FilGraphNode*. Los nombres de estas dos clases también comienzan por "Fil" debido a que, a su vez, la estructura de cada nodo está constituida por la agregación de una serie de conjuntos de (cero o más) fórmulas FIL, cuyos nombres son: *m_atomic*, *m_new*, *m_old* y *m_next*. Además, dos nodos del grafo están relacionados si entre ellos existe una transición, lo que se ha representado mediante una asociación con dos nombres de rol (*predecesor* y *sucesor*) y que en la implementación se corresponde con el atributo *m_incoming*, que en cada nodo almacena el conjunto de (uno o más) identificadores de nodos (sus predecesores), por lo que representa al conjunto de sus arcos entrantes.

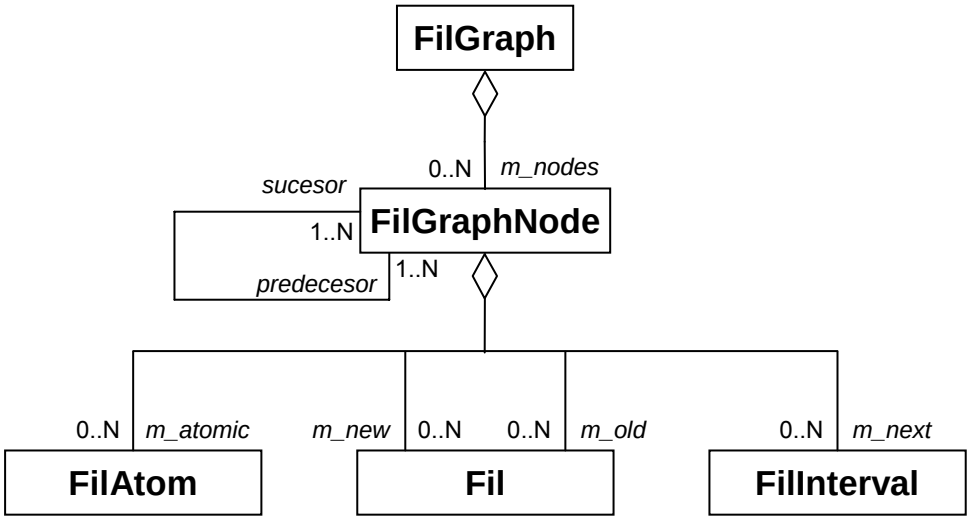


Figura 6.2. Estructura en UML del grafo generado a partir de una fórmula FIL

Rápidamente puede verse que los atributos *m_new*, *m_old* y *m_next* de la clase *FilGraphNode* representan respectivamente a los campos *New*, *Old* y *Next* explicados en el apartado 5.1.2. Estructura de un nodo. Sin embargo, en dicho apartado no hay ningún campo que encaje con *m_atomic*, ya que la inclusión de dicho atributo corresponde a una decisión tomada a la hora de realizar la implementación. En realidad, este atributo representa a los literales almacenados en *m_old*, por lo que se puede considerar que es redundante. No obstante, lo hemos mantenido en nuestro programa por reutilizar parte de la implementación de LBT. Con este atributo redundante se consigue una mayor eficiencia, al no tener que

calcular el conjunto de literales cada vez que sea necesario utilizarlo. Obsérvese también que, dado que las fórmulas de intervalo son las únicas que pueden posponer su cumplimiento, en *m_next* sólo se pueden introducir este tipo de fórmulas. Esto se puede considerar como otra decisión de implementación, razón por la que, cuando se explicó el campo *Next* en el Capítulo 5, no se mencionó nada acerca de su composición (integrada exclusivamente por fórmulas de intervalo).

6.2. Interfaz de FBT

FBT, que ha sido diseñado e implementado para poder ser invocado (como subproceso) por un comprobador de modelos *on-the-fly*, analiza una fórmula FIL, suministrada a través de la entrada estándar en formato textual. Tras su ejecución, escribe el autómata de Büchi generalizado que es semánticamente equivalente a dicha fórmula en la salida estándar, también en formato textual. Ambos formatos, de entrada y de salida, se describen respectivamente en los dos primeros apartados de esta sección. Como puede apreciarse en ellos, en ambos se utiliza la *notación prefija*, debido a que facilita el análisis descendente-recursivo que debe llevar a cabo nuestro traductor. En el tercer y último apartado de esta sección se explica cómo se consigue una representación gráfica a partir de la salida textual que FBT genera.

6.2.1. Sintaxis de las fórmulas de entrada

A continuación se presenta la gramática que describe la sintaxis de todos y cada uno de los tipos de fórmulas que FBT acepta como entrada. Para ello se utiliza la notación BNF (*Backus-Naur Form*) [Naur60], que es universalmente conocida. A la derecha de algunas reglas de producción se han añadido algunos *comentarios*, delimitados por los símbolos /* y */. Estos comentarios no son parte de la gramática formal, al igual que tampoco lo son las *expresiones* que se ponen *en negrita*, y que separan las reglas de la gramática en distintos grupos o secciones, con el fin de facilitar su lectura. Los símbolos terminales se encierran entre comillas simples o se presentan como expresiones regulares en el estilo utilizado por el generador de analizadores léxicos FLEX[2]. Como es habitual, los símbolos no terminales se encierran entre ángulos, representándose además en el estilo de los hipervínculos

[2] http://www.gnu.org/software/flex/flex.html

(subrayados y en azul) cuando aparecen en la parte derecha de una regla de producción.

Debido a que todas las reglas de producción no caben en el espacio disponible en un folio, éstas se han dividido en dos figuras, para que su presentación sea más elegante. En la primera de ellas (Figura 6.3) se exponen las reglas que construyen los tipos más simples de fórmulas FIL, mientras que en la segunda (Figura 6.4) se describe cómo se forman las propiedades más complejas, o sea, aquellas fórmulas que contienen algún operador temporal. Obsérvese que en ellas tanto las reglas de producción como los comentarios a las mismas se han redactado en inglés.

Types of formulas

```
<f> ::=  <constant> |
         <proposition> |
         <negated formula> |
         <propositional formula> |
         <interval formula> |
         <LTL temporal formula> |      /* although actually they aren´t part of FIL, they can be
                                          used as abbreviations for interval formulas */
         [ \t\n\r\v\f]<f> |            /* white space is ignored */
         <f>[ \t\n\r\v\f]              /* white space is ignored */
```

Basic formulas

```
<constant> ::=          't' |         /* true */
                        'f'           /* false */
<proposition> ::=       'p'[0-9]+     /* atomic proposition */
<negated formula> ::=   '!'<f>
```

Propositional formulas

```
<propositional formula> ::=  <binary operator> <f1> <f2>
<binary operator> ::=        '&' |    /* conjunction */
                             '|' |    /* disjunction */
                             'i' |    /* implication: "i <f1> <f2>" is shorthand for "|
                                         ! <f1> <f2>" */
                             'e' |    /* equivalence */
                             '^'      /* exclusive disjunction (xor) */
```

Figura 6.3. Sintaxis de las fórmulas que FBT acepta (I): Fórmulas más simples

Interval formulas

```
<interval formula> ::=   <interval> <f>                    /* formula nested to an interval. <f>
                                                              must hold within the context
                                                              provided by the interval */

<interval> ::=           '[' <non-trivial pattern>         /* an "standard" interval, i.e. none of
                             <non-trivial pattern> |         its search patterns is trivial */
                         '[' '-' <non-trivial pattern> |   /* trivial left search pattern: it leaves us
                                                              at the point where we are */
                         '[' <non-trivial pattern> '>'     /* trivial right search pattern: it takes
                                                              us to the end of the current context */

<non-trivial pattern> ::= <f> |                            /* a non-trivial search pattern is made
                                                              up of one or more searches,
                         ',' <f> <non-trivial pattern>     each of them locates its target
                                                              formula in the reflexive future */
```

LTL temporal formulas (used as abbreviations for interval formulas)

```
<LTL temporal formula> ::= <LTL unary operator> <f> |
                           <LTL binary operator> <f1> <f2>

<LTL unary operator> ::=   'F' |   /* finally, eventually: "F <f>" is shorthand for
                                      "! [ <f> > f" */
                           'G'     /* globally, henceforth: "G <f>" is shorthand for
                                      "[ ! <f> > f" */

<LTL binary operator> ::=  'U' |   /* (strong) until: "U <f1> <f2>" is shorthand for
                                      "! [ | ! <f1> <f2> > ! <f2>"  */
                           'V'     /* release: "V <f1> <f2>" is shorthand for
                                      "! [ | ! <f2> & <f1> <f2> > ! & <f1> <f2>" */
```

Figura 6.4. Sintaxis de las fórmulas que FBT acepta (II): Fórmulas temporales

Como se puede decir que ambas figuras son "autoexplicativas", de ellas sólo apuntaremos el hecho de que FBT acepta también los operadores temporales de la LTL, pero sólo como abreviaciones de las correspondientes fórmulas de intervalo FIL (tal y como se indica en el primer comentario de la Figura 6.3 y se vuelve a señalar en la Figura 6.4). Esto quiere decir que cuando FBT analiza una de estas fórmulas, inmediatamente la transforma en la fórmula de intervalo equivalente, que es la que realmente almacena y procesa. Por tanto, durante el proceso de expansión de las fórmulas, en ningún momento el algoritmo tiene que descomponer fórmulas que contengan operadores temporales de la LTL, motivo por el que en el apartado 5.1.4. Reglas de expansión no se ha contemplado ninguna regla para este tipo de fórmulas. Obsérvese finalmente que FBT ignora (ver las dos últimas líneas de la primera regla de producción en la Figura 6.3) los espacios en blanco,

tabuladores horizontales (\t) y verticales (\v), saltos de línea (\n), retornos de carro (\r) y avances de página (\f). Por esta razón, a la hora de suministrar una fórmula como entrada a FBT, se puede introducir cualquiera de esos separadores y en el número que se estime conveniente entre cualesquiera dos términos de la misma.

6.2.2. Sintaxis de la salida generada

La Figura 6.5 muestra la sintaxis utilizada por FBT para devolver el autómata de Büchi generalizado (de ahí, el acrónimo *gba* utilizado en la parte izquierda de su primera regla de producción) equivalente a la fórmula FIL que se le suministró como entrada. Para ello, se hace uso de la misma notación que se ha explicado en el apartado anterior. Obsérvese que incluso se emplea el símbolo no terminal <proposition>, definido en la Figura 6.3.

El siguiente ejemplo trata de clarificar y mejorar la comprensión de la gramática expuesta en la Figura 6.5. Al mismo tiempo, se aprovecha para ilustrar algunas de las cuestiones mencionadas en el apartado anterior.

Ejemplo 6.1: Salida generada por FBT para la fórmula `!Gp0`

Supongamos que queremos obtener la salida correspondiente a la fórmula `!Gp0` en un fichero denominado `automaton.txt`, la orden que tendríamos que teclear (en el *prompt* del sistema) sería: `echo '!Gp0' | fbt >automaton.txt`. Si esa fórmula fuera el contenido de un fichero, por ejemplo: `formula.txt`, entonces se podría teclear la siguiente orden alternativa: `fbt <formula.txt >automaton.txt`. Obviamente, en ambos casos, si no ponemos la última parte de la orden (a partir del símbolo de redireccionamiento) la salida generada se obtendría en la pantalla. La Figura 6.6 muestra lo que FBT devuelve tras ejecutar una de las órdenes anteriores, es decir, el contenido del fichero `automaton.txt`, explicando su significado. FBT convierte la fórmula suministrada en su fórmula de intervalo equivalente: `![!p0 > f`, que es la que realmente almacena y procesa. Como es lógico, FBT efectuaría exactamente el mismo procesamiento y la misma salida si la fórmula introducida hubiera sido, por ejemplo, cualquiera de las siguientes: `F!p0`, `| F!p0 !Gp0` o `& F!p0 !Gp0`, ya que son equivalentes entre sí.

```
<gba> ::=                <no. states> <ws> <no. accept. sets> <states>
<no. states> ::=         [0-9]+        /* number of total states, included the initial one (if 0, the
                                          provided formula is not satisfiable) */
<ws> ::=                 [ \n]+        /* white space */
<no. accept. sets> ::=   [0-9]+        /* number of acceptance sets (if 0, all states are accepting) */
<states> ::=             <states> <ws> <state> |
                         /* empty */
<state> ::=              <state id.> <ws> <initial?> <ws> <acceptance sets> <end> <transitions>
                            <end>
<state id.> ::=          [0-9]+        /* state identifiers can be arbitrary unsigned integers (the
                                          initial state is always numbered 0) */
<initial?> ::=           '0' |         /* it is not an initial state */
                         '1'           /* initial state (exactly one state must be initial) */
<acceptance sets> ::=    <acceptance sets> <accept. id.> <ws> |
                         /* empty */
<accept. id.> ::=        [0-9]+                    /* acceptance set identifiers can be
                                                      arbitrary unsigned integers */
<end> ::=                '-1'                      /* it marks the end of either the state
                                                      label or the set of state transitions */
<transitions> ::=        <transitions> <ws> <transition> |
                         /* empty */
<transition> ::=         <state id. > <ws> 't' |   /* constantly enabled transition to the
                                                      node whose number is <state id.> */
                         <state id.> <ws> <gate>   /* conditionally enabled transition to the
                                                      node whose number is <state id.> */
<gate> ::=               <literal> |
                         '&' <ws> <gate> <ws> <gate>   /* conjunction of literals */
<literal> ::=            <proposition> |           /* atomic proposition */
                         '!' <ws> <proposition>    /* negated atomic proposition */
```

Figura 6.5. Sintaxis de la salida que FBT genera

6.2.3. Visualización gráfica de los autómatas generados

Con el fin de visualizar gráficamente la salida textual generada por FBT, debemos ejecutar un filtro, al que hemos denominado GBA2DOT. Se le ha puesto este nombre debido a que convierte el autómata de Büchi generalizado que FBT produce al lenguaje *dot*, de modo que el resultado de dicha conversión es directamente

aceptado por la herramienta de visualización de grafos dirigidos GRAPHVIZ[3] [Gansne00].

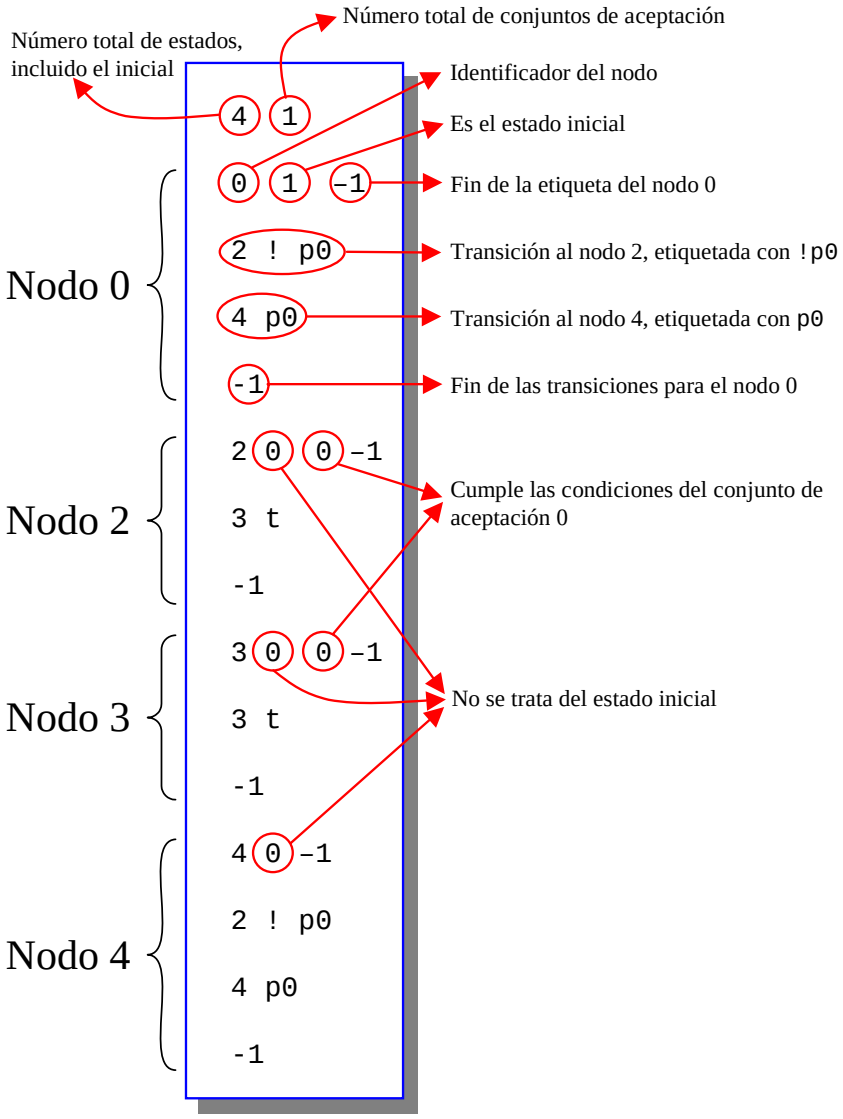


Figura 6.6. Salida textual que FBT genera para la fórmula `!Gp0`

El siguiente ejemplo muestra, haciendo uso del ejemplo expuesto en el apartado anterior, cómo se puede obtener finalmente una representación gráfica a partir de la salida textual producida por nuestro traductor.

[3] http://www.graphviz.org/

Ejemplo 6.2: Representación gráfica de la salida generada por FBT para la fórmula `!Gp0`

A partir del contenido del fichero `automaton.txt` (presentado en la Figura 6.6) y mediante la siguiente orden: `gba2dot <automaton.txt >graph.txt`, se obtendría lo que se muestra en la Figura 6 .7, que se almacenaría en el fichero `graph.txt`. Este formato, redactado en el lenguaje *dot*, es ya más comprensible, no sólo por nosotros, sino también por la herramienta que debe generar automáticamente la correspondiente representación gráfica a partir de él. Así, sólo debemos introducir la siguiente orden: `dotty - <graph.txt`, que llama a la herramienta de visualización, produciendo como resultado la ventana que se presenta en la Figura 6.8, y que contiene la representación gráfica del autómata de Büchi generalizado que FBT produce para la fórmula `!Gp0`. Si, por el contrario, deseamos obtener este autómata en el mismo formato que los autómatas mostrados en el subapartado 5.1.6.3 y en un fichero gráfico, para insertarlo en algún documento, tal y como se hizo en el subapartado mencionado, entonces tendremos que teclear una orden un poco más larga: `dot -Tpng -Grankdir=LR -Nfontname=Efont_Serif -Efontname=Efont_Serif <graph.txt >nGp0.png`, cuyo resultado se muestra en la Figura 6.9.

```
digraph g {
0[style=filled,label="0"];
0->2[label="!p0"];
0->4[label="p0"];
2[label="2\n0"];
2->3[label="t"];
3[label="3\n0"];
3->3[label="t"];
4[label="4"];
4->2[label="!p0"];
4->4[label="p0"];
}
```

Figura 6.7. Resultado de ejecutar el filtro para la salida generada por FBT para la fórmula `!Gp0`

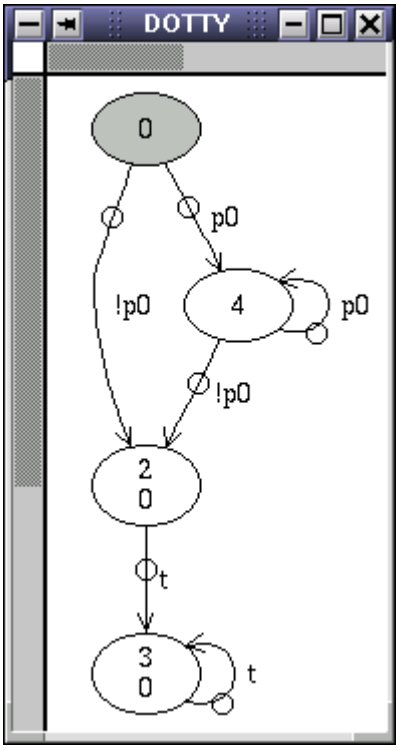


Figura 6.8. Ventana que muestra gráficamente el autómata de Büchi generalizado para `!Gp0`

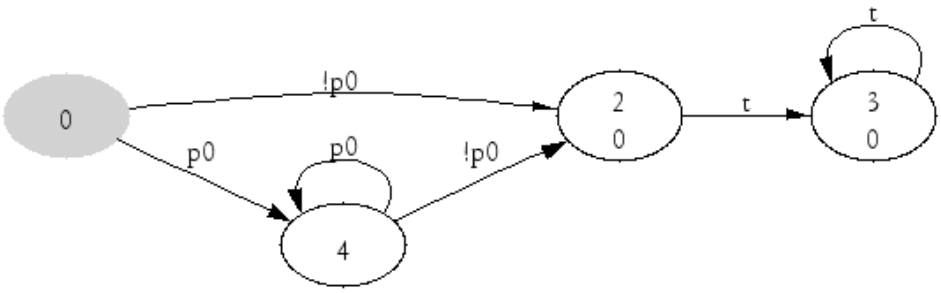


Figura 6.9. Autómata de Büchi generalizado para la fórmula `!Gp0`

6.3. Resultados experimentales

En la Tabla 6.1 se muestran algunos resultados experimentales obtenidos con nuestro traductor para ciertas propiedades especificadas mediante fórmulas FIL. Como puede apreciarse, esta tabla se divide verticalmente (mediante una línea azul más gruesa) en tres zonas:

- La situada más a la izquierda presenta las fórmulas o especificaciones que deseamos que FBT transforme en el autómata de Büchi equivalente. La primera de sus columnas contiene la especificación que se le suministra como

entrada en cada caso, mientras que la segunda alberga la fórmula (equivalente a la de la columna anterior) que FBT realmente almacena y procesa. Obsérvese que las diez primeras filas se corresponden con casos en los que a FBT se le introduce una fórmula cuyos operadores temporales pertenecen a la LTL. Tal y como se dijo en el apartado 2.4.1, esos operadores se definen en FIL para extender su sintaxis restringida y, como se ha comentado en el apartado 6.2.1, en nuestra herramienta se utilizan sólo como abreviaciones de las correspondientes fórmulas de intervalo, que son las que se muestran en la segunda columna. Así, se puede decir también que en la primera de las columnas de esta zona se representa la correspondiente fórmula en la sintaxis extendida de FIL, mientras que en la segunda se representa en su sintaxis restringida. En los ejemplos expuestos en las doce filas inferiores, FBT se ejecuta sobre una fórmula expresada en la sintaxis restringida de FIL, razón por la que en esos casos ambas columnas contienen la misma fórmula, excepto en las filas 15 y 16, en las que se introduce una pequeña modificación, que se explicará en el apartado 6.4.1 (ver Tabla 6.2).

- La zona central muestra el tamaño de los autómatas generados; así, sus dos columnas representan respectivamente el número de nodos (estados) y arcos (transiciones) del autómata correspondiente. En su primera columna no se ha considerado el nodo inicial (recuérdese que en el Ejemplo 5.16 se dijo que era un nodo ficticio), pero sí se han contabilizado los arcos que salen de él en su segunda columna.

- La zona de la derecha se reserva para mostrar todo lo relacionado con los estados de aceptación del autómata resultante. Así, siguiendo la terminología utilizada en la Definición 5.29: Autómata de Büchi generalizado, la columna k indica el número de conjuntos de estados de aceptación que posee el autómata de Büchi generalizado producido para cada especificación. La siguiente columna, $|F_i|$, muestra el número de estados de aceptación de cada uno de dichos conjuntos, o sea, el número de estados que cumplen la condición codificada por cada conjunto de aceptación. Cuando $k > 1$, la última columna contiene, según lo establecido en el subapartado 5.1.6.4, el número de estados de aceptación del autómata de Büchi (clásico) que se obtendría finalmente.

Algunos de los resultados expuestos en las diez primeras filas de la Tabla 6.1 se comentarán en el apartado 6.4.2, donde se compararán además con los obtenidos por LBT para las mismas fórmulas o fórmulas equivalentes. No obstante:

Tabla 6.1. Resultados experimentales obtenidos con FBT para algunas fórmulas FIL

	Fórmula o especificación que FBT ...		Tamaño		Estados de aceptación		
	acepta como entrada	almacena y procesa	Nodos	Arcos	k	$\lvert F_i \rvert$	$\lvert \cap F_i \rvert$
1	Gp0	[!p0 > f	1	2	0		
2	Fp1	![p1 > f	3	6	1	2	
3	U p1 p2	![|!p1p2 > !p2	4	9	1	3	
4	V p1 p2	[|&p1p2!p2 > &p1p2	4	8	1	3	
5	FFp1	![![p1 > f > f	4	8	2	3, 3	2
6	!GGp0	![![!p0 > f > f	4	8	2	3, 3	2
7	F|p0&p1p2	![|p0&p1p2 > f	5	15	1	3	
8	& Fp0 Fp1	& ![p0 > f ![p1 > f	9	20	2	6, 6	4
9	!e FFp1 Fp1	^ ![p1 > f ![![p1 > f > f	1	2	1	0	
10	i GFp1 GFp2	| ![[p1 > f > f [[p2 > f > f	5	15	3	4, 3, 4	2
11	[- p3 ![- p2 [- p1 p0	[- p3 ![- p2 [- p1 p0	9	18	1	3	
12	[- p0 [- p1 [p2 > f	[- p0 [- p1 [p2 > f	7	16	1	3	
13	![- p0 [- p1 [p2 > f	![- p0 [- p1 [p2 > f	7	14	1	2	
14	[- p0 ![p1 p2 p3	[- p0 ![p1 p2 p3	8	17	1	3	
15	![- p0 !| p1 [p2 > f	![- p0 & !p1 ![p2 > f	5	10	1	2	
16	![p0 p3 !^ p1 p2	![p0 p3 e p1 p2	6	14	2	2, 5	2
17	[p1 p2 f	[p1 p2 f	9	17	2	8, 6	6
18	[p0 p1 [p2 > f	[p0 p1 [p2 > f	12	26	2	9, 9	7
19	![p0 p1 [p2 > f	![p0 p1 [p2 > f	8	18	2	2, 7	2
20	[p1 ,p1p2 p0	[p1 ,p1p2 p0	10	21	2	7, 9	7
21	![,p1p2 ,p3p4 !p0	![,p1p2 ,p3p4 !p0	21	58	2	12, 3	3
22	![p0 > [- ,p1p0 !p2	![p0 > [- ,p1p0 !p2	8	17	1	3	

- Obsérvese que la fórmula de la fila 8 es una FPM (ver Definición 5.5).

- El autómata resultante para la fórmula de la fila 9 no acepta ninguna secuencia, dado que tiene un conjunto de estados de aceptación y su único estado no pertenece a él. Esto es lógico, ya que esa fórmula no es satisfacible, puesto que su negación (`e FFp1 Fp1`) es una tautología.

- Fórmulas como la representada en la fila 10 se suelen utilizar en verificación cuando se desea comprobar alguna propiedad bajo cierta condición de equidad (*fairness*) [France86] que no está implementada en el comprobador de modelos. Así, en la fórmula de la fila 10, `p1` expresaría que algún elemento (por ejemplo: un proceso, una transición, etc.) está habilitado, mientras que `p2` representaría la ejecución de ese elemento.

En las doce últimas filas hemos intentado poner fórmulas representativas de los distintos tipos de fórmulas de intervalo que se explicaron en el apartado 5.1.1. Así, las fórmulas incluidas en las filas 11 a 15, ambas inclusive, se corresponden con FIAs, mientras que las contenidas en las filas 16 a 22 son FINAs. Más concretamente:

- La fórmula de la fila 11 es una FPP anidada a una SMIA (ver Definición 5.14).

- La fila 12 contiene una FFB anidada a una SMIA, y, dado que el patrón de búsqueda no trivial de su última subfórmula sólo incluye una fórmula, puede decirse que se trata de una propiedad invariante anidada a una SMIA (ver Definición 5.19). Su negación, representada en la fila 13, se trata por tanto de una FE anidada a una SMIA (ver Definición 5.20). El propósito de incluir, tanto en estas dos filas como en las filas 18 y 19, una fórmula y su correspondiente negación es mostrar que la negación de una fórmula no supone una explosión exponencial en el tamaño del autómata resultante.

- La fórmula representada en la fila 14 es una FINA anidada a una SMIA (ver Definición 5.14), mientras que la que se muestra en la fila 15 es una FPM anidada a una SMIA (ver Definición 5.15), por lo que FBT tendrá que aplicar la regla de expansión correspondiente de la Tabla 5.2 a la hora de traducir esta última fórmula.

- La FINA de la fila 16 afirma que existe un intervalo (en el futuro reflexivo) cuyos extremos se definen respectivamente por las ocurrencias de `p0` y `p3`, de modo que en el primer estado del mismo se cumple la disyunción exclusiva de `p1` y `p2`.

- La fórmula de la fila 17 establece la imposibilidad de construir el intervalo especificado.

- El significado de la fórmula de la fila 18 es que si su intervalo más externo puede ser construido, entonces `!p2` se satisfaría invariantemente sobre cualquier estado de dicho intervalo. Por el contrario, su negación (fórmula de la fila 19) afirma que dicho intervalo existe (su construcción no puede fallar) y que `p2` se satisface eventualmente dentro de él.

- En las tres últimas filas, a diferencia de las anteriores, se presentan fórmulas que tienen al menos un patrón de búsqueda con más de una fórmula (búsqueda); de ahí que en ellas aparezca la coma, que es el operador que FBT utiliza para unir las fórmulas objetivo de las distintas búsquedas que forman un mismo patrón de búsqueda (ver Figura 6.4). Dentro de este grupo de fórmulas, las dos primeras definen un intervalo al que se anida una FPP (más concretamente, un literal), mientras que la última anida una MIA a un intervalo que define un sufijo de la ejecución completa (a partir del estado en que se cumple `p0`).

6.4. Comparativa entre LBT y FBT

Dado que, como se ha comentado al principio de la sección 6.1, para implementar FBT nos hemos basado en el traductor LBT, reutilizando parte de su código, en esta sección vamos a comparar ambas herramientas entre sí. Dicha comparación se establecerá en primer lugar a nivel de sus respectivos diseños e implementaciones (ver apartado 6.4.1), con el fin de indicar qué es lo que tienen en común y de resaltar las mejoras y optimizaciones que hemos introducido en FBT con respecto a LBT. Por último, en el apartado 6.4.2 se compararán los autómatas de Büchi que ambos traductores generan para la misma fórmula (si es aceptada por los dos) o para fórmulas equivalentes (caso de no ser admitida alguna fórmula por uno de ellos).

6.4.1. En cuanto a sus diseños e implementaciones

En FBT hemos seguido la misma sintaxis que se utiliza en LBT para todos aquellos componentes de fórmulas que son comúnmente aceptados por ambos traductores; así, éstos comparten la misma sintaxis para las constantes lógicas, los literales, los operadores proposicionales y ciertos operadores temporales de la LTL: `F`, `G`, `U` y `V` (cuyos significados se muestran en la Figura 6.4). Adicionalmente, FBT implementa el operador de *intervalo*, `[`, mientras que LBT incluye el operador `X` (*next*), que no hemos considerado en nuestra implementación por no estar definido ni tener una construcción equivalente en FIL.

LBT representa internamente y procesa cualquier fórmula en su *forma normal de negación*; así, si la fórmula que se le suministra como entrada contiene alguna negación que afecta a algún operador, entonces convierte dicha fórmula en otra equivalente, donde los operadores de negación sólo se aplican a las variables proposicionales. FBT conserva esta característica de empujar las negaciones tan dentro como sea posible, pero sólo para los operadores de la LTL, esto es, en las fórmulas que FBT procesa y almacena, una negación puede aparecer no sólo delante de una proposición atómica sino también delante del operador de intervalo[4] (compruébese esto en la segunda columna de la Tabla 6.1). Para ello aplica las reglas que se presentan en la Tabla 6.2, donde <f1> y <f2> hacen referencia a cualquier fórmula aceptada por FBT (ver Figura 6.3 y Figura 6.4).

Tabla 6.2. Relación existente entre los operadores de la LTL con respecto a la negación

Fórmula de entrada	Se almacena como
i <f1> <f2>	| !<f1> <f2>
!e <f1> <f2>	^ <f1> <f2>
!^ <f1> <f2>	e <f1> <f2>
!& <f1> <f2>	| !<f1> !<f2>
!| <f1> <f2>	& !<f1> !<f2>
!G <f1>	F !<f1>
!F <f1>	G !<f1>
!U <f1> <f2>	V !<f1> !<f2>
!V <f1> <f2>	U !<f1> !<f2>

[4] Obsérvese que las reglas de expansión (ver apartado 5.1.4) tienen en cuenta este hecho.

Como puede apreciarse, las reglas de la Tabla 6.2 muestran: (a) la relación existente entre los operadores `i` y `|`, (b) que los operadores `e` y `^` son opuestos, y (c) que los operadores `&` y `|` son duales, al igual que lo son los operadores `G` y `F` y `U` y `V` respectivamente. Puede comprobarse fácilmente que dichas reglas se han aplicado a las fórmulas contenidas en las filas 6, 9, 10, 15 y 16 de la Tabla 6.1.

En el Apéndice B se indica cómo se puede modificar la implementación actual de FBT para contemplar reglas similares a las mostradas en la Tabla 6.2, pero para el operador de intervalo, o sea, para poder empujar la negación colocada directamente delante de dicho operador dentro del intervalo. Para ello, habría que extender la sintaxis de las fórmulas aceptadas con dos nuevos operadores: *búsqueda fuerte* e *intervalo fuerte*, con el fin de obtener con facilidad la fórmula dual de una fórmula de intervalo.

Una vez que se han comentado los puntos fundamentales que tienen en común LBT y FBT, a continuación se resaltarán las principales diferencias existentes en el diseño e implementación de ambas herramientas:

(1) Durante el proceso de expansión de los nodos del grafo, FBT considera que dos nodos casan o concuerdan, esto es, representan al mismo estado, cuando contienen el mismo conjunto de literales y sus campos *Next* almacenan el mismo conjunto de fórmulas; por esta razón integra ambos nodos en uno (líneas 3-6 de la Figura 5.2). Sin embargo, LBT sólo fusiona dos nodos cuando los campos *Old* y *Next* de un nodo contienen respectivamente los mismos conjuntos de fórmulas que sus homólogos en el otro nodo. La ventaja de aplicar nuestro criterio consiste en que en muchos casos se consigue reducir el número de nodos generados con respecto a los que se producirían si se hubiera implementado la condición considerada en LBT, tal y como se pondrá de manifiesto en el Ejemplo 6.6 del apartado 6.4.2.

(2) Lo primero que FBT hace con la fórmula elegida, η, del campo *New* es comprobar si su negación ($\neg\eta$) está incluida en el campo *Old*, en cuyo caso ese nodo debe descartarse (líneas 11-13 de la Figura 5.2). De este modo, se detectan rápidamente las contradicciones en un nodo, evitando tener que procesar un montón de fórmulas de un nodo (que finalmente se deberá descartar) hasta llegar al nivel de los literales, que es donde se detectan las contradicciones en LBT. En una implementación anterior de nuestro traductor, también se detectaban las contradicciones en este mismo nivel (en los literales), pero se

introdujo una mejora con respecto a la implementación efectuada en LBT. Así, la búsqueda de la negación del literal correspondiente se realizaba sobre el conjunto *m_atomic*[5], que sólo contiene los literales del nodo, en lugar de realizarla como LBT sobre el campo *Old*, que no sólo almacena los literales, sino cualquier fórmula procesada en el mismo. Con ello conseguíamos llevar a cabo dicha comprobación de una forma más rápida y eficiente.

(3) En FBT, el proceso de expansión de las fórmulas con operadores temporales (fórmulas de intervalo) se basa en la ocurrencia de alguno de sus reductores o en que ninguno de ellos se dé, en cuyo caso se introduce explícitamente la negación de todos sus reductores en el nodo que representa ese caso (líneas 32-33 de la Figura 5.2). Por el contrario, al expandir una fórmula cuyo operador principal es un operador temporal (`G`, `F`, `U` y `V`), LBT sólo introduce su fórmula "reductora" (utilizando el mismo término que en FIL), no añadiendo la negación de la misma al nodo que representa el caso alternativo en el que la fórmula analizada no se "reduce". El siguiente ejemplo trata de aclarar esto:

Ejemplo 6.3: Comparación de las reglas de expansión aplicadas por LBT y FBT para la fórmula `Fp1`

La Tabla 6.3 muestra las reglas de expansión que LBT y FBT aplican respectivamente a la fórmula almacenada en la fila 2 de la Tabla 6.1. En ella puede observarse que, aparte de cómo se representa internamente dicha fórmula en cada traductor, la única diferencia existente está en que FBT introduce `!p1` en el campo *New* del nodo actual, mientras que LBT no lo hace. Repárese en que la regla mostrada para FBT es una instanciación de la regla general para ese tipo de fórmulas, expuesta en la fila 23 de la Tabla 5.3.

Tabla 6.3. Reglas de expansión que LBT y FBT aplican para la fórmula `Fp1`

	Fórmula		*NewNode*	*CurrentNode*	
Traductor	Introducida	Procesada	*New*	*New*	*Next*
LBT	`Fp1`	`Fp1`	{`p1`}	—	{`Fp1`}
FBT	`Fp1`	`![p1 > f`	{`p1`}	{`!p1`}	{`![p1 > f`}

[5] Recuérdese que en el apartado 6.1.1 se dijo que este conjunto contenía los literales almacenados en el campo *Old* (*m_old* en la implementación) de ese nodo.

(4) Para el establecimiento de los conjuntos o condiciones de aceptación (subapartado 5.1.6.2), nuestro traductor emplea una heurística mejorada con respecto a la utilizada en LBT, donde la búsqueda de las fórmulas que expresan eventualidad, o sea, las que tienen como operador principal el operador U (*strong until*) o el operador F (*finally, eventually*), se realiza sobre el campo *Old* de los nodos almacenados en el conjunto *GraphNodes*, en lugar de sobre su campo *Next*, como hace FBT. Obviamente, este último campo contiene bastantes menos fórmulas que el primero, dado que en él sólo se almacenarán las fórmulas temporales (fórmulas de intervalo, en el caso de FBT) que se cumplen, pero no inmediatamente, en el estado que representa ese nodo, mientras que el campo *Old* alberga no sólo dichas fórmulas, sino también una serie de fórmulas proposicionales, además de todas aquellas fórmulas temporales que se reducen (se satisfacen total o parcialmente) en dicho estado. Por lo tanto, esta heurística permite que FBT calcule más rápidamente las condiciones de aceptación.

(5) Para determinar los estados de aceptación (subapartado 5.1.6.3), esto es, los estados que pertenecen a cada conjunto de aceptación, FBT sólo ha de comprobar que ninguna de las fórmulas de eventualidad asociadas a la condición de aceptación correspondiente se encuentra almacenada en el campo *Next* de un nodo. Efectuar estas comprobaciones sobre el campo *Next* (en vez de sobre el campo *Old*, que es donde lo hace LBT) de nuevo aporta ventajas (rapidez y sencillez) a nuestro procedimiento. Además, para determinar si un estado pertenece o no a un conjunto de aceptación, LBT no sólo ha de comprobar la presencia o ausencia de la fórmula que expresa la correspondiente eventualidad (fórmula del tipo F <f> o U <g> <f>, donde <f> y <g> representan a cualquier fórmula aceptada por LBT), sino también la de la fórmula que la satisface (<f>). Todo ello hace que el procedimiento para calcular los estados de aceptación sea más complejo en LBT que en FBT.

6.4.2. En los resultados obtenidos

Como ya se ha comentado, FBT no sólo reconoce el operador temporal propio de FIL, es decir, el operador de intervalo, sino también todos los operadores temporales que LBT admite, excepto el operador X (*next*); pero, eso sí, como abreviaciones de las correspondientes fórmulas de intervalo FIL (nuestra herramienta sólo almacena y procesa éstas últimas). Así, el propósito de la

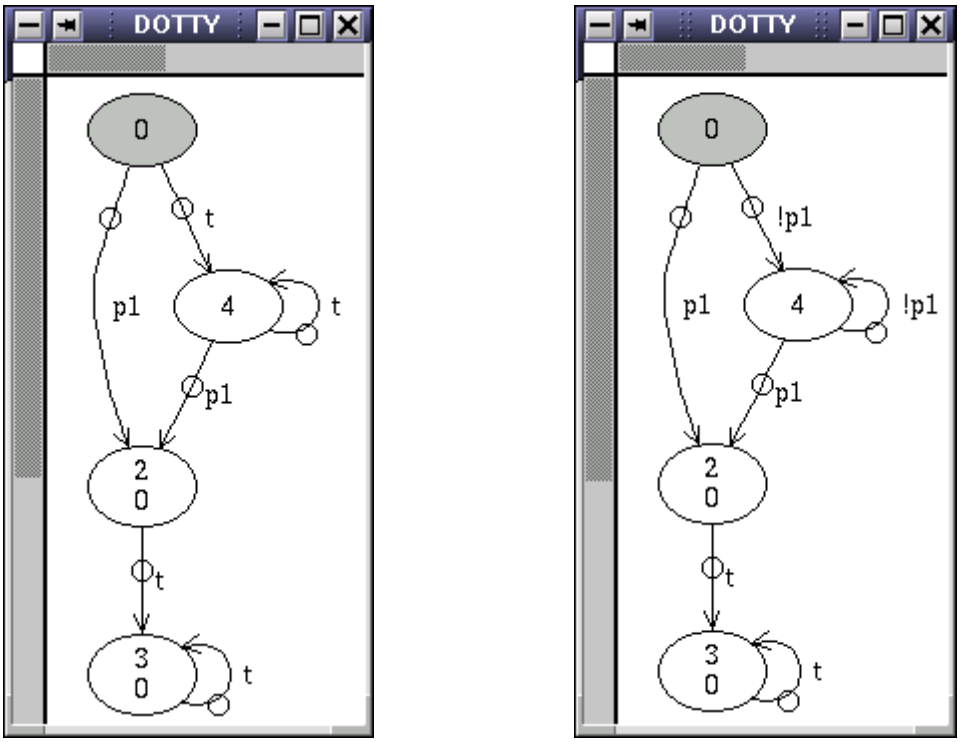
inclusión de las fórmulas que aparecen en las diez primeras filas de la Tabla 6.1 es el de que ambos traductores las puedan aceptar directamente como entrada. Los resultados obtenidos para las fórmulas más simples de este grupo (filas 1 y 2) indican que ambas herramientas generan autómatas de Büchi con la misma complejidad. En el Ejemplo 6.4 se comparan los autómatas producidos para una de esas fórmulas, concretamente la contenida en la fila 2. Para la fórmula de la fila 1, ambos traductores obtienen exactamente el mismo autómata. Por otra parte, para las fórmulas que contienen al operador U (*strong until*) o a su dual V (*release*), LBT produce normalmente autómatas algo más simples que FBT. El Ejemplo 6.5 muestra esto para la fórmula de la fila 3, ocurriendo algo similar para la de la fila 4. Sin embargo, por regla general, FBT genera autómatas más simples que LBT para las fórmulas más complejas del grupo mencionado (contenidas en las filas 5 a 10). Así, excepto para la fórmula de la fila 7, para la que LBT produce un autómata de 4 nodos y 9 arcos (algo más simple que el generado por FBT, con 5 nodos y 15 arcos), el resto de dichas fórmulas dan como resultado autómatas menos complejos por parte de FBT. Dos de estos casos se comentan en el Ejemplo 6.6 y en el Ejemplo 6.7 respectivamente.

Como LBT no admite directamente como entrada las fórmulas de intervalo, para proporcionarle una fórmula equivalente a las contenidas en las doce últimas filas de la Tabla 6.1 habría que recurrir al anidamiento de operadores *hasta*, tal y como se indicó en el apartado 2.3.3, lo que hace que la fórmula procesada por LBT sea más compleja que la procesada por FBT y, por tanto, también lo sean los autómatas resultantes. El Ejemplo 6.8 aporta más detalles acerca de esto para la fórmula de la fila 18.

Ejemplo 6.4: Comparación de resultados obtenidos por LBT y FBT para la fórmula `Fp1`

La Figura 6.10 muestra los autómatas de Büchi que LBT y FBT generan a partir de la fórmula contenida en la fila 2 de la Tabla 6.1. Repárese en que ambos grafos son equivalentes. La única diferencia observable en ellos, o sea, las etiquetas en las arcos entrantes al nodo 4, es consecuencia de lo comentado en el punto (3) del apartado 6.4.1. Recuérdese que la etiqueta '`t`' debe interpretarse como "cualquier combinación posible de los literales que se pueden formar a partir del conjunto de proposiciones atómicas considerado". Así, para la fórmula de este ejemplo, FBT

construye un autómata determinista[6], mientras que el generado por LBT es no determinista, ya que si se produce `p1`, estando en el estado 0 o en el 4, podemos pasar tanto al nodo 2 como al 4. Esto resta algo de intuición al grafo de la Figura 6.10(a) con respecto al de la Figura 6.10(b), a la hora de ser interpretado por un ser humano. Consecuentemente, y esto suele ser la tónica general para casi todas las fórmulas analizadas por ambos traductores, los grafos generados por FBT son más intuitivos y fáciles de entender para el usuario de una herramienta de este tipo.



(a) **(b)**

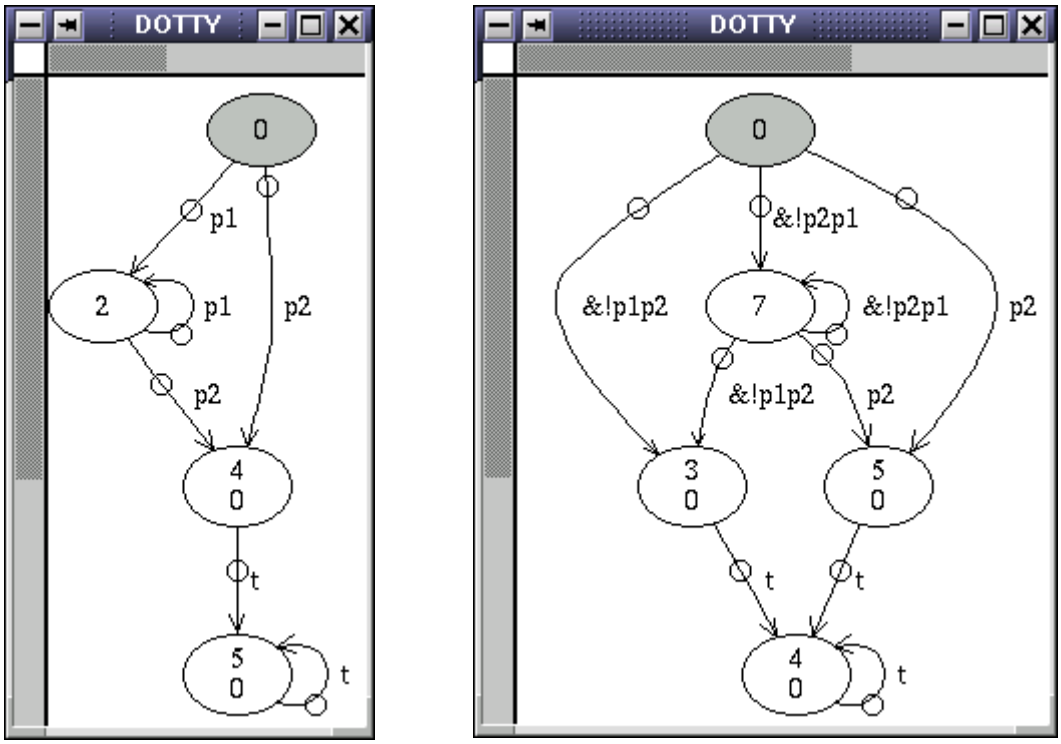
Figura 6.10. Autómatas de Büchi generados para la fórmula `Fp1` por: **(a)** LBT; **(b)** FBT

Ejemplo 6.5: Comparación de resultados obtenidos por LBT y FBT para la fórmula `U p1 p2`

La Figura 6.11 muestra los autómatas de Büchi que ambos traductores construyen para la fórmula `U p1 p2`. En la fila 3 de la Tabla 6.1 puede verse que FBT la transforma automáticamente en la fórmula de intervalo `![|!p1p2 > !p2`, siendo ambas semánticamente equivalentes. En este caso, el grafo producido por LBT es un poco más simple que el que FBT genera. Esto quizás sea debido a que la fórmula

[6] Aunque, por regla general, FBT produce autómatas *no deterministas*.

de intervalo que FBT analiza intenta "simular" la propiedad que, de un modo más simple, establece el operador *hasta*.

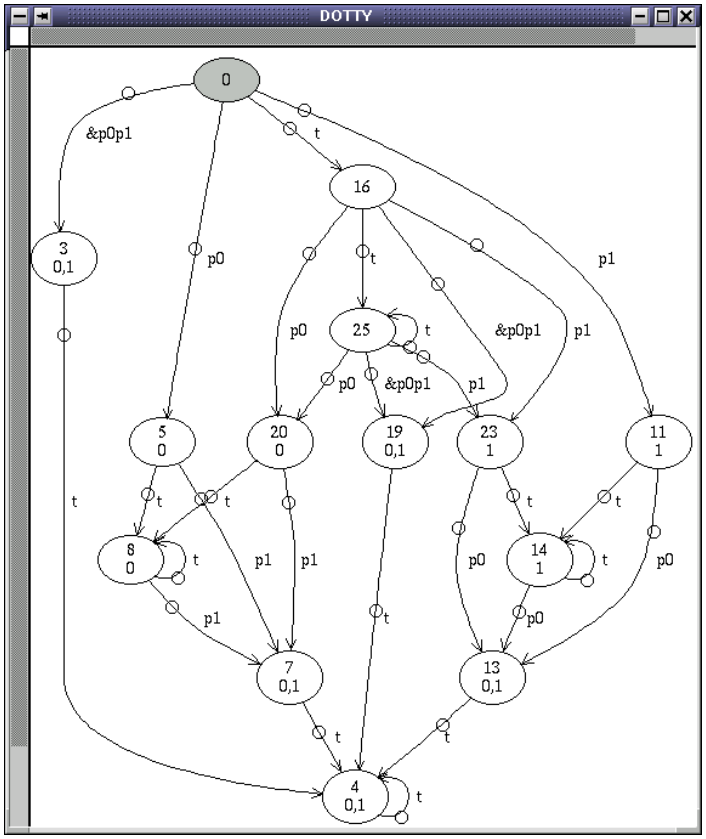


(a) **(b)**

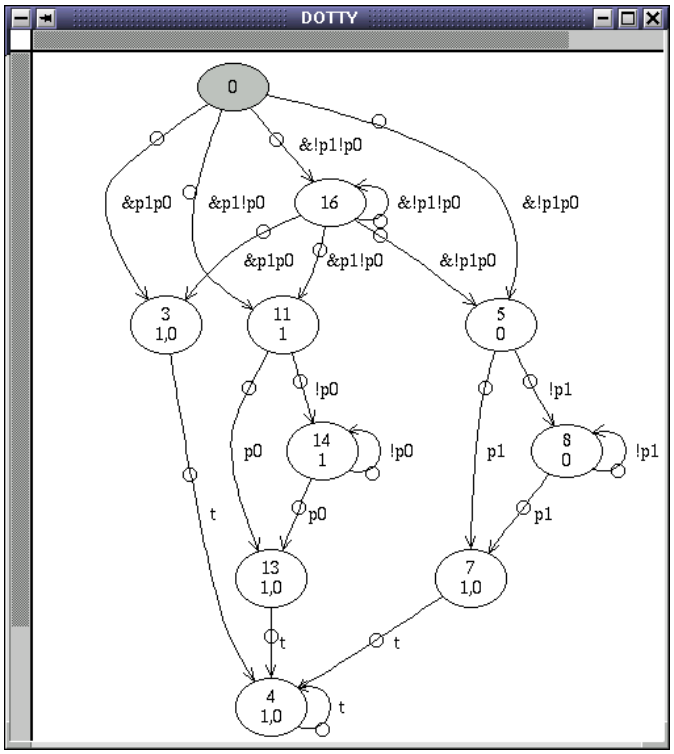
Figura 6.11. Autómatas de Büchi generados para la fórmula `U p1 p2` por: **(a)** LBT; **(b)** FBT

Ejemplo 6.6: Comparación de resultados obtenidos por LBT y FBT para la fórmula `& Fp0 Fp1`

La Figura 6.12 muestra cómo, para la fórmula almacenada en la fila 8 de la Tabla 6.1, FBT genera un autómata más simple que el construido por LBT. Obsérvese que en éste último (Figura 6.12(a)) los arcos entrantes a los nodos 5 y 20 tienen la misma etiqueta (`p0`), lo que indica que contienen el mismo conjunto de literales; además, tienen los mismos sucesores (nodos 7 y 8). Esto significa que están representando al mismo estado, por lo que su duplicidad es redundante, pudiéndose simplificar el grafo si ambos se fundieran en un único nodo. Lo mismo es aplicable a los nodos 11 y 23. Esto, que ocurre a menudo en los autómatas generados por LBT, no se produce nunca en los obtenidos con FBT, gracias a lo explicado en el punto (1) del apartado 6.4.1.



(a)



(b)

Figura 6.12. Autómatas de Büchi generados para la fórmula `& Fp0 Fp1` por: **(a)** LBT; **(b)** FBT

Ejemplo 6.7: Comparación de resultados obtenidos por LBT y FBT para la fórmula `!e FFp1 Fp1`

En la Figura 6.13 se muestran los autómatas que LBT y FBT generan a partir de la fórmula contenida en la fila 9 de la Tabla 6.1. Ninguno de ellos acepta secuencia alguna. Obsérvese que el de LBT no tiene ningún estado etiquetado con los dos identificadores de sus conjuntos de aceptación y que no existe ningún ciclo que recorra estados de aceptación de uno y otro conjunto. Como se explicó en la sección 6.3, el de FBT tiene un conjunto de aceptación, pero ningún estado pertenece a él. En ocasiones, se utilizan fórmulas que no son satisfacibles, como la de este ejemplo, para verificar que su negación es una fórmula válida, es decir, una tautología.

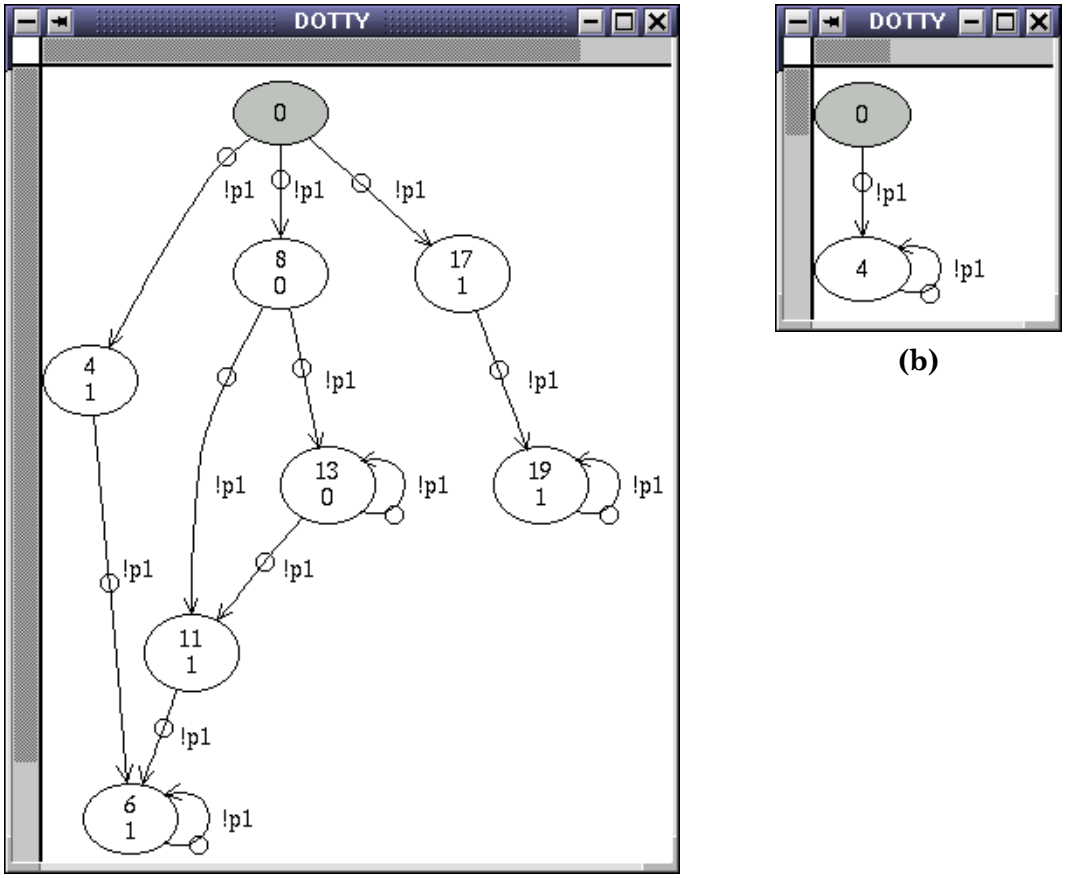


Figura 6.13. Autómatas de Büchi generados para la fórmula `!e FFp1 Fp1` por: **(a)** LBT; **(b)** FBT

Según afirman sus propios autores, LBT es una implementación altamente optimizada del algoritmo presentado en [Gerth95]. De hecho, al final del artículo citado aparece, dentro de una tabla con resultados experimentales, la fórmula de este ejemplo, indicándose que el autómata obtenido para ella tenía 22 nodos, 41

arcos y 2 conjuntos de aceptación. Así, puede verse (Figura 6.13(a)) que la reducción de tamaño lograda por LBT con respecto al algoritmo original es considerable. Pues bien, obsérvese que FBT consigue reducir aún más el tamaño del autómata resultante (Figura 6.13(b)).

Ejemplo 6.8: Comparación de resultados obtenidos por LBT y FBT para la fórmula `[p0 p1 [p2 > f`

Dado que LBT no reconoce la fórmula `[p0 p1 [p2 > f`, habrá que suministrarle se fórmula equivalente expresada en la sintaxis que admite. Recuérdese que la semántica asociada a dicha fórmula indica que ésta se satisface siempre y cuando se cumpla una de estas cuatro condiciones:

(a) Que la primera ocurrencia de `p0` se dé estrictamente antes que la primera ocurrencia de `p1`, y desde el mismo instante en que se dé `p0` debe satisfacerse invariantemente `!p2` hasta que `p1` se cumpla. Esto se expresa en la sintaxis admitida por LBT mediante la siguiente fórmula: `U !p1 &p0 U !p2 p1`.

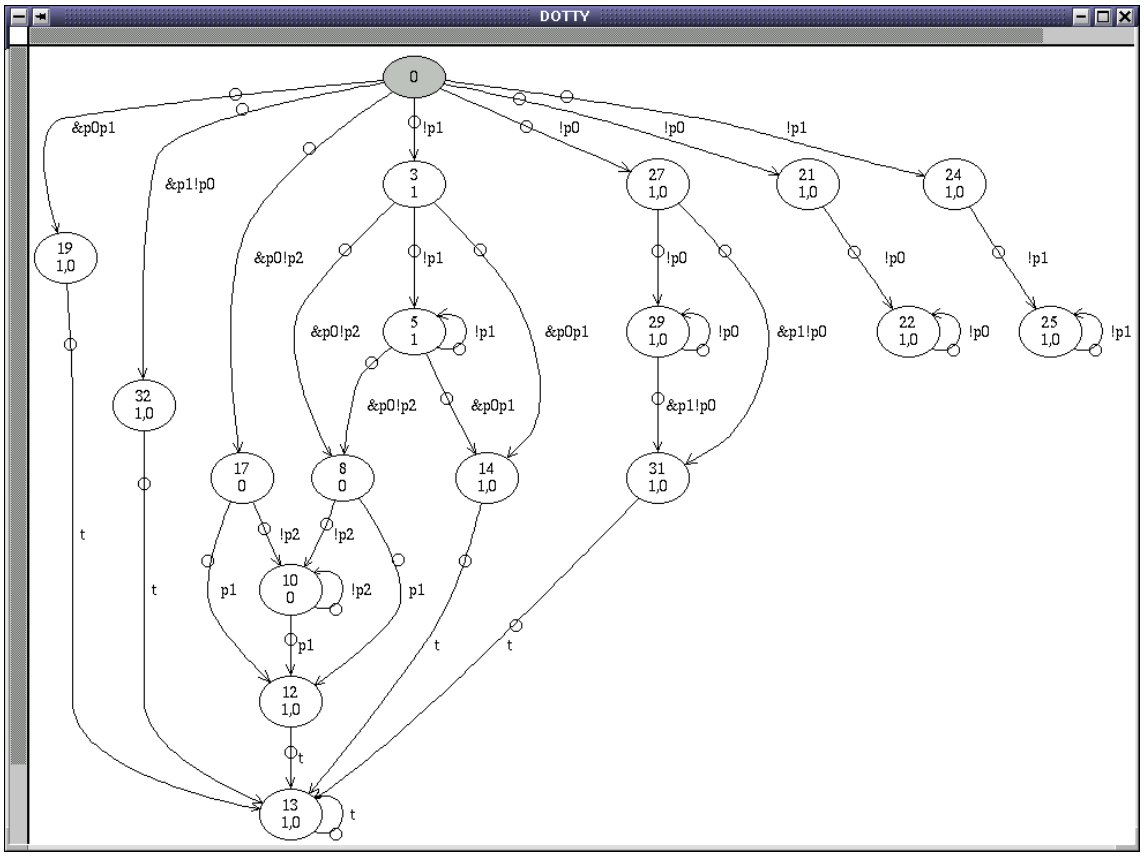
(b) Que no haya ningún estado en el futuro en el que se satisfaga `p0`, lo que equivale a decir que se cumpla `G!p0`.

(c) Que nunca se satisfaga `p1`, esto es, que `G!p1` sea cierto.

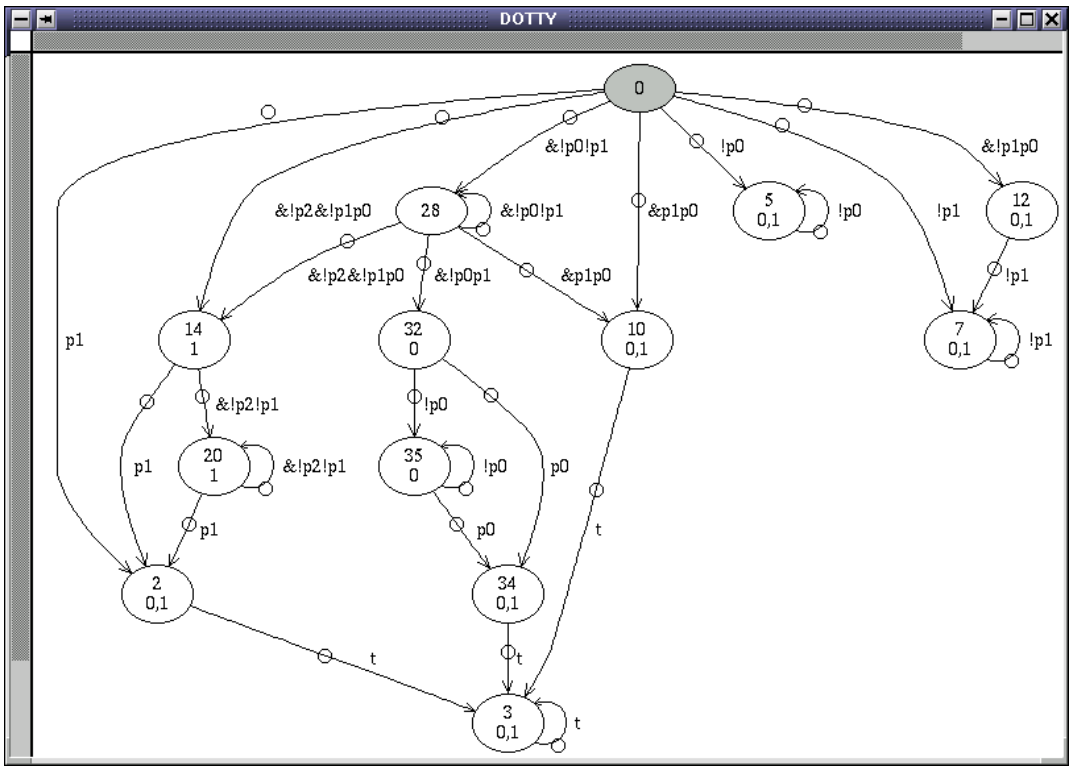
(d) Que `p1` preceda a `p0`, lo que en LBT se expresa mediante la fórmula `V p1 !p0`, que afirma que tanto en el estado donde se satisfaga `p1` como en todos los anteriores a él se da `!p0`.

Por tanto, la fórmula equivalente a `[p0 p1 [p2 > f` que se le debe introducir como entrada a LBT es la disyunción de las cuatro fórmulas que se acaban de explicar, o sea: `| U !p1 &p0 U !p2 p1 | G!p0 | G!p1 V p1 !p0`.

Obsérvese que la fórmula de intervalo de este ejemplo expresa de forma más concisa y elegante la propiedad explicada. Además, el análisis y descomposición que FBT hace de ella, produce como resultado un autómata (Figura 6.14(b)) más simple que el que LBT genera para su fórmula equivalente (Figura 6.14(a)).



(a)



(b)

Figura 6.14. Autómatas de Büchi generados para la fórmula `[p0 p1 [p2 > f` por: **(a)** LBT; **(b)** FBT

Conclusión: Aunque tanto el número de nodos de los autómatas generados por ambas herramientas como el tiempo que emplean en construirlos es exponencial con respecto al tamaño de la fórmula introducida, la experiencia nos muestra que los resultados obtenidos para las especificaciones más utilizadas en verificación son, por regla general, pequeños. En este apartado se ha mostrado que los autómatas resultantes al ejecutar ambos traductores para fórmulas equivalentes son de complejidad similar, saliendo ligeramente favorecida nuestra herramienta en la mayoría de los casos analizados. Además, como apunta Doron Peled en [Peled01a], debe tenerse en cuenta que un buen formalismo de especificación es aquél que en la práctica describe las propiedades utilizadas más frecuentemente con especificaciones relativamente cortas y no difíciles de comprobar. Por todo ello, podemos concluir que FIL es un buen formalismo de especificación y que FBT es una buena herramienta para la traducción eficiente de sus fórmulas a autómatas de Büchi.

ÍNDICE DE REFERENCIAS EMPLEADAS

[Deitel99], *155*
[France86], *169*
[Gansne00], *164*
[Gerth95], *155*, *179*
[Naur60], *159*
[Peled01a], *182*
[Rumbau96], *156*
[Rumbau99], *156*
[Strous93], *155*

Capítulo 7

CONCLUSIONES

CONTENIDO

RESUMEN Y ORGANIZACIÓN

Como es norma habitual en la redacción de una memoria de tesis, este último capítulo se dedica a las conclusiones y consideraciones finales. Así, en la primera de sus secciones se presentan los trabajos relacionados con el nuestro desde diferentes perspectivas: formalismos para razonamiento temporal, enfoques para generación de autómatas y su aplicación a distintas técnicas de verificación y formalismos con representación visual. En la segunda sección se exponen las conclusiones propiamente dichas del trabajo de investigación que se ha llevado a cabo, destacando las principales contribuciones y aportaciones del mismo. Finalmente, en la tercera sección se indica cómo podemos continuar este trabajo, es decir, se aportan ideas sobre posibles orientaciones o líneas futuras de investigación y actuación.

7.1. Trabajos relacionados

En esta sección se van a presentar los trabajos previos relacionados de algún modo con el nuestro. Para ello, los contemplaremos desde distintas ópticas. Así, en el primer apartado se exponen los principales formalismos existentes para razonamiento temporal, centrándonos especialmente en los formalismos lógicos y describiendo la relación existente entre la lógica utilizada en esta tesis (FIL) y aquellos formalismos más próximos a ella. En el apartado 7.1.2 se revisan los principales enfoques existentes para generar autómatas a partir de fórmulas de una lógica temporal lineal (dado que FIL pertenece a este grupo concreto de lógicas) y su aplicación a diferentes técnicas de verificación, empleando todas ellas una exploración explícita de su espacio de estados (puesto que éste es el ámbito de aplicación de nuestro trabajo). Finalmente, en el apartado 7.1.3 se presentan diferentes formalismos con representación visual aplicables en distintos momentos del ciclo de vida del desarrollo de un sistema, tratando de mostrar el creciente interés de los investigadores y profesionales implicados en este campo en la utilización de formalismos gráficos, ya que se ha demostrado que facilitan la compresión y el razonamiento. Aunque este último aspecto (la representación gráfica) no se trata con gran profundidad en esta tesis, no debe olvidarse que uno de sus objetivos es el de propiciar la implementación de herramientas amigables con representaciones gráficas intuitivas que ayuden en la difícil tarea de especificar y verificar sistemas concurrentes y reactivos, lo que se propone como continuación del trabajo hasta ahora desarrollado (ver sección 7.3).

7.1.1. Formalismos para razonamiento temporal

En 1977, Amir Pnueli introdujo la lógica temporal como formalismo para razonar con los programas concurrentes [Pnueli77]. Desde entonces, muchos trabajos de bastantes investigadores han hecho que dicha lógica sea un formalismo efectivo para la especificación y verificación de sistemas concurrentes y reactivos [Manna82] [Pnueli86a] [Emerso90] [Manna92] [Manna95].

Dentro de las lógicas temporales hay distintas variantes o clases, que difieren fundamentalmente en la representación subyacente del tiempo y en la expresividad de sus operadores temporales. Entre ellas, la variante más conocida y utilizada es la LTL (*Linear Temporal Logic*), que representa el tiempo de forma discreta y lineal, con punto inicial, pero sin punto final (o sea, isomorfo con el conjunto de los

enteros no negativos) y cuyos operadores temporales son: *siempre* (*always*), *eventualmente* (*eventually*), *siguiente* (*next*) y *hasta* (*until*). También se conocen varias extensiones de la LTL, entre ellas las que añaden: operadores que hacen referencia al pasado [Lichte85b], acciones [Lampor94], o elementos para trabajar con tiempo real [Alur89] [Ostrof89]. Otra variante de la LTL es la que utiliza un modelo de tiempo continuo (o sea, denso, no discreto) [Barrin86].

Por otra parte, las lógicas de tiempo ramificado [Ben-Ar83] representan el tiempo con una estructura ramificada (en forma de árbol), donde cada instante tiene varios sucesores, permitiendo así especificar los distintos comportamientos posibles de un sistema. De entre ellas, la CTL (*Computational Tree Logic*) [Clarke86] es la más utilizada. También ha habido intentos, como por ejemplo [Pinter84], para definir una lógica cuyo dominio de interpretación sea un orden parcial.

Pratt introdujo la *Lógica de Procesos* [Pratt79], que fue más tarde refinada en [Harel82]. En esta lógica, las fórmulas se interpretan sobre intervalos o "caminos" de una ejecución. Surgieron a continuación una serie de lógicas de procesos más potentes [Harel84] [Kozen90]. Sin embargo, y como era de esperar, se vio que existía una relación directa entre la expresividad de una lógica y la complejidad de su problema de decisión, descubriéndose que la mayoría de las lógicas de procesos más expresivas o eran indecidibles o su procedimiento de decisión era no elemental (ver página 811 de [Kozen90]). A partir de aquí, y dada la creciente necesidad de herramientas automatizadas que soportaran los correspondientes formalismos, el interés se fue desplazando desde la búsqueda de formalismos con una expresividad cada vez mayor hacia la utilización de formalismos que contaran con un algoritmo de decisión eficiente. Así, se reconoció que la expresividad de la lógica temporal con el operador *hasta* (*until*) era suficiente para establecer la mayoría de las propiedades que son de interés a la hora de especificar los sistema concurrentes [Emerso90].

Las lógicas temporales tradicionales permiten razonar acerca de las propiedades que se aplican a todos los estados de una ejecución del sistema considerado. Sin embargo, no es fácil especificar con ellas propiedades que sólo deben cumplirse en determinados intervalos o fragmentos de una ejecución. Para este último fin surgieron las *lógicas de intervalos* [Allen83] [Halper83] [Schwar83] [Halper91], que son lógicas temporales de alto nivel que permiten restringir el

ámbito del razonamiento a contextos específicos de una ejecución, estableciendo los correspondientes intervalos de tiempo. Estas lógicas de intervalos también facilitan el razonamiento composicional (en el sentido de que un intervalo grande puede formarse a partir de otros más pequeños) y jerárquico (pudiéndose establecer distintos niveles de abstracción, de modo que un intervalo de una especificación más abstracta puede ser refinado en otro "más grande" de una especificación más concreta).

En 1983, Schwartz, Melliar-Smith y Vogt introdujeron IL (*Interval Logic*) [Schwar83], una lógica temporal de alto nivel que utiliza operadores de *búsqueda* para construir intervalos temporales sobre los que poder establecer propiedades para determinar su cumplimiento. Sobre el mismo tiempo y de forma independiente, Halpern, Manna y Moszkowski crearon la ITL (*Interval Temporal Logic*) [Halper83], una extensión de la lógica temporal en la que las fórmulas se interpretan sobre caminos, y que proporciona dos operadores temporales: *siguiente* (*next*) y *corte* (*chop*). Entre ambas lógicas, IL e ITL, existen importantes diferencias semánticas y de representación, siendo distintas en expresividad, complejidad y procedimientos de decisión. Sin embargo, ambas tratan adecuadamente la cuestión de la representación de contextos temporales, pero sufren el problema de la complejidad computacional, ya que en ambos casos el mejor procedimiento de decisión disponible es no elemental [Plaist83] [Halper83]. Lo mismo ocurre en [Rosner86], donde se describe una lógica temporal proposicional con el operador *corte* (*chop*). Así, la verificación automatizada usando estas lógicas es difícil de solucionar algorítmicamente. *Duration Calculus* [Chaoch91] es una extensión de la ITL que permite el razonamiento con las integrales de las duraciones de los intervalos sobre los que se cumplen los predicados o propiedades.

Un enfoque alternativo, desarrollado en [Allen83] y en [Halper91], considera los intervalos como primitivas. En este enfoque, las proposiciones atómicas se cumplen para (todo) el intervalo, proporcionándose modalidades para relacionar intervalos y adoptándose varias formas de tiempo. Sin embargo, en su forma más general, estas lógicas se muestran como indecidibles en [Halper91], donde también se introducen algunas simplificaciones, bajo las cuales algunos fragmentos se hacen decidibles, pero el algoritmo de decisión es aún así no elemental. Las lógicas de intervalos de [Halper91] han sido adicionalmente desarrolladas y sus teorías de demostración investigadas por Venema en una serie de artículos (ver [Venema94] para referencias relacionadas).

FIL (*Future Interval Logic*) [Ramakr93a] está inspirada en gran parte en IL, a partir de la que ha evolucionado. Sin embargo, y a diferencia de ésta, FIL cuenta con un procedimiento de decisión elemental [Ramakr92] [Ramakr96a]. Al igual que IL, FIL utiliza operadores de *búsqueda* para construir explícitamente los intervalos. De este modo, en ambas lógicas (al igual que en ITL) los intervalos se derivan a partir de secuencias de estados y de las transiciones que forman sus extremos. No obstante, FIL difiere de IL en su sintaxis y semántica. Así, mientras que en IL la búsqueda del extremo derecho de un intervalo comienza en el extremo izquierdo del mismo, en FIL cada uno de sus extremos es localizado por una secuencia de búsquedas independientes que comienzan en el mismo punto. Otra diferencia es que en IL las búsquedas localizan eventos que marcan cambios en los valores de las fórmulas, mientras que en FIL las búsquedas localizan estados en los que las fórmulas se cumplen. Esta semántica basada en estados es con frecuencia más conveniente para su utilización en especificación y verificación. También existe una extensión de FIL para tiempo real, denominada RTFIL (*Real-Time Future Interval Logic*), para la que también existe un procedimiento de decisión elemental [Ramakr93b] [Ramakr96b].

ISL (*Interval Specification Logic*) [Goswam92] es otra lógica basada en intervalos que, sin su constructo de tiempo real, es bastante parecida a FIL, aunque el uso y la construcción de las modalidades de intervalos está bastante restringido en ISL. Sin embargo, esta lógica permite cuantificación sobre el número de ocurrencias de eventos asociados con una fórmula, lo que, unido a su capacidad para construir intervalos, la hace apropiada para su aplicación a metodologías de razonamiento basadas en refinamiento.

En lo que a nuestro trabajo se refiere, desde un primer momento hemos estado interesados en la aplicación de formalismos lógicos a la especificación y verificación de sistema reactivos. Así, en nuestro primer trabajo en este campo [Hornos97] utilizamos como formalismo lógico subyacente la LTL, pero con una semántica más restrictiva, en el sentido de que en cada instante de una ejecución sólo se permitía la ocurrencia de un único suceso o evento. Poco tiempo después descubrimos FIL y su intuitiva representación gráfica, GIL, pensando que era un formalismo bastante atractivo y de gran valor práctico, por lo que a partir de [Hornos99] nuestro trabajo de investigación se ha centrado en la aplicación de esta lógica a la especificación y verificación de sistemas concurrentes y reactivos.

Desde un punto de vista más algebraico, otra dirección distinta a los enfoques lógicos mencionados en este apartado la constituyen las *Álgebras de Procesos* [Baeten90]. Trabajos especialmente relevantes dentro de esta área de investigación han sido la teoría de CSP (*Communicating Sequential Processes*) [Hoare78] y el desarrollo de CCS (*Calculus of Communicating Systems*) [Milner80]. También se han hecho esfuerzos, como por ejemplo el de [Stirli88], para combinar estas álgebras con enfoques de lógicas temporales.

7.1.2. Generación de autómatas a partir de fórmulas lógicas y su aplicación a distintas técnicas de verificación

La primera traducción de una fórmula LTL a un autómata de Büchi se obtuvo mediante la intersección de dos autómatas [Wolper83]: el *autómata local*, que reconoce el componente de seguridad de la fórmula, y el *autómata de eventualidad*, que reconoce su componente de vivacidad. Éste es el denominado *enfoque basado en teoría de autómatas* (*automata-theoretic approach*), que se propuso para mostrar la conexión teórica existente entre la LTL y los autómatas de Büchi y para establecer su corrección. Se trata de una construcción *global*, que primeramente genera todos los nodos posibles (uno para cada conjunto maximalmente consistente de subfórmulas de la propiedad) para ambos autómatas. A continuación se construyen los arcos entre pares de nodos, si se satisfacen ciertas condiciones de consistencia. Finalmente, se halla el producto de ambos autómatas. Sólo al final es posible comprobar qué nodos son realmente alcanzables desde los estados iniciales. Aunque es una manera simple de describir la construcción del autómata, claramente no es una buena forma de implementarla, ya que inmediatamente produce el peor caso de complejidad exponencial.

Este enfoque, ahora clásico, fue utilizado para resolver el problema de la comprobación de modelos para la LTL en [Vardi86], donde primeramente se construyen ambos espacios de estados, representados mediante autómatas de Büchi: uno para el sistema a ser verificado y otro para la *negación* de la propiedad. Éste último incluye todas las secuencias de ejecución (modelos) que violan la propiedad. A continuación se analizan conjuntamente los dos espacios de estados para comprobar si comparten alguna secuencia de ejecución, en cuyo caso significaría que el sistema puede incumplir la propiedad. Por esta razón, en el momento en que se encuentra una de estas secuencias, el análisis finaliza y dicha secuencia se devuelve como contraejemplo. En caso contrario, se puede afirmar que

todas las secuencias infinitas de ejecución generadas a partir del espacio de estados del sistema satisfacen (son modelos de) la especificación, o en otras palabras, que ninguna de ellas la incumple. Por lo tanto, este análisis es equivalente al problema de verificar si el lenguaje aceptado por el producto (síncrono) de ambos autómatas es *vacío*, al que normalmente se conoce como *comprobación del vacío del autómata producto*. Un enfoque general para resolver este problema consiste en comprobar los *componentes fuertemente conectados*, que es lo que se hace en [Lichte85a].

Una construcción que también sigue el mencionado enfoque basado en teoría de autómatas se emplea en [Ramakr96a], pero esta vez para comprobar la *satisfacibilidad* de una fórmula FIL, de tal modo que se determina que una fórmula es satisfacible si y sólo si el lenguaje aceptado por el autómata de propiedad es *no vacío*. Para ello, dicho autómata debe aceptar alguna secuencia. Así, la comprobación de la satisfacibilidad de una fórmula se reduce al problema de comprobar el *no vacío* de su autómata de propiedad. FIL es, según sus creadores, la primera lógica de intervalos para la que se ha conseguido obtener un procedimiento de decisión elemental. Este problema es, por lo general, no elemental y a menudo indecidible para el resto de las lógicas de intervalos, en parte debido a la dificultad de codificar sucintamente la noción de subcontexto (establecido por un determinado intervalo) dentro de las ejecuciones del autómata.

El peor caso de complejidad exponencial producido por el enfoque basado en teoría de autómatas puede evitarse a menudo siguiendo el denominado método *tableau* [D'Agos99], que construye el autómata incrementalmente. Este método, del que se ofrece una excelente revisión general en [Wolper85], analiza simultáneamente y de forma eficaz un conjunto completo de estados del autómata, evitando además tener que explorar los estados inalcanzables del mismo. Ambos factores ayudan con frecuencia a reducir sustancialmente las necesidades de espacio en memoria y de tiempo de ejecución del correspondiente algoritmo.

Una construcción basada en *tableau* para la LTL [Kesten93] empieza con un autómata de dos nodos, que se refina repetidamente. Así, en cada paso, un par de nodos adyacentes es globalmente inspeccionado para comprobar si satisfacen o no las condiciones de consistencia del *tableau*. En caso de no cumplirlas, uno de esos nodos se sustituye por un conjunto de nodos que las satisfacen. Este algoritmo, que comprueba la satisfacibilidad de una fórmula, continua refinando nodos hasta que todos los arcos satisfacen esas condiciones, para lo que repetidamente inspecciona y

corrige el grafo completo (añadiendo y eliminando nodos y arcos). Con este algoritmo, se requiere finalizar completamente la construcción del autómata de propiedad antes de poder aplicarlo a la comprobación de modelos; por consiguiente, no puede ser utilizado en comprobación de modelos *on-the-fly*.

Otra construcción basada en *tableau* más reciente y eficiente para la LTL opera *on-the-fly* [Gerth95], en el sentido de que cada nodo se genera sólo cuando se necesita. Comienza construyendo un nodo para la fórmula, que es expandido en sucesivos pasos siguiendo una BPP muy simple. Una versión mejorada de este algoritmo se presenta en [Daniel99], que construye autómatas más pequeños y en menos tiempo. Las mejoras se basan en técnicas sintácticas que permiten eliminar la necesidad de almacenar cierta información en los nodos generados. Aunque ambos algoritmos pueden emplearse para comprobar la satisfacibilidad de una fórmula, están pensados para ser aplicados en *comprobación de modelos on-the-fly*, generando cada nodo del autómata sólo cuando el proceso de verificación lo demanda. La comprobación del vacío se realiza utilizando un simple esquema de *detección de ciclos* [Courco92], que usa una estrategia de BPP y que, a diferencia del análisis de los *componentes fuertemente conectados* efectuado en [Lichte85a] y [Kesten93], sólo necesita que una pequeña parte del autómata producto (la que está siendo explorada por la BPP) esté en la memoria principal en un momento dado. El resultado es que el sistema se verifica construyendo el autómata producto *on-the-fly*, durante una BPP que comprueba su vacío. Esto significa que si la propiedad no se satisface, el algoritmo lo puede detectar construyendo y visitando sólo una pequeña parte de su espacio de estados. Por consiguiente, aunque tanto la comprobación de modelos como la de satisfacibilidad para fórmulas de la LTL son, en lo que a complejidad espacial se refiere, problemas NP-completos [Sistla85], claramente el enfoque *on-the-fly* permite aumentar el tamaño del problema que puede resolverse con respecto al enfoque clásico [Vardi86].

Otro algoritmo para generar autómatas de Büchi generalizados a partir de fórmulas de la LTL [Somenz00] extiende el trabajo realizado en [Gerth95] y [Daniel99], aplicando heurísticas en tres fases: (1) antes de la traducción, mediante reglas de reescritura que simplifican la fórmula, (2) durante la traducción, mediante técnicas de optimización booleana que reducen el número de estados generados y (3) al autómata resultante, para simplificar sus transiciones y condiciones de aceptación. Independientemente y casi al mismo tiempo, Etessami y Holzmann desarrollaron una serie de optimizaciones similares [Etessa00] para un

algoritmo también basado en el de [Gerth95], centrándose en las fases (1) y (3) mencionadas anteriormente. Las optimizaciones apuntadas en ambos trabajos ([Somenz00] y [Etessa00]) tienen como objetivo su aplicación en comprobación de modelos; de hecho, un subconjunto de las mismas se han utilizado para aumentar la eficiencia de las últimas versiones de la conocida herramienta SPIN[1] [Holzma97], cuyo algoritmo de traducción de fórmulas de la LTL a autómatas de Büchi también se basa en el de [Gerth95].

En [Gastin01] se presenta un nuevo algoritmo para generar autómatas de Büchi a partir de fórmulas de la LTL, más rápido y eficiente que los hasta ahora mencionados. Este algoritmo, que se basa en la construcción clásica [Vardi96] en lugar de en la construcción *tableau* presentada en [Gerth95], primero genera un autómata (denominado *very weak alternating co-Büchi automaton*) que se transforma en un autómata de Büchi, utilizando un autómata de Büchi generalizado como paso intermedio. Por tanto, la construcción se realiza en varios pasos, aplicando una simplificación *on-the-fly* durante la construcción de cada uno de esos autómatas, con el fin de ahorrar espacio en memoria y tiempo.

El algoritmo que hemos desarrollado (explicado detalladamente en el Capítulo 5, publicado en [Hornos02] y del que puede encontrarse una versión anterior en [Hornos01a]) construye un autómata finito equivalente a una especificación del sistema, descrita mediante una fórmula FIL. Aunque tiene sus raíces en el algoritmo presentado en [Gerth95], compartiendo también el mismo objetivo (o sea, la verificación de sistemas mediante *comprobación de modelos on-the-fly*), su ventaja con respecto a éste último consiste en que el formalismo de especificación utilizado dispone de una representación gráfica (GIL) que es mucho más intuitiva, comprensible y natural que la representación textual de la LTL. Para FIL/GIL, se estableció un procedimiento de decisión elemental en [Ramakr96a], pero usando el enfoque basado en teoría de autómatas, que no puede usarse en comprobación de modelos *on-the-fly*, debido a que el autómata se construye de una manera global.

En un importante artículo sobre el método *tableau* [Wolper85], su autor dijo que este método no parecía ser aplicable a las lógicas de intervalos. Si bien es verdad que en [Plaist83] y en [Aaby88] se daban procedimientos de decisión parecidos a los basados en *tableau* para la IL (*Interval Logic*) [Schwar83] (que, como se ha dicho, es la lógica en la que se inspira FIL), en ambos casos se utiliza

[1] http://netlib.bell-labs.com/netlib/spin/whatispin.html

una traducción a un lenguaje intermedio que tiene un problema de satisfacibilidad no elemental. Así, sus entradas en el *tableau* no son fórmulas de la propia lógica, por lo que se considera que no siguen el método *tableau* clásico. Sin embargo, nosotros hemos aplicado con éxito este método a FIL, de modo que todas las fórmulas manejadas por nuestro algoritmo pertenecen a dicha lógica. Además, nuestro algoritmo ha sido diseñado para operar *on-the-fly*, por lo que puede ser integrado en una herramienta de comprobación de modelos *on-the-fly*. Que sepamos, ésta es la primera vez que se ha desarrollado un algoritmo de este tipo para una lógica de intervalos.

En [Bowman98] se aplica el método *tableau* a la ITL (*Interval Temporal Logic*) extendida con el operador *proyección*. Este operador permite derivar construcciones de iteración parecidas a las de los lenguajes de programación imperativos. El algoritmo utilizado primero construye el grafo, aplicando una serie de reglas *tableau* para descomponer las fórmulas en otras más simples. La última fase, a la que sus autores denominan *reducción del grafo*, consiste en eliminar los nodos que no son satisfacibles. En este caso, el objetivo es comprobar la satisfacibilidad de fórmulas de esta lógica. A diferencia de nuestro algoritmo, éste sólo trabaja con secuencias *finitas* de estados, por lo que no se necesita imponer condiciones de aceptación. Además, no está pensado para operar *on-the-fly*.

La lógica que hemos utilizado en esta tesis (FIL/GIL) ha sido empleada también para generar *oráculos de prueba* (*test oracles*) basados en especificaciones [Dillon94b] [O'Mall96], donde se construyen autómatas finitos deterministas para automáticamente verificar que las *ejecuciones de prueba* se ajustan a (satisfacen) las especificaciones. Mientras que esta técnica comprueba si *una* determinada ejecución (finita) de prueba viola la correspondiente especificación, la comprobación de modelos determina si *alguna* ejecución infinita del sistema (de entre todas las posibles) incumple dicha especificación. Por tanto, la primera es una técnica más simple que la segunda, que es a la que tratamos de aplicar nuestro algoritmo. Además, la sintaxis de FIL/GIL utilizada en las referencias citadas es muy restrictiva, dado que en ellas la fórmula objetivo de cada búsqueda tiene que ser puramente proposicional y no está permitido anidar intervalos.

7.1.3. Formalismos con representación visual

Los primeros métodos formales, especialmente los enfoques para especificación y verificación de sistemas software, se basaban en formalismos matemáticos rigurosos, tales como la lógica temporal, con una representación exclusivamente textual, que a menudo llevan a especificaciones que los diseñadores software encuentran difíciles de entender y nada intuitivas. Esta tendencia está cambiando en los últimos años, ya que los investigadores y profesionales implicados en el desarrollo de software han empezado a darse cuenta de que la utilización de formalismos con representación visual en las distintas etapas del ciclo de vida del software: desde la especificación hasta la implementación y realización de pruebas, pasando por el diseño y verificación, puede aumentar la efectividad de todo el proceso de desarrollo e incluso servir de documentación para el mismo. Así, por ejemplo, observando una visualización del código de un programa, se puede obtener cierta información que no se podría conseguir observando simplemente las líneas textuales del código.

Los lenguajes visuales normalmente establecen una correspondencia entre una sintaxis textual y los objetos o representaciones gráficas que muestran de un modo más intuitivo y comprensible el correspondiente razonamiento. Dado que los ordenadores aún procesan mejor la información textual que las representaciones gráficas, por regla general, las herramientas basadas en formalismos visuales necesitan realizar una traducción entre las representaciones gráficas y textual, que debe estar formalmente definida. El almacenamiento se mantiene en un fichero de texto, que se compila para generar y visualizar la correspondiente representación gráfica. Cuando el usuario hace alguna modificación, actuando sobre ésta última, la herramienta debe efectuar el correspondiente cambio en el fichero de texto.

Los desarrolladores de software a menudo emplean notaciones gráficas, tales como por ejemplo: diagramas de transición de estados, de secuencia de mensajes, de temporización, de flujo de datos, etc., para ayudarse en sus razonamientos y diseños. En este apartado se van a referenciar algunos de los más representativos e importantes.

MSC (*Message Sequence Chart*) es una notación gráfica bastante extendida para la descripción de protocolos de comunicación entre distintos procesos, que cuenta con una representación textual estándar [ITU99]. Permite representar los mensajes enviados y recibidos por cada proceso, mostrando además el orden en que

deben ocurrir. Se pueden utilizar para describir la estructura de comunicación de una ejecución (típica o excepcional) del sistema, o de un contraejemplo encontrado durante la comprobación de modelos o la realización de pruebas. Existen herramientas, tales como MSC [Alur96], que dan soporte a este formalismo, permitiendo trabajar tanto con sus representaciones gráficas como con sus descripciones textuales estándar.

Una forma efectiva de *simular* las ejecuciones de un sistema consiste en visualizar un autómata o diagrama de flujos que representa a un programa y *animar* el progreso de un nodo (estado) a otro, resaltando el nodo actual con un color o tono diferente, por ejemplo. Así, tras ejecutar una transición, el nodo destino se resalta, mientras que el nodo origen vuelve a mostrarse con su apariencia normal. Para muchos sistemas no es factible la visualización de su espacio de estados completo (su número de estados es muy grande o infinito), aunque siempre se puede visualizar sólo una parte del mismo (la que contenga al nodo actual). Un enfoque más práctico consiste en no visualizar todos los estados del grafo de un sistema, sino sólo los que representan a sus *puntos de control* (cada uno de ellos se corresponden con un valor del contador de programa). Cada proceso de un sistema concurrente se puede visualizar en una ventana diferente. Así, para cada proceso activo (ventana) habrá un nodo resaltado, de modo que es la combinación de todos ellos la que determina el estado actual del sistema. Cada transición es ejecutada por uno o más procesos, realizándose la correspondiente animación en las distintas ventanas.

Para realizar lo que se acaba de comentar, primero hay que construir una representación visual del sistema. Esto puede hacerse de diferentes formas. Una de ellas consiste en realizar algo parecido a lo que hace la herramienta PET [Gunter99] [Gunter00], que compila el código del sistema (escrito en *Pascal Concurrente*) y genera como resultado un grafo para cada proceso, realzando la comunicación entre los mismos. Los grafos se visualizan utilizando el programa DOT [Gansne93]. Otras herramientas permiten al usuario construir y modificar el grafo mientras modela o diseña el sistema [Selic94], de modo que los nodos y arcos del grafo pueden contener información adicional, incluyendo código real para cambiar el valor de las variables, enviar o recibir mensajes, etc. En otros casos, el diseño del sistema empieza utilizando herramientas de visualización, algunas de las cuales genera automáticamente código ejecutable a partir del diseño realizado.

Aunque dicho código no debe tomarse como implementación final, constituye una buena base para modificar, en lugar de empezar a programar desde cero.

La utilidad de los grafos de estados (simples) mencionados hasta ahora es bastante limitada a la hora de representar sistema software reales y su estructura compleja. Frente a ellos, los *grafos de estados jerárquicos* capturan mejor la estructura jerárquica y concurrente de los sistemas, reduciendo significativamente el número de nodos a representar. Dentro de éstos, la notación gráfica más popular es *statecharts* [Harel87], que permite agrupar varios estados (denominados *subestados*) en un *superestado*, de modo que las transiciones pueden tener como origen o destino a cualquier (sub/super)estado. Este formalismo, que está especialmente pensado para especificar sistemas concurrentes y reactivos complejos [Harel98], es la base de STATEMATE [Harel90], un sofisticado entorno para el desarrollo de este tipo de sistemas. Este formalismo también ha sido adoptado para describir el comportamiento de un sistema en metodologías de diseño orientadas a objetos más recientes, como OMT (*Object Modeling Technique*) [Rumbau96], que a su vez se ha integrado posteriormente en UML [Fowler97]. No cabe duda de que en la actualidad UML (*Unified Modeling Language*) [Jacobs99] [Rumbau99] [Booch99] constituye la metodología de diseño visual más popular. En ella se utilizan también diagramas muy parecidos a los MSCs mencionados anteriormente. Otras técnicas visuales más especializadas pueden encontrarse en [Marca88] [Yourdo89] [Buhr90].

Las *redes de Petri* parecen ser las modelos gráficos para sistemas concurrentes más antiguos [Petri65]. Este formalismo cuenta con una representación visual atractiva, por lo que se ha utilizado bastante como técnica para la descripción y el análisis de sistemas concurrentes [Murata84], existiendo muchas herramientas que le dan soporte. Hay diferentes versiones de redes de Petri, desde las más *elementales* [Thiaga86], que no pueden tener más de un *símbolo* (*token*) en cada *lugar* (*place*), hasta las *coloreadas* [Jensen97], en las que se distinguen símbolos con distintos colores. En [Murata89] se ofrece un amplio estudio sobre las redes de Petri. Este formalismo también ha servido para inspirar una notación gráfica estandarizada, denominada *Grafcet* [David92].

Algunos lenguajes textuales de especificación han sido extendidos con representaciones gráficas; entre ellos LOTOS, para el que se han desarrollado dos notaciones gráficas diferentes [Bologn89] [Cheung90]. Éstas se han utilizado como

fundamento para implementar herramientas que permiten la creación de especificaciones visuales [New90].

IDCCS [Giacal88], una versión gráfica del conocido CCS (comentado al final del apartado 7.1.1), ha servido de base para la construcción de un entorno de especificación y diseño para sistemas concurrentes. Los ∀-autómatas [Manna87] constituyen otro formalismo gráfico para especificar las propiedades temporales de un sistema. Estos formalismos, al igual que la mayoría de los hasta ahora presentados en este apartado, se orientan hacia la representación de estados y transiciones entre ellos.

En lo que respecta a la fase de implementación, la *programación visual* [Shu88], que permite crear un programa mediante gráficos bidimensionales, ha recibido también una considerable atención. Este tipo de programación se basa en la premisa de que las figuras que utilizan son más expresivas y fáciles de entender que las líneas de código textual que emplean la mayoría de los lenguajes de programación. Además, los sistemas de programación que le dan soporte tienen una interfaz de usuario bastante intuitiva, que ayuda a los principiantes y que, a diferencia del uso intensivo del teclado requerido por la programación tradicional, potencia la utilización de dispositivos apuntadores (tipo ratón) y de pantallas gráficas de alta resolución. Se han propuesto y desarrollado varios lenguajes y entornos de programación visual, entre ellos: [Pong83], [Gliner84] y [Myers86], que difieren en su ámbito de aplicación, naturaleza y objetivos.

También se han desarrollado herramientas que permiten la *visualización de programas* [Brown85], donde, a pesar de que el programa se redacta en la manera convencional (en modo texto), se ilustran gráficamente algunos aspectos del mismo o de su ejecución. Otro enfoque interesante consiste en colorear diferentes fragmentos del texto de un programa [Ball96] para resaltar, por ejemplo: las partes que se han cambiado más recientemente, las ejecutadas por un mayor (o menor) número de pruebas, en las que se detectaron un mayor número de errores, etc. Esto puede ser de gran utilidad, ya que se ha demostrado experimentalmente [Myers79] que en la parte de código donde se han encontrado más errores es el lugar donde con gran probabilidad se pueden encontrar nuevos errores.

Finalmente, y dado que en esta tesis se propone GIL (*Graphical Interval Logic*) como lenguaje de especificación visual a utilizar, vamos a comentar y a comparar

(con GIL) los formalismos y entornos más cercanos a él. En [Yau88] se sugirió el uso de lenguajes visuales para describir las especificaciones software, como medio para mejorar su compresión, argumentando que las representaciones gráficas de los lenguajes de especificación deben tener una semántica formal perfectamente definida que soporte el razonamiento formal acerca de la corrección y consistencia de las especificaciones realizadas con esos lenguajes visuales. GIL cumple esto a la perfección, ya que todas sus construcciones se corresponden con fórmulas FIL (ver sección 2.4: FIL como base formal de GIL). Además, se ha desarrollo un entorno gráfico para GIL [Kutty94a] [Kutty93b], que permite crear directamente especificaciones visuales y comprobar su validez (mediante un demostrador de teoremas).

Para la IL (*Interval Logic*) [Schwar83] se propuso una representación gráfica en [Mellia88], que mostraba de una forma intuitiva la secuencia de eventos de una ejecución y permitía la construcción de intervalos delimitados por dos de esos eventos. A partir de dicha representación se ha derivado la representación gráfica de GIL [Dillon94a] (que hemos mostrado en el Capítulo 2).

Los *diagramas de temporización* (*Timing Diagrams*) [Schlör93] contituyen un lenguaje gráfico para expresar relaciones de precedencia y causalidad entre eventos de una ejecución. Su semántica se define mediante traducción a un subconjunto de la lógica temporal. Al igual que en el entorno para GIL, estos diagramas se crean usando un editor gráfico, pudiéndose además comprobar su validez. Este formalismo esté especialmente pensado para especificar sistemas asíncronos distribuidos, tales como sistemas hardware, mientras que GIL está más orientado a mostrar la evolución temporal de las propiedades de los sistemas software.

Otros entornos visuales anteriores al mencionado para GIL son, por ejemplo: TECTON [Kapur92], un sistema de verificación formal que emplea tablas, gráficos e hipertexto, y FORMED [Henry90], un sistema que visualiza fórmulas de la lógica de primer orden en un formato bidimensional. En [Théry92] se estudian interfaces gráficas para sistemas de demostración de teoremas tradicionales. En ellos, al igual que en el sistema de razonamiento temporal TIMELOGIG [Koomen89] y a diferencia del entorno para GIL, se utiliza una lógica textual, sirviendo la visualización sólo para facilitar la construcción de la demostración.

En [Moser96] y [Moser97] se describe un entorno para RTGIL (*Real-Time Graphical Interval Logic*), totalmente interactivo y gráfico, para especificar y

verificar (demostrar formalmente) propiedades de los sistemas de tiempo real concurrentes. Aunque, a excepción del operador que restringe la duración de un intervalo, la sintaxis gráfica de RTGIL coincide con la de GIL, las semánticas de los modelos teóricos subyacentes son diferentes. Así, las fórmulas en RTGIL se interpretan sobre una línea de tiempo continuo (los reales no negativos), mientras que las fórmulas en GIL se interpretan sobre una línea de tiempo discreto (los enteros no negativos), lo que hace que ésta no tenga capacidad para expresar y razonar con propiedades de tiempo real, mientras que la primera sí.

7.2. Conclusiones y principales aportaciones del trabajo desarrollado

De todo el estudio teórico que se ha llevado a cabo en este trabajo de investigación se puede concluir que, hoy por hoy, todas las técnicas de verificación se basan en la construcción de autómatas, dado que constituyen las estructuras sobre las que se comprueba si se cumple o no la correspondiente propiedad.

En este sentido, la principal aportación de esta tesis es el desarrollo de un algoritmo para la traducción automática de fórmulas FIL a autómatas de Büchi, cuya descripción a nivel conceptual y teórico (incluida una demostración formal de su corrección) se presenta en el Capítulo 5 y del que pueden encontrarse detalles de su implementación en el Capítulo 6 y en el Apéndice C. Éste es el primer algoritmo conocido para una lógica de intervalos basado en la extensión del método *tableau* y capaz de operar *on-the-fly*. Creemos que esto es de gran relevancia, ya que hasta hace muy poco tiempo se consideraba que el método *tableau* no era aplicable a las lógicas de intervalos (ni se conocía la forma de extender este método, tan habitual entre las lógicas temporales tradicionales, para aplicarlo a las lógicas de intervalos, ni parecía previsible que se pudiera realizar), por lo que adaptarlo con éxito para FIL y además para operar *on-the-fly* suponía un reto aún mayor y más difícil de conseguir.

En efecto, FIL es la primera lógica de intervalos en la que el método *tableau* se ha aplicado con éxito. La dificultad de aplicar este método a las lógicas de intervalos proviene de la necesidad de codificar sucintamente la noción de ámbito temporal limitado (donde una fórmula debe ser evaluada) en las ejecuciones del autómata. Así, la principal novedad de FIL con respecto a las lógicas temporales

tradicionales, la noción de un contexto temporal acotado (establecido por un determinado intervalo), también resulta ser una fuente importante de dificultad al extender los métodos usuales aplicables a éstas para la construcción del correspondiente autómata de propiedad. Por tanto, cuando una fórmula temporal está anidada a algún intervalo, el alcance de dicha fórmula no es el contexto global, sino que debe restringirse al contexto temporal establecido por ese intervalo. Las reglas de expansión que hemos desarrollado (ver apartado 5.1.4), y que constituyen el núcleo central de nuestro algoritmo, tienen en cuenta este hecho, resultando finalmente que las fórmulas FIL que etiquetan los nodos (estados) del autómata que se construye, codifican correctamente esa noción de alcance temporal acotado, tal y como se ha explicado (de manera intuitiva) en el apartado 4.2.1. Descripción general de la estrategia de construcción.

En el Capítulo 3 se han estudiado y comparado entre sí las distintas técnicas automatizadas que se utilizan actualmente para comprobar la corrección del diseño y/o implementación de un sistema, mostrando además las principales fortalezas y debilidades de cada una de esas técnicas. Lo ideal (o sea, el objetivo al que se debe tender) es que se pudiera verificar un sistema automáticamente, es decir, sin intervención del usuario o, en su defecto, con el mínimo esfuerzo por parte de éste. Teniendo en cuenta esto, se debe concluir que la técnica más prometedora en este sentido es la comprobación de modelos, denominada en muchas ocasiones por esta razón *verificación automática*. El trabajo de investigación que hemos llevado a cabo se aplica precisamente en esta técnica de verificación.

Aunque nuestro algoritmo de traducción de fórmulas FIL a autómatas de Büchi está especialmente pensado para ser utilizado en la verificación automática de sistemas concurrentes, utilizando el método de la comprobación de modelos *on-the-fly* (ver apartado 4.3.3), también puede emplearse para determinar la *satisfacibilidad* o la *validez* de una determinada especificación FIL, tal y como se ha comentado en la sección 4.4. Comprobación de especificaciones FIL.

Las dos razones más importantes por las que se ha adoptado la lógica de intervalos FIL/GIL como el marco lógico (ver Capítulo 2) sobre el que desarrollar nuestro trabajo han sido:

(1) Su natural e intuitiva representación gráfica. Así, la representación visual de GIL, en la que explícitamente se representa una traza o secuencia de estados mediante una línea (que modela el paso del tiempo) y los intervalos (que

establecen contextos temporales específicos dentro de la traza), aumenta considerablemente la comprensión de sus fórmulas, debido a que dicha representación se ajusta bastante bien a la forma en que los seres humanos razonan. De hecho, las fórmulas GIL se parecen bastante a los diagramas temporales que muchos diseñadores de sistemas utilizan para describir informalmente y razonar acerca del comportamiento temporal de sus diseños. Esto ciertamente constituye una gran ventaja frente a la representación textual y nada intuitiva de las lógicas temporales tradicionales.

(2) Su capacidad para expresar sucintamente contextos temporales en los que deben satisfacerse determinadas propiedades. Las lógicas temporales tradicionales, al no disponer ni de noción semántica de intervalo ni de ningún elemento sintáctico que exprese tal noción, deben recurrir al anidamiento de operadores y expresiones, complicando excesivamente las fórmulas resultantes, con lo que la comprensión de las mismas disminuye considerablemente.

Las ventajas de la lógica empleada se han puesto particularmente de manifiesto con respecto a la LTL en el apartado 2.3.3. Comparación con la LTL, debido a que, entre las lógicas temporales tradicionales, sin lugar a dudas, ésta es la más utilizada actualmente para la especificación y verificación de sistemas.

En la sección 6.3 se han presentado una serie de resultados experimentales obtenidos con nuestra herramienta, FBT, para algunas especificaciones formuladas en FIL (tanto en su sintaxis restringida como en la extendida con operadores temporales de la LTL). Además, se ha puesto de manifiesto que los autómatas de Büchi que ésta genera son de complejidad similar (e incluso ligeramente inferior, en la mayoría de los casos) a los construidos por otros traductores, como por ejemplo: LBT, con el que se ha establecido una detenida comparación en la sección 6.4, no sólo a nivel de resultados, sino también en cuanto a sus respectivos diseños e implementaciones.

Como consecuencia de que en los últimos años se ha establecido un marco formal sólido para la construcción de herramientas de verificación efectivas y de las continuas y notables mejoras en la velocidad del hardware y en la capacidad de las memorias, parece previsible y probable que en el futuro las herramientas de verificación formal bien diseñadas se ganen su puesto entre las herramientas centrales de cada diseñador de sistemas (hardware o software). Para incrementar el impacto que dichas herramientas puedan tener en la práctica, éstas deben

diseñarse teniendo en mente al usuario final, es decir, que deben poder ser utilizadas por personas no expertas, y no sólo por expertos en formalismos matemático-lógicos y en verificación formal. Todo esto requiere altas exigencias en el diseño de interfaces de usuario, en la retroalimentación sobre los resultados de la verificación al usuario y en la elección de lenguajes y notaciones para expresar los requisitos de corrección. Es en este último punto donde tiene aplicación esta tesis, ya que uno de sus objetivos consiste en la integración de un lenguaje gráfico y bastante intuitivo, pero al mismo tiempo riguroso y formal, dentro de una herramienta de verificación automática. En definitiva, se puede decir que esta tesis supone un paso en la dirección correcta hacia la implantación de herramientas de verificación automática que, sin perder el rigor de las herramientas actuales, sean además amigables, para lo que es fundamental que sus formalismos de especificación utilicen representaciones intuitivas y se ajusten a la forma en que sus usuarios razonan normalmente. De este modo, se conseguirá extender la aceptación y utilización de los métodos formales entre los profesionales dedicados al diseño y desarrollo de sistemas software y hardware.

7.3. Orientación futura del trabajo

El trabajo de investigación llevado a cabo para realizar esta tesis se puede continuar en varias líneas distintas, entre ellas las siguientes:

- Extender la lógica que admite nuestra herramienta con dos nuevos operadores: *búsqueda fuerte* e *intervalo fuerte*, que se definen formalmente en el Apéndice B. También en dicho apéndice se describen las modificaciones que sería necesario realizar para implementar la mencionada extensión. La inclusión de estos nuevos operadores haría posible la fácil obtención de la fórmula dual de una fórmula de intervalo, lo que permitiría la conversión automática de todas las fórmulas aceptadas por FBT, incluidas las de intervalo, a su forma normal de negación (donde las negaciones sólo se aplican a las variables proposicionales). Además, cualquier fórmula generada durante el proceso de expansión de la fórmula introducida también debería estar en esta forma normal.

- Construir un editor gráfico para GIL, similar al implementado en [Kutty93b] [Kutty94a], de modo que el diseñador del sistema pueda realizar la especificación de requisitos o propiedades temporales directamente en GIL.

Recuérdese que, en nuestra implementación actual, las fórmulas que especifican dichas propiedades se suministran en la sintaxis textual de FIL, extendida con algunos operadores temporales de la LTL (ver apartado 6.2.1). Aunque, para verificar que el sistema cumple la correspondiente especificación, se pretende aplicar nuestro algoritmo (presentado en el Capítulo 5) tal y como está, es decir, para que opere internamente sobre fórmulas expresadas en la sintaxis restringida de FIL, la idea al añadir el mencionado editor gráfico es que dicha herramienta traduzca automáticamente las especificaciones gráficas que el usuario realice en GIL a sus equivalentes fórmulas FIL, sobre las que debe operar nuestro algoritmo.

- Integrar nuestro traductor en una herramienta de comprobación de modelos *on-the-fly*. Para ello, ya tenemos un principio de acuerdo de colaboración con unos investigadores finlandeses (miembros de la *Helsinki University of Technology*) para incorporar FBT a la herramienta MARIA[2] [Mäkelä02], que es capaz de efectuar simulación, análisis exhaustivo de alcanzabilidad y comprobación de modelos *on-the-fly* con restricciones de equidad (*fairness*) [Latval00] [Latval01]. La idea (ver Figura 7.1) es o bien añadir FBT como un traductor adicional y alternativo a LBT, con lo que se podrían establecer comparaciones entre los autómatas de propiedad que ambos producen, su eficiencia en cuanto a tiempo de respuesta, etc., o bien sustituir completamente a LBT, con lo que FBT sería el único traductor disponible en MARIA. Ésta ha sido la principal razón que nos ha llevado a implementar FBT a partir de LBT, respetando su interfaz, tanto de entrada como de salida, y aceptando todos los operadores temporales (excepto el operador *next*) que LBT admite, pero como abreviaciones de las correspondientes fórmulas de intervalo FIL. Todo ello con el fin de que FBT fuera totalmente "compatible" con LBT.

- Aplicar nuestra herramienta, una vez integrada en un comprobador de modelos *on-the-fly*, para especificar y verificar automáticamente no sólo sistemas concurrentes relativamente simples (con propósitos educativos o académicos), sino también sistemas concurrentes del mundo real, o sea, sistemas de tamaño industrial. Éste es el fin último para el que se ha desarrollado el trabajo descrito en esta memoria de tesis.

[2] http://www.tcs.hut.fi/maria/

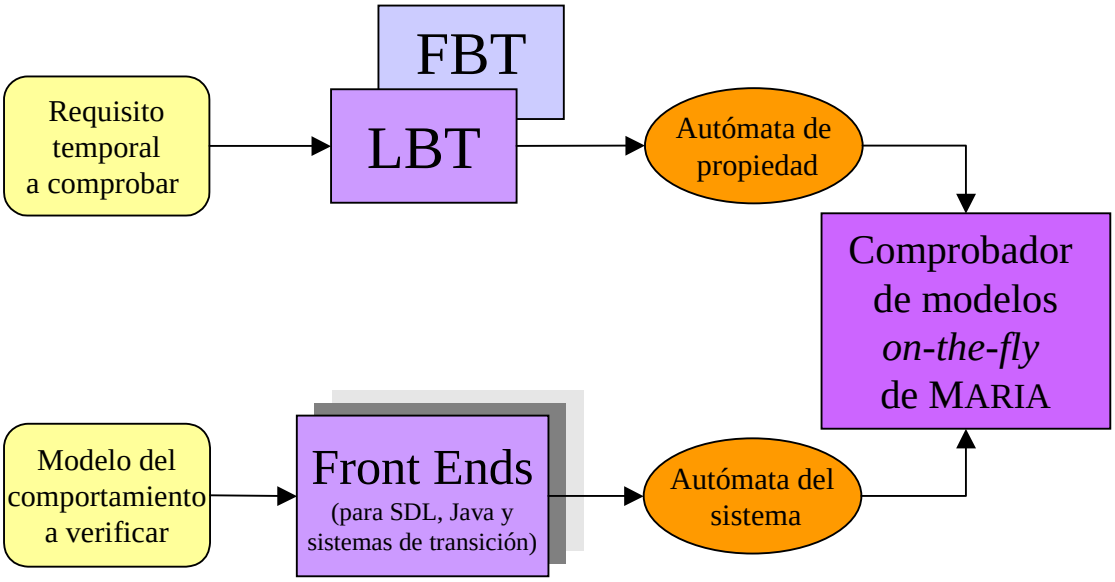


Figura 7.1. Entradas al comprobador de modelos *on-the-fly* de MARIA

- Extender el algoritmo que hemos diseñado e implementado para que pueda ser aplicado a sistemas de tiempo real. Para ello, nos debemos basar en las lógicas que, tanto en formato textual como gráfico, constituyen las extensiones para tiempo real de los lenguajes de descripción formal que se han empleado en esta tesis, denominadas respectivamente RTFIL (*Real-Time Future Interval Logic*) [Ramakr93b] [Ramakr96b] y RTGIL (*Real-Time Graphical Interval Logic*) [Moser96] [Moser97]. Sin duda, éste es el punto que supone el reto más importante de todos los que se han planteado dentro de este apartado, ya que se deberá obtener un algoritmo basado en el método *tableau* y capaz de operar *on-the-fly*, que además codifique de forma sucinta el paso del tiempo, con el fin de llevar la cuenta del mismo.

Índice de Referencias Empleadas

[Aaby88], *192*
[Allen83], *186*, *187*
[Alur89], *186*
[Alur96], *195*
[Baeten90], *189*
[Ball96], *197*
[Barrin86], *186*
[Ben-Ar83], *186*
[Bologn89], *196*
[Booch99], *196*
[Bowman98], *193*
[Brown85], *197*
[Buhr90], *196*
[Chaoch91], *187*
[Cheung90], *196*
[Clarke86], *186*
[Courco92], *191*
[D'Agos99], *190*
[Daniel99], *191*
[David92], *196*
[Dillon94a], *198*
[Dillon94b], *193*
[Emerso90], *185*, *186*
[Etessa00], *191*, *192*
[Fowler97], *196*
[Gansne93], *195*
[Gastin01], *192*
[Gerth95], *191*, *192*
[Giacal88], *197*
[Gliner84], *197*
[Goswam92], *188*
[Gunter00], *195*
[Gunter99], *195*
[Halper83], *186*, *187*
[Halper91], *186*, *187*
[Harel82], *186*
[Harel84], *186*
[Harel87], *196*
[Harel90], *196*
[Harel98], *196*
[Henry90], *198*
[Hoare78], *189*
[Holzma97], *192*
[Hornos01a], *192*
[Hornos02], *192*
[Hornos97], *188*
[Hornos99], *188*
[ITU99], *194*
[Jacobs99], *196*
[Jensen97], *196*
[Kapur92], *198*
[Kesten93], *190*, *191*
[Koomen89], *198*
[Kozen90], *186*
[Kutty93b], *198*, *202*
[Kutty94a], *198*, *202*
[Lampor94], *186*
[Latval00], *203*
[Latval01], *203*
[Lichte85a], *190*, *191*
[Lichte85b], *186*
[Mäkelä02], *203*
[Manna82], *185*
[Manna87], *197*
[Manna92], *185*
[Manna95], *185*
[Marca88], *196*
[Mellia88], *198*
[Milner80], *189*
[Moser96], *198*, *204*
[Moser97], *198*, *204*
[Murata84], *196*
[Murata89], *196*
[Myers79], *197*
[Myers86], *197*
[New90], *197*
[O'Mall96], *193*
[Ostrof89], *186*
[Petri65], *196*
[Pinter84], *186*
[Plaist83], *187*, *192*
[Pnueli77], *185*
[Pnueli86a], *185*
[Pong83], *197*
[Pratt79], *186*
[Ramakr92], *188*
[Ramakr93a], *188*
[Ramakr93b], *188*, *204*
[Ramakr96a], *188*, *190*, *192*
[Ramakr96b], *188*, *204*
[Rosner86], *187*
[Rumbau96], *196*
[Rumbau99], *196*
[Schlör93], *198*
[Schwar83] , *186*, *187*, *192*, *198*
[Selic94], *195*
[Shu88], *197*
[Sistla85], *191*
[Somenz00], *191*, *192*
[Stirli88], *189*
[Théry92], *198*
[Thiaga86], *196*
[Vardi86], *189*, *191*
[Vardi96], *192*
[Venema94], *187*
[Wolper83], *189*
[Wolper85], *190*, *192*
[Yau88], *198*
[Yourdo89], *196*

Apéndice A

Glosario de Términos y Acrónimos

- **Autómata de Büchi**: ω-autómata o autómata finito sobre palabras infinitas, por lo que tiene un criterio de aceptación especial. Así, cuenta con un conjunto de estados de aceptación (en lugar de un conjunto de estados finales), de modo que para que cualquiera de sus ejecuciones sea aceptada debe visitar al menos uno de esos estados infinitamente a menudo. Véase Definiciones 4.1, 5.23 y 5.26 para más detalles.

- **Autómata de Büchi generalizado**: Autómata de Büchi que tiene múltiples conjuntos de estados de aceptación en lugar de uno solo. Véase Definición 5.29 para más detalles.

- **Autómata de propiedad**: Autómata que es semánticamente equivalente a una fórmula lógica o especificación, es decir, que acepta exactamente las mismas secuencias infinitas que satisfacen dicha especificación.

- **Autómata del sistema**: Autómata que modela el comportamiento del sistema que se desea verificar. En la bibliografía sobre comprobación de modelos, a este término se le denomina a veces *modelo*, ya que se trata de una versión simplificada del sistema original, obtenida a partir del mismo mediante un proceso de abstracción. Véase la entrada *modelo* en este mismo apéndice para comprobar que utilizamos este término con otro significado.

- **BPP** (Búsqueda Primero en Profundidad): Algoritmo de búsqueda en el espacio de estados de un autómata, que utiliza una pila como almacén temporal de los nodos aún no visitados, por lo que el último nodo generado (introducido en la pila) será el primero en ser explorado.

- **Comprobación de modelos** (*model checking*): Técnica de verificación *automática* de sistemas, consistente en comprobar que una determinada propiedad (expresada mediante una fórmula de una lógica temporal) se satisface en cualquiera de las ejecuciones del sistema de transición que representa al sistema a verificar.

- **Comprobación de modelos *on-the-fly***: Denominada así porque los estados de los autómatas de propiedad y del sistema (o sólo de éste último) sólo se generan cuando son necesarios, mientras se está comprobando el no vacío de la intersección de los lenguajes de ambos autómatas. Así, sólo se construyen los estados alcanzables del autómata producto (de los dos mencionados) hasta que se determina su no vacío, pudiéndose por tanto detectar que el sistema viola la especificación construyendo y visitando sólo una pequeña parte de su espacio de estados, con lo que en muchos casos se evita la construcción completa del mismo.

- **Constante lógica**: Fórmula que representa a uno de los dos posibles valores lógicos: **T** (*true* o valor de verdad) y **F** (*false* o valor de falsedad), por lo que su interpretación es siempre la misma, no dependiendo del modelo sobre el que dicha fórmula se evalúe.

- **Demostración de teoremas** (*theorem proving*): Técnica de verificación que trata de aplicar el razonamiento deductivo para demostrar la corrección de un sistema, motivo por el que también se denominada *verificación deductiva*. Así, a partir de una serie de axiomas se trata de obtener la fórmula (teorema) que desea demostrar, aplicando las reglas del sistema de demostración utilizado. Para la verificación de programas se necesita traducir el programa a fórmulas, axiomatizar los objetos que el programa manipula y finalmente desarrollar la demostración.

- **Estructura absoluta de una fórmula de intervalo**: Estructura resultante de no tener en cuenta ninguno de los operadores de negación que afectan directamente a (colocados inmediatamente delante de) los distintos operadores de intervalos que componen dicha fórmula. Se denota como $|\eta|$, donde η es una fórmula de intervalo.

- **FBT** (*FIL to Büchi automaton Translator*): Traductor que a partir de una fórmula FIL genera un autómata de Büchi generalizado.
- **FE** (Fórmula de eventualidad): Fórmula de intervalo del tipo $\neg[\theta_1 \mid \rightarrow)\mathbf{F}$, donde θ_1 es un patrón de búsqueda no trivial, denominada así porque expresa que dicho patrón de búsqueda se debe cumplir obligatoriamente (en el futuro reflexivo).
- **FE anidada a una SMIA**: FIA cuya estructura absoluta viene dada por $\mathcal{I}[\theta_1 \mid \rightarrow)\mathbf{F}$, donde θ_1 es un patrón de búsqueda no trivial, y donde el número de operadores de negación que afectan directamente a (colocados inmediatamente delante de) los distintos intervalos que componen dicha fórmula es impar.
- **FFB** (Fórmula Fallo de Búsqueda) **anidada a una SMIA**: FIA cuya estructura absoluta viene dada por $\mathcal{I}[\theta_1 \mid \rightarrow)\mathbf{F}$, donde θ_1 es un patrón de búsqueda no trivial, y donde el número de operadores de negación que afectan directamente a (colocados inmediatamente delante de) los distintos intervalos que componen dicha fórmula es par.
- **FIA** (Fórmula de Intervalo Actual): Fórmula de intervalo cuyo intervalo más externo es una MIA.
- **FIL** (*Future Interval Logic*): Lógica temporal proposicional de tiempo lineal y discreto, cuya construcción clave es el intervalo, lo que permite efectuar un razonamiento lógico a nivel de intervalos de tiempo en lugar de instantes. Se denomina así porque las búsquedas utilizadas para la construcción de los intervalos siempre se realizan en el futuro, nunca en el pasado. Es la base formal de GIL.
- **FINA** (Fórmula de Intervalo No Actual): Fórmula de intervalo cuyo intervalo más externo no es una MIA.
- **FINA anidada a una SMIA**: FIA que tiene una FINA anidada a la SMIA que constituye su prefijo.
- **Forma normal de negación**: Sólo permite que el operador de negación pueda ser aplicado a las proposiciones atómicas de una fórmula.
- **Forma normal de una FE anidada a una SMIA**: Denotada como $\langle\eta\rangle$, donde η es una FE anidada a una SMIA, es la fórmula FIL que coincide con la negación de la estructura absoluta de η, es decir, $\langle\eta\rangle = \neg \mid \eta \mid$.

- **Forma normal de una FFB anidada a una SMIA**: Denotada como $\langle \eta \rangle$, donde η es una FFB anidada a una SMIA, es la fórmula FIL que coincide con la estructura absoluta de η, o sea, $\langle \eta \rangle = | \eta |$.

- **Fórmula de intervalo**: Define un intervalo que establece el contexto en el que se debe satisfacer la fórmula anidada al mismo, es decir, su operador principal es el operador de intervalo.

- **Fórmula proposicional**: Fórmula que o bien pertenece a la Lógica Proposicional (o sea, es una FPP) o bien su operador principal pertenece a dicha lógica (es decir, es una FPM).

- **FPM** (Fórmula Proposicional Mixta): Fórmula cuyo operador principal pertenece a la Lógica Proposicional clásica, pudiendo ser (en nuestra implementación) uno de los siguientes: *conjunción*, *disyunción*, *implicación*, *equivalencia* o *disyunción exclusiva* (*xor*), y en la que al menos uno de sus operandos contiene alguna fórmula de intervalo.

- **FPM anidada a una SMIA**: FIA que tiene una FPM anidada a la SMIA que constituye su prefijo.

- **FPP** (Fórmula Puramente Proposicional): Fórmula FIL que no contienen ninguna modalidad de intervalo. Por tanto, puede ser una constante lógica, un literal o una combinación de dichas fórmulas mediante los siguientes operadores de la Lógica Proposicional (que son los reconocidos por nuestro algoritmo): *conjunción*, *disyunción*, *implicación*, *equivalencia* o *disyunción exclusiva* (*xor*).

- **FPP anidada a una SMIA**: FIA que tiene anidada una FPP a la SMIA que constituye su prefijo. Por tanto, esa FPP ha de cumplirse en el instante actual.

- **GIL** (*Graphical Interval Logic*): Lógica temporal proposicional de tiempo lineal y discreto, cuya construcción clave es el intervalo, lo que permite efectuar un razonamiento lógico a nivel de intervalos de tiempo en lugar de instantes. Se denomina así porque sus fórmulas se representan gráficamente. Su semántica se define en base a la representación textual de FIL, por lo que se dice que FIL constituye su base formal.

- **Intervalo más externo de una fórmula de intervalo**: Aquél que está situado más a la izquierda en su representación en FIL y más arriba en su representación en GIL.

- **LBT** (*LTL to Büchi automaton Translator*): Traductor que a partir de una fórmula LTL genera un autómata de Büchi generalizado.
- **Literal**: Proposición atómica, negada o no.
- **LTL** (*Linear Temporal Logic*): Lógica temporal proposicional de tiempo lineal y discreto, bastante extendida entre la comunidad científica para la especificación y verificación de sistemas.
- **MIA** (Modalidad de Intervalo Actual): Modalidad de intervalo cuyo primer patrón de búsqueda (el izquierdo) es el trivial, mientras que el segundo es un patrón de búsqueda no trivial. Por tanto, define un intervalo que se extiende desde el instante actual (extremo izquierdo del intervalo) hasta el instante previo al localizado por su segundo patrón de búsqueda.
- **Modelo**: ω-traza o secuencia lineal (infinita) de estados en la que se evalúa o interpreta una fórmula. Así, cada uno de sus estados asigna un valor de verdad o falsedad a cada una de las variables proposicionales que integran el conjunto de proposiciones atómicas considerado.
- **Patrón de búsqueda**: Secuencia de cero (patrón trivial) o más búsquedas que define uno de los extremos de un intervalo. Cada búsqueda localiza el primer estado (del futuro reflexivo) en el que se cumple su fórmula objetivo. Cuando un patrón de búsqueda contiene varias búsquedas, cada búsqueda subsiguiente comienza en el estado localizado por la búsqueda previa, de modo que su última búsqueda localiza el estado que corresponde al extremo del intervalo que define.
- **Prefijo de una FIA**: Secuencia formada por todas las MIAs por las que empieza dicha FIA.
- **Proposición atómica**: Predicado o propiedad elemental de un sistema que puede tomar diferentes valores de verdad en distintos instantes de tiempo o estados de una ejecución del sistema.
- **Pruebas** (*testing*): Técnica que trata de encontrar errores en un programa ejecutándolo para un muestreo de sus posibles ejecuciones (seleccionadas de acuerdo a algún criterio).
- **Regla de expansión**: Define cómo se expande o descompone un tipo de fórmula en las distintas alternativas posibles para que dicha fórmula se satisfaga, y si se trata de una fórmula temporal, separa además lo que se tiene que cumplir en el

instante actual de lo que tiene que satisfacerse desde el siguiente instante en adelante.

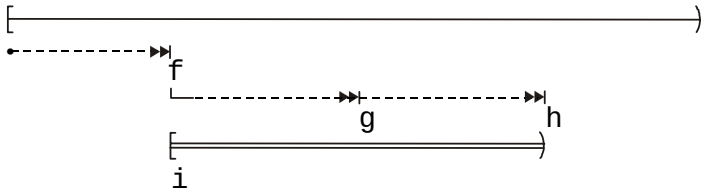
- **SMIA** (Secuencia de Modalidades de Intervalo Actuales): Secuencia de uno o más intervalos formada exclusivamente por MIAs.
- **Tartamudeo** (*Stuttering*): Repetición del último estado de una secuencia. De este modo, una subtraza o secuencia finita de estados se puede modelar como una secuencia infinita.
- **Verificación automática**: Denominada así porque sólo requiere una mínima intervención del usuario, comparada con otras técnicas de verificación. Se le conoce también con el nombre de *comprobación de modelos*.
- **Verificación deductiva**: Véase *demostración de teoremas*.

Apéndice B

Extendiendo la Lógica con Nuevos Operadores: Búsquedas e Intervalos Fuertes

En el Capítulo 2: El Marco Lógi co se han definido dos operadores específicos para las fórmulas de intervalo de la lógica que empleamos: el operador de *búsqueda* y el operador de *intervalo*. Recuérdese también que una fórmula de intervalo se satisface vacuamente si cualquiera de sus búsquedas falla o si el intervalo es vacío (esto es, su extremo derecho se encuentra antes o en el mismo instante que su extremo izquierdo). Por esta razón, se dice que ambos operadores son *débiles*, no habiéndose mencionado antes esto debido a que nosotros sólo hemos utilizamos una única versión para estos operadores. Sin embargo, los inventores de la lógica utilizada en esta tesis, definen también las versiones *fuertes* para estos operadores, extendiendo de este modo dicha lógica, con el fin de proporcionar abreviaciones útiles a la hora de especificar ciertas propiedades [Dillon94a]. Así, el operador *búsqueda fuerte*, representado en GIL mediante una doble punta de flecha, impone el requisito de que esa búsqueda no debe fallar, es decir, que debe encontrarse su fórmula objetivo (a menos que alguna búsqueda débil previa falle). De modo similar, el operador *intervalo fuerte*, dibujado con una línea doble, exige que ese intervalo no se colapse, esto es, que no sea vacío. Ambos operadores pueden combinarse para establecer requisitos más fuertes sobre una determinada fórmula de intervalo; así, por ejemplo, la siguiente fórmula de intervalo GIL exige que todas

sus búsquedas tengan éxito y que su extremo izquierdo preceda (estrictamente) a su extremo derecho.



La versión textual (esto es, en FIL) de esta fórmula es la siguiente:

$$[\!]{\Rightarrow}f \mid {\Rightarrow}f, {\Rightarrow}g, {\Rightarrow}h[\!]\, i \tag{B.1}$$

A continuación se presentan las definiciones formales de las distintas versiones posibles de intervalos fuertes que se pueden utilizar:

$$\begin{aligned}
[\!]\theta_1 \mid \theta_2[\!]\, f &= [\theta_1 \mid \theta_2)\, f \wedge ([\theta_1 \mid \rightarrow)\mathbf{F} \vee [\theta_2 \mid \rightarrow)\mathbf{F} \vee \neg[\theta_1 \mid \theta_2)\mathbf{F}) \\
&= \neg[\theta_1 \mid \theta_2)\neg f \vee [\theta_1 \mid \rightarrow)\mathbf{F} \vee [\theta_2 \mid \rightarrow)\mathbf{F}
\end{aligned}$$

$$\begin{aligned}
[\!][- \mid \theta_2[\!]\, f &= [- \mid \theta_2)\, f \wedge ([\theta_2 \mid \rightarrow)\mathbf{F} \vee \neg[- \mid \theta_2)\mathbf{F}) \\
&= \neg[- \mid \theta_2)\neg f \vee [\theta_2 \mid \rightarrow)\mathbf{F}
\end{aligned}$$

$$\begin{aligned}
[\!]\theta_1 \mid \rightarrow[\!]\, f &= [\theta_1 \mid \rightarrow)\, f \wedge ([\theta_1 \mid \rightarrow)\mathbf{F} \vee \neg[\theta_1 \mid \rightarrow)\mathbf{F}) \quad = \quad [\theta_1 \mid \rightarrow)\, f \\
&= \neg[\theta_1 \mid \rightarrow)\neg f \vee [\theta_1 \mid \rightarrow)\mathbf{F}
\end{aligned}$$

De igual modo, las siguientes expresiones muestran cómo se define formalmente el operador de búsqueda fuerte. En ellas, deben sustituirse las llaves { } por la versión correspondiente del operador de intervalo utilizado: [ ) o [\!]\!).

$$\{\theta_1, {\Rightarrow}g, \theta_2 \mid \theta_3\}\, f = \{\theta_1, {\rightarrow}g, \theta_2 \mid \theta_3\}\, f \wedge [\theta_1 \mid \rightarrow)\, \Diamond g$$

$$\{\theta_1 \mid \theta_2, {\Rightarrow}g, \theta_3\}\, f = \{\theta_1 \mid \theta_2, {\rightarrow}g, \theta_3\}\, f \wedge ([\theta_1 \mid \rightarrow)\mathbf{F} \vee [\theta_2 \mid \rightarrow)\, \Diamond g)$$

Para una completa definición del operador de búsqueda fuerte se tendrían que contemplar todos los casos posibles, añadiendo las correspondientes definiciones para los casos en que el operador ⇒ se da al principio o al final de cada patrón de búsqueda o para cuando ocurre en una modalidad de intervalo actual o en un intervalo sufijo. Además, puede haber más de un operador ⇒ dentro de un patrón

de búsqueda (pudiendo incluso ser todos ellos de este tipo). Las definiciones para todos estos casos se pueden obtener de una forma bastante sencilla y clara, modificando adecuadamente las dos definiciones expuestas, por lo que no vemos la necesidad de presentar explícitamente todas ellas.

Obsérvese que la definición de estos dos nuevos operadores no aumenta la expresividad de la lógica. Así, la siguiente equivalencia pone de manifiesto que se puede afirmar exactamente lo mismo que expresa la fórmula (B.1) anterior (fórmula de la izquierda) mediante una fórmula que sólo contenga operadores débiles (fórmula de la derecha).

$$[\!]\Rightarrow\!\!\!\!\rightarrow f \mid \Rightarrow\!\!\!\!\rightarrow f, \Rightarrow\!\!\!\!\rightarrow g, \Rightarrow\!\!\!\!\rightarrow h)\, i \quad \equiv \quad \neg[\rightarrow f \mid \rightarrow f, \rightarrow g, \rightarrow h)\, \neg i \tag{B.2}$$

Modifiquemos ahora la fórmula (B.1), cambiando el operador que busca la fórmula *g* por su versión débil y dejando el resto de la fórmula tal y como está. Con ello, se obtiene la siguiente fórmula:

$$[\!]\Rightarrow\!\!\!\!\rightarrow f \mid \Rightarrow\!\!\!\!\rightarrow f, \rightarrow g, \Rightarrow\!\!\!\!\rightarrow h)\, i \tag{B.3}$$

Esta fórmula se satisfará vacuamente sólo si falla la búsqueda para *g*; pero si ésta tiene éxito, entonces para que se satisfaga (propiamente) la fórmula (B.3) se requiere que las otras dos búsquedas localicen sus fórmulas objetivos y que el punto localizado para *f* preceda (estrictamente) al localizado para *h*.

La importancia de la introducción de estos dos nuevos operadores es que permiten obtener fácilmente la *fórmula dual de una fórmula de intervalo*, intercambiando las versiones (fuerte por débil y viceversa) de los operadores de intervalo y de búsqueda que haya en la misma. Esta dualidad sirve para poder introducir el símbolo de negación que afecte a una fórmula de intervalo dentro de la misma, intercambiando las versiones de sus intervalos y búsquedas. Así, por ejemplo, la negación de la fórmula (B.3) sería equivalente a la siguiente fórmula:

$$[\rightarrow f \mid \rightarrow f, \Rightarrow\!\!\!\!\rightarrow g, \rightarrow h)\, \neg i$$

Ahora, a la luz de lo que se acaba de exponer, se puede entender mejor la expresión (B.2) anterior.

Finalmente, decir que en la implementación de nuestro algoritmo (ver Capítulo 6 y Apéndice C) no hemos considerado los operadores definidos en este apéndice, debido a que, como se ha dicho, éstos sólo representan abreviaciones para expresar determinadas propiedades, por lo que su no inclusión no resta expresividad a la lógica. Sin embargo, se podrían contemplar sin ningún problema. Para ello, habría que añadir un nuevo atributo (variable booleana) a la clase *FilInterval*, para poder distinguir si el intervalo de la correspondiente fórmula es fuerte o débil. De modo similar, para diferenciar entre búsquedas débiles y fuertes, se tendría que asociar una variable (también booleana) a cada una de las fórmulas que componen un patrón de búsqueda. Además, se debería implementar la relación dual que se acaba de explicar, con el fin de poder empujar las negaciones dentro de las fórmulas de intervalo, consiguiendo así hallar la forma normal de negación de una fórmula de intervalo y simplificar determinadas expresiones. Por último, habría que extender las reglas de expansión para las fórmulas de intervalo (ver subapartado 5.1.4.2), añadiendo nuevas reglas para expandir las fórmulas que incluyan a los nuevos operadores, modificando las reglas actuales para que cualquier fórmula generada esté en su forma normal de negación y eliminando todas aquellas reglas en las que el operador de negación se aplique directamente sobre un determinado intervalo.

Apéndice C

Documentación del Código Fuente de FBT

Excepto esta introducción al contenido del presente apéndice, el resto del mismo se presenta en inglés, debido a que ha sido generado automáticamente a partir del código fuente de FBT, haciendo uso de la herramienta DOXYGEN[1]. Para ello, hemos tenido que emplear una sintaxis específica a la hora de comentar los distintos elementos que componen dicho código fuente, ya que, entre otras cosas, DOXYGEN utiliza esos comentarios para generar la documentación que aquí se expone.

El apéndice se organiza como sigue: En primer lugar se muestran tres índices, que indican la página donde se encuentra la descripción de los principales elementos de FBT, el primero lista sus componentes (clases y estructuras), el segundo, los distintos ficheros que lo conforman y el tercero, la jerarquía de clases existente. A continuación se incluye el apartado Graph Legend, que explica cómo interpretar los diagramas de colaboración y/o herencia que aparecen en la descripción de cada clase con el objetivo de mostrar la relación existente entre dicha clase y el resto de clases implementadas. Inmediatamente después se presenta lo que se puede considerar como el cuerpo de este apéndice, esto es, la descripción de los elementos referenciados en los tres índices iniciales. Finalmente se adjuntan dos listas, ordenadas alfabéticamente, que contienen los miembros

[1] http://www.stack.nl/~dimitri/doxygen/

(variables y funciones) documentados en las clases y ficheros que integran el código fuente de FBT. Estas listas indican respectivamente las clases y ficheros en los que cada componente está implementado.

Como es norma habitual y extendida, todas las expresiones que aparecen subrayadas y en azul denotan que son hipervínculos, que pueden ser utilizados en la versión electrónica de la documentación impresa en este apéndice.

FBT: FIL to Büchi automaton Translator

Version 1.0

FBT Compound Index

FBT Compound List

Here are the classes, structs, unions and interfaces with brief descriptions:

FBT File Index

FBT File List

Here is a list of all documented files with brief descriptions:

FBT Hierarchical Index

FBT Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

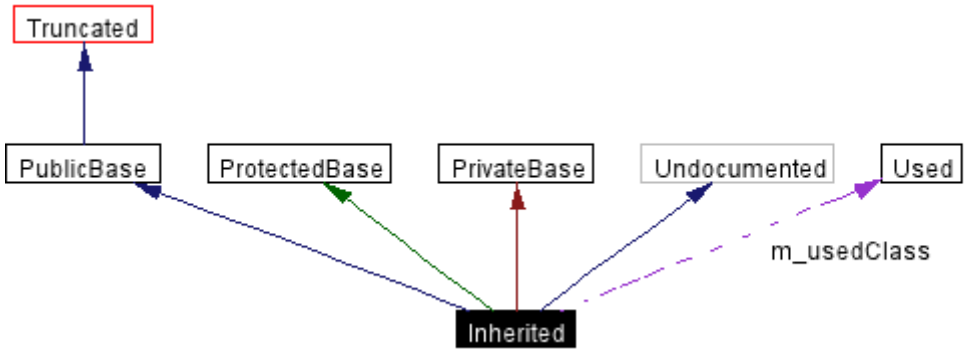
Graph Legend

This page explains how to interpret the graphs that are generated by doxygen.

Consider the following example:

```
/*! Invisible class because of truncation */
class Invisible { };

/*! Truncated class, inheritance relation is hidden */
class Truncated : public Invisible { };

/* Class not documented with doxygen comments */
class Undocumented { };

/*! Class that is inherited using public inheritance */
class PublicBase : public Truncated { };

/*! Class that is inherited using protected inheritance */
class ProtectedBase { };

/*! Class that is inherited using private inheritance */
class PrivateBase { };

/*! Class that is used by the Inherited class */
class Used { };

/*! Super class that inherits a number of other classes */
class Inherited : public PublicBase,
                  protected ProtectedBase,
                  private PrivateBase,
                  public Undocumented
{
  private:
    Used *m_usedClass;
};
```

This will result in the following graph:



The boxes in the above graph have the following meaning:

- A filled black box represents the struct or class for which the graph is generated.
- A box with a black border denotes a documented struct or class.
- A box with a grey border denotes an undocumented struct or class.
- A box with a red border denotes a documented struct or class for which not all inheritance/containment relations are shown. A graph is truncated if it does not fit within the specified boundaries.

The arrows have the following meaning:

- A dark blue arrow is used to visualize a public inheritance relation between two classes.
- A dark green arrow is used for protected inheritance.
- A dark red arrow is used for private inheritance.
- A purple dashed arrow is used if a class is contained or used by another class. The arrow is labeled with the variable(s) through which the pointed class or struct is accessible.

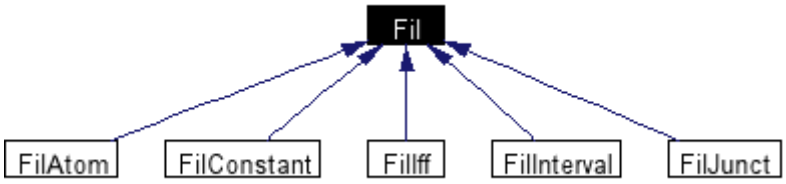
FBT Class Documentation

Fil Class Reference

FilAbstract base class for FIL formulas.

`#include <Fil.h>`

Inheritance diagram for Fil:



Collaboration diagram for Fil:

Public Types

- enum Kind { **Atom**, **Constant**, **Junct**, **Iff**, **Interval** }
  Formula kinds.

Public Methods

- virtual class Fil& negClone () const=0
  Returns a clone of the negation of this FIL formula.
- virtual enum Kind getKind () const=0
  Determine the kind of the formula.
- virtual bool isPurelyProp () const=0
  Determine whether the formula is purely propositional.
- bool operator< (const class Fil &other) const
  Less-than comparison.
- bool operator!= (const class Fil &other) const
  Non-equal comparison.
- virtual void expand (unsigned state, class FilGraphNode &node, class FilGraph &graph) const=0
  Implements operator/operand specific details of the expansion algorithm.

Parameters:

state	state number in the generalized Büchi automaton
node	the node representing the formula
graph	graph representation of the generalized Büchi automaton

- virtual void display (FILE *out) const=0
  Stream output.

Protected Methods

- Fil ()
  Constructor.
- virtual ~Fil ()
  Destructor.

Static Protected Methods

- class Fil& insert (class Fil &fil_f)
  Registrator of FIL formula references.

  Parameters:

fil_f	formula to be registered

  Returns:

  an equivalent FIL formula

Private Methods

- Fil (const class Fil &old)
  Copy constructor.
- class Fil& operator= (const class Fil &old)
  Assignment operator.

Static Private Attributes

- class Store m_store
  Set of instantiated FIL formulae.

Friends

- class **Store**

The documentation for this class was generated from the following file:

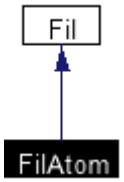
- Fil.h

FilAtom Class Reference

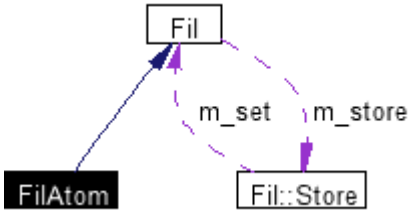
FilAtomFIL literals, i.e. atomic propositions, negated or not.

`#include <Fil.h>`

Inheritance diagram for FilAtom:



Collaboration diagram for FilAtom:



Public Methods

- class Fil& negClone () const
  Returns a clone of the negation of this FIL formula.
- enum Kind getKind () const
  Determine the kind of the formula.
- bool isNegated () const
  Determine whether the atom is negated.
- unsigned getValue () const
  Determine the value of the proposition.
- bool isPurelyProp () const
  Determine whether the formula is purely propositional.
- bool operator< (const class FilAtom &other) const
  Less-than comparison.
- void expand (unsigned state, class FilGraphNode &node, class FilGraph &graph) const
  Implements operator/operand specific details of the expansion algorithm.

  Parameters:

state	state number in the generalized Büchi automaton
node	the node representing the formula
graph	graph representation of the generalized Büchi automaton

- void display (FILE *out) const
  Stream output.

Static Public Methods

- class Fil& construct (unsigned value, bool negated)
  Constructor.

  Parameters:

value	proposition number
negated	flag: is the proposition negated?

  Returns:

  a FIL literal, i.e. atomic proposition, negated or not

Private Methods

- FilAtom (unsigned value, bool negated=false)
  Constructor.
- FilAtom (const class FilAtom &old)
  Copy constructor.
- class FilAtom& operator= (const class FilAtom &old)
  Assignment operator.
- ~FilAtom ()
  Destructor.

Private Attributes

- unsigned m_value
  The proposition number.
- bool m_negated
  Flag: is the proposition negated?

The documentation for this class was generated from the following file:

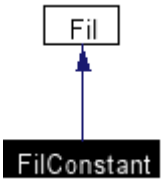
- Fil.h

FilConstant Class Reference

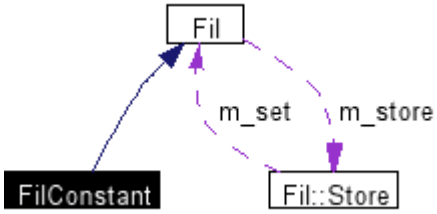
FilConstantFIL constants 'true' and 'false'.

`#include <Fil.h>`

Inheritance diagram for FilConstant:



Collaboration diagram for FilConstant:



Public Methods

- class Fil& negClone () const
  Returns a clone of the negation of this FIL formula.
- enum Kind getKind () const
  Determine the kind of the formula.
- bool isPurelyProp () const
  Determine whether the formula is purely propositional.
- operator bool () const
  Convert to truth value.
- bool operator< (const class FilConstant &other) const
  Less-than comparison.
- void expand (unsigned state, class FilGraphNode &node, class FilGraph &graph) const
  Implements operator/operand specific details of the expansion algorithm.

  Parameters:

state	state number in the generalized Büchi automaton
node	the node representing the formula
graph	graph representation of the generalized Büchi automaton

- void display (FILE *out) const
  Stream output.

Static Public Methods

- class Fil& construct (bool true_)
  Constructor.

  Parameters:

  true_ the constant 'true' or 'false'

  Returns:

  a FIL constant

Private Methods

- FilConstant (bool true_)
  Constructor.
- FilConstant (const class FilConstant &old)
  Copy constructor.
- class FilConstant& operator= (const class FilConstant &old)
  Assignment operator.
- ~FilConstant ()
  Destructor.

Private Attributes

- bool m_true
  The truth value of the constant.

The documentation for this class was generated from the following file:

- Fil.h

Fil::Filess Struct Reference

Fil::FilessInner struct for comparing FIL formulas.

Public Methods

- bool operator() (const class Fil *f1, const class Fil *f2) const
  Lexicographic order of FIL formulae.

  Parameters:

f1	pointer to first formula to compare
f2	pointer to second formula to compare

  Returns:

  true if f1 is before f2

The documentation for this struct was generated from the following file:

- Fil.h

Fil::Store Class Reference

Fil::StoreInner class for permanent and unique storing of each FIL formula.

Collaboration diagram for Fil::Store:

Public Methods

- Store ()
  Constructor.
- ~Store ()
  Destructor.
- class Fil& insert (class Fil &fil)
  Insert a formula to the set.

Private Methods

- Store (const class Store &)
  Copy constructor.
- class Store& operator= (const class Store &)
  Assignment operator.

Private Attributes

- std::set<class Fil*,struct Filess> m_set
  Set of FIL formulae.

The documentation for this class was generated from the following file:

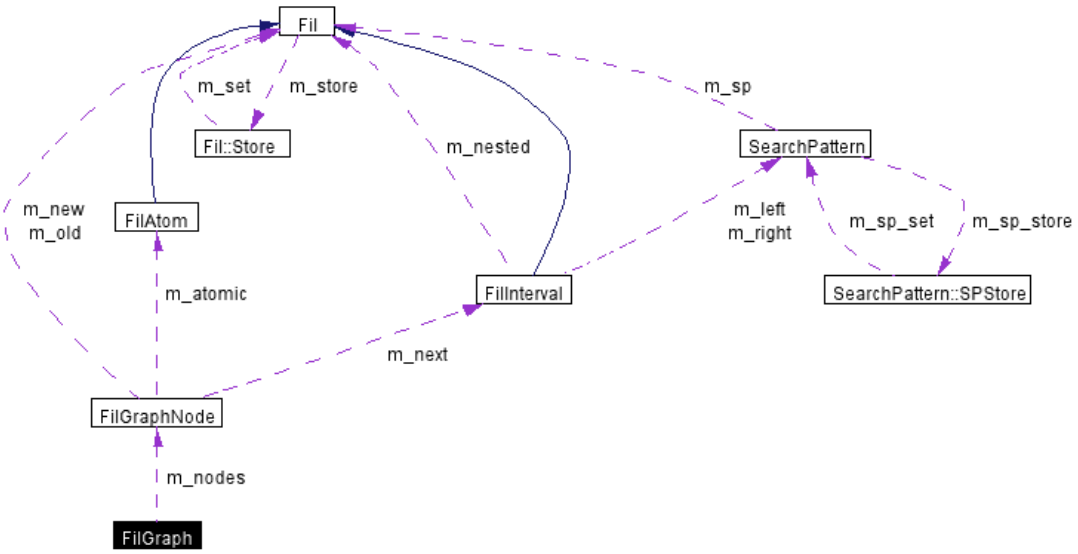
- Fil.h

FilGraph Class Reference

FilGraphGraph generated from a FIL formula.

`#include <FilGraph.h>`

Collaboration diagram for FilGraph:



Public Types

- typedef std::map<unsigned,class FilGraphNode> Map
  Map from state numbers to graph nodes.
- typedef Map::iterator iterator
  Iterator to the map.
- typedef Map::const_iterator const_iterator
  Constant iterator to the map.

Public Methods

- void expand (unsigned state, class FilGraphNode &node)
  Expand a FIL subformula into the graph.

  Parameters:

state	state number in the automaton
node	the node representing the FIL subformula

- void add (class FilGraphNode &node)
  Add a new node that is immediately expanded.

Parameters:

node the node representing a new FIL subformula

- FilGraph (const class Fil &formula)
  Constructs a graph from a given FIL formula.
- ~FilGraph ()
  Destructor.

Accessors to the graph nodes

- bool **empty** () const
- void **clear** ()
- iterator **begin** ()
- iterator **end** ()
- const_iterator **begin** () const
- const_iterator **end** () const

Private Methods

- FilGraph (const class FilGraph &old)
  Copy constructor.
- class FilGraph& operator= (const class FilGraph &other)
  Assignment operator.

Private Attributes

- unsigned m_next
  Next available state number.
- std::map<unsigned,class FilGraphNode> m_nodes
  Map from state numbers to graph nodes (0=initial state).

Detailed Description

Graph representation of a generalized Büchi automaton corresponding to a FIL formula.

A FilGraph is automatically constructed from a given FIL formula. All relevant information is fully stored to the graph, so that the formula itself may well be destroyed right after the conversion. The FIL operators/operands provide themselves the knowledge of how to be expanded into a FilGraph.

The documentation for this class was generated from the following file:

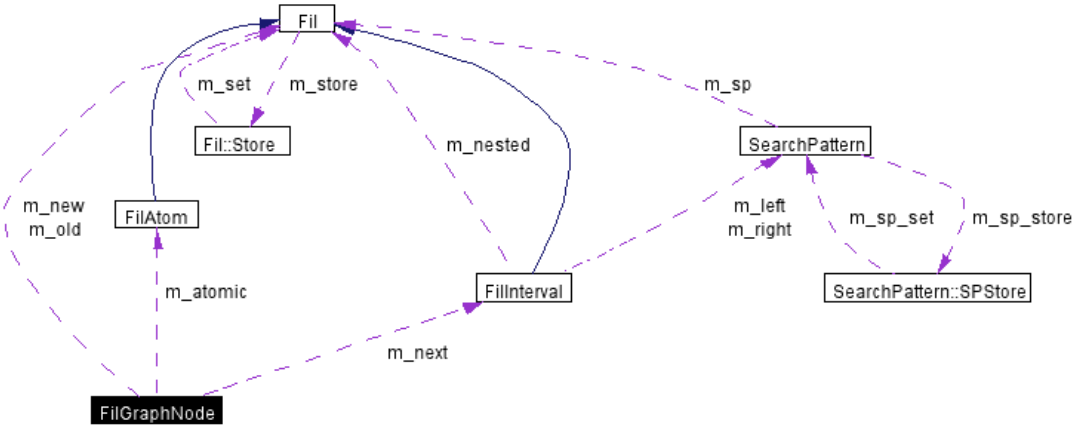
- FilGraph.h

FilGraphNode Class Reference

FilGraphNodeNode of a graph that represents a FIL formula.

`#include <FilGraph.h>`

Collaboration diagram for FilGraphNode:



Public Methods

- FilGraphNode ()
  Default constructor.
- FilGraphNode (const std::set< unsigned > &incoming, const std::set< const class Fil *> &neww, const std::set< const class Fil *> &old, const std::set< const class FilAtom *> &atomic, const std::set< const class FilInterval *> &next)
  Constructs a specific node.

  Parameters:

incoming	predecessors or incoming edges to the node
neww	set of formulas to be processed
old	set of formulas already processed
atomic	set of literals
next	set of formulas that must hold in next nodes

- FilGraphNode (const class FilGraphNode &node)
  Copy constructor.
- ~FilGraphNode ()
  Destructor.

Public Attributes

- std::set<unsigned> m_incoming
  From which nodes can one come to this node.
- std::set<const class Fil*> m_new
  Set of formulas to be processed that must hold in the node.

- std::set<const class Fil*> m_old
  Set of formulas already processed that must hold in the node.
- std::set<const class FilAtom*> m_atomic
  Set of literals that must hold in the node, when it is entered.
- std::set<const class FilInterval*> m_next
  Set of formulas that must hold in all the immediate next nodes.

Private Methods

- class FilGraphNode& operator= (const class FilGraphNode &node)
  Assignment operator.

Detailed Description

A node in a graph representing a FIL formula.

The graph itself is a set of nodes of this kind. The FilGraphNode class is merely a front end for the set representation.

The documentation for this class was generated from the following file:

- FilGraph.h

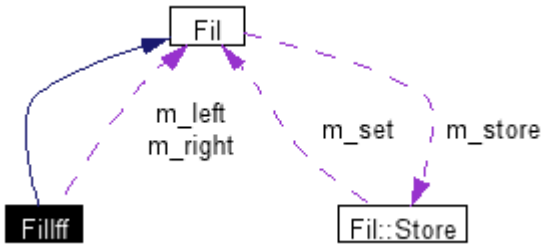
FillIff Class Reference

FilIffFIL equivalences and exclusive disjunctions.

`#include <Fil.h>`

Inheritance diagram for FilIff:

Collaboration diagram for FilIff:



Public Methods

- class Fil& negClone () const
  Returns a clone of the negation of this FIL formula.
- enum Kind getKind () const
  Determine the kind of the formula.
- bool isPurelyProp () const
  Determine whether the formula is purely propositional.
- bool isIff () const
  Determine the operator kind: equivalence instead of exclusive disjunction.
- class Fil& getLeft () const
  Get the left-hand-side subformula.
- class Fil& getRight () const
  Get the right-hand-side subformula.
- bool operator< (const class FillIff &other) const
  Less-than comparison.
- void expand (unsigned state, class FilGraphNode &node, class FilGraph &graph) const
  Implements operator/operand specific details of the expansion algorithm.

  Parameters:

  state state number in the generalized Büchi automaton

node the node representing the formula
graph graph representation of the generalized Büchi automaton

- void display (FILE *out) const
  Stream output.

Static Public Methods

- class Fil& construct (bool iff, class Fil &l, class Fil &r, bool pp)
  Optimizing constructor.

  Parameters:

  iff flag: equivalence instead of exclusive disjunction
  l the left-hand-side formula
  r the right-hand-side formula
  pp flag: is the formula purely propositional?

  Returns:

  an equivalent formula

Private Methods

- FilIff (bool iff, class Fil &l, class Fil &r, bool pp)
  Constructor.

  Parameters:

  iff flag: equivalence instead of exclusive disjunction
  l the left-hand-side formula
  r the right-hand-side formula
  pp flag: is the formula purely propositional?

- FilIff (const class FilIff &old)
  Copy constructor.
- class FilIff& operator= (const class FilIff &old)
  Assignment operator.
- ~FilIff ()
  Destructor.

Private Attributes

- bool m_iff
  Flag: equivalence instead of exclusive disjunction?
- class Fil& m_left
  The left-hand-side sub-formula.
- class Fil& m_right
  The right-hand-side sub-formula.

- bool m_purely
  Flag: is the formula purely propositional?

The documentation for this class was generated from the following file:

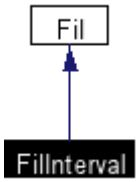
- Fil.h

FilInterval Class Reference

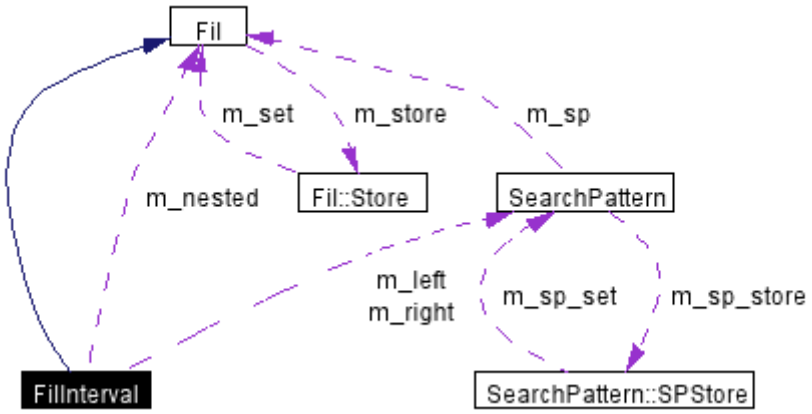
FilIntervalFIL interval formulas.

`#include <Fil.h>`

Inheritance diagram for FilInterval:



Collaboration diagram for FilInterval:



Public Types

- typedef std::pair<const class Fil*, const class Fil*> red_pair
  Pairs of formulas (reductor, reduct).
- typedef std::set<red_pair> red_pair_set
  Set of pairs (reductor, reduct).
- enum BinOp { **Con**, **Dis**, **Iff**, **Xor** }
  Kind of binary propositional operators that can be nested to an interval.

Public Methods

- class Fil& negClone () const
  Returns a clone of the negation of this FIL formula.
- enum Kind getKind () const
  Determine the kind of the formula.
- bool isPurelyProp () const
  Determine whether the formula is purely propositional.

- bool isNegated () const
  Determine whether this is a negated formula.
- class SearchPattern& getLeftPattern () const
  Get the left search pattern.
- class SearchPattern& getRightPattern () const
  Get the right search pattern.
- class Fil& getNested () const
  Get the subformula nested to the interval.
- bool isEventuality () const
  Determine whether this is an eventuality formula.
- bool isSeqCurrentMod () const
  Is the formula a pure sequence of current interval modalities?
- bool isNestedInvarOrEvent (bool &eventuality, bool negation=false) const
  Is the formula an invariant or eventuality nested to a sequence of current intervals?

  Parameters:

eventuality	flag: eventuality instead of invariant property nested
negation	flag: must the nested formula be negated?

  See also:

  isSeqCurrentMod()

- bool isNegFirstNotCurrentIntv () const
  Is the first nested non-current interval negated?

  See also:

  isSeqCurrentMod()

- class Fil& getSubformula (bool left, bool negated=false) const
  When the formula must be split, get one of the formulas in which it is split.

  Parameters:

left	flag: left instead of right subformula operand
negated	flag: must the nested formula be negated?

  See also:

  mustBeSplit()

- class Fil& getLastSubformula (bool negation=false) const
  When the formula is a pure sequence of current intervals, get its last subformula.

  Parameters:

negation	flag: must the nested formula be negated?

  See also:

  isSeqCurrentMod()

- class Fil& getInvariantNormalForm () const
  Get the normal form of an invariant property nested to a current interval sequence.
- class Fil& getPrefixNotCollapsed () const
  Get the prefix not collapsed of the formula.
- bool operator< (const class FilInterval &other) const
  Less-than comparison.
- bool mustBeSplit (enum BinOp &in_op, bool negation=false) const
  Must the formula be split in two?

  Parameters:

in_op	kind of operator of the Junct or Iff subformula
negation	flag: must the nested formula be negated?

- void reduction (red_pair_set &reductors_reducts) const
  Calculate all the reductors and reducts of the formula.

  Parameters:

reductors_reducts	set of pairs (reductor, reduct)

- void expand (unsigned state, class FilGraphNode &node, class FilGraph &graph) const
  Implements operator/operand specific details of the expansion algorithm.

  Parameters:

state	state number in the generalized Büchi automaton
node	the node representing the formula
graph	graph representation of the generalized Büchi automaton

- void display (FILE *out) const
  Stream output.

Static Public Methods

- class Fil& construct (bool negated, class SearchPattern &l, class SearchPattern &r, class Fil &n)
  Constructor.

  Parameters:

negated	flag: is the interval formula negated?
l	the left search pattern
r	the right search pattern
n	the formula nested to the interval

  Returns:

  a FIL formula

Private Methods

- FilInterval (bool negated, class SearchPattern &l, class SearchPattern &r, class Fil &n)
  Constructor.

  Parameters:

negated	flag: is the interval formula negated?
l	the left search pattern
r	the right search pattern
n	the formula nested to the interval

- class Fil& getSuffix () const
  Get the suffix of the prefix not collapsed of a formula.

  See also:

  getPrefixNotCollapsed()

- FilInterval (const class FilInterval &old)
  Copy constructor.
- class FilInterval& operator= (const class FilInterval &old)
  Assignment operator.
- ~FilInterval ()
  Destructor.

Private Attributes

- bool m_negated
  Flag: is the interval formula negated?
- class SearchPattern& m_left
  Search pattern that locates the left end-point of the interval.
- class SearchPattern& m_right
  Search pattern that locates the right end-point of the interval.
- class Fil& m_nested
  The sub-formula nested to the interval.

Member Function Documentation

class Fil & FilInterval::getInvariantNormalForm () const [inline]

Get the "normal form" of an invariant property nested to a prefix of current interval modalities, i.e. all their intervals are not negated.

class Fil & FilInterval::getLastSubformula (bool *negation* = false) const [inline]

Get the properly signed last subformula (i.e. the purely propositional one) of a formula that is made up of a sequence of current interval modalities.

class Fil & FilInterval::getPrefixNotCollapsed () const [inline]

Get the formula asserting that the prefix of current interval modalities cannot be collapsed until its nested formula is reduced.

class Fil & FilInterval::getSubformula (bool *left*, bool *negated* = false) const [inline]

Get the requested subformula of the current formula, which has a non-purely propositional Junct or Iff subformula nested to a prefix made up of a sequence of current interval modalities. Thus, the returned formula has only one of the subformula operands (the left or right one, negated or not, according to the function parameters) nested to the same prefix.

class Fil & FilInterval::getSuffix () const [inline, private]

Get the "suffix" or nested formula (made up of current interval modalities) of the formula returned by the function getPrefixNotCollapsed().

bool FilInterval::isNegFirstNotCurrentIntv () const [inline]

Determine whether the first nested interval that is not a current interval modality is negated.

bool FilInterval::isNestedInvarOrEvent (bool & *eventuality*, bool *negation* = false) const [inline]

Determine whether there is either an invariant or an eventuality property nested to a prefix made up of a sequence of current interval modalities.

bool FilInterval::isSeqCurrentMod () const [inline]

Determine whether the formula is a pure sequence of current interval modalities, i.e. all their left search patterns are trivial (empty) and the last subformula nested is purely propositional.

bool FilInterval::mustBeSplit (enum BinOp & *in_op*, bool *negation* = false) const

Determine whether the formula must be split in two. This happens when, nested to a prefix of current interval modalities, it has either a Junct or an Iff subformula with at least one of its operands containing an interval formula. In addition, the sign (negation) of each formula of the prefix is taken into account in order to determine the type of operator (returned by the last parameter) of the Junct or Iff subformula.

The documentation for this class was generated from the following files:

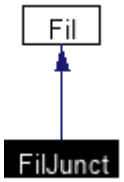
- Fil.h
- Fil.C

FilJunct Class Reference

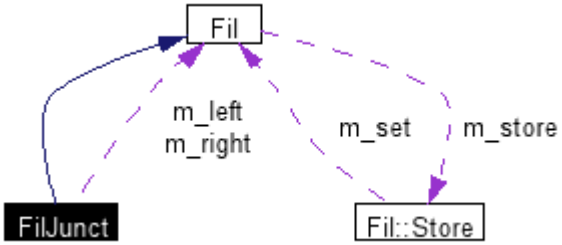
FilJunctFIL conjunctions and disjunctions.

`#include <Fil.h>`

Inheritance diagram for FilJunct:



Collaboration diagram for FilJunct:



Public Methods

- class Fil& negClone () const
  Returns a clone of the negation of this FIL formula.
- enum Kind getKind () const
  Determine the kind of the formula.
- bool isPurelyProp () const
  Determine whether the formula is purely propositional.
- bool isCon () const
  Determine the operator kind: conjunction instead of disjunction.
- class Fil& getLeft () const
  Get the left-hand-side subformula.
- class Fil& getRight () const
  Get the right-hand-side subformula.
- bool operator< (const class FilJunct &other) const
  Less-than comparison.
- void expand (unsigned state, class FilGraphNode &node, class FilGraph &graph) const
  Implements operator/operand specific details of the expansion algorithm.

  Parameters:

  state state number in the generalized Büchi automaton

node the node representing the formula
graph graph representation of the generalized Büchi automaton

- void display (FILE *out) const
  Stream output.

Static Public Methods

- class Fil& construct (bool con, class Fil &l, class Fil &r, bool pp)
  Optimizing constructor.

  Parameters:

  con flag: conjunction instead of disjunction
  l the left-hand-side formula
  r the right-hand-side formula
  pp flag: is the formula purely propositional?

  Returns:

  an equivalent formula

Private Methods

- FilJunct (bool con, class Fil &l, class Fil &r, bool pp)
  Constructor.

  Parameters:

  con flag: conjunction instead of disjunction
  l the left-hand-side formula
  r the right-hand-side formula
  pp flag: is the formula purely propositional?

- FilJunct (const class FilJunct &old)
  Copy constructor.
- class FilJunct& operator= (const class FilJunct &old)
  Assignment operator.
- ~FilJunct ()
  Destructor.

Private Attributes

- bool m_con
  Flag: conjunction instead of disjunction?
- class Fil& m_left
  The left-hand-side sub-formula.
- class Fil& m_right
  The right-hand-side sub-formula.

- bool m_purely
  Flag: is the formula purely propositional?

The documentation for this class was generated from the following file:

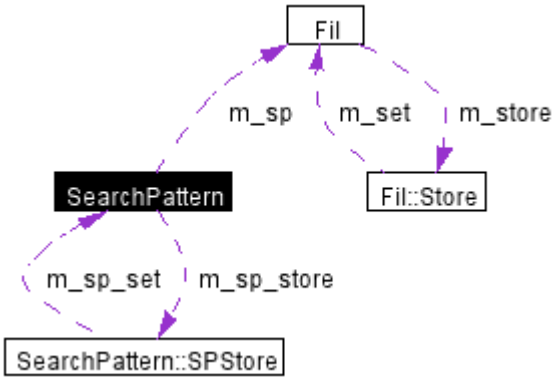
- Fil.h

SearchPattern Class Reference

SearchPatternFIL search patterns.

`#include <Fil.h>`

Collaboration diagram for SearchPattern:



Public Methods

- bool isTrivial () const
  Is a trivial search pattern?
- int size () const
  Return the number of searches that comprise the search pattern.
- const class Fil* operator[] (int i) const
  Return the i-th element (search) of the pattern.
- class Fil* getHead () const
  Get the first search of the pattern.
- const class Fil* getLastSearch () const
  Get the last search of the pattern.
- class SearchPattern& getTail () const
  Get all the search pattern except its first search.
- bool operator< (const class SearchPattern &other) const
  Less-than comparison.
- void display (FILE *out, bool first) const
  Stream output.

  Parameters:

  first flag: first instead of second search pattern

Static Public Methods

- class SearchPattern& construct (std::deque< const class Fil *> &sp)
  Search pattern constructor.

Parameters:

sp the search pattern

Returns:

a FIL search pattern

Private Methods

- SearchPattern (std::deque< const class Fil *> &sp)
  Non-trivial search pattern constructor.
- virtual ~SearchPattern ()
  Destructor.
- SearchPattern (const class SearchPattern &old)
  Copy constructor.
- class SearchPattern& operator= (const class SearchPattern &old)
  Assignment operator.

Private Attributes

- std::deque<const class Fil*> m_sp
  A FIL search pattern, i.e. a sequence of FIL formulas (searches).

Static Private Attributes

- class SPStore m_sp_store
  Set of instantiated FIL search patterns.

Friends

- class SPStore

The documentation for this class was generated from the following file:

- Fil.h

SearchPattern::SPless Struct Reference

SearchPattern::SPlessInner struct for comparing FIL search patterns.

Public Methods

- bool operator() (const class SearchPattern *sp1, const class SearchPattern *sp2) const

  Lexicographic order of FIL search patterns.

  Parameters:

  sp1 pointer to first search pattern to compare
  sp2 pointer to second search pattern to compare

  Returns:

  true if sp1 is before sp2

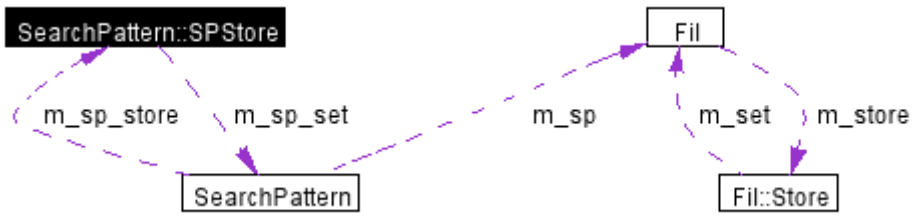
The documentation for this struct was generated from the following file:

- Fil.h

SearchPattern::SPStore Class Reference

SearchPattern::SPStoreInner class for permanent and unique storing of each FIL search pattern.

Collaboration diagram for SearchPattern::SPStore:



Public Methods

- SPStore ()
  Constructor.
- ~SPStore ()
  Destructor.
- class SearchPattern& insert (class SearchPattern &sp)
  Insert a search pattern to the set.

Private Methods

- SPStore (const class SPStore &)
  Copy constructor.
- class SPStore& operator= (const class SPStore &)
  Assignment operator.

Private Attributes

- std::set<class SearchPattern*,struct SPless> m_sp_set
  Set of FIL search patterns.

The documentation for this class was generated from the following file:

- Fil.h

FBT File Documentation

fbt.C File Reference

fbt.CThe main program of FBT: FIL to Büchi automaton Translator.

Structure of the acceptance sets

- typedef std::set<const class FilInterval*> formulas_to_find_t
  Eventuality formulas to find in a node to determine that it is not accepting.
- typedef std::pair<unsigned, formulas_to_find_t> acceptance_set_t
  The first component is the acceptance set number.
- typedef std::map<const class Fil*, acceptance_set_t> acceptance_map_t
  The map key is the target formula of the last search of all its eventualities.

Functions

- class Fil* readFormula (bool negation)
  Read a FIL formula from standard input.

  Parameters:

  negation flag: negate the formula

  Returns:

  the parsed formula, or NULL on error

- class SearchPattern* readPattern (bool first)
  Read a search pattern from standard input.

  Parameters:

  first flag: first search pattern instead of the second one

  Returns:

  the parsed search pattern, or NULL on error

- void printGates (const class FilGraph &gba, unsigned state)
  Display the gates for arcs leaving a specific Büchi automaton state.

  Parameters:

  gba the generalized Büchi automaton
  state the state whose successor arcs are to be displayed

- void translateFormula (const class Fil &formula)
  Translate a formula into a generalized Büchi automaton, which is output to standard output.

  Parameters:

  formula the formula to be translated

- void printInternalContents (const class Fil &formula)

Print the contents of the main internal structures used during the generation of the automaton.

Parameters:

formula the formula from which the automaton is generated

- int main (int argc, char *argv[])
  The main program.

  Parameters:

  argc number of command-line arguments
  argv the command-line arguments

Function Documentation

void printInternalContents (const class Fil & *formula*) [static]

This function, which is only used in debugging mode, prints the contents of the set of nodes representing the automaton generated from a formula, its total number of nodes, transitions and acceptance sets, and the contents of the structure storing the acceptance sets.

void translateFormula (const class Fil & *formula*) [static]

Firstly, the graph of the generalized Büchi automaton is constructed. Secondly, the acceptance sets are determined from the set of processed eventuality formulas previously built. Finally, the accepting states are determined and the output is printed.

Fil.C File Reference

Fil.CSource file for FIL formulas.

Detailed Description

It contains the implementation of the largest functions of the classes that implement the different types of FIL formulas.

Fil.h File Reference

Fil.hHeader file for FIL formulas.

Compounds

- class Fil
  Abstract base class for FIL formulas.
- struct Fil::Filess
  Inner struct for comparing FIL formulas.
- class Fil::Store
  Inner class for permanent and unique storing of each FIL formula.
- class FilAtom
  FIL literals, i.e. atomic propositions, negated or not.
- class FilConstant
  FIL constants 'true' and 'false'.
- class FilIff
  FIL equivalences and exclusive disjunctions.
- class FilInterval
  FIL interval formulas.
- class FilJunct
  FIL conjunctions and disjunctions.
- class SearchPattern
  FIL search patterns.
- struct SearchPattern::SPless
  Inner struct for comparing FIL search patterns.
- class SearchPattern::SPStore
  Inner class for permanent and unique storing of each FIL search pattern.

FilGraph.C File Reference

FilGraph.CSource file for graphs generated from a FIL formula.

Variables

- const std::set<const class Fil*> emp_fil
  An empty set of FIL formulas.
- const std::set<const class FilAtom*> emp_lit
  An empty set of literals.
- const std::set<const class FilInterval*> emp_int
  An empty set of FIL interval formulas.

Detailed Description

It contains the implementation of constructor FilGraph::FilGraph and function FilGraph::expand.

FilGraph.h File Reference

FilGraph.hHeader file for graphs generated from a FIL formula.

Compounds

- class FilGraph
  Graph generated from a FIL formula.
- class FilGraphNode
  Node of a graph that represents a FIL formula.

---

gba2dot.c File Reference

gba2dot.cFilter that converts a generalized Büchi automaton to a format viewable with GraphViz.

Functions

- int parseGate (void)
  Copy a gate condition from standard input to standard output.

  Returns:

  zero if everything is ok; nonzero on error

- int main (int argc, char **argv)
  The main program.

  Parameters:

argc	number of command-line arguments
argv	the command-line arguments

Detailed Description

This filter converts a generalized Büchi automaton (generated by either LBT or FBT) to a GraphViz directed graph.

FBT Compound Members

a | b | c | d | e | f | g | i | k | m | n | o | r | s | ~

List of all documented class members with links to the classes they belong to:

- a -

- add() : **FilGraph**

- b -

- BinOp : **FilInterval**

- c -

- const_iterator : **FilGraph**
- construct() : **FilInterval**, **SearchPattern**, **FilIff**, **FilJunct**, **FilConstant**, **FilAtom**

- d -

- display() : **FilInterval**, **SearchPattern**, **FilIff**, **FilJunct**, **FilConstant**, **FilAtom**, **Fil**

- e -

- expand() : **FilGraph**, **FilInterval**, **FilIff**, **FilJunct**, **FilConstant**, **FilAtom**, **Fil**

- f -

- Fil() : **Fil**
- FilAtom() : **FilAtom**
- FilConstant() : **FilConstant**
- FilGraph() : **FilGraph**
- FilGraphNode() : **FilGraphNode**
- FilIff() : **FilIff**
- FilInterval() : **FilInterval**
- FilJunct() : **FilJunct**

- g -

- getHead() : **SearchPattern**

- m -

- n -

- o -

FBT File Members

a | e | f | m | p | r | t

List of all documented file members with links to the documentation:

- a -

acceptance_map_t : **fbt.C**

acceptance_set_t : **fbt.C**

- e -

emp_fil : **FilGraph.C**

emp_int : **FilGraph.C**

emp_lit : **FilGraph.C**

- f -

formulas_to_find_t : **fbt.C**

- m -

main() : **gba2dot.c**, **fbt.C**

- p -

parseGate() : **gba2dot.c**

printGates() : **fbt.C**

printInternalContents() : **fbt.C**

- r -

readFormula() : **fbt.C**

readPattern() : **fbt.C**

- t -

translateFormula() : **fbt.C**

REFERENCIAS BIBLIOGRÁFICAS

[Aaby88] A. AABY, K. T. NARAYANA, "Propositional temporal interval logic is PSPACE-complete", *Proceedings of the Conference on Automated Deduction*, Argonne, Illinois, USA, May 1988, Lecture Notes in Computer Science 193, Springer-Verlag, pp. 218-237.

[Abadi95] M. ABADI, L. LAMPORT, "Conjoining Specifications", *ACM Transactions on Programming Languages and Systems*, 17(3), May 1995, pp. 507-535.

[Aggarw83] S. AGGARWAL, R. P. KURSHAN, K. K. SABNANI, "A calculus for protocol specification and validation", *Proceedings of the 3rd Workshop on Protocol Specification, Testing and Verification*, Rüschlikon, Switzerland, May/June 1983, North-Holland, pp. 19-34.

[Allen83] J. F. ALLEN, "Maintaining knowledge about temporal intervals", *Communication of the ACM*, 26(11), November 1983, pp. 832-843.

[Alur89] R. ALUR, T. HENZINGER, "A really temporal logic", *Proceedings of the 30th IEEE Symposium on Foundations of Computer Science*, 1989, IEEE Computer Society Press, New York, NY, USA, pp. 164-169.

[Alur96] R. ALUR, G. J. HOLZMANN, D. PELED, "An analyzer for message sequence charts", *Software Concepts and Tools*, 17(2), March 1996, pp. 70-77. A preliminary version appeared in *Proceedings of the 2nd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems* (TACAS'96), Passau, Germany, March 1996, Lecture Notes in Computer Science 1055, pp. 35-48.

[Apt97] K. R. APT, E.-R. OLDEROG, *Verification of Sequential and Concurrent Programs*, 2nd edition, Graduate Texts in Computer Science, Springer-Verlag, New York, NY, USA, 1997.

[Baeten90] J. C. M. BAETEN, W. P. WEIJLAND, "Process Algebra", *Cambridge Tracts in Theoretical Computer Science*, 18, Cambridge University Press, Cambridge, England, 1990.

[Ball96] T. A. BALL, S. G. EICK, "Software visualization in the large", *IEEE Computer*, 29(4), April 1996, pp. 33-43.

[Barrin86] H. BARRINGER, R. KUIPER, A. PNUELI, "A really abstract concurrent model and its temporal logic", *Proceedings of the 18th ACM Symposium on Principles of Programming Languages*, St. Petersburg Beach, Florida, USA, January 1986, pp. 173-183.

[Beizer90] B. BEIZER, *Software Testing Techniques*, 2nd edition, Van Nostrand Reinhold, New York, NY, USA, 1990.

[Beizer95] B. BEIZER, *Black-Box Testing: Techniques for Functional Testing of Software and Systems*, John Wiley & Sons, 1995.

[Ben-Ar83] M. BEN-ARI, A. PNUELI, Z. MANNA, "The temporal logic of branching time", *Acta Informatica*, 20, 1983, pp. 207-226.

[Bensal98] S. BENSALEM, Y. LAKHNECH, S. OWRE, "Computing abstractions of innite state systems compositionally and automatically", *Proceedings of the 10th International Conference on Computer Aided Verification* (CAV'98), Vancouver, British Columbia, Canada, June/July 1998, Lecture Notes in Computer Science 1427, Springer-Verlag, pp. 319-331.

[Bérard01] B. BERARD, M. BIDOIT, A. FINKEL, F. LAROUSSINIE, A. PETIT, L. PETRUCCI, PH. SCHNOEBELEN, *Systems and Software Verification: Model-Checking Techniques and Tools*, P. McKenzie (tr.), Springer-Verlag, Berlin, Germany, 2001.

[Bjorne00] N. BJORNER, A. BROWNE, M. COLON, B. FINKBEINER, Z. MANNA, H. SIPMA, T. URIBE, "Verifying temporal properties of reactive systems: A STeP tutorial", *Formal Methods in System Design*, 16(3), June 2000, pp. 227-270.

[Boehm81] B. BOEHM, *Software Engineering Economics*, Prentice Hall, Englewood Cliffs, New Jersey, USA, 1981.

[Bologn89] T. BOLOGNESI, D. LATELLA, "Techniques for the formal definition of the G-LOTOS syntax", *Proceedings of the IEEE Workshop on Visual Languages*, Rome, Italy, October 1989, pp. 43-49.

[Booch99] G. BOOCH, J. RUMBAUGH, I. JACOBSON, *The Unified Modeling Language User Guide*, Addison-Wesley, Reading, Massachusetts, USA, 1999.

[Bowem95] J. P. BOWEN, M. G. HINCHEY, "Seven more myths of formal methods", *IEEE Software*, 12(4), July 1995, pp. 34-41.

[Bowman98] H. BOWMAN, S. J. THOMPSON, "A tableaux method for interval temporal logic with projection", *Proceedings of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods* (TABLEAUX'98), Oisterwijk (near Tilburg), The Netherlands, May 1998, Lecture Notes in Artificial Intelligence 1397, Springer-Verlag, pp. 108-123.

[Brown85] G. P. BROWN, R. T. CARLING, C. F. HEROT, D. A. KRAMLICH, P. SOUZA, "Program visualization: graphical support for software development", *IEEE Computer*, 18(8), August 1985, pp. 27-35.

[Bryant92] R. E. BRYANT, "Symbolic boolean manipulation with ordered bynary-decision diagrams", *ACM Computing Surveys*, 24(3), September 1992, pp. 293-318.

[Büchi62] J. R. BÜCHI, "On a decision method in restricted second-order arithmetic", *Proceedings of the 1960 International Congress on Logic, Methodology and Philosophy of Science*, Stanford University Press, Standford, California, USA, 1962, pp. 1-11.

[Buhr90] R. J. A. BUHR, *Practical visual techniques in system design with applications to Ada*, Prentice Hall, Englewood Cliffs, New Jersey, USA, 1990.

[Burch92] J. R. BURCH, E. M. CLARKE, K. L. MCMILLAN, D. L. DILL, L. J. HWANG, "Symbolic model checking: 10^{20} and beyond", *Information and Computation*, 98(2), June 1992, pp. 142-170.

[Chaoch91] Z. CHAOCHEN, C. A. R. HOARE, A. P. RAVN, "A calculus of durations", *Information Processing Letters*, 40(5), December 1991, pp. 269-276.

[Cheung90] T.-Y. CHEUNG, Y. YE, "An executor for graphical LOTOS", *Proceedings of the IFIP TC6/WG6.1 3rd International Conference on Formal Description Techniques for Distributed Systems and Communications Protocols* (FORTE'90), Madrid, Spain, November 1990, North-Holland, pp. 547-50.

[Clarke81] E. M. CLARKE, E. A. EMERSON, "Design and synthesis of synchronization skeletons using branching time temporal logic", *Proceedings of Workshop on Logic of Programs*, Yorktown Heights, New York, USA, May 1981, Lecture Notes in Computer Science 131, Springer-Verlag, pp. 52-71.

[Clarke86] E. M. CLARKE, E. A. EMERSON, A. P. SISTLA, "Automatic verification of finite-state concurrent systems using temporal logic specifications", *ACM Transactions on Programming Languages and Systems*, 8(2), April 1986, pp. 244-263.

[Clarke92] E. M. CLARKE, O. GRUMBERG, D. E. LONG, "Model checking and abstraction", *Proceedings of the 19th ACM Symposium on Principles of Programming Languages*, Albuquerque, New Mexico, USA, January 1992, pp. 343-354.

[Clarke96] E. M. CLARKE, K. L. MCMILLAN, S. CAMPOS, V. HARTONAS-GARMHAUSEN, "Symbolic model checking", *Proceedings of the 8th International Conference on Computer Aided Verification* (CAV'96), New Brunswick, New Jersey, USA, July/August 1996, Lecture Notes in Computer Science 1102, Springer-Verlag, pp. 419-422.

[Clarke99] E. M. CLARKE, O. GRUMBERG, D. A. PELED, *Model Checking*, The MIT Press, Cambridge, Massachusetts, USA, 1999.

[Couvre99] J.-M. COUVREUR, "On-the-fly verification of linear temporal logic", *Proceedings of the World Congress on Formal Methods in the Development of Computing Systems* (FM'99), *Volume I*, Toulouse, France, September 1999, Lecture Notes in Computer Science 1708, Springer-Verlag, pp. 253-271.

[Cole00] O. COLE, "White-box testing should check every line of code", *Dr. Dobb's Journal*, March 2000.

[Courco92] C. COURCOUBETIS, M. Y. VARDI, P. WOLPER, M. YANNAKAKIS, "Memory-efficient algorithms for the verification of temporal properties", *Formal Methods in System Design*, 1(2-3), October 1992, pp. 275-288.

[D'Agos99] M. D'AGOSTINO, D. M. GABBAY, R. HÄHNLE, J. POSEGGA (eds.), *The handbook of tableau methods*, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1999.

[Daniel99] M. DANIELE, F. GIUNCHIGLIA, M. Y. VARDI, "Improved automata generation for linear temporal logic", *Proceedings of the 11th Conference on Computer Aided Verification* (CAV'99), Trento, Italy, July 1999, Lecture Notes in Computer Science 1633, Springer-Verlag, pp. 249-260.

[David92] R. DAVID, *Petri nets and Grafcet: Tools for modeling discrete event systems*, Prentice Hall, New York, NY, USA, 1992.

[Deitel99] H. M. DEITEL, P. J. DEITEL, *C++ Cómo programar*, S. L. M. Ruiz Faudón, D. Morales Peake (trs.), Prentice Hall, 2ª edición, México, 1999.

[Dieste00] R. DIESTEL, *Graph Theory*, Springer-Verlag, 2nd edition, New York, NY, USA, 2000.

[Dill92] D. L. DILL, A. J. DREXLER, A. J. HU, C. H. YANG, "Protocol verification as a hardware design aid". *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors*, Cambridge, Massachusetts, USA, October 1992, IEEE Computer Society, pp. 522-525.

[Dillon94a] L. K. DILLON, G. KUTTY, P. M. MELLIAR-SMITH, L. E. MOSER, Y. S. RAMAKRISHNA, "A graphical interval logic for specifying concurrent systems", *ACM Transactions on Software Engineering Methodology*, 3(2), April 1994, pp. 131-165.

[Dillon94b] L. K. DILLON, Q. YU, "Oracles for checking temporal properties of concurrent systems", *Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, New Orleans, Louisiana, USA, December 1994, pp. 140-153.

[Emerso80] E. A. EMERSON, E. M. CLARKE, "Characterizing correctness properties of parallel programs using fixpoints", *Proceedings of the 7th International Colloquium on Automata, Languages and Programming* (ICALP'80), Noordweijkerhout, The Netherland, July 1980, Lecture Notes in Computer Science 85, Springer-Verlag, pp. 169-181.

[Emerso90] E. A. EMERSON, "Temporal and modal logic", *Handbook of Theoretical Computer Science, Volume B (Formal Models and Semantics)*, Chapter 16, J. van Leeuwen (ed.), Elsevier Science Publishers, Amsterdam, The Netherlands, 1990, pp. 995-1072.

[Emerso93] E. A. EMERSON, A. P. SISTLA, "Symmetry and model checking", *Proceedings of the 5th International Conference on Computer Aided Verification* (CAV'93), Elounda, Crete, Greece, June/July 1993, Lecture Notes in Computer Science 697, Springer-Verlag, pp. 463-478.

[Etessa00] K. ETESSAMI, G. HOLZMANN. "Optimizing Büchi automata", *Proceedings of the 11th International Conference on Concurrency Theory* (CONCUR'2000), University Park, Pennsylvania, USA, August 2000, Lecture Notes in Computer Science 1877, Springer-Verlag, pp. 153-167.

[Fairle85] R. FAIRLEY, *Software Engineering Concepts*, McGraw-Hill, New York, NY, USA, 1985.

[Floyd67] R. W. FLOYD, "Assigning meaning to programs", *Proceedings of Symposium on Applied Mathematics*, Providence, Rhode Island, USA, 1967, appeared in *Mathematical Aspects of Computer Science*, 19, American Mathematical Society, pp. 19-32.

[Fowler97] M. FOWLER, K. SCOTT, *UML Distilled: Applying the Standard Object Modeling Language*, Addison-Wesley, Reading, Massachusetts, USA, 1997.

[France86] N. FRANCEZ, *Fairness*, Springer-Verlag, Berlin, Germany, 1986.

[France92] N. FRANCEZ, *Program verification*, Addison-Wesley, Wokingham, England, 1992.

[Gansne00] E. R. GANSNER, S. C. NORTH, "An open graph visualization system and its applications to software engineering", *Software: Practice and Experience*, 30(11), September 2000, pp. 1203-1233.

[Gansne93] E. R. GANSNER, E. KOUTSOFIOS, S. C. NORTH, K.-P. VO, "A technique for drawing directed graphs", *IEEE Transactions on Software Engineering*, 19(13), May 1993, pp. 214--230.

[Gastin01] P. GASTIN, D. ODDOUX, "Fast LTL to Büchi automata translation", *Proceedings of the 13th International Conference on Computer Aided*

Verification (CAV'2001), Paris, France, July 2001, Lecture Notes in Computer Science 2102, Springer-Verlag, pp. 53-65.

[Gerth95] R. GERTH, D. PELED, M. Y. VARDI, P. WOLPER, "Simple on-the-fly automatic verification of linear temporal logic", *Proceedings of the 15th Workshop on Protocol Specification, Testing and Verification*, Warsaw, Poland, June 1995, Chapman & Hall, pp. 3-18.

[Giacal88] A. GIACALONE, S. A. SMOLKA, "Integrated environments for formally well-founded design and simulation of concurrent systems", *IEEE Transactions on Software Engineering*, 14(6), June 1988, pp. 787-801.

[Gibbon85] A. GIBBONS, *Algorithmic Graph Theory*, Cambridge University Press, Cambridge, England, 1985.

[Gliner84] E. P. GLINERT, S. L. TANIMOTO, "Pict: An interactive graphical programming environment", *IEEE Computer*, 17(11), November 1984, pp. 7-25.

[Godefr90] P. GODEFROID, "Using partial orders to improve automatic verification methods", *Proceedings of the 2nd Workshop on Computer Aided Verification* (CAV'90), New Brunswick, New Jersey, USA, June 1990, Lecture Notes in Computer Science 531, Springer-Verlag, pp. 176-185.

[Gordon93] M. J. C. GORDON, T. F. MELHAM (eds.), *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*, Cambridge University Press, Cambridge, England, 1993.

[Goswam92] A. GOSWAMI, M. BELL, M. JOSEPH, "ISL: An interval logic for the specification of real-time programs", *Proceedings of the 2nd Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, Nijmegen, The Netherlands, January 1992, Lecture Notes in Computer Science 571, Springer-Verlag, pp. 1-20.

[Graf93] S. GRAF, C. LOISEAUX, "A tool for symbolic program verification and abstraction", *Proceedings of the 5th International Conference on Computer Aided Verification* (CAV'93), Elounda, Crete, Greece, June/July 1993, Lecture Notes in Computer Science 697, Springer-Verlag, pp. 71-84.

[Graf97] S. GRAF, H. SAIDI, "Construction of abstract state graphs with PVS", *Proceedings of the 9th International Conference on Computer Aided Verification* (CAV'97), Haifa, Israel, June 1997, Lecture Notes in Computer Science 1254, Springer-Verlag, pp. 72-83.

[Grahlm97] B. GRAHLMANN, "The PEP tool", *Tool Presentations of the 18th International Conference on Application and Theory of Petri Nets*, Toulouse, France, June 1997, pp. 1-10.

[Grahlm99] B. GRAHLMANN, "The state of PEP", *Proceedings of the 7th International Conference on Algebraic Methodology and Software Technology*, Manaus, Amazonia, Brazil, January 1999, Lecture Notes in Computer Science 1548, Springer-Verlag, pp. 522-526.

[Gribom89] P. GRIBOMONT, P. WOLPER, "Temporal logic", *From modal logic to deductive databases*, A. Thayse (ed.), John Wiley & Sons, Chichester, England, 1989, pp. 165-233.

[Groce02] A. GROCE, D. PELED, M. YANNAKAKIS, "Adaptive Model Checking", *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (TACAS'2002), Grenoble, France, April 2002, Lecture Notes in Computer Science 2280, Springer-Verlag, pp. 357-370.

[Gunter99] E. L. GUNTER, D. PELED, "Path Exploration Tool", *Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (TACAS'99), Amsterdam, The Netherlands, March 1999, Lecture Notes in Computer Science 1579, Springer-Verlag, pp. 405-419.

[Gunter00] E. L. GUNTER, R. P. KURSHAN, D. PELED, "PET: An Interactive Software Testing Tool", *Proceedings of the 12th International Conference on Computer Aided Verification* (CAV'2000), Chicago, Illinois, USA, July 2000, Lecture Notes in Computer Science 1855, Springer-Verlag, pp. 552-556.

[Hall90] A. HALL, "Seven myths of formal methods", *IEEE Software*, 7(5), September 1990, pp. 11-19.

[Halper83] J. Y. HALPERN, Z. MANNA, B. MOSZKOWSKI, "A hardware semantics based on temporal intervals", *Proceedings of the 10th International Colloquium on Automata, Languages and Programming* (ICALP'83), Barcelona, Spain, July 1983, Lecture Notes in Computer Science 154, Springer-Verlag, pp. 278-291.

[Halper91] J. Y. HALPERN, Y. SHOHAM, "A propositional modal logic of time intervals", *Journal of the ACM*, 38(4), October 1991, pp. 935-962.

[Har'El90] Z. HAR'EL, R. P. KURSHAN, "Software for analytical development of communication protocols", *AT&T Technical Journal*, 69(1), January/February 1990, pp. 45-59.

[Harel82] D. HAREL, D. KOZEN, R. PARIKH, "Process logic: Expressiveness, decidability, completeness", *Journal of Computer and System Sciences*, 25(2), October 1982, pp. 145-180.

[Harel84] D. HAREL, "Dynamic Logic", *Handbook of Philosophical Logic, Volume II (Extensions of Classical Logic)*, D. Gabbay, F. Guenthner (eds.), Synthese Library, Volume 165, Reidel, Dordrecht, The Netherlands, 1984, pp. 497-604.

[Harel87] D. HAREL, "Statecharts: A visual formalism for complex systems", *Science of Computer Programming*, 8(3), June 1987, pp. 231-274.

[Harel90] D. HAREL, H. LACHOVER, A. NAAMAD, A. PNUELI, M. POLITI, R. SHERMAN, A. SHTULL-TRAURING, M. TRAKHTENBROT, "STATEMATE: A working environment for the development of complex reactive systems", *IEEE Transactions on Software Engineering*, 16(4), April 1990 , pp. 403-413.

[Harel98] D. HAREL, M. POLITI, *Modeling Reactive Systems with Statecharts: The STATEMATE Approach*, McGraw-Hill, 1998.

[Henry90] T. HENRY, W. MCCUNE, *FormEd: An X Window System application for managing first-order formulas*, Technical Memorandum ANL/MCS-TM-141, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois, USA, October 1990.

[Hoare69] C. A. R. HOARE, "An axiomatic basis for computer programming", *Communications of the ACM*, 12(10), October 1969, pp. 576-583.

[Hoare78] C. A. R. HOARE, "Communicating Sequential Processes", *Communications of the ACM*, 21(8), August 1978, pp 666-677.

[Holzma91] G. J. HOLZMANN, *Design and Validation of Computer Protocols*, Prentice Hall, Englewood Cliffs, New Jersey, USA, 1991.

[Holzma96a] G. J. HOLZMANN, "On-the-fly model checking", *ACM Computing Surveys*, 28A(4), December 1996, http://www.acm.org/surveys/1996/Formatting/.

[Holzma96b] G. J. HOLZMANN, D. PELED, M. YANNAKAKIS, "On nested depth first search", *Proceedings of the 2nd SPIN Workshop*, New Brunswick, New Jersey, USA, August 1996, appeared in *The SPIN Verification System*, DIMACS/32, American Mathematical Society, pp. 23-32.

[Holzma97] G. J. HOLZMANN, "The model checker SPIN", *IEEE Transactions on Software Engineering*, 23(5), May 1997, pp. 279-295.

[Hopcro79] J. E. HOPCROFT, J. D. ULLMAN, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, Reading, Massachusetts, USA, 1979.

[Hornos97] M. J. HORNOS, M. I. CAPEL, "Implementación de una herramienta de especificación y validación para sistemas reactivos", *Actas de las III Jornadas de Informática*, El Puerto de Santa María (Cádiz), Julio 1997, pp. 21-30.

[Hornos99] M. J. HORNOS, *Lógica de intervalos aplicada a la especificación de sistema reactivos*, Trabajo de Suficiencia Investigadora,

Departamento de Lenguajes y Sistemas Informáticos, Universidad de Granada, Septiembre 1999.

[Hornos01a] M. J. HORNOS, M. I. CAPEL, "Automata generation for on-the-fly automatic verification using formulas of an interval logic", *Proceedings of the 2nd International Conference on Application of Concurrency to System Design*, Newcastle upon Tyne, England, June 2001, IEEE Computer Society, pp. 221-230.

[Hornos01b] M. J. HORNOS, M. I. CAPEL, "Verificación automatizada eficiente de sistemas concurrentes especificando sus propiedades con fórmulas de una lógica de intervalos", *Actas de las VI Jornadas de Ingeniería del Software y Bases de Datos*, Almagro (Ciudad Real), Noviembre 2001, pp. 375-389.

[Hornos02] M. J. HORNOS, M. I. CAPEL, "On-the-fly model checking from interval logic specifications", *Proceedings of the 3rd International Workshop on Verification and Computational Logic*, Pittsburgh, Pennsylvania, USA, October 2002, Technical Report DSSE-TR-2002-5, Department of Electronics and Computer Science, University of Southampton, Southampton, United Kingdom.

[ITU99] ITU TELECOMMUNICATION STANDARDS SECTOR SG 10, *ITU-T Recommendation Z.120: Message Sequence Chart (MSC)*, ITU (International Telecommunication Union), Geneva, Switzerland, November 1999.

[Jacobs99] I. JACOBSON, G. BOOCH, J. RUMBAUGH, *The Unified Software Development Process*, Addison-Wesley, Reading, Massachusetts, USA, 1999.

[Jard89] C. JARD, T. JÉRON, "On-line model-checking for finite linear temporal logic specifications", *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, France, June 1989, Lecture Notes in Computer Science 407, Springer-Verlag, pp. 189-196.

[Jard91] C. JARD, T. JÉRON, "Bounded memory algorithms for verification on the fly", *Proceedings of the 3rd Workshop on Computer Aided Verification* (CAV'91), Aalborg, Denmark, July 1991, Lecture Notes in Computer Science 575, Springer-Verlag, pp. 192-202.

[Jensen97] K. JENSEN, *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*, Monographs in Theoretical Computer Science, Springer-Verlag, 2nd corrected printing, Berlin, Germany, 1997.

[Kaner93] C. KANER, J. FALK, H. Q. NGUYEN, *Testing Computer Software*, Van Nostrand Reinhold, New York, NY, USA, 1993.

[Kapur92] D. KAPUR, D. R. MUSSER, X. NIE, "The Tecton proof system", *Proceedings of the Workshop on Formal Methods in Databases and*

Software Engineering, Montreal, Quebec, Canada, May 1992, Workshop in Computing Series, Springer-Verlag, pp. 54-79.

[Kaufma00a] M. KAUFMANN, P. MANOLIOS, J. S. MOORE, *Computer-Aided Reasoning: An Approach*, Kluwer Academic Publishers, 2000.

[Kaufma00b] M. KAUFMANN, P. MANOLIOS, J. S. MOORE (eds.), *Computer-Aided Reasoning: ACL2 Case Studies*, Kluwer Academic Publishers, 2000.

[Kesten93] Y. KESTEN, Z. MANNA, H. MCGUIRE, A. PNUELI, "A decision algorithm for full propositional temporal logic", *Proceedings of the 5th Conference on Computer Aided Verification* (CAV'93), Elounda, Crete, Greece, June/July 1993, Lecture Notes in Computer Science 697, Springer-Verlag, pp. 97-109.

[Koomen89] J. A. KOOMEN, *The TIMELOGIC Temporal Reasoning System*, Technical Report 231, Computer Science Department, University of Rochester, Rochester, NY, USA, March 1989 (revised).

[Kozen90] D. KOZEN, J. TIURYN, "Logics of Programs", *Handbook of Theoretical Computer Science, Volume B (Formal Models and Semantics)*, J. van Leeuwen (ed.), The MIT Press, Cambridge, Massachusetts, USA, 1990, pp. 789-840.

[Kursha94] R. P. KURSHAN, *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*, Princeton University Press, Princeton, New Jersey, USA, 1994.

[Kutty93a] G. KUTTY, Y. S. RAMAKRISHNA, L. E. MOSER, L. K. DILLON, P. M. MELLIAR-SMITH, "A graphical interval logic toolset for verifying concurrent systems", *Proceedings of the 5th International Conference on Computer Aided Verification* (CAV'93), Elounda, Crete, Greece, June/July 1993, Lecture Notes in Computer Science 697, Springer-Verlag, pp. 138-153.

[Kutty93b] G. KUTTY, L. K. DILLON, L. E. MOSER, P. M. MELLIAR-SMITH, Y. S. RAMAKRISHNA, "Visual tools for temporal reasoning", *Proceedings of the IEEE Symposium on Visual Languages*, Bergen, Norway, August 1993, pp. 152-159.

[Kutty94a] G. KUTTY, *A graphical environment for temporal reasoning*, Ph.D. Thesis, Department of Electrical and Computer Engineering, University of California, Santa Barbara, USA, February 1994.

[Kutty94b] G. KUTTY, L. E. MOSER, P. M. MELLIAR-SMITH, Y. S. RAMAKRISHNA, "A graphical methodology for concurrent system design", *Proceedings of the ACM Computer Science Conference*, Phoenix, Arizona, USA, March 1994, pp. 52-59.

[Kutty95] G. KUTTY, L. E. MOSER, P. M. MELLIAR-SMITH, Y. S. RAMAKRISHNA, L. K. DILLON, "Axiomatizations of interval logics", *Fundamenta Informaticae*, 24(4), December 1995, pp. 313-332.

[Lampor83] L. LAMPORT, "What good is temporal logic?", *Proceedings of the IFIP (International Federation for Information Processing) Congress*, Paris, France, 1983, pp. 657-668.

[Lampor94] L. LAMPORT, "The temporal logic of actions", *ACM Transactions on Programming Languages and Systems*, 16(3), May 1994, pp. 872-923.

[Latval00] T. LATVALA, K. HELJANKO, "Coping with strong fairness", *Fundamenta Informaticae*, 43(1-4), 2000, pp. 175-193.

[Latval01] T. LATVALA, "Model checking LTL properties of high-level Petri nets with fairness constraints", *Proceedings of the 22nd International Conference on Application and Theory of Petri Nets*, Newcastle upon Tyne, England, June 2001, Lecture Notes in Computer Science 2075, Springer-Verlag, pp. 242-262.

[Lee96] D. LEE, M. YANNAKAKIS, "Principles and methods of testing finite state machines - a survey", *Proceedings of the IEEE*, 84(8), August 1996, pp. 1090-1123.

[Lewis97] H. R. LEWIS, CH. PAPADIMITIOU, *Elements of the Theory of Computation*, Prentice Hall, 2nd edition, Englewood Cliffs, New Jersey, USA, 1997.

[Lichte85a] O. LICHTENSTEIN, A. PNUELI, "Checking that finite-state concurrent programs satisfy their linear specification", *Proceedings of the 12th ACM Symposium on Principles of Programming Languages*, New Orleans, Louisiana, USA, January 1985, pp. 97-107.

[Lichte85b] O. LICHTENSTEIN, A. PNUELI, L. ZUCK, "The glory of the past", *Proceedings of the Conference on Logics of Programs*, Brooklyn, New York, USA, June 1985, Lecture Notes in Computer Science 193, Springer-Verlag, pp. 196-218.

[Mäkelä02] M. MÄKELÄ, "Maria: modular reachability analyser for algebraic system nets", *Proceedings of the 23rd International Conference on Application and Theory of Petri Nets*, Adelaide, Australia, June 2002, Lecture Notes in Computer Science 2360, Springer-Verlag, pp. 434-444.

[Manna82] Z. MANNA, A. PNUELI, "Verification of concurrent programs: The temporal framework", *The Correctness Problem in Computer Science*, R. S. Boyer, J. S. Moore (eds.), Academic Press, London, England, 1982, pp. 215-273.

[Manna83] Z. MANNA, A. PNUELI, "How to cook a temporal proof system for your pet language", *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*, Austin, Texas, January 1983, pp. 141-154.

[Manna87] Z. MANNA, A. PNUELI, "Specification and verification of concurrent programs by ∀-automata", *Proceedings of the Conference on Temporal Logic in Specification*, Altrincham, England, April 1987, Lecture Notes in Computer Science 398, Springer-Verlag, pp. 124-164.

[Manna89] Z. MANNA, A. PNUELI, "The anchored version of the temporal framework", *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, J. W. de Bakker, W.-P. de Roever, G. Rozenberg (eds.), Lecture Notes in Computer Science 354, Springer-Verlag, 1989, pp. 201-284.

[Manna92] Z. MANNA, A. PNUELI, *The Temporal Logic of Reactive and Concurrent Systems: Specification*, Springer-Verlag, New York, NY, USA, 1992.

[Manna94] Z. MANNA, A. ANUCHITANUKUL, N. BJORNER, A. BROWNE, E. CHANG, M. COLON, L. DE ALFARO, H. DEVARAJAN, H. SIPMA, T. URIBE, *STeP: The Stanford Temporal Prover*. Technical Report STAN-CS-TR-94-1518, Computer Science Department, Stanford University, Standford, California, USA, July 1994.

[Manna95] Z. MANNA, A. PNUELI, *Temporal Verifications of Reactive Systems: Safety*, Springer-Verlag, New York, NY, USA, 1995.

[Marca88] D. MARCA, C. L. MCGOWAN, *SADT : Structured analysis and design technique*, McGraw-Hill, New York, NY, USA, 1988.

[McMill93] K. L. MCMILLAN, *Symbolic Model Checking*, Kluwer Academic Publishers, Boston, Massachusetts, USA, 1993.

[Meinel98] C. MEINEL, T. THEOBALD, *Algorithms and Data Structures in VLSI Design*, Springer-Verlag, Berlin, Germany, 1998.

[Mellia88] P. M. MELLIAR-SMITH, "A graphical representation of interval logic", *Proceedings of the International Conference on Concurrency*, Hamburg, Germany, October 1988, Lecture Notes in Computer Science 335, Springer-Verlag, pp. 106-120.

[Mellia94] P. M. MELLIAR-SMITH, LOUISE E. MOSER, Y. S. RAMAKRISHNA, G. KUTTY, LAURA K. DILLON, "A system for automated deduction in graphical interval logic", *Proceedings of the 1st International Conference on Temporal Logic*, Bonn, Germany, July 1994, Lecture Notes in Computer Science 827, Springer-Verlag, pp. 540-542.

[Mills75] H. D. MILLS, "The new math of computer programming", *Communications of the ACM*, 18(1), January 1975, pp.43-48.

[Milner80] R. MILNER, *A Calculus of Communicating Systems*, Lecture Notes in Computer Science 92, Springer-Verlag, Berlin, Germany, 1980.

[Moser96] L. E. MOSER, P. M. MELLIAR-SMITH, Y. S. RAMAKRISHNA, G. KUTTY, L. K. DILLON, "The real-time graphical interval logic toolset", *Proceedings of the 8th Conference on Computer Aided Verification* (CAV'96), New Brunswick, New Jersey, USA, July/August 1996, Lecture Notes in Computer Science 1102, pp. 446-449.

[Moser97] L. E. MOSER, Y. S. RAMAKRISHNA, G. KUTTY, P. M. MELLIAR-SMITH, L. K. DILLON, "A graphical environment for the design of concurrent real-time systems", *ACM Transactions on Software Engineering Methodology*, 6(1), January 1997, pp. 31-79.

[Murata84] T. MURATA, "Modeling and analysis of concurrent systems", *Handbook of Software Engineering*, Chapter 3, C. R. Vick, C. V. Ramamoorthy, (eds.), Van Nostrand Reinhold, New York, NY, USA, 1984, pp. 39-63.

[Murata89] T. MURATA, "Petri Nets: Properties, analysis and applications", *Proceedings of the IEEE*, 77(4), April 1989, pp. 541-580.

[Myers79] G. J. MYERS, *The Art of Software Testing*, Wiley-Interscience, New York, NY, USA, 1979.

[Myers86] B. A. MYERS, "Visual programming, programming by example, and program visualization: A taxonomy", *Proceedings of the Conference on Human Factors in Computing Systems* (CHI'86), Boston, Massachusetts, USA, April 1986, pp. 59-66.

[Namjos00] K. S. NAMJOSHI, R. P. KURSHAN, "Syntactic program transformations for automatic abstraction", *Proceedings of the 12th International Conference on Computer Aided Verification* (CAV'2000), Chicago, Illinois, USA, July 2000, Lecture Notes in Computer Science 1855, Springer-Verlag, pp. 435-449.

[Nascim94] M. A. NASCIMENTO, M. M. TANIK, *Towards Zero Defect Software: The Cleanroom Approach*, Technical Report 94-CSE-31, Department of Computer Science and Engineering, Southern Methodist University, Dallas, Texas, USA, August 1994.

[Naur60] P. NAUR (ed.), "Revised Report on the Algorithmic Language ALGOL 60", *Communications of the ACM*, 3(5), May 1960, pp. 299-314.

[New90] D. NEW, P. AMER, "Protocol visualization of Estelle specifications", *Proceedings of the IFIP TC6/WG6.1 3rd International Conference on Formal Description Techniques for Distributed Systems and*

Communications Protocols (FORTE'90), Madrid, Spain, November 1990, North-Holland, pp. 551-554.

[Nipkow02] T. NIPKOW, L. C. PAULSON, M. WENZEL, *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, Lecture Notes in Computer Science 2283, Springer-Verlag, Berlin, Germany, 2002.

[O'Mall96] T. O. O'Malley, D. J. Richardson and L. K. Dillon, "Efficient specification-based oracles for critical systems", *Proceedings of 1996 California Software Symposium*, Los Angeles, California, USA, April 1996, pp. 50-59.

[Ostrof89] J. S. OSTROFF, *Temporal logic for real-time systems*, Advanced Software Development Series, Research Studies Press, John Wiley and Sons, Taunton, England, 1989.

[Owre92] S. OWRE, J. M. RUSHBY, N. SHANKAR, "PVS: A Prototype Verification System", *Proceedings of the 11th Conference on Automated Deduction*, Saratoga, New York, USA, June 1992, Lecture Notes in Artificial Intelligence 607, Springer-Verlag, pp. 748-752.

[Papadi94] CH. PAPADIMITIOU, *Computational Complexity*, Addison-Wesley, Reading, Massachusetts, USA, 1994.

[Patton00] R. PATTON, *Software Testing*, Sams, 2000.

[Peled01a] D. A. PELED, *Software Reliability Methods*, Springer-Verlag, New York, NY, USA, 2001.

[Peled01b] D. PELED, L. D. ZUCK, "From model checking to a temporal proof", *Proceedings of the 8th International* SPIN Workshop on *Model Checking of Software*, Toronto, Canada, May 2001, Lecture Notes in Computer Science 2057, Springer-Verlag, pp. 1-14.

[Peled93] D. PELED, "All from one, one for all: on model checking using representatives", *Proceedings of the 5th International Conference on Computer Aided Verification* (CAV'93), Elounda, Crete, Greece, June/July 1993, Lecture Notes in Computer Science 697, Springer-Verlag, pp. 409-423.

[Peled99] D. PELED, M. Y. VARDI, M. YANNAKAKIS, "Black Box Checking", *Proceedings of the 12th International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols* (FORTE'99), Beijing, China, October 1999, Kluwer Academic Publishers, pp. 225-240.

[Petri65] C. A. PETRI, *Communication with automata*, Technical Report RADC-TR-65-377, Volume 1, Supplement 1, Rome Air Development Center, Griffis Air Force Base, Rome, New York, USA, 1965.

[Pinter84] S. S. PINTER, P. WOLPER, "A temporal logic for reasoning about partially ordered computations", *Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing*, Vancouver, British Columbia, Canada, August 1984, pp. 28-37.

[Plaist83] D. A. PLAISTED, "A low-level language for obtaining decision procedures for clases of temporal logics", *Proceedings of the CMU Workshop on Logics of Programs*, Pittsburgh, Pennsylvania, USA, June 1983, Lecture Notes in Computer Science 164, Springer-Verlag, pp. 403-420.

[Pnueli77] A. PNUELI, "The temporal logics of programs", *Proceedings of the 18th IEEE Symposium on Foundation of Computer Science*, Providence, Rhode Island, USA, November/October 1977, pp. 46-57.

[Pnueli86a] A. PNUELI, "Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends", *Current Trends in Concurrency*, J. W. de Bakker, W.-P. de Roever, G. Rozenberg (eds.), Lecture Notes in Computer Science 224, Springer-Verlag, 1986, Berlin, Germany, pp. 510-584.

[Pnueli86b] A. PNUELI, "Specification and development of reactive systems", *Proceedings of the Conference on Information Processing* (IFIP'86), Dublin, Ireland, North-Holland, 1986, pp. 845-858.

[Pong83] M. C. PONG, N. NG, "PIGS: A system for programming with interactive graphical support", *Software: Practice and Experience*, 13(9), September 1983 , pp. 847-855.

[Pratt79] V. R. PRATT, "Process logic", *Proceedings of the 6th Annual Symposium on the Principles of Programming Languages*, San Antonio, Texas, USA, January 1979, pp. 93-100.

[Quiell82] J. P. QUIELLE, J. SIFAKIS, "Specification and verification of concurrent systems in CESAR", *Proceedings of the 5th International Symposium on Programming*, Turin, Italy, April 1982, Lecture Notes in Computer Science 137, Springer-Verlag, pp. 337-351.

[Ramakr92] Y. S. RAMAKRISHNA, L. K. DILLON, L. E. MOSER, P. M. MELLIAR-SMITH, G. KUTTY, "An automata-theoretic decision procedure for future interval logic", *Proceedings of the 12th Conference on Foundations of Software Technology and Theoretical Computer Science*, New Delhi, India, December 1992, Lecture Notes in Computer Science 652, Springer-Verlag, pp. 51-67.

[Ramakr93a] Y. S. RAMAKRISHNA, *Interval logics for temporal specification and verification*, Ph.D. Thesis, Department of Electrical and Computer Engineering, University of California, Santa Barbara, USA, August 1993.

[Ramakr93b] Y. S. RAMAKRISHNA, L. K. DILLON, L. E. MOSER, P. M. MELLIAR-SMITH, G. KUTTY, "A Real-Time Interval Logic and its Decision Procedure", *Proceedings of the 13th Conference on Foundations of Software Technology and Theoretical Computer Science*, Bombay, India, December 1993, Lecture Notes in Computer Science 761, Springer-Verlag, pp. 173-192.

[Ramakr96a] Y. S. RAMAKRISHNA, P. M. MELLIAR-SMITH, L. E. MOSER, L. K. DILLON, G. KUTTY, "Interval logics and their decision procedures. Part I: An interval logic", *Theoretical Computer Science*, 166(1-2), October 1996, pp. 1-47.

[Ramakr96b] Y. S. RAMAKRISHNA, P. M. MELLIAR-SMITH, L. E. MOSER, L. K. DILLON, G. KUTTY, "Interval logics and their decision procedures. Part II: A real-time interval logic", *Theoretical Computer Science*, 170(1-2), December 1996, pp. 1-46.

[Rosner86] R. ROSNER, A. PNUELI, "A choppy logic", *Proceedings of the 1st IEEE Symposium on Logic in Computer Science*, Cambridge, Massachusetts, USA, June 1986, pp. 306-313.

[Rumbau96] J. RUMBAUGH, M. BLAHA, W. PREMERLANI, F. EDDY, W. LORENSEN, *Modelado y diseño orientados a objetos. Metodología OMT*, J. R. García-Bermejo Giner (tr.), Prentice Hall, Madrid, España, 1996.

[Rumbau99] J. RUMBAUGH, I. JACOBSON, G. BOOCH, *The Unified Modeling Language Reference Manual*, Addison-Wesley, Reading, Massachusetts, USA, 1999.

[Schlör93] R. SCHLÖR, W. DAMM, "Specification and verification of system-level hardware designs using timing diagrams", *Proceedings of the European Conference on Design Automation with the European Event in ASIC Design*, Paris, France, February 1993, pp. 518-524.

[Schnei97] F. B. SCHNEIDER, *On Concurrent Programming*, Springer-Verlag, New York, NY, USA, 1997.

[Schwar83] R. L. SCHWARTZ, P. M. MELLIAR-SMITH, F. VOGT, "An interval based temporal logic", *Proceedings of the 2nd Annual ACM Symposium on Principles of Distributed Computing*, Montreal, Canada, August 1983, pp. 173-186.

[Selic94] B. SELIC, G. GULLEKSON, P. T. WARD, *Real-Time Object-Oriented Modeling*, John Wiley and Sons, New York, NY, USA, 1994.

[Sharyg01] N. SHARYGINA, D. PELED, "A Combined Testing and Verification Approach for Software Reliability", *Proceedings of the International Symposium of Formal Methods Europe*, Berlin, Germany, March 2001, Lecture Notes in Computer Science 2021, Springer-Verlag, pp. 611-628.

[Shu88] N. C. SHU, *Visual programming*, Van Nostrand Reinhold, New York, NY, USA, 1988.

[Sipser96] M. SIPSER, *Introduction to the Theory of Computation*, PWS, Boston, Massachusetts, USA, 1996.

[Sistla85] A. P. SISTLA, E. M. CLARKE, "The complexity of propositional linear temporal logics", *Journal of the ACM*, 32(3), (1985), pp. 733-749.

[Sistla87] A. P. SISTLA, M. Y. VARDI, P. WOLPER, "The complementation problem for Büchi automata with applications to temporal logic", *Theoretical Computer Science*, 49(2-3), July 1987, pp. 217-237.

[Somenz00] F. SOMENZI, R. BLOEM, "Efficient Büchi automata from LTL formulae", *Proceedings of the 12th International Conference on Computer Aided Verification* (CAV'2000), Chicago, Illinois, USA, July 2000, Lecture Notes in Computer Science 1855, Springer-Verlag, pp. 248-263.

[Stavel99] A. M. STAVELY, *Towards Zero-Defect Programming*, Addison-Wesley, 1999.

[Stirli88] C. STIRLING, "Temporal Logics for CCS", *Proceedings of the REX Workshop on Linear Time, Branching Time and Partial Order Logics and Models of Concurrency*, Noordwijkerhout, The Netherlands, May/June 1988, Lecture Notes in Computer Science 354, Springer-Verlag, pp 660-672.

[Strous93] B. STROUSTRUP, *El Lenguaje de Programación C++*, R. Escalona (tr.), Addison-Wesley Iberoamericana, Wilmington, Delaware, USA, 1993.

[Tarjan72] R. E. TARJAN, "Depth first search and linear graph algorithms", *SIAM Journal of Computing*, 1, 1972, pp. 146-160.

[Théry92] L. THÉRY, Y. BERTOT, G. KAHN, "Real theorem provers deserve real user interfaces", *Proceedings of the 5th ACM SIGSOFT Symposium on Software Development Environments*, Tyson's Corner, Virginia, USA, December 1992, pp. 120-129.

[Thiaga86] P. S. THIAGARAJAN, "Elementary net systems", *Proceedings of Advances in Petri Nets*, Bad Honnef, Germany, September 1986, Lecture Notes in Computer Science 254, Springer-Verlag, pp. 26-59.

[Thomas90] W. THOMAS, "Automata on infinite objects", *Handbook of Theoretical Computer Science, Volume B (Formal Models and Semantics)*, Chapter 4, J. van Leeuwen (ed.), The MIT Press, Cambridge, Massachusetts, USA, 1990, pp. 133-191.

[Thomas97] W. THOMAS, "Lenguages, automata, and logic", *Handbook of Formal Language Theory,* Volume 3, G. Rozenberg, A. Salomaa (eds.), Springer-Verlag, Berlin, Germany, 1997, pp. 389-455.

[Valmar90] A. VALMARI, "A stubborn attack on state explosion", *Proceedings of the 2nd Workshop on Computer Aided Verification* (CAV'90), New Brunswick, New Jersey, USA, June 1990, Lecture Notes in Computer Science 531, Springer-Verlag, pp. 156-165.

[Vardi86] M. Y. VARDI, P. WOLPER, "An automata-theoretic approach to automatic program verification", *Proceedings of the 1st IEEE Symposium on Logic in Computer Science*, Cambridge, Massachusetts, USA, June 1986, pp. 322-331.

[Vardi96] M. Y. VARDI, "An Automata-Theoretic Approach to Linear Temporal Logic", *Logics for Concurrency: Structure versus Automata*, F. Moller, G. M. Birtwistle (eds.), Lecture Notes in Computer Science 1043, Springer-Verlag, New York, NY, USA, 1996, pp. 238-266.

[Venema94] Y. VENEMA, "Completeness through flatness in two-dimensional temporal logic", *Proceedings of the 1st International Conference on Temporal Logic*, Bonn, Germany, July 1994, Lecture Notes in Computer Science 827, Springer-Verlag, pp. 149-164.

[VIS96] THE VIS GROUP, "VIS: A system for Verification and Synthesis", *Proceedings of the 8th International Conference on Computer Aided Verification* (CAV'96), New Brunswick, New Jersey, USA, July/August 1996, Lecture Notes in Computer Science 1102, Springer-Verlag, pp. 428-432.

[Wolper83] P. WOLPER, M. Y. VARDI, A. P. SISTLA, "Reasoning about infinite computations paths", *Proceedings of the 24th IEEE Symposium on Foundation of Computer Science*, Tucson, Arizona, USA, November 1983, pp. 185-194.

[Wolper85] P. WOLPER, "The tableau method for temporal logic: An overview", *Logique et Analyse*, 110-111, June-September 1985, pp. 119-136.

[Wolper87] P. WOLPER, "On the relation of programs and computations to models of temporal logic", *Proceedings of the Conference on Temporal Logic in Specification*, Altrincham, England, April 1987, Lecture Notes in Computer Science 398, pp. 75-123.

[Wolper95] P. WOLPER, "An introduction to model checking", *Proceedings of the Software Quality Week*, San Francisco, California, USA, May 1995, http://www.montefiore.ulg.ac.be/~pw/papers/psfiles/Wol95b.ps.

[Yau88] S. S. YAU, X. JIA, "Visual languages and software specifications", *Proceedings of the IEEE International Conference on Computer Languages*, Miami Beach, Florida, USA, October 1988, pp. 322-328.

[Yourdo89] E. YOURDON, *Modern structured analysis*, Yourdon Press, Englewood Cliffs, New Jersey, USA, 1989.