UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

POSGRADO EN CIENCIA E INGENIERÍA EN COMPUTACIÓN

Un sencillo y eficiente verificador al vuelo para la lógica temporal CTL* escrito en Haskell

---

TESIS

QUE PARA OPTAR POR EL GRADO DE
MAESTRO EN CIENCIA E INGENIERÍA EN COMPUTACIÓN

Presenta:
Cenobio Moisés Vázquez Reyes

Tutor:
Dr. David Arturo Rosenblueth Laguette

Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas
Ciudad de México Noviembre del 2018

To me, mathematics, computer science, and the arts are insanely related. They're all creative expressions.
Sebastian Thrun

Proof is the idol before whom the pure mathematician tortures himself.
Arthur Eddington

A main challenge for the field of computer science is to provide formalisms, techniques, and tools that will enable the efficient design of correct and well-functioning systems despite their complexity.
Kim Guldstrand Larsen

Índice general

Nunca olvidaré el primer día que llegué a la universidad. Llegué al paradero de pumabuses del metro Universidad y pregunté cuál de todos los camiones me llevaría a la Facultad de Ciencias. Ese, me dijo un señor, así que subí al camión y pedí al que estaba sentado a mi lado que me avisara cuando llegáramos a la facultad, por fortuna él también se dirigía a Ciencias. Cuando llegué, vi a la facultad distinta a la fotografía que vi en internet, supuse que era una foto desde otra fachada y decidí entrar de todas maneras. No me quería perder la bienvenida y realmente me emocionaba saber que ya era un universitario. La ceremonia de bienvenida comenzó y el discurso hablaba de la situación del país, de la terrible forma de gobierno y cosas que en nada tenían que ver con computación.

—Oye—le dije al de a lado—,¿estamos en Ciencias?
—Sí.
—¿Pero aquí dan la bienvenida a los de "compu"?
—¿"Compu"?
—Sí, Ciencias de la Computación.
—No, aquí es Ciencias Políticas.

Salí corriendo de ese lugar. No conocía Ciudad Universitaria y mucho menos sabía si el lugar a donde tenía que ir estaba lejos o cerca de donde me encontraba. Seguí corriendo guiado únicamente por mi corazón y por mis ganas de llegar a mi facultad. Lo crean o no, aparecí frente a la facultad y llegué a tiempo a mi bienvenida. Todo esto ocurrió en el año 2008.

Así como encontré el camino a Ciencias el primer día que llegué a esta universidad, lo crean o no, fue como encontré a mis profesores Favio y Lulú, guiado por mi instinto y por mi corazón. Estoy convencido de que mis deseos de encontrar gente con ganas de enseñar y apasionados por lo que estudian trazaron una hebra en la enorme red de la vida y el destino. Esa hebra me llevó a ellos, y no sólo encontré profesores valiosos sino seres humanos a quienes quiero con todo mi corazón. Profesor, Lulú, gracias.

Al llegar al posgrado volví a trazar una hebra en la red de la vida y el destino. Recuerdo que yo quería entrar en 2016 a la maestría, pero no se pudo. Entré al año siguiente y descubrí que si hubiera entrado cuando yo lo deseaba no habría podido conocer al Dr David pues él estaba fuera del país. ¿Coincidencia? ¿Destino? No lo sé. Me gusta pensar que mis ganas de encontrar a alguien con quien trabajar y que me hiciera conocer nuevos horizontes fueron lo que hizo que las cosas se alinearan.

Siempre que recuerdo la primera vez que me reuní con el Dr David para hablar de matemáticas confirmo que lo correcto fue haber entrado al posgrado en 2017 y no en 2016. Sus primeras palabras en esa reunión fueron «Moisés, las matemáticas son para los ángeles. Nosotros hacemos nuestro mejor intento». A la fecha, esas palabras me hacen vibrar y me animan a seguir esforzándome en lo que hago. Gracias, profesor, gracias por enseñarme el camino de los ángeles. Lo admiro y le quiero mucho.

Quiero agradecer al profesor Francisco H. Quiroz por su curso de autómatas y por sus valiosos comentarios para que este trabajo tuviera un mejor nivel. Y quiero dar un agradecimiento muy especial al profesor Miguel Carillo porque sin sus valiosos consejos y observaciones no se habrían podido encontrar tantos resultados. En serio, gracias.

Ahora quiero agradecer a mis padres y a mis hermanitas por amarme y cuidarme siempre. Ustedes son mi vida, y por ustedes daría mi vida sin dudar. Gracias por su ejemplo y por su convivencia.

Gracias a todos mis amigos, siento que no necesito nombrarlos porque tengo la fortuna de que sean muchos y porque me esfuerzo en manifestar mi aprecio por la gente que me quiere. Ustedes también son familia, gracias por estar conmigo siempre.

Gracias a mis alumnos, por ustedes he aprendido el triple. Los llevo en el corazón y son mi más grande orgullo. Les deseo lo mejor siempre y espero sepan que siempre cuentan conmigo.

Ya les conté como encontré el camino a la facultad y que de esa misma manera encontré a mis maestros. Creo que de la misma manera encontré el amor. Siempre quise encontrar a alguien con quien cada momento fuera mágico y con quien cada beso fuera alcanzar el infinito. Así te encontré, Karen. Y creo que así me encontraste. Soy un maldito suertudo.

19 de octubre del 2018, Ciudad de México.

Resumen

En este trabajo se muestra un algoritmo para hacer verificación de modelos LTL que no hace uso de autómatas de Büchi. Dicho algoritmo es conceptualmente sencillo, se puede extender a un algoritmo para hacer verificación de modelos CTL* y es correcto al obtenerse directamente de la semántica formal. Además, se da una implementación de los algoritmos antes mencionado en el lenguaje de programación Haskell.

Capítulo 1 *¿Qué es la verificación de modelos?*

Se da una breve introducción a la verificación de modelos y se muestra su importancia dentro de la industria.

Capítulo 2 *Lógicas temporales*

Se definen formalmente la sintaxis y la semántica formal de las lógicas LTL, CTL y CTL*. También se muestran algunas equivalencias que ayudan a que los verificadores propuestos en este trabajo sean más eficientes.

Capítulo 3 *Un eficiente verificador para CTL**

Se muestra el algoritmo para hacer verificación de modelos LTL propuesto por Girish Bhat, Rance Cleaveland y Orna Grumberg [BCG95]. Este algoritmo utiliza la semántica formal de la lógica temporal LTL y el algoritmo de Tarjan para detectar componentes fuertemente conexas. Dicho algoritmo se hace conceptualmente más sencillo y se obtiene un nuevo algoritmo para hacer verificación de modelos LTL que está basado únicamente en la semántica formal y que no hace uso del algoritmo de Tarjan.

Capítulo 4 *Cómodo y eficiente: Haskell*

Se presenta de forma breve el lenguaje de programación puramente funcional Haskell. También se mencionan algunas de las ventajas que brinda este lenguaje para implementar los algoritmos propuestos en el capítulo anterior.

Capítulo 5 *Implementando los verificadores de modelos mcALTL y mcCTL**

Finalmente, se muestra una implementación de los algoritmos descritos en el capítulo 3 que es correcta al obtenerse directamente de la semántica formal. Adicionalmente se muestran resultados experimentales sobre sus desempeños en comparación con el verificador de modelos NuSMV.

Parte I

Verificación de modelos

Capítulo 1

¿Qué es la verificación de modelos?

Nuestra calidad de vida depende de la calidad de nuestro software.
Hanna Jadwiga Oktaba

I was quite skeptical about the scalability of hand constructed proofs.
Edmund M. Clarke

Working programmers may devote more than half of their time on testing and debugging in order to increase reliability .
E. Allen Emerson

Cuando realizamos un trabajo con la computadora pensamos que será un trabajo sin errores. Usamos las computadoras para casi todo, lo que ha hecho impensable un mundo sin estos dispositivos. Ya sea portátil, para la oficina, o para el hogar, una computadora es un dispositivo indispensable en nuestras vidas. En la actualidad, prácticamente todas las transacciones bancarias o los enormes cálculos de población se calculan completamente por computadora; también varios procedimientos de construcción y de medicina son asistidos por computadoras. Las computadoras también ayudan en tareas como procesamiento de imágenes, reconocimiento facial en tiempo real, complejas animaciones 3D, predicción del clima, y más. Las computadoras se han convertido en una herramienta que nunca abandonará a la humanidad, y esto hace indispensable garantizar que no fallen en el trabajo que realizan. ¿Cómo garantizamos que una computadora no cometerá errores? Son artefactos cuyos componentes son sumamente complicados, y cuyo funcionamiento en conjunto no es sencillo de analizar. ¿Cómo sabemos si un programa introducido a una computadora es correcto? El código de un programa puede ser tan corto como un haiku[1] o tan largo como *El Quijote*. Los sistemas computacionales están cambiando y creciendo constantemente, lo que hace que sea cada vez más complicado garantizar su buen comportamiento. Incluso si una computadora está compuesta por hardware que no tiene errores de fabricación estos componentes podrían tener errores en su diseño. Por otra parte, el software instalado en una computadora está escrito por personas que pudieron haber cometido un error de programación. Muchas veces estos errores son diminutos y se *parchan* con una actualización, otras veces son errores que pueden costar millones de dólares , o en el peor de los casos, pueden costar vidas:

- [Wik18] En 1994, el profesor Thomas Nicely detectó un error en los procesadores Intel Pentium al momento de realizar la operación de división con punto flotante. En un inicio, Intel negó la existencia del error. Al poco tiempo, IBM se unió a la demanda colectiva de parte de los usuarios para que Intel reemplazara todas

[1] Poema japonés de diecisiete sílabas.

las unidades que presentaban el error y la compañía terminó reemplazando cada unidad afectada. Ésto significó una pérdida millonaria para Intel.

- [Arn17] En 1996, el cohete Ariane-5, lanzado por la Agencia Espacial Europea, explotó cuarenta segundos después de su despegue debido a un error en la conversión de números de punto flotante de 64 bits a enteros con signo de 16 bits. El proyecto había costado una década de esfuerzo y $7 mil millones de dólares.

- [Lev+95] Entre 1985 y 1987, la máquina de radioterapia Therac-25, que era controlada por computadora, mató a tres personas al aplicarles sobredosis de radiación. Las investigaciones determinaron que la causa del mal funcionamiento de la máquina fueron malas prácticas en el desarrollo del software de la máquina.

- [Koc+18] Recientemente se encontraron errores de seguridad en los procesadores Intel, AMD y ARM. Estos errores, llamados *Meltdown* y *Spectre*, son consecuencia de errores en el diseño de los procesadores.

Verificar un sistema computacional a mano no es práctico; podría ser un trabajo tardado y elaborado, y podría acarrear muchos errores incluso si este análisis lo hace un experto. Además, este proceso usualmente lo entiende únicamente quien lo lleva a cabo, y esto no siempre logra agilizar la corrección de los errores encontrados. Floyd [Flo67] y Hoare [Hoa69] establecieron sistemas axiomáticos para la verificación de programas secuenciales, aunque su trabajo tuvo un enorme impacto no tuvieron mucho éxito en la práctica por estar orientados hacia la construcción de pruebas a mano.

En la actualidad muchos equipos de desarrollo hacen *testing* para asegurar que sus programas no fallen. Esta técnica consiste en correr el sistema en casos particulares, llamados *pruebas*, y observar los resultados de la ejecución. Si alguna *prueba* falla, ésta sugiere dónde podría encontrarse un error. El *testing* es una práctica que hacen casi todos los equipos de desarrollo pero no necesariamente garantiza que un sistema no va a fallar, pues incluso si se prueba con muchos casos particulares siempre existe la posibilidad de que se escape un escenario en el cual el sistema falle (encontrar un *bug* en un sistema concurrente haciendo *testing* es una tarea sumamente difícil).

La *verificación de modelos* permite verificar de manera formal la correctud de un sistema computacional en automático. Esta técnica fue desarrollada de manera independiente por Edmund M. Clarke y E. Allen Emerson [EC80]; [CE81]; [CES86] y por J. P. Queille y J. Sifakis [QS82] en la década de los ochenta. En 2007, Clarke, Emerson y Sifakis obtuvieron el premio Turing por haber sido los pioneros de esta área de las Ciencias de la Computación. Los fundamentos de la verificación de modelos vienen de la lógica matemática, haciendo este método totalmente confiable pues sus técnicas de verificación se obtienen de teoremas y algoritmos cuya correctud ya fue demostrada.

El esquema a seguir para hacer verificación de modelos es sencillo:

1. El sistema que se va a verificar se formaliza con una *estructura de Kripke*.

2. La propiedad que el sistema debe cumplir se expresa con una *fórmula*.

3. La estructura de Kripke y la fórmula se pasan como argumentos a un verificador.

4. Cuando el verificador termina nos dice si la estructura cumple, o no, con la propiedad especificada.

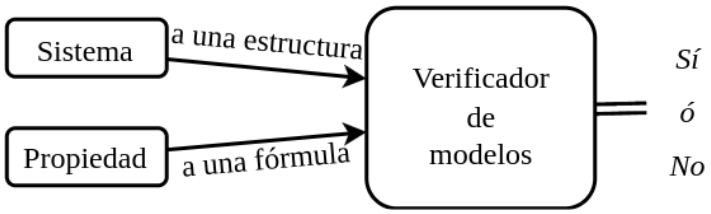


FIGURA 1.1: Esquema de la verificación de modelos

De manera precisa, el problema de verificación de modelos se define de la siguiente manera:

> *Dada una estructura de Kripke M, una fórmula φ escrita en alguna lógica temporal, y un estado s de M, decidir si $(M, s) \vDash \varphi$.*

De manera intuitiva, una *estructura de Kripke* es un sistema de transiciones con un número finito de estados y usualmente se representa con una gráfica dirigida. Los estados de la estructura de Kripke están *etiquetados* con variables proposicionales y la relación de accesibilidad entre estados debe ser *total*[2] en el siguiente sentido: cada estado debe tener al menos un sucesor. Las fórmulas están escritas en alguna *lógica temporal* y describen propiedades que puede cumplir una estructura de Kripke a lo largo del tiempo.

Para dar una idea de cómo se hace la verificación podemos pensar en propiedades acerca de la estructura mostrada en la figura 1.2 y verificarlas de manera intuitiva:

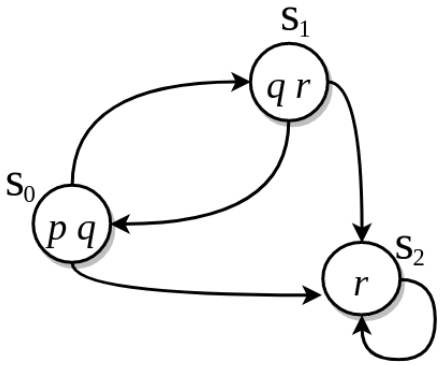


FIGURA 1.2

- *En el estado s_1 se cumple $p \vee q$*: esta propiedad se cumple pues el estado s_1 está etiquetado con q.
- *Desde el estado s_0, sin importar cuál trayectoria se elija, siempre se alcanza un estado que cumple r*: esta propiedad se cumple pues los sucesores del estado s_0 son $\{s_1, s_2\}$ y ambos están etiquetados con r.
- *Hay una trayectoria que parte del estado s_1 en el que en todo momento se cumple q*: esta propiedad se cumple pues la trayectoria $\{s_1, s_0, s_1, s_0, \dots\}$ siempre cumple q.

[2]En verificación de modelos, la palabra *total* no tiene el significado usual al hablar de una relación.

- *En todas las trayectorias que salen del estado s_0 eventualmente se cumple $p \wedge q$*: esta propiedad no se cumple pues la trayectoria $\{s_0, s_2, s_2, \dots\}$ no cumple $p \wedge q$ a partir del estado s_2.

- *Cualquier trayectoria que empiece en s_1 que eventualmente cumpla p, eventualmente cumplirá r*: esta propiedad es cierta pues la trayectoria que empieza en $\{s_1, s_0\}$ cumple p en s_0, y esa misma trayectoria cumple r desde s_1; por otra parte, la trayectoria $\{s_1, s_2, s_2, s_2, \dots\}$ cumple la propiedad, pues el antecedente *eventualmente cumplir p* es falso, lo que hace que cumpla la propiedad de manera trivial.

A grandes rasgos, hacer verificación de modelos es hacer una exploración exhaustiva en los estados de una estructura de Kripke para corroborar que dicha estructura cumple con una fórmula dada. Las lógicas temporales son decidibles, esto permite que existan algoritmos para realizar la verificación de forma automática. Usualmente, las lógicas temporales utilizadas en verificación de modelos son LTL (*Linear Time Logic*) y CTL (*Computational Tree Logic*). Estas lógicas temporales extienden a la lógica proposicional con operadores de tiempo y con cuantificadores de trayectorias (ambas lógicas se presentan en el capítulo 2).

La verificación de modelos ha ganado popularidad por brindar muchas ventajas:

- **Sin demostraciones a mano:** Todo el proceso se realiza completamente en automático. En palabras de E. Clarke [CE81]:

> *"The task of proof construction is in general quite tedious and a good deal of ingenuity may be required to organize the proof in a manageable fashion.We argue that proof construction is unnecessary in the case of finite state concurrent systems and can be replaced by a model-theoretic approach which will mechanically determine if the system meets a specification expressed in propositional temporal logic. The global state graph of the concurrent systems can be viewed as a finite Kripke structure and an efficient algorithm can be given to determine whether a structure is a model of a particular formula (i.e. to determine if the program meets its specification)."*

- **Contraejemplos:** Si una estructura de Kripke no cumple una especificación, es posible generar una traza de ejecución y un contraejemplo que es útil a la hora de *debuguear*. Varios equipos de desarrollo utilizan la verificación de modelos sólo por ésto.

- **Prevención:** No es necesario especificar por completo el sistema que se está revisando ya que es posible verificarlo mientras se está desarrollando. Esto permite encontrar errores antes de que el sistema esté completamente terminado.

- **Expresividad:** Las lógicas temporales permiten expresar con facilidad propiedades acerca de sistemas concurrentes (exclusión mutua, ausencia de *deadlocks*, entre otras) para verificarlas en automático. Esto es importante porque intentar verificar a mano estas propiedades es difícil.

Algunas posibles desventajas de esta técnica son las siguientes:

- **Escribir especificaciones es difícil:** Aunque es complicado escribir especificaciones, es preferible escribir una fórmula que buscar la causa y la solución de un *bug* en un diseño con miles de estados o en un programa con miles de líneas de código.
- **Explosión de estados:** Esta es la principal desventaja en la verificación de modelos: el número de estados en un sistema concurrente puede ser enorme e intentar manejarlos de forma explícita puede crear un espacio de búsqueda en el que la verificación se vuelve prácticamente imposible de realizar. Todos los verificadores sufren esta desventaja, pero ha habido mucho progreso en los últimos años para contrarrestarlo.

Existen muchas herramientas para hacer verificación de modelos y a continuación se mencionan algunas:

- **Verificadores lógico-temporales**: reciben una estructura de Kripke que representa el sistema a analizar y las propiedades a verificar son fórmulas escritas en alguna lógica temporal.

    - Los primeros verificadores lógico temporales fueron EMC (*Extended Model Checker*) [CES86] y CÆSAR [QS82]. El primero fue implementado por Clarke y verifica fórmulas escritas en lógica temporal CTL. El segundo se ha convertido en CADP (*Construction and Analysis of Distributed Processes* o *CAESAR/ALDEBARAN Development Package*) y es utilizado en el diseño de sistemas concurrentes asíncronos [Cad].
    - En 1992, Kenneth L. McMillan presentó en su tesis doctoral la verificación simbólica de modelos por medio del verificador *SMV* [McM93], permitiendo verificar modelos de tamaño mucho más grandes y atacando directamente el problema de explosión de estados. Esta técnica utiliza una estructura llamada OBDD que permite representar de manera eficiente conjuntos de estados dentro de un modelo. Gracias a la verificación simbólica se han verificado sistemas incluso con 10^{20} estados[Bur+92]. SMV evolucionó a NuSMV [Nus] y luego a nuXmv [Nux]. Este verificador permite verificar fórmulas LTL y CTL y es capaz de analizar sistemas con una cantidad infinita de estados.
    - Spin es una herramienta desarrollada por Laboratorios Bell. Esta herramienta realiza la verificación de modelos *al vuelo* [3], es decir, no necesita contruir todo el espacio de búsqueda antes de la verificación, sino que lo construye sobre la marcha. [Ger+95].
    - Verus [Ver] y Kronos [Kro] verifican sistemas de tiempo real.
    - Hytech es una herramienta para verificar sistemas integrados [Hyt].
    - Murφ es una herramienta que originalmente fue desarrollada en Stanford y que es ampliamente utilizada en la verificación de protocolos de coherencia de caché [Mur] [Dil+92].

[3] Del inglés *on the fly*.

- **Verificadores *behavior conformance***: reciben tanto el modelo como las propiedades a analizar en forma de autómata. Ejemplos son Cospan/FormatCheck [HK90], FDR [Ros94] y Concurrency Workbench [Con].

La verificación de modelos ha tenido muchas contribuciones a la verificación formal de hardware, por mencionar algunos ejemplos:

- Bud Mishra (alumno de Clarke) fue el primero en utilizar técnicas de verificación de modelos para verificar hardware de manera formal. Mishra utilizó el verificador *EMC* para encontrar un *bug* en la *cola Sietz* del libro de Mead & Conway *Introduction to VLSI Systems* [MC85].

- David Dill y Mike Browne se unieron al equipo Clarke-Mishra para hacer verificación de hardware y realizaron trabajos en conjunto [BCD85] y [Bro+86].

- Clarke, Burch, Grumberg, Long y McMillan mostraron técnicas para verificar circuitos secuenciales como un contador síncrono, un circuito de *data pipeline*, y la consistencia del protocolo de caché del multiprocesador Encore Gigamax [Cla+92].

- Clarke y sus alumnos utilizaron SMV para verificar el protocolo de coherencia de caché del *Futurebus+* [Cla+93].

- En Stanford, David L. Hill y sus alumnos usaron el verificador *Murφ* para verificar el protocolo de coherencia de caché de la *Scalable Coherent Interface* [DR96].

- Investigadores de Bull y Verimag utilizaron *LOTOS (Languaje Of Temporal Ordering Specification)* para describir el funcionamiento de la arquitectura del multiprocesador PowerScale. Con las herramientas *CÆSAR/ALDÉBARAN* se verificaron cuatro propiedades que garantizaban la correctud del multiprocesador en cuestión de minutos [Che+96].

- Johan Bengtsson y otros colaboradores utilizaron el verificador de modelos *UppAal* para verificar protocolos de audio en componentes de reproductores Philips. Los protocolos ya habían sido verificados a mano, pero Bengtsson y compañía los verificaron completamente en automático [Ben+96].

- Laboratorios Bell se ofreció a verificar un controlador ideado por AT&T que supuestamente no tenía errores en su diseño. La herramienta que se utilizó para verificar el controlador fue *FormalCheck*. Luego de cinco horas, se verificaron seis propiedades y únicamente se cumplieron cinco. Una vez detectado el error, fue corregido en minutos y posteriormente se verificó que también estaba corregido. [CRP97].

- El microprocesador PowerPC 620 tenía un error de hardware que lo hacía fallar al momento de arrancar un sistema operativo. Richard Raimi utilizó el verificador de modelos de Motorola, *Verdict*, y en cuestión de segundos se encontró el error: un *deadlock* en el *BIU*[4] del microprocesador [RL97].

[4] Bus Interface Unit

- Con el verificador *Concurrency Workbench* se analizó un sistema de control estructural especificado con la notación gráfica *Modechart* para hacer edificios más resistentes a temblores. Dicho sistema hace un constante muestreo de la fuerza aplicada a la estructura y con ayuda de un actuador hidráulico contrarresta la fuerza ejercida. El sistema debe cumplir estrictos requerimientos de sincronización, ya que una respuesta deficiente del sistema podría empeorar las vibraciones en lugar de moderarlas. Gracias al análisis formal del sistema se logró reducir el modelo inicial que tenía 10^{19} estados a uno mucho más pequeño, permitiendo detectar en poco tiempo un error en el diseño y así evitando muchas catástrofes [ECB97].

La verificación de modelos es una área prolífica dentro de los métodos formales, incluso agencias gubernamentales como la NASA hacen uso e investigación de esta técnica. Empresas dedicadas a la construcción de hardware como IBM, Intel, Cadence, Synopsys hacen uso de la verificación de modelos para garantizar el buen comportamiento de sus microcomponentes. Microsoft hace uso de este método en la verificación formal de *drivers*, los cuales pueden contener miles de líneas de código. En general es mucho más difícil verificar software que hardware por el uso de la recursión, el estado de las variables y las diversas estructuras de datos que se pueden utilizar, pero los *drivers* se comportan más o menos como un sistema secuencial, lo que hace posible verificarlos con esta técnica.

El problema de explosión de estados permanece vigente, pero hoy en día es posible verificar sistemas incluso con una infinidad de estados utilizando una abstracción adecuada del modelo y haciendo uso de técnicas como la *verificación simbólica* o la *verificación al vuelo*.

Y hay otro problema que aún persiste: a los usuarios les resulta complicado aprender a usar un verificador (y no siempre son fáciles de usar). Incluso si se sabe utilizar un verificador de modelos, sigue siendo difícil especificar correctamente los modelos y las fórmulas que representan adecuadamente el comportamiento de un sistema. Hay mucha investigación enfocada en hacer la verificación de modelos más fácil, y se espera que en el futuro en lugar de hacer *testing* siempre se opte por un método formal.

En el siguiente capítulo se presenta formalmente qué es la verificación de modelos desde el punto de vista matemático y de las ciencias de la computación. También se presentan con lujo de detalle la sintaxis y la semántica de las distintas lógicas temporales LTL, CTL y CTL* junto con la complejidad algorítmica para hacer verificación de modelos en cada una de estas lógicas.

Capítulo 2

Lógicas temporales

Temporal Logic is suggested as an appropriate tool for formalizing the semantics of concurrent programs.
Amir Pnueli

Las lógicas temporales permiten describir eventos a lo largo del tiempo al representar el tiempo como una sucesión infinita de estados. Dependiendo de cómo asuman el tiempo, las lógicas temporales se clasifican en *lineales* y *ramificadas*. Curiosamente, las primeras personas en estudiar lógicas temporales no fueron matemáticos ni computólogos, sino filósofos y lingüistas. En 1953, el filósofo Arthur Prior introdujo la lógica modal proposicional en su libro *Time and modality* [Pri03]; esta lógica contaba con dos operadores de tiempo: F para referirse a eventos a futuro y P para referirse a eventos en el pasado. Al inicio, Prior consideró el tiempo de manera lineal, pero Saul Kripke le dijo que esta suposición estaba injustificada. Poco después, Prior desarrolló dos lógicas temporales ramificadas y posteriormente publicó el libro *Past, Present, and Future* [Art67].

Tiempo después, R. M. Burstall [Bur74], Fred Kröger [Krö77] y Amir Pnueli [Pnu77] fueron los primeros en proponer el uso de lógicas temporales para hacer un razonamiento formal acerca de programas computacionales. Pnueli hizo razonamiento de programas concurrentes haciendo uso de la lógica temporal LTL (*Linear Temporal Logic*), su trabajo fue probar propiedades de programas a partir de varios axiomas. Poco después, su trabajo fue extendido por Gregor V. Bochmann para especificar formalmente hardware [Boc82]. Yonatan Malachi y S. S. Owicki lo extendieron a circuitos secuenciales [MO81] pero no tuvo mucho éxito por ser una técnica en la que las pruebas se hacían a mano, lo que hizo que fuese difícil llevarlo a la práctica. Unos años más tarde, E. M. Clarke y E. A. Emerson inventaron la lógica temporal CTL (*Computation Tree Logic*) para verificar circuitos de manera automatizada [EC80]; [CE81]. Posteriormente, Emerson y Joseph Y. Halpern desarrollaron la lógica temporal CTL* como una lógica que contiene a las lógicas LTL y CTL [EH86]. A continuación, se introducen las lógicas LTL, CTL y CTL*.

2.1. LTL (*Linear Temporal Logic)*

La lógica temporal LTL representa el tiempo como una sucesión infinita de estados. Cada estado está etiquetado con las proposiciones atómicas que se cumplen en ese preciso momento. A esta sucesión infinita de estados se le conoce como *trayectoria*.

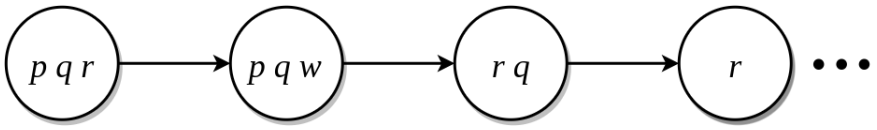


FIGURA 2.1

Las trayectorias sirven para representar la ejecución de un programa a lo largo del tiempo. Las etiquetas indican las condiciones en las que se encuentra el programa para poder decidir si cumple con una propiedad. Esta idea se ilustra en la figura 2.1.

En adelante, $\mathcal{A}$ es un conjunto de proposiciones atómicas.

Definición 1. La sintaxis de la lógica temporal LTL se da a continuación con la siguiente gramática:

$$\mathcal{P} ::= a \mid \neg a \mid \mathcal{P} \vee \mathcal{P} \mid \mathcal{P} \wedge \mathcal{P} \mid \mathcal{P}U\mathcal{P} \mid \mathcal{P}R\mathcal{P} \mid X\mathcal{P} \qquad \text{donde } a \in \mathcal{A}$$

Las fórmulas LTL sirven para expresar propiedades que pueden cumplirse a futuro. A las fórmulas LTL también se les conoce como fórmulas de trayectoria. Por esta razón, el símbolo inicial de la gramática es «$\mathcal{P}$» (del inglés *path*). Una fórmula LTL puede ser una literal, es decir, una variable proposicional o una variable proposicional negada, o bien puede ser la conjunción (disyunción) de dos fórmulas LTL. Esta lógica temporal extiende la sintaxis de la lógica proposicional con los operadores *ne***X***t,* **U***ntil* y **R***elease*. Es importante observar que en esta sintaxis únicamente se niegan fórmulas atómicas.

En adelante, se hará uso del símbolo π para denotar una trayectoria. De lo anterior, si $\pi = \langle s_0, s_1, \dots \rangle$, se define:

- $\pi_i = s_i$ (el i-ésimo estado de la trayectoria)
- $\pi^i = \langle s_i, s_{i+1}, s_{i+2}, \dots \rangle$ (la subtrayectoria a partir del i-ésimo estado)

A continuación, se define la semántica formal de la lógica LTL.

Definición 2. Sea π una trayectoria, L una función que asigna a cada estado un conjunto de etiquetas y ϕ una fórmula LTL. Se define la relación $\pi \vDash \phi$ de manera recursiva como sigue:

- $\pi \vDash a$ sii $a \in L(\pi_0)$.
- $\pi \vDash \neg a$ sii $a \notin L(\pi_0)$.
- $\pi \vDash \phi_1 \vee \phi_2$ sii $\pi \vDash \phi_1$ o $\pi \vDash \phi_2$.
- $\pi \vDash \phi_1 \wedge \phi_2$ sii $\pi \vDash \phi_1$ y $\pi \vDash \phi_2$.
- $\pi \vDash \phi_1 U \phi_2$ sii existe $i \geq 0$ tal que $\pi^i \vDash \phi_2$ y $\forall j < i$, $\pi^j \vDash \phi_1$.
- $\pi \vDash \phi_1 R \phi_2$ sii para toda $i \geq 0$ se cumple $\pi^i \vDash \phi_2$, o existe $i \geq 0$ tal que $\pi^i \vDash \phi_1$ y para toda $j \leq i$, $\pi^j \vDash \phi_2$.
- $\pi \vDash X\phi'$ sii $\pi^1 \vDash \phi'$.

Una trayectoria cumple una fórmula atómica si y sólo si su primer estado se encuentra etiquetado con dicha fórmula (análogo cuando es una fórmula atómica negada). Una trayectoria cumple una conjunción (disyunción) de dos fórmulas si y sólo si cumple ambas (alguna) de las fórmulas. El significado del operador X es intuitivo: una trayectoria π cumple la fórmula $X\phi$ si y sólo si la subtrayectoria π^1 cumple la fórmula ϕ (a partir del *siguiente* estado). El operador U trabaja de la siguiente manera: una trayectoria cumple $\phi_1 U \phi_2$ si y sólo si a partir de un estado s la trayectoria cumple con ϕ_2 y a partir de todos los estados anteriores a s se cumple ϕ_1. Por ejemplo, la trayectoria de la figura 2.2 cumple pUq pues en el estado s_4 se cumple q y en todos los estados anteriores se cumple p. En cambio, no cumple qUr pues en el estado s_5 se cumple r pero no en todos los anteriores se cumple q; lo mismo se cumple para el estado s_6 (los estados que van de s_7 en adelante no tienen etiquetas).

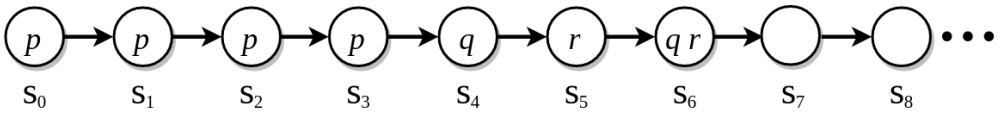


FIGURA 2.2

Una trayectoria tiene dos posibilidades para cumplir con la fórmula $\phi_1 R \phi_2$. El primer caso es cuando ϕ_2 se cumple en todo momento. Por ejemplo, la trayectoria de la figura 2.3 cumple con la fórmula qRr porque en todos los estados se cumple r. El segundo caso

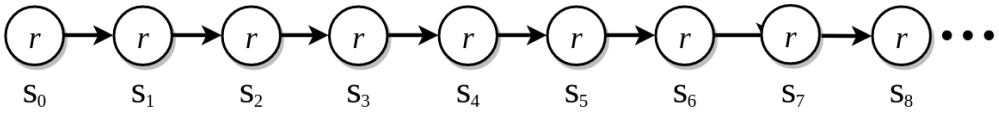


FIGURA 2.3

es cuando ϕ_1 *libera* a ϕ_2, es decir, a partir de un momento se cumple ϕ_2 y desde de todos los estados anteriores junto con el presente se cumple ϕ_1. Por ejemplo, la trayectoria en la figura 2.4 cumple pRq porque en el estado s_3 se cumple q y en todos los estados anteriores junto con s_3 se cumple p. En cambio, la fórmula qRr no se cumple porque la trayectoria cumple r en el estado s_6 pero q no se cumple en todos los anteriores; tampoco se cumple r en todo momento porque a partir del estado s_7 no hay etiquetas.

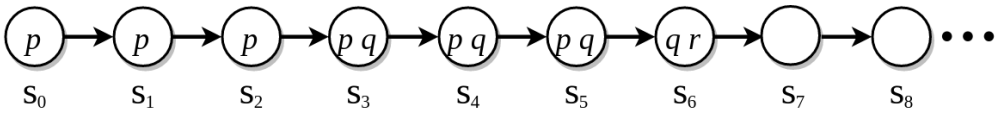


FIGURA 2.4

A continuación se definen dos operadores de tiempo de suma importancia:

Definición 3. Se definen los operadores F y G como sigue:

- $F\phi = \top U \phi$
- $G\phi = \bot R \phi$

De esta manera, una trayectoria cumple con la fórmula $F\phi$ si eventualmente cumple ϕ. La letra «F» proviene del inglés *finally*. Aunque en la sintaxis no está incluida la constante $\top$ esto no representa un problema porque si a es cualquier variable en A, se define $\top = a \vee \neg a$. Análogamente, una trayectoria cumple con la fórmula $G\phi$ si en todo momento cumple ϕ. La letra «G» proviene del inglés *globally*. La constante $\bot$ tampoco representa un problema al no estar en la sintaxis porque $\bot = a \wedge \neg a$, donde a es cualquier variable en A.

Aunque la sintaxis obliga que en las fórmulas LTL sólo se puedan negar fórmulas atómicas (forma normal negativa) es posible negar fórmulas de trayectoria en general.

> **Lema 1.** *Para cualquier fórmula LTL ϕ existe una fórmula $\mathbf{neg}_{\mathcal{P}}(\phi)$ en forma normal negativa de tal manera que se cumple lo siguiente: si π es una trayectoria, entonces $\pi \vDash \phi$ sii $\pi \nvDash \mathbf{neg}_{\mathcal{P}}(\phi)$.*

Demostración: Sean π una trayectoria y ϕ una fórmula LTL. La prueba sigue por inducción sobre ϕ.

- $\phi = a$:

  Sea $\mathbf{neg}_{\mathcal{P}}(\phi) = \neg a$. Es inmediato que $\pi \vDash a$ sii $\pi \nvDash \neg a$. (El caso $\phi = \neg a$ es análogo).

- $\phi = \phi_1 \wedge \phi_2$:

  Por hipótesis de inducción, para ϕ_1 y ϕ_2 existen $\mathbf{neg}_{\mathcal{P}}(\phi_1)$ y $\mathbf{neg}_{\mathcal{P}}(\phi_2)$ que cumplen las hipótesis, se sigue de inmediato tomando $\mathbf{neg}_{\mathcal{P}}(\phi) = \mathbf{neg}_{\mathcal{P}}(\phi_1) \vee \mathbf{neg}_{\mathcal{P}}(\phi_2)$. (El caso $\phi = \phi_1 \vee \phi_2$ es análogo tomando $\mathbf{neg}_{\mathcal{P}}(\phi) = \mathbf{neg}_{\mathcal{P}}(\phi_1) \wedge \mathbf{neg}_{\mathcal{P}}(\phi_2)$).

- $\phi = X\phi'$:

  Por hipótesis de inducción para ϕ' existe $\mathbf{neg}_{\mathcal{P}}(\phi')$ que cumple las hipótesis. Se sigue al tomar $\mathbf{neg}_{\mathcal{P}}(\phi) = X(\mathbf{neg}_{\mathcal{P}}(\phi'))$.

- $\phi = \phi_1 U \phi_2$:

  Por hipótesis de inducción, para ϕ_1 y ϕ_2 existen $\mathbf{neg}_{\mathcal{P}}(\phi_1)$ y $\mathbf{neg}_{\mathcal{P}}(\phi_2)$ que cumplen las hipótesis, se sigue al tomar $\mathbf{neg}_{\mathcal{P}}(\phi) = \mathbf{neg}_{\mathcal{P}}(\phi_1) R \mathbf{neg}_{\mathcal{P}}(\phi_2)$. (El caso $\phi = \phi_1 R \phi_2$ es análogo tomando $\mathbf{neg}_{\mathcal{P}}(\phi) = \mathbf{neg}_{\mathcal{P}}(\phi_1) U \mathbf{neg}_{\mathcal{P}}(\phi_2)$).

□

De esta demostración se obtiene un algoritmo recursivo para negar fórmulas de trayectoria. Dicho algoritmo se muestra en la figura 2.5.

Aunque en la sintaxis no está explícitamente el operador de implicación es posible agregarlo utilizando la función antes descrita.

Definición 4. Se define el operador de implicación como sigue:

$$\phi_1 \rightarrow \phi_2 = \mathbf{neg}_{\mathcal{P}}(\phi_1) \vee \phi_2$$

$$
\begin{array}{rcl}
\mathbf{neg}_{\mathcal{P}}(\phi) = \mathbf{case} \quad \phi \quad \mathbf{of} & & \\
a & \mapsto & \neg a \\
\neg a & \mapsto & a \\
\phi_1 \wedge \phi_2 & \mapsto & \mathbf{neg}_{\mathcal{P}}(\phi_1) \vee \mathbf{neg}_{\mathcal{P}}(\phi_2) \\
\phi_1 \vee \phi_2 & \mapsto & \mathbf{neg}_{\mathcal{P}}(\phi_1) \wedge \mathbf{neg}_{\mathcal{P}}(\phi_2) \\
X\phi' & \mapsto & X\mathbf{neg}_{\mathcal{P}}(\phi') \\
\phi_1 U \phi_2 & \mapsto & \mathbf{neg}_{\mathcal{P}}(\phi_1) R \mathbf{neg}_{\mathcal{P}}(\phi_2) \\
\phi_1 R \phi_2 & \mapsto & \mathbf{neg}_{\mathcal{P}}(\phi_1) U \mathbf{neg}_{\mathcal{P}}(\phi_2)
\end{array}
$$

FIGURA 2.5

Dos fórmulas LTL, ϕ_1 y ϕ_2, son equivalentes si y sólo si para cualquier trayectoria π se tiene que $\pi \vDash \phi_1$ sii $\pi \vDash \phi_2$. En la figura 2.6 se muestran algunas equivalencias de fórmulas LTL que serán de utilidad más adelante.

En el capítulo anterior, de manera informal se presentó una estructura de Kripke con la figura 1.2. A continuación se da su definición formal.

> **Definición 5.** Una *estructura de Kripke* es una terna $\langle S, R, L \rangle$, donde S es un conjunto finito de estados, $R \subseteq S \times S$ es una relación *total* en el siguiente sentido: por cada estado $s \in S$ debe existir por lo menos un $s' \in S$ tal que $(s, s') \in R$, y $L : S \to 2^{\mathcal{A}}$ asigna a cada estado un conjunto de proposiciones atómicas llamadas *etiquetas*.

De manera intuitiva, una estructura de Kripke es una digráfica que captura el comportamiento secuencial de un sistema vía la relación R. La función L etiqueta cada estado con las proposiciones atómicas que se cumplen en cada preciso momento. De esta manera, si $M = \langle S, R, L \rangle$ es una estructura de Kripke, una trayectoria en M es una sucesión $s : \mathbb{N} \to S$ tal que, para cada $i \in \mathbb{N}$, $(s_i, s_{i+1}) \in R$.

De esta manera, se define la verificación de modelos LTL como sigue:

> **Definición 6.** Sea ϕ una fórmula LTL. Si M es una estructura de Kripke y s es un estado de M, entonces $(M, s) \vDash \phi$ sii cualquier trayectoria π en M tal que $\pi_0 = s$ cumple $\pi \vDash \phi$.

Una estructura de Kripke cumple una una especificación a partir de un estado si todas las trayectorias que emanan de ese estado cumplen con dicha especificación.

A continuación se muestran algunos ejemplos de verificación LTL con la estructura de Kripke de la figura 2.7:

- $s_0 \vDash Xr$ se cumple porque todos los sucesores de s_0 cumplen r. En cambio, no se cumple $s_1 \vDash X(p \wedge r)$ porque en el estado s_0 no se cumple r.
- $s_1 \vDash G(q \vee r)$ se cumple porque partiendo del estado s_1 en todo momento se cumple q o se cumple r.

- Distributividad:

$$X(\phi_1 \wedge \phi_2) \equiv X\phi_1 \wedge X\phi_2 \qquad X(\phi_1 \vee \phi_2) \equiv X\phi_1 \vee X\phi_2 \qquad X(\phi_1 U \phi_2) \equiv X\phi_1 U X\phi_2$$

$$F(\phi_1 \vee \phi_2) \equiv F(\phi_1) \vee F(\phi_2) \qquad G(\phi_1 \wedge \phi_2) \equiv G(\phi_1) \wedge G(\phi_2)$$

- Idempotencia:

$$FF\phi \equiv F\phi \qquad GG\phi \equiv G\phi$$

$$\phi_1 U(\phi_1 U\phi_2) \equiv \phi_1 U\phi_2 \qquad (\phi_1 U\phi_2)U\phi_2 \equiv \phi_1 U\phi_2$$

- Absorción:

$$FGF\phi \equiv GF\phi \qquad GFG\phi \equiv FG\phi$$

- Expansión:

$$\phi_1 U\phi_2 \equiv \phi_2 \vee (\phi_1 \wedge X(\phi_1 U\phi_2)) \qquad \phi_1 R\phi_2 \equiv \phi_2 \wedge (\phi_1 \vee X(\phi_1 R\phi_2))$$

$$F\phi \equiv \phi \vee XF\phi \qquad G\phi \equiv \phi \wedge XG\phi$$

FIGURA 2.6: Algunas equivalencias de fórmulas LTL

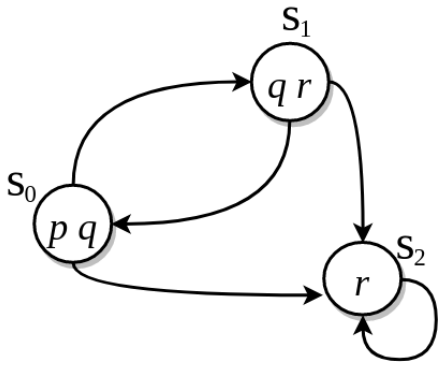


FIGURA 2.7

- $s_0 \vDash GFp$ se lee como «*a partir del estado s_0, la variable p ocurre de forma infinitamente frecuente*». Es decir, partiendo del estado s_0, y sin importar cuánto se avance, siempre se alcanzan estados en los que se cumple p. La fórmula GFp no se cumple porque en la trayectoria $\langle s_0, s_2, s_2, \dots \rangle$ nunca vuelve a alcanzar s_0. Por otra parte, $s_0 \vDash GFr$ se cumple porque, sin importar cuál trayectoria se elija ni cuanto se avance en ella, siempre se alcanza un estado que cumple r.

- $s_1 \vDash FGr$ se lee como «*a partir del estado s_1, eventualmente la variable r se cumple por siempre*». Es decir, a partir del estado s_1 eventualmente se alcanza un estado s' en el que, a partir de s', r se cumple por siempre. Así, $s_1 \vDash FGr$ no se cumple pues en la trayectoria $\langle s_1, s_0, s_1, s_0, \dots \rangle$ nunca se alcanza una trayectoria que siempre cumpla r. En cambio, $s_2 \vDash FGr$ se cumple porque desde el primer momento se alcanza una trayectoria que siempre cumple r.

Sistla y Clarke demostraron que el problema de verificación de modelos LTL es *PSPACE-completo* [SC85]; [Sch03]. Pnueli y Lichtenstein dieron un algoritmo que es lineal en el tamaño del modelo y exponencial en la longitud de la fórmula [LP85], lo que sugiere que es posible hacer verificación LTL en un tiempo razonable cuando las fórmulas son relativamente cortas.

2.2. CTL (*Computation Tree Logic)*

La lógica temporal CTL no considera el tiempo como una trayectoria sino como un árbol infinito en el que el futuro no siempre está determinado: hay momentos en los que es posible elegir entre varios caminos para llegar a un futuro diferente. Al igual que en la lógica LTL, cada estado está etiquetado con las proposiciones que se cumplen en ese preciso momento, pero en esta lógica un estado puede tener más de un sucesor, haciendo que el futuro se ramifique. Esta idea se ilustra en la figura 2.8.

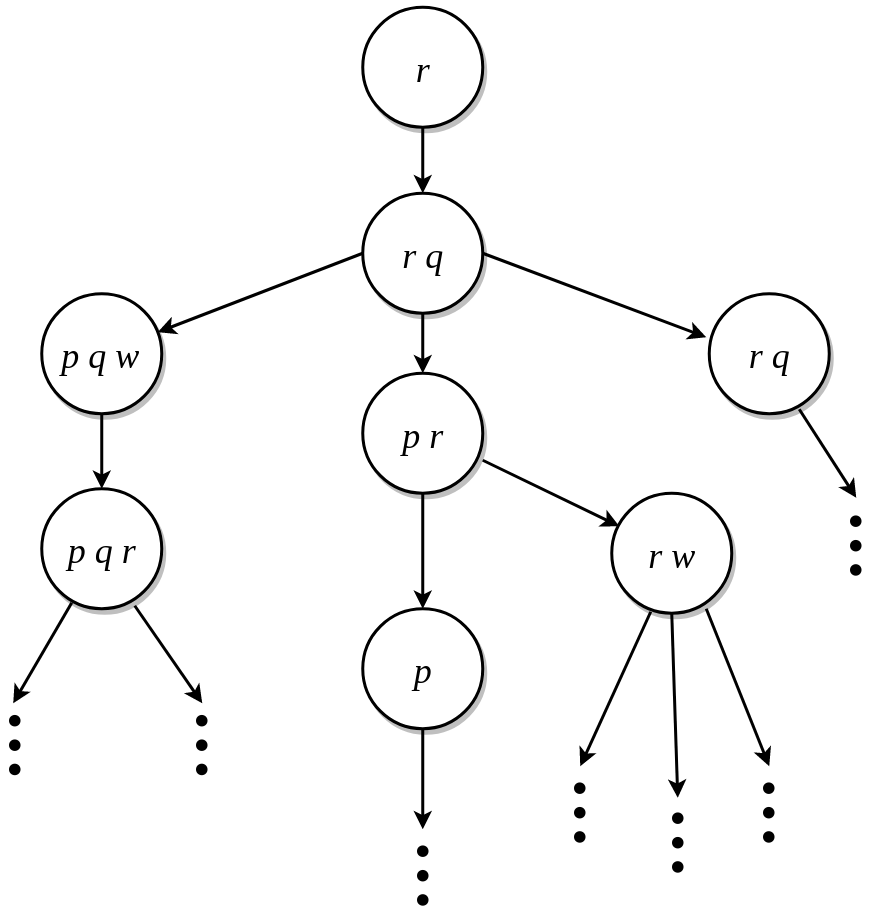


FIGURA 2.8

Como se vio en la definición 6, una fórmula LTL se cumple en una estructura a partir de un estado si *todas* las trayectorias que emergen de dicho estado cumplen dicha fórmula. De forma implícita, LTL cuantifica de forma universal las trayectorias que emergen de un estado. Sin embargo, a veces se desea que una propiedad se cumpla solamente

para algunas trayectorias, lo que hace que LTL no sea la lógica adecuada para realizar esta clase de verificación. Una posible solución a este problema es hacer uso de la dualidad del cuantificador que está implícito en LTL, es decir, hacer uso de la equivalencia $\forall\pi[\pi \vDash \phi] \equiv \neg(\exists\pi[\pi \nvDash \phi])$. Sin embargo, esta técnica no es de utilidad cuando se mezclan cuantificadores de trayectoria en una misma fórmula. Este problema se resuelve utilizando una lógica temporal ramificada.

La lógica temporal CTL es una lógica ramificada que permite cuantificar trayectorias de manera explícita, haciendo posible especificar si se desea que una propiedad se cumpla para todas o para algunas de las trayectorias que emergen de un estado.

Definición 7. La sintaxis de las fórmulas CTL se describe con la siguiente gramática:

$$\mathcal{S} ::= a \mid \neg a \mid \mathcal{S} \wedge \mathcal{S} \mid \mathcal{S} \vee \mathcal{S} \mid \mathcal{Q}X\mathcal{S} \mid \mathcal{Q}[\mathcal{S}U\mathcal{S}] \mid \mathcal{Q}[\mathcal{S}R\mathcal{S}]$$

$$\mathcal{Q} ::= A \mid E$$

Los símbolos «$\mathcal{S}$» y «$\mathcal{Q}$» provienen del inglés *state* y *quantifier*. Al igual que la lógica LTL, CTL es una extensión de la lógica proposicional. Pero a diferencia de la lógica LTL, cada operador de tiempo debe estar precedido de un cuantificador de trayectorias, ya sea A (universal) o E (existencial). Además, la sintaxis permite combinar ambos cuantificadores en una misma fórmula.

La semántica formal de la lógica CTL se describe a continuación:

Definición 8. Sean $M = \langle S, R, L\rangle$ una estructura de Kripke, s un estado de M y φ una fórmula CTL. Se define la relación $(M, s) \vDash \varphi$ de manera recursiva como sigue:

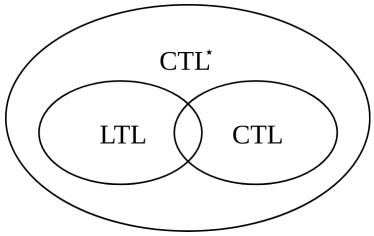
- $(M, s) \vDash a$ sii $a \in L(s)$ (análogo para $s \vDash \neg a$).
- $(M, s) \vDash \varphi_1 \wedge \varphi_2$ sii $(M, s) \vDash \varphi_1$ y $(M, s) \vDash \varphi_2$.
- $(M, s) \vDash \varphi_1 \vee \varphi_2$ sii $(M, s) \vDash \varphi_1$ o $(M, s) \vDash \varphi_2$.
- $(M, s) \vDash AX\varphi'$ sii $\forall s' \in \mathcal{S}$, $(s, s') \in R$ y $(M, s') \vDash \varphi'$.
- $(M, s) \vDash EX\varphi'$ sii $\exists s' \in \mathcal{S}$, $(s, s') \in R$ y $(M, s') \vDash \varphi'$.
- $(M, s) \vDash A[\varphi_1 U \varphi_2]$ sii $\forall\pi$ en M, si $\pi_0 = s$ entonces existe $i \geq 0$ tal que $\pi_i \vDash \varphi_2$ y para toda $j < i$, $\pi_j \vDash \varphi_1$ (análogo para $s \vDash E[\varphi_1 U \varphi_2]$).
- $(M, s) \vDash A[\varphi_1 R \varphi_2]$ sii $\forall\pi$ en M, si $\pi_0 = s$ entonces, o para toda $i \geq 0$ se cumple $\pi_i \vDash \varphi_2$, o existe $i \geq 0$ tal que $\pi_i \vDash \varphi_1$ y para toda $j \leq i$, $\pi_j \vDash \varphi_2$ (análogo para $s \vDash E[\varphi_1 R \varphi_2]$).

En 1986, Clarke mostró un algoritmo para hacer verificación de modelos CTL cuyo desempeño es polinomial (el tamaño de la estructura por la longitud de la fórmula a verificar) [CES86]; el mismo Clarke implementó dicho algoritmo en el lenguaje *Franz Lisp* para verificar circuitos secuenciales y protocolos de red. Schnoebelen dio una prueba rigurosa para mostrar que el problema de verificación CTL es *P-completo* [Sch03].

2.3. CTL*

Después de haber visto LTL y CTL surgen varias preguntas: ¿Cuál de las dos lógicas tiene más poder de expresividad? ¿Al ser CTL ramificada contiene propiamente a LTL? ¿LTL tiene menos expresividad que CTL por representar el tiempo en una trayectoria? La respuesta es que LTL y CTL son incomparables. Hay fórmulas en LTL que no pueden expresarse en CTL (GFp o FGp, por ejemplo) y fórmulas en CTL que no pueden expresarse en LTL ($AXEXp$, por ejemplo). Algunas fórmulas en LTL sí pueden expresarse en CTL: la fórmula LTL $G(p \to Fq)$ equivale a la fórmula CTL $AG(p \to AFq)$; y lo mismo ocurre para CTL, la fórmula $AXAFp$ es equivalente a la fórmula LTL XFp (y también a la fórmula FXp, pues, en LTL, $XFp \equiv FXp$).

La lógica temporal CTL* es una generalización común de las lógicas LTL y CTL. Como vimos antes, LTL describe propiedades de trayectorias y CTL describe propiedades de estados. CTL* fusiona el poder de ambas lógicas teniendo dos tipos de fórmulas en su sintaxis: fórmulas de estado y fórmulas de trayectorias:



Definición 9. La sintaxis de la lógica CTL* se define de manera recursiva como sigue:

$$\mathcal{S} ::= a \mid \neg a \mid \mathcal{S} \wedge \mathcal{S} \mid \mathcal{S} \vee \mathcal{S} \mid A\mathcal{P} \mid E\mathcal{P}$$

$$\mathcal{P} ::= \mathcal{S} \mid \mathcal{P} \wedge \mathcal{P} \mid \mathcal{P} \vee \mathcal{P} \mid X\mathcal{P} \mid \mathcal{P}U\mathcal{P} \mid \mathcal{P}R\mathcal{P}$$

Es importante notar que las fórmulas de estado y las fórmulas de trayectoria hacen recursión simultanea. CTL* tiene más expresividad que LTL y CTL. Las fórmulas de trayectoria pueden tener subfórmulas de estado más elaboradas que simples literales (contrario de LTL). Por ejemplo, es posible expresar $AGF(Ep)$. Y en las fórmulas de estado un cuantificador de trayectorias no necesariamente debe preceder inmediatamente a un operador de tiempo (contrario a CTL), lo que permite expresar $EFXp$.

Tanto LTL como CTL se desarrollaron de forma independiente antes que CTL* apareciera, pero ambas lógicas resultaron ser fragmentos de CTL*:

- LTL: fórmulas de la forma $A\phi$, donde ϕ una fórmula de trayectoria y todas las subfórmulas de estado de ϕ únicamente son literales.
- CTL: fórmulas de estado en las que cada operador de tiempo está inmediatamente precedido por un cuantificador de trayectorias.

En adelante, el símbolo ϕ se usará para denotar fórmulas de trayectoria y φ para denotar fórmulas de estado.

Definición 10. Sean $M = \langle S, R, L \rangle$ una estructura de Kripke y $s \in S$. Se define la relación $s \vDash_M \varphi$ como sigue:

- $s \vDash_M a$ sii $a \in L(s)$ (análogo para $s \vDash_M \neg a$).
- $s \vDash_M \varphi_1 \wedge \varphi_2$ sii $s \vDash_M \varphi_1$ y $s \vDash_M \varphi_2$.
- $s \vDash_M \varphi_1 \vee \varphi_2$ sii $s \vDash_M \varphi_1$ o $s \vDash_M \varphi_2$.
- $s \vDash_M A\phi$ sii $\forall \pi \in M$, si $\pi_0 = s$ entonces $\pi \vDash_M \phi$.
- $s \vDash_M E\phi$ sii $\exists \pi \in M$, si $\pi_0 = s$ entonces $\pi \vDash_M \phi$.

  La relación $\pi \vDash_M \phi$ se define recursivamente como sigue:

  - $\pi \vDash_M \varphi$ sii $\pi_0 \vDash_M \varphi$.
  - $\pi \vDash_M \phi_1 \wedge \phi_2$ sii $\pi \vDash_M \phi_1$ y $\pi \vDash_M \phi_2$.
  - $\pi \vDash_M \phi_1 \vee \phi_2$ sii $\pi \vDash_M \phi_1$ o $\pi \vDash_M \phi_2$.
  - $\pi \vDash_M X\phi'$ sii $\pi^1 \vDash_M \phi'$.
  - $\pi \vDash_M \phi_1 U \phi_2$ sii existe $i \geq 0$ tal que $\pi^i \vDash_M \phi_2$ y para toda $j < i$, $\pi^j \vDash_M \phi_1$.
  - $\pi \vDash_M \phi_1 R \phi_2$ sii para toda $i \geq 0$ se cumple $\pi^i \vDash_M \phi_2$, o existe $i \geq 0$ tal que $\pi^i \vDash_M \phi_1$ y para toda $j \leq i$, $\pi^j \vDash_M \phi_2$.

La semántica CTL* resulta natural luego de haber visto las semánticas LTL y CTL. Al igual que la sintaxis, la semántica para CTL* hace recursión simultanea para fórmulas de estado y fórmulas de trayectoria. Esto sugiere que para decidir si una estructura de Kripke cumple con una fórmula CTL* bastaría tener un algoritmo de verificación LTL que haga recursión simultanea entre fórmulas de estado y fórmuulas de trayectoria, pues para revisar las fórmulas $A\phi$ bastaría tratar a ϕ como una fórmula LTL cuidando que cada que se vea una fórmula de estado se regrese a la semántica CTL* (a las fórmulas $E\phi$ se les puede tratar con la equivalencia $E\phi \equiv \neg A \neg \phi$). Recalcamos que esta observación es fundamental para construir un algoritmo para hacer verificación CTL*.

$$
\begin{aligned}
\mathbf{neg}_{\mathcal{S}}(\phi) = \mathbf{case}\ & \varphi\ \mathbf{of} \\
a &\mapsto \neg a \\
\neg a &\mapsto a \\
\varphi_1 \wedge \varphi_2 &\mapsto \mathbf{neg}_{\mathcal{S}}(\varphi_1) \vee \mathbf{neg}_{\mathcal{S}}(\varphi_2) \\
\varphi_1 \vee \varphi_2 &\mapsto \mathbf{neg}_{\mathcal{S}}(\varphi_1) \wedge \mathbf{neg}_{\mathcal{S}}(\varphi_2) \\
A\phi &\mapsto E(\mathbf{neg}_{\mathcal{P}}(\phi)) \\
E\phi &\mapsto A(\mathbf{neg}_{\mathcal{P}}(\phi)) \\
\mathbf{neg}_{\mathcal{P}}(\phi) = \mathbf{case}\ & \phi\ \mathbf{of} \\
\varphi &\mapsto \mathbf{neg}_{\mathcal{S}}(\varphi) \\
\phi_1 \wedge \phi_2 &\mapsto \mathbf{neg}_{\mathcal{P}}(\phi_1) \vee \mathbf{neg}_{\mathcal{P}}(\phi_2) \\
\phi_1 \vee \phi_2 &\mapsto \mathbf{neg}_{\mathcal{P}}(\phi_1) \wedge \mathbf{neg}_{\mathcal{P}}(\phi_2) \\
X\phi' &\mapsto X\mathbf{neg}_{\mathcal{P}}(\phi') \\
\phi_1 U \phi_2 &\mapsto \mathbf{neg}_{\mathcal{P}}(\phi_1) R \mathbf{neg}_{\mathcal{P}}(\phi_2) \\
\phi_1 R \phi_2 &\mapsto \mathbf{neg}_{\mathcal{P}}(\phi_1) U \mathbf{neg}_{\mathcal{P}}(\phi_2)
\end{aligned}
$$

FIGURA 2.9

Es posible negar cualquier fórmula CTL* utilizando las dualidades A y E, $\wedge$ y $\vee$, U y R, y la autodualidad del operador X. En la figura 2.9 se muestran las funciones $\mathbf{neg}_{\mathcal{S}}(\_)$ y $\mathbf{neg}_{\mathcal{P}}(\_)$ para negar fórmulas de trayectoria y fórmulas de estado, respectivamente. Ambas funciones se obtienen de la semántica formal y también hacen recursión simultanea.

Hacer verificación de modelos en la lógica CTL* es un problema *PSPACE-completo* al igual que el problema de verificación de modelos LTL [CES86]; [Sch03].

En el próximo capítulo se explica cómo construir un algoritmo para hacer verificación de modelos CTL*, el cual tiene como base un verificador LTL que funciona *al vuelo* y que se extiende a CTL* haciendo recursión simultanea en las subfórmulas de estado y en las fórmulas de trayectoria.

Capítulo 3

Un verificador al vuelo para LTL

While simulation and testing explore some of the possible behaviors and scenarios of the system, leaving open the question of whether the unexplored trajectories may contain the fatal bug, formal verification conducts an exhaustive exploration of all possible behaviors.
Clarke, Grumberg & Peled

The best programs are written so that computing machines can perform them quickly and so that human beings can understand them clearly. A programmer is ideally an essayist who works with traditional aesthetic and literary forms as well as mathematical concepts, to communicate the way that an algorithm works and to convince a reader that the results will be correct.
Donald Knuth

Es posible hacer verificación de modelos para la lógica CTL de forma eficiente, E. Clarke dio un algoritmo que resuelve el problema en tiempo polinomial [CES86]. La forma en la que Clarke representó los modelos fue con listas de adyacencias; de esta manera, para sistemas con un número pequeño de procesos la estructura resultante era relativamente pequeña y fácil de especificar, pero para sistemas con una gran cantidad de estados esta forma de representar las estructuras se volvía complicada y engorrosa. En 1987, McMillan implementó el algoritmo de Clarke pero de forma *simbólica*: esta técnica codifica la estructura de Kripke en un OBDD (*Ordered Binary Decision Diagram*), el cual permite representar de forma eficiente incluso modelos con 10^{120} estados [Bur+92]. Aunque esta forma de verificar es conveniente para tratar modelos grandes el problema de verificación CTL simbólico es *PSPACE-completo* [Sch03].

A diferencia de la verificación CTL, hacer verificación de modelos LTL no es algo conceptualmente sencillo porque la mayoría de los algoritmos de verificación para esta lógica utilizan autómatas de Büchi. Estos autómatas reconocen lenguajes ω-regulares, es decir, lenguajes cuyas cadenas tienen longitud infinita [Var96]; [LP85]; [BKL08].

Hay una gran variedad de enfoques que utilizan autómatas de Büchi para hacer verificación de modelos LTL: verificación simbólica, permitiendo verificar modelos con una gran cantidad de estados [CGP00]; [CGH94]; [Roz11]. Combinando el algoritmo de Tarjan para que la verificación LTL sea eficiente [GV05]. Verificación LTL *al vuelo*, esto es, la estructura de Kripke y la fórmula de entrada se exploran sobre la marcha evitando hacer trabajo de más [Cou99]. Verificación LTL al vuelo y de forma simbólica [Ger+95]; [HKM05]. Con algoritmos distribuidos [Bri+01]; [ČP03]. Utilizando el algoritmo de búsqueda BFS de forma distribuida [BBS01]. En paralelo [Bar+18]; [BBC03] o con un enfoque que aproveche una arquitectura *multi-core* [BBR07]. Incluso es posible hacer síntesis de propiedades CTL* vía LTL utilizando autómatas de Büchi [BSK17].

Aunque los autómatas de Büchi han sido ampliamente estudiados dentro de la verificación de modelos los algoritmos que los involucran suelen ser complicados de entender y de implementar.

A continuación se muestra un algoritmo para hacer verificación LTL que no está basado en autómatas de Büchi, que se obtiene de la semántica formal y que se puede extender a un verificador de modelos para la lógica CTL*.

3.1. Un verificador LTL *al vuelo*

En esta sección se muestra la construcción de un algoritmo propuesto en [BCG95] para hacer verificación de modelos LTL al vuelo y que no está basado en autómatas de Büchi. La base del algoritmo es el uso de *aserciones* y *reglas* para construir una *derivación*. Además, este algoritmo se extiende de forma natural a CTL*. En la sección 3.2 este algoritmo se simplifica para obtener un algoritmo que es aún más fácil de entender y de implementar y que también se extiende de forma natural a la lógica CTL*.

> **Definición 11.** ([BCG95]) Sean M una estructura de Kripke, Φ un conjunto finito de fórmulas de trayectoria y s un estado en M. Decimos que $s \vdash_M A\Phi$ es una *aserción* sobre M y su significado es el siguiente:
>
> $$s \vdash_M A\Phi \text{ se cumple sii } s \vDash_M A\left(\bigvee_{\phi\in\Phi} \phi\right)$$

Esencialmente, una aserción es un par: un estado s y un conjunto finito de fórmulas de trayectoria Φ. Una aserción $s \vdash_M A\Phi$ se cumple en una estructura M si alguna fórmula $\phi \in \Phi$ cumple $s \vDash_M A\phi$. Es importante observar que Φ puede ser vacío, y en ese caso $\bigvee_{\phi\in\Phi} \phi = \bigvee_{\phi\in\emptyset} \phi \equiv \bot$. El símbolo σ se usará para denotar aserciones.

Las aserciones se utilizan para definir las reglas de inferencia con estilo *top-down* que aparecen en la figura 3.1. Estas reglas sirven para construir *derivaciones*.

> **Definición 12.** Una derivación para la aserción σ es una gráfica dirigida que cumple lo siguiente:
>
> - Los nodos de la gráfica son aserciones o *true*.
> - Hay un arco de σ' a σ'' si al aplicar alguna regla a σ' se obtiene σ''.
> - Cada nodo σ' es alcanzado desde σ.

Como puede verse en la figura 3.1, en la parte superior de cada regla se encuentran las premisas requeridas para aplicar las reglas y en la parte inferior se indican las *metas* que se generan al aplicar cada una de las reglas. La notación $A(\phi, \Phi)$ denota $A(\{\phi\} \cup \Phi)$. En las reglas R_1 y R_2 hay condiciones adicionales, en estas condiciones se tiene que $a \in A$. La regla R_7 se puede aplicar únicamente cuando todas las fórmulas de la aserción son X-fórmulas. Estas reglas se obtienen directamente de la semántica para LTL y están inspiradas de forma directa en las reglas dadas por Dam para trasladar fórmulas CTL* dentro del cálculo-μ [Dam94].

$$R_1 : \frac{s \vdash_M A(\varphi, \Phi)}{true}(s \vDash_M \varphi) \qquad R_2 : \frac{s \vdash_M A(\varphi, \Phi)}{s \vdash_M A\Phi}(s \nvDash_M \varphi)$$

$$R_3 : \frac{s \vdash_M A(\phi_1 \vee \phi_2, \Phi)}{s \vdash_M A(\phi_1, \phi_2, \Phi)} \qquad R_4 : \frac{s \vdash_M A(\phi_1 \wedge \phi_2, \Phi)}{s \vdash_M A(\phi_1, \Phi) \qquad s \vdash_M A(\phi_2, \Phi)}$$

$$R_5 : \frac{s \vdash_M A(\phi_1 U \phi_2, \Phi)}{s \vdash_M A(\phi_1, \phi_2, \Phi) \qquad s \vdash_M A(\phi_2, X(\phi_1 U \phi_2), \Phi)}$$

$$R_6 : \frac{s \vdash_M A(\phi_1 R \phi_2, \Phi)}{s \vdash_M A(\phi_2, \Phi) \qquad s \vdash_M A(\phi_1, X(\phi_1 R \phi_2), \Phi)}$$

$$R_7 : \frac{s \vdash_M A(X\phi_1, X\phi_2, \ldots, X\phi_n)}{s_1 \vdash_M A(\phi_1, \phi_2, \ldots, \phi_n) \ \ldots \ s_m \vdash_M A(\phi_1, \phi_2, \ldots, \phi_n)} \quad \{(s, s_i) \in R\}_{i=1}^{m}$$

FIGURA 3.1: $M = \langle S, R, L \rangle$ y φ es una literal.

Consideremos la aserción $\sigma = s \vdash_M A(p \wedge q, q \vee r, X(pUq))$. Al aplicar la regla R_3 a σ se obtiene la meta:

$$\frac{s \vdash_M A(p \wedge q, q \vee r, X(pUq))}{s \vdash_M A(p \wedge q, q, r, X(pUq))}(R_3)$$

A aplicar la regla R_4 a σ se obtienen las metas:

$$\frac{s \vdash_M A(p \wedge q, q \vee r, X(pUq))}{s \vdash_M A(p, q \vee r, X(pUq)) \qquad s \vdash_M A(q, q \vee r, X(pUq))}(R_4)$$

Como se observa arriba, se puede elegir más de una regla para aplicarla a una aserción. Esto hace que una derivación pueda tener varias derivaciones.

Suponiendo que los sucesores del estado s son $\{s_1, s_2, s_3\}$, al aplicar la regla R_5 a la aserción $s \vdash_M (X(p \wedge q), X(qRp))$ se obtienen las metas:

$$\frac{s \vdash_M A(X(p \wedge q), X(qRp))}{s_1 \vdash_M A(p \wedge q, qRp) \qquad s_2 \vdash_M A(p \wedge q, qRp) \qquad s_3 \vdash_M A(p \wedge q, qRp)}(R_7)$$

Las derivaciones sirven para inferir si una estructura de Kripke satisface una fórmula LTL sin necesidad de escribir una demostración formal, lo cual sirve como núcleo para un verificador de modelos LTL. El siguiente lema establece que las reglas para construir una derivación son consistentes con la semántica formal para LTL.

Lema 2. *([BCG95] Lema 3.1) Sea $\sigma = s \vdash_M A\Phi$ una aserción.*

1. *Si la única submeta que resulta de aplicar alguna regla a σ es true, entonces $s \vDash_M A\Phi$.*
2. *Si ninguna regla puede ser aplicada a σ, entonces $s \nvDash_M A\Phi$.*
3. *Si todas las submetas que resultan de aplicar alguna de las reglas a σ son de la forma $s_1 \vdash_M A\Phi_1 \ \ldots \ s_m \vdash_M A\Phi_m$, entonces*

$$s \vDash_M A\Phi \text{ sii } s_i \vDash_M A\Phi_i,\ 1 \leq i \leq m$$

Demostración:

1. Si la única submeta que resulta de aplicar una regla a σ fue *true* es porque la regla aplicada fue la regla R_1. Eso quiere decir que existe una fórmula atómica $a \in \Phi$ tal que $a \in L(s)$. Entonces, $s \vDash_M a$. Por tanto $s \vDash_M A\Phi$.

2. Si ninguna regla puede ser aplicada a $s \vdash_M A\Phi$, quiere decir que Φ no empata con ninguna de las premisas de las reglas en la figura 3.1. Eso quiere decir que forzosamente Φ es vacío. Como $\bigvee_{\phi \in \emptyset} \phi \equiv \bot$, $s \vdash_M A\emptyset$ si y sólo si $s \vDash_M A\bot$. Por lo tanto, $s \nvDash_M A\Phi$.

3. Se procede a hacer un análisis de casos sobre las reglas:

   R_1: Si se cumple $s \vDash_M A(a, \Phi)$ y $a \in L(s)$, entonces $s \vDash_M A(a, \Phi)$ si y sólo si *true*.

   R_2: De forma similar al caso anterior, si se cumple $s \vDash_M A(a, \Phi)$ y $a \notin L(s)$, entonces existe $\phi \in \Phi$, $a \neq \phi$, tal que $s \vDash_M A\phi$. Por tanto, $s \vDash_M A(a, \Phi)$ si y sólo si $s \vDash_M A\Phi$.

   R_3: Si $s \vDash_M A(\phi_1 \vee \phi_2, \Phi)$, entonces se cumple $s \vDash_M A(\phi_1 \vee \phi_2)$ o $s \vDash_M A\Phi$. En cualquier caso, se cumple $s \vDash_M A(\phi_1 \vee \phi_2, \Phi)$ si y sólo si $s \vDash_M A(\phi_1, \phi_2, \Phi)$

   R_4: Si $s \vDash_M A(\phi_1 \wedge \phi_2, \Phi)$, entonces se cumple $s \vDash_M A(\phi_1 \wedge \phi_2)$ o $s \vDash_M A\Phi$. De $s \vDash_M A(\phi_1 \wedge \phi_2)$ se obtiene que $s \vDash_M A\phi_1$ y $s \vDash_M A\phi_2$. Entonces, se cumple $s \vDash_M A(\phi_1 \wedge \phi_2, \Phi)$ si y sólo si se cumple $s \vDash_M A(\phi_1, \Phi)$ y $s \vDash_M A(\phi_2, \Phi)$.

   R_5, R_6 y R_7: Similar al caso anterior, se siguen de la equivalencias:

$$\phi_1 U \phi_2 \equiv \phi_2 \vee (\phi_1 \wedge X(\phi_1 U \phi_2)) \equiv (\phi_1 \vee \phi_2) \wedge (\phi_2 \vee X(\phi_1 U \phi_2))$$

$$\phi_1 R \phi_2 \equiv \phi_2 \wedge (\phi_1 \vee X(\phi_1 R \phi_2))$$

$$X\phi_1 \vee X\phi_2 \ \ldots \ \vee X\phi_n \equiv X(\phi_1 \vee \phi_2 \ \ldots \ \vee \phi_n)$$

   que aparecen en la figura 2.6, respectivamente.

□

El lema anterior asegura que una derivación V para una aserción σ en la que todos los nodos terminales son *true* muestra la validez de σ, en este caso se dice que la derivación V es *exitosa* o que es una prueba de la validez de σ; y garantiza que cuando se alcanza a una aserción cuyo conjunto de fórmulas es vacío no tiene caso seguir construyendo la derivación porque la aserción inicial no se cumple, en este caso se dice que la derivación *falla*. Veamos algunos ejemplos para la estructura de Kripke de la figura 3.2:

- $s_0 \vdash_M A(r \vee (p \wedge q))$ se cumple y una derivación que lo prueba es la siguiente:

$$\dfrac{\dfrac{\dfrac{\dfrac{s_0 \vdash_M A(r \vee (p \wedge q))}{s_0 \vdash_M A(r, p \wedge q)}(R_3)}{s_0 \vdash_M A(p \wedge q)}(R_2)}{\dfrac{s_0 \vdash_M A(p)}{true}(R_1) \qquad \dfrac{s_0 \vdash_M A(q)}{true}(R_1)}(R_4)$$

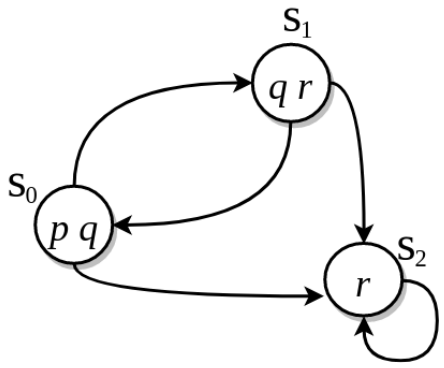


FIGURA 3.2

- $s_1 \vdash_M (X(p \vee q))$ no se cumple; la siguiente derivación falla al alcanzar una aserción cuyo conjunto de fórmulas es vacío:

$$\dfrac{\dfrac{\dfrac{s_0 \vdash_M A(p \vee q)}{s_0 \vdash_M A(p,q)}(R_3)}{true}(R_1) \qquad \dfrac{\dfrac{\dfrac{s_2 \vdash_M A(p \vee q)}{s_2 \vdash_M A(p,q)}(R_3)}{s_2 \vdash_M A(q)}(R_2)}{s_2 \vdash_M A\emptyset}(R_2)}{\times}(R_7)$$

  Los dos ejemplos anteriores muestran que es posible saber si una derivación es exitosa haciendo una conjunción de todos sus nodos terminales. Sin embargo, estas observaciones no son criterio suficiente para saber si una derivación es exitosa pues una derivación puede contener ciclos.

  Por las leyes de expansión mostradas en la figura 2.6, es posible añadir las siguientes reglas:

$$R_F : \frac{s \vdash_M A(F\phi, \Phi)}{s \vdash_M A(\phi, X(F\phi), \Phi)} \qquad R_G : \frac{s \vdash_M A(G\phi, \Phi)}{s \vdash_M A(\phi, \Phi) \qquad s \vdash_M A(X(G\phi), \Phi)}$$

  Estas reglas siguen siendo consistentes con la semántica para LTL pues son casos particulares de las reglas R_5 y R_6.

  Para comprobar que las derivaciones se pueden ciclar, se muestran estos ejemplos:

- $s_2 \vdash_M A(Fp)$ no se cumple porque desde el estado s_2 no es posible alcanzar un estado que cumpla p, pero la derivación se cicla:

$$\dfrac{\dfrac{\dfrac{\dfrac{s_2 \vdash_M A(Fp)}{s_2 \vdash_M A(p, XFp)}(R_2)}{s_2 \vdash_M A(XFp)}(R_7)}{s_2 \vdash_M A(Fp)}(R_F)}{\vdots}$$

La regla R_F cicla una derivación cuando encuentra una fórmula $F\phi$ en la que, a lo largo de alguna trayectoria, nunca alcanza a cumplirse ϕ.

- $s_2 \vdash_M A(Gr)$ se cumple porque desde s_2 siempre se cumple r, pero la derivación también se cicla:

$$\dfrac{\dfrac{s_2 \vdash_M A(r)}{true}(R_1) \qquad \dfrac{s_2 \vdash_M A(Gr)}{\dfrac{s_2 \vdash_M A(XGr)}{\dfrac{s_2 \vdash_M A(Gr)}{\dfrac{s_2 \vdash_M A(r)}{true}(R_1) \qquad \dfrac{s_2 \vdash_M A(XGr)}{\vdots}(R_7)}(R_G)}(R_7)}(R_G)}{s_2 \vdash_M A(Gr)}(R_G)$$

De forma dual a la regla R_F, la regla R_G cicla una derivación cuando encuentra una fórmula $G\phi$ tal que ϕ se cumple a lo largo de todas las trayectorias.

Lo anterior sugiere dos cosas: que un ciclo generado por una U-fórmula hace que una derivación falle, y de forma dual, que un ciclo generado por una R-fórmula no anula la validez de una derivación.

Para caracterizar las derivaciones que son válidas en una estructura de Kripke hace falta un criterio que contemple el caso cuando éstas contienen ciclos. Ya que las derivaciones pueden verse como gráficas dirigidas, se puede ver a los ciclos como trayectorias.

Definición 13. Si V es una derivación, la sucesión $\langle \sigma_0, \sigma_1, \sigma_2, \dots \rangle$ es una *trayectoria* en V si para cada $i \geq 0$, σ_{i+1} se obtiene de aplicar alguna regla a σ_i.

Como el número de estados de una estructura es finito y la relación de accesibilidad es total, forzosamente las trayectorias en una estructura repiten estados. Esto hace que las trayectorias infinitas dentro de una derivación se ciclen.

A continuación se clasifican las trayectorias dentro de una derivación.

Definición 14. Sea V una derivación:

- Una trayectoria finita $\langle \sigma_0, \sigma_1, \dots, \sigma_k \rangle$ en V es exitosa sii $\sigma_k = true$.
- Una trayectoria infinita $\langle \sigma_0, \sigma_1, \dots \rangle$ en V es exitosa sii existe $i \geq 0$ tal que existe $\sigma_i \in V$ que cumple lo siguiente: existe una fórmula $\phi_1 R \phi_2 \in \sigma_i$ tal que para toda $j \geq i$, $\phi_2 \notin \sigma_j$.
- V es una derivación exitosa sii cada trayectoria en V es exitosa.

Al igual que una fórmula LTL, la validez de una aserción depende de todas sus trayectorias. Definir una trayectoria que termina en *true* como exitosa es un corolario del lema 3.1. De forma intuitiva, una trayectoria infinita es exitosa si es infinitamente regenerada al aplicar la regla R_6 a una aserción con una fórmula $\phi_1 R \phi_2$, pero la condición que pide que ϕ_2 no aparezca en las aserciones posteriores puede resultar confusa. Esta condición viene de la dualidad entre los operadores U y R y de la semántica para la lógica LTL: una trayectoria infinita $\langle \sigma_0, \sigma_1, \dots \rangle$ es exitosa si *para toda* $i \geq 0$ tal que $\phi_1 U \phi_2 \in \sigma_i$ *existe* $j \geq i$ tal que $\phi_2 \in \sigma_j$ (se cumple ϕ_1 hasta cumplirse ϕ_2). De esta forma, los únicos ciclos que no anulan la validez de una derivación son los generados por la regla R_6.

Si se encuentra una derivación exitosa para una aserción, entonces dicha aserción se cumple. Sin embargo, para una misma aserción puede haber múltiples derivaciones.

El siguiente teorema nos asegura que no importa el orden en que se construya una derivación pues todas nos llevan al mismo resultado.

> **Teorema 1.** *([BCG95] Teorema 3.4): Sean M un estructura de Kripke, s un estado en M y V una derivación para la aserción $\sigma = s \vdash_M A(\phi)$. Se cumple $s \vDash_M A\phi$ sii V es una derivación exitosa para σ.*

Este teorema garantiza que basta encontrar una derivación para σ en la que todas sus trayectorias sean exitosas para determinar que σ es válida. También garantiza que si se encuentra una derivación exitosa cualquier otra derivación también será exitosa, y esto hace que no sea necesario hacer *backtracking* para encontrar una derivación que muestre la validez de una aserción. Además, garantiza que si se encuentra una trayectoria no exitosa en una derivación no es necesario seguir revisando la estructura de Kripke pues la aserción inicial no se cumple.

Como las derivaciones se pueden ver como gráficas dirigidas, esto permite utilizar resultados de la Teoría de Gráficas para encontrar rápidamente una derivación exitosa y para cancelar su construcción en cuanto se detecte que una aserción no se cumple. La siguiente definición es de utilidad para detectar trayectorias infinitas dentro de una derivación:

> **Definición 15.** Sea G una digráfica. $G' \subseteq G$ es una componente fuertemente conexa no trivial si G' es una subgráfica maximal de G y si para cualesquiera $v, w \in G'$, existe un camino dirigido de v a w y de w a v.

Las trayectorias infinitas dentro de una derivación son componentes fuertemente conexas no triviales, y con la siguiente definición es posible saber si representan una trayectoria infinita exitosa:

> **Definición 16.** Si C es un conjunto finito de aserciones, se define:
>
> $$Success(C) = \{\phi_1 R \phi_2 \mid \exists \sigma \in C \text{ tal que } \phi_1 R \phi_2 \in \sigma \wedge \forall \sigma' \in C, \phi_2 \notin \sigma'\}$$

Si V es una derivación y C es un conjunto de aserciones contenidas en V, decimos que C es un subconjunto exitoso de V si $Success(C) \neq \emptyset$, y en particular, si C es una componente fuertemente conexa basta usar esta definición para saber si es una trayectoria infinita exitosa.

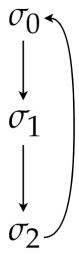
El siguiente teorema es otra forma de caracterizar a las derivaciones que son exitosas:

> **Teorema 2.** *([BCG95] Teorema 3.5): Una derivación V en la que cada nodo terminal es true es exitosa sii toda componente conexa no trivial en V es exitosa.*

Usando los ejemplos antes vistos podemos ver a la derivación:

$$\dfrac{\dfrac{\dfrac{\dfrac{s_2 \vdash_M A(Fp)}{s_2 \vdash_M A(p, XFp)}(R_F)}{s_2 \vdash_M A(XFp)}(R_2)}{s_2 \vdash_M A(Fp)}(R_7)}{\vdots}(R_F)$$

como la digráfica:

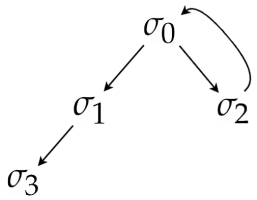


donde $\sigma_0 = s_2 \vdash_M A(Fp)$, $\sigma_1 = s_2 \vdash_M A(p, XFp)$ y $\sigma_2 = s_2 \vdash_M A(XFp)$. Es claro que $Success(\{\sigma_0, \sigma_1, \sigma_2\}) = \emptyset$ pues ninguna σ_i contiene una R-fórmula. Por tanto, esta derivación no es exitosa pues contiene una trayectoria infinita no exitosa.

Por otra parte, la derivación:

$$\dfrac{\dfrac{s_2 \vdash_M A(r)}{true}(R_1) \qquad \dfrac{\dfrac{s_2 \vdash_M A(Gr)}{s_2 \vdash_M A(XGr)}(R_G)}{s_2 \vdash_M A(Gr)}(R_7)}{\dfrac{\dfrac{s_2 \vdash_M A(r)}{true}(R_1) \qquad \dfrac{s_2 \vdash_M A(XGr)}{\vdots}(R_7)}{s_2 \vdash_M A(Gr)}(R_G)}(R_G)$$

puede verse como la digráfica:



donde $\sigma_0 = s_2 \vdash_M A(Gr)$, $\sigma_1 = s_2 \vdash_M A(r)$, $\sigma_2 = s_2 \vdash_M A(XGr)$ y $\sigma_3 = true$. Como el único nodo terminal es *true*, basta confirmar que la componente conexa $\{\sigma_0, \sigma_2\}$ es exitosa, y en efecto, la fórmula Gr está en σ_2 y r no está ni en σ_0 ni en σ_2.

Ahora disponemos de toda la teoría matemática necesaria para construir un eficiente algoritmo de verificación LTL al vuelo. Todas estas ideas culminan en combinar el uso de las reglas para construir una derivación mostradas en la figura 3.1 y el algoritmo de Tarjan [Tar72]. Dicho algoritmo sirve para detectar componentes fuertemente conexas mediante un recorrido a profundidad (DFS[1]). Esto último es de suma importancia pues hace que la derivación se construya por trayectorias.

El algoritmo de verificación LTL se describe informalmente de la siguiente manera:

1. Se aplica alguna de las reglas a la aserción inicial y se continúa haciendo DFS en las metas generadas con el algoritmo de Tarjan.

2. Si se alcanza un nodo terminal se hace lo siguiente:

   a) Si el nodo terminal es una aserción cuyo conjunto de fórmulas es vacío, la derivación no es exitosa y el algoritmo termina.

   b) Si el nodo terminal es *true*, se continua con las metas pendientes.

[1] Depth First Search

3. Si se encuentra una componente fuertemente conexa no trivial se hace lo siguiente:

   a) Si la componente no es exitosa, la derivación tampoco lo es y el algoritmo termina.

   b) Si la componente es exitosa, se continua con las metas pendientes.

4. Si no hay metas pendientes, la derivación es exitosa y el algoritmo termina.

En resumen, la derivación se construye mientras no se anule su validez, y si dicha validez se anula el algoritmo inmediatamente termina.

Para poder hacer uso del algoritmo de Tarjan, a cada nodo de la derivación se le agrega información extra:

- *dfsn*: un natural que etiqueta a cada nodo en el orden en que se va encontrando.
- *low*: un natural que indica cuál es el *dfsn* del antecesor más alejado y que puede ser alcanzado desde la aserción actual. Esto sirve para detectar las componentes fuertemente conexas.
- *valid*: Un conjunto de pares ordenados de la forma $(\phi_1 R \phi_2, sp)$, donde sp es un número natural. Este conjunto sirve para mostrar el éxito de una componente fuertemente conexa.

El algoritmo hace uso de una subrutina llamada *subgoals(σ)* que devuelve el conjunto de metas generadas al aplicar alguna de las reglas de la figura 3.1 a σ. Se utiliza un *stack* para apilar las metas generadas por *subgoals* y se hace uso de las operaciones usuales *push, pop* y *top*. Además, se utilizan dos conjuntos de aserciones: V para guardar las aserciones encontradas, y F para guardar las aserciones que son falsas.

El núcleo del algoritmo es la subrutina llamada *dfs* que se encarga de construir una derivación para una aserción σ siguiendo la estrategia antes descrita.

El pseudo-código con estilo imperativo del verificador LTL que se obtiene de lo mencionado con anterioridad aparece en el cuadro 3.1 y aparece originalmente en [BCG95].

El siguiente teorema nos garantiza que el algoritmo es correcto para hacer verificación de modelos LTL.

> **Teorema 3.** *([BCG95] Corolario 4.3) Si M es una estructura de Kripke, s un estado de M y ϕ una fórmula LTL,* modchkLTL*($s \vdash_M A(\phi)$) =* true *sii $s \models_M A\phi$.*

Por el lema 3.7 contenido en [BCG95], si V es una derivación para $s \vdash_M A(\phi)$, entonces $|V| < 2^{|\phi|} * |M|$. En el peor caso, la derivación generada por *modchkLTL* es maximal. De esta manera, la complejidad en tiempo del algoritmo es $O(|M| * 2^{O(|\phi|)})$. Esto hace que el algoritmo empate en desempeño con los mejores algoritmos para hacer verificación LTL [Cou+92]; [LP]; [VW86].

3.1.1. Extendiendo el verificador LTL a un verificador CTL*

Como se muestra en la definición 9, la sintaxis para la lógica CTL* hace recursión simultanea: las fórmulas de estado dependen de las fórmulas de trayectoria y viceversa. De esta observación se obtiene un algoritmo recursivo que extiende el verificador *modchkLTL* a un verificador CTL*: al analizar una fórmula de estado, si se encuentra

```
procedure modchkLTL(σ){
  var dfn:=0;
  var stack:=emptyStack();
  subroutine init(σ,valid){
    dfn:=dfn+1;
    σ.dfsn:=dfn;
    σ.low:=dfn;
    σ.valid:={(φ1Rφ2,sp) | φ2 ∉ σ ∧ (φ1Rφ2 ∈ σ ∨ X(φ1Rφ2) ∈ σ)∧
              if((φ1Rφ2,sp') ∈ valid) sp=sp' else sp=dfn};
  }
  subroutine dfs(σ,valid){
    var flag:=true;
    init(σ,valid);
    stack.push(σ);
    V:=V∪{σ};
    case subgoals(σ):
      ∅ → flag:=false;
      {true} → flag:=true;
      otherwise → foreach σ' ∈ subgoals(σ)
                  while flag{
                    if(σ' ∈ V){
                      if(σ' ∈ F) flag:=false;
                      else{
                        if(σ' ∈ stack){
                          σ.low:=min(σ.low,σ'.low);
                          σ.valid:={(φ1Rφ2,sp)∈ valid|sp≤ σ'.dfsn};
                          if(σ.valid=∅) flag:=false;
                        }
                      }
                    }else{
                      flag:=dfs(σ',σ.valid);
                      if(σ'.low≤ σ.dfsn){
                        σ.low:=min(σ.low,σ'.low);
                        σ.valid:=σ'.valid;
                      }
                    }
                  }
                  endforeach
    endcase
    if(σ.dfsn=σ.low){
      repeat{
        σ':=stack.top();
        stack.pop();
        if(flag=false) F:=F∪{σ'};
      }until(σ'=σ);
    }
    return flag;
  }
  return dfs(σ,∅);
}
```

CUADRO 3.1

```
procedure modchkCTL*(σ){
  subroutine update(b,σ){
    flag:=b;
    if(not b) F:=F∪{σ};
  }
  if(σ∈V){
    if(σ∈F) flag:=false else flag:=true
  }else{
    V:=V∪{σ};
    case σ:
    (s ⊢_M a) → update(a∈L(s),σ);
    (s ⊢_M ¬a) → update(a∉L(s),σ);
    (s ⊢_M φ1∧φ2) → update(modchkCTL*(s ⊢_M φ1) and modchkCTL*(s ⊢_M φ2),σ);
    (s ⊢_M φ1∨φ2) → update(modchkCTL*(s ⊢_M φ1) or modchkCTL*(s ⊢_M φ2),σ);
    (s ⊢_M Aφ) → update(modchkLTL(s ⊢_M A{φ}),σ);
    (s ⊢_M Eφ) → update(not modchkLTL(s ⊢_M A{neg_P(φ)}),σ);
    endcase
  }
  return flag;
}
```

CUADRO 3.2

una fórmula de trayectoria se invoca al verificador LTL, y si al estar verificando una fórmula de trayectoria se encuentra con una fórmula de estado, se regresa al verificador para CTL*.

En el cuadro 3.2 se muestra cómo se extender el verificador *modchkLTL* al verificador *modchkCTL**. Dicho cuadro aparece originalmente en [BCG95].

Los casos base para *modchkCTL** son cuando una fórmula de estado es una literal. Para el caso donde la fórmula es una conjunción o una disyunción de fórmulas de estado basta descomponer la fórmula y hacer una llamada recursiva en cada unas de las subfórmulas generadas. Para tratar a las fórmulas $A\phi$ se llama al verificador *modchkLTL*, y para las fórmulas $E\phi$ se hace uso de la equivalencia $E\phi \equiv \neg A \neg \phi$. Es importante notar que en la implementación del algoritmo la primer negación corresponde a la negación booleana del lenguaje de programación y la segunda corresponde a la función $\mathbf{neg}_{\mathcal{P}}$ mostrada en la figura 2.9.

Las reglas R_1 y R_2 deben modificarse para que el verificador LTL se pueda extender a CTL* de manera correcta, ya que ahora las fórmulas de trayectoria pueden contener subfórmulas de estado más complicadas que simples literales: podría haber varios cuantificadores de trayectoria anidados en una misma fórmula. Las modificaciones a las reglas se muestran en la figura 3.3.

El siguiente teorema garantiza que esta forma de extender el verificador LTL a un verificador CTL* corresponde a la semántica formal para CTL*:

> **Teorema 4.** *([BCG95] Teorema 5.1) Si M es una estructura, s un estado de M y φ una fórmula CTL**, modchkCTL**(s* $\vdash_M$ *φ) devuelve* true *sii s* $\models_M$ *φ.*

La demostración es directa haciendo inducción estructural sobre φ.

$$R_1 : \frac{s \vdash_M A(\varphi, \Phi)}{true}(modchkCTL^\star(s \vdash_M \varphi) = true)$$

$$R_2 : \frac{s \vdash_M A(\varphi, \Phi)}{s \vdash_M A\Phi}(modchkCTL^\star(s \vdash_M \varphi) = false)$$

FIGURA 3.3

La complejidad en tiempo de este nuevo verificador es $O(|M| * 2^{O(|\phi|)})$ [BCG95] pues en el peor caso se revisan fórmulas de trayectoria.

3.2. Haciendo que el verificador LTL sea más sencillo

El algoritmo *modchkLTL* no está basado en autómatas de Büchi, se extiende a CTL* y es al vuelo. Sin embargo, la implementación puede ser conceptualmente complicada cuando se combinan las ideas del algoritmo de Tarjan y las reglas para construir una derivación. Aunque en el cuadro 3.1 se da un pseudo-código bastante detallado hay varios aspectos técnicos que deben codificarse con mucho cuidado para evitar que el algoritmo tenga un comportamiento inesperado.

Las ideas antes presentadas pueden simplificarse para obtener un algoritmo más fácil de entender. Si bien el algoritmo *modchkLTL* trata las trayectorias infinitas como componentes fuertemente conexas éstas pueden verse como ciclos pues el número de estados en la estructura es finito y la relación de accesibilidad es total, lo que obliga a que se repitan estados.

Para simplificar las cosas, en lugar de ocupar un *stack* y dos conjuntos de aserciones, únicamente se utilizará una lista simplemente ligada para detectar ciclos. La estrategia a seguir es construir la derivación por trayectorias y continuar mientras no se anule su validez. Para esto, basta hacer la versión simple del recorrido DFS en la construcción de la derivación y no la versión del algoritmo de Tarjan. Esto hace que no sea necesario agregar información extra a los nodos de la derivación para detectar el éxito de una trayectoria infinita, pues basta con que se repita una aserción en la construcción de la derivación para saber dónde empieza y termina un ciclo. De esta manera, únicamente se necesita recordar la trayectoria que se está revisando. Esto simplifica las cosas de manera notable pues ya no necesitamos toda la maquinaria descrita en la implementación del algoritmo *modchkLTL* (cuadro 3.1). Además, esta idea sigue siendo al vuelo pues en cuanto se detecta una trayectoria que no es exitosa el algoritmo se detiene. Se usa la notación del lenguaje de programación Haskell para describir listas: la lista vacía se denota por `[]`, y la lista cuya cabeza es x y cola es xs se denota por $x : xs$. Así, el primer elemento de una lista aparece a la derecha y el último a la izquierda.

Dados σ una aserción y ℓ una lista de aserciones que representa la trayectoria que se está revisando, el algoritmo simplificado funciona de la siguiente manera:

1. Si $\sigma \in \ell = [\sigma_k, \sigma_{k-1}, \ldots, \sigma_0]$, se calcula el éxito de la lista $[\sigma_k, \sigma_{k-1}, \ldots, \sigma_i]$ tal que $\sigma_i = \sigma$, usando la función *Success* de la definición 16.

2. Si $\sigma \notin \ell$, se calcula $subgoals(\sigma)$:

   a) Si $subgoals(\sigma)$=`[`$true$`]`, se continúa.

b) Si *subgoals*(σ)=`[]`, el algoritmo termina pues se anuló la validez de la derivación.

c) En otro caso, por cada $\sigma' \in$ *subgoals*(σ) se continúa de forma recursiva con σ' y la lista $(\sigma : \ell)$.

El algoritmo simplificado y con estilo funcional se describe en el cuadro 3.3: Así, para saber si una fórmula ϕ se cumple a partir de un estado $s \in M$, basta pasar al algoritmo simplificado, *mcALTL*, la aserción $s \vdash_M A(\phi)$ y la lista vacía como entrada.

```
mcALTL(σ) = dfs(σ,[]) where
            dfs(σ,ℓ) = if σ ∈ ℓ then Success(σ:(takeWhile(σ ≠ _,ℓ)))
                       else case subgoals(σ) of
                                []→ false
                                [true]→ true
                                [σ_k,σ_{k-1},...,σ_0]→ ⋀_{i=0}^{k} dfs(σ_i,σ:ℓ)
```

CUADRO 3.3

La función *takeWhile*(P,ℓ) usada al inicio de la función *dfs* devuelve los primeros elementos de ℓ que cumplen P. Por ejemplo, al tomar $P(x) = x < 4$, la ejecución *takeWhile*(P(_),$[1,2,3,4,5]$) devuelve $[1,2,3]$.

En comparación con el algoritmo mostrado en el cuadro 3.1, este nuevo algoritmo es conceptualmente más sencillo de entender y de manejar.

El siguiente lema sirve para mostrar que el nuevo algoritmo es correcto.

> **Lema 3.** *Sea* $\ell = \langle \sigma_m, \sigma_{m-1}, \ldots, \sigma_0 \rangle$ *una lista de aserciones tal que* σ_{j+1} *es submeta de* σ_j *y* σ *es una submeta de* σ_m. *Entonces* dfs(σ_0,ℓ) = true *sii* $\langle \ldots, \sigma, \sigma_m, \sigma_{m-1}, \ldots, \sigma_0 \rangle$ *es una derivación exitosa.*

Demostración:
$\Longrightarrow$: Si *dfs*(σ_0,ℓ) = *true*, hay dos casos:

1. $\sigma \in \ell$

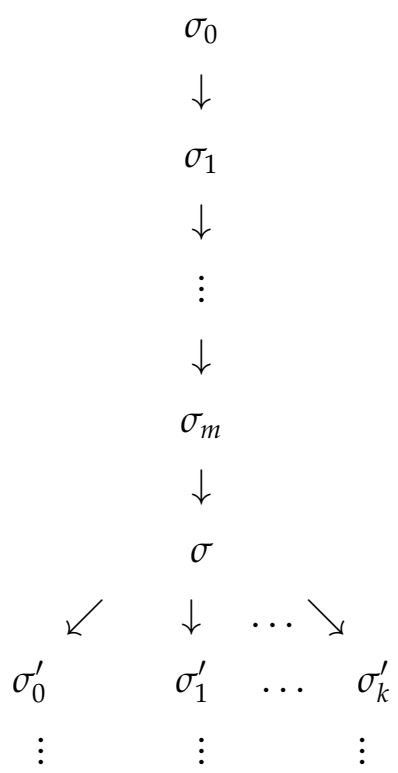
   Como *dfs*(σ_0,ℓ) = *true*, se tiene que *Success*(σ:(*takeWhile* ($\sigma \neq$) ℓ)) = *true*. Entonces existe $\sigma_i \in \ell$ tal que $\sigma_i = \sigma$. Por hipótesis, σ_{j+1} es submeta de σ_j, de esta manera σ:(*takeWhile*($\sigma \neq$ _,ℓ))=$\langle \sigma, \sigma_m, \sigma_{m-1}, \ldots, \sigma_{i+1} \rangle$ es un ciclo pues σ_{i+1} es submeta de σ, y así, $\langle \sigma, \sigma_m, \sigma_{m-1}, \ldots, \sigma_{i+1} \rangle$ es un ciclo que representa una trayectoria infinita exitosa. Por tanto, $\langle \ldots, \sigma, \sigma_m, \sigma_{m-1}, \ldots, \sigma_{i+1}, \sigma_i = \sigma, \ldots, \sigma_0 \rangle$ es una derivación exitosa pues la única trayectoria infinita que contiene es exitosa.

2. $\sigma \notin \ell$

   - *subgoals*(σ) = $\{true\}$
     Por definición, $\langle \sigma, \sigma_m, \sigma_{m-1}, \ldots, \sigma_0 \rangle = \langle true, \sigma_m, \sigma_{m-1}, \ldots, \sigma_0 \rangle$ es una trayectoria finita exitosa. De esta manera, $\langle \sigma, \sigma_m, \sigma_{m-1}, \ldots, \sigma_0 \rangle$ es una derivación exitosa pues la única finita que contiene es exitosa.

- $subgoals(\sigma)=\{\sigma'_k, \sigma'_{k-1}, \ldots, \sigma'_0\}$

  Como $dfs(\sigma_0,\ell) = true$, se tiene que $\left(\bigwedge_{i=0}^{k} dfs(\sigma'_i, \sigma : \ell)\right) = true$.



  Entonces $dfs(\sigma'_i, \sigma : \ell) = true$. Entonces cada trayectoria $\langle \ldots, \sigma'_i, \sigma_m, \ldots, \sigma_1, \sigma_0 \rangle$, es una trayectoria (finita o infinita) exitosa.

Por lo tanto, $\langle \ldots, \sigma, \sigma_m, \sigma_{m-1}, \ldots, \sigma_0 \rangle$ es una derivación exitosa.

$\Longleftarrow$: (Demostración por contrapositiva)

Si $dfs(\sigma,\ell) = false$ y $\sigma \in \ell$, significa que se detectó una trayectoria infinita que no es exitosa, y así, $\langle \ldots, \sigma, \sigma_m, \ldots, \sigma_0 \rangle$ no es una derivación exitosa. Por otra parte, si $\sigma \notin \ell$, al calcular las submetas de σ hay dos casos: que $subgoals(\sigma)=\emptyset$ ó que en alguna submeta σ'_i de σ se cumpla que $dfs(\sigma'_i,\sigma : \ell)=false$. En cualquier caso hay una trayectoria que no es exitosa dentro de la derivación $\langle \ldots, \sigma, \sigma_m, \ldots, \sigma_1, \sigma_0 \rangle$, y esto anula su éxito.

□

El siguiente teorema garantiza que el algoritmo simplificado es correcto:

Teorema 5. *Sea $\sigma = s \vdash_M A(\phi)$ una aserción.*
Entonces mcALTL*(σ)*=true *sii $s \vDash_M A\phi$.*

Demostración:

$$
\begin{aligned}
mcALTL(\sigma) = true &\iff dfs(\sigma, [\,]) = true && \text{[Def.]}\\
&\iff \text{cualquier derivación que empieza en } \sigma \text{ es exitosa} && \text{[Lema 3]}\\
&\iff s \vDash_M A\phi \quad \square
\end{aligned}
$$

Al igual que con el algoritmo *modchkLTL*, en el peor caso, se genera una derivación maximal pues cada aserción se visita, a lo más, una vez. De esta manera, la complejidad en tiempo del verificador simplificado también es $O(|M| * 2^{O(|\phi|)})$. Sin embargo, es conceptualmente más sencillo que el algoritmo *modchkLTL* al no utilizar el algoritmo de Tarjan.

3.2.1. Extendiendo el verificador *mcALTL* a CTL*

La extensión del algoritmo *mcALTL* a un algoritmo para hacer verificación de modelos CTL* es similar a la extensión del algoritmo *modchkLTL* al algoritmo *modchkCTL**. Basta modificar las reglas R_1 y R_2 de forma similar a la modificación mostrada en la figura 3.3 e invocar al verificador *mcALTL* de forma adecuada cada que se encuentre una fórmula de trayectoria.

El nuevo algoritmo para CTL* con estilo funcional queda descrito en el cuadro 3.4

$\texttt{mcCTL}^\star(s, \varphi)$ = **case** φ **of**
$a \rightarrow a \in L(s)$
$\neg a \rightarrow a \notin L(s)$
$\varphi_1 \wedge \varphi_2 \rightarrow$ $\texttt{mcCTL}^\star(s, \varphi_1)$ **and** $\texttt{mcCTL}^\star(s, \varphi_2)$
$\varphi_1 \vee \varphi_2 \rightarrow$ $\texttt{mcCTL}^\star(s, \varphi_1)$ **or** $\texttt{mcCTL}^\star(s, \varphi_2)$
$A\phi \rightarrow$ $\texttt{mcALTL}(s \vdash_M A(\phi))$
$E\phi \rightarrow$ **not** $\texttt{mcALTL}(s \vdash_M A(\mathbf{neg}_{\mathcal{P}}(\phi)))$

CUADRO 3.4

Es inmediato que este nuevo algoritmo es correcto pues se obtiene directamente de la semántica para CTL* (ver definición 10).

Aunque es conceptualmente más sencillo que el algoritmo *modchkCTL**, la complejidad de este algoritmo sigue siendo $O(|M| * 2^{O(|\phi|)})$ pues, en el peor caso, se revisan fórmulas de trayectoria. Se recalca que esta versión es más sencilla porque cada vez que se revisa una fórmula de trayectoria no se detectan componentes fuertemente conexas sino que únicamente se hace la versión sencilla del recorrido DFS.

3.3. Verificando a partir de un conjunto de estados iniciales

En la práctica, la verificación de modelos se realiza a partir de un conjunto de estados iniciales. Esto es, para que una estructura cumpla con una fórmula a partir de un conjunto de estados iniciales, cada estado inicial debe satisfacer dicha fórmula. Los algoritmos antes mostrados pueden extenderse fácilmente a un conjunto de estados iniciales usando adecuadamente el operador temporal X.

Para hacer verificación LTL a partir de un conjunto de estados iniciales con el verificador *mcALTL* se hace lo siguiente: dados $M = \langle S, R, L\rangle$ una estructura de Kripke y $S' \subseteq S$ un conjunto de estados, sea $\overline{s}$ un nuevo estado tal que $\overline{s} \notin S$, $\{(\overline{s}, s') \mid s' \in S'\} \subseteq R$ y $\forall s \in S, (s, \overline{s}) \notin R$. Así, para verificar una fórmula ϕ a partir del conjunto S' basta hacer *mcALTL*$(\overline{s} \vdash_M A(X\phi))$. El esquema de esta idea se muestra en la figura 3.4. De esta manera, $\overline{s} \vDash_M \phi$ sii $\forall s' \in S', s' \vDash_M \phi$. Es inmediato demostrar que esta forma de extender el

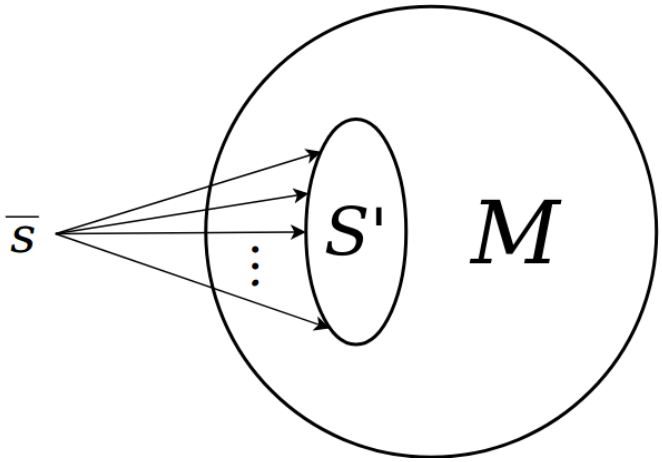


FIGURA 3.4

verificador es correcta porque, para que $\overline{s}$ cumpla con la fórmula ϕ, al aplicar la regla R_7 necesariamente cada sucesor de $\overline{s}$ debe cumplir ϕ, y esto es precisamente lo que se espera. Es importante observar que agregar el estado $\overline{s}$ a M no genera resultados erróneos pues $\overline{s}$ es inalcanzable en M.

Para verificar una fórmula de estado φ a partir del conjunto de estados S' basta hacer *mcCTL*$^\star(\overline{s}, AX\varphi)$. Análogamente, $\overline{s} \vDash_M \varphi$ sii $\forall s' \in S', s' \vDash_M \varphi$, pues con la entrada $(\overline{s}, AX\varphi)$, *mcCTL** llama al verificador *mcALTL* con la aserción $\overline{s} \vdash_M A(X\varphi)$, al aplicar la regla R_7 se llama a *mcALTL* con la aserción $s' \vdash_M A(\varphi)$, y al ser φ una fórmula de estado, de manera recursiva se llama a *mcCTL*$^\star(s', \varphi)$ con $s' \in S'$, que es precisamente lo esperado.

La segunda parte de este trabajo presenta la implementación de los algoritmos *mcALTL* y *mcCTL**. El lenguaje de programación elegido para dar vida a estos algoritmos es Haskell: un lenguaje puramente funcional con una sintaxis concisa que lleva a cabo sus cómputos de forma perezosa; es decir, un término no se evalúa a menos que sea necesario. Esta caracteristica combinada con la estrategia al vuelo del algoritmo para hacer verificación LTL hacen que la implementación de los verificadores sea eficiente. Todo esto se explica con lujo de detalle en el siguiente capítulo.

Parte II

Implementación en Haskell

Capítulo 4

Cómodo y eficiente: Haskell

Haskell is faster than C++, more concise than Perl, more regular than Python, more flexible than Ruby, more typeful than C#, more robust than Java, and has absolutely nothing in common with PHP.
Audrey Tang

Pure functional languages have this advantage: all flow of data is made explicit.
Philip Wadler

Wadler also observed that the methods of the Monad interface are sufficient to implement a notation based on the set comprehensions of Zermelo–Fraenkel set theory.
Jeremy Gibbons

Los algoritmos *modchkLTL* y *modchkCTL** propuestos en [BCG95], y que aparecen en los cuadros 3.1 y 3.2 respectivamente, tienen un estilo imperativo; pero las respectivas simplificaciones, *mcALTL* y *mcCTL** (que aparecen en los cuadros 3.3 y 3.4 respectivamente), se presentaron con estilo de programación funcional. Esto no fue casualidad pues el lenguaje de programación elegido para dar vida a los verificadores es Haskell: un lenguaje funcional (los términos del lenguaje son funciones matemáticas), puro (sin efectos en la memoria como asignación de variables) y con una sintaxis elegante y concisa. Otras características que lo hacen un lenguaje adecuado para implementar los verificadores simplificados son:

- Evaluación perezosa: Un término no se reduce a menos que sea necesario. Esta característica combinada con la estrategia de verificación al vuelo hacen que la implementación de los verificadores sea eficiente en la práctica.
- Además de ser un lenguaje fuertemente tipado, se pueden definir nuevos tipos con facilidad para hacer representaciones fáciles de manejar.
- Por su sintaxis sencilla y por ser un lenguaje sin efectos sobre la memoria, demostrar que un programa escrito en Haskell es correcto suele ser sencillo en comparación a otros lenguajes programación. Además, muchas veces un programa escrito de forma funcional resulta ser la especificación formal del algoritmo.

En resumen, Haskell es un lenguaje en el que el programador se enfoca más en qué hace un algoritmo y menos en cómo lo hace.

En este trabajo se utiliza la versión 8.4.3 del compilador GHC[1] de Haskell.

A continuación, se define todo lo necesario para la implementación de los verificadores.

[1] http://www.haskell.org/ghc/

4.1. Definiendo estructuras de Kripke

Como se ve en la definición 5, una estructura de Kripke es una terna: un conjunto finito $\mathcal{S}$ de estados, una relación R entre estados que es total, y una función L que etiqueta estados con fórmulas atómicas. Para nuestros propósitos y por simplicidad, los estados de un modelo se representan con números enteros y las fórmulas atómicas con cadenas. Esto no requiere definir nuevos tipos puesto que en Haskell ya hay enteros y cadenas, pero podemos crear un alias que nos haga entender mejor el comportamiento de una función. La palabra reservada *type* crea un alias para un tipo ya existente.

```
type State = Int
type At = String
```

Con esto, se define el tipo *KripkeS* para estructuras de Kripke:

```
data KripkeS = KS (State,State→[State],State→(At→Bool))
```

A diferencia de la palabra `type`, la palabra `data` sirve para crear nuevos tipos de dato. El constructor de tipos *KS* recibe una tupla con los componentes de una estructura de Kripke. La primera componente es un entero n que indica los estados que tiene la estructura de Kripke: $\{0, 1, \ldots, n\}$. La segunda componente es una función que asigna a cada estado su lista de sucesores. La tercera componente es una función que asigna a cada estado una función l que, dada $a \in At$, $(l\ s)\ a = true$ sii s está etiquetado con a. Por ejemplo, en la figura 4.1 se muestra una estructura de Kripke y su respectiva representación en Haskell.

Es importante señalar que, en Haskell, los nombres para variables comienzan estrictamente con minúscula y que las constantes, nombres de tipo y nombres para constructores comienzan con mayúscula.

4.2. Definiendo fórmulas LTL

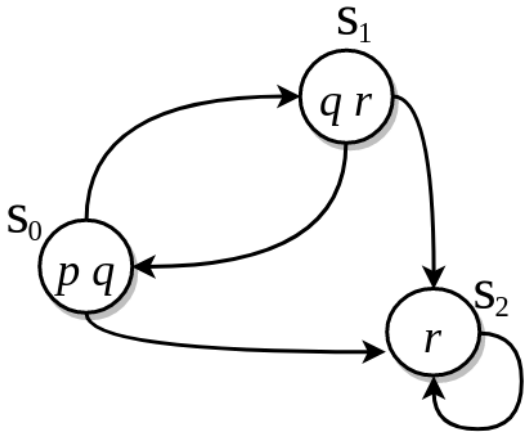
Una de las muchas bondades de Haskell son los *tipos opción*. Estos tipos permiten trasladar de forma natural la sintaxis de fórmulas LTL dentro del lenguaje. El siguiente tipo representa fórmulas LTL dentro de Haskell y corresponde a la definición 1.

```
data PathF = Var At |
             Neg At |
             DisyP PathF PathF |
             ConjP PathF PathF |
             U PathF PathF |
             R PathF PathF |
             X PathF deriving (Eq,Ord)
```

Al final de la definición aparece '*deriving (Eq,Ord)*', esto significa dos cosas:

1. El tipo *PathF* pertenece a la clase *Eq* de forma natural, es decir, podemos usar el operador == para distinguir fórmulas LTL.

2. El tipo *PathF* pertenece a la clase *Ord* de forma natural, es decir, hay un orden total para términos de tipo *PathF*, por lo que pueden usarse los operadores '<' y '>' entre fórmulas LTL. Dicho orden es el orden explícito de la definición.




```
m = KS (2,r,l) where
    r s = case s of
        0 → [1,2]
        1 → [0,2]
        2 → [2]
    l s = case s of
        0 → λa → case a of
                   p → True
                   q → True
                   _ → False
        1 → λa → case a of
                   q → True
                   r → True
                   _ → False
        2 → λa → case a of
                   r → True
                   _ → False
```

FIGURA 4.1

Literales < Disyunciones < Conjunciones < U-fórmulas < R-fórmulas < X-fórmulas

La gramática que define fórmulas LTL sólo niega fórmulas atómicas, por lo que es necesario implementar una función para negar fórmulas en general. En la figura 4.2 se muestra la función para negar fórmulas LTL correspondiente a la función del lema 1.

Haskell permite utilizar símbolos en unicode (como 'ϕ', 'Φ', 'σ',...) para hacer el código más legible y elegante.

Las constantes $\bot$ y $\top$ se definen en la figura 4.3. La constante $\bot$ es la cadena vacía y la constante $\top$ se define de forma dual.

Los operadores temporales F y G se definen en la figura 4.4 y corresponden a la definición 3. Además, se utilizan algunas de las equivalencias que aparecen en la figura 2.6 para simplificar fórmulas al momento de verificar (por ejemplo, $FF\phi \equiv F\phi$).

```
negP::PathF→PathF
negP φ = case φ of
          Var a → Neg a
          Neg a → Var a
          ConjP φ1 φ2 → DisyP (negP φ1) (negP φ2)
          DisyP φ1 φ2 → ConjP (negP φ1) (negP φ2)
          X φ' → X (negP φ')
          U φ1 φ2 → R (negP φ2) (negP φ2)
          R φ1 φ2 → U (negP φ1) (negP φ2)
```

FIGURA 4.2

```
bot = Var ""
top = Neg ""
```

FIGURA 4.3

La definición del operador implicación dentro del lenguaje se define como sigue:

```
impP φ1 φ2 = if φ1==φ2 then top else DisyP (negP φ1) φ2
```

Esta forma de definir la implicación hace que el verificador no haga trabajo de más, por ejemplo, cuando $\phi_1 = \phi_2$, $\phi_1 \rightarrow \phi_2 = \phi_1 \rightarrow \phi_1 \equiv \top$.

4.3. Definiendo aserciones

Como se vio en la definición 11, una aserción $s \vdash_M A\Phi$ esencialmente es el par (s, Φ) donde s es un estado y Φ es un conjunto de fórmulas LTL. Es importante señalar que al ser Φ un conjunto, sus elementos no se repiten, por lo que una lista no es conveniente para su representación.

El tipo de dato *Set*, que está contenido en la biblioteca *Data*, es el indicado para el manejo de aserciones. Esta implementación representa conjuntos de forma muy eficiente utilizando árboles binarios balanceados [NR73]; [Ada93]. Para construir un *Set a*, es necesario que el tipo *a* pertenezca a la clase *Ord*. Como se vio previamente, el tipo *PathF* deriva de la clase *Ord*. Además de no repetir elementos, un *Set* mantiene sus elementos ordenados de menor a mayor.

La definición de aserciones dentro del lenguaje se muestra a continuación:

```
data Assertion = Assrt (State,Set PathF) deriving Eq
```

Al hacer que el tipo *Assertion* derive de la clase *Eq*, dos aserciones pueden compararse con el operador '==', lo cual es indispensable para detectar ciclos en una derivación: dos aserciones (s_1,Φ_1) y (s_2,Φ_2) son iguales sii $s_1 = s_2$ y $\Phi_1 = \Phi_2$; esto es posible porque los tipos *Int* y *Set PathF* forman parte de la clase *Eq*.

En la figura 4.5 se definen dos operaciones: insertar y borrar una fórmula de una aserción. Ambas funciones son bastante eficientes, ya que borrar e insertar un elemento en un conjunto implementado como un *Set* de Haskell tiene desempeño $O(log(n))$.

```
opF φ = case φ of
        U (St (Neg "")) φ' → opF φ'
        R (St (Var "")) (U (St (Neg "")) φ')→ opG (opF φ')
        _ → U (St top) φ

opG φ = case φ of
        R (St (Var "")) φ' → opG φ'
        U (St (Neg "")) (R (St (Var "")) φ')→ opF (opG φ')
        _ → R (St bot) φ
```

FIGURA 4.4

```
deleteF::PathF→Assertion→Assertion
deleteF φ (Assrt (s,Φ)) = Assrt (s,delete φ Φ)

insertF::PathF→Assertion→Assertion
insertF φ (Assrt (s,Φ)) = Assrt (s,insert φ Φ)
```

FIGURA 4.5

4.4. Definiendo la función *subgoals*

Al aplicar las reglas para construir una derivación hay dos posibilidades para las metas generadas:

- La constante *true*
- Un conjunto de aserciones $\{\sigma_0, \sigma_1, \ldots, \sigma_k\}$

El tipo para representar las submetas generadas se muestra a continuación:

data `Subgoals = T | Subg [Assertion]`

El constructor *T* representa la constante *true*, y el conjunto de aserciones $\{\sigma_0, \sigma_1, \ldots, \sigma_k\}$ se representa por *Subg* $[\sigma_0, \sigma_1, \ldots, \sigma_k]$.

En la figura 4.6 se muestra la definición de la función *subgoals*, la cual, se basa en la definición 3.1.

Hay varias cosas que comentar sobre esta función [2]:

- El primer argumento es una estructura de Kripke que aparece en repetidas ocasiones a lo largo de la definición de la función *subgoals*.
- Si el conjunto de fórmulas de una aserción es vacío, la derivación falla. De esta forma, cuando se detecta una aserción cuya segunda componente es el conjunto vacío se devuelve la lista de submetas *Subg* `[]`.

[2]En Haskell, el operador "." denota la composición de funciones. El operador "$" ayuda a ahorrar paréntesis, pues asocia a la derecha la aplicación de una función; por ejemplo, la expresión "$f\ \$\ g\ \$\ x + 4 * y$" equivale a la expresión "$f\ (g\ (x + 4 * y))$". Además, Haskell permite ligar un término a una variable con el símbolo «@», por lo que la estructura "*KS (_,r,l)*" está ligada a la variable "*ks*". Cuando el argumento de una función es irrelevante se reemplaza por un guión bajo.

```
subgoals::KripkeS→Assertion→Subgoals
subgoals ks@(KS (_,r,l)) σ@(Assrt (s,Φ)) =
  if Φ == empty then Subg [] else
  let φ = elemAt 0 Φ in case φ of
    Var a → if (l s a) then T else Subg [deleteF φ σ]
    Neg a → if ((not . l s) a) then T else subgoals ks $ deleteF φ σ
    DisyP φ1 φ2 → Subg [insertF φ1 $ insertF φ2 $ deleteF φ σ]
    ConjP φ1 φ2 → if φ1==φ2 then Subg [insertF φ1 $ deleteF φ σ] else
                     if φ1 < φ2 then
                Subg [insertF φ1 $ deleteF φ σ,insertF φ2 $ deleteF φ σ]
                     else
                Subg [insertF φ2 $ deleteF φ σ,insertF φ1 $ deleteF φ σ]
    U φ1 φ2 → if φ1==φ2 then Subg [insertF φ1 $ deleteF φ σ] else
                Subg [insertF φ1 $ insertF φ2 $ deleteF φ σ,
                              insertF φ2 $ insertF (X φ) $ deleteF φ σ]
    R φ1 φ2 → if φ1==φ2 then Subg [insertF φ1 $ deleteF φ σ] else
                Subg [insertF φ2 $ deleteF φ σ,
                             insertF φ1 $ insertF (X φ) $ deleteF φ σ]
    X _ → let Φ' = Data.Set.map (λ(X φ) → φ) Φ' in
          Subg [Assrt (s',Φ') | s' ← r s]
```

FIGURA 4.6

- Como se vio antes, las fórmulas dentro de una aserción están ordenadas así:

  Literales $<$ Disyunciones $<$ Conjunciones $<$ U-fórmulas $<$ R-fórmulas $<$ X-fórmulas

  Revisar en este orden las fórmulas hace eficiente el verificador por lo siguiente:

Literales: Revisar si un estado está etiquetado con una fórmula atómica es inmediato, ya que si un estado está etiquetado con a se devuelve T; en caso contrario, la fórmula se deshecha del conjunto de fórmulas.

Disyunción: Revisar disyunciones antes que conjunciones no crea nuevas trayectorias.

Conjunciones: Para aprovechar que las fórmulas *PathF* se pueden comparar se hace lo siguiente:

- Si $\phi_1 = \phi_2$, basta revisar ϕ_1.
- Si $\phi_1 < \phi_2$, primero se revisa la submeta que se obtiene de ϕ_1 y luego a la que se obtiene de ϕ_2. El caso $\phi_1 > \phi_2$ es análogo.

U-fórmulas: Revisar las U-fórmulas antes que las R-fórmulas hace que las trayectorias infinitas no exitosas se detecten antes que las trayectorias infinitas exitosa. Cuando se revisa una fórmula $\phi_1 U \phi_2$ en la que $\phi_1 = \phi_2$, sólo se revisa ϕ_1.

R-fórmulas: De forma análoga, cuando se revisa una fórmula $\phi_1 R \phi_2$ en la que $\phi_1 = \phi_2$, sólo se revisa ϕ_1.

X-fórmulas: Al haber descartado los casos anteriores, forzosamente el conjunto de fórmulas de la aserción que se está revisando únicamente contiene X-fórmulas, lo que permite aplicar la regla R_7. El poder de las *listas por comprensión* (inspiradas en los conjuntos por comprensión $\{x \mid x \in A, P(x)\}$, donde P es un predicado) permiten

calcular las submetas de forma sencilla y concisa. Al escribir $\lambda x \to y$ nos estamos refiriendo a la función anónima $x \mapsto y$, y en este caso, $\lambda(X\phi) \to \phi$ con $X\phi \mapsto \phi$. Por último, la función *Data.Set.map* recibe una función f, un conjunto A y devuelve el conjunto $\{f\ x \mid x \in A\}$.

Las bondades del lenguaje hacen que el código sea fácil de entender y que probar su correctud sea inmediato. Además de ser un código legible es una implementación bastante eficiente pues todas las operaciones hechas en el conjunto de fórmulas de la aserción de entrada tienen desempeño logarítmico.

4.5. Definiendo la función *check_success*

Para determinar el éxito de una trayectoria infinita basta probar que es regenerada infinitamente por una R-fórmula. Una vez detectado un ciclo $V = \{\sigma_0, \sigma_1, \ldots, \sigma_{k-1}, \sigma_k, \sigma_0\}$, donde σ_{i+1} es submeta de σ_i, basta decidir si $Success(V) \neq \emptyset$, donde *Success* es la función de la definición 16.

La función *check_success* se muestra a continuación:

```
check_success::[Assertion]→Bool
check_success V =
          let φs = (nub . concat) [toList Φ | Assrt (_,Φ)← V] in
          (not . null) [R φ1 φ2 | R φ1 φ2 ← φs, (not . elem φ2) φs]
```

La bondad del lenguaje permite que la función *check_success* sea una transliteración de la función *Success*.

En el próximo capítulo se implementan los verificadores simplificados para hacer verificación de modelos LTL y CTL*, *mcALTL* y *mcCTL**, respectivamente.

Capítulo 5

Implementando los verificadores de modelos *mcALTL* y *mcCTL**

Simplicity is prerequisite for reliability.
Edsger W. Dijkstra

Algorithms are the computational content of proofs.
Benjamin C. Pierce

There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.
Charles Antony Richard Hoare

5.1. Implementando el verificador *mcALTL*

Como se vio en la sección 3.1, el verificador *modchkLTL* originalmente dado por Bhat, Cleaveland y Grumberg no está basado en autómatas de Büchi y se extiende fácilmente a CTL* pero su implementación puede ser complicada por la forma en la que emplea el algoritmo de Tarjan para detectar componentes fuertemente conexas.

El algoritmo *mcALTL*, que aparece en el cuadro 3.3, es una simplificación del algoritmo *modchkLTL* (que aparece en el cuadro 3.1) pues sigue la misma idea de construir una derivación por trayectorias pero sin utilizar el algoritmo de Tarjan para detectar trayectorias infinitas. El algoritmo *mcALTL* únicamente hace un recorrido DFS sencillo sin tener que agregar información extra a cada aserción encontrada.

La implementación del verificador *mcALTL* en Haskell se muestra en el cuadro 5.1.

Siendo *ks* una estructura de Kripke y σ una aserción, se construye una derivación para σ como sigue:

- Si la aserción σ ya fue visitada, se calcula el éxito del ciclo que empieza y termina en σ con la función *check_success*.
- En caso contrario, se aplica alguna regla a σ y se analizan las submetas generadas:
  - Si la única submeta generada es T, se devuelve *True*.
  - En otro caso, se genera una lista de aserciones σs. Si $\sigma s =$ `[]`, entonces la trayectoria que se está revisando no es exitosa y se devuelve *False*. En caso contrario, se aplica de forma recursiva *dfs* en las submetas generadas pero añadiendo a σ en la lista *stack* de aserciones visitadas.

```
mcALTL::KripkeS→Assertion→Bool
mcALTL ks σ = dfs ks σ [] where
     dfs ks σ stack = if elem σ stack
                      then check_success (σ : takeWhile (σ ≠) stack)
                      else case subgoals ks σ of
                           T → True
                           Subg σs → case σs of
                                [] → False
                                _ → and [dfs ks σ' (σ:stack) | σ' ← σs]
```

CUADRO 5.1

En pocas líneas queda descrito el verificador *mcALTL*. Es inmediato que esta implementación es correcta pues es una transliteración del algoritmo descrito en el cuadro 3.3. Al igual que el verificador *modchkLTL*, este verificador se extiende fácilmente a un verificador para la lógica CTL*.

5.2. Implementando el verificador *mcCTL**

En la definición 9 se muestra la sintaxis para la lógica CTL*. La respectiva definición de esta sintaxis se describe en la figura a continuación:

```
data StateF = Var At | Neg At
              ConjS StateF StateF |
              DisyS StateF StateF |
              A PathF |
              E PathF deriving (Eq,Ord)

data PathF = St StateF |
             DisyP PathF PathF |
             ConjP PathF PathF |
             U PathF PathF | R PathF PathF |
             X PathF deriving (Eq,Ord)
```

Dado que ahora las fórmulas de estado y las fórmulas de trayectoria hacen recursión simultanea, es necesario modificar la función *subgoals* para que se comporte como se describe en la figura 3.3. Dicha modificación se muestra en la figura 5.1.

El verificador para la lógica CTL*, *mcCTL**, se muestra el cuadro 5.2. Para las fórmulas atómicas simplemente se revisan las etiquetas del estado que se está revisando. Para el caso de la conjunción y la disyunción de fórmulas de estado se hace recursión, y para el caso de una fórmula de trayectoria cuantificada se invoca al verificador *mcALTL* adecuadamente. La función *singleton* crea un conjunto unitario. Al igual que la sintaxis y la semántica para CTL*, los verificadores *mcALTL* y *mcCTL** hacen recursión simultanea. Es necesario modificar la función $neg_{\mathcal{P}}$ para que niegue adecuadamente las fórmulas de estado como se muestra en la figura 2.9.

El verificador *mcCTL** es correcto al obtenerse directamente de la semántica formal para CTL* y porque es una transliteración del algoritmo mostrado en el cuadro 3.4.

```
subgoals::KripkeS→Assertion→Subgoals
subgoals ks@(KS (_,r,_)) σ@(Assrt (s,Φ)) =
  if Φ == empty then Subg [] else
  let φ = elemAt 0 Φ in case φ of
    St φ → if mcCTL* (ks,s) φ then T else Subg [deleteF φ σ]
    DisyP φ1 φ2 → Subg [insertF φ1 $ insertF φ2 $ deleteF φ σ]
    ConjP φ1 φ2 → if φ1==φ2 then Subg [insertF φ1 $ deleteF φ σ] else
                     if φ1 < φ2 then
                  Subg [insertF φ1 $ deleteF φ σ,insertF φ2 $ deleteF φ σ]
                     else
                  Subg [insertF φ2 $ deleteF φ σ,insertF φ1 $ deleteF φ σ]
    U φ1 φ2 → if φ1==φ2 then Subg [insertF φ1 $ deleteF φ σ] else
                  Subg [insertF φ1 $ insertF φ2 $ deleteF φ σ,
                                insertF φ2 $ insertF (X φ) $ deleteF φ σ]
    R φ1 φ2 → if φ1==φ2 then Subg [insertF φ1 $ deleteF φ σ] else
                  Subg [insertF φ2 $ deleteF φ σ,
                               insertF φ1 $ insertF (X φ) $ deleteF φ σ]
    X _ → let Φ' = Data.Set.map (λ(X φ) → φ) Φ' in
          Subg [Assrt (s',Φ') | s' ← r s]
```

FIGURA 5.1

```
mcCTL*::(KripkeS,State)→StateF→Bool
mcCTL* (ks@(KS (_,_,l)),s) φ = case φ of
    Var a → l s a
    Neg a → (not . l s) a
    ConjS φ1 φ2 → mcCTL* (ks,s) φ1 && mcCTL* (ks,s) φ2
    DisyS φ1 φ2 → mcCTL* (ks,s) φ1 || mcCTL* (ks,s) φ2
    A φ → mcALTL ks (Assrt (s,singleton φ))
    E φ → (not . mcALTL ks) (Assrt (s,(singleton . negP) φ))
```

CUADRO 5.2

5.3. *mcALTL* a partir de un conjunto de estados iniciales

Como se vio en la sección 3.3, es posible verificar una estructura de Kripke a partir de un conjunto de estados iniciales haciendo una pequeña modificación a la relación de accesibilidad.

La función para alterar la relación de accesibilidad de una estructura de Kripke se muestra a continuación:

```
upd_r::KripkeS→State→[State]→KripkeS
upd_r (KS (n,r,l)) s ss = KS (n,λs' → if s'==s then ss else r s',l)
```

Esta función reemplaza los sucesores de un estado s por la lista de estados ss. Es importante recordar que, en esta forma de representar una estructura de Kripke, la primera componente es un entero n que indica que los estados del modelo son $\{0 \ldots n\}$, la segunda componente r es la relación de accesibilidad y l es la función de etiquetamiento, por lo que al modificar r, ni n ni l se modifican.

La forma de hacer verificación LTL a partir de un conjunto de estados iniciales se muestra a continuación:

```
mcALTL_set::KripkeS→[State]→PathF→Bool
mcALTL_set ks ss φ = let ks' = upd_r ks (-1) ss in
                     mcALTL ks' (Assrt (-1,singleton $ X φ))
```

Para verificar una fórmula ϕ a partir del conjunto ss, basta verificar $(X\,\phi)$ a partir del estado -1, cuyos sucesores son ss. Es claro que el estado -1 no está en ks y que ningún estado en ks tiene como sucesor al estado -1. Es inmediato probar que esta forma de hacer la verificación es correcta pues se obtiene de la semántica para LTL.

5.4. *mcCTL** a partir de un conjunto de estados iniciales

De forma similar, para verificar una fórmula de estado φ a partir de un conjunto de estados iniciales ss, basta verificar $-1 \vDash AX\varphi$, donde los sucesores de -1 son ss. La respectiva implementación se muestra a continuación:

```
mcCTL*_set::(KripkeS,[State])→StateF→Bool
mcCTL*_set (ks,ss) φ = let ks' = upd_r ks (-1) ss in
                       mcCTL* (ks',-1) (A $ X $ St φ)
```

Esta implementación es correcta pues se obtiene directamente de la semántica para la lógica CTL*.

5.5. Experimentos y comparación de rendimiento con NuSMV

La implementación de los verificadores *mcALTL*, mcCTL* y todo lo necesario para reproducir los experimentos de esta sección se encuentran en el siguiente sitio:

https://github.com/spidermoy/Model_Checking-LTL-CTLS

Requisitos:

- GHC (The Glasgow Haskell Compiler). En este trabajo se utilizó la versión 8.4.3.
- Haskell Cabal
- Una vez instalado Cabal es necesario instalar la biblioteca *Random* tecleando:

```
cabal install random
```

- NuSMV (no es necesario instalarlo si no desea comparar desempeño).

El código se organizó en módulos de la siguiente manera:

Core: Contiene todo lo necesario para implementar los verificadores de modelos: la definición de estructura de Kripke, fórmulas de trayectoria, fórmulas de estado, entre otras. También contiene a los verificadores *mcALTL* y *mcCTL** que sirven para hacer verificación LTL y CTL*, respectivamente.

RandomForms: Para generar fórmulas LTL y CTL aleatorias. Dados n y m, se crean fórmulas aletatorias con n variables y cuyo árbol de parseo tiene profundidad m.

RandomKS: Para generar estructuras de Kripke de forma aleatoria. Dado un entero n, se crea un modelo aleatorio con 2^n estados. A cada estado se le asigna de forma aletatoria una lista de sucesores y k etiquetas, con $0 \leq k \leq n$.

ParserForms: Para traducir fórmulas LTL y CTL a fórmulas compatibles con NuSMV.

ParserNuXmv: Para transformar una estructura de Kripke y fórmulas LTL o CTL en un módulo para NuSMV.

Exp_LTL: Para generar experimentos LTL.

Exp_CTL: Para generar experimentos CTL.

InputNuSMV: Para recibir como entrada un archivo de NuSMV. [1]

Main: El módulo principal donde se corre el proyecto.

El modo de uso es el siguiente:

1. Se compila el código tecleando: `ghc -O2 Main.hs`
2. El programa puede correrse de la siguiente manera:
   - Para correr un experimento aleatorio LTL:
     `./Main random LTL num_vars length_forms`
   - Para correr un experimento aleatorio LTL y comparar los resultados con NuSMV:
     `./Main random nusmv LTL num_vars length_forms`
   - Para correr un experimento aleatorio CTL:
     `./Main random CTL num_vars length_forms`
   - Para correr un experimento aleatorio CTL y comparar los resultados con NuSMV:
     `./Main random nusmv CTL num_vars length_forms`
   - Para correr un experimento LTL con semillas generadoras:
     `./Main seeds ranInit ranNumInit ranKS ranF LTL num_vars length_forms`
   - Para correr un experimento LTL con semillas generadoras y comparar los resultados con NuSMV:
     `./Main seeds ranInit ranNumInit ranKS ranF nusmv LTL num_vars length_forms`
   - Para correr un experimento CTL con semillas generadoras:
     `./Main seeds ranInit ranNumInit ranKS ranF CTL num_vars length_forms`

[1] Este módulo funciona gracias al trabajo de Henning Günther publicado en https://github.com/hguenther/language-nusmv

- Para correr un experimento CTL con semillas generadoras y comparar los resultados con NuSMV:

  `./Main seeds` *`ranInit ranNumInit ranKS ranF`* `nusmv` *`CTL num_vars length_forms`*

  donde *num_vars* = número de variables del utilizadas y *length_forms* = la lóngitud de las fórmulas.
- Para correr *n* experimentos LTL de forma automática:

  `./Main tabla` *`n`* `LTL`
- Para correr *n* experimentos CTL de forma automática:

  `./Main tabla` *`n`* `CTL`
- Para recibir como entrada un archivo de NuSMV:

  `./Main input` *`filePath`*

  Un ejemplo del formato admitido por el verificador es:

```
------------------------------
MODULE main

VAR
p: boolean;
q: boolean;
r: boolean;

DEFINE
s0:= p & q & !r;
s1:= !p & q & r;
s2:= !p & !q & r;

INIT
s0 | s1;

TRANS
(s0 & next(s1|s2))
| (s1 & next(s0|s2))
| (s2 & next(s2))

LTLSPEC
G !(p&r)

LTLSPEC
(G F p) ->(G F r)

CTLSPEC
AX AF q
------------------------------
```

Algunos ejemplos son:

- `./Main random LTL 4 3`
- `./Main random nusmv CTL 4 3`
- `./Main seeds 100 76 4 0 nusmv CTL 10 2`
- `./Main tabla 5 CTL`

A continuación se muestran varios experimentos significativos donde se comparó el rendimiento de los verificadores *mcALTL* y *mcCTL** con el verificador NuSMV. Al inicio de cada experimento se encuentra el comando que permite reproducirlo. Los experimentos se realizaron con una computadora portátil ASUS E402SA que tiene el procesador Intel(R) Celeron(R) N3050 a 1.60GHz y 2Gb de memoria RAM.

5.5.1. Experimentos LTL

1. Este primer experimento fue con un modelo pequeño:

```
./Main seeds -505342864493949445 8 553379102978463915 3 -7168765502673159231 -7141886416844860059 nusmv LTL 5 2

MODELO ALEATORIO DE TAMAÑO 2^5
Profundidad de las fórmulas: 2
Estados iniciales: 6

    mcALTL:
(p1∧¬p2)∨(p3∨p4) : True

(Xp1)∨(Gp3) : False

G(Fp3) : False

Tiempo de verificación: 0.014113057s

    NuSMV:
-- specification ((p1 & !p2) | (p3 | p4))  is true
-- specification ( X p1 |  G p3)  is false
-- specification  G ( F p3)  is false

Tiempo de verificación: 0.173221146s
```

2. En este experimento, tanto *mcALTL* como NuSMV prácticamente tienen el mismo desempeño:

```
./Main seeds 7295322385161293443 9119911361258526469 -3976311874439943576 -8020079916767466332 nusmv LTL 7 3

MODELO ALEATORIO DE TAMAÑO 2^7
Profundidad de las fórmulas: 3
Estados iniciales: 113

    mcALTL:
((G¬p0)U(Fp3))∧((p1Rp3)R(Fp5)) : False

G(F(p3U¬p0)) : False

G((¬p0Up4)R(G¬p6)) : False

Tiempo de verificación: 0.039594538s

    NuSMV:
-- specification ((( G !p0) U ( F p3)) & !(!(!(!p1 U !p3)) U !( F p5)))  is false
-- specification  G ( F (p3 U !p0))  is false
-- specification  G !(!(!p0 U p4) U !( G !p6))  is false

Tiempo de verificación: 1.02512283s
```

3. Otro experimento con un modelo de tamaño 2^7 pero en el que algunos experimentos dieron *True*:

```
./Main seeds -4936023892743165299 450223482440517 10 -318369156885917136 7664302682932608404 nusmv LTL 7 2
```

```
MODELO ALEATORIO DE TAMAÑO 2^7
Profundidad de las fórmulas: 2
Estados iniciales: 4

    mcALTL:
F(p4∨p5) : True

F(p5∨p6) : True

G(p2Up6) : False

Tiempo de verificación: 0.034615194s

    NuSMV:
-- specification  F (p4 | p5)  is true
-- specification  F (p5 | p6)  is true
-- specification  G (p2 U p6)  is false

Tiempo de verificación: 0.605554901s
```

4. El siguiente experimento muestra que el verificador *mcALTL* puede consumir menos recursos por ser al vuelo ya que, en cuanto se detecta que el modelo no cumple con una fórmula automáticamente devuelve *False*. Mientras que NuSMV utilizó 264Mb, el verificador *mcALTL* sólo utilizó 9Mb. Además, el verificador *mcALTL* tardó milisegundos en verificar las fórmulas, mientras que NuSMV tardó 6 segundos:

```
./Main seeds -2600085101192994995 5103933280507166601 7321885015983263168 -3980215730978928173 nusmv LTL 9 3

MODELO ALEATORIO DE TAMAÑO 2^9
Profundidad de las fórmulas: 3
Estados iniciales: 477

    mcALTL:
((¬p3Rp7)∨(¬p4Rp8))∨(G(¬p3Rp7)) : False

G((X¬p1)R(¬p1Rp5)) : False

(G¬p3)R((¬p4∧p0)∨(Gp0)) : False

Tiempo de verificación: 0.181191898s

    NuSMV:
-- specification ((!(!(!p3) U !p7) | ... |  G !(!(!p3) U !p7))  is false
-- specification  G !(!( X !p1) U !(!(!(!p1) U !p5)))  is false
-- specification !(!( G !p3) U !((!p4 & p0) |  G p0))  is false

Tiempo de verificación: 5.367861722s
```

5. El verificador NuSMV tardó 13 segundos en revisar las tres fórmulas y utilizó 633Mb. En cambio, el verificador *mcALTL* únicamente utilizó 41Mb y le tomó milisegundos revisar las fórmulas:

```
./Main seeds -76398304999172336 -8563057924574850918 5654463647766606429 3977323623621745794 nusmv LTL 10 3

MODELO ALEATORIO DE TAMAÑO 2^10
Profundidad de las fórmulas: 3
Estados iniciales: 647

    mcALTL:
((p9∧p5)∧(Fp9))∨((F¬p4)∧(X¬p0)) : False
```

```
(G(¬p0∨p1))U((¬p0∨p1)U(Gp6)) : False

X((Fp6)∨(¬p3∧p9)) : False

Tiempo de verificación: 0.557211603s

    NuSMV:
-- specification (((p9 & p5) &  F p9) | ( F !p4 &  X !p0))  is false
-- specification (( G (!p0 | p1)) U ((!p0 | p1) U ( G p6)))  is false
-- specification  X ( F p6 | (!p3 & p9))  is false

Tiempo de verificación: 13.997534672s
```

6. Mientras el verificador *mcALTL* tardó casi un segundo en revisar cada fórmula y utilizó 84Mb, NuSMV utilizó 1.5Gb y tardó casi un minuto en revisar las fórmulas:

```
./Main seeds 2080258398381246525 -5652026877346931720 -6901248074493425579 -5647506115212334317 nusmv LTL 11 3

MODELO ALEATORIO DE TAMAÑO 2^11
Profundidad de las fórmulas: 3
Estados iniciales: 234

    mcALTL:
(X(Gp0))∧(G(Fp7)) : False

F((p2∨p3)∧(p9∨¬p10)) : False

(X(Gp8))U(X(¬p10∧p6)) : False

Tiempo de verificación: 1.272414606s

    NuSMV:
-- specification ( X ( G p0) &  G ( F p7))  is false
-- specification  F ((p2 | p3) & (p9 | !p10))  is false
-- specification (( X ( G p8)) U ( X (!p10 & p6)))  is false

Tiempo de verificación: 50.47612793s
```

   Al intentar revisar un modelo de tamaño 2^{12} NuSMV se acabó la memoria RAM. En adelante, sólo se muestra el tiempo que tardó el verificador *mcALTL* en hacer la verificación sin recibir un archivo como entrada.

7. Para este modelo con 2^{12} estados, el verificador *mcALTL* tardó milisegundos en verificar las fórmulas:

```
./Main seeds -462212670351055610 -6062112136505055477 -3474006495536599128 7263318818191457324 LTL 12 3

MODELO ALEATORIO DE TAMAÑO 2^12
Profundidad de las fórmulas: 3
Estados iniciales: 3921

(F(p11∧p7))∨((p5∧p1)R(¬p10Up7)) : False
Tiempo de verificación: 0.49415935s

G(F(p11Rp6)) : False
Tiempo de verificación: 0.010149995s

((Fp3)∧(p4Rp11))R((Fp8)U(Gp6)) : False
Tiempo de verificación: 0.003315007s
```

8. En este experimento hay fórmulas más complicadas que en los experimentos anteriores y el modelo tiene 2^{13} estados. Sin embargo, el verificador *mcALTL* tardó segundos en revisar las fórmulas:

```
./Main seeds 1210007372340279273 6577387527259573693 7288838695625089897 1550804476981824494 LTL 13 4

MODELO ALEATORIO DE TAMAÑO 2^13
Profundidad de las fórmulas: 4
Estados iniciales: 8155

(((X¬p1)∨(¬p6Up3))R((p0Up10)∨(p12∨p0)))∨(G((Xp0)U(p12∨p0))) : False
Tiempo de verificación: 3.37841541s

(X((Gp3)R(p12∨p0)))∧(((p5∧¬p1)∨(p4U¬p1))U((¬p1R¬p9)∨(G¬p11))) : False
Tiempo de verificación: 0.027096926s

G((X(p2Up12))U((Xp10)∨(p2Up12))) : False
Tiempo de verificación: 0.004473355s
```

9. Aunque el modelo de este experimento cuenta con 2^{14} estados, tomó segundos revisar las fórmulas. Gracias a que el verificador *mcALTL* es al vuelo, no se necesita revisar todo el modelo si se detecta que la fórmula que se está revisando es falsa:

```
./Main seeds -5513877223295918523 -1129439877177454557 -1335035135597581770 -7224920461052687264 LTL 14 5

MODELO ALEATORIO DE TAMAÑO 2^14
Profundidad de las fórmulas: 5
Estados iniciales: 13322

(F((X(p7R¬p2))R((Gp4)∧(Gp0))))∧(((X(p9Rp4))R((Gp6)∧(G¬p2)))∧(X(Gp11))) : False
Tiempo de verificación: 3.584735578s

(G((p10Rp5)R(Xp0)))∧(G(X((p5Rp0)R(Xp9)))) : False
Tiempo de verificación: 0.006955711s

X((((Xp5)∨(Gp7))R(G(p4Rp13)))R(X(F(p12Rp7)))) : False
Tiempo de verificación: 0.132716741s
```

10. En este experimento el modelo tiene 2^{15} estados pero el verificador *mcALTL* tardó casi 10 segundos:

```
./Main seeds 4245131971143730679 -5160046389823251879 -2541419881106205587 7875966796276176735 LTL 15 3

MODELO ALEATORIO DE TAMAÑO 2^15
Profundidad de las fórmulas: 3
Estados iniciales: 9973

((Fp13)U(p14Up11))U((X¬p8)∨(Xp9)) : False
Tiempo de verificación: 7.311343853s

G((p13U¬p10)R(Gp5)) : False
Tiempo de verificación: 0.0000862s

G((p14Up11)R(p9U¬p6)) : False
Tiempo de verificación: 0.001080055s
```

11. Para este modelo con 2^{17} estados el verificador *mcALTL* tardó cerca de 2 minutos:

```
./Main seeds -2317318845971560383 63174928610846957 7740644631840446483 6914601918853315128 LTL 17 3

MODELO ALEATORIO DE TAMAÑO 2^17
Profundidad de las fórmulas: 3
Estados iniciales: 40076
```

```
(F(p13Rp8))R((Xp9)∧(Fp5)) : False
Tiempo de verificación: 166.124197252s

X(F(p9Rp4)) : False
Tiempo de verificación: 7.95908991s

X(G(p11R¬p6)) : False
Tiempo de verificación: 0.024927514s
```

12. Tomó cerca de 5 minutos verificar un modelo con 2^{18} estados:

```
./Main seeds 57632024823226759 6995788245303145922 76794323844842538 56 -3512037660807785758 LTL 18 3

MODELO ALEATORIO DE TAMAÑO 2^18
Profundidad de las fórmulas: 3
Estados iniciales: 109747

(G(p13R¬p8))∨((Xp14)∧(p15Rp10)) : False
Tiempo de verificación: 289.065524976s

F((p11U¬p8)U(p9Rp4)) : False
Tiempo de verificación: 7.394068113s

((Xp13)U(p16∧p12))R(F(p6∨p7)) : False
Tiempo de verificación: 4.776200319s
```

13. El modelo de este experimento tiene 2^{19} estados y tomó casi 8 minutos verificarlo:

```
./Main seeds -444226188325987799 902136017974027903 6074936912016031953 -1716998379019504120 LTL 19 3

MODELO ALEATORIO DE TAMAÑO 2^19
Profundidad de las fórmulas: 3
Estados iniciales: 150596

(F(¬p11∧p7))∨(F(Xp6)) : False
Tiempo de verificación: 529.055779575s

((p18R¬p13)∨(F¬p14))∧((p16R¬p11)U(¬p13Rp8)) : False
Tiempo de verificación: 0.017109552s

F(X(¬p9Rp4)) : False
Tiempo de verificación: 8.103812868s
```

14. A pesar de ser un modelo con 2^{20} estados, el verificador *mcALTL* tardó casi 3 minutos en verificarlo:

```
./Main seeds 124380107722107864 -2021925931603360927 6315226998243820130 -3066682603604533009 LTL 20 3

MODELO ALEATORIO DE TAMAÑO 2^20
Profundidad de las fórmulas: 3
Estados iniciales: 76891

F(Xp3) : False
Tiempo de verificación: 197.919989502s

((p0Up17)∧(p18∧p14))R(XXp5) : False
Tiempo de verificación: 0.052876905s

(XX¬p9)R(G(p10Rp5)) : False
Tiempo de verificación: 0.000074951s
```

5.5.2. Experimentos CTL

1. En este primer experimento ambos verificadores tuvieron prácticamente el mismo desempeño:

```
./Main seeds 5332018860263000951 -5659695516944381498 6131573798985584345 7270582556455303268 nusmv CTL 5 3

MODELO ALEATORIO DE TAMAÑO 2^5
Profundidad de las fórmulas: 3
Estados iniciales: 6

    mcCTLS:
((EF p3)∨(p1∨p4))∨(EG (p2∨p0)) : True

(AG (p0∨p3))∨(EX (EG p0)) : True

(EG (EG p3))∨(AF (EG p4)) : True

Tiempo de verificación: 0.020458418s

    NuSMV:
-- specification ((EF p3 | (p1 | p4)) | EG (p2 | p0))  is true
-- specification (AG (p0 | p3) | EX (EG p0))  is true
-- specification (EG (EG p3) | AF (EG p4))  is true

Tiempo de verificación: 0.100693462s
```

2. Tanto *mcCTL** y NuSMV tienen casi el mismo desempeño:

```
./Main seeds -3634750090013039070 -5400738926027436334 -4039355692805230431 -8152133867975338019 nusmv CTL 7 3

MODELO ALEATORIO DE TAMAÑO 2^7
Profundidad de las fórmulas: 3
Estados iniciales: 103

    mcCTLS:
(AF (EF ¬p3))∧(EF (EG ¬p6)) : True

(EG (AG p4))∨((EF ¬p1)∨(¬p6∨p2)) : True

EF (E[(EG p5)U(¬p1∧p4)]) : True

Tiempo de verificación: 0.168060836s

    NuSMV:
-- specification (AF (EF !p3) & EF (EG !p6))  is true
-- specification (EG (AG p4) | (EF !p1 | (!p6 | p2)))  is true
-- specification EF E [ (EG p5) U (!p1 & p4) ]   is true

Tiempo de verificación: 0.175071326s
```

3. Un experimento con un modelo con 2^8 estados:

```
./Main seeds -6856179037350201869 -2206205337164536481 410507466240855853 -1681413090854369932 nusmv CTL 8 3

MODELO ALEATORIO DE TAMAÑO 2^8
Profundidad de las fórmulas: 3
Estados iniciales: 111

    mcCTLS:
AF (E[(p7∨¬p2)U(EX p1)]) : True
```

```
A[(EF (EG p7))U(EF (AF ¬p0))] : True

AX (AG (¬p6∧¬p2)) : False

Tiempo de verificación: 0.147563007s

    NuSMV:
-- specification AF E [ (p7 | !p2) U (EX p1) ]   is true
-- specification A [ (EF (EG p7)) U (EF (AF !p0)) ]   is true
-- specification AX (AG (!p6 & !p2))  is false

Tiempo de verificación: 0.420808891s
```

4. Aunque ambos verificadores tuvieron casi el mismo desempeño, NuSMV utilizó 63Mb pero el verificador *mcCTL** sólo utilizó 15Mb:

```
./Main seeds 8835624876428712241 4848600760005918176 8955329516851429801 -1966452871176380 75 nusmv CTL 9 3

MODELO ALEATORIO DE TAMAÑO 2^9
Profundidad de las fórmulas: 3
Estados iniciales: 215

    mcCTLS:
EG ((A[p6U¬p2])∧(EG p1)) : False

EG (AG (EX p7)) : False

EX (EG (p7∧p3)) : False

Tiempo de verificación: 0.264902805s

    NuSMV:
-- specification EG (A [ p6 U !p2 ]  & EG p1)  is false
-- specification EG (AG (EX p7))  is false
-- specification EX (EG (p7 & p3))  is false

Tiempo de verificación: 1.271005132s
```

5. A pesar de que ambos verificadores tardaron casi lo mismo en verificar las fórmulas, NuSMV utilizó 91Mb mientras que *mcCTL** utilizó 50Mb:

```
./Main seeds 9210208927398959434 4419626295539301520 -8267301267196941026 7796684326958031327 nusmv CTL 10 3

MODELO ALEATORIO DE TAMAÑO 2^10
Profundidad de las fórmulas: 3
Estados iniciales: 463

    mcCTLS:
A[((EX p4)∨(EF p8))U(AX (EF p6))] : True

AX (EG (E[p5Up7])) : False

E[(EG (p0∨p3))U(AX (A[p9Up4]))] : False

Tiempo de verificación: 2.037052922s

    NuSMV:
-- specification A [ (EX p4 | EF p8) U (AX (EF p6)) ]   is true
-- specification AX (EG E [ p5 U p7 ] )  is false
-- specification E [ (EG (p0 | p3)) U (AX A [ p9 U p4 ] ) ]   is false
```

```
Tiempo de verificación: 3.157964262s
```

6. La diferencia en tiempo en que los verificadores tardaron en este experimento es apenas de unos 5 segundos; pero en memoria, NuSMV utilizó 416Mb mientras que *mcCTL** utilizó 85Mb:

```
./Main seeds 2380479831102832854 -6986691139666666669 2195935036964894474 -8834836482381054136 nusmv CTL 11 3

MODELO ALEATORIO DE TAMAÑO 2^11
Profundidad de las fórmulas: 3
Estados iniciales: 186

    mcCTLS:
A[(AX (p5∨¬p8))U(EG (p5∨¬p8))] : False

EX (A[(AX ¬p8)U(EG ¬p8)]) : True

EX (EX (A[p5Up10])) : True

Tiempo de verificación: 3.318289555s

    NuSMV:
-- specification A [ (AX (p5 | !p8)) U (EG (p5 | !p8)) ]   is false
-- specification EX A [ (AX !p8) U (EG !p8) ]   is true
-- specification EX (EX A [ p5 U p10 ] )  is true

Tiempo de verificación: 8.164761049s
```

7. En este experimento, el verificador *mcCTL** tardó casi 3 minutos en verificar las fórmulas, mientras que NuSMV tardó 21 segundos en revisarlas todas. Sin embargo, NuSMV utilizó 926Mb mientras que *mcCTL** sólo utilizó 333Mb:

```
./Main seeds 2436629998256113239 -6923507982247632929 -4253175763147895628 -8973650143097053412 nusmv CTL 12 3

MODELO ALEATORIO DE TAMAÑO 2^12
Profundidad de las fórmulas: 3
Estados iniciales: 2118

    mcCTLS:
(AX (EF p10))∧(E[(EF p0)U(AF p2)]) : True

AF (E[(EG p1)U(AF p7)]) : False

EF (EX (AF p10)) : True

Tiempo de verificación: 222.646207011s

    NuSMV:
-- specification (AX (EF p10) & E [ (EF p0) U (AF p2) ] )  is true
-- specification AF E [ (EG p1) U (AF p7) ]   is false
-- specification EF (EX (AF p10))  is true

Tiempo de verificación: 21.528006459s
```

8. En este experimento el verificador *mcCTL** tardó casi hora y media en verificar las fórmulas; en cambio, NuSMV tardó 21 segundos en revisarlas todas. Sin embargo, NuSMV utilizó 925Mb y *mcCTL** utilizó 286Mb:

```
./Main seeds 949953449846448731 -406275148541526948 -38425110258874639 21 410426493641737230 7 nusmv CTL 12 3

MODELO ALEATORIO DE TAMAÑO 2^12
Profundidad de las fórmulas: 3
```

```
Estados iniciales: 1086

    mcCTLS:
AF (E[(EG p0)U(AX p7)]) : False

EF (AX (p7∨p10)) : True

EF (AX (EG p7)) : True

Tiempo de verificación: 5159.164413827s

    NuSMV:
-- specification AF E [ (EG p0) U (AX p7) ]   is false
-- specification EF (AX (p7 | p10))  is true
-- specification EF (AX (EG p7))  is true

Tiempo de verificación: 21.100717779s
```

9. El verificador *mcCTL** utilizó 278mb y tardó casi 5 segundos en verificar todas las fórmulas, mientras que NuSMV tardó casi 20 segundos y utilizó 930Mb:

```
./Main seeds 100 76 4 2 nusmv CTL 12 2

MODELO ALEATORIO DE TAMAÑO 2^12
Profundidad de las fórmulas: 2
Estados iniciales: 1154

    mcCTLS:
EF (p11∨¬p2) : True

EF (EF ¬p2) : True

EG (p6∨p9) : False
Tiempo de verificación: 5.106908062s

    NuSMV:
-- specification EF (p11 | !p2)  is true
-- specification EF (EF !p2)  is true
-- specification EG (p6 | p9)  is false

Tiempo de verificación: 19.924916985s
```

10. Mientras que el verificador *mcCTL** tardó casi 6 segundos en verificar las fórmulas y sólo utilizó 382Mb, NuSMV tardó casi 3 minutos y utilizó 1.5Gb:

```
./Main seeds 6237097947817666748 7051789713999892797 616016117592706892 4966506196981833716 nusmv CTL 13 2

MODELO ALEATORIO DE TAMAÑO 2^13
Profundidad de las fórmulas: 2
Estados iniciales: 6983

    mcCTLS:
(A[p12Up4])∧(EX ¬p10) : False

EG (E[p6Up3]) : False

EX (¬p0∧¬p9) : False

Tiempo de verificación: 6.358932356s
```

```
    NuSMV:
-- specification (A [ p12 U p4 ]  & EX !p10)  is false
-- specification EG E [ p6 U p3 ]   is false
-- specification EX (!p0 & !p9)  is false

Tiempo de verificación: 187.459283826s
```

Para un modelo con 2^{15} estados NuSMV agotó la memoria RAM antes de poder construir el modelo. En adelante, sólo se incluye el tiempo que tardó el verificador *mcCTL⋆* sin recibir un archivo de NuSMV como entrada.

11. El verificador *mcCTL⋆* tardó segundos en verificar las fórmulas:

```
./Main seeds 8666625547701607053 -970530960044127767 3270900684283699943 -9112910988615369691 CTL 15 2

MODELO ALEATORIO DE TAMAÑO 2^15
Profundidad de las fórmulas: 2
Estados iniciales: 10076

(AG p8)∨(E[p5U¬p7]) : False
Tiempo de verificación: 2.66468138s

(E[p3Up5])∨(EX p13) : False
Tiempo de verificación: 0.200768941s

A[(¬p6∨p9)U(AG p14)] : False
Tiempo de verificación: 0.004018043s
```

12. En este experimento el modelo tiene 2^{17} estados y el verificador *mcCTL⋆* tardó cerca de 5 minutos en verificar las fórmulas:

```
./Main seeds -5077824685923578894 859138999552865391 5683394313923832767 -7032287586481233253 CTL 17 2

MODELO ALEATORIO DE TAMAÑO 2^17
Profundidad de las fórmulas: 2
Estados iniciales: 96521

(AX p15)∧(EX p9) : False
Tiempo de verificación: 290.271448781s

A[(AX p14)U(AF p8)] : False
Tiempo de verificación: 2.450429438s

E[(E[p12Up6])U(p11∨p14)] : False
Tiempo de verificación: 0.000072502s
```

13. En casi un minuto, el verificador *mcCTL⋆* verificó este modelo con 2^{18} estados:

```
./Main seeds 7489358293504925504 -4212458674216479258 2806816292527658512 -1650435001088151317 CTL 18 2

MODELO ALEATORIO DE TAMAÑO 2^18
Profundidad de las fórmulas: 2
Estados iniciales: 4354

AF (E[p11Up1]) : False
Tiempo de verificación: 8.346829858s

E[(EF p4)U(E[p1Up9])] : True
Tiempo de verificación: 49.646144294s

EX (E[¬p2Up10]) : False
Tiempo de verificación: 2.30701987s
```

14. En menos de un minuto, el verificador *mcCTL⋆* verificó las tres fórmulas en este modelo con 2^{19} estados:

```
./Main seeds -3020335431298968450 -1085283208950323474 7907697534437260499 1709226432342667921 CTL 19 2

MODELO ALEATORIO DE TAMAÑO 2^19
Profundidad de las fórmulas: 2
Estados iniciales: 18806

(¬p9∨p12)∧(AF p15) : False
Tiempo de verificación: 14.934685478s

AX (p16∧p12) : False
Tiempo de verificación: 0.012260929s

EX (E[p3Up8]) : False
Tiempo de verificación: 30.075345237s
```

15. Para este modelo con 2^{20} estados el verificador *mcCTL⋆* tardó en verificar las tres fórmulas casi una hora y media:

```
./Main seeds 6830968738545262399 -7400582486838530919 -5225068410809829748 4640882333315414212 CTL 20 2

MODELO ALEATORIO DE TAMAÑO 2^20
Profundidad de las fórmulas: 2
Estados iniciales: 462843

E[(p2∧¬p18)U(AF ¬p8)] : False
Tiempo de verificación: 5228.806262611s

E[(¬p16∨p19)U(E[p10Up12])] : False
Tiempo de verificación: 967.169428909s

EX (AX ¬p16) : False
Tiempo de verificación: 20.1397927s
```

En resumen, los resultados de los experimentos para LTL fueron los siguientes:

Experimento	Tamaño del modelo	Profundidad de las fórmulas	mcALTL	NuSMV
1	2^5 estados	2	0.014113057s	0.173221146s
2	2^7 estados	3	0.039594538s	1.02512283s
3	2^7 estados	2	0.034615194s	0.605554901s
4	2^9 estados	3	0.181191898s	5.367861722s
5	2^{10} estados	3	0.557211603s	13.997534672s
6	2^{11} estados	3	1.272414606s	50.47612793s
7	2^{12} estados	3	0.507624352s	—
8	2^{13} estados	4	6.761304175s	—
9	2^{14} estados	5	3.724408029s	—
10	2^{15} estados	3	7.312510108s	—
11	2^{17} estados	3	2.901803577m	—
12	2^{18} estados	3	5.0205965568m	—
13	2^{19} estados	3	8.952945033m	—
14	2^{20} estados	3	3.299549022m	—

Los resultados de los experimentos para CTL fueron los siguientes:

Experimento	Tamaño del modelo	Profundidad de las fórmulas	mcCTL*	NuSMV
1	2^5 estados	3	0.020458418s	0.100693462s
2	2^7 estados	3	0.168060836s	0.175071326s
3	2^8 estados	3	0.147563007s	0.420808891s
4	2^9 estados	3	0.264902805s	1.271005132s
5	2^{10} estados	3	2.037052922s	3.157964262s
6	2^{11} estados	3	3.318289555s	8.164761049s
7	2^{12} estados	3	3.710770116m	21.528006459s
8	2^{12} estados	3	1h43m	21.100717779s
9	2^{12} estados	2	5.106908062s	19.924916985s
10	2^{13} estados	3	6.358932356s	3.12432139m
11	2^{15} estados	2	2.86946836s	—
12	2^{17} estados	2	4.87869917m	—
13	2^{18} estados	2	1.00499990m	—
14	2^{19} estados	2	45.022291644s	—
15	2^{20} estados	2	1h72m	—

Esto muestra que, en varios casos, los verificadores *mcALTL* y *mcCTL** aventajan en tiempo y en consumo de memoria al verificador NuSMV por ser al vuelo y por la evaluación perezosa de Haskell.

Conclusiones y trabajo futuro

If debugging is the process of removing software bugs,
then programming must be the process of putting them in.
Edsger W. Dijkstra

Computers are good at following instructions, but not at reading yor mind.
Donald Knuth

Los verificadores *mcALTL* y *mcCTL** mostrados en este trabajo son una simplificación del trabajo de Girish Bhat, Rance Cleaveland y Orna Grumberg [BCG95]. Estos algoritmos son mucho más fáciles de entender en comparación con otros algoritmos que utilizan autómatas de Büchi, y su implementación en Haskell es confiable ya que se obtuvieron directamente de la semántica formal.

Sobre el trabajo a futuro, un camino a seguir es lograr que los verificadores *mcALTL* y *mcCTL** sean simbólicos. Otro camino a seguir es lograr que el verificador *mcALTL* corra en paralelo para aprovechar las arquitecturas multi-core. En Haskell, es posible agregar paralelismo determinista de forma implícita a un programa utilizando la mónada *Eval*, es decir, se puede hacer que la ejecución de un programa sea más rápida sin cambiar el significado del programa y sin generar *deadlocks* [Mar12]. Sin embargo, requiere de cierta habilidad lograr hacer que un programa en Haskell aproveche al máximo el paralelismo implícito, ya que muchas veces en lugar de hacer que un programa corra más rápido se hace que corra más lento y consuma más memoria pues al agregar paralelismo se podría perder la evaluación perezosa.

Un camino a seguir es verificar formalmente los algoritmos *mcALTL* y *mcCTL** en el asistente de pruebas Coq[2].

Otra dirección de trabajo sería implementar un *actualizador de modelos* basado en el verificador *mcALTL*. Un actualizador es un programa que recibe una fórmula y un modelo como entrada, y si el modelo no cumple la fórmula el actualizador hace modificaciones mínimas al modelo para que la cumpla. Esto es importante ya que la actualización de modelos para la lógica CTL ya se comenzó a estudiar [CR11]; [CR09]; [CR14] pero para la lógica LTL apenas se ha estudiado.

La *verificación de modelos en tiempo real* considera una semántica para trayectorias finitas llamada LTL_F, otra para prefijos de trayectorias infinitas llamada LTL_3, y RV-LTL (*Runtime Verification LTL*) que combina LTL_F y LTL_3. Es posible utilizar autómatas de Büchi para hacer verificación en tiempo real [BLS10]; [BLS11]; [Mag+11]. Un trabajo a futuro es utilizar el verificador *mcALTL* para este problema.

[2] https://coq.inria.fr/

Referencias

[Ada93] Stephen Adams. «Functional pearls efficient sets—a balancing act». En: *Journal of functional programming* 3.4 (1993), págs. 553-561.

[Arn17] Douglas N. Arnold. *The Explosion of the Ariane 5*. 2017. URL: `http://www-users.math.umn.edu/~arnold/disasters/ariane.html`.

[Art67] Prior Arthur. «Past, present and future». En: (1967).

[Bar+18] Jiri Barnat y col. «Parallel Model Checking Algorithms for Linear-Time Temporal Logic». En: *Handbook of Parallel Constraint Reasoning*. Springer, 2018, págs. 457-507.

[BBC03] Jiri Barnat, Lubos Brim y Jakub Chaloupka. «Parallel breadth-first search LTL model-checking». En: *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*. IEEE. 2003, págs. 106-115.

[BBR07] Jiří Barnat, Luboš Brim y Petr Ročkai. «Scalable multi-core LTL model-checking». En: *International SPIN Workshop on Model Checking of Software*. Springer. 2007, págs. 187-203.

[BBS01] Jiri Barnat, Lubos Brim y Jitka Stribrna. «Distributed LTL model-checking in SPIN». En: *Proceedings of the 8th international SPIN workshop on Model checking of software*. Springer-Verlag New York, Inc. 2001, págs. 200-216.

[BCD85] MC Browne, EM Clarke y D Dill. «Checking the correctness of sequential circuits». En: *1985 IEEE Proceedings of the International Conference on Computer Design*. 1985, págs. 545-548.

[BCG95] Girish Bhat, Rance Cleaveland y Orna Grumberg. «Efficient on-the-fly model checking for CTL*». En: *Logic in Computer Science, 1995. LICS'95. Proceedings., Tenth Annual IEEE Symposium on*. IEEE. 1995, págs. 388-397.

[Ben+96] Johan Bengtsson y col. «Verification of an audio protocol with bus collision using UppAal». En: *International Conference on Computer Aided Verification*. Springer. 1996, págs. 244-256.

[BKL08] Christel Baier, Joost-Pieter Katoen y Kim Guldstrand Larsen. *Principles of model checking*. MIT press, 2008.

[BLS10] Andreas Bauer, Martin Leucker y Christian Schallhart. «Comparing LTL semantics for runtime verification». En: *Journal of Logic and Computation* 20.3 (2010), págs. 651-674.

[BLS11] Andreas Bauer, Martin Leucker y Christian Schallhart. «Runtime verification for LTL and TLTL». En: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 20.4 (2011), pág. 14.

[Boc82] Gregor V Bochmann. «Hardware specification with temporal logic: An example». En: *IEEE Transactions on Computers* 3 (1982), págs. 223-231.

[Bri+01] Luboš Brim y col. «Distributed LTL model checking based on negative cycle detection». En: *International Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer. 2001, págs. 96-107.

[Bro+86] Michael C. Browne y col. «Automatic verification of sequential circuits using temporal logic». En: *IEEE Transactions on Computers* 12 (1986), págs. 1035-1044.

[BSK17] Roderick Bloem, Sven Schewe y Ayrat Khalimov. «CTL* synthesis via LTL synthesis». En: *arXiv preprint arXiv:1711.10636* (2017).

[Bur+92] Jerry R Burch y col. «Symbolic model checking: 1020 states and beyond». En: *Information and computation* 98.2 (1992), págs. 142-170.

[Bur74] Rodney Matineau Burstall. *Program proving as hand simulation with a little induction*. North-Holland, 1974.

[Cad] *Construction and Analysis of Distributed Processes-Software Tools for Designing Reliable Protocols and Systems*. `http://cadp.inria.fr`. Accessed: 2017-11-26. 2017.

[CE81] Edmund M Clarke y E Allen Emerson. «Design and synthesis of synchronization skeletons using branching time temporal logic». En: *Workshop on Logic of Programs*. Springer. 1981, págs. 52-71.

[CES86] Edmund M. Clarke, E Allen Emerson y A Prasad Sistla. «Automatic verification of finite-state concurrent systems using temporal logic specifications». En: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 8.2 (1986), págs. 244-263.

[CGH94] Edmund Clarke, Orna Grumberg y Kiyoharu Hamaguchi. «Another look at LTL model checking». En: *International Conference on Computer Aided Verification*. Springer. 1994, págs. 415-427.

[CGP00] Edmund M Clarke, Orna Grumberg y Doron Peled. *Model Checking. 2000*. 2000.

[Che+96] Ghassan Chehaibar y col. «Specification and Verification of the PowerScaleTM bus arbitration protocol: An industrial experiment with lotos». En: *Formal Description Techniques IX*. Springer, 1996, págs. 435-450.

[CJ03] Edmund M Clarke Jr. «Model Checking Overview». En: *Presentation Slides* (2003).

[Cla+92] EM Clarke y col. «Automatic verification of sequential circuit designs». En: *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* 339.1652 (1992), págs. 105-120.

[Cla+93] Edmund M Clarke y col. «Verification of the Futurebus+ Cache Coherence Protocol.» En: *CHDL*. Vol. 93. 1993, págs. 15-30.

[Con] *CAAL (Concurrency Workbench, Aalborg Edition)*. `http://caal.cs.aau.dk`. Accessed: 2017-11-26. 2017.

[Cou+92] Costas Courcoubetis y col. «Memory-efficient algorithms for the verification of temporal properties». En: *Formal methods in system design* 1.2-3 (1992), págs. 275-288.

[Cou99] Jean-Michel Couvreur. «On-the-fly verification of linear temporal logic». En: *International Symposium on Formal Methods*. Springer. 1999, págs. 253-271.

[CR09] Miguel Carrillo y David A Rosenblueth. «A method for CTL model update, representing Kripke Structures as table systems». En: *IJPAM* 52 (2009), págs. 401-431.

[CR11] Miguel Carrillo y David A Rosenblueth. «Nondeterministic update of CTL models by preserving satisfaction through protections». En: *International Symposium on Automated Technology for Verification and Analysis*. Springer. 2011, págs. 60-74.

[CR14] Miguel Carrillo y David A Rosenblueth. «CTL update of Kripke models through protections». En: *Artificial Intelligence* 211 (2014), págs. 51-74.

[CRP97] J Calero, C Roman y GD Palma. «A practical design case using formal verification». En: *Proc. of Design-SuperCon*. Vol. 97. 1997, pág. 1997.

[Dam94] Mads Dam. «CTL and ECTL as fragments of the modal μ-calculus». En: *Theoretical Computer Science* 126.1 (1994), págs. 77-96.

[Dil+92] David L Dill y col. «Protocol verification as a hardware design aid». En: *Computer Design: VLSI in Computers and Processors, 1992. ICCD'92. Proceedings, IEEE 1992 International Conference on*. IEEE. 1992, págs. 522-525.

[DR96] David L Dill y John Rushby. «Acceptance of formal methods: Lessons from hardware design». En: *IEEE Computer* 29.4 (1996), págs. 23-24.

[EC80] E Allen Emerson y Edmund M Clarke. «Characterizing correctness properties of parallel programs using fixpoints». En: *International Colloquium on Automata, Languages, and Programming*. Springer. 1980, págs. 169-181.

[ECB97] Wael M Elseaidy, Rance Cleaveland y John W Baugh. «Modeling and verifying active structural control systems». En: *Science of Computer Programming* 29.1-2 (1997), págs. 99-122.

[EH86] E Allen Emerson y Joseph Y Halpern. «"Sometimes" and "not never" revisited: on branching versus linear time temporal logic». En: *Journal of the ACM (JACM)* 33.1 (1986), págs. 151-178.

[EL87] E Allen Emerson y Chin-Laung Lei. «Modalities for model checking: Branching time logic strikes back». En: *Science of computer programming* 8.3 (1987), págs. 275-306.

[Flo67] Robert W Floyd. «Assigning meanings to programs». En: *Program Verification*. Springer, 1967, págs. 65-81.

[Fuj98] Masahiro Fujita. «Model checking: Its basics and reality». En: *Design Automation Conference 1998. Proceedings of the ASP-DAC'98. Asia and South Pacific*. IEEE. 1998, págs. 217-222.

[Ger+95] Rob Gerth y col. «Simple on-the-fly automatic verification of linear temporal logic». En: *Protocol Specification, Testing and Verification XV*. Springer, 1995, págs. 3-18.

[GV05] Jaco Geldenhuys y Antti Valmari. «More efficient on-the-fly LTL verification with Tarjan's algorithm». En: *Theoretical Computer Science* 345.1 (2005), págs. 60-82.

[GV08] Orna Grumberg y Helmut Veith. *25 years of model checking: history, achievements, perspectives*. Vol. 5000. Springer, 2008.

[HK90] Zvi Har'El y Robert P Kurshan. «Software for analytical development of communications protocols». En: *Bell Labs Technical Journal* 69.1 (1990), págs. 45-59.

[HKM05] Moritz Hammer, Alexander Knapp y Stephan Merz. «Truly on-the-fly LTL model checking». En: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2005, págs. 191-205.

[Hoa69] Charles Antony Richard Hoare. «An axiomatic basis for computer programming». En: *Communications of the ACM* 12.10 (1969), págs. 576-580.

[HR04] Michael Huth y Mark Ryan. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge university press, 2004.

[Hyt] *HyTech: The HYbrid TECHnology Tool*. `https://embedded.eecs.berkeley.edu/research/hytech/`. Accessed: 2017-11-26. 2017.

[Koc+18] Paul Kocher y col. «Spectre attacks: Exploiting speculative execution». En: *arXiv preprint arXiv:1801.01203* (2018).

[Kro] *Open-Kronos: a model-checker for timed (Buchi) automata*. `http://www-verimag.imag.fr/~tripakis/openkronos.html`. Accessed: 2017-11-26. 2017.

[Krö77] Fred Kröger. «LAR: A logic of algorithmic reasoning». En: *Acta Informatica* 8.3 (1977), págs. 243-266.

[Lev+95] Nancy Leveson y col. «Medical devices: The therac-25». En: *Appendix of: Safeware: System Safety and Computers* (1995).

[LP] O Lichtenstein y A Pnueli. «Checking that finite state concurrent programs satisfy their linear specification». En: *Proceedings of the Twelfth ACM Symposium on Principles of Programming Languages*, págs. 97-107.

[LP85] Orna Lichtenstein y Amir Pnueli. «Checking that finite state concurrent programs satisfy their linear specification». En: *Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM. 1985, págs. 97-107.

[Mag+11] Fabrizio Maria Maggi y col. «Monitoring business constraints with linear temporal logic: An approach based on colored automata». En: *International Conference on Business Process Management*. Springer. 2011, págs. 132-147.

[Mar12] Simon Marlow. «Parallel and concurrent programming in Haskell». En: *Central European Functional Programming School*. Springer, 2012, págs. 339-401.

[MC85] Bhubaneswaru Mishra y E Clarke. «Hierarchical verification of asynchronous circuits using temporal logic». En: *Theoretical Computer Science* 38 (1985), págs. 269-291.

[McM93] Kenneth L McMillan. «Symbolic model checking». En: *Symbolic Model Checking*. Springer, 1993, págs. 25-60.

[MO81] Yonatan Malachi y Susan S Owicki. «Temporal specifications of self-timed systems». En: *VLSI Systems and Computations* (1981), págs. 203-212.

[Mur] *Murphi Model Checker*. `http://formalverification.cs.utah.edu/Murphi/`. Accessed: 2017-11-26. 2017.

[NR73] Jürg Nievergelt y Edward M Reingold. «Binary search trees of bounded balance». En: *SIAM journal on Computing* 2.1 (1973), págs. 33-43.

[Nus] *NuSMV: a new symbolic model checker*. http://nusmv.fbk.eu. Accessed: 2017-11-26. 2017.

[Nux] *The nuXmv model checker*. https://nuxmv.fbk.eu. Accessed: 2017-11-26. 2017.

[Pnu77] Amir Pnueli. «The temporal logic of programs». En: *Foundations of Computer Science, 1977., 18th Annual Symposium on*. IEEE. 1977, págs. 46-57.

[Pnu81] Amir Pnueli. «The temporal semantics of concurrent programs». En: *Theoretical computer science* 13.1 (1981), págs. 45-60.

[Pri03] Arthur N Prior. *Time and modality*. John Locke Lecture, 2003.

[QS82] Jean-Pierre Queille y Joseph Sifakis. «Specification and verification of concurrent systems in CESAR». En: *International Symposium on programming*. Springer. 1982, págs. 337-351.

[RL97] Richard Raimi y James Lear. «Analyzing a PowerPC/sup TM/620 microprocessor silicon failure using model checking». En: *Test Conference, 1997. Proceedings., International*. IEEE. 1997, págs. 964-973.

[Ros94] Andrew W Roscoe. *A classical mind: essays in honour of CAR Hoare*. Prentice Hall International (UK) Ltd., 1994.

[Roz11] Kristin Y Rozier. «Linear temporal logic symbolic model checking». En: *Computer Science Review* 5.2 (2011), págs. 163-203.

[SC85] A Prasad Sistla y Edmund M Clarke. «The complexity of propositional linear temporal logics». En: *Journal of the ACM (JACM)* 32.3 (1985), págs. 733-749.

[Sch03] Philippe Schnoebelen. «The Complexity of Temporal Logic Model Checking.» En: *Advances in modal logic* 4.393-436 (2003), pág. 35.

[Tar72] Robert Tarjan. «Depth-first search and linear graph algorithms». En: *SIAM journal on computing* 1.2 (1972), págs. 146-160.

[Var96] Moshe Y Vardi. «An automata-theoretic approach to linear temporal logic». En: *Logics for concurrency*. Springer, 1996, págs. 238-266.

[Ver] *Verus Model Checking*. http://www.cs.cmu.edu/~modelcheck/verus.html. Accessed: 2017-11-26. 2017.

[VW86] Moshe Y Vardi y Pierre Wolper. «An automata-theoretic approach to automatic program verification». En: *Proceedings of the First Symposium on Logic in Computer Science*. IEEE Computer Society. 1986, págs. 322-331.

[Wik18] Wikipedia. *Pentium FDIV flaw*. 2018. URL: https://en.wikipedia.org/wiki/Pentium_FDIV_bug.

[ČP03] Ivana Černá y Radek Pelánek. «Distributed explicit fair cycle detection (set based approach)». En: *International SPIN Workshop on Model Checking of Software*. Springer. 2003, págs. 49-73.