UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

Implementación de un verificador de modelos CTL simbólico en Haskell

TESIS

Que para obtener el título de

Ingeniero en Computación

P R E S E N T A

Javier Diego Fernández Chaparro Plata

DIRECTOR DE TESIS

M. en C. Noé Salomón Hernández Sánchez

Ciudad Universitaria, Cd. Mx., 2019

Agradecimientos

En primer lugar quiero agradecer a mis queridos mamá y papá. Sin ellos no estaría aquí en este momento y sin su apoyo incondicional a lo largo de toda mi vida me habría sido imposible cursar la universidad o terminar esta tesis.

También me gustaría agradecer a la UNAM, a sus académicos por educarme y a sus demás trabajadores por haberme dado la oportunidad de formarme intelectual, técnica y personalmente. No exagero al decir que mi paso por esta institución me transformó para toda la vida. Hago una mención especial para mi director de tesis, el M. en C. Noé Salomón Hernández Sánchez, por haber dedicado una cantidad incontable de horas a revisar el documento, escuchar mis argumentos, sugerir correcciones y recomendar cambios para que este escrito tuviera la mayor calidad posible. No tengo más que gratitud hacia él por todo su apoyo y paciencia hacia mí.

Finalmente, quiero dar gracias a Malvin Gattinger, ahora investigador postdoctoral del Instituto Bernoulli de la Facultad de Ciencia e Ingeniería de la Universidad de Groninga. Contactado originalmente por mi para pedir ayuda con algunos problemas de software, Malvin resultó ser un dispuesto guía y maestro en varios aspectos relacionados con la implementación del código presentado en este trabajo.

Índice general

Agradecimientos I

Introducción V

1. Verificación de modelos 1
1.1. Recapitulación histórica . . . 1
1.1.1. Lógica temporal . . . 1
1.1.2. Verificación de modelos . . . 3
1.2. Lógica de árbol de computación (CTL) . . . 6
1.2.1. Sintaxis . . . 6
1.2.2. Semántica . . . 8
1.2.3. Equivalencia entre operadores . . . 11

2. Algoritmos de verificación 15
2.1. Algoritmos de verificación explícitos . . . 15
2.2. Algoritmos de verificación con equidad . . . 18
2.3. Verificación simbólica . . . 23
2.3.1. Funciones booleanas . . . 23
2.3.2. Diagramas de decisión binarios . . . 25
2.3.3. Codificación simbólica del modelo . . . 32
2.3.4. Algoritmos de verificación simbólica . . . 33
2.3.5. Equidad simbólica . . . 34

3. Descripción del sistema 37
3.1. Herramientas utilizadas . . . 37
3.1.1. Haskell . . . 37
3.1.2. Foreign Function Interface . . . 39
3.1.3. CacBDD . . . 40
3.1.4. HasCacBDD . . . 41
3.1.5. Parsec . . . 41
3.1.6. Hspec . . . 42
3.1.7. Stack . . . 42
3.2. Arquitectura del programa . . . 42
3.2.1. Lenguaje de entrada . . . 43

3.2.2. Definición del lenguaje y análisis sintáctico . . . 45
3.2.3. Representación interna . . . 47
3.2.4. Transformación y análisis semántico . . . 47
3.2.5. Verificación de modelos . . . 48

4. Evaluación **51**
4.1. Contador . . . 52
4.1.1. Descripción . . . 52
4.1.2. Codificación . . . 52
4.1.3. Propiedades a verificar . . . 53
4.1.4. Resultado . . . 53
4.2. Registro de corrimiento . . . 54
4.2.1. Descripción . . . 54
4.2.2. Codificación . . . 55
4.2.3. Propiedades a verificar . . . 55
4.2.4. Resultado . . . 56
4.3. Exclusión Mutua . . . 57
4.3.1. Descripción . . . 57
4.3.2. Codificación . . . 59
4.3.3. Propiedades a verificar . . . 60
4.3.4. Resultado . . . 61
4.4. Equidad . . . 61
4.5. Desempeño en tiempo y memoria . . . 63
4.5.1. Características generales . . . 63
4.5.2. Componentes principales . . . 64

5. Conclusiones y trabajo futuro **67**
5.1. Conclusiones . . . 67
5.2. Trabajo futuro . . . 69

A. Apéndices **71**
A.1. Grafos . . . 71
A.2. Árboles . . . 72

Bibliografía **74**

Introducción

La sociedad depende cada vez más de los sistemas computacionales: la bolsa de valores, los programas de planeación logística, la máquinas fabriles automatizadas, los buscadores de internet, los dispositivos de navegación basados en GPS (*Global Positioning System*), entre otros, se han convertido en herramientas esenciales para el funcionamiento de una economía moderna. Esta dependencia, consecuentemente, provoca grandes pérdidas materiales y sociales cuando alguno de estos sistemas falla. Es por esta razón que el diseño de software robusto y a prueba de errores es esencial.

Dos técnicas comúnmente utilizadas para probar el funcionamiento de los sistemas computacionales son el uso de casos de prueba y la simulación [1]. En el uso de casos de prueba, el dispositivo creado es puesto en funcionamiento y alimentado con entradas particulares, para cada una de estas entradas se conoce con anticipación la salida correcta que debería de tener. La prueba consiste en verificar que el sistema efectivamente produce dichas salidas. La simulación, por su parte, consiste en la creación de un programa de computadora que asemeje el comportamiento del aparato que se desea probar. Utilizando esta simulación, se puede analizar el sistema original de manera indirecta, verificando que el comportamiento de la simulación cumple con las especificaciones hechas durante el diseño.

A pesar de su popularidad comercial, estos métodos tienen deficiencias que los pueden volver insuficientes para asegurar el correcto funcionamiento de sistemas críticos, en los que una falla implica grandes pérdidas económicas e incluso humanas. Por ejemplo, ambos métodos únicamente pueden mostrar la presencia de errores, pero carecen de la capacidad para determinar con certeza su ausencia. Cuando se encuentra un error en un caso de prueba o en una simulación y el mismo se corrige, sigue siendo incierto si aún existen errores presentes, la cantidad de estos y su naturaleza. Un famoso estudio sobre técnicas para reducir el número de defectos en software estima que del 40 % al 50 % de los programas utilizados por los usuarios presenta defectos no triviales [2].

Otra limitación importante de estas técnicas es la carencia de rigor en su uso. La calidad de un caso de prueba muchas veces depende de la habilidad y la creatividad de la persona que lo está diseñando y es vulnerable a sus limitaciones personales (por ejemplo, el mismo estudio menciona cómo la adición de más personas a la revisión de código puede permitir encontrar hasta el 60 % de los errores).

Con el objetivo de corregir algunas de las deficiencias que tienen los métodos informales de verificación de hardware y software, se han creado y estudiado técnicas basadas en teorías matemáticas rigurosas que puedan ayudar a asegurar el funcionamiento de los dispositivos. Dos ejemplos relevantes de estas técnicas son el uso de demostraciones acerca del sistema

estudiado [3] y la verificación de modelos [4, pág. 16].

Huth y Ryan [5, pág. 172] resumen de la siguiente manera la diferencia entre ambos métodos: Para poder verificar un sistema utilizando demostraciones se debe de contar con una codificación del sistema a estudiar como un conjunto de fórmulas Γ, y con una codificación ϕ de cada una de las propiedades que se desean probar utilizando algún lenguaje lógico (se suelen utilizar lógicas de primer y segundo orden). El proceso de verificación consiste en obtener una demostración de las propiedades a partir de la codificación del sistema, esto es, mostrar que $\Gamma \vdash \phi$. Para la demostración, se suelen utilizar herramientas de software llamadas *asistentes de prueba* que poseen técnicas que simplifican el proceso de comprobación.

Algunas de las cualidades que presenta el uso de demostraciones sobre sistemas son: la capacidad para lidiar con sistemas infinitos o ilimitados, así como la expresividad para especificar propiedades de alto nivel acerca de los mismos. Entre sus inconvenientes se encuentran la necesidad de contar con extensa supervisión humana, las limitaciones inherentes del asistente de prueba con el que se está trabajando, así como el alto grado de abstracción que tiene el sistema formal respecto al sistema real que se desea estudiar [3]. Otra gran desventaja, no independiente de las ya mencionadas, es el alto grado de conocimiento técnico necesario para poder realizar este tipo de pruebas. Se trata de una actividad que requiere una gran capacidad matemática y que suele ser hecha por lógicos profesionales.

Por otro lado, la verificación de modelos está basada en un principio lógico distinto. En ella, el sistema estudiado no es codificado como un conjunto de fórmulas, sino como una *interpretación* de alguna lógica formal[1], esto es, una estructura matemática $\mathcal{M}$ en la que se puede satisfacer dicha lógica. La propiedad que se desea estudiar se codifica como una fórmula ϕ. El objetivo es probar que la fórmula que representa a la propiedad se satisface en la interpretación del sistema, es decir, que $\mathcal{M} \models \phi$.

Para la especificación de las propiedades dentro de la verificación de modelos suelen utilizarse lógicas temporales. Estas poseen la propiedad deseable de modelar de manera natural el comportamiento de sistemas que cambian con el tiempo, en los que suelen estar incluidos muchos sistemas convencionales de hardware y de software.

Para asistir en la verificación de modelos también se usan aplicaciones de software especializadas llamadas *verificadores de modelos*, que suelen contar con un lenguaje formal para especificar la interpretación $\mathcal{M}$, así como las n propiedades $\phi_1, \phi_2, \ldots, \phi_n$ que se quieren verificar en el sistema. Además, poseen herramientas para visualizar el modelo, notificar errores en la codificación y estudiar el comportamiento del sistema en ejecución.

Baier y Katoen [4, pág. 14] destacan algunas fortalezas de la verificación de modelos:

- La generalidad en la aplicación de la técnica a diferentes problemas de hardware, software y sistemas embebidos.
- A diferencia de otras técnicas (por ejemplo, el uso de casos de pruebas), la verificación de modelos no es vulnerable a la probabilidad de la presencia de un error, es decir, no tiene una dificultad inherente para encontrar errores poco probables.

[1] Se utiliza este término en lugar del de *modelo* con el objetivo minimizar la posibilidad de confusión con sus otros usos en esta introducción.

- Es una técnica para la cual existen algoritmos bien definidos y que requiere poca interacción con el diseñador.
- Tiene un fundamento matemático sólido. Está basada en la lógica matemática, los algoritmos sobre grafos y las estructuras de datos.

Entre las desventajas que tiene, los autores mencionan:

- Verifica un modelo del sistema y no el producto real que se busca analizar. La calidad de la verificación depende de la calidad del modelo creado y está limitada a las características que se pueden especificar en la abstracción que implica la creación del modelo.
- Únicamente verifica aquellas propiedades especificadas por el diseñador y, por lo tanto, se ve afectada por la calidad y la exhaustividad de estas especificaciones.
- Es vulnerable al problema de la "explosión de estados". El número de estados posibles en el sistema crece de manera exponencial con el incremento en el número de variables involucradas en el mismo, esto puede impedir la verificación de sistemas relativamente complejos.
- Debido a lo explicado en el punto anterior. La verificación de modelos suele ser menos efectiva en sistemas que tienen tipos de datos muy grandes (por ejemplo, el modelado de programas escritos en lenguajes de programación comunes). El número de estados posibles en este tipo de aplicaciones suele volverse intratable computacionalmente.
- Suele ser ineficiente para lidiar con generalizaciones, tipos de datos parametrizados o sistemas con un número arbitrario de componentes.

La interpretación $\mathcal{M}$ suele ser implementada utilizando grafos dirigidos, sin embargo, esta codificación es especialmente susceptible al problema de la "explosión de estados" mencionado en la lista anterior. Una técnica para intentar minimizar este problema es la utilización de una representación simbólica del grafo, a esta técnica se le llama *verificación de modelos simbólica* y fue desarrollada principalmente por Kenneth McMillan [6] y Edmund M. Clarke [7] durante los años 90. La idea detrás de la verificación simbólica es la de codificar conjuntos de estados en el grafo utilizando funciones booleanas y manipular estas funciones en lugar del grafo. Las funciones booleanas correctamente implementadas poseen la capacidad de eliminar redundancias y de explotar regularidades existentes en los conjuntos de estados codificados [7, pág. 143], esto permite, en algunos casos, representar conjuntos relativamente grandes con fórmulas pequeñas.

La utilidad de la verificación de modelos simbólica depende en gran medida de la estructura de datos con la que se implementan las funciones booleanas. Representaciones comunes como las *tablas de verdad* o formas normales como la *forma normal conjuntiva* y la *forma normal disyuntiva* poseen propiedades que las hacen inadecuadas para la verificación de modelos (por ejemplo, las tablas de verdad ocupan una cantidad de espacio en memoria exponencial respecto al número de variables presentes en la fórmula y en las formas normales suele ser complicado realizar ciertas operaciones booleanas [5, pág. 361]). La estructura de datos más

comúnmente utilizada para la verificación simbólica son los *BDT (Binary Decision Trees*, árboles de decisión binarios), y específicamente su versión reducida, los *BDD (Binary Decision Diagrams*, diagramas de decisión binarios). En los BDT, la función booleana es representada como un árbol binario, en el cual los nodos padres representan las variables presentes en la fórmula booleana, un hijo representa la elección del valor *verdadero* a esta variable y el otro la elección de un valor de *falso* para la misma variable, además, todos los nodos hoja están asociados a las constantes booleanas *verdadero* ($\top$) y *falso* ($\bot$). De esta manera, un camino de la raíz del árbol a una hoja representa una asignación de valores de verdad a las variables presentes en la fórmula y el nodo hoja al que se llega indica el valor de verdad de la fórmula para dicha asignación. Los BDD, por su parte, son grafos acíclicos dirigidos que representan BDT a los que se les han eliminado estados redundantes (perdiendo en el proceso la propiedad matemática de ser árboles). Los BDD poseen características que los hacen útiles en la verificación de modelos. Por ejemplo, los BDD suelen ser representaciones compactas, verificar si un BDD representa a una tautología o a una contradicción toma una cantidad de tiempo constante y las operaciones booleanas entre ellos suelen ser razonablemente rápidas [5, pág. 361]).

Con esta pequeña explicación, ya se puede resumir el objetivo principal de este proyecto de tesis:

> El objetivo de este proyecto de tesis es el diseño y la construcción de un verificador de modelos simbólico para la lógica temporal CTL. Esto es, la construcción de un programa de computadora donde se pueda especificar rigurosamente el comportamiento de un sistema en el tiempo, así como las propiedades que se quieren verificar del mismo. Una vez hecha la especificación, el sistema se codificará simbólicamente y se verificará que dicha codificación cumpla con las especificaciones hechas utilizando algoritmos definidos sobre los BDD.

El lenguaje en el que se programará el verificador de modelos es *Haskell*. Haskell[2] es un lenguaje de programación funcional puro, perezoso y con una jerarquía de tipos elaborada [8]. Creado durante los años 80 y 90, Haskell se ha vuelto un estándar en la programación funcional. Basado en elegantes teorías de la computabilidad (el cálculo lambda y la lógica combinatoria, principalmente), y diseñado por un comité integrado por algunos de los pioneros en el campo, la creación de Haskell materializó el deseo de tener un lenguaje de referencia para este paradigma de programación [8].

En la investigación realizada para esta tesis, no se han encontrado verificadores de modelos implementados en Haskell o en algún otro lenguaje funcional, por lo que en ese sentido, el verificador sería el primero en su tipo.

La utilización de Haskell implica retos teóricos y prácticos interesantes. En la parte teórica, la implementación del verificador en este lenguaje obliga al replanteamiento de todos los algoritmos necesarios para la verificación, que generalmente se presentan en forma procedimental, a un marco funcional. También, las estructuras de datos deben de ser modificadas para poder ser procesadas de manera correcta y eficiente bajo este paradigma. Características interesantes de Haskell como las funciones de orden superior y las clases de tipos, por mencionar algunas,

[2]Nombrado en honor al matemático y lógico estadounidense Haskell Brooks Curry (1900 – 1982).

pueden ser de utilidad para dar una solución elegante al problema de la verificación de modelos. En la parte práctica, la creación del sistema requiere de la utilización de herramientas de ingeniería de software que se han creado en Haskell y que difieren considerablemente de sus contrapartes procedimientales, algunas de las más importantes son:

- Parsec: analizador sintáctico funcional basado en *mónadas*, una estructura computacional adaptada de la teoría de categorías a la programación funcional por primera vez en 1922 por Philip Wadler [9]. Wadler formó parte del comité que diseñó Haskell.

- Haskell Cabal: arquitectura para construir e instalar aplicaciones desarrolladas en Haskell. Posee una interfaz de línea de comandos que facilita la construcción e instalación de bibliotecas y aplicaciones [10].

- Haskell Stack: entorno para el desarrollo de aplicaciones en Haskell, hace uso de Cabal y busca superar algunas de sus limitaciones. Stack permite la creación de paquetes de software y ambientes de desarrollo replicables, donde se pueda distribuir código con la seguridad de que funcionarán en cualquier máquina.

- *FFI (Foreign Function Interface*, interfaz de funciones externas): extensión de Haskell que permite definir funciones que hacen referencia a código escrito en otros lenguajes de programación. Permite la manipulación, dentro de Haskell, de funciones que son ejecutadas en última instancia como código escrito en otro lenguaje. Esta extensión permite la reutilización de código sin perder la pureza funcional característica de Haskell.

La realización de este proyecto de tesis nos permitirá resolver algunas preguntas importantes sobre la verificación de modelos:

- ¿Qué tan adecuada es en general la verificación de modelos para se tratada funcionalmente?

- Específicamente, ¿qué partes del proceso de verificación se ven afectadas positiva o negativamente por el uso de lenguajes funcionales en general, y Haskell en particular?

- Como herramienta para el desarrollo de software, ¿cuán conveniente resulta Haskell para construir un verificador de modelos?

El presente trabajo se encuentra dividido en dos grandes secciones: los capítulos 1 y 2 explican los fundamentos teóricos de la verificación de modelos y los capítulos 3 y 4 se encargan de describir los aspectos prácticos del sistema construido.

La primera sección del capítulo 1 hace una recapitulación histórica, tanto de la lógica temporal como de la verificación de modelos, con la intención de dar una visión general de qué se quiere estudiar al utilizar este tipo de lógicas y cómo es que una rama del conocimiento originalmente creada con fines filosóficos, principalmente, encontró aplicación en el estudio de los sistemas computacionales. La segunda sección del capítulo introduce los elementos principales (sintaxis y semántica) de la lógica temporal que utilizará el verificador: la lógica CTL. Se mencionan también algunos teoremas que serán de utilidad posteriormente.

El capítulo 2 describe los algoritmos de verificación que implementará el verificador creado. Como es convencional en la bibliografía, primero se explicarán los algoritmos sobre la forma más directa de entender la interpretación mediante grafo dirigido. Posteriormente se mostrará cómo codificar la interpretación simbólicamente y, finalmente, se describirán los algoritmos de verificación en este nuevo dominio, que es el que en última instancia utiliza el verificador.

El capítulo 3 explica el sistema computacional construido. En la primera sección se mencionan las herramientas de software utilizadas para su construcción, posteriormente se describe, utilizando una notación formal, el lenguaje de entrada al sistema y finalmente se explican a grandes rasgos los módulos que conforman al verificador creado y su función, se muestran además fragmentos de código para cada uno de ellos.

El capítulo 4 se encarga de poner a prueba el sistema construido y de medir su desempeño. En la primera parte del capítulo se codificarán diferentes sistemas computacionales junto con las propiedades que se buscan verificar en ellos y se alimentarán al sistema. Estos ejemplos tienen la intención de comprobar que, para cada uno de ellos, el verificador creado produce salidas correctas y que el sistema codificado tiene el comportamiento deseado. En la segunda parte se medirá el desempeño, tanto en tiempo como en memoria, del verificador para los ejemplos descritos en la sección anterior y se analizará qué etapas de la verificación están consumiendo la mayor cantidad de recursos y por qué.

Capítulo 1

Verificación de modelos

1.1. Recapitulación histórica

El lenguaje que la verificación de modelos utiliza para estudiar los sistemas computacionales es la lógica temporal. En las siguientes dos subsecciones se hará una reseña superficial de la historia de la lógica temporal y de la verificación de modelos. Esta recapitulación tiene la intención de introducir al lector a los temas centrales detrás de las explicaciones técnicas que se harán más adelante en la tesis.

1.1.1. Lógica temporal

El desarrollo histórico de la lógica temporal puede ser dividido, a grandes rasgos, en tres etapas principales: la antigüedad, la edad media y el desarrollo moderno. Aquí se describirán algunos de los temas y debates característicos de cada una de las épocas y el lugar en el que estos encajan en la concepción contemporánea que tenemos sobre la lógica temporal.

Las reflexiones sobre la naturaleza del tiempo y las particularidades de las proposiciones y argumentos que lo toman en cuenta se remonta hasta la antigüedad. Al igual que en muchas otras ramas de la lógica, la primera articulación conocida sobre reflexiones acerca de razonamientos temporales fue hecha por el filósofo griego Aristóteles (384 – 322 a.C.) [11, pág. 10]. En el capítulo IX de su obra *Sobre la interpretación*, Aristóteles hace notar la dificultad asociada a establecer valores de verdad definitivos a proposiciones que hablan sobre el futuro. ¿Se puede saber con certeza si la proposición "habrá una batalla en el mar mañana" es verdadera o falsa hoy? Si no es así, ¿se debe de considerar que esta tienen un valor de verdad adicional a los dos tradicionales? Los debates en torno a la correcta interpretación de los pasajes de Aristóteles se extiende al menos hasta el siglo XX[1]. Las preguntas que él planteó se encuentran en el centro del debate sobre la correcta forma de razonar sobre el tiempo y las formalizaciones asociadas a estos razonamientos.

[1]Por ejemplo, el lógico polaco Jan Łukasiewicz (1878 – 1956) argumentó que Aristóteles consideraba que las proposiciones sobre un futuro contingente no podían considerarse verdaderas o falsas. Lukasiewicz propuso un sistema lógico trivalente en el que, además de los valores convencionales de "verdadero" y "falso", se añadiera un valor "indeterminado" que caracterizara a estas proposiciones.

Otra figura antigua importante para la lógica temporal es Diodoro Cronos (c.a. 340 – 280 a.C.). Diodoro fue un filósofo de la escuela megárica, reconocido por construir argumentos y paradojas filosóficas ingeniosas [11, pág. 15]. Su principal aportación a la lógica temporal fue el llamado "Argumento Maestro", interpretado en la antigüedad como un argumento a favor del determinismo filosófico [11, pág. 15]. A pesar de que el argumento no se conserva en la actualidad, se sabe por los relatos que hizo Epicteto [12] que el mismo relacionó los conceptos lógicos de "necesidad" y "posibilidad" con las nociones temporales de futuro y pasado, y que probablemente buscaba argumentar que únicamente son posibles aquellas proposiciones que ocurrirán eventualmente. Los temas de los que trata el argumento y la relación de los conceptos que se realiza en él tendrían una gran importancia para el debate en las etapas futuras del desarrollo de la lógica temporal:

- Para la edad media; el debate sobre la relación entre el libre albedrío, el conocimiento de Dios de los hechos futuros y el determinismo filosófico.
- Para el desarrollo moderno; las formulaciones modales de la lógica temporal y la semántica de estos sistemas.

Para los que ignoramos la historia, la Edad Media (que se extendió desde el siglo V hasta el siglo XV) podría parecer un periodo de dogmatismo religioso y poca actividad intelectual, la lógica temporal nos brinda un excelente ejemplo de que este no es el caso. Inspirados por problemas relacionados con la teología como el libre albedrío, la interpretación de los textos sagrados y la independencia de la fe del tiempo [11, pág. 33], los pensadores medievales se adentraron en un análisis sumamente detallado de nociones temporales que hacen parecer a los sistemas modernos como superficiales y simples. Entre los problemas con los que se entretuvieron, se pueden enumerar los siguientes:

- La ampliación temporal: el valor de verdad de proposiciones como "el rey de Francia era calvo" depende del momento temporal al que refiere la proposición. Durante este periodo se buscó aclarar y reformular estas proposiciones para poder determinar a qué tiempo hacían referencia.
- La duración del presente: en el lenguaje coloquial utilizamos el término "presente" para referirnos a periodos de tiempo variados; este preciso instante, este día, este año, esta era, entre otros. Los pensadores medievales estudiaron este fenómeno a profundidad, ideando un sistema donde se especificara claramente el intervalo de tiempo al que hace referencia una afirmación sobre el presente y determinando los criterios bajo los cuales una proposición se consideraría verdadera o falsa.
- Los conceptos de inicio y el fin: proposiciones como "me comienzo a bañar" o "me termino de bañar" indican un estado de las cosas en las que suceden cambios y el valor de verdad de una proposición ("me estoy bañado") se transforma (de falso a verdadero o de verdadero a falso). Los pensadores medievales intentaron aclarar este tipo de proposiciones.
- Silogismos temporales: los pensadores medievales estudiaron argumentos cuya validez dependía de las características temporales de las proposiciones involucradas en ellos y articularon clasificaciones generales para estudiarlos.

- La naturaleza del tiempo: ¿puede Dios saber todo lo que va a suceder en el futuro? si es así, ¿el futuro está predeterminado y únicamente existe un futuro posible? o por el contrario, ¿el futuro está indeterminado y existen muchas posibilidades de eventos futuros? Todas estas preguntas relacionan problemas metafísicos y teológicos con la lógica temporal y fueron analizados y discutidos en la época medieval, lo que llevó a la creación de diferentes lógicas del tiempo.

El renacimiento trajo consigo una concepción negativa de la lógica temporal. Los argumentos medievales se consideraron abstractos, artificiales y barrocos. La lógica únicamente debía dedicarse a estudiar el razonamiento correcto para la actividad científica, y, bajo la concepción renacentista, la ciencia lidiaba con verdades atemporales. Estos ataques debilitaron gradualmente el interés en el estudio lógico del tiempo, llevando a que, para el siglo XIX, se consideraran como "irrelevantes" las nociones temporales en la lógica [11, pág. 122].

Poco a poco, la lógica temporal fue recobrando interés en la modernidad. Charles Sanders Peirce (1839 – 1914) fue uno de los primeros filósofos modernos en interesarse por aspectos temporales; estudiando los conceptos de presente, pasado y futuro; describiendo la naturaleza temporal de las proposiciones y reflexionando sobre la continuidad del tiempo. Jan Łukasiewicz (1878 – 1956) creó un sistema lógico formal donde además de los valores tradicionales de verdad "verdadero" y "falso", se añadía un valor "indeterminado" que podía ser interpretado temporalmente en proposiciones futuras contingentes. El filósofo alemán Hans Reichenbach (1891 – 1953) creó un sistema formal para articular los distintos tiempos verbales (presente, pasado, futuro, entre otros) [11, pág. 156]. A pesar de todos estos avances preliminares, la concepción moderna de la lógica temporal fue creada por Arthur Norman Prior (1914 – 1969).

A Prior le debemos la utilización de los operadores P y F para hacer referencia a los tiempos pasado y futuro, respectivamente. La creación, junto con Charles Hamblin (1922 – 1985) del primer sistema axiomático para formalizar la lógica temporal de tiempo lineal [11, pág. 176]. La formalización, junto con Saul Kripke (nacido en 1940), de la noción de futuro ramificado, en la que, a partir del presente, se considera el futuro como una ramificación de posibles estados (y sobre la cual está basada la lógica temporal CTL) [11, pág. 190]. El estudio de los aspectos temporales de la teoría de la relatividad espacial [11, pág. 197], así como la creación de un sistema formal donde se especifica métricamente cuántas unidades de tiempo en el futuro o en el pasado sucede una proposición ($P(x)$ y $F(x)$ que indican: "ocurrió hace x unidades de tiempo" y "ocurrirá en x unidades de tiempo", respectivamente), entre otros descubrimientos.

1.1.2. Verificación de modelos

Concentramos ahora nuestra atención a la que probablemente sea la aplicación práctica más exitosa de la lógica temporal: la verificación de modelos. La verificación de modelos hace uso de la lógica temporal para demostrar propiedades temporales de abstracciones de sistemas de hardware y de software que permiten asegurar su correcto funcionamiento. La verificación de modelos tiene aproximadamente 35 años de existir [13], aquí se hará una descripción breve de algunos de los momentos más importantes de su historia.

Durante los años 70, diversos autores propusieron la utilización de la lógica temporal lineal para razonar sobre programas de computadora [13, pág. 9], especialmente importante fue el

artículo de 1977 "The Temporal Semantics of Concurrent Programs"[14] de Amir Pnueli, ya que fue el primero en utilizarla para la especificación de programas concurrentes, problema que resultaría central para la verificación de modelos. En dicho artículo, Pnueli mostró cómo asignar a un programa P una fórmula en lógica temporal $W(P)$ y cómo la fórmula R de una propiedad del programa se podría demostrar a través de la implicación $W(P) \rightarrow R$. Una desventaja de esta formulación era que la demostración se hacía a mano, lo que complicaba el proceso.

A inicios de los años 80, se comenzaron a estudiar lógicas de ramificación y su aplicación en el estudio de sistemas computacionales. En 1980, Edmund M. Clarke y E. Allen Emerson propusieron un sistema basado en árboles y lo relacionaron con el cálculo-μ de puntos fijos. En 1981, Ben-Ari, Manna y Pnueli utilizaron una sintaxis que hacía uso de letras mayúsculas en lugar de la notación basada en lógica de primer orden utilizada hasta entonces. Esta sintaxis resultó muy popular y es la que se usará en esta tesis. Lamport en 1980 [15] fue el primero en comparar el poder computacional de una lógica de ramificación con una lógica de tiempo lineal, mostrando cómo existían propiedades que podían ser expresadas en cada una de ellas pero no en la otra, haciendo su poder expresivo estrictamente distinto. En 1986, Emerson y Halpern [16] analizaron el problema planteando una tercera lógica CTL* que podía expresar tanto propiedades de lógica lineal (LTL) como propiedades de lógica de árbol (CTL), utilizando esta lógica ampliada, se pudo comparar de manera unificada qué formulas de LTL no podían ser especificadas en CTL y viceversa.

La articulación moderna de la verificación de modelos fue dada en 1981 por Clarke y Emerson [17] y en 1982 por Quielle y Sifakis [18]. En ambos artículos, se utilizaba un enfoque basado en la teoría de modelos y no en la teoría de pruebas (a diferencia del artículo de Pnueli del 77). En ellos, el sistema a estudiar se expresaba como un diagrama de transiciones (también llamado estructura de Kripke[2]) y las propiedades del sistema que se deseaban probar se articulaban como una fórmula en lógica temporal. El objetivo era mostrar que el diagrama era modelo de la fórmula especificada (en el sentido lógico del término). En ambos artículos, además, se desarrollaron los algoritmos que permitían probar las propiedades automáticamente, Clarke y Emerson además analizaron la complejidad del algoritmo que propusieron. Esta mecanización del proceso de verificación supuso una simplificación considerable respecto a los métodos basados en demostraciones a mano.

Junto con estos desarrollos teóricos vinieron los primeros programas verificadores de modelos. En 1983, Clarke, Emerson y Sistla propusieron un algoritmo que mejoraba asintóticamente la complejidad con respecto al descrito en [17] y lo implementaron en el verificador *EMC* (*Extended Model Checker*, verificador de modelos extendido), con esta mejora, se pudo lidiar con sistemas de hasta 10^5 estados [13, pág. 14]. El verificador de modelos *MCB* (*Model Checker B*, verificador de modelos B) implementó por primera vez la capacidad de mostrar contraejemplos en propiedades universales CTL que no se cumplieran y testigos en propiedades existenciales CTL que sí se cumplieran. Esta funcionalidad resultaría esencial en verificadores futuros. También durante los años 80, se utilizó por primera vez la verificación de modelos para comprobar un circuito electrónico, este ejercicio sería la antesala de una serie de artículos

[2] En honor al lógico y filósofo norteamericano Saul Kripke, cuyas contribuciones al campo ya se han mencionado en la recapitulación histórica de la lógica temporal.

donde se aplicaría esta técnica a diferentes dispositivos de hardware [13, pág. 15].

Alrededor de 1990 se inventó una técnica que permitiría lidiar con sistemas considerablemente más grandes que con los que se había trabajado anteriormente y en la cual está basada esta tesis: la verificación de modelos simbólica. Kenneth McMillan, estudiante doctoral de Edmund Clarke, ideó codificar los diagramas de transiciones utilizando funciones booleanas e implementar estas mediante árboles de decisión binarios. Esta representación permite capturar, en ciertos casos, regularidades presentes en el diagrama de transiciones que permiten lidiar con sistemas considerablemente más grandes que las representaciones explícitas basadas en grafos. Utilizando esta representación, se pudieron lidiar con sistemas de hasta 10^{20} estados [7]. Como parte de su doctorado, McMillan, creó el verificador *SMV* (*Symbolic Model Checker*, verificador de modelos simbólico) que implementaba esta técnica para la verificación. El verificador mostró su utilidad al modelarse en él el protocolo de coherencia de caché en el estándar IEEE Futurebus+ en 1992. El verificador creado en el trabajo presente utiliza una sintaxis restringida de SMV para la especificación, tanto de la estructura de Kripke como de las fórmulas en lógica temporal.

Un problema que históricamente ha sido complicado para la verificación de modelos es el análisis de sistemas asíncronos (aquellos en los que ocurren eventos que no se encuentran sincronizados con la ejecución principal), ya que la "explosión de estados" es particularmente grave en ellos. Una técnica para tratar con este tipo de sistemas es la *reducción parcial del orden*, la cual busca explotar las partes "independientes" de los procesos asíncronos (aquellas cuyo orden no afecta el resultado final del proceso) con el objetivo de reducir el tamaño del sistema a verificar. Durante los años 90 McMillan y otros publicaron estudios detallándola [13, pág. 18].

Además de estos métodos establecidos, existen algunos otros procedimientos ad hoc para la solución de ciertos tipos de problemas que se desarrollaron durante los 90. El *razonamiento composicional* busca explotar la estructura modular de algunos circuitos para poder separar su estudio en partes más pequeñas que sean tratables. La *abstracción* busca simplificar el estudio de un sistema mediante el uso de un número pequeño de estados que representen de manera general operaciones complejas. La *reducción simétrica* busca aprovechar simetrías inherentes en un sistema para reducir su tamaño. Finalmente, el estudio parcial de *sistemas parametrizados* puede ayudar, en ciertos casos, a reducir el tamaño de un circuito que se desea estudiar.

Edmund Clarke, en una reseña sobre la historia de la verificación de modelos, cita los siguientes desarrollos como algunos de los más importantes que se han hecho en el campo desde los años 90 [13, pág. 20]:

- Autómatas cronometrados e híbridos,
- La verificación de modelos para protocolos de seguridad,
- Verificación de modelos adjunta,
- Verificación de modelos y aprendizaje composicional,
- Abstracción de predicados,

- Sistemas con un número infinito de estados.

Entre los retos más importantes para el futuro, el autor destaca:

- La verificación de modelos para software,
- El estudio de la interacción entre la demostración de teoremas y la verificación de modelos,
- Explotar el problema de la satisfacción booleana (SAT) y el de satisfacción módulo teorías (SMT) para el beneficio de la verificación de modelos,
- Verificación de modelos probabilística,
- La interpretación de contraejemplos,
- El análisis de la cobertura en las especificaciones, es decir, estudiar si las especificaciones cubren adecuadamente el comportamiento del dispositivo.

Como podemos ver, la verificación es un campo de estudio fértil, con una vida relativamente corta y con una gran oportunidad para el desarrollo futuro.

1.2. Lógica de árbol de computación (CTL)

La lógica temporal CTL se basa en una interpretación no determinística del tiempo. A partir del presente, existen múltiples estados futuros a los que se puede llegar y de estos, también se puede llegar a otros múltiples estados, de ahí que a esta lógica también se le suele llamar de ramificación.

Al igual que otros sistemas formales, la lógica temporal está conformada por dos partes: sintaxis y semántica. La sintaxis indica el conjunto de expresiones correctas dentro del sistema, a las que se les suele llamar *fórmulas bien formadas*. La semántica, por su parte, determina la forma en la que estas expresiones adquieren un valor de verdad específico (verdadero, falso, posiblemente verdadero, necesariamente falso, entre otros), asignando entonces un "significado" a las expresiones inertes especificadas en la sintaxis.

1.2.1. Sintaxis

Para formar las expresiones de la lógica CTL debemos contar con un conjunto de *variables proposicionales* (AP) que representan hechos atómicos de interés. La lógica temporal CTL amplía a la lógica proposicional con operadores de dos tipos: de *trayectoria* y de *estado*. Los operadores de *trayectoria*: *estado siguiente* (X), *globalmente* (G), *hasta* (U) y *futuro* (F) indican propiedades de una trayectoria a lo largo del árbol. Los operadores de *estado*: *existe* (E) y *para todo* (A) que cuantifican a las trayectorias a partir de un estado. Todas estas nociones se formalizarán cuando se estudie la semántica del sistema.

El conjunto de *fórmulas bien formadas* de la lógica CTL en notación Backus-Naur es:

Definición 1.1 (Sintaxis de la lógica CTL)**.**

$$\phi ::= \top \mid p \mid \neg\,\phi \mid \phi_1 \wedge \phi_2 \mid \mathrm{AX}\,\phi \mid \mathrm{EX}\,\phi \mid \mathrm{AG}\,\phi \mid \mathrm{EG}\,\phi \mid \mathrm{AF}\,\phi \mid \mathrm{EF}\,\phi \mid$$

$$\mathrm{A}[\,\phi_1\,\mathrm{U}\,\phi_2\,] \mid \mathrm{E}[\,\phi_1\,\mathrm{U}\,\phi_2\,]$$

donde $p \in AP$. Es importante notar cómo todo operador de trayectoria está precedido por un operador de estado y que las combinaciones de estos se pueden tratar de manera independiente.

Como es convencional en la lógica proposicional, se pueden definir otros símbolos y operadores a partir de la negación ($\neg$), la conjunción ($\wedge$) y verdadero ($\top$):

Definición 1.2 (Operadores proposicionales derivados, en orden decreciente de precedencia)**.**

$$\bot \stackrel{\text{def}}{=} \neg\,\top$$

$$\phi_1 \vee \phi_2 \stackrel{\text{def}}{=} \neg\,((\neg\,\phi_1) \wedge (\neg\,\phi_2))$$

$$\phi_1 \oplus \phi_2 \stackrel{\text{def}}{=} (\phi_1 \wedge (\neg\,\phi_2)) \vee ((\neg\,\phi_1) \wedge\ \phi_2)$$

$$\phi_1 \rightarrow \phi_2 \stackrel{\text{def}}{=} (\neg\,\phi_1) \vee \phi_2$$

$$\phi_1 \leftrightarrow \phi_2 \stackrel{\text{def}}{=} (\phi_1 \rightarrow \phi_2) \wedge (\phi_2 \rightarrow \phi_1)$$

Existe una convención para simplificar la notación de las fórmulas CTL. Los operadores unarios tienen la precedencia más alta, y dentro de estos, la negación ($\neg$) tiene mayor precedencia que los operadores temporales (EX, AX, EF, AF, EG y AG). En los operadores binarios, los operadores proposicionales ($\wedge$ y $\vee$, $\oplus$, $\rightarrow$ y $\leftrightarrow$) tienen mayor precedencia que los operadores temporales (EU y AU). Cuando los criterios de precedencia no sean suficientes para determinar inequívocamente la agrupación de una expresión, se debe de tomar en cuenta la asociatividad de los operadores lógicos; todos los operadores unarios son asociativos por la derecha; $\wedge$, $\vee$, $\leftrightarrow$ y $\oplus$ son independientes de la asociatividad y $\rightarrow$ es asociativo por la derecha. Bajo esta convención, la expresión:

$$\mathrm{EX}\,\neg\,p\ \wedge\ q\ \vee\ s\ \leftrightarrow\ \mathrm{AG}\,\neg\,s\ \rightarrow\ p\ \rightarrow\ s$$

se agruparía como:

$$(((\mathrm{EX}\,(\neg\,p))\ \wedge\ q)\ \vee\ s)\ \leftrightarrow\ ((\mathrm{AG}\,(\neg\,s))\ \rightarrow\ (p\ \rightarrow\ s))$$

1.2.2. Semántica

Una vez establecidas las reglas para la construcción de las *fórmulas bien formadas* en la lógica CTL, debemos de especificar una estructura matemática sobre la cual se van a interpretar las fórmulas y un método para determinar si para una fórmula y una estructura dadas, la fórmula se satisface en la estructura, es decir, debemos de especificar una *interpretación* de las fórmulas y una *relación de satisfacción* para las mismas [4, pág. 320]. Estos dos elementos conforman la *semántica* de la lógica.

La interpretación de las fórmulas se hará sobre un diagrama de transiciones (también llamado estructura de Kripke) definido formalmente como:

Definición 1.3 (Diagrama de transiciones)**.** Sea $\mathcal{M} = (S, \longrightarrow, I, AP, L)$ un diagrama de transiciones, los elementos que lo conforman son:

- S es un conjunto finito de estados,
- $\longrightarrow \subseteq S \times S$ es una relación de transición,
- $I \subseteq S$ es un conjunto de estados iniciales,
- AP es un conjunto finito de variables proposicionales y
- $L : S \to 2^{AP}$ es una función de etiquetado que asigna a cada estado un conjunto de etiquetas.

Durante el trabajo presente se considerará que S y AP son finitos, reescribiremos (s, s') como $s \to s'$ para dos estados cualesquiera s y s', diremos que s' es sucesor de s y que s es antecesor de s'. Se establecerá la restricción de que para todo estado $s \in S$ debe existir al menos un s' tal que $s \to s'$, esta restricción evita la condición de "punto muerto" donde un sistema, estando en un determinado estado, ya no tiene a dónde ir.

Se suele representar gráficamente a los diagramas de transiciones mediante un grafo dirigido. El diagrama de transiciones representado por la Figura 1.1 está conformado por los siguientes elementos:

- $S = \{s_0, s_1, s_2, s_3\}$,
- $\longrightarrow = \{s_0 \to s_1, s_1 \to s_0, s_0 \to s_2, s_2 \to s_0, s_2 \to s_3, s_3 \to s_2, s_1 \to s_3, s_3 \to s_1\}$,
- $I = \{s_0\}$,
- $AP = \{q, r\}$,
- $L = \{(s_0, \varnothing), (s_1, \{q\}), (s_2, \{r\}), (s_3, \{r, q\})\}$.

También se considerará que L es una función inyectiva, esto es, que estados distintos nunca tendrán asociados el mismo conjunto de variables proposicionales. Esta restricción nos permite identificar a un estado por las variables que se satisfacen en él de manera única y permite eliminar la etiqueta con el nombre de los estados de las figuras (s_0, s_1 , s_2 y s_3 en la Figura 1.1), con lo que también se simplifica el diagrama.

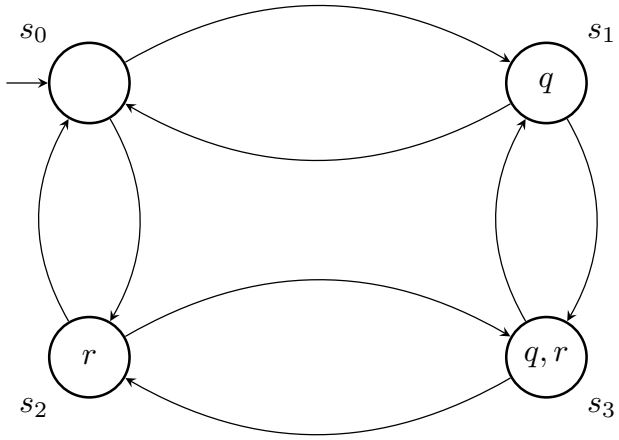

Figura 1.1: Representación de una estructura de Kripke como una grafo dirigido.

Ahora definiremos un *camino*, que intuitivamente se refiere a una ejecución particular del sistema, el cual transcurre a través de estados de manera indefinida.

Definición 1.4 (Camino en un diagrama de transiciones)**.** Sea un diagrama de transiciones $\mathcal{M} = (S, \longrightarrow, I, AP, L)$ cualquiera con las restricciones establecidas, un camino π es una secuencia infinita de estados $s_0, s_1, ...$ en S tales que para todo $i \geq 0$ se tiene que $s_i \rightarrow s_{i+1}$. $\pi[i]$ se utilizará para representar el i-ésimo estado de la secuencia.

Una forma de visualizar la ejecución de un diagrama de transiciones es a través de un árbol en el cual los hijos de un nodo representen las transiciones que ese nodo puede realizar. La Figura 1.2 muestra la ejecución de las tres primeras transiciones del sistema representado por la Figura 1.1, evidentemente, este árbol tienen una profundidad infinita. Bajo esta representación, un camino del sistema de transiciones sería una rama particular del árbol.

Definición 1.5 (Conjunto de caminos de un diagrama y conjunto de caminos a partir de un estado)**.** Sea $\mathcal{M} = (S, \longrightarrow, I, AP, L)$ un diagrama de transiciones, $Cam(\mathcal{M})$ es el conjunto de todos los caminos posibles de formar en $\mathcal{M}$ y $Cam(s)$ es el conjunto de todos los caminos π en $\mathcal{M}$ tales que $\pi[0] = s$ con $s \in S$.

Con estos conceptos, ya podemos definir la interpretación de una fórmula temporal en un diagrama de transiciones.

Definición 1.6 (Relación de satisfacción CTL para estados)**.** Sea $\mathcal{M} = (S, \longrightarrow, I, AP, L)$ un diagrama de transiciones , $a \in AP$, $s \in S$ y ϕ, ϕ_1 y ϕ_2 tres fórmulas CTL cualesquiera. La relación de satisfacción $\models$ sobre $\mathcal{M}$ se define inductivamente de la siguiente manera:

- $s \models \top$ siempre,
- $s \models a$ syss $a \in L(s)$,

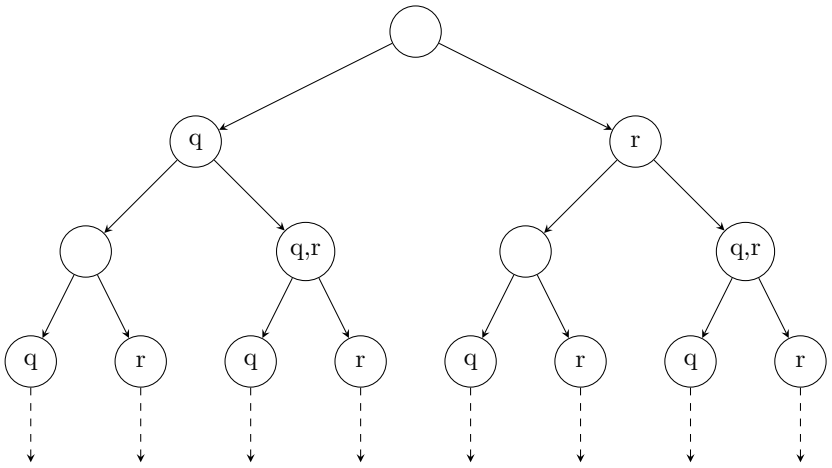

Figura 1.2: Representación de la ejecución de un diagrama de transiciones a través de un árbol.

- $s \models \neg\, \phi$ syss $s \not\models \phi$,
- $s \models \phi_1 \wedge \phi_2$ syss $s \models \phi_1$ y $s \models \phi_2$,
- $s \models \mathrm{AX}\, \phi$ syss $\forall \pi \in Cam(s)$ se cumple que $\pi[1] \models \phi$,
- $s \models \mathrm{EX}\, \phi$ syss $\exists \pi \in Cam(s)$ tal que $\pi[1] \models \phi$,
- $s \models \mathrm{AG}\, \phi$ syss $\forall \pi \in Cam(s)$ se cumple que $\forall j \geq 0\ (\pi[j] \models \phi)$,
- $s \models \mathrm{EG}\, \phi$ syss $\exists \pi \in Cam(s)$ tal que $\forall j \geq 0\ (\pi[j] \models \phi)$,
- $s \models \mathrm{AF}\, \phi$ syss $\forall \pi \in Cam(s)$ se cumple que $\exists j \geq 0\ (\pi[j] \models \phi)$,
- $s \models \mathrm{EF}\, \phi$ syss $\exists \pi \in Cam(s)$ tal que $\exists j \geq 0\ (\pi[j] \models \phi)$,
- $s \models \mathrm{A}[\, \phi_1 \,\mathrm{U}\, \phi_2\,]$ syss $\forall \pi \in Cam(s)$ se cumple que $\exists j \geq 0\ ((\pi[j] \models \phi_2)$ y $\forall\, 0 \leq k < j\ (\pi[k] \models \phi_1))$ y
- $s \models \mathrm{E}[\, \phi_1 \,\mathrm{U}\, \phi_2\,]$ syss $\exists \pi \in Cam(s)$ tal que $\exists j \geq 0\ ((\pi[j] \models \phi_2)$ y $\forall\, 0 \leq k < j\ (\pi[k] \models \phi_1))$.

Hasta ahora, la definición de satisfacción se ha hecho sobre estados de un diagrama de transiciones. Se dirá que un diagrama satisface una fórmula si todos los estados iniciales de este la satisfacen.

Definición 1.7 (Relación de satisfacción CTL para diagramas de transiciones)**.** Sea $\mathcal{M} = (S,\ \longrightarrow,\ I,\ AP,\ L)$ un diagrama de transiciones y sea ϕ un fórmula CTL, la relación de satisfacción entre ambos se define como:

$$\mathcal{M} \models \phi \text{ si y solamente si } \forall s_i \in I \text{ se cumple que } s_i \models \phi.$$

1.2.3. Equivalencia entre operadores

Con las definiciones 1.6 y 1.7 ya se cuenta con un proceso para saber, a partir de una estructura de Kripke arbitraria $\mathcal{M}$ y una fórmula cualquiera ϕ, si $\mathcal{M} \models \phi$ ó $\mathcal{M} \not\models \phi$, sin embargo, el proceso descrito puede ser simplificado. Si se logra mostrar que algunos operadores temporales pueden ser escritos en términos de otros, entonces se puede reducir el número de algoritmos de verificación que se deben de programar a un número menor a los ocho definidos con anterioridad. Para lograr esto será necesario definir dos conceptos: el de *equivalencia lógica* entre fórmulas CTL y el de *conjunto adecuado de conectivos*.

Definición 1.8 (Equivalencia lógica entre fórmulas CTL)**.** Dos fórmulas CTL ϕ_1 y ϕ_2 son equivalentes, denotado como $\phi_1 \equiv \phi_2$, si y solamente si, para todo diagrama de transiciones $\mathcal{M}$ se cumple que:

$$\mathcal{M} \models \phi_1 \text{ syss } \mathcal{M} \models \phi_2$$

Definición 1.9 (Conjunto adecuado de conectivos)**.** Un *conjunto adecuado de conectivos* para un sistema lógico es un subconjunto propio $C_m \subset C$ del conjunto de todos los conectivos C de manera que toda fórmula generada con los conectivos en C es equivalente a una fórmula generada con los conectivos en C_m.

Un artículo importante sobre este concepto (y del cual se tomó la definición 1.9) es [19]. En él, se demuestra el siguiente teorema:

Teorema 1.1 (Teorema de Adecuación)**.** Los *conjuntos adecuados de conectivos* para la lógica CTL son exactamente aquellos que contienen al menos uno de los operadores en $\{\mathrm{AX}, \mathrm{EX}\}$; al menos uno de los operadores en $\{\mathrm{EG}, \mathrm{AF}, \mathrm{AU}\}$ y el operador EU.

Corolario 1.1. El conjunto $\{\mathrm{EX}, \mathrm{EU}, \mathrm{EG}\}$ es un *conjunto adecuado de conectivos* para la lógica CTL.

De este corolario, se define el siguiente concepto:

Definición 1.10 (Forma normal existencial para CTL)**.** Una fórmula CTL se encuentra en *forma normal existencial* si los únicos operadores temporales que aparecen en ella se encuentran en el conjunto $\{\mathrm{EX}, \mathrm{EU}, \mathrm{EG}\}$.

Aunque por razones de espacio no se demostrará el resultado principal descrito en el Teorema 1.1, sí se mostrará cómo obtener una fórmula en forma normal existencial a partir de cualquier fórmula CTL y las equivalencias necesarias para hacerlo.

Teorema 1.2 (Transformación a Forma Normal Existencial)**.** En la lógica CTL se cumplen las siguientes equivalencias:

1. $\mathrm{AX}\,\phi \quad \equiv \quad \neg\,\mathrm{EX}\,\neg\,\phi,$
2. $\mathrm{AF}\,\phi \quad \equiv \quad \neg\,\mathrm{EG}\,\neg\,\phi,$
3. $\mathrm{EF}\,\phi \quad \equiv \quad \mathrm{E}[\,\top\,\mathrm{U}\,\phi\,],$

4. $\mathrm{AG}\,\phi \quad \equiv \quad \neg\,\mathrm{EF}\,\neg\,\phi$ y
5. $\mathrm{A}[\,\phi_1\,\mathrm{U}\,\phi_2\,] \quad \equiv \quad \neg\,\mathrm{E}[\,\neg\,\phi_2\,\mathrm{U}\,(\neg\,\phi_1 \wedge \neg\,\phi_2)] \wedge \neg\,\mathrm{EG}\,\neg\,\phi_2$.

Demostración. Las primeras cuatro equivalencias se pueden mostrar utilizando únicamente equivalencias de la lógica de primer orden:

1. De acuerdo con la definición de $\mathrm{EX}\,\phi$:

$$\begin{aligned}\neg\,\mathrm{EX}\,\neg\,\phi &\Leftrightarrow \neg(\exists\pi \in Cam(s)\,(\pi[1] \not\models \phi)) \\ &\Leftrightarrow \forall\pi \in Cam(s)\,\neg\,(\pi[1] \not\models \phi) \\ &\Leftrightarrow \forall\pi \in Cam(s)\,(\pi[1] \models \phi) \\ &\Leftrightarrow \mathrm{AX}\,\phi\end{aligned}$$

2. De acuerdo con la definición de $\mathrm{EG}\,\phi$:

$$\begin{aligned}\neg\mathrm{EG}\,\neg\,\phi &\Leftrightarrow \neg\,(\exists\pi \in Cam(s)\,(\forall j \geq 0\,(\pi[j] \not\models \phi))) \\ &\Leftrightarrow\ \forall\pi \in Cam(s)\,(\neg\,(\forall j \geq 0\,(\pi[j] \not\models \phi))) \\ &\Leftrightarrow\ \forall\pi \in Cam(s)\,(\exists j \geq 0\,\neg\,(\pi[j] \not\models \phi)) \\ &\Leftrightarrow\ \forall\pi \in Cam(s)\,(\exists j \geq 0\ (\pi[j] \models \phi)) \\ &\Leftrightarrow\ \mathrm{AF}\,\phi\end{aligned}$$

   Las demostraciones de los incisos 3 y 4 se realizan de manera similar a los ya demostrados.

5. La demostración de la última equivalencia implica un razonamiento más complicado que no es traducible directamente a lógica de primer orden. Se prueba la equivalencia:

$$\mathrm{A}[\,\phi_1\,\mathrm{U}\,\phi_2\,] \ \Leftrightarrow\ \neg\,\mathrm{E}[\,\neg\,\phi_2\,\mathrm{U}\,(\neg\,\phi_1 \wedge \neg\,\phi_2)] \wedge \neg\,\mathrm{EG}\,\neg\,\phi_2$$

   De lo obtenido en el punto 2, $\neg\,\mathrm{EG}\,\neg\,\phi_2$ se puede reescribir como $\mathrm{AF}\,\phi_2$:

$$\mathrm{A}[\,\phi_1\,\mathrm{U}\,\phi_2\,] \ \Leftrightarrow\ \neg\,\mathrm{E}[\,\neg\,\phi_2\,\mathrm{U}\,(\neg\,\phi_1 \wedge \neg\,\phi_2)] \wedge \mathrm{AF}\,\phi_2$$

   Se trata de un argumento de la forma $p \Leftrightarrow q$:
   $\Rightarrow$:
   Se debe mostrar que:

$$\mathrm{A}[\,\phi_1\,\mathrm{U}\,\phi_2\,] \Rightarrow \mathrm{AF}\,\phi_2$$

   y que:

$$\mathrm{A}[\,\phi_1\,\mathrm{U}\,\phi_2\,] \Rightarrow \neg\,\mathrm{E}[\,\neg\,\phi_2\,\mathrm{U}\,(\neg\,\phi_1 \wedge \neg\,\phi_2)]$$

   $\mathrm{AF}\,\phi_2$ se obtiene de manera directa de la definición de $\mathrm{A}[\,\phi_1\,\mathrm{U}\,\phi_2\,]$.
   Para demostrar:

$$\mathrm{A}[\,\phi_1\,\mathrm{U}\,\phi_2\,] \Rightarrow \neg\,\mathrm{E}[\,\neg\,\phi_2\,\mathrm{U}\,(\neg\,\phi_1 \wedge \neg\,\phi_2)]$$

Se utilizará la equivalencia:

$$\neg\,(\mathrm{A}[\,\phi_1\,\mathrm{U}\,\phi_2\,]\ \wedge\ \mathrm{E}[\,\neg\,\phi_2\,\mathrm{U}\,(\neg\,\phi_1 \wedge \neg\,\phi_2)])$$

Supongamos que se cumple $\mathrm{A}[\,\phi_1\,\mathrm{U}\,\phi_2\,]$, esta fórmula establece que, para todo camino, ϕ_2 se cumple en un estado y para todos los estados anteriores se cumple ϕ_1. Por su parte, $\mathrm{E}[\,\neg\,\phi_2\,\mathrm{U}\,(\neg\,\phi_1 \wedge \neg\,\phi_2)]$ establece que, existe un camino en el que en un estado se satisface $\neg\,\phi_1 \wedge \neg\,\phi_2$ y en todos los estados anteriores se satisface $\neg\,\phi_2$. Si por $\mathrm{A}[\,\phi_1\,\mathrm{U}\,\phi_2\,]$, ϕ_2 se ha de cumplir en algún estado, este estado debe de ocurrir después de que haya ocurrido $\neg\,\phi_1 \wedge \neg\,\phi_2$ (ya que hasta ese momento, siempre se cumple $\neg\,\phi_2$ de acuerdo con $\mathrm{E}[\,\neg\,\phi_2\,\mathrm{U}\,(\neg\,\phi_1 \wedge \neg\,\phi_2)]$), sin embargo, en el estado en el que se cumple $\neg\,\phi_1 \wedge \neg\,\phi_2$ evidentemente no se cumple ϕ_1, contradiciendo lo que postula $\mathrm{A}[\,\phi_1\,\mathrm{U}\,\phi_2\,]$. Ambas fórmulas no pueden ocurrir simultáneamente.

$\Longleftarrow$:
Debemos probar que:

$$\neg\,\mathrm{E}[\,\neg\,\phi_2\,\mathrm{U}\,(\neg\,\phi_1 \wedge \neg\,\phi_2)] \wedge \mathrm{AF}\,\phi_2\ \Rightarrow\ \mathrm{A}[\,\phi_1\,\mathrm{U}\,\phi_2\,]$$

Para un camino arbitrario, $\mathrm{E}[\,\neg\,\phi_2\,\mathrm{U}\,(\neg\,\phi_1 \wedge \neg\,\phi_2)]$ puede ser falso en cualquiera de los dos siguientes casos:

a) Si $\neg\,\phi_1 \wedge \neg\,\phi_2$ nunca se cumple, entonces $\phi_1 \vee \phi_2$ se cumple indefinidamente. $\mathrm{AF}\,\phi_2$ forza a que en algún estado se cumpla ϕ_2 eliminando la posibilidad de que satisfacer esta condición cumpliendo globalmente solo ϕ_1. Si ϕ_2 se cumple eventualmente y $\phi_1 \vee \phi_2$ se cumple en todos lo estados, entonces $\mathrm{A}[\,\phi_1\,\mathrm{U}\,\phi_2\,]$ se cumple.

b) Si $\neg\,\phi_1 \wedge \neg\,\phi_2$ se cumple en algún estado, pero en un estado anterior ϕ_2 se cumple. Tomemos la primera ocurrencia de $\neg\,\phi_1 \wedge \neg\,\phi_2$, en los estados anteriores a este estado donde se cumpla $\neg\,\phi_2$, forzosamente se debe de cumplir ϕ_1 (de no ser así, no sería la primera ocurrencia de $\neg\,\phi_1 \wedge \neg\,\phi_2$) esto fuerza a que, sin importar en qué estado se cumpla ϕ_2, en todos los anteriores se habrá cumplido ϕ_1. Con ello se cumple $\mathrm{A}[\,\phi_1\,\mathrm{U}\,\phi_2\,]$.

□

Capítulo 2

Algoritmos de verificación

2.1. Algoritmos de verificación explícitos

En el capítulo anterior se describió un método para determinar si una estructura de Kripke satisface una fórmula CTL o no. A pesar de ser matemáticamente correcto, este método resulta inapropiado para ser implementado en una computadora, por dos razones principalmente:

1. Se debe de trabajar con caminos de profundidad infinita, lo cual es evidentemente imposible en una computadora real.

2. El procedimiento solamente está descrito para la satisfacción en un estado, lo cual lo hace ineficiente para lidiar con estructuras que tienen múltiples estados.

Para tratar con el primer problema, debemos recordar que todo diagrama de transiciones tiene un número de estados finito, de esta manera, todo camino infinito estará conformado por una secuencia finita de estados que se repite de manera indefinida. Si se lleva un registro de los estados que se han visitado, entonces se puede determinar el criterio de satisfacción en una cantidad de tiempo finita. El segundo de los problemas se resuelve utilizando el concepto de *conjunto de satisfacción*, para verificar si una determinada fórmula se satisface en una estructura, se determinará, para cada subfórmula de esta, el conjunto de estados en los que dicha subfórmula se satisface y, posteriormente, se combinarán estos conjuntos de acuerdo con la semántica de la lógica CTL para determinar el conjunto de estados que satisfacen la fórmula original. Finalmente, se debe de verificar si todo estado inicial de la estructura de Kripke satisface a la fórmula o no y emitir el criterio de satisfacción final.

Una forma de visualizar la estructura recursiva del algoritmo es a partir de un árbol de derivación. Supongamos la siguiente fórmula CTL en forma normal existencial (véase la Definición 1.10):

$$\mathrm{EX}\, q \ \wedge\ \mathrm{EG}(\neg\, q \ \vee\ \neg\, r)$$

Un árbol de derivación es un árbol que representa la estructura sintáctica de una expresión definida inductivamente. Para una fórmula arbitraria CTL, los subárboles del árbol de derivación indican las subfórmulas que lo conforman, la raíz es la fórmula principal que se está estudiando. El árbol de derivación de la fórmula anterior se muestra en la Figura 2.1.

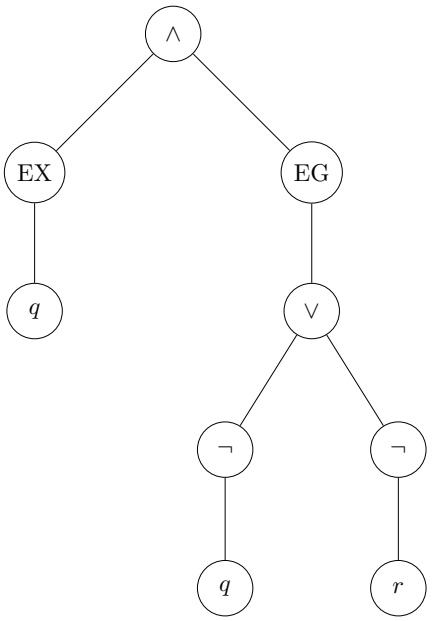

Figura 2.1: Árbol de derivación de la expresión $\mathrm{EX}\, q \ \wedge\ \mathrm{EG}(\neg\, q \vee \neg\, r)$.

Bajo esta interpretación, el conjunto de satisfacción un nodo dependerá directamente de los conjuntos de satisfacción de sus hijos. El caso base del algoritmo son las hojas, que siempre estarán conformadas por variables o constantes proposicionales. El conjunto de satisfacción para variables proposicionales se obtiene directamente y se trata de todos los estados donde se satisface la variable proposicional que aparece en la hoja, los conjuntos de satisfacción de $\top$ y $\bot$ son S y $\varnothing$, respectivamente. Inductivamente, para los nodos que no son hoja, se combinan los conjuntos de satisfacción de sus nodos hijos una vez que estos se han obtenido utilizando la semántica descrita para la lógica CTL.

La función $Ant(X)$ es de utilidad para introducir conceptos futuros. Esta obtiene el conjunto de estados que tienen como sucesor un estado que pertenece a un conjunto arbitario X.

Definición 2.1 (Función anterior)**.** Sea $\mathcal{M} = (S, \longrightarrow, I, AP, L)$ un diagrama de transiciones y $s \in S$, la función Ant se define de la siguiente manera:

$$Ant(s) = \{q \ \in\ S \mid \text{ existe una transición } q \to s\}$$

de manera similar, si $X \subseteq S$:

$$Ant(X) = \{q \ \in\ S \mid \text{ existe una transición } q \to r \ \text{ y } r \in X\}$$

Con esta definición, ya podemos describir un proceso algorítmico para determinar el conjunto de satisfacción de cualquier fórmula CTL:

Algoritmo 2.1 Conjunto de satisfacción

```
1: def satSet(M, φ):
2:     if φ == ⊤ :
3:         return S
4:     else if φ ∈ AP:
5:         return {s ∈ S | φ ∈ L(s)}
6:     else if φ == ¬φ1:
7:         return S \ satSet(M, φ1)
8:     else if φ == φ1 ∧ φ2:
9:         return satSet(M, φ1) ∩ satSet(M, φ2)
10:    else if φ == φ1 ∨ φ2:
11:        return satSet(M, φ1) ∪ satSet(M, φ2)
12:    else if φ == φ1 → φ2:
13:        return satSet(M, ¬φ1 ∨ φ2)
14:    else if φ == φ1 ↔ φ2:
15:        return satSet(M, (φ1 → φ2) ∧ (φ2 → φ1))
16:    else if φ == φ1 ⊕ φ2:
17:        return satSet(M, (¬φ1 ∧ φ2) ∨ (φ1 ∧ ¬φ2))
18:    else if φ == AX φ1:
19:        return satSet(M, ¬EX¬φ1)
20:    else if φ == EX φ1:
21:        return satSetEX(M, φ1)
22:    else if φ == AG φ1:
23:        return satSet(M, ¬EF¬φ1)
24:    else if φ == EG φ1:
25:        return satSetEG(M, φ1)
26:    else if φ == AF φ1:
27:        return satSet(M, ¬EG¬φ1)
28:    else if φ == EF φ1:
29:        return satSet(M, E[ ⊤ U φ1])
30:    else if φ == A[ φ1 U φ2]:
31:        return satSet(M, ¬E[¬φ2U(¬ φ1 ∧ ¬ φ2)] ∧ ¬EG¬φ2)
32:    else if φ == E[ φ1U φ2]:
33:        return satSetEU(M, φ1, φ2)
```

El Algoritmo 2.1 está basado en otras tres subrutinas que corresponden a la obtención de los conjuntos de satisfacción para los operadores presentes en la Forma Normal Existencial, el pseudocódigo de estos se muestra a continuación:

Algoritmo 2.2 Conjunto de satisfacción de EX

```
1: def satSetEX(M, φ):
2:     X = satSet(φ)
3:     Y = Ant(X)        # Todos los estados anteriores
4:     return Y
```

Algoritmo 2.3 Conjunto de satisfacción de EG

```
1: def satSetEG(M, φ):
2:     X = ∅
3:     Y = satSet(φ)
4:     while X ≠ Y :
5:         X = Y
6:         Y = Y ∩ Ant(Y)        # Todos los estados con una transición hacia Y
7:     return Y
```

Algoritmo 2.4 Conjunto de satisfacción de EU

```
1: def satSetEU(M, φ1, φ2):
2:     X = ∅
3:     Y = satSet(φ2)
4:     Z = satSet(φ1)
5:     while X ≠ Y :
6:         X = Y
7:         Y = Y ∪ (Z ∩ Ant(Y))        # Todos los estados anteriores donde se cumple φ1
8:     return Y
```

2.2. Algoritmos de verificación con equidad

A pesar de tener la capacidad de expresar muchas propiedades útiles sobre sistemas computacionales, la lógica CTL posee limitaciones para describir otras que también resultan importantes. Una especialmente relevante es la incapacidad para expresar propiedades de *equidad*. Las propiedades de equidad buscan eliminar ciertos caminos de ejecución de un sistema que podrían considerarse "injustos" si se tratara de un dispositivo real. Para ejemplificar el concepto imaginemos una ventanilla de atención en un banco que cuenta con dos filas que atienden a clientes distintos, una fila es para clientes "preferentes" y la otra para clientes "convencionales", además, imaginemos que existe una política empresarial en la que, de haber una persona esperando en la fila preferente, los asistentes del banco siempre la deberán de atender

antes que la persona que se encuentre en la fila convencional. Si los clientes preferentes del banco no pararan de llegar a la sucursal, los clientes convencionales nunca serían atendidos, lo que indicaría una operación injusta del sistema que idealmente debería de ser eliminada. Técnicamente, a esta situación se le conoce como *inanición* y es un ejemplo común de un sistema donde no se cumple la propiedad de equidad.

Matemáticamente, la situación descrita en el párrafo anterior podría modelarse de la siguiente manera: asignemos la fórmula ϕ a la situación en la que se atiende al siguiente cliente que se encuentra en la fila convencional. Eliminar las ejecuciones injustas sería equivalente a no considerar aquellos caminos en los que nunca se cumple ϕ. Una posible fórmula CTL para describir esto podría ser $\mathrm{AG}\,(\,\mathrm{AF}\phi \rightarrow \beta\,)$ donde β es la propiedad que queremos probar solamente en los caminos "justos". Esta fórmula intenta eliminar de la consideración de β a aquellos estados en los que, para algún camino no se cumple ϕ en ningún estado futuro. El problema con esta especificación es que, si existe algún estado donde se da esta situación, todos los caminos que parten de ese estado se eliminarían aunque solamente uno fuera injusto, eliminando de la consideración caminos que deberían de analizarse para β. Esta limitación para seleccionar caminos específicos se debe a que, toda propiedad de camino en CTL (F, X, G, etc.) siempre es precedida por un cuantificador (A y E), lo que imposibilita el poder discriminar caminos de ejecución individualmente. Si queremos incluir este tipo de propiedades dentro de un verificador CTL, tenemos que ir más allá de los algoritmos descritos en la sección anterior.

La siguiente definición (tomada de [5, pág. 231]) clarifica esta idea de equidad, generalmente llamada *equidad incondicional* [4, pág. 129] o *justicia* [20, págs. 24–25]:

Definición 2.2 (Camino con equidad incondicional o justicia)**.** Sea $\mathcal{M} = (S, \longrightarrow, I, AP, L)$ un diagrama de transiciones arbitrario, $equi = \{\phi_1, \phi_2, \ \ldots\ , \phi_n\}$ un conjunto de n fórmulas CTL llamadas *restricciones de equidad* y π un camino cualquiera en $\mathcal{M}$. π es un camino justo con respecto a $equi$, denotado como $\underset{equi}{\pi}$, syss para toda restricción ϕ_i existe un número infinito de estados $\pi[j]$ sobre π tales que $\pi[j] \models \phi_i$. Es decir, que para todo estado $\pi[k]$ con $k \geq 0$ sobre π existe $l \geq k$ tal que $\pi[l] \models \phi_i$. Se llamará $\underset{equi}{Cam}(\mathcal{M})$ al conjunto de todos los caminos justos en $\mathcal{M}$ respecto a $equi$ y se llamará $\underset{equi}{Cam}(s)$ al conjunto de todos los caminos justos en $\mathcal{M}$ tales que $\underset{equi}{\pi}[0] = s$ con $s \in S$.

La siguiente definición describe de manera general la semántica CTL con restricciones de equidad añadidas.

Definición 2.3 (Semántica CTL bajo equidad incondicional o justicia)**.** Sea $\mathcal{M}$ un diagrama de transiciones arbitrario, $equi = \{\phi_1, \phi_2, \ \ldots\ , \phi_n\}$ un conjunto de n fórmulas CTL llamadas restricciones de *equidad* y ϕ una fórmula CTL cualquiera. $\mathcal{M}$ satisface a ϕ bajo $equi$, denotado como $\mathcal{M} \underset{equi}{\models} \phi$ syss se cumple con la descripción hecha en la Definición 1.6 para satisfacción de la lógica CTL, con la única diferencia siendo que los operadores de cuantificación existencial (E) y universal (A) únicamente cuantifican sobre caminos con equidad $equi$ de acuerdo con la Definición 2.2. Los operadores temporales bajo la restricción $equi$ se reescribirán como $\underset{equi}{\mathrm{AX}}$, $\underset{equi}{\mathrm{AF}}$, etc. La relación de satisfacción bajo equidad $\underset{equi}{\models}$ podrá reescribirse como la relación

de satisfacción convencional $\models$, si es evidente que todos los operadores temporales presentes en la fórmula a evaluar se han sustituido por sus versiones "justas".

De la misma manera que para los operadores convencionales, existe la Forma Normal Existencial cuando se consideran restricciones de equidad.

Teorema 2.1 (Transformación a Forma Normal Existencial bajo restricciones de equidad)**.** Las equivalencias descritas en el Teorema 1.2 se cumplen de la misma manera bajo restricciones de equidad arbitrarias *equi*.

Demostración. La única diferencia que existe entre los operadores tradicionales y los que se encuentran bajo la restricción *equi* es que la cuantificación se realiza sobre caminos justos ($\underset{equi}{Cam}$) en lugar de caminos convencionales (Cam). Las equivalencias en lógica de primer orden se siguen satisfaciendo bajo esta nueva interpretación y los razonamientos metalógicos siguen siendo válidos también. □

El proceso de verificación bajo restricciones de equidad puede ser simplificado aún más si se analiza con detenimiento la naturaleza del operador $\underset{equi}{\mathrm{EG}}$. $\underset{equi}{\mathrm{EG}}\,\top$ determina si, para un estado cualquiera, existe un camino donde se cumple $\top$ de manera indefinida (que sabemos que son todos los caminos), bajo la condición de que se cumplan las restricciones de equidad *equi* en él, es decir, los estados que satisfacen esta fórmula son aquellos que tienen al menos un camino justo que emana de ellos. Todo estado que satisface alguna fórmula en FNE bajo las restricciones *equi* debe de satisfacer $\underset{equi}{\mathrm{EG}}\,\top$. Utilizando esta operación, podemos definir $\underset{equi}{\mathrm{E[U]}}$ en términos de E[U] y de $\underset{equi}{\mathrm{EG}}$ y también podemos definir $\underset{equi}{\mathrm{EX}}$ en términos de EX de $\underset{equi}{\mathrm{EG}}$. Formalmente:

Teorema 2.2 (Interdependencia de operadores existenciales con equidad)**.** Las siguientes equivalencias se cumplen para operadores existenciales con equidad:

1. $\underset{equi}{\mathrm{EX}}\,\phi \;\equiv\; \mathrm{EX}(\phi \wedge \underset{equi}{\mathrm{EG}}\,\top)$
2. $\underset{equi}{\mathrm{E}}[\phi_1\ \mathrm{U}\ \phi_2] \;\equiv\; \mathrm{E}[\phi_1\ \mathrm{U}\ (\phi_2 \wedge \underset{equi}{\mathrm{EG}}\,\top)]$

Este es un resultado conocido y común en la bibliografía sobre la verificación de modelos, por lo que se omite la demostración por razones de extensión. Si se desea un desarrollo detallado de todos los elementos necesarios para obtenerlo se puede consultar [4, pág. 366].

Con este teorema se puede reducir el problema de la verificación de modelos con restricciones de equidad *equi* al problema de verificación convencional junto con la adición de un algoritmo $\underset{equi}{\mathrm{EG}}\,\phi$ para la obtención de los caminos justos. El siguiente teorema establece un criterio para determinar los estados que cumplen con $\underset{equi}{\mathrm{EG}}\,\phi$ en una estructura de Kripke $\mathcal{M} = (S, \longrightarrow, I, AP, L)$, vista $\mathcal{M}$ como un grafo $G = (V, E)$ donde $V = S$ y $E = \longrightarrow$:

Teorema 2.3 (Satisfacción de $\underset{equi}{\mathrm{EG}}\phi$)**.** Sea s un estado de un diagrama de transiciones arbitrario $\mathcal{M}$, y sea $equi = \{\phi_1, \phi_2, ..., \phi_n\}$, $s \models \underset{equi}{\mathrm{EG}}\,\phi$ syss existe un camino que comienza con el fragmento: $s_0 \to s_1 \to ... \to s_k$ y un ciclo $\sigma_0 \to \sigma_1 \to ... \to \sigma_l$ tal que:

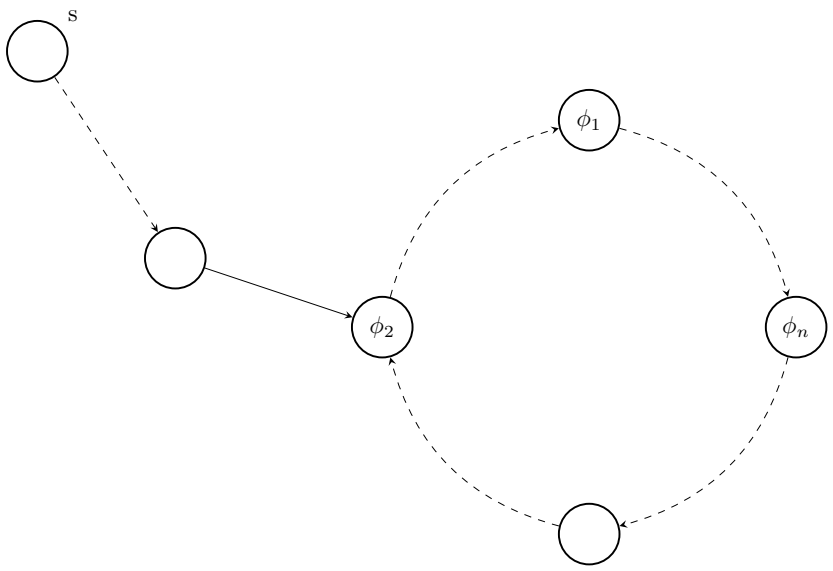

Figura 2.2: Descripción gráfica de $s \models \underset{equi}{\mathrm{EG}}\ \top$

1. $\forall\ 0 \leq j \leq k\ (s_j \models \phi)$ y $\forall\ 0 \leq i \leq l\ (\sigma_i \models \phi)$, es decir, ϕ se satisface en todos los estados.

2. $s = s_0$ y $s_k = \sigma_0 = \sigma_l$.

3. Existe al menos un $0 \leq i \leq l$ tal que $\sigma_i \models \phi_r$ para todo $1 \leq r \leq n$.

En otras palabras, el camino tiene un componente fuertemente conexo[1] que se repite indefinidamente en el cual se satisfacen todas las fórmulas de equidad.

Demostración. $\Rightarrow$: Supongamos $s \models \underset{equi}{\mathrm{EG}}\ \phi$, el camino infinito que satisface la fórmula es $s \to s_1 \to ...$ y por la definición de camino justo bajo *equi* sabemos que, para todo estado del camino s_p, existe un estado futuro s_t $(t \geq p)$ tal que $s_t \models \phi_r$. Como el número de estados del diagrama de transiciones es finito, los estados donde ϕ_r se cumple se deben de repetir indefinidamente, lo que obliga a la existencia del ciclo (punto 2). El ciclo debe de cerrarse en algún momento, lo que forza al punto 1. Como se trata de la fórmula EGϕ, ϕ se debe de satisfacer en todo estado del camino.
$\Leftarrow$:
Supongamos que existe el fragmento finito $s_0 \to s_1 \to ... \to s_k$ y el ciclo $\sigma_0 \to \sigma_1 \to ... \to \sigma_l$ con las características especificadas en los puntos 1 y 2. De acuerdo con el punto 1, el ciclo es alcanzable por el estado inicial del camino s y, de acuerdo con el punto 2, en el ciclo se cumplen indefinidamente todas las fórmulas de equidad ϕ_r. Como todo estado en el camino satisface a ϕ, entonces el estado cumple con $\underset{equi}{\mathrm{EG}}\ \phi$. □

[1]Para la definición de este y otros conceptos de grafos véase el Apéndice A.1.

Como se menciona en el punto 1 del teorema anteriormente demostrado, solamente aquellos estados que tengan al menos un camino en el que se cumpla indefinidamente ϕ podrán satisfacer a $\underset{equi}{\mathrm{EG}}\ \phi$ por lo que, dado $\mathcal{M} = (S,\ \longrightarrow,\ I,\ AP,\ L)$, debemos restringir S y $\longrightarrow$ de la siguiente manera:

$$S_\phi \ = \ \{s \in S \mid s \models \phi\}$$

$$\longrightarrow_\phi \ = \{(s, s') \in \longrightarrow \ \mid \ s \models \phi \ \ y \ \ s' \models \phi\}$$

Se puede describir ya un algoritmo para la satisfacción de este problema:

Algoritmo 2.5 Conjunto de satisfacción $\underset{equi}{\mathrm{EG}}\ \phi$

```
1: def satSetEGEqui(M, φ, [φ_1, ... , φ_n]):
2:     S = S_φ        # Obtenemos el diagrama restringido
3:     → = →_φ
4:     equiC = ∅      # Componentes fuertemente conexos justos
5:     SCC = Componentes fuertemente conexos en M
6:     for scc in SCC :
7:         for i in [1, ... , n] :
8:             if ∃s ∈ scc (s ⊨ φ_i) :
9:                 if i == n :
10:                    equiC = equiC ∪ scc      # scc cumple con todo φ_i ∈ [φ_1, ... , φ_n]
11:                else:
12:                    continue
13:            else:
14:                break
15:        # equiC ya contiene todos los componentes fuertemente conexos justos
16:    return {s ∈ S | s puede alcanzar a cualquier estado en equiC}
```

Donde el concepto de alcance refiere a la existencia de un camino mediante el cual se pueda llegar de un estado a otro de $\mathcal{M}$.

Con este procedimiento, el problema de verificación de modelos ϕ bajo n restricciones de equidad arbitrarias $equi = \{\phi_1,\ \phi_2,\ \ldots\ ,\ \phi_n\}$ se podría subdividir en los siguientes pasos:

1. Transformar ϕ y toda fórmula en $equi$ a forma normal existencial.

2. Reescribir ϕ utilizando las equivalencias descritas en el Teorema 2.2, de manera que la fórmula únicamente se encuentre en términos de los operadores existenciales sin equidad y de $\underset{equi}{\mathrm{EG}}\ \phi$ (un caso particular de esta fórmula es $\underset{equi}{\mathrm{EG}}\ \top$).

3. Realizar la verificación de modelos con equidad utilizando el Algoritmo 2.5 y los algoritmos de verificación convencional (Algoritmos [2.1 - 2.4]).

2.3. Verificación simbólica

Cuando se introdujo la semántica de la lógica CTL en la Sección 1.2.2 se mencionó que, dado un diagrama de transiciones arbitrario $\mathcal{M} = (S, \longrightarrow, I, AP, L)$, un estado $s \in S$ sería identificado por un subconjunto único de AP, por lo que si $|AP| = n$, el número de posibles estados en $\mathcal{M}$ puede ser de hasta 2^n. Como las variables proposicionales generalmente denotan situaciones de interés en el sistema, esta característica produce un crecimiento potencialmente exponencial en el número de estados conforme se agregan variables de interés. A este fenómeno se le conoce como el "problema de la explosión de estados" y hace que la verificación de modelos se vuelva intratable incluso en sistemas relativamente pequeños.

La forma en la que se intentará lidiar con este problema durante el presente trabajo es mediante la codificación de la estructura de Kripke utilizando variables y funciones booleanas. Se definirán las variables y funciones booleanas así como ciertas operaciones útiles sobre ellas y se mostrará una estructura de datos eficiente para representarlas en una computadora: los diagramas de decisión binarios. Utilizando estos elementos, se describirá cómo hacer la codificación de la estructura y cómo resolver el problema de la verificación de modelos en este nuevo dominio. Se espera que utilizando esta nueva codificación se puedan representar conjuntos de estados de manera compacta, así como explotar ciertas regularidades en la estructura de Kripke que nos permitan lidiar con sistemas considerablemente más grandes que las representaciones explícitas basadas en grafos.

2.3.1. Funciones booleanas

Definición 2.4 (Variable y función booleana)**.** Una variable booleana, denotada por los símbolos en el conjunto $var = \{x, y, z, x_1, y_1, z_1, x_2, y_2, z_2, ...\}$ es una variable que puede tomar valores en el dominio booleano $\mathbf{B} = \{0, 1\}$. Una función booleana ψ es una función de la forma $\psi : \mathbf{B}^n \to \mathbf{B}$.
Las funciones booleanas se pueden definir inductivamente de la siguiente manera:

1. 0 y 1 son funciones booleanas sin argumentos que toman el valor de la constante.

2. Una variable cualquiera $x \in var$ es una función booleana cuyo valor es el asignado a x.

3. Sean ψ, ψ_1 y ψ_2 tres funciones booleanas cualquiera:

   - $\neg\, \psi$ es una función booleana con valor 1 syss $\psi = 0$. A este operador se le conoce como *negación*.
   - $\psi_1 + \psi_2$ es una función booleana cuyo valor es 0 syss $\psi_1 = 0$ y $\psi_2 = 0$. A este operador se le conoce como *suma*.
   - $\psi_1 \bullet \psi_2$ es una función booleana cuyo valor es 1 syss $\psi_1 = 1$ y $\psi_2 = 1$. A este operador se le conoce como *multiplicación*.
   - $\psi_1 \oplus \psi_2$ es una función booleana cuyo valor es 1 syss los valores de ψ_1 y ψ_2 son diferentes. A este operador se le conoce como *suma exclusiva*.

Como es convencional, denotaremos una función booleana cualquiera ψ con $f(v_1, v_2, \ldots, v_n)$ donde las variables $v_1, v_2, \ldots, v_n$ son las variables de las que depende la función ψ. Dado un elemento cualquiera $A = (a_1, a_2, \ldots a_n) \in \mathbf{B}^n$ del dominio de ψ, denotaremos mediante $\psi(A)$ ó $\psi(a_1, a_2, \ldots, a_n)$ a la evaluación de la función ψ para A.

De manera similar a como lo hicimos para la lógica CTL, definiremos operadores booleanos derivados.

Definición 2.5 (Operadores booleanos derivados)**.**

$$\psi_1 \Rightarrow \psi_2 \stackrel{\text{def}}{=} (\neg\, \psi_1) + \psi_2$$
$$\psi_1 \Leftrightarrow \psi_2 \stackrel{\text{def}}{=} \neg\, (\psi_1 \oplus \psi_2)$$

Una operación que nos será útil para definir conceptos futuros es la operación que nos da la capacidad de "renombrar" o "reemplazar" una variable por otra en una función booleana.

Definición 2.6 (Operador renombre)**.** Sea $\psi = f(v_1, v_2, \ldots, v_n)$ una función booleana cualquiera, definimos al operador *renombre*, escrito como $\psi\,[v_i = v]$ donde $1 \leq i \leq n$ y v es una variable cualquiera $v \in var$, como la reescritura de toda ocurrencia de la variable v_i en la fórmula ψ por la variable v. Extendiendo esta notación, la función booleana $(\ldots((\psi\,[v_{l_1} = v_{s_1}])[v_{l_2} = v_{s_2}])\ldots)[v_{l_m} = v_{s_m}]$ se reescribirá como $\psi\,[v_{l_1} = v_{s_1}, v_{l_1} = v_{s_1}, \ldots, v_{l_m} = v_{s_m}]$.

Definición 2.7 (Operadores sustitución, para todo y existe)**.** Sea $\psi = f(v_1, v_2, \ldots, v_n)$ una función booleana cualquiera, definimos al operador *sustitución*, denotado por $\psi|_{v_k=c} = f(v_1, v_2, \ldots, v_{k-1}, v_{k+1}, \ldots, v_n)$ donde $c \in \mathbf{B}$ y $1 \leq k \leq n$, como la sustitución de toda presencia de la variable v_k en ψ por c. Se representará a $(\ldots((\psi|_{v_{s_1}=c_1})|_{v_{s_2}=c_2})\ldots)|_{v_{s_m}=c_m}$ mediante $\psi|_{[v_{s_1}=c_1, v_{s_2}=c_2, \ldots, v_{s_m}=c_m]}$.

La operación *existe* se define de la siguiente manera:

$$\exists v.\psi \;=\; \psi|_{v=0} \;+\; \psi|_{v=1}$$

Denotando mediante $\exists[v_1, v_2, \ldots, v_n].\psi$ a $\exists v_n.(\ldots(\exists v_2.(\exists v_1.\psi))\ldots)$ La operación *para todo* a su vez como:

$$\forall v.\psi \;=\; \psi|_{v=0} \;\bullet\; \psi|_{v=1}$$

Donde la abreviación para la aplicación iterada del operador a una función es similar a la de la función existe.

Un concepto esencial es de la equivalencia entre dos funciones booleanas, para definirla, necesitamos algunas otras definiciones.

Definición 2.8 (Tautología y contradicción)**.** Sea $\psi = f(v_1, v_2, \ldots, v_n)$ una función booleana cualquiera, decimos que ψ es una *tautología* si, para todo elemento $A \in \mathbf{B}^n$ en el dominio de la función se cumple que $\psi(A) = 1$. Decimos que ψ es una *contradicción* si $\psi(A) = 0$.

Definición 2.9 (Equivalencia de funciones)**.** Sea ψ_1 y ψ_2 dos funciones booleanas cualesquiera, decimos que ψ_1 y ψ_2 son *equivalentes*, denotado por $\psi_1 \equiv \psi_2$, syss $\psi_1 \oplus \psi_2$ es una contradicción.

A continuación se establece la siguiente convención para simplificar la notación de las funciones booleanas. Los operadores unarios tienen mayor precedencia que los operadores binarios y en los operadores binarios, la precedencia en orden decreciente es: $\bullet$, $+$, $\oplus$, $\Rightarrow$, $\Leftrightarrow$ y $\equiv$.

Teorema 2.4 (Teorema de expansión de Boole)**.** Sea $\psi = f(v_1, v_2, \ldots, v_n)$ cualquier función booleana de n variables, se cumple la siguiente equivalencia:

$$\psi \;\equiv\; v_k \bullet \psi|_{v_k=1} \;+\; \neg\, v_k \bullet \psi|_{v_k=0}$$

donde $1 \leq k \leq n$.

Demostración. Tomemos una tupla cualquiera $A = (a_1, a_2, \ldots, a_n) \in \mathbf{B}^n$ del dominio de ψ. De acuerdo con la definición de equivalencia:

$$\psi \;\oplus\; v_k \bullet \psi|_{v_k=0} \;+\; \neg\, v_k \bullet \psi|_{v_k=0}$$

Sustituyendo el valor de las variables en la función:

$$\begin{aligned}\psi(a_1, a_2, \ldots, a_k, \ldots, a_n) \;\oplus\; & a_k \bullet \psi|_{v_k=1}(a_1, a_2, \ldots, a_{k-1}, a_{k+1}, \ldots, a_n) \\ +\; & \neg\, a_k \bullet \psi|_{v_k=0}(a_1, a_2, \ldots, a_{k-1}, a_{k+1}, \ldots, a_n)\end{aligned}$$

Evaluando el caso en el que $a_k = 0$:

$$\begin{aligned}\psi(a_1, a_2, \ldots, 0, \ldots, a_n) \;\oplus\; & 0 \bullet \psi|_{v_k=1}(a_1, a_2, \ldots, a_{k-1}, a_{k+1}, \ldots, a_n) \\ +\; & 1 \bullet \psi|_{v_k=0}(a_1, a_2, \ldots, a_{k-1}, a_{k+1}, \ldots, a_n)\end{aligned}$$

Utilizando las definiciones de los operadores booleanos podemos simplificar la expresión anterior de la siguiente manera:

$$\psi(a_1, a_2, \ldots, 0, \ldots, a_n) \;\oplus\; \psi|_{v_k=0}(a_1, a_2, \ldots, a_{k-1}, a_{k+1}, \ldots, a_n)$$

El valor del lado derecho y del izquierdo de la suma exclusiva anterior es el mismo ya que, si se evalúa la función con A sustituyendo 0 por v_k, o si primero se sustituye v_k por 0 y luego se evalúa la función con $(a_1, a_2, \ldots, a_{k-1}, a_{k+1}, \ldots, a_n)$, el resultado es el mismo.
Para la demostración con $a_k = 1$ el proceso es dual, por lo que la equivalencia se cumple. □

2.3.2. Diagramas de decisión binarios

Primero se introducirán los árboles de decisión binarios y, a partir de estos, los diagramas de decisión binarios.

Definición 2.10 (Árbol de decisión binario)**.** Un árbol de decisión binario (*BDT*, *Binary Decision Tree*) es un árbol binario completo $T = (V, E)$[2] al que se añaden las siguientes propiedades:

[2]Para la definiciones formales de árbol, árbol binario y árbol completo véase el Apéndice A.2

- El conjunto de nodos V está subdividido en dos conjuntos disjuntos V_h (el conjunto de nodos hoja) y V_p (el conjunto de nodos padre).
- El conjunto de aristas E está subdividido en dos conjuntos disjuntos E_0 y E_1 de manera que cada uno forma una función[3]:
  - $E_0 : V_p \to V$,
  - $E_1 : V_p \to V$.
- Existen dos funciones de etiquetado:
  - $l_h : V_h \to \{0, 1\}$, que asigna a cada nodo hoja una constante booleana,
  - $l_p : V_p \to var$, que asigna a cada nodo padre una variable booleana.

En los árboles de decisión binarios, un camino del nodo raíz a un nodo hoja está determinado por una asignación A de las variables asociadas a los nodos padres en V_p. Comenzando con el nodo raíz, para cualquier nodo padre v_p en el árbol, el siguiente nodo en el camino de la raíz a una hoja estará determinado por la asignación que A hace de la variable $x = l_p(v_p)$, si $x = 0$ en A, entonces el siguiente nodo será $E_0(v_p)$, si $x = 1$ en A, el siguiente nodo será $E_1(v_p)$. Gráficamente, la arista que une a un vértice padre con un vértice hijo suele ser punteada si tiene asociado el valor 0 y sólida si tiene asociado a 1. La Figura 2.3 muestra un ejemplo de un árbol de decisión binario asociado a una función booleana bajo esta convención.

El siguiente resultado nos permite crear un árbol de decisión binario para toda fórmula booleana.

Teorema 2.5 (Existencia de un árbol binario asociado a toda función booleana)**.** Dada una función booleana cualquiera $\psi = f(v_1, v_2, \ldots, v_n)$, se puede construir al menos un árbol de decisión binario tal que, para cualquier elemento $A = (a_1, a_2, \ldots, a_n)$ del dominio de ψ, existe en el árbol un camino de la raíz a alguna hoja determinado por la asignación A como se mostró anteriormente, tal que la constante booleana asociada a esa hoja por l_h es exactamente el valor numérico $\psi(A)$.

La demostración de este teorema hace uso de la construcción del árbol utilizando reiteradamente el Teorema de Expansión de Boole, así como de la posibilidad de la existencia de nodos redundantes [4, pág. 385] [5, pág. 361].

Los diagramas de decisión binarios (*BDD*, *Binary Decision Diagrams*) consisten en una representación compacta de los árboles de decisión binarios. Los BDD se obtienen a partir de la eliminación de ciertos elementos redundantes en los árboles:

1. Eliminación de nodos padre repetidos: puede existir el caso en el que dos o más nodos distintos de un árbol de decisión binario formen exactamente el mismo subárbol, esta repetición se puede eliminar redirigiendo las aristas que llevan a la raíz de estos subárboles a un solo subárbol idéntico a estos, eliminando de esta manera la repetición.

[3]Para que se pueda cumplir la condición de función, forzosamente una de las dos aristas asociadas a un padre debe de pertenecer a E_0 y la otra a E_1.

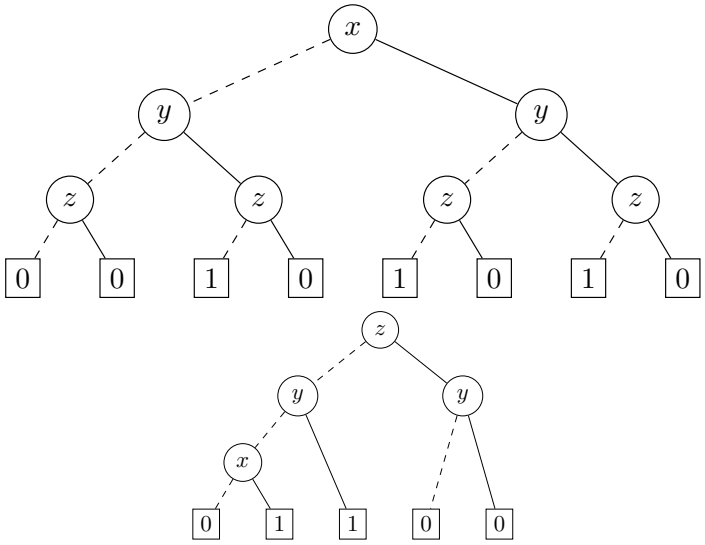

Figura 2.3: Dos representaciones distintas, en forma de árbol de decisión binario, de la función booleana $(x + y) \bullet \neg z$.

2. Eliminación de decisiones redundantes: cuando los dos hijos de un nodo en un árbol de decisión binario sean exactamente el mismo nodo, el valor de la variable asociada a dicho nodo no tiene importancia en el hijo siguiente y, por lo tanto, el nodo puede eliminarse.

3. Eliminación de nodos hoja repetidos: un árbol de decisión binario puede tener dos o más nodos hoja que tengan asociada la misma etiqueta numérica (véase Figura 2.3). En lugar de mantener todos estos nodos separados, se pueden utilizar únicamente dos localidades de memoria con los valores numéricos 0 y 1 y redirigir todos los nodos únicamente a estas dos localidades.

Como se explica en el Apéndice A.2, una de las propiedades que definen a los árboles es la existencia de un camino único que une a la raíz con cualquier nodo del árbol, la aplicación del punto 3 de la lista anterior elimina esta propiedad, por lo que la nueva estructura de datos pierde el nombre de *árbol* y recibirá el nombre de *diagrama*.

Definición 2.11 (Diagrama de decisión binario)**.** Un diagrama de decisión binario es un grafo acíclico dirigido que cuenta con un único vértice inicial y donde todo vértice no terminal v cumple con $|Post(v)| = 2$. Además, se cumple que:

- El conjunto de vértices V está subdividido en dos conjuntos V_n (vértices no terminales) y V_t (vértices terminales).
- El conjunto de aristas E está subdividido en dos conjuntos disjuntos E_0 y E_1 de manera que cada uno forma una función:
  - $E_0 : V_n \to V$,

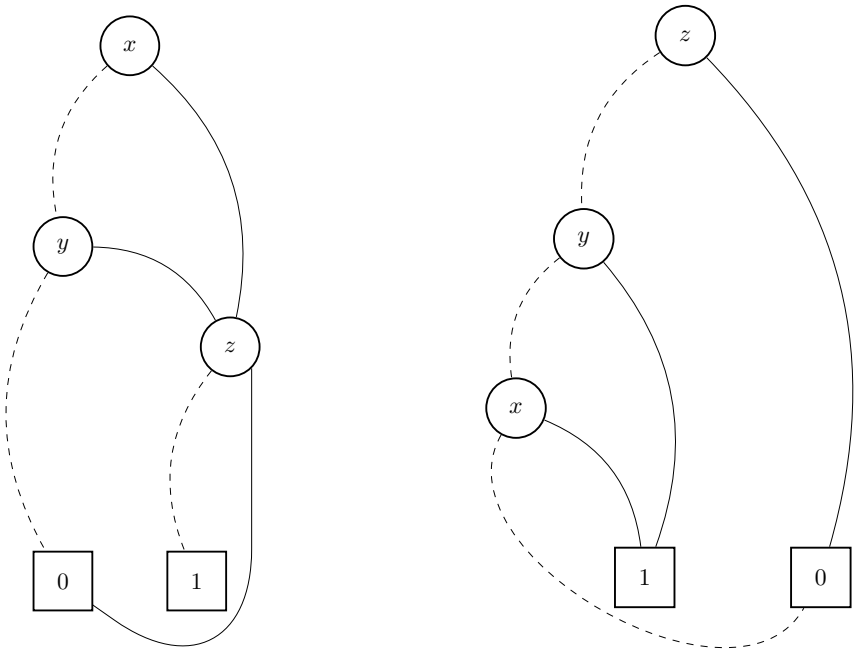

Figura 2.4: Diagramas de decisión binarios reducidos obtenidos de los árboles de la Figura 2.3.

- $E_1 : V_n \to V$.

- Existen dos funciones de etiquetado:
  - $l_t : V_t \to \{0, 1\}$,
  - $l_n : V_n \to var$.

La interpretación de los BDD se realiza de la misma manera que para los BDT.

Definición 2.12 (BDD reducido o RBDD)**.** Se dirá que un diagrama de decisión binario está *reducido* (*RBDD*, *Reduced Binary Decision Diagram*) si ya no es posible aplicar en él alguna de la reglas de eliminación 1-3.

La Figura 2.4 muestra los BDD reducidos obtenidos de los árboles en la Figura 2.3.

Una operación que resultará computacionalmente útil más adelante es la de probar la equivalencia entre dos RBDD, sin embargo, la representación con la que contamos actualmente no permite hacer esto de manera sencilla. A pesar de que los RBDD de la Figura 2.4 representan a la misma función booleana $((x + y) \bullet \neg z)$, estos son notoriamente distintos y no parece sencillo determinar esta equivalencia. Una forma de lograr una representación única o *canónica* de un RBDD es forzando un ordenamiento en la ocurrencia de las variables para cualquier camino del nodo inicial a los nodos terminales.

Definición 2.13 (Ordenamiento de variables)**.** Sea $var_1 \subset var$ un conjunto finito de variables booleanas de tamaño n. Un ordenamiento $l = [u_1, u_2, \ ... \ , u_n]$ es una lista ordenada que

contiene a todas las variables presentes en var_1. l además induce un orden total estricto $<_l$ [4] sobre var_1, donde $u_i <_l u_j$ syss $i < j$, esto es, si u_i ocurre antes de u_j en l.

Definición 2.14 (Ordenamientos compatibles)**.** Sean l_1 y l_2 dos ordenamientos. Se dice que l_1 y l_2 son compatibles syss no existen dos variables v y w distintas tales que $v <_{l_1} w$ y $w <_{l_2} v$.

Definición 2.15 (Ordenamiento en un BDD)**.** Sea B un BDD cualquiera, sea $l = [u_1, u_2, \ ... \ , u_n]$ un ordenamiento del conjunto V_n de B y $\pi = v_0\, v_1\, v_2 \ldots v_r$ un camino cualquiera en B. Se dirá que B se encuentra ordenado de acuerdo con l (*OBDD*, *Ordered Binary Decision Diagram*) si, para cualesquiera vértices v_p y v_q en π tales que v_p ocurre antes que v_q en π (esto es: $p < q$ y $p, q \in [0, r]$) entonces la variable u_i asociada al vértice v_p ocurre antes que la variable u_j asociada al vértice v_q en la lista l ($l_n(u_i) <_l l_n(u_j)$).

El siguiente Teorema es un resultado importante:

Teorema 2.6 (Canonicidad de un OBDD)**.** Sea una función booleana cualquiera $\psi = f(v_1, v_2, \ ... \ , v_n)$, y sea una $l = [u_1, u_2, \ ... \ , u_n]$ un ordenamiento cualquiera de las variables presentes en ψ. Existe solamente un RBDD asociado a ψ que se encuentre ordenado (*ROBDD*, *Reduced Ordered Binary Decision Diagram*) de acuerdo con l. La demostración de este teorema puede ser consultada en [6, pág. 34].

Algoritmo de reducción

Existe un algoritmo inductivo para reducir un OBDD que aplica de manera ordenada las reducciones 1-3 mostradas para BDT comenzando por los vértices iniciales y subiendo de manera que todas las reducciones ya hayan sido aplicadas a los nodos hijos de un nodo cualquiera. El algoritmo procede de la siguiente manera:

1. Vértices terminales: Si para dos vértices $v_i, v_j \in V_t$ se cumple que $l_t(v_i) = l_t(v_j)$, entonces dichos vértices son equivalentes y se deben de colapsar en uno solo. Esto se hace convirtiendo las aristas $E_0(v) = v_j$ a $E_0(v) = v_i$ y $E_1(v) = v_j$ a $E_1(v) = v_i$ para cualquier nodo v, eliminando el nodo v_j de V_t posteriormente.

2. Eliminación de decisiones redundantes: Si para un vértice $v \in V_n$ se cumple que $E_0(v) = E_1(v) = u$, entonces el vértice v realiza una decisión redundante y esta se debe de eliminar. Esto se hace convirtiendo las aristas $E_0(w) = v$ a $E_0(w) = u$ y $E_1(w) = v$ a $E_1(w) = u$ para cualquier nodo w y eliminar el nodo v de V_t.

[4] Un *orden estricto* [21] $<$ es una relación binaria sobre un conjunto A que es:

1. Irreflexiva: $a < a$ no se cumple para elemento alguno de A.
2. Asimétrica: Si $a < b$, entonces $b < a$ no se cumple.
3. Transitiva: Si $a < b$ y $b < c$ entonces $a < c$.

Además, un orden estricto se dice que es *total* si, para todo $a, b \in A$ se cumple que $a < b$ ó $b < a$ ó $a = a$.

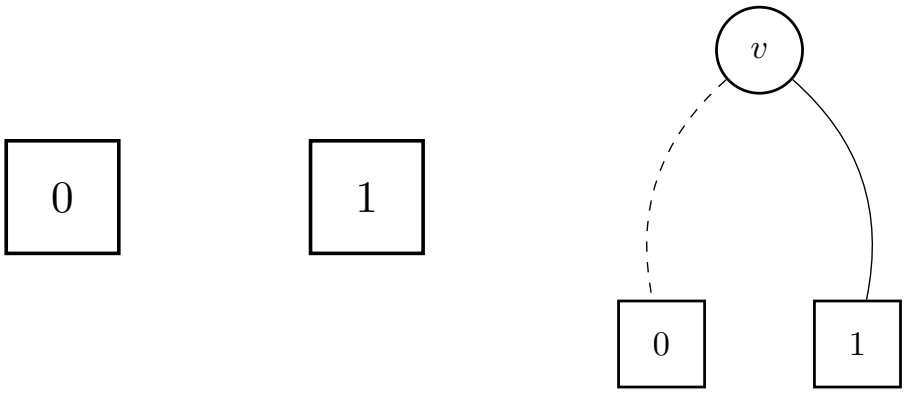

Figura 2.5: De izquierda a derecha, ROBDD de 0, de 1 y de una variable proposicional cualquiera v, respectivamente.

3. Eliminación de no terminales repetidos: Si para dos vértices $v_i, v_j \in V_n$ se cumple que $l_n(v_i) = l_n(v_j)$, $E_0(v_i) = E_0(v_j)$ y $E_1(v_i) = E_1(v_j)$ entonces sus subárboles son equivalentes y se deben de colapsar en uno solo. Esto se hace convirtiendo todas las aristas $E_0(w) = v_j$ a $E_0(w) = v_i$ y $E_1(w) = v_j$ a $E_1(w) = v_i$ y eliminando el nodo v_j de V_n.

Este proceso garantiza la obtención de un ROBDD a partir de un OBDD y se suele llamar *reduce* [4, pág. 402].

Algoritmo de combinación

Aún falta describir un procedimiento mediante el cual, dada una función booleana cualquiera, podamos construir un ROBDD asociado a esta. Esto se puede hacer utilizando la definición inductiva de las fórmulas booleanas: se describen los ROBDD asociados a los casos base (0, 1 y las variables booleanas) y luego se describe un procedimiento inductivo para obtener los ROBDD de expresiones compuestas. La Figura 2.5 muestra los ROBDD para los elementos base en la sintaxis. La dificultad para implementar el paso inductivo se debe principalmente a dos requisitos: la necesidad de mantener el resultado de la operación reducido y la necesidad de mantener un ordenamiento en el diagrama. El primer problema se puede solucionar aplicando el procedimiento de reducción descrito con anterioridad, el segundo requisito necesita la introducción de un algoritmo que asegure que el ordenamiento se preserva.

Descrito por primera vez por Bryant en [22], el procedimiento *apply* permite, dados los nodos iniciales de los ROBDD de las funciones booleanas f_1 y f_2, obtener el ROBDD de la función booleana f_1 **op** f_2, donde **op** puede ser $\bullet$, $+$ ó $\oplus$. Además, mantiene el ordenamiento de las variables presentes en los ROBDD correspondientes f_1 y a f_2 cuando estos tienen ordenamientos compatibles. El algoritmo está basado en una descomposición de la función booleana f_1 **op** f_2 utilizando el Teorema de Expansión de Boole:

$$f_1 \textbf{ op } f_2 \;\equiv\; v_k \bullet (f_1|_{v_k=1} \textbf{ op } f_2|_{v_k=1}) \;+\; \neg\, v_k \bullet (f_1|_{v_k=0} \textbf{ op } f_2|_{v_k=0}) \tag{2.1}$$

Donde v_k es cualquier variable que ocurre en f_1 o en f_2.

El algoritmo procede de manera inductiva comenzando con los vértices iniciales de los ROBDD asociados a f_1 y f_2 hasta llegar a los nodos hoja, utilizando el ordenamiento l que se forma de combinar los ordenamientos compatibles l_1 y l_2 asociados a f_1 y f_2, respectivamente. Dependiendo de la naturaleza de los vértices n_1 y n_2 visitados en el procedimiento $apply(\mathbf{op}, n_1, n_2)$, se presentan distintos casos de comportamiento de la Ecuación 2.1:

- Si n_1 y n_2 son ambos terminales. Ambos representan constantes booleanas por lo que $f_1|_{v_k=1} = f_1|_{v_k=0} = l_t(n_1)$ y $f_2|_{v_k=1} = f_2|_{v_k=0} = l_t(n_2)$. Como ni f_1 ni f_2 dependen de variable alguna, el resultado de la descomposición descrita en la Ecuación 2.1 sería un vértice terminal con etiqueta $l_t(n_1)$ **op** $l_t(n_2)$.

- Si n_1 y n_2 son vértices no terminales con la misma etiqueta v_k (esto es, $l_n(n_1) = l_n(n_2) = v_k$), entonces $f_1|_{v_k=0}$ estará asociado al vértice $E_0(n_1)$, $f_2|_{v_k=0}$ estará asociado al vértice $E_0(n_2)$ y lo mismo para $f_1|_{v_k=1}$ y $f_2|_{v_k=1}$ con los vértices asociados a $E_1(n_1)$ y $E_1(n_2)$. De acuerdo con la descomposición, se debe de crear un nodo n etiquetado con v_k cuyo hijo izquierdo (es decir, $E_0(n)$) sea la llamada recursiva al procedimiento $apply(\mathbf{op}, E_0(n_1), E_0(n_2))$ y su hijo derecho (es decir, $E_1(n)$) sea $apply(\mathbf{op}, E_1(n_1), E_1(n_2))$.

- En caso de que n_1 sea un nodo no terminal con etiqueta v_k y n_2 sea un nodo terminal o un nodo no terminal con etiqueta $v_m >_l v_k$, sabemos que la función f_2 no depende de v_k (debido que es un ROBDD) y por lo tanto $f_2|_{v_k=0} = f_2|_{v_k=1} = f_2$. Debemos crear un nodo n con etiqueta v_k cuyo hijo izquierdo sea la llamada recursiva a $apply(\mathbf{op}, E_0(n_1), n_2)$ y su hijo derecho sea $apply(\mathbf{op}, E_1(n_1), n_2)$. El caso en el que el $v_k >_l v_m$ se trata de manera simétrica utilizando el mismo razonamiento.

Para que el resultado de la operación anterior se encuentre reducido se debe de aplicar el algoritmo de reducción descrito en la sección anterior.

Algoritmo de restricción

Finalmente, para poder codificar cualquier estructura de Kripke utilizando ROBDD debemos de contar con un procedimiento que, dado un ROBDD de una función booleana f cualquiera, y una variable v presente en esta, obtenga el ROBDD de las funciones existe $(\exists v.f)$ y para todo $(\forall v.f)$. Las sumas y las multiplicaciones involucradas en estas funciones se realizan con el proceso *apply* descrito en el parrafo anterior; sin embargo, aún requerimos un procedimiento que realice la operación sustitución. A este procedimiento le llamaremos *restrict(a, v, B)* (donde a es la constante booleana que sustituye a v y B es el ROBDD que representa a f) al igual que lo hizo Bryant [22]. El algoritmo realiza la operación descrita visitando cada vértice no terminal p (con el algoritmo BFS descrito en el Apéndice 1.11). Si $l_n(p) = v$, se redirigirán todas las aristas que lleguen a este nodo a su hijo izquierdo (en caso de que $a = 0$) o a su hijo derecho (en caso de que $a = 1$), si p es el nodo raíz, entonces este se sustituirá por el hijo correspondiente. Formalmente, se convertirá toda arista $E_0(q) = p$ a $E_0(q) = E_a(p)$ y $E_1(q) = p$ a $E_1(q) = E_a(p)$. Finalmente, el resultado se debe minimizarse.

Procedimiento	**Entrada**	**Salida**	**Complejidad Algorítmica**
reduce	OBDD B.	ROBDD equivalente a B.	$O(\|B\| log \|B\|)$
apply	ROBDD B_1 y B_2 de las funciones f_1 y f_2 y operación **op**.	ROBDD de f_1 **op** f_2.	$O(\|B_1\| \|B_2\|)$
restrict	ROBDD de f variable v en f y constante booleana a.	ROBDD de $f\|_{v=a}$.	$O(\|B_1\| \|B_2\|)$

Tabla 2.1: Operaciones sobre los OBDD y sus complejidades algorítmicas.

La complejidad algorítmica de las operaciones anteriormente descritas depende considerablemente de la implementación computacional tanto de los BDD como de los algoritmos sobre estos. Para mayor detalle acerca de los algoritmo descritos, su implementación y su comportamiento se puede consultar el artículo original de Bryant [22]. La Tabla 2.1 muestra la complejidad algorítmica convencional para las operaciones descritas. Para un OBDD B cualquiera, se denotará mediante $|B|$ al número de vértices en él.

2.3.3. Codificación simbólica del modelo

A lo largo de la sección anterior se introdujeron las funciones booleanas y una representación eficiente de estas: los diagramas de decisión binarios. La verificación de modelos simbólica consiste en codificar un diagrama de transiciones arbitrario $\mathcal{M}$ utilizando funciones booleanas (implementadas con ROBDD) y resolver el problema de verificación de modelos utilizando algoritmos sobre funciones booleanas. Como ya se ha mencionado anteriormente, en la práctica esta codificación ha sido sumamente útil y ha permitido incrementar radicalmente el tamaño de los sistemas que se pueden verificar [7]. Para que este objetivo se pueda realizar, evidentemente es necesario traducir los sistemas y algoritmos de verificación descritos hasta ahora a este nuevo dominio. Este es el objetivo de la presente sección.

Codificación de los estados

Dada una estructura de Kripke arbitraria $\mathcal{M} = (S, \longrightarrow, I, AP, L)$, el primer paso será la codificación de los estados $s \in S$ y subconjuntos de S utilizando funciones booleanas. Como se mencionó en la Sección 1.2.2, la función de etiquetado L es una función inyectiva, lo que permite identificar de manera única a un estado por las variables proposicionales que se satisfacen en él. Aprovechando esta característica, podemos identificar a cada estado $s \in S$ con una asignación booleana $cod(s) = (a_1, a_2, \ ... \ , a_n) \in \mathbf{B}^n$ de las n variables proposicionales en AP. La asignación se hará de la siguiente manera: dado un estado $s \in S$ cualquiera y una variable proposicional $p \in AP$ cualquiera, el valor de la constante booleana a_i asociada p será 1 si $s \models p$ y 0 si $s \not\models p$, de esta manera se puede determinar la asignación correspondiente a cada estado dada L.

Utilizando esta convención, representaremos cualquier subconjunto S_0 de S mediante una función booleana.

Definición 2.16 (Función característica de un conjunto de estados)**.** Sea $\mathcal{M} = (S, \longrightarrow, I, AP, L)$ un diagrama de transiciones arbitrario con $|AP| = n$, representaremos a cualquier subconjunto $S_0 \subseteq S$ mediante una función booleana, llamada *función característica de* S_0, definida de la siguiente manera:

$$f_{S_0} : \mathbf{B}^n \to \mathbf{B}$$

tal que:

$$f_{S_0}(cod(s)) = 1 \quad \text{syss} \quad s \in S_0$$

Es decir, la función evaluará con 1 únicamente para la codificación de aquellos estados que pertenecen a S_0 y con 0 en caso contrario.

Codificación de la relación de transición

Definición 2.17 (Función característica de la relación de transición)**.** Sea $\mathcal{M} = (S, \longrightarrow, I, AP, L)$ un diagrama de transiciones arbitrario con $AP = \{a_1, a_2, \ldots, a_n\}$, la función característica de la relación de transición $\longrightarrow$ es una función $f_{\longrightarrow} : \mathbf{B}^{2n} \to \mathbf{B}$. Por cada variable $x_i \in AP$ crearemos una nueva variable x'_i de manera que $f_{\longrightarrow}$ será de la forma $f(x_1, x_2, \ldots, x_n, x'_1, x'_2, \ldots, x'_n)$ y su comportamiento será el siguiente:

$$f_{\longrightarrow}(cod(s), cod(s')) = 1 \quad \text{syss} \quad s \to s'$$

2.3.4. Algoritmos de verificación simbólica

La función *Ant* definida en la Sección 2.1 es esencial para poder implementar los algoritmos de verificación, por lo que debemos de representarla simbólicamente.

Definición 2.18 (Función *Ant* simbólica para un estado)**.** Sea $\mathcal{M} = (S, \longrightarrow, I, AP, L)$ un diagrama de transiciones arbitrario con $|AP| = n$, sea $f_{\longrightarrow}$ la función característica de su relación de transición y sea $cod(s) = (a_1, a_2, \ldots, a_n) \in \mathbf{B}^n$ la codificación de un estado $s \in S$. La función característica del conjunto $Ant(s)$ de estados anteriores a s se define de la siguiente manera:

$$f_{Ant(s)} = f_{\longrightarrow}|_{[x'_1 = a_1, x'_2 = a_2, \ldots, x'_n = a_n]}$$

Esta definición se puede extender para conjuntos de estados de la siguiente manera:

Definición 2.19 (Función *Ant* simbólica para conjuntos de estados)**.** Sea $\mathcal{M} = (S, \longrightarrow, I, AP, L)$ un diagrama de transiciones arbitrario con $|AP| = n$, sea $f_{\longrightarrow}$ la función característica de su relación de transición $\longrightarrow$ y sea f_{S_0} la función característica de $S_0 \subseteq S$. La función característica del conjunto $Ant(S_0)$ se define de la siguiente manera:

$$f_{Ant(S_0)} = \exists[x'_1, x'_2, \ldots, x'_n](f_{\longrightarrow} \bullet f_{S_0}[x_1 = x'_1, x_2 = x'_2, \ldots, x_n = x'_n])$$

La intuición detrás de esta representación es que la fórmula $f_{S_0}[x_1 = x'_1, x_2 = x'_2, \ldots, x_n = x'_n]$ forza a que los estados "siguientes" en la relación de transición estén en S_0, es por ello que se

multiplica. Sin embargo no nos interesa ningún elemento de S_0 en particular sino cualquier posible combinación de valores que produzca un elemento en el conjunto, es por esta razón que utilizamos la función existe con todas las variables "siguientes".

Con esta función definida, podemos ya codificar los equivalentes simbólicos a los algoritmos convencionales de satisfacción [Algoritmo 2.1 - Algoritmo 2.4]. A continuación se supondrá que el modelo de entrada $\mathcal{M}$ ya se encuentra codificado simbólicamente.

Algoritmo 2.6 Conjunto de satisfacción EX simbólico

```
1: def satSetSymbEX(M, f_φ):
2:     f_Y = f_Ant(φ)
3:     return f_Y
```

Algoritmo 2.7 Conjunto de satisfacción EG

```
1: def satSetSymbEG(M, f_φ):
2:     f_X = 0
3:     f_Y = f_φ
4:     while f_X ≠ f_Y :
5:         f_X = f_Y
6:         f_Y = f_Y • f_Ant(Y)      # Todos los estados con una transición hacia Y
7:     return f_Y
```

Algoritmo 2.8 Conjunto de satisfacción EU

```
1: def satSetSymbEU(M, f_φ1, f_φ2):
2:     f_X = 0
3:     f_Y = f_φ2
4:     f_Z = f_φ1
5:     while f_X ≠ f_Y :
6:         f_X = f_Y
7:         f_Y = f_Y + f_Z • f_Ant(Y)      # Todos los estados anteriores donde se cumple φ1
8:     return f_Y
```

2.3.5. Equidad simbólica

Para describir en su totalidad el proceso de verificación simbólica también se deben representar simbólicamente los algoritmos de verificación con equidad. El Teorema 2.2 explicó cómo solamente es necesario describir un algoritmo que obtenga el conjunto de estados que satisfacen la fórmula $\underset{equi}{\mathrm{EG}}\,\phi$, junto con los algoritmos de verificación sin equidad, para poder realizar la verificación de modelos con equidad. Dada una estructura de Kripke arbitraria $\mathcal{M} = (S, \longrightarrow, I, AP, L)$ y un conjunto de restricciones de equidad $equi = \{\phi_1, \phi_2, ..., \phi_n\}$,

existe una caracterización del conjunto de estados S_0 que satisfacen la fórmula $\underset{equi}{\mathrm{EG}}\,\phi$ que sugiere un proceso algorítmico para obtenerlo. El conjunto $S_0 \subseteq S$ es el subconjunto más grande que cumple con las siguientes dos propiedades [7, pág. 157]:

- Todos los estados en S_0 satisfacen a ϕ.
- Para todo estado $s \in S_0$ y para cada una de las restricciones ϕ_i en *equi*, existe un camino que comienza en s de longitud al menos uno que satisface a ϕ_i y todo estado en el camino satisface a ϕ.

Traduciendo estas características a funciones booleanas:

- En un inicio, se consideran todos los elementos en el conjunto $Y_0 = \{s \mid s \models \phi\}$ como posibles candidatos.
- Mientras el resultado de la operación siga cambiando, restringir el conjunto Y a aquellos que cumplan con la caracterización hecha anteriormente:

$$Y_{n+1} = f_\phi \bullet \prod_{i=1}^{n} \mathrm{EX}\ \mathrm{E}[f_\phi\ \mathrm{U}\ (Y_n \bullet f_{\phi_i})]$$

El Algoritmo 2.9 muestra cómo realizar este procedimiento.

Algoritmo 2.9 Conjunto de satisfacción $\underset{equi}{\mathrm{EG}}\phi$ simbólico

```
1: def satSetSymbEGEqui(M, f_φ, f_equi = {f_φ1, f_φ2, ..., f_φn}):
2:     X = 0
3:     Z = f_φ        # Suposición inicial
4:     Y = Z
5:     while X ≠ Y :
6:         X = Y
7:         Y = 1
8:         for f_φi in equi :
9:             U = satSetSymbEU(M, Z, X • f_φi)
10:            Y = Y • satSetSymbEX(M, U)
11:        Y = Z • Y
12:    return Y
```

Finalmente, juntando todo lo que se ha desarrollado hasta ahora, el Algoritmo 2.10 muestra cómo realizar el proceso de verificación simbólica con equidad en su totalidad. Todos los operadores derivados se pueden obtener a partir de los que se muestran en el código de acuerdo con la Definición 1.2 y las equivalencias descritas en el Teorema 1.2. Se omite el pseudocódigo del algoritmo general de satisfacción sin equidad debido a su similitud con el Algoritmo 2.1, la implementación de dicho algoritmo se realizaría de igual manera, pero sustituyendo cada una de las operaciones y llamadas a código por sus equivalentes simbólicas definidas a lo largo de esta sección.

Algoritmo 2.10 Verificación de modelos simbólica con equidad

```
1: def satSetSymbEqui(M, φ, equi = {φ_1, φ_2, ..., φ_n}):
2:     if φ == ⊤ :
3:         return 1
4:     else if φ ∈ AP:
5:         return f_φ
6:     else if φ == ¬ψ:
7:         return ¬ satSetSymbEqui(M, ψ, equi)
8:     else if φ == ψ_1 ∧ ψ_2:
9:         return satSetSymbEqui(M, ψ_1, equi) • satSetSymbEqui(M, ψ_2, equi)
10:    else if φ == EXψ:
11:        F = satSetSymbEGEqui(M, 1, equi)      # Estados justos
12:        X = satSetSymbEqui(M, ψ, equi)
13:        return satSetSymbEX(M, X • F)
14:    else if φ == EGψ:
15:        return satSetSymbEGEqui(M, ψ, equi)
16:    else if φ == E[ψ_1 U ψ_2]:
17:        F = satSetSymbEGEqui(M, 1, equi)      # Estados justos
18:        X = satSetSymbEqui(M, φ_1)
19:        Y = satSetSymbEqui(M, φ_2)
20:        return satSetSymbEU(M, X, Y • F)
```

A lo largo de este capítulo se describieron los algoritmos que permiten realizar la verificación de modelos. Inicialmente, esta descripción se hizo sobre la representación directa del diagrama de transiciones basada en grafos y, posteriormente, sobre la codificación simbólica mediante funciones booleanas implementadas sobre diagramas de decisión binarios. Con esta nueva representación y los algoritmos descritos sobre ella, contamos ya con todos los elementos teóricos para poder construir el verificador. El siguiente capítulo se encargará de explicar el sistema de software concreto que incorpora todos estos elementos teóricos.

Capítulo 3

Descripción del sistema

A lo largo de este breve capítulo se explicará de manera concisa la implementación ingenieril del sistema verificador de modelos. Inicialmente se hará un resumen de las tecnologías utilizadas para implementar el verificador y posteriormente se describirá cómo se realiza el procesamiento de la información, desde el código de entrada hasta el criterio de verificación de modelos.

3.1. Herramientas utilizadas

3.1.1. Haskell

El principal diferenciador del verificador de modelos implementado en este proyecto de tesis con respecto a otras alternativas comerciales es que se trata de un verificador *funcional*. Prácticamente toda la implementación, desde el procesamiento del código fuente hasta la prueba del funcionamiento de los algoritmos de verificación se hizo utilizando el paradigma funcional.

La escritura de un programa utilizando un lenguaje funcional no implica solamente un cambio de sintaxis en el código con respecto a otras alternativas no funcionales, sino que involucra una reestructuración completa en la forma que se realizan todos los algoritmos. Elementos de programación utilizados en los lenguajes estructurados de manera rutinaria como los ciclos *for* no existen en los lenguajes funcionales rigurosos. El origen de las diferencias se debe a que, en última instancia, los lenguajes de programación estructurados y los lenguajes funcionales están basados en diferentes *paradigmas de la programación*.

Tomemos el ejemplo del lenguaje *C*. *C* surgió de la necesidad de incrementar la capacidad expresiva de los lenguajes ensambladores para la realización del sistema operativo UNIX (que hasta entonces había sido programado en lenguaje ensamblador). *C* posee además la capacidad de referir directamente localidades de memoria mediante apuntadores, reservar memoria del sistema operativo de manera dinámica, así como la capacidad para manipular directamente los bits en alguna localidad de memoria especifica [23]. Todas estas características sugieren un enfoque de programación orientado al hardware, donde el lenguaje de programación se encarga de manipular los elementos de la máquina en la que está implementado. La programación funcional proviene de una tradición distinta. Desde su surgimiento con el lenguaje LISP en

los años 50, tanto la sintaxis como el funcionamiento de los lenguajes funcionales estuvieron profundamente influenciados por un modelo de la computación inspirado en las matemáticas teóricas y alejado de las consideraciones sobre el dispositivo físico en el que sería ejecutado: el cálculo-λ (pronunciado como "cálculo lambda").

Desarrollado durante los años 30 por Alonzo Church (1903 - 1995), lógico de la Universidad de Princeton, el cálculo-λ es un sistema formal para expresar computación basado en funciones; el cálculo-λ hace explícitas ciertas nociones del comportamiento de las funciones en las matemáticas como la dependencia del valor de una función f en el valor de la variable x (conocida como abstracción), la sustitución de un valor determinado v por una variable x en una función f (aplicación), así como el proceso de evaluación de la función para dicho argumento (reducción-β). Church utilizaría su modelo para demostrar resultados fundamentales sobre los límites de lo que es computable [24]. El cálculo-λ, además, resultaría ser equivalente a otros modelos teóricos de la computación como las máquinas de Turing [25] y, en última instancia, con la noción "convencional" de computación[1]. A pesar de sus notables diferencias con la teoría matemática, los lenguajes de programación funcional modernos, incluido Haskell, comparten con el cálculo-λ tanto el poder de expresar cualquier computación posible como el enfoque central de la función matemática como elemento computacional.

Volviendo al tema que concierne este capítulo, Haskell, nombrado así en honor al matemático estadounidense Haskell Brooks Curry (1900-1982), es un lenguaje de programación funcional, *estáticamente tipado*, *puro*, *perezoso* y con *clases de tipos*, entre otras características [8]. Haskell surgió de la necesidad de muchos investigadores a medidados de los años 80, trabajando todos de manera independiente en sus propios lenguajes funcionales, de contar con un lenguaje de programación puro y perezoso del que todos se pudieran beneficiar. Entre las motivaciones iniciales para el diseño del lenguaje se encontraban las siguientes:

- El lenguaje de programación debería ser útil para la enseñanza, investigación y aplicación comercial.
- Debería ser disponible sin costo.
- Debería implementar ideas probadas y que fueran ampliamente aceptadas por la comunidad.
- El lenguaje debería reducir la innecesaria diversidad de alternativas de lenguajes de programación funcionales que existía en aquel momento.

Los años 90 fueron un periodo de arduo trabajo para el grupo encargado de diseñar y desarrollar Haskell. Se crearon diversas versiones y especificaciones comenzando con el reporte de Haskell 1.0 en 1990 y culminando con el estándar conocido como Haskell 98. La siguiente revisión mayor del lenguaje no vendría sino hasta el reporte de la versión Haskell 2010.

En lo que respecta al lenguaje en sí, algunos elementos importantes lo caracterizan:

[1] A esta caracterización se le conoce como la tesis de Church-Turing y hace referencia a la hipótesis (no demostrable matemáticamente) que las nociones intuitivas de un proceso "mecánico", "algorítmico", "computable", etc. son precisamente las que formalizan los modelos matemáticos como el cálculo-λ y las máquinas de Turing.

- Estáticamente tipado: Se dice que un lenguaje es estáticamente tipado si el valor de una variable u operación se puede saber en tiempo de compilación. Generalmente, esta característica exige la definición de un tipo cada vez que se crea una variable. En Haskell, el valor de toda variable y función debe de tener un tipo específico asociado.

- Pureza: Una de las características que hace la experiencia de programar en Haskell tan distinta a otros lenguajes es la pureza de las funciones al momento de programar. En un lenguaje funcional puro, la llamada a una función con un argumento determinado siempre da el mismo resultado. En *C*, por el contrario, dos llamadas a la misma función con el mismo argumento podrían dar valores distintos si la función depende de otro valor que ha cambiado de una llamada a otra. Esta característica lleva a que en Haskell no exista un "estado" del programa que pueda cambiar el resultado de una llamada a la función respecto a otra, el valor de una función solamente depende de su definición y del valor de sus argumentos.

- Pereza: En un lenguaje perezoso, el valor de una expresión no se calcula cuando se le asigna una variable, más bien, se espera hasta que el resultado de la expresión se requira explícitamente (por ejemplo, para imprimir en pantalla el resultado) [26]. Una de las características más importantes que surgen de este tipo de implementación es la capacidad de definir estructuras potencialmente infinitas en el código de Haskell, como dichas estructuras no se utilizan inmediatamente sino hasta que se requiera su valor, no existe problema en definirlas.

- Clases de tipos: En Haskell, las clases de tipos permiten definir un conjunto de funciones que un tipo debe de implementar si desea formar parte de la clase (convertirse en una *instancia* de la clase). Por ejemplo, la clase `Eq`, que implementan tipos que se quieren comparar para la igualdad de alguna manera, define las funciones de *igualdad* (==) y de *desigualdad* (/=) y algunas de sus instancias son los tipos `Integer`, `Bool` y `Float`. Las clases de tipos en Haskell son una forma de lograr polimorfismo funcional.

Haskell ha cumplido con los objetivos que motivaron su creación con creces. Hoy en día Haskell es un lenguaje internacionalmente famoso, su compilador más popular, el Glasgow Haskell Compiler (GHC) es ampliamente utilizado tanto en la academia [27] como en la industria [28]. Empresas como Google, AT&T, Intel y Bank of America hacen uso de él y frecuentemente se publican libros y desarrollan herramientas para expandir su funcionalidad.

3.1.2. Foreign Function Interface

Al inicio del diseño del verificador de modelos que se presenta en esta tesis, se buscó realizar una implementación puramente funcional de los BDD y de sus operaciones. La referencia más completa que se encontró sobre el tema fue la tesis de licenciatura [29] y la posterior publicación en un artículo científico [30] del trabajo de Jan Christiansen de la Universidad de Kiel en Alemania.

De acuerdo con el resumen del artículo científico, el objetivo del trabajo es aprovechar la evaluación perezosa para mejorar el desempeño de algunos de los algoritmos sobre los ROBDD que se han mostrado previamente en este trabajo. En el artículo se explican dos

formas distintas de implementar la estructura de datos. La primera de ellas hace uso de mapas basados en árboles para identificar individualmente a los nodos, pero debido a que no aprovecha al máximo la pereza del lenguaje, en la segunda implementación se decide relajar la propiedad de no redundancia en los nodos y por lo tanto permitir la existencia separada de nodos equivalentes en el diagrama. Esta modificación, a pesar de eliminar propiedades útiles de los BDD como la canonicidad y alterar la implementación de ciertos algoritmos, trae ventajas en términos de desempeño, brindadas por la pereza.

A pesar de contener ideas interesantes sobre un enfoque funcional para la implementación de los BDD, en última instancia se decidió no utilizar alguna de las implementaciones mostradas en el artículo por diversas razones. Para la primera propuesta se consideraron los siguientes puntos:

- Los nodos equivalentes en la estructura no se presentan de manera única. Estos siguen estando presentes en una localidad distinta de memoria pero con un identificador especial que señala su redundancia. Esto implica un gasto innecesario de espacio.

- Los marcadores de redundancia complican considerablemente la implementación de las operaciones sobre los BDD.

- Debido a que se trata de una implementación puramente funcional, el mapa en el que se busca la información de los nodos está implementado utilizando árboles. La consulta en este tipo de mapas es asintóticamente más lenta ($O(log\ n)$ donde n es el número de nodos en el mapa) que su equivalente no funcional ($O(1)$ si se utiliza una función hash).

Para la segunda implementación, en la que se aprovecha la pereza y se evita la visita de todos los nodos en las operaciones de los BDD, la característica principal que desincentivó su uso fue la pérdida de la canonicidad en los ROBDD. La canonicidad permite verificar la equivalencia de dos funciones booleanas en tiempo constante, por lo que al perderla se empeora el desempeño en esta operación.

Finalmente se decidió utilizar un paquete de BDD reconocido y eficiente para realizar la operaciones: CacBDD [31] (el cual se explicará en la siguiente sección). Como CacBDD no está escrito en Haskell, fue necesario utilizar una interfaz entre el código del verificador en Haskell y el paquete escrito en C++, ahí es donde entra la *FFI* (*Foreign Function Interface*, *interfaz de funciones externas*) de Haskell. La FFI permite crear definiciones funcionales dentro de Haskell que ejecuten código escrito en algún otro lenguaje de programación y regresen un determinado valor que posteriormente pueda ser utilizado. La declaración debe incluir la especificación de la conversión entre los tipos de datos tanto de las entradas como de las salidas del lenguaje origen del código que se desea utilizar y Haskell. La documentación oficial de Haskell incluye información sobre la realización de este proceso [32] y existen un sin número de paquetes en la base oficial de Haskell que hacen uso de la interfaz.

3.1.3. CacBDD

Los paquetes modernos de BDD generalmente utilizan dos tablas basadas en funciones hash para agilizar los algoritmos de reducción y las operaciones booleanas sobre los ROBDD

[4, págs. 409–421] [31], [33]. La primera, llamada generalmente *tabla única*, es una tabla redimensionable que identifica a cada nodo a partir de la variable presente en él y sus nodos hijos, y permite la eliminación de nodos no terminales repetidos. La segunda, llamada *tabla computada* (también llamada tabla caché [33]), es una tabla de tamaño fijo que guarda resultados parciales de las operaciones sobre nodos de los BDD para evitar recalcularlos si es que se utilizan más adelante. Como tiene un tamaño fijo, cuando ocurre una colisión en la función hash, el valor presente previamente se reemplaza por el nuevo valor (de manera similar a como ocurre en las memorias físicas del mismo nombre presentes en la computadora). Es un problema de investigación abierto cómo gestionar correctamente la tabla computada para lograr un desempeño óptimo en los algoritmos sobre los ROBDD [31].

CacBDD es un paquete relativamente reciente (publicado en 2013) de operaciones sobre ROBDD que utiliza un algoritmo dinámico de gestión de la tabla computada con la intención de mejorar el desempeño sobre otras alternativas disponibles. El paquete calcula constantemente el tamaño de la tabla computada, así como la proporción de todas las llamadas que son atendidas por la tabla computada (es decir, que no tienen que ser recalculadas) y altera dinámicamente el tamaño de esta con la intención de aprovechar la memoria de la mejor manera posible. El artículo reporta un desempeño ligeramente mejor que el del famoso paquete CUDD (*Colorado University Decision Diagrams, diagramas de decisión de la Universidad de Colorado*) tanto en uso de memoria como en tiempo de ejecución, para la mayoría de los casos probados.

3.1.4. HasCacBDD

La interfaz utilizada entre el código en *C++* y Haskell es *HasCacBDD*. HasCacBDD fue creada por Malvin Gattinger del ILCC (*Institute for Logic, Language and Computation*, *Instituto para la Lógica, el Lenguaje y la Computación*) de la Universidad de Amsterdam. Diseñada inicialmente como parte de un proyecto para implementar un verificador simbólico para Lógica Epistémica Dinámica [34], esta interfaz funciona impecablemente, a diferencia de las demás opciones encontradas en otros repositorios [35], [36]. La interfaz implementa todas las operaciones comunes que se pueden necesitar para hacer la verificación simbólica y cuenta con una suite de pruebas robusta que verifica su funcionamiento correcto.

3.1.5. Parsec

El análisis sintáctico de oraciones, dada una gramática formal, es uno de los grandes éxitos de la programación funcional en aplicaciones reales [37]. Existen varias clases de tipos que se adecuan particularmente bien a este proceso (principalmente las clases *Functor*, *Monad* y *Applicative* [38]). La investigación original sobre el tema dio lugar a diversos paquetes para el análisis sintáctico, siendo *Parsec* [39] uno de los más exitosos. Parsec es una biblioteca de análisis sintáctico basadas en mónadas, eficiente en términos de memoria y espacio, y con la capacidad para documentar errores sintácticos. Utilizando Parsec, se pueden especificar todos los elementos de la gramática: la definición de los componentes léxicos, la especificación de expresiones con operandos con distinta precedencia y asociatividad, así como la definición de la gramática misma y mensajes para reportar errores. Se utilizaron todas estas funciones para

la definición del lenguaje de entrada.

3.1.6. Hspec

La herramienta que se utiliza para probar el correcto funcionamiento de los programas es *Hspec* [40]. Hspec es un marco de pruebas puramente funcional desarrollado en Haskell. Hspec tiene una sencilla interfaz para explicar los casos de pruebas y los resultados esperados para cada uno de ellos y posee además interoperabilidad con otras herramientas de prueba en Haskell como *HUnit* y *QuickCheck*. La motivación original para utilizar este paquete fue su simplicidad de uso vista en el paquete HasCacBDD.

3.1.7. Stack

Finalmente, y también motivado por la recomendación personal por parte de Malvin Gattinger (creador de HasCacBDD), la herramienta utilizada para importar las bibliotecas y crear el proyecto fue Stack. De acuerdo con su documentación [41], *Stack* tiene un fuerte enfoque en la creación de proyectos reproducibles, multi-paquete, con un alto grado de personalización y fáciles de usar. Stack incluye el compilador de Haskell GHC, Cabal y repositorios con paquetes para la creación de software aislado. Stack fue creado con la intención de extender a Cabal y corregir algunas de sus limitaciones más notorias [42], principalmente los problemas ocasionados por solicitudes de paquetes incompatibles en diferentes proyectos. Stack crea un entorno aislado para cada paquete que se define, con una instalación propia de GHC así como de todas las dependencias necesarias en cada proyecto. Este aislamiento ayuda a minimizar los problemas de compatibilidad. La experiencia personal en el desarrollo de este proyecto confirma estas ventajas; la instalación de HasCacBDD, que fue desarrollado en Stack, fue infinitamente más sencilla y rápida que la de los demás paquetes de BDD que utilizaban solamente Cabal. Ya en el desarrollo del proyecto, la utilización de las dependencias, la creación de los casos de pruebas y la creación de los diferentes módulos se dieron sin mayor dificultad, salvo ciertas excepciones relacionadas con la utilización de código en *C++* incluido en HasCacBDD.

3.2. Arquitectura del programa

Volvemos nuestra atención al estudio del sistema construido. En esta sección se explicará la estructura general del programa y los diferentes módulos que lo conforman. La estructura asemeja a la de otros verificadores de modelos. La entrada al programa es un archivo con la especificación de la fórmula CTL y el modelo a verificar utilizando una sintaxis formal, posteriormente se lee tal archivo y si la especificación es correcta, sintáctica y semánticamente, entonces se obtiene una representación interna de los elementos especificados. Esta representación interna debe de convertirse a sus correspondientes BDD para la verificación simbólica y, una vez que se tienen las estructuras como BDD, se deben de implementar los algoritmos de verificación y emitir el juicio de satisfacción final al usuario. En esta sección se describirá formalmente la sintaxis que debe de tener la entrada y posteriormente se explicará de manera general qué realiza cada módulo del programa.

3.2.1. Lenguaje de entrada

La entrada al programa es un archivo de texto escrito de acuerdo con una gramática formal. La sintaxis escogida asemeja a la del famoso verificador de modelos NuSMV, con algunas simplificaciones hechas [20]. A continuación se describen los elementos con los que debe de contar un programa y se especifica formalmente su sintaxis utilizando la notación *EBNF* (*Extended Backus-Naur Form*, notación Backus-Naur extendida)[2].

Variables de estado

De esta manera se nombran a los elementos en el conjunto AP. Cada estado en el diagrama de transiciones será identificado de manera única por un subconjunto de AP, por lo que todas las variables que se usen en la definición del diagrama o en la especificación de la fórmula CTL deberán haber sido declaradas con anterioridad. La definición formal de una declaración de una o más variables es la siguiente:

```
vardec = "VAR" varlist
varlist = variable ";" {variable ";"}
variable = lower {lower | digit | "_"}
lower = "a" | "b" | ... | "z"
digit = "1" | "2" | ... | "9"
```

Variables de entrada

Muchas veces es útil especificar variables booleanas de entrada al programa que pueden modificar su comportamiento en algún momento (como se verá más adelante en los ejemplos mostrados en las Secciones 4.2 y 4.3). Su valor no se sabe con anticipación por lo que puede ser 0 ó 1 en cualquier instante, además, no forman parte de la definición de los estados. Su sintaxis es:

```
ivardec = "IVAR" varlist
```

Evidentemente, no se puede utilizar el mismo identificador que el de una variable que ya haya sido definida anteriormente como de estado.

Variables compuestas

Las variables compuestas permiten asociar una etiqueta a una expresión booleana. Esta etiqueta puede utilizarse como abreviatura posteriormente en el programa para evitar el manejo de expresiones muy extensas.

[2]Para esta tesis, los elementos de la notación que se utilizarán son los siguientes: las palabras entre comillas denotarán símbolos terminales (p.e. `"VAR"` y `"a"`); los símbolos no terminales estarán ligeramente remarcados y sin comillas (p.e. `vardec`); las llaves indicarán elementos que pueden repetirse 0 o más veces (p.e. `{variable ";"}`); y las barras verticales indicarán la separación de opciones alternativas (p.e. `"a" | "b" | "c"`).

```
definedec = "DEFINE" defexplist
defexplist = variable ":=" simple_exp ";"
           {variable ":=" simple_exp ";"}
simple_exp =   constant
             | variable
             | "(" simple_exp ")"
             | "!" simple_exp
             | simple_exp  "&"  simple_exp
             | simple_exp  "|"  simple_exp
             | simple_exp "xor" simple_exp
             | simple_exp  "->" simple_exp
             | simple_exp "<->" simple_exp

constant = "TRUE" | "FALSE"
```

Conjunto de estados iniciales

Como ya se ha visto en la sección dedicada a la verificación de modelos simbólica, un conjunto de estados en el diagrama de transiciones puede ser especificado mediante su función característica. Así especificamos al conjunto de estados iniciales I utilizando la siguiente sintaxis:

```
initdec = "INIT" simple_exp {";"}
```

Relación de transición

Para expresar la relación de transición debemos de poder especificar las variables "actuales" (que corresponden al estado del que sale una transición) y las variables "siguientes" (que corresponden al estado al que llega la transición). Eso se hace a través de la introducción del no terminal **nextvariable** que se muestra a continuación:

```
transdec = "TRANS" nextexp {";"}
nextexp =   constant
          | nextvariable
          | variable
          | "(" nextexp ")"
          | "!" nextexp
          | nextexp  "&"  nextexp
          | nextexp  "|"  nextexp
          | nextexp "xor" nextexp
          | nextexp  "->" nextexp
          | nextexp "<->" nextexp
nextvariable = "next" "(" variable ")"
```

Especificación CTL

La especificación de la fórmula CTL que se quiere verificar en el programa es una traducción directa de la sintaxis vista en la Definición 1.1. Formalmente:

```
ctlspec = "CTLSPEC" ctlformula {";"}
ctlformula =  constant
            | variable
            | "(" ctlformula ")"
            | "!" ctlformula
            | "EX" ctlformula
            | "EF" ctlformula
            | "EG" ctlformula
            | "AX" ctlformula
            | "AF" ctlformula
            | "AG" ctlformula
            | ctlformula  "&"  ctlformula
            | ctlformula  "|"  ctlformula
            | ctlformula "xor" ctlformula
            | ctlformula  "->" ctlformula
            | ctlformula "<->" ctlformula
            | ctlformula "EU"  ctlformula
            | ctlformula "AU"  ctlformula
```

Propiedades de equidad

Finalmente, describimos cómo se especifican propiedades de equidad en el verificador. Al igual que como lo hace NuSMV [20], restringimos las propiedades de equidad a cumplirse únicamente a expresiones booleanas sin operadores temporales, esto es, a la restricción de aquellos caminos donde una fórmula booleana arbitraria ϕ se cumple de manera infinitamente frecuente. En general esta característica es suficiente para definir las propiedades de equidad de interés. La sintaxis de la especificación se muestra a continuación:

```
faircons = "FAIRNESS" simple_exp {";"}
```

El programa está subdivido en 5 módulos distintos que se muestran en la Figura 3.1, donde una flecha del módulo A al módulo B indica que B depende de A. A continuación se describirá cada una de las fases del verificador y se hará referencia a los módulos de acuerdo con su relevancia para cada una de estas.

3.2.2. Definición del lenguaje y análisis sintáctico

El analizador sintáctico se encarga de leer un archivo escrito de acuerdo con la especificación realizada en la subsección anterior y generar una representación interna que pueda ser posteriormente manipulada. Parsec hace este trabajo sumamente simple [39], la definición del lenguaje genera de manera prácticamente automática el analizador sintáctico, solamente es

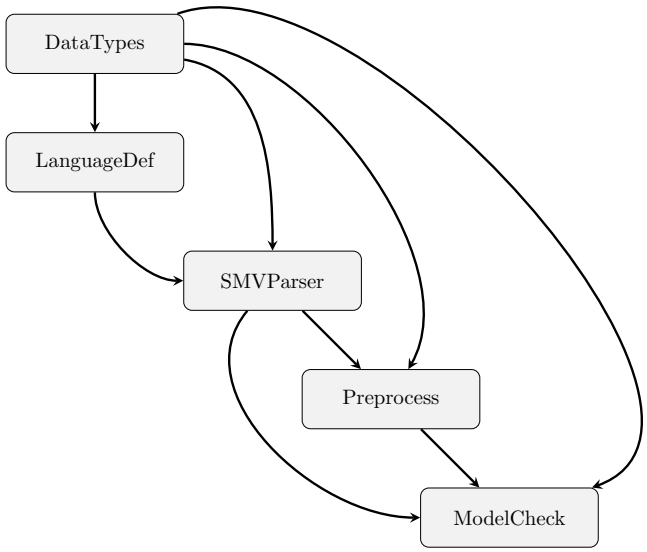

Figura 3.1: Interdependencia de los módulos en el sistema.

necesario especificar las palabras reservadas, la estructura gramatical de los identificadores, la definición de los comentarios y otros elementos y Parsec genera automáticamente los analizadores sintácticos. Parsec además permite definir operadores unarios y binarios y definir su precedencia y asociatividad de manera muy simple. El módulo **LanguageDef** es donde se realiza esta especificación. El módulo **SMVParser**, por su parte, utiliza la definición del lenguaje para generar los tipos correspondientes dentro de Haskell.

```
22  languageDef =
23      emptyDef{
24          Token.commentLine    = "--",
25          Token.identStart     = lower,
26          Token.identLetter    = lower <|> digit,
27          Token.reservedNames  = [
28                                       "next",
29                                       "MODULE",
30
...     ...
200
201 sOperators = [
202                 [Prefix(
203                     reservedOp "!"   >> return (SUnary  Not)
204                  )],
205     ...
```

Código 3.1: Definición de palabras clave y definición de operandos en expresiones CTL en el módulo **LanguageDef**.

```
82 varDecParser :: Parser VarDec
83    varDecParser = do
84                     reserved "VAR" -- uno o mas
85                     list <- endBy1 variableParser semi  -- punto y coma
86                     return $ VarDec list -- varDec es tipo en Haskell
```

Código 3.2: Definición del analizador sintáctico para la declaración de variables en el módulo **SMVParser**.

3.2.3. Representación interna

Para poder convertir el archivo de entrada a objetos que puedan ser manipulados dentro de Haskell se deben de definir tipos que representen cada uno de los elementos. Se deben de crear tipos para expresiones en lógica CTL, para la definición de variables, identificadores y los otros elementos sintácticos definidos anteriormente. Todas estas definiciones se hacen en el módulo **DataTypes**, del cual se muestra un fragmento en el Código 3.3.

```
59 data CTLF  =   CConst BConstant
60                |  CVariable Variable
61                |  CBUnary  BUnOp    CTLF        -- Operadores booleanos unarios (Not)
62                |  CCUnary  CUnOp    CTLF        -- Operadores CTL unarios (AX, ...)
63                |  CBBinary BBinOp   CTLF  CTLF  -- Operadores booleanos binarios
64                |  CCBinary CBinOp   CTLF  CTLF  -- Operadores CTL binarios (AU, EU)
65                deriving(Show)
66
67
68
69 newtype VarDec    = VarDec    [Variable]     deriving (Show)
70 newtype IVarDec   = IVarDec   [Variable]     deriving (Show)
71 newtype InitCons  = InitCons   BSimple       deriving (Show)
72 newtype DefineDec = DefineDec [DefineExp]    deriving (Show)
73 newtype TransCons = TransCons  BNext         deriving (Show)
74 newtype CTLSpec   = CTLSpec   CTLF           deriving (Show)
```

Código 3.3: Módulo **DataTypes**, se muestra la definición de tipos que codifican a las fórmulas CTL y a otras declaraciones necesarias para la verificación.

3.2.4. Transformación y análisis semántico

Una vez que se cuenta con una representación interna de los elementos descritos en el archivo de entrada, se debe revisar que se cumplan con todos los criterios necesarios para poder hacer la verificación de modelos. La revisión sintáctica ya fue hecha por el analizador sintáctico, sin embargo, aún se deben de verificar aspectos semánticos antes de proceder a la verificación. Entre las comprobaciones que se deben de hacer se encuentran la verificación de que todas las variables usadas hayan sido declaradas, que se encuentren todos los elementos necesarios para la verificación, que se sustituyan las variables compuestas donde se utilicen,

que se unan las variables que fueron declaradas en distintos lugares del código y demás operaciones. El módulo **Preprocess** se encarga de hacer estas verificaciones.

```
233 -- Verificamos que ninguna variable declarada en IVAR fue declarada en VAR
234    checkInputDec :: Set.Set Variable -> Set.Set Variable -> Bool
235    checkInputDec vdc ivdc = Set.null (vdc `Set.intersection` ivdc)
236
237    checkInitVars :: InitCons  -> Set.Set Variable -> Bool
238    checkInitVars (InitCons bsimple) = checkSimpleVars bsimple
239
240    checkInitIVars :: InitCons -> Set.Set Variable -> Bool
241    checkInitIVars (InitCons bsimple) = checkNotSimpleVars bsimple
242
243    checkTransVars :: TransCons -> Set.Set Variable -> Set.Set Variable -> Bool
244    checkTransVars (TransCons bnext) = checkNextVars bnext
```

Código 3.4: Módulo **Preprocess**, en el fragmento de código mostrado se verifica que las variables utilizadas se hayan declarado adecuadamente.

3.2.5. Verificación de modelos

Finalmente, y ya que se ha verificado que la entrada fue correcta tanto sintáctica como semánticamente, se debe de realizar la verificación de modelos, esto es, implementar funcionalmente los algoritmos de verificación simbólica utilizando Haskell. El módulo **ModelCheck** se encarga de esto. Como ya se ha mencionado en la subsección anterior, para la implementación de los BDD y sus operaciones se utilizó el paquete CacBDD y la interfaz para Haskell HasCacBDD, es por ello que una parte considerable de las operaciones que se realizan en este módulo consisten en la utilización de funciones descritas en esta interfaz para implementar los algoritmos de verificación simbólicos [Algoritmo 2.6 - Algoritmo 2.10]. Esto incluye la consideración de propiedades de equidad en caso de que estas hayan sido especificadas.

```
259 satEU :: Bdd -> Bdd -> Bdd -> Bdd
260 satEU fc fj trans =  let
261                            postj = satEX fj trans
262                            term  = fc `con` postj
263                            fjp1  = fj `dis` term
264                         in
265                            if(fj `equ` fjp1) == top
266                               then fjp1
267                               else satEU fc fjp1 trans
268
269
...     ...
314 satEGFairA :: Bdd -> Bdd -> Bdd -> [Bdd] -> Bdd
315 satEGFairA subf fixed trans xs =  let
316                                        bigconj = satEGFairCon subf trans fixed xs
317                                        newfixed = subf `con` bigconj
318                                     in
319                                        if (newfixed `equ` fixed) == top
320                                           then newfixed
321                                           else satEGFairA subf newfixed trans xs
```

Código 3.5: Fragmentos de funciones que implementan los algoritmos de verificación para los operadores EU y $\underset{equi}{\mathrm{EG}}\,\phi$ en el módulo **ModelCheck**.

Con la descripción de las herramientas de software utilizadas para construir el verificador, la sintaxis del lenguaje de entrada y los diferentes módulos de software que realizan la verificación, se cuenta ya con una noción suficientemente clara del funcionamiento del programa como para describir algunos ejemplos prácticos. El siguiente capítulo se encargará de poner en marcha el sistema creado verificando varios dispositivos sencillos.

Capítulo 4

Evaluación

A lo largo de este capítulo se mostrarán varios ejemplos de sistemas temporales con la intención de poner a prueba el verificador diseñado, así como para evaluar su desempeño en tiempo y memoria. Este capítulo, además, servirá para entender de manera más clara cómo es que la verificación de modelos ayuda a probar el correcto funcionamiento de los sistemas estudiados y cómo este propósito se relaciona con los conceptos matemáticos abstractos vistos durante los capítulos anteriores. Cada uno de los ejemplos que se mostrarán tiene la intención de estudiar diferentes aspectos de los sistemas temporales, por lo que tanto el dispositivo como las propiedades a verificar son de distinta naturaleza en cada uno de ellos.

Antes de mostrar cada uno de los ejemplos, es útil describir las categorías generales de propiedades que se buscan verificar en los sistemas que se estudiarán.

- Ausencia de *puntos muertos*: la mayoría de los sistemas que se estudian se espera que trabajen por una cantidad de tiempo indefinida. En sistemas de esta naturaleza, aquellos estados que no tienen estado siguiente indican situaciones de error y deben de evitarse.

- *Seguridad*: este tipo de propiedades buscan evitar situaciones que, de acuerdo con la especificación del sistema, se consideran no deseables o de error, o verificar que ciertas propiedades deseables del sistema se cumplen.

- *Equidad*: las propiedades de equidad, como ya se ha mencionado anteriormente en este trabajo, buscan evitar situaciones que se consideran "injustas" dentro del sistema y que se solucionan asegurando que eventualmente se satisfaga alguna condición en todos los caminos. En el sistema creado, las propiedades de equidad se especifican utilizando la palabra reservada `"FAIRNESS"` seguida de la fórmula booleana que se quiere que se cumpla indefinidamente en el sistema.

- *Vitalidad*: las propiedades de vitalidad están estrechamente relacionadas a las de equidad, y hacen referencia a que siempre que un sistema solicite acceder a una situación particular, se le permita hacerlo. Como se verá más adelante, ciertas propiedades de vitalidad se pueden asegurar añadiendo propiedades de equidad al sistema.

4.1. Contador

4.1.1. Descripción

El primer sistema que mostraremos es un ejemplo sumamente sencillo: un contador de n bits. Se trata de un dispositivo digital que cuenta desde el número 0 hasta el número $2^n - 1$, utilizando la representación binaria del mismo. En cada ciclo de reloj, el número en el contador aumenta en uno, esto se hace prendiendo y apagando los bits correspondientes en el dispositivo. En este ejemplo se considerará que $n = 3$.

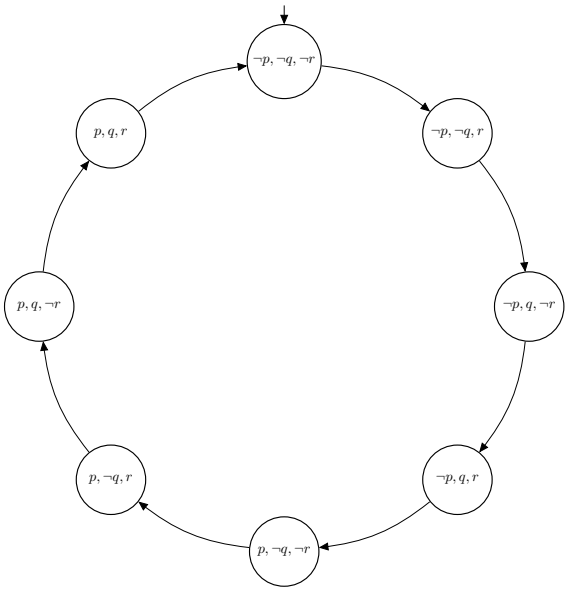

Figura 4.1: Representación gráfica del diagrama de transiciones de un contador de 3 bits.

4.1.2. Codificación

Para poder verificar el modelo es necesario describir su comportamiento en términos de las variables en el conjunto AP para después alimentarlo al sistema. Una posible caracterización de un contador de n bits se puede hacer si se considera que, para cada bit en el contador, su valor numérico cambiará (de 0 a 1 ó de 1 a 0) en el siguiente ciclo de reloj, únicamente cuando todos los bits menos significativos que dicho bit se encuentren prendidos. El bit menos significativo siempre cambia en cada ciclo de reloj. Este comportamiento se puede modelar utilizando sumas exclusivas y conjunciones, como se describe en el Código 4.1.

```
1  VAR                -- Declaracion de variables a utilizar
2     p;              -- Bit mas significativo
3     q;
4     r;              -- Bit menos significativo
5  INIT
6     !p & !q & !r;   -- Estado inicial con todos los bits apagados
7  TRANS              -- Estructura de Kripke
8     (next(r) xor r) & (next(q) <-> q xor r) & (next(p) <-> p xor q & r);
```

Código 4.1: Especificación de la estructura de Kripke para un contador de 3 bits.

4.1.3. Propiedades a verificar

A pesar de ser un dispositivo sencillo, se puede utilizar la lógica CTL para demostrar algunas propiedades que un contador bien diseñado debería de tener. Algunas opciones de propiedades, junto con su categoría, se muestran a continuación:

1. Todo estado tiene un camino que eventualmente llega al estado con el número $2^n - 1$ (seguridad).
2. El siguiente estado del estado con el número $2^n - 1$ es el estado con el número 0 (seguridad).
3. Para todo estado, existe un estado siguiente que es diferente a él (ausencia de puntos muertos).
4. Todo estado solamente tiene un camino que emana de él (seguridad).

Cada una de estas cuatro propiedades se muestran especificadas en el Código 4.2, en el mismo orden que en la lista.

```
9  CTLSPEC
10    AG(AF(p & q & r));
11 CTLSPEC
12    AG(p & q & r -> AX(!p & !q & !r));
13 CTLSPEC
14    AG(p -> EX(!p) | q -> EX(!q) | r -> EX(!r));
15 CTLSPEC
16    AG((EX(p) <-> AX(p)) & (EX(q) <-> AX(q)) & (EX(r) <-> AX(r)));
```

Código 4.2: Especificación de las propiedades a verificar para el contador de 3 bits.

4.1.4. Resultado

El programa debería de cumplir con la especificación y efectivamente lo hace como lo muestra la salida del código. Esta es la salida de la prueba realizada con la suite Hspec de Haskell.

```
1 Especificacion #1: satisface
2 Especificacion #2: satisface
3 Especificacion #3: satisface
4 Especificacion #4: satisface
5   Counter3
```

Código 4.3: Salida de prueba para contador de 3 bits.

4.2. Registro de corrimiento

4.2.1. Descripción

El segundo ejemplo que se mostrará trata con un sistema considerablemente más complicado al contador: un *registro de corrimiento universal*. Los registros de corrimiento son memorias electrónicas volátiles que almacenan información en diferentes localidades de memoria conectadas en cascada y que tienen la capacidad de mover dicha información a través de las localidades de manera sincronizada en uno u otro sentido de la conexión. Los registros de corrimiento universales, además de poder realizar los corrimientos, cuentan con la capacidad para cargar de manera paralela valores arbitrarios a todas las localidades de memoria en un solo ciclo de reloj, así como la de mantener el contenido de la memoria intacto. Los registros de corrimiento son uno de los bloques básicos para la construcción de todo tipo de sistemas digitales entre los que se encuentran los siguientes: convertidores serie paralelo y paralelo serie, contadores, generadores de secuencias, sumadores y restadores binarios y generadores pseudoaleatorios binarios [43].

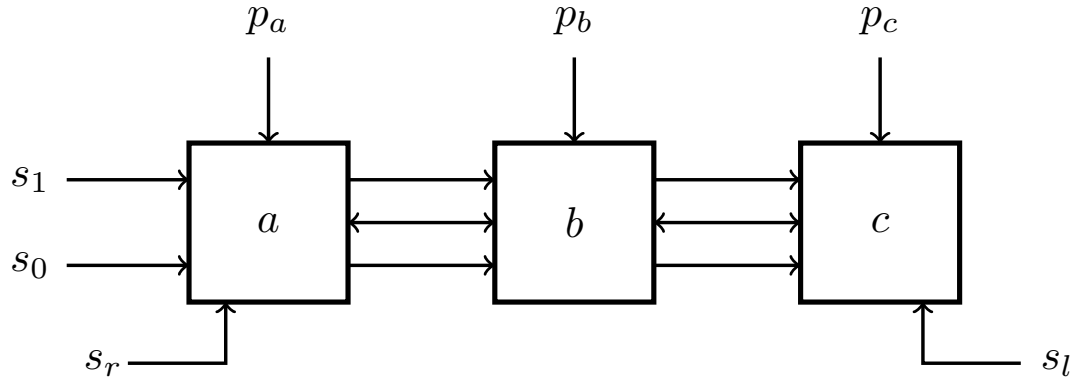

Figura 4.2: Registro universal de corrimiento de 3 bits.

El registro de corrimiento que se estudiará aquí se muestra en la Figura 4.2. Este registro tiene tres localidades de memoria a, b y c, las cuales almacenan un bit cada una. Los valores de las líneas de control s_0 y s_1 determinan la operación que realizará el registro en el siguiente

s_1	s_0	Operación
0	0	Sin cambio
0	1	Corrimiento derecho
1	0	Corrimiento izquierdo
1	1	Carga paralela

Tabla 4.1: Operaciones que realiza el registro de corrimiento en función de las líneas s_1 y s_0.

ciclo de reloj, la operación asociada a cada combinación de los valores de estas líneas se muestra en la Tabla 4.1. La línea s_r alimenta el valor de la localidad de memoria a cuando se realiza un corrimiento hacia la derecha y la línea s_l alimenta a la localidad c cuando se hace un corrimiento a la izquierda. Finalmente, cuando se selecciona la carga paralela, las localidades a, b y c tomarán los valores de las líneas p_a, p_b y p_c, respectivamente.

4.2.2. Codificación

En este ejemplo se utilizará por primera vez en el verificador a las variables de entrada. Como se explicó en la Subsección 3.2.1, las variables de entrada son variables booleanas cuyo valor no determina el usuario del dispositivo sino que lo adquieren de manera no determinística. Para este ejemplo, es una suposición razonable que las líneas de corrimiento s_r y s_l y las líneas de carga paralela p_a, p_b y p_c se consideren como variables de entrada, esto quiere decir que el verificador no tiene la capacidad de determinar su valor durante el funcionamiento del dispositivo, por lo que se tiene que considerar tanto el caso en el que la línea toma el valor de 0, como el caso en el que toma el valor de 1.

La clave para poder realizar la codificación del modelo en el lenguaje descrito es recordar que el valor de las variables del registro de corrimiento en el siguiente ciclo de reloj estará determinado por el valor actual de las líneas s_0 y s_1, esta caracterización se puede realizar utilizando implicaciones lógicas. Además se considerará que el estado inicial del sistema es aquel con las localidades de memoria a, b y c en cero. El Código 4.4 muestra la codificación del registro para el verificador.

4.2.3. Propiedades a verificar

Como ya se ha visto con el ejemplo anterior, la especificación depende del diseñador y del poder expresivo de la lógica CTL. El proceso de diseño, por lo tanto, involucra el pensar qué comportamientos debería tener el sistema y qué situaciones de error se deberían de evitar. Algunos ejemplos interesantes para el registro de corrimiento pueden ser:

1. Cuando las lineas s_0 y s_1 se encuentran ambas apagadas (valor se mantiene), la memoria tendrá exactamente el mismo valor en el siguiente ciclo que en el actual, independientemente de las líneas de entrada (seguridad).

2. Sin importar en qué estado nos encontremos, siempre podremos llegar a un estado dos unidades de tiempo más adelante con la memoria vacía (considérese el caso en el que en

```
1  VAR
2     a;
3     b;
4     c;
5     s0;
6     s1;
7  IVAR
8     sr;
9     sl;
10    pa;
11    pb;
12    pc;
13 INIT
14    !a & !b & !c;
15 TRANS
16    ((!s0 & !s1) -> ((next(a) <-> a ) & (next(b) <-> b ) & (next(c) <-> c ))) & -- igual
17    ((!s1 &  s0) -> ((next(a) <-> sr) & (next(b) <-> a ) & (next(c) <-> b ))) & -- sr
18    (( s1 & !s0) -> ((next(a) <-> b ) & (next(b) <-> c ) & (next(c) <-> sl))) & -- sl
19    (( s1 &  s0) -> ((next(a) <-> pa) & (next(b) <-> pb) & (next(c) <-> pc)));  -- carga
```

Código 4.4: Especificación del registro universal de 3 bits.

una unidad de tiempo se prenden ambas líneas s_0 y s_1 y en la siguiente se carga con 0 a todas las localidades) (seguridad).

3. Si en un instante de tiempo está seleccionado el corrimiento a la derecha, siempre habrá un estado el en instante de tiempo siguiente con el valor 0 en la localidad de memoria a y otro con el valor de 1, pero eso no pasa con la localidad b ni con la c (seguridad).

La traducción a lógica CTL de estas especificaciones se muestra en el Código 4.5.

```
20 CTLSPEC
21    AG((!s1 & !s0) -> ((a <-> AX(a)) & (b <-> AX(b)) & (c <-> AX(c))))
22 CTLSPEC
23    AG(EX (EX (!a & !b & !c)));
24 CTLSPEC
25    AG((!s1 & s0) -> EX(a) & EX(!a) & !(EX(b) & EX(!b)) & !(EX(c) & EX(!c)));
```

Código 4.5: Especificación del registro universal de 3 bits.

4.2.4. Resultado

Se puede ver cómo efectivamente el modelo diseñado cumple con la especificación:

```
1 Especificacion #1: satisface
2 Especificacion #2: satisface
3 Especificacion #3: satisface
4   Shift3
```

Código 4.6: Salida de la prueba para el registro universal de 3 bits.

4.3. Exclusión Mutua

Un campo de conocimiento importante en la computación concierne al estudio de sistemas en los que existen dos o más componentes que pueden acceder a un mismo recurso compartido de manera independiente. Un caso particular de este problema es el de la *exclusión mutua* en el que, por la naturaleza del recurso compartido, los componentes no pueden tener acceso a este de manera simultánea (al acceso al componente compartido en este tipo de sistemas se le suele llamar acceso a una *sección crítica*). En problemas de esta naturaleza, es necesario coordinar el acceso a la sección crítica de los componentes para evitar condiciones de punto muerto o condiciones donde propiedades de vitalidad no se cumplen, ya que un componente acapara por completo algún recurso, evitando que otros puedan tener acceso a él (condición de *inanición*).

Un ejemplo clásico que ilustra las dificultades asociadas con la exclusión mutua es el problema de los filósofos comedores. Propuesto inicialmente por Edsger W. Dijkstra (1930-2002) y posteriormente formalizado por Tony Hoare (1934) en 1978. La formulación textual del problema es la siguiente:

> Cinco filósofos pasan su vida pensando y comiendo. Los filósofos comparten un comedor común donde se encuentra una mesa redonda rodeada de cinco sillas, cada una de las cuales pertenece a uno de los filósofos. En el centro de la mesa se encuentra un gran tazón con espagueti, además, la mesa está puesta con cinco tenedores (véase la Figura 4.3 tomada del artículo original de Hoare). En cuanto le da hambre, un filósofo entra al comedor, se sienta en su silla y toma el tenedor a la izquierda de su lugar. Desafortunadamente, el espagueti está tan enredado que también necesita utilizar el tenedor de la derecha para poder comérselo. Una vez que ha terminado, pone ambos tenedores sobre la mesa y sale del cuarto [44].

El objetivo del ejercicio es el de coordinar correctamente cómo es que los filósofos toman los tenedores para que todos puedan comer y no ocurran problemas de sincronización entre ellos.

4.3.1. Descripción

Una solución relativamente sencilla al problema, y la que modelaremos aquí es la de usar un *árbitro*. En este caso, el árbitro es una entidad central que coordina cuándo un filósofo puede tomar los tenedores, de manera que se asegure que no hay problemas de coordinación.

Por razones de simplicidad, se modelará el problema únicamente para dos filósofos y dos tenedores en la mesa, como se muestra en la Figura 4.4. Las variables presentes en la misma denotan diferentes situaciones de interés:

- Las variables s_1 y s_2 representarán el deseo del filósofo 1 y del filósofo 2 de comer, respectivamente. Estas variables se modelarán como variables no determinísticas de entrada, ya que no se sabe *a priori* cuándo un filósofo solicitará comer o dejar de hacerlo.
- Las variables i_1 y d_1 representarán que el filósofo 1 está ocupando el tenedor izquierdo y derecho, respectivamente. Y la misma situación con las variables i_2 y d_2 para el filósofo 2.

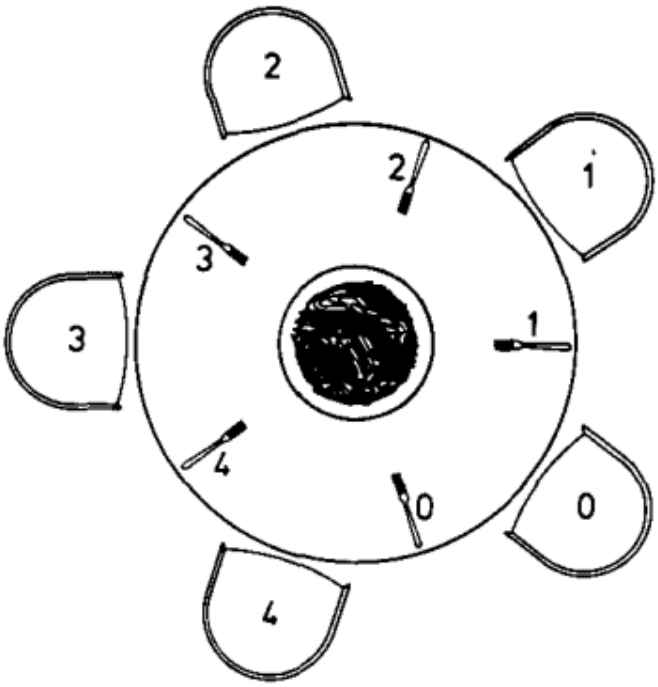

Figura 4.3: Diagrama del problema de los filósofos comedores.

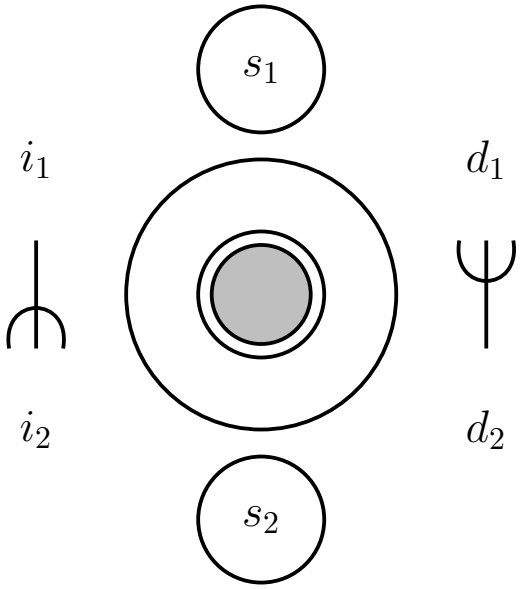

Figura 4.4: Modelado booleano del problema de los filósofos comedores.

Por su parte, el sistema tendrá el siguiente comportamiento:

1. El filósofo 1 y el filósofo 2 podrán solicitar al árbitro los tenedores en cualquier momento.
2. En cuanto a un filósofo se le concedan los dos tenedores, este deberá empezar a comer.
3. El árbitro concederá al filósofo ambos tenedores en el siguiente instante de tiempo únicamente cuando el otro filósofo no los esté utilizando ni solicitando en el instante de tiempo actual. Si se le conceden los tenedores a un filósofo, este debe tomar los dos y no únicamente uno.
4. Si los dos filósofos solicitan los tenedores en un instante de tiempo y estos se encuentran libres, el árbitro se los concederá a uno de ellos y al otro no, de manera no determinística.
5. Si en un instante determinado un filósofo está ocupando los tenedores y continua solicitándolos para el siguiente instante, se le permitirá mantenerlos en dicho instante siguiente.
6. Si un filósofo se encuentra comiendo y deja de solicitar seguir haciéndolo, se liberarán los tenedores en el siguiente instante de tiempo.

4.3.2. Codificación

La formulación matemática de la descripción hecha en la lista anterior es considerablemente más elaborada que con los ejemplos anteriores. Por esta razón, antes de mostrar el código se irán traduciendo las diferentes características del comportamiento del sistema a sus equivalentes lógicos:

- Como ambos tenedores se tomarán y regresarán al mismo tiempo de la mesa, crearemos variables booleanas compuestas `"oca := da & ia"` y `"noca := !da & !ia"` donde `a` es `1` ó `2`, que representarán las situaciones en las que el filósofo `a` está haciendo uso de ambos tenedores y cuando no los está usando, respectivamente.
- La consideración hecha en el punto 2 de la lista anterior se modelará implícitamente, suponiendo que en cuanto se cumpla `oca`, el filósofo `a` está comiendo.
- La restricción 3 se modelará con la implicación `"sa & nocb & !sb -> next(oca)"`, donde si al filósofo `1` se le conceden los tenedores `a` será `1` y `b` será `2` y viceversa si se le conceden al filósofo `2`.
- La condición 4 se traducirá de manera directa a la implicación:

```
"s1 & s2 & noc1 & noc2 ->
(next(oc1) & next(noc2) xor next(noc1) & next(oc2))"
```

- La característica 5 se modelará con la implicación `"sa & oca -> next(oca)"` con `a` siendo `1` ó `2` dependiendo del filósofo del que se trate.
- El comportamiento 6 se traduce a `"!sa -> next(noca)"` donde `a` toma el valor del filósofo correspondiente.

- Finalmente, para cubrir algunas situaciones no contempladas aún en las que un filósofo está ocupando los tenedores al mismo tiempo que el otro los solicita, crearemos dos condiciones que fuerzan a que un filósofo únicamente pueda tomar los cubiertos una vez que el otro los haya puesto sobre la mesa. La codificación de esta propiedad se hace a través de la fórmula `"(oca -> next(nocb))"`, donde si el filósofo `1` está ocupando los tenedores entonces `a` será `1` y `b` será `2`, y viceversa si el filósofo `2` los está ocupando.

Una vez hecho esta explicación, la definición de las variables y del conjunto de estados iniciales es:

```
1  VAR
2      i1;
3      i2;
4      d1;
5      d2;
6  IVAR
7      s1;
8      s2;
9  DEFINE
10     oc1 := i1 & d1;
11     oc2 := i2 & d2;
12     noc1 := !i1 & !d1;
13     noc2 := !i2 & !d2;
14 INIT
15     noc1 & noc2;
```

Código 4.7: Especificación de las variables y estados iniciales del problema de los filósofos comedores.

La codificación de la función de transición es entonces:

```
16 TRANS
17     (s1 & noc2 & !s2 -> next(oc1)) & -- pt. 3
18     (s2 & noc1 & !s1 -> next(oc2)) & -- pt. 3
19     -- pt. 4
20     (s1 & s2 & noc1 & noc2 -> (next(oc1) & next(noc2) xor next(noc1) & next(oc2))) &
21     (s1 & oc1 -> next(oc1)) & -- pt. 5
22     (s2 & oc2 -> next(oc2)) & -- pt. 5
23     (!s1 -> next(noc1)) &     -- pt. 6
24     (!s2 -> next(noc2)) &     -- pt. 6
25     (oc2 -> next(noc1)) &     -- extra
26     (oc1 -> next(noc2));      -- extra
```

Código 4.8: Función de transición en el problema de los filósofos comedores.

4.3.3. Propiedades a verificar

Las cuatro propiedades que se verificarán son:

- No existe un estado donde se cumplan de manera simultánea i_1 e i_2 o d_1 y d_2. Esto es, no existe la situación en la que ambos filósofos están tomando el mismo tenedor (seguridad).

- Para todo estado y para cada uno de los filósofos; o dicho filósofo está utilizando ambos tenedores, o no está utilizando ninguno de ellos (seguridad).
- Existe un estado futuro en el que el filósofo 1 está comiendo y otro en el que el filósofo 2 lo está haciendo (vitalidad).
- Existe un estado futuro en el que el filósofo 1 no está comiendo y otro en el que el filósofo 2 tampoco (vitalidad).

La entrada con cada una de las especificaciones descritas se muestra en el Código 4.9.

```
27 CTLSPEC
28     !(EF (i1 & i2)) & !(EF (d1 & d2));
29 CTLSPEC
30     AG((oc1) | (noc1)) & AG((oc2) | (noc2));
31 CTLSPEC
32     EF (oc1) & EF(oc2);
33 CTLSPEC
34     EF (noc1) & EF(noc2);
```

Código 4.9: Propiedades a verificar en el problema de los filósofos comedores.

4.3.4. Resultado

Si el modelo está bien diseñado, debería cumplir con las 4 especificaciones, lo cual efectivamente hace, como muestra el Código 4.10.

```
1 Especificacion #1: satisface
2 Especificacion #2: satisface
3 Especificacion #3: satisface
4 Especificacion #4: satisface
5   Dining2
```

Código 4.10: Propiedades a verificar en el problema de los filósofos comedores.

4.4. Equidad

A pesar de que la especificación hecha ya cumple con algunas de las características deseadas para el sistema, aún existen situaciones "injustas" que pueden presentarse en él. Una situación evidentemente injusta en el sistema creado se puede dar si, una vez que un filósofo cuenta con ambos tenederos, este sigue solicitándolos de manera indefinida, acaparando los tenedores y evitando que el otro filósofo pueda comer. El punto 5 de la lista de características descritas para el sistema concede al filósofo los tenedores de nuevo en caso de que se dé esta situación. Como las variables de solicitud s_1 y s_2 son variables de entrada, existe un camino en el que esto sucede de manera indefinida violando la propiedad de vitalidad en la que a un filósofo que solicita comer se le permite eventualmente hacerlo. Formalmente, las dos especificaciones mostradas en el Código 4.11 denotan los casos en los que el filósofo 1 y el filósofo 2 acaparan ambos tenedores a partir del segundo instante de tiempo (el operador EX se tiene que añadir ya que en todos los estados iniciales los tenedores están sobre la mesa).

```
35 CTLSPEC
36     EX(EG(oc1));
37 CTLSPEC
38     EX(EG(oc2));
```

Código 4.11: Especificación de situaciones injustas en el problema de los filósofos comedores.

Y el sistema satisface ambas situaciones injustas, como se muestra en el Código 4.12.

```
1 Dining2
2     expected: [True,True,True,True,False,False]
3     but got: [True,True,True,True,True,True]
```

Código 4.12: De las seis especificaciones hechas hasta ahora, el sistema cumple con las últimas dos propiedades injustas.

Para evitar que el sistema cumpla con dichas características injustas podemos agregar propiedades de equidad que fuercen a que eventualmente se cumplan las fórmulas booleanas `"oc1"` como `"oc2"` en todos los caminos. Los elementos sintácticos necesarios para especificar dichas propiedades ya han sido definidos en la subsección 3.2.1. El Código 4.13 muestra la entrada el sistema de estas dos nuevas propiedades.

```
39 FAIRNESS
40     oc1;
41 FAIRNESS
42     oc2;
```

Código 4.13: Propiedades de equidad añadidas al problema.

Con estas propiedades adicionales, el sistema ya no satisface a la condiciónes injustas con las que cumplía antes, como muestra el Código 4.14.

```
1 Especificacion #5: no satisface
2 Especificacion #6: no satisface
3   Dining2
```

Código 4.14: Las dos situaciones injustas se han resulto agregando propiedades de equidad al sistema.

El sistema construido, junto con los ejemplos descritos y el código para probarlos se encuentra disponible en el siguiente repositorio de Github:
`https://github.com/javierdiegof/HaskellSMV` [1].

[1] La instalación y la ejecución es casi instantánea si se cuenta con Stack. El repositorio está ligado a la herramienta de integración continua Travis CI para asegurar que el paquete funciona en un sistema arbitrario sin las configuraciones particulares de la máquina en la que se creó.

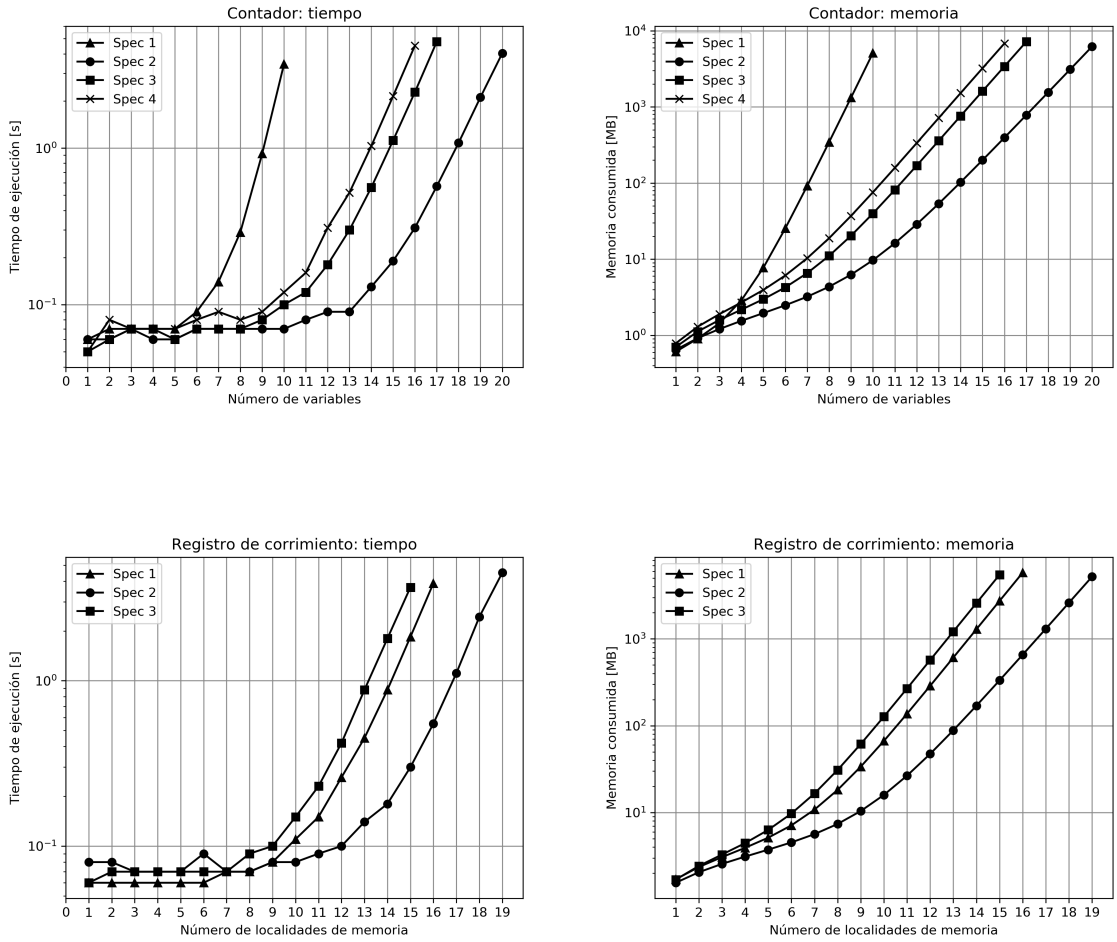

Figura 4.5: Consumo de tiempo y memoria para el contador (parte superior) y el registro de corrimiento (parte inferior) en función del tamaño del modelo. Nótese que el eje de las ordenadas tiene una escala logarítmica.

4.5. Desempeño en tiempo y memoria

4.5.1. Características generales

En la verificación de modelos, tanto la naturaleza como el tamaño del sistema estudiado, así como las fórmulas a verificar en el mismo influyen en la cantidad de memoria utilizada por la computadora y en el tiempo que le toma en realizar la verificación. La Figura 4.5 muestra el desempeño en tiempo y memoria para el contador y el registro de corrimiento universal

en función del tamaño del modelo[2,3] (con el tamaño de fórmula creciendo proporcionalmente para cada modelo). De esta figura se puede ver que, a pesar de que para todas las especificaciones se observa un crecimiento exponencial tanto en el tiempo que toma realizar la verificación como en la cantidad de memoria consumida, la tasa de crecimiento depende en gran medida de la fórmula CTL a verificar. Como se esperaría de los pseudocódigos descritos para obtener los conjuntos de satisfacción simbólicos, aquellas fórmulas con un mayor número de cuantificadores toman la mayor cantidad de tiempo en verificarse para los ejemplos mostrados.

4.5.2. Componentes principales

```
1  	Tue Jun 25 18:39 2019 Time and Allocation Profiling Report  (Final)
2
3  	   test-exe2 +RTS -N -p -RTS 15
4
5  	total time  =        3.81 secs   (3811 ticks @ 1000 us, 1 processor)
6  	total alloc = 5,475,663,936 bytes  (excludes profiling overheads)
7
8  COST CENTRE MODULE         SRC                                     time alloc
9
10 withBDD     Data.HasCacBDD hs/Data/HasCacBDD.hs:(93,1)-(94,65)      25.3    8.7
11 allVarsOf   Data.HasCacBDD hs/Data/HasCacBDD.hs:(273,1)-(276,101)   24.8   25.6
12 fromTwoBDDs Data.HasCacBDD hs/Data/HasCacBDD.hs:(109,1)-(111,38)    19.3   27.3
13 finalize    Data.HasCacBDD hs/Data/HasCacBDD.hs:46:1-70             13.4   32.6
14 fromBDD     Data.HasCacBDD hs/Data/HasCacBDD.hs:(104,1)-(105,31)     3.5    2.8
15 same        Data.HasCacBDD hs/Data/HasCacBDD.hs:179:1-37             3.4    0.0
16 firstVarOf  Data.HasCacBDD hs/Data/HasCacBDD.hs:(256,1)-(259,70)     2.7    2.8
17 ==          Data.HasCacBDD hs/Data/HasCacBDD.hs:176:3-23             2.7    0.0
18 manager     Data.HasCacBDD hs/Data/HasCacBDD.hs:82:1-83              1.7    0.0
19 withTwoBDDs Data.HasCacBDD hs/Data/HasCacBDD.hs:(98,1)-(100,68)      1.4    0.0
```

Código 4.15: Funciones que ocupan el mayor porcentaje de tiempo y memoria en la verificación para el registro de corrimiento con 15 celdas de memoria y la tercera especificación.

Si se desea analizar más detalladamente el tiempo y la memoria consumida en cada instancia de la verificación de modelos se puede estudiar qué porcentaje del total está siendo consumido por cada función que se ejecuta en el código. Esta información puede ayudar a entender qué componentes están acaparando la mayor cantidad de recursos, y a estudiar si se puede hacer algo para mejorar el desempeño en estos.

Afortunadamente, el compilador GHC de Haskell cuenta con una herramienta para obtener precisamente estas métricas: se trata del módulo *RTS* (*Runtime System*, sistema de tiempo de ejecución). Dentro de este módulo existen opciones para entender, al nivel del sistema

[2]El sistema en el que se realizaron las verificaciones es una computadora Linux de 64 bits, 8 GB de memoria RAM y un procesador Intel® Core™i5-7300HQ@2.50GHz×4.

[3]Estas gráficas fueron creadas con un código en Python que automáticamente generaba el programa a verificar para el número de variables establecido, lo ejecutaba y posteriormente obtenía los valores de tiempo y memoria, finalmente utilizaba estos para crear dichas gráficas con la biblioteca matplotlib.

operativo, cómo es que el programa está utilizando los recursos a lo largo de su ejecución, desglosando qué componentes están ocupando la mayor cantidad de tiempo y memoria.

De manera consistente para todos los ejemplos estudiados, e independiente del número de variables en el modelo, RTS muestra que al menos el 90 % de la cantidad de memoria consumida por el verificador y del tiempo de ejecución están siendo utilizadas por funciones relacionadas con las operaciones sobre los BDD, es decir, con operaciones que realiza el módulo HasCacBDD discutido en la Subsección 3.1.4. El Código 4.15 muestra un ejemplo de la salida de la medición para uno de los puntos en la gráfica anterior.

A pesar de que esta distribución es la esperada para un verificador de modelos simbólico ya que las operaciones sobre los BDD son el proceso más computacionalmente demandante de la verificación. Los tiempos de ejecución observados para el verificador junto con el porcentaje correspondiente a HasCacBDD hacen difícil de explicar por qué el verificador creado es considerablemente más lento que otras alternativas en el mercado como NuSMV[4]. Las dos referencias encontradas en la bibliografía que comparan el desempeño de CacBDD con otros paquetes indican un desempeño comparable en tiempo y memoria [31] [45]. Esta observación sugiere que es probable que la interfaz entre el código en Haskell y el código en *C++* tenga la responsabilidad, al menos en parte, de la diferencia de desempeño. En las conclusiones se sugieren posibles acciones para lidiar con ella. En última instancia, el verificador puede verificar sistemas cuyo número de estados sobrepasa sin problemas los millones, por lo que las limitaciones no resultan especialmente restrictivas en términos prácticos.

[4]La comparación de los tiempos de ejecución entre el verificador creado y un sistema de estado del arte (como NuSMV) no se agrega en el trabajo. Basta con decir que el desempeño de dichos verificadores es considerablemente mayor al aquí construido.

Capítulo 5

Conclusiones y trabajo futuro

5.1. Conclusiones

Una vez construido el sistema verificador y realizadas las pruebas y mediciones correspondientes, nos encontramos en posición de contestar algunas de las preguntas que motivaron la realización de este proyecto y que se mencionaron en la introducción.

¿Qué tan adecuada es en general la verificación de modelos para se tratada funcionalmente?

En principio, cualquier algoritmo que pueda ser programado en un lenguaje de programación procedimental puede ser programado en un lenguaje funcional. Por lo tanto, esta pregunta no hace referencia a si fundamentalmente la verificación de modelos puede ser realizada de manera funcional, si no más bien a qué tan robustos y eficientes son los algoritmos con los que se cuenta en este paradigma para poder realizar todo el proceso de verificación, desde la lectura del código fuente, hasta la emisión del criterio de satisfacción final. Es importante recordar que la verificación de modelos no solamente involucra la implementación de los algoritmos de verificación sobre la estructura de datos de elección sino que existe una cantidad considerable de elementos adicionales que se deben de construir: la lectura del código fuente, el despliegue de información al usuario, la recabación de información acerca de la ejecución, la prueba y distribución del código, entre otras tareas. La realización de este proyecto de tesis muestra que la gran mayoría de los elementos que involucra la verificación de modelos pueden ser implementados de manera adecuada utilizando el paradigma funcional. Desde su invención a finales de los años 50, la programación funcional ha sido ampliamente desarrollada y estudiada desde el punto de vista práctico y teórico. Se han inventado técnicas para lidiar con problemas (como el análisis sintáctico e incluso el manejo de dispositivos de entrada-salida) para los cuales no existían anteriormente métodos robustos de solución. Se puede decir entonces que ya existen todas las herramientas necesarias para hacer la verificación de modelos simbólica sin mayores impedimentos bajo este paradigma.

Específicamente, ¿qué partes del proceso de verificación se ven afectadas positiva o negativamente por el uso de lenguajes funcionales en general, y Haskell en particular?

Esta pregunta puede ser analizada desde distintos puntos de vista y la solución a la misma dependerá del enfoque que se le dé. Es verdad que, si se analiza únicamente desde el punto de vista del desempeño en tiempo y memoria, es probable que no haya implementación funcional que pueda competir con los paquetes programados en lenguajes como *C* ó *C++*. Estos son lenguajes de bajo nivel comparados con cualquier lenguaje funcional y tienen la capacidad de manipular directamente algunos elementos de la computadora, sin tener que pasar por capas de traducción como el caso de los lenguajes de alto nivel, entre los que se encuentra Haskell. Esta relativa desventaja tiene beneficios para el diseñador y programador del verificador. Al ser un lenguaje de alto nivel, Haskell tiene la capacidad de representar computaciones complejas y elaboradas con relativa simplicidad y con una elegancia matemática de la que carecen la mayoría de los lenguajes procedimentales. Por ejemplo, en etapas como el análisis sintáctico o la realización de casos de prueba, el paradigma funcional permite la programación de los algoritmos de manera sumamente simple y robusta. Este es el caso no solamente para estos dos elementos sino también para muchas otras etapas del proceso: la definición de los tipos de datos en el verificador, la integración de los diferentes paquetes en una solución, la definición de las operaciones asociadas a los BDD, la creación de la interfaz que llama a la paquetería de los BDD, entre otras. Quizás la omisión más importante para el paradigma funcional en este proyecto sea que los BDD no fueron programados en Haskell (las razones por las cuales se tomó esta decisión se mencionan en la Subsección 3.1.2). A pesar de las fuentes que se han mencionado que investigan cómo implementar estos componentes funcionalmente, se considera que las limitaciones asociadas a dichas implementaciones son considerables y aún es necesario investigar si es posible aprovechar el paradigma funcional y otras propiedades de Haskell (como la pereza) para crear una paquetería funcional robusta y eficiente. En el resto de los elementos del verificador, el programa construido nos sugiere que nos hay limitaciones importantes de funcionalidad o desempeño para hacer un verificador funcional tan completo como se desee.

Como herramienta para el desarrollo de software, ¿cuán conveniente resulta Haskell para construir un verificador de modelos?

Finalmente, queda discutir si las herramientas y paquetes con los que cuenta Haskell son suficientes para realizar un producto de software de calidad que pueda ser utilizado en la academia y en la industria. Nuestra experiencia sugiere que este sí es el caso. Ya se han mencionado en la Sección 3.1 algunos módulos y programas que sirven para trabajar las diferentes etapas del desarrollo de un producto de software, como la implementación o el mantenimiento. Existen en Haskell herramientas para crear paquetes reproducibles y distribuibles (Stack); módulos para implementar suites de pruebas (Hspec, Criterion y otros); repositorios de software libre para distribuir código (Hackage); opciones para implementar integración continua (Stack posee configuraciones para utilizar Travis CI); entre otras. El compilador GHC, como ya se ha visto en la Sección 4.4, tiene la capacidad de dar información detallada acerca del desempeño en tiempo y memoria, y muchas otras herramientas se encuentran disponibles en el archivo antes mencionado. Haskell ha alcanzado una etapa de maduración en la cual no se tienen omisiones importantes para realizar productos de calidad que se puedan utilizar ampliamente.

5.2. Trabajo futuro

Los verificadores de modelos modernos son herramientas de software bastante avanzadas y con funciones que escapan lo que se puede construir de manera realista en una tesis de licenciatura. El verificador implementado en este trabajo aún puede ser ampliado tanto como se quiera e incluir funciones que son comunes en verificadores del estado del arte, entre las que se encuentran: la capacidad de lidiar con tipos de datos elaborados (enteros, arreglos, etc.), la capacidad de declarar, instanciar e incluir módulos dentro de un archivo, la capacidad de analizar la ejecución y muchas otras. De manera más urgente, existen tareas a realizar para incrementar la utilidad del verificador creado y mejorar su desempeño.

Una función útil en la verificación de modelos que no se pudo incluir en el verificador construido es la capacidad para mostrar contraejemplos en caso de que alguna propiedad no se cumpla, esta información resulta útil para el diseñador del sistema ya que sugiere una situación errónea que pudiera estar provocando que el sistema no cumpla con la especificación. Para poder mostrar estos contraejemplos es necesario contar con una representación del diagrama de estados como un grafo, como en el verificador creado los elementos sintácticos se traducen directamente a diagramas simbólicos, no es posible mostrar estos contraejemplos.

En lo que respecta al desempeño del verificador, una labor importante sería investigar con mayor detalle la razón de la discrepancia en el desempeño de las operaciones sobres los BDD entre el sistema creado y las alternativas del estado del arte. Una posible forma de aumentar el desempeño es a través del estudio de diferentes ordenamientos en las variables sobre los BDD, ya que el tamaño de un ROBDD depende considerablemente del ordenamiento utilizado en él. Para el verificador creado se utilizó un ordenamiento simple sugerido en la bibliografía [4, p. 409], sin embargo, es posible que el uso de otros ordenamientos o de un reordenamiento en tiempo de ejecución pudiera aumentar el desempeño. También, seguir probando la interfaz entre el código de Haskell y CacBDD, o probar una interfaz con alguna otra paquetería también podría ser benéfico. Idealmente, sería conveniente poder implementar las operaciones sobre los BDD de manera puramente funcional y sin recurrir a un paquete externo escrito en otro lenguaje. Este cambio, sin embargo, implica investigación a detalle sobre cómo hacer las operaciones sobre estas estructuras de datos eficientes con Haskell, de manera que su desempeño pueda ser comparable a la alternativa utilizada aquí.

El desarrollo de software es una tarea de mejora continua en la que siempre se pueden incluir funciones nuevas y mejorar componentes creados anteriormente, no existe un límite real al trabajo que se puede hacer para mejorar esta herramienta.

Apéndice A

Apéndices

En estos apéndices se definen de manera más detallada algunos conceptos técnicos que se utilizan a lo largo del presente trabajo. En su mayoría, las definiciones extraídas en estos apéndices fueron tomadas de [4, págs. 920–925] con algunas diferencias de estilo.

A.1. Grafos

Definición A.1 (Grafo, sucesores y predecesores directos)**.** Un grafo dirigido es una pareja $G = (V, E)$ formada por un conjunto V cuyos elementos se llaman *vértices* o *nodos* y una relación $E \subseteq V \times V$ cuyos elementos son llamados *aristas*. En el presente trabajo se considerará que V es finito (y por lo tanto E también lo es).

Dado un grafo $G = (V, E)$, y un nodo $v \in V$, definiremos a $Post(v)$ como el conjunto "sucesores directos" de v de la siguiente manera:

$$Post(v) = \{\ w\ \mid\ (v, w) \in E\}$$

El conjunto de "predecesores directos" $Pre(v)$ se define de la siguiente manera:

$$Pre(v) = \{\ w\ \mid\ (w, v) \in E\}$$

Definición A.2 (Nodos iniciales y terminales)**.** Sea un grafo dirigido cualquiera $G = (V, E)$ y sea un vértice cualquiera $v \in V$, si $Pre(v) = \varnothing$ diremos que v es un vértice *inicial*, si $Post(v) = \varnothing$ diremos que v es un vértice *terminal*.

Definición A.3 (*Camino* en grafos)**.** Dado un grafo $G = (V, E)$, un camino π se define como una secuencia de vértices finita, $\pi = v_0\, v_1 \ldots v_r$ con $r \geq 0$, o infinita, $\pi = v_0\, v_1 \ldots$, tal que para todo $i = 0, 1, \ldots$ se cumple que $v_{i+1} \in Post(v_i)$.

Definición A.4 (*Alcance* en grafos)**.** Dado $G = (V, E)$, el conjunto $Alc(v)$ denota el conjunto de todos los vértices a los que se puede llegar a partir de un vértice dado v, formalmente, $Alc(v)$ es el conjunto de todos los vértices $w \in V$ tales que existe un camino finito $\pi = v_0\, v_1 \ldots v_r$ en G con $v_0 = v$ y $v_r = w$.

Definición A.5 (*Ciclo* en grafos)**.** Dado $G = (V, E)$, un *Ciclo* en G es un camino finito $\pi = v_0\, v_1\, ...\, v_r$ tal que $v_0 = v_r$ y $r \geq 0$. Se dice que G es *acíclico* si no contiene ciclo alguno, en caso contrario se dice que G es *cíclico*.

Definición A.6 (*Componente fuertemente conexo*)**.** Dado $G = (V, E)$ y $C \subseteq V$, se dice que C es *fuertemente conexo* si, para todo par de vértices (v_1, v_2) tales que $v_1 \in C$ y $v_2 \in C$, v_1 puede alcanzar a v_2 y viceversa, formalmente, $v_1 \in Alc(v_2)$ y $v_2 \in Alc(v_1)$. Un *componente fuertemente conexo* C de G es un conjunto fuertemente conexo de G que es también máximo, esto es, C es un *componente fuertemente conexo* si C es fuertemente conexo y no está contenido en algún otro conjunto *fuertemente conexo* D de G, esto es, no existe D tal que $C \subseteq D$ y $C \neq D$. La complejidad algorítmica temporal de determinar los componentes fuertemente conexos de una gráfica G es $\Theta(N + M)$ con $N = |V|$ y $M = |E|$ [4, pág. 921].

Existen algoritmos para visitar sistemáticamente cada uno de los vértices de un grafo, dos comúnmente utilizados son *DFS* (*Depth-First Search*, búsqueda primero en profundidad) y *BFS* (*Breadth-First Search*, búsqueda primero en anchura) [46]. El código siguiente detalla una implementación general del algoritmo de BFS, donde a cada nodo del grafo se le asocia una "profundidad" desde uno o más vértices arbitrarios considerados como "vértices origen". Para propósitos de esta tesis, nos será de utilidad describir la distancia máxima que existe entre cada vértice y su origen, así como la distancia máxima que cada vértice tiene con el más lejano de sus orígenes, debido a que comúnmente solo se asocia un vértice origen al algoritmo, a este lo llamaremos "BFS generalizado". Para implementar este algoritmo haremos uso de una estructura de datos llamada "cola", esta estructura de datos almacena la información en forma *FIFO* (*First-In, First-Out*, primero en entrar, primero en salir). Se supone familiaridad básica con este concepto. En las estructuras de datos que se utilicen (conjuntos, colas, diccionarios, etc.), se hará uso de notación de Python 3 para métodos y funciones. Para que el algoritmo funcione correctamente, debemos de ser cuidadosos de que todos los vértices en la lista O sean vértices terminales, de no ser así, puede ocurrir un comportamiento inesperado donde se asignen profundidades incorrectas.

A.2. Árboles

Definición A.7 (*Árbol*)**.** Una gráfica dirigida $T = (V, E)$ es un árbol si existe un vértice $v_0 \in V$ que cumple con $Pre(v_0) = \varnothing$ y además, para todo vértice $v \in V$ v_0 puede alcanzar a v únicamente por un solo camino (en los árboles, los caminos también suelen ser llamados "ramas"). El vértice v_0 debe de ser único y recibe el nombre de *raíz* de T. A un vértice v sin sucesores directos ($Post(v) = \varnothing$) se le llama *hoja* de T. Cualquier vértice v distinto del vértice *raíz* tiene únicamente un predecesor ($|Pre(v)| = 1$), donde a este predecesor único w de v se le llama el *padre* de v y se dice que v es *hijo* de w.

Definición A.8 (*Subárbol*)**.** Un subárbol de un árbol $T = (V, E)$ es un árbol que consiste en tomar a cualquier nodo $v \in V$ como el nodo raíz (esto es, descartar a todos sus predecesores) y formar un árbol con todos los sucesores de v, de esta manera, existe un subárbol por cada nodo en V. Cuando el nodo v es el nodo raíz de T, el subárbol que se forma es precisamente T.

Algoritmo 1.11 Búsqueda primero en anchura generalizada (BFS)

```
1: def bfsGen(G = (V, E), O = [o_1, ... , o_n]):
2:         # ∀o_i (o_i ∈ V)
3:     profundidad = {}       # Diccionario de Python
4:     for o in O :
5:         cola = []
6:         visitados = ∅()
7:         cola.append(o)
8:         profundidad[o] = 0
9:         while cola ≠ [] :      # Mientras no este vacia
10:            act = cola.remove()
11:            for h in Post(act) :
12:                if h not in visitados :
13:                    visitados = visitados ∪ {h}
14:                    cola.append(h)
15:                if h not in profundidad :
16:                    profundidad[h] = profundidad[act] + 1
17:                else:
18:                    profundidad[h] = max(profundidad[h], profundidad[act]+1)
19:    return profundidad
```

Definición A.9 (*Árbol binario*)**.** Se dice que un árbol $T = (V, E)$ es binario si además cumple con la propiedad de que, para todo vértice $v \in V$ se cumple que $|Post(v) \leq 2|$, es decir, que cada nodo del árbol tiene como máximo dos hijos. Si, para todos los nodos padre se cumple que $|Post(v) = 2|$, entonces se dirá que es un *árbol binario completo*.

Bibliografía

[1] E. M. Clarke, O. Grumberg y D. Peled, *Model Checking (Cyber-Physical Systems Series)*. Cambridge, MA, USA: MIT Press, 1999, pág. xiii.

[2] B. Boehm y V. R. Basili, "Software Defect Reduction Top 10 List", *Foundations of Empirical Software Engineering: The Legacy of Victor R. Basili*, vol. 426, n.º 37, pág. 430, 2005.

[3] J. Rushby, "Theorem Proving for Verification", en *Summer School on Modeling and Verification of Parallel Processes*, Springer, 2000, págs. 39-57.

[4] C. Baier y J.-P. Katoen, *Principles of Model Checking*. Cambridge, MA, USA: MIT Press, 2008.

[5] M. Huth y M. Ryan, *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge, U.K.: Cambridge Univ. Press, 2004.

[6] K. L. McMillan, *Symbolic Model Checking*. Dordrecht, The Netherlands: Kluwer Academic Publishers, 1993.

[7] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill y L.-J. Hwang, "Symbolic model checking: 1020 States and beyond", *Information and computation*, vol. 98, n.º 2, págs. 142-170, 1992.

[8] P. Hudak, J. Hughes, S. Peyton Jones y P. Wadler, "A history of Haskell: being lazy with class", en *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, ACM, 2007, págs. 12.1-12.55.

[9] P. Wadler, "Monads for functional programming", en *International School on Advanced Functional Programming*, Springer, 1995, págs. 24-52.

[10] I. Jones, "The Haskell Cabal: a Common Architecture for Building Applications and Libraries", en *6th Symposium on Trends in Functional Programming*, Intellect Books, 2005, págs. 340-354.

[11] P. Øhrstrøm y P. Hasle, *Temporal Logic: From Ancient Ideas to Artificial Intelligence*, ép. Studies in Linguistics and Philosophy. Heildeberg, Germany: Springer, 2007, vol. 57.

[12] Epictetus y R. Dobbin, *Discourses and Selected Writings*. Baltimore, MD, USA: Penguin, 2008, pág. 122.

[13] O. Grumberg y H. Veith, *25 Years of Model Checking: History, Achievements, Perspectives*, ép. LNCS sublibrary: Theoretical computer science and general issues. Heildeberg, Germany: Springer, 2008.

[14] A. Pnueli, "The Temporal Semantics of Concurrent Programs", *Theoretical computer science*, vol. 13, n.º 1, págs. 45-60, 1981.

[15] L. Lamport, ""Sometime" is Sometimes "Not Never": On the Temporal Logic of Programs", en *Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ACM, 1980, págs. 174-185.

[16] E. Allen Emerson y J. Halpern, ""Sometimes" and "Not Never" Revisited: On Branching versus Linear Time Temporal Logic", *Journal of the ACM*, vol. 33, págs. 151-178, 1986.

[17] E. M. Clarke y E. A. Emerson, "Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic", en *Workshop on Logic of Programs*, Springer, 1981, págs. 52-71.

[18] J.-P. Queille y J. Sifakis, "Specification and verification of concurrent systems in CESAR", en *International Symposium on programming*, Springer, 1982, págs. 337-351.

[19] A. Martin, "Adequate Sets of Temporal Connectives in CTL", *Electr. Notes Theor. Comput. Sci.*, vol. 52, n.º 1, págs. 21-31, 2001.

[20] R. Cavada, A. Cimatti, C. A. Jochim, G. Keighren, E. Olivetti, M. Pistore, M. Roveri y A. Tchaltsev, "NuSMV 2.4 User Manual", *CMU and ITC-irst*, 2005.

[21] Sakharov, Alex, *Strict Order*, 2019. dirección: `http://mathworld.wolfram.com/StrictOrder.html` (visitado 5 de mayo de 2019).

[22] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation", *IEEE Transactions on Computers*, vol. C-35, n.º 8, págs. 677-691, 1986.

[23] A. Arora y S. Bansal, *Unix and C Programming*. New Delhi, India: Laxmi Publications Pvt Limited, 2005, págs. 461-462.

[24] A. Church, "An Unsolvable Problem of Elementary Number Theory", *American journal of mathematics*, vol. 58, n.º 2, págs. 345-363, 1936.

[25] A. M. Turing, "On Computable Numbers, with an Application to the Entscheidungsproblem", *Proceedings of the London mathematical society*, vol. 2, n.º 1, págs. 230-265, 1937.

[26] HaskellWiki contributors, *Lazy evaluation*, 2015. dirección: `https://wiki.haskell.org/index.php?title=Lazy_evaluation&oldid=60051` (visitado 1 de jun. de 2019).

[27] ——, *Haskell in education*, 2019. dirección: `https://wiki.haskell.org/Haskell_in_education` (visitado 1 de jun. de 2019).

[28] ——, *Haskell in industry*, 2019. dirección: `https://wiki.haskell.org/Haskell_in_industry` (visitado 1 de jun. de 2019).

[29] J. Christiansen. (2006). A purely functional implementation of ROBDDs in Haskell, dirección: `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.90.2879&rep=rep1&type=pdf` (visitado 10 de abr. de 2019).

[30] J. Christiansen y F. Huch, "A purely functional implementation of ROBDDs in Haskell.", *Trends in Functional Programming*, vol. 7, págs. 55-71, 2006.

[31] G. Lv, K. Su e Y. Xu, "CacBDD: A BDD Package with Dynamic Cache Management", en *International Conference on Computer Aided Verification*, Springer, 2013, págs. 229-234.

[32] HaskellWiki contributors, *Foreign Function Interface — HaskellWiki,* 2019. dirección: `https://wiki.haskell.org/index.php?title=Foreign_Function_Interface&oldid=62872` (visitado 15 de abr. de 2019).

[33] K. S. Brace, R. L. Rudell y R. E. Bryant, "Efficient implementation of a BDD package", en *27th ACM/IEEE design automation conference*, IEEE, 1990, págs. 40-45.

[34] J. Van Benthem, J. Van Eijck, M. Gattinger y K. Su, "Symbolic Model Checking for Dynamic Epistemic Logic", en *International Workshop on Logic, Rationality and Interaction*, Springer, 2015, págs. 366-378.

[35] P. Gammie. (2015). hBDD-CMUBDD: An FFI binding to CMU/Long's BDD library, dirección: `https://hackage.haskell.org/package/hBDD-CUDD` (visitado 15 de abr. de 2019).

[36] ——, (2015). hBDD-CMUBDD: An FFI binding to CMU/Long's BDD library, dirección: `hBDD-CMUBDD:%20An%20FFI%20binding%20to%20CMU/Long's%20BDD%20library` (visitado 15 de abr. de 2019).

[37] G. Hutton y E. Meijer, "Monadic Parsing in Haskell", *Journal of functional programming*, vol. 8, n.º 4, págs. 437-444, 1998.

[38] G. Hutton, *Programming in Haskell*, 2.ª ed. Cambridge, U.K.: Cambridge Univ. Press, 2016, págs. 153-176.

[39] D. Leijen, P. Martini y A. Latter. (2018). Parsec: Monadic parser combinators, dirección: `http://hackage.haskell.org/package/parsec` (visitado 16 de abr. de 2019).

[40] S. Hengel. (2019). Hspec: A Testing Framework for Haskell, dirección: `http://hackage.haskell.org/package/hspec` (visitado 16 de abr. de 2019).

[41] Stack contributors. (2019). The Haskell Tool Stack, dirección: `https://docs.haskellstack.org/en/stable/README/` (visitado 15 de abr. de 2019).

[42] S. Thomasson, *Haskell High Performance Programming*, ép. Community experience distilled. Birmingham, U.K.: Packt Publishing, 2016, pág. 99.

[43] A. Godse y D. Godse, *Digital Logic Design & applications*. Pune, India: Technical Publications, 2008, págs. 7.10-7.12.

[44] C. A. R. Hoare, "Communicating Sequential Processes", en *The origin of concurrent programming*, Springer, 1978, págs. 413-443.

[45] T. van Dijk, E. M. Hahn, D. N. Jansen, Y. Li, T. Neele, M. Stoelinga, A. Turrini y L. Zhang, "A comparative study of BDD packages for probabilistic symbolic model checking", en *International Symposium on Dependable Software Engineering: Theories, Tools, and Applications*, Springer, 2015, págs. 35-51.

[46] T. H. Cormen, C. E. Leiserson, R. L. Rivest y C. Stein, *Introduction to Algorithms*, 3.ª ed. Cambridge, MA, USA: MIT Press, 2009, págs. 594-611.